The Hidden Grammar Problem: An Empirical Analysis of Ad Hoc Parsers in Python

Anonymous Author(s)

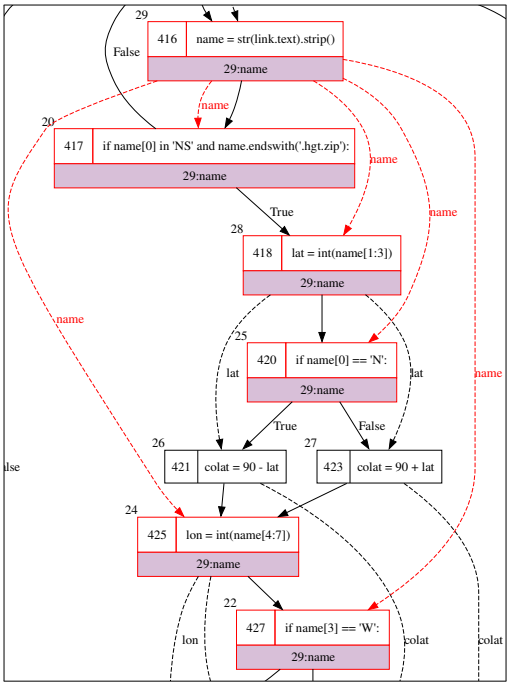
***Abstract*—Ad hoc parsers are pieces of code that use common string functions like `split`, sequence operations like slicing `s[i:j]`, or type constructors like `int()` to effectively perform parsing. Whether to handle command-line arguments, read configuration files, massage semi-structured data, or perform any other minor string processing task, ad hoc parsing is ubiquitous yet poorly understood. This study aims to reveal the common syntactic characteristics of ad hoc parsing code in real-world Python projects, based on large-scale mining of open-source repositories. Our goal is to understand the nature of ad hoc parsers in order quantify their risks, improve their testing, and strengthen their static analysis. We analyzed 189 393 Python projects and found that 97 % utilize ad hoc parsing, uncovering over 33 million ad hoc parsers. To locate these parsers, we computed full intra-procedural control flow and data dependence graphs for each Python file in the corpus, isolated string variables with a typing oracle, and constructed forward parser slices based on string data flow. We dissected these ad hoc parser slices and their surrounding contexts to extract and summarize quantitative information about features of interest. Our findings validate long-standing LangSec concerns and show that hidden grammars are an ecosystem-level challenge, with direct implications for fuzzing and grammar mining. They also show that ad hoc parsers are statically analyzable, and pave the way for novel grammar inference approaches. Our vast corpus of ad hoc parser slices provides a robust foundation to construct evaluation datasets, benchmark suites, and training data.**

***Index Terms*—ad hoc parsers, string grammars, static analysis**

I. Introduction

Software systems are full of input languages. Some are visible and deliberate: programming languages, file formats, network protocols, query languages, and configuration schemas are often described by grammars, validators, parser generators, or serialization libraries. Many others are not. Version strings, command-line arguments, resource identifiers, paths, headers, environment variables, prompts, spreadsheet cells, and log lines are routinely accepted, rejected, decomposed, and transformed by ordinary program code. Their grammars are not written down in specifications or concentrated in parser modules; they are implicit in operations such as `msg.split(":")`, `s.strip()`, `int(n)`, `email[email.find("@") + 1:]`, or `re.match(r"[0-9]+", p)`.

Such parsing operations have *hidden grammars*: the possibly infinite sets of strings they accept are defined operationally by the code that processes them. A hidden grammar may be simple or complex, intentional or accidental, stable or brittle, but it is hidden inside ordinary code. It emerges from string operations, type conversions, conditionals, regular expressions, container accesses, and exceptions that an input must survive. We call the code fragment that operationalizes such a hidden grammar an *ad hoc parser*: the observable program slice through which hidden grammars become executable behavior. The grammar itself remains latent; source code exposes the parsing work through which strings are inspected, transformed, converted, and rejected. This paper studies that observable layer. Hidden grammars matter because they shape which executions are possible. For testing and fuzzing, an input that fails a hidden local grammar may never reach the intended




```
416  name = str(link.text).strip()
417  if name[0] in 'NS' and name.
       ↪ endswith('.hgt.zip'):
418    lat = int(name[1:3])
419    # Change to co-latitude.
420    if name[0] == 'N':
421      colat = 90 - lat
422    else:
423      colat = 90 + lat
424
425    lon = int(name[4:7])
426    # Ensure positive.
427    if name[3] == 'W':
428      lon = 360 - lon
```

scitools/cartopy/lib/cartopy/io/srtm.py

Fig. 1: An ad hoc parser slice from our corpus.

behavior. For static analysis, hidden grammars determine data-flow feasibility, exception behavior, and string constraints, but are difficult to observe when parsing is interleaved with application logic. For maintenance, changes to ordinary string manipulation code can silently change a component's accepted input language. The language-theoretic security community has long warned that interleaving recognition, validation, and computation can make input handling fragile [1, 2, 3].

However, the scale and shape of this phenomenon in real software remain largely unknown. Recent grammar-mining and grammar-inference techniques aim to recover input languages from programs [4, 5, 6, 7, 8, 9]. Yet applying these techniques at ecosystem scale first requires finding the code that imposes those languages. Dynamic techniques need executable programs, suitable harnesses, and representative inputs; static techniques must know which string flows, operations, guards, and failure sites belong to parsing rather than unrelated application logic. If hidden grammars are scattered across ordinary functions, overlap with each other, or depend on common library operations and implicit exceptions, then grammar mining must extend beyond easily localizable file format entry-points and protocol front-ends. A prerequisite is an empirical account of where ad hoc parsers occur and what forms they take in real systems. The key difficulty is measurement. Generated parsers, schema validators, and regular expressions are named in source code; hidden grammars are not. To study them empirically, we need a scalable operational unit: ad hoc parser slices. Starting from a string-typed source variable, we follow intra-procedural string data flow through the statements that inspect, transform, split, combine, or convert that string. The resulting slice approximates the code whose successful execution constrains the source string's input language—its hidden grammar. This operationalization is intentionally conservative: it does not recover a complete grammar, but it makes the code behind hidden grammars visible enough to study prevalence, location, structure, and failure behavior across a large corpus.

This paper presents the first large-scale empirical study of ad hoc parser slices as the code substrate of hidden grammars. We analyze 189 393 open-source Python repositories, containing more than ten million Python files, and locate over 33 million non-trivial ad hoc parsers. Our results show that ad hoc parsers are not a rare boundary concern, as they appear in $97.05\,\%$ of projects, $47.77\,\%$ of files, and $19.95\,\%$ of functions; often begin well inside function bodies rather than at their entry points; are usually small but frequently branching or overlapping; and commonly rely on operations whose parsing failures remain exposed as ordinary exceptions. These findings identify opportunities and constraints for program analysis, fuzzing, testing, and grammar mining: the slices are often local enough to analyze, but numerous and embedded enough that tools must find them automatically.

A. Research Questions and Contributions

We contribute the conceptual and methodological basis for studying hidden grammars at scale: we connect ad hoc parsing to hidden grammars, treat parser slices as observable code artifacts through which hidden input languages are enforced, and provide a scalable graph-based approach for locating those slices in Python code, including a formalization of string-flow marking and parser-slice construction in our online appendix. We organize the empirical contribution around four questions.

- **RQ1. How prevalent is ad hoc parsing in Python?** We measure how often ad hoc parser slices occur across projects, files, and functions.
- **RQ2. Where and how are ad hoc parsers embedded within programs?** We study whether parsing is concentrated near function boundaries or scattered throughout ordinary function bodies.
- **RQ3. What is the shape and structure of ad hoc parser slices?** We characterize parser size, extent, branching, convergence, and overlap between parser slices.
- **RQ4. Which programming constructs and failure modes characterize ad hoc parsers?** We examine the operations used in ad hoc parser slices and the extent to which parsing-related exceptions are handled or exposed, deriving concrete implications for testing, fuzzing, static analysis, and grammar mining.

The resulting dataset of ad hoc parser slices and associated metadata provides raw material for benchmark construction, evaluation corpora, and training data for future parser-analysis and grammar-mining tools.

II. Related Work

a) Empirical Studies of Source Code: Our work follows a long tradition of studying what programs programmers actually write. In his seminal study of FORTRAN programs, Knuth [10] counted how frequently certain constructions (such as variable assignments, `IF` conditions, or `GO TO` statements) were used in practice, concluding that "a small number of basic patterns account for most of the programming constructions in use" and "compilers spend most of their time doing surprisingly simple things." Since then, continued interest in how various programming languages are actually used by practitioners in the real world has led to a number of investigations into the usage patterns of particular language features, such as exception handling [11, 12, 13], inheritance [14, 15], or lambda expressions [16, 17, 18]. Dyer et al. [19] showed that, at least in the Java ecosystem, new language features are adopted very slowly. Sihler et al. [20] conducted a large-scale study of R programs to identify their common characteristics, notably with the express purpose of providing insights for static analysis tools. We take a similar empirical-anatomical approach, but focus on a cross-cutting phenomenon rather than a particular language feature: ad hoc parsing, performed through ordinary code, with whatever means a programmer has at their disposal.

b) Anatomy of Python Programs: Several studies provide baselines for Python code in the wild. Peng et al. [21] were the first to systematically analyze the use of language features in Python programs. They found that popular projects tend to use relatively simple features and that the use of more

complex features differs highly between application domains (e.g., DevOps-related projects use the most exception handling, whereas machine learning frameworks use the most decorators). Yang et al. [22] dig deeper into the use of complex Python features, finding that "usage of dynamic features that pose a threat to static analysis is infrequent" but that certain non-trivial features such as context managers and decorators are widely used. They suggest a minimal Python syntax that should be supported by most static analysis tools. Dyer and Chauhan [23] explore the predominant programming paradigms present in Python code. Their results indicate a positive correlation between project size and object-oriented and procedural paradigms, while functional features are commonly used even in predominantly non-functional programs. These studies characterize Python programs generally; our work investigates how Python is used specifically when programs impose structure on strings.

c) Ad Hoc Parsing: To our knowledge, the present work is the first study of ad hoc parsing code in the wild—in any language. The closest related work is by Underwood and Locasto [24], who have used taint tracking to identify the presence of shotgun parsing in Android applications, finding that "most untrusted input propagates a relatively short distance within the application code" but also that "several specific instances of very long data propagations" exist. While they are able to locate shotgun parsing patterns in the control-flow graph of certain methods, their work stops short of identifying the concrete pieces of code that perform the actual parsing. Chapman and Stolee [25] have investigated the use of *regular expressions* in Python programs. They found that over $42\,\%$ of the projects they studied contained at least one regular expression and that the most common uses of regular expressions embedded in Python are for "capturing the contents of brackets, searching for delimiter characters and matching alternative values." Regular expressions are not themselves ad hoc parsers—they are their own grammar!—but they are often used as part of larger ad hoc parsing constructs, where their visible language is further constrained by ordinary Python operations.

III. Locating Ad Hoc Parsers

Any attempt at engaging with the phenomenon of ad hoc parsing—whether for the present purpose of studying the characteristics of ad hoc parsers and the breadth of the phenomenon, or to mine their hidden grammars, or indeed perform any other kind of static or dynamic analysis or transformation of ad hoc parsing code—must necessarily begin by locating those parts of a program that are performing ad hoc parsing.

The challenge in locating ad hoc parsers is deciding when parsing ends: beginning with some input string, if one followed every downstream constraint put on the value or values derived from or structured out of the original string, almost all application logic would need to be considered part of the parsing process; on the other hand, if one stopped immediately after the first string operation, parsers could never consist of more than a single step.

Pragmatically, the clearest cut is at the point of transition from string type to some other object—regardless of the actual program operations performed on either the string or the (partially) structured downstream objects. We can always recognize this transition, and it is always *at least* part of the structuring/parsing process. There might be some edge cases that we also want to include—such as when the structured object is a container of strings (e.g., a list of tokens) or some specialized string-like intermediate structure (e.g., a regex match object)—but the fundamental boundary remains conceptually clear: any operations that are performed directly on the input string itself are parsing; any downstream inspection or manipulation of non-string objects constitutes application logic that is not fundamentally part of string parsing. The delineation between ad hoc parsing and application logic is fluid—a defining characteristic of ad hoc parsing—but we want to firmly plant our feet on the rock-solid island of syntactic parsing amidst the sea of semantic constraints.

Definition 1. *The ad hoc parser of a designated string-typed input variable is the set of program statements, contained within a single function, that read this input variable, or any string-typed variable transitively derived from it, whose hidden grammar comprises exactly those input values that do not cause any of these statements to fail.*

This definition has three important consequences: first, a function may contain multiple ad hoc parsers, one per visible string source; second, ad hoc parsers may overlap: two different input string sources can feed into a shared downstream operation; third, the grammar of an ad hoc parser is defined behaviorally through non-failure, which matches how parsers usually reject inputs in practice.

A. Approach

A complete formalization of our approach to locate ad hoc parser slices is available in our online appendix [26]. Our procedure is language-agnostic and consists of four steps:

① *Control Flow:* Given a program $\mathcal{P}$, we construct its *control-flow graph* G. For Python programs, the CFG can be constructed by a (mostly) top-down traversal of the program's abstract syntax tree (AST). Care must be taken to accurately represent the somewhat baroque control flow occasionally introduced by exception handling.

② *Data Dependence:* We add *data-dependence edges* to G, which model relationships between program statements in terms of variable usage. The classic algorithm [27], based on computing reachable nodes, is quadratic in the number of nodes and entirely sufficient in practice, including for large-scale analysis.

③ *String Data Flow:* We mark all *string data flow* in G using a string oracle Ω. The local oracle classifies variable uses as string-compatible, string-incompatible, or unknown, based on syntax and available type information; a global fixpoint computation propagates these judgments over data-dependence edges until the edge markings stabilize. Provided the oracle satisfies certain properties (purity, monotonicity,

sound negatives, and plausible positives), even incomplete heuristics can yield a conservative approximation of string data flow, where no genuine string flow is ruled out, and every definite string marking is backed by local evidence. Our reproducibility package includes the full formalization of the marking lattice, oracle requirements, fixpoint algorithm, and proofs of termination, confluence, and the resulting soundness guarantees for negative and positive edge markings.

④ *Parser Slices:* We collect the complete set of *parser slices* $\mathcal{A}$ from the marked graph G. Each unique slice starts with a source node, defined by having at least one outgoing string data flow and no incoming string flows. A source node can have multiple distinct variables with outgoing string flows; each node/variable pair establishes a separate parser slice. The slice is grown by forward reachability through data dependence edges. The first step from the slice source node is restricted to string data flow involving the original source variable. All subsequent steps include any string data flow, regardless of variable. This conforms to definition 1 and the notion that any transitively derived string-typed variable continues to be part of the same parsing process. The resulting slice contains all and only those program nodes that are immediately grammar-relevant.

As a final post-processing step, we eliminate all slices deemed *trivial* by a simple conservative procedure that checks if a parser consists entirely of a small set of neutral operations that accept any input string and can never fail, e.g., `print(s.upper())`.

IV. Methodology

This study follows the guidelines for mining software repositories described by Vidoni [28], who delineates three main phases of a systematic study: *planning*, *execution*, and *results*.

A. Planning

a) Research Scope: The objective of this study is to identify the common characteristics and recurring syntactic features of real-world Python code that employs ad hoc parsing techniques. To obtain generalizable results, a large, diverse set of publicly available Python code must be observed.

b) Sources: We use GitHub as the mining ground for our study, and source repositories through the *SEART GitHub Search Engine* (GHS) [29], a continuously updated metadata set containing a selection of repository characteristics for all GitHub repositories with at least ten stars. GHS has been used in studies of Jupyter Notebooks [30] and JavaScript [31, 32].

c) Repository Selection: We are interested in repositories containing Python source code, including multi-language repositories where Python appears only in setup scripts, tests, notebooks, or support tools. Taking into account the known perils of mining GitHub [33], we consider only repositories with at least ten stars and an explicitly permissive software license. These filters reduce purely experimental, ephemeral, or legally unusable repositories; star count and license serve as useful proxies for intentionally engineered code [34, 35].

TABLE I: Repository selection and data extraction funnel.

Phase & Criteria	Count	%
Repository Selection		
All Python repositories in GHS	348 802	100.00
Excluding forks	338 797	97.13
Containing ≥ 1 line of Python	337 153	96.66
Permissive software license	197 771	56.70
Data Extraction		
Size < 100 MB and retrievable	195 640	56.09
Information Extraction		
Containing ≥ 1 analyzable file	191 480	54.90
Excluding clones	189 393	54.30

B. Execution

a) Selection Execution: GHS provides enough metadata to pre-select repositories by the inclusion and exclusion criteria above, resulting in the selection funnel shown in table I. Because GHS calculates line metrics for each repository, repositories without apparent Python code can be excluded before retrieving their contents. Repository selection was executed on 28 August 2025 by downloading metadata for 348 802 repositories indexed by GHS where Python was indicated as the main project language, and filtering this list to 197 771 repositories. The largest reduction came from excluding insufficiently licensed code.

b) Data Extraction: Using the GitHub API, the contents of each selected repository were downloaded as compressed tarballs, provided the archive was smaller than 100 MB. Of the 197 771 selected repositories, 195 640 could be retrieved successfully. The combined size of all downloaded compressed repository tarballs is roughly 1.4 TB.

C. Results

a) Information Extraction: Information extraction was implemented as a high-performance Python pipeline and run on a compute cluster. Repository data was processed directly from compressed tarballs without intermediate on-disk extraction, minimizing I/O overhead and storage requirements. Python files were parsed using the `ast` module from Python's standard library, assuming feature-version compatibility with Python 3.13. For each Python file in each repository archive, the approach described in section III-A identified individual ad hoc parser slices, using a string oracle based on type annotations bundled with version 2.9 of the `typeshed_client` library.Together with slice information, the system extracted code metrics for both parser slices and their surrounding code. A full extraction run was performed 14–16 May 2026. The extracted features were persisted using Apache Parquet and analyzed using DuckDB [36]. The complete pipeline is available open-source at https://anonymous.4open.science/r/parsli, and the complete dataset is archived at (anonymized for review).

b) Results: The analysis pipeline successfully analyzed 189 393 projects, either fully or partially, representing 96.81 % of all selected repositories. Another 6247 projects (3.19 %) contained no analyzable files or were exact clones and were excluded from further analysis. Of the 10 719 714 Python files processed by the system, 197 171 (1.84 %) could not

TABLE II: Prevalence and distribution of parsers

	Total	Prevalence		Distribution			
		≥ 1 Parser	%	Q_1	Med	Q_3	P_{99}
Projects	189 393	183 805	97.05	13	40	119	2017
Files	10 428 379	4 978 623	47.74	1	3	7	54
Functions	91 621 670	18 277 697	19.95	1	1	2	9

TABLE III: Parser retention

	Projects		Files		Functions	
Parsers	%	P_{ret}	%	P_{ret}	%	P_{ret}
≥ 1	97.05	-	47.74	-	19.95	-
≥ 3	92.50	95.31	27.25	57.07	3.34	16.75
≥ 10	78.71	85.09	8.26	30.32	0.16	4.68
≥ 30	55.76	70.83	1.51	18.22	0.01	6.17
≥ 100	27.71	49.70	0.14	9.02	<0.01	5.59
≥ 300	10.38	37.45	0.01	9.05	<0.01	7.89

Number of projects, files, and functions with at least k parsers, for k taking values at uniform log steps of $\approx \sqrt{10}$. The column P_{ret} reports the retention at each step k, i.e., the conditional probability that there are $\geq k$ parsers given that there are already $\geq k/\sqrt{10}$.

be analyzed, most commonly because of outdated Python 2 syntax or timeouts. After excluding cloned projects, the corpus consists of 10 428 379 Python files. The full analysis run took approximately 2.5 days, with a median analysis time of 201 ms (IQR 61–660 ms) per project.

V. Findings

We interleave findings with implications: each subsection reports measurements and closes with a boxed summary of the corresponding research question and its implication for downstream testing and analysis efforts.

A. Ad Hoc Parsers are Everywhere

97.05 % of all Python projects contain at least one ad hoc parser and most contain far more: among projects with parsers, the median is 40 parsers per project, with an IQR of 13–119 (table II). The distribution of parsers is heavily right-skewed: the top 10 % of projects contain more than 319 parsers, the top 1 % more than 2017, and the project with the most parsers contains an astounding 314 889 parsers. In total, our corpus contains 33 215 138 non-trivial ad hoc parsers.

If a project contains at least one parser, it is exceedingly likely to contain three or more (with a probability of 95.31 %; table III), and if a project contains at least three parsers, it is very likely to contain ten or more (with a probability of 85.09 %). Slightly more than half of all projects (55.76 %) contain 30 or more parsers, and about half of those (49.70 %) contain 100 or more. Only above 100 parsers does it become unlikely for a project to contain even more parsers: once a project has reached 100 parsers it has only a 37.45 % chance of reaching 300 parsers.

The small share of projects without parsers (5588, 2.95 % of the corpus) is explained by size and domain: The median project with zero parsers contains only 2 Python files with 7 functions total (against medians of 15 files and 100 functions for projects with at least one parser); and the larger zero-parser projects are mostly data-like repositories.

While almost every project in the entire corpus contains at least one parser, less than half of all Python files (47.77 %) contain any parsers; of those that do, 57.07 % contain three or more.

Of the 9 162 673 functions in the corpus, across all files and projects, 19.95 % contain at least one ad hoc parser, and those functions typically contain just one (IQR 1–2). Only 16.75 % of parser-containing functions contain three or more parsers, and only 4.68 % of those reach ten or more. Parsers accumulate in files by being spread across many functions.

RQ1. How prevalent is ad hoc parsing in Python? Ad hoc parsing is a near-universal phenomenon in our large corpus of 189 393 real-world Python repositories. It occurs in 97.05 % of projects, 47.77 % of files, and 19.95 % of functions, yielding more than 33 million non-trivial ad hoc parsers.

$\Longrightarrow$ **LangSec concerns are empirically significant.** The LangSec community has long cautioned that input recognition and validation should be explicit and separated from computation [3, 37]. Our study cannot show whether any particular parser is insecure—this requires deeper analysis, including inference of the parser's actual grammar—but it clearly shows that the phenomenon LangSec warns about is empirically significant. In Python, at least, ad hoc parsing is nearly universal, affecting one in five functions.

B. Outliers are Mostly Machine-Generated

The top ten projects with the most absolute number of ad hoc parsers account for 4.28 % of all parsers in the dataset. The majority of these are SDKs and client APIs related to Cloud computing and AI, such as the Azure SDK for Python (42 836 files, 314 889 parsers), and the Google Cloud Python Client (30 636 files, 249 706 parsers). The top ten files by parser count, mostly from these projects, are almost all machine-generated (as determined by manual repository inspection); only one seems to be genuinely human-written.

$\Longrightarrow$ **Code generation is not parser generation.** The parser-heaviest files appear entirely machine-generated. Even though the code in these files is generated, the embedded parsers therein are not generated parsers. The specifications from which these codes originate describe API surfaces (e.g., via OpenAPI schemas) but generally do not specify the micro-grammar within messages exchanged by the implemented APIs. The parsing of resource identifiers and similar semi-structured data remains ad hoc. Automatic code generation by itself does not make parsing any more principled—in fact, it multiplies unprincipled ad hoc parsing.

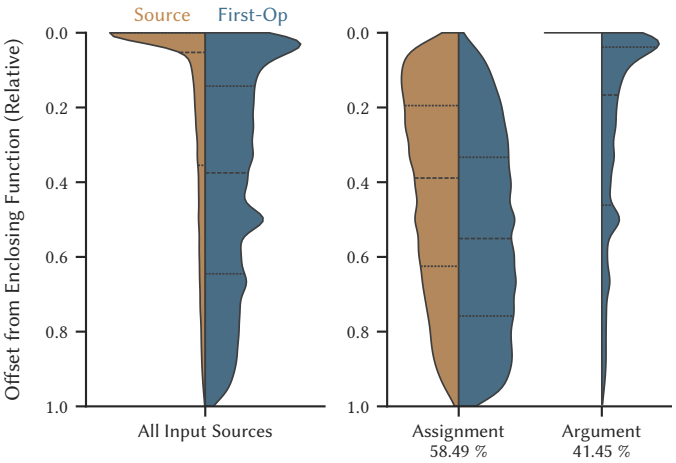


Fig. 2: Relative location of parsers within their enclosing functions, measured from each parser's input source (left) and the first parsing operation acting on that input (First-Op, right); dashed lines mark quartiles. The plots on the right divide parsers by input source.

C. Shotgun Parsing is Real

Looking at where ad hoc parsers are located within their enclosing functions, one could assume that "parse, don't validate" [38] is well-heeded advice: 44.03 % of all ad hoc parser slices begin within the first 5 % of their enclosing function, suggesting that unstructured strings are quickly converted into more precise representations and confounding the assumption that ad hoc parsing is mostly scattershot.

However, these figures are highly misleading: 41.45 % of parsers take their input from a function argument, thus necessarily anchoring them to the beginning of their enclosing functions. When locating parsers by their actual *first parsing operations*—the first statement in a parser slice where the input variable is actually processed, rather than merely introduced—quite a different picture emerges: the median first-operation offset, relative to a parser's enclosing function, indicates that, typically, ad hoc parsing does not begin until about 41.2 % of the way through the enclosing function's body; only 12.69 % of ad hoc parsers actually parse within the first 5 % of a function, and parsers are actually distributed across the entire body of a function rather than amassed at the head (fig. 2). For the gap between location of parser input source and location of first parsing operation, we can see clear differences between types of sources: function arguments (41.45 % of parser inputs) are obviously all sourced at the head of the function and processed later, mostly not far from the beginning, but also distinctly often in the middle of functions; parser inputs from variable assignments (58.49 %) show a slight skew towards earlier sourcing and later processing; input variables introduced via scope directives (`global` or `nonlocal`; 0.06 % of inputs) are similarly introduced early, but are processed all throughout the function; and inputs from `case` pattern matching (<0.01 %) arise all throughout functions and are processed almost immediately after being sourced.

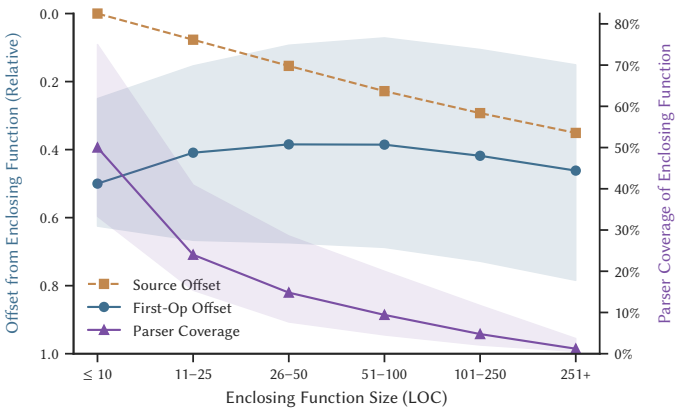


Fig. 3: Relative location of parsers (median) together with parser coverage (i.e., fraction of enclosing function consisting of parsing operations) against size of enclosing function.

The central position of the first parsing operation is remarkably indifferent to the size of the enclosing function (fig. 3). The median relative first-operation offset holds steady between 0.385 and 0.500 across every size class, from ten-line functions to functions of several hundred lines. Parsing proper tends to begin near the midpoint of functions, whether they are short or long.[1] What does change with function size is parser coverage, the fraction of the enclosing function consisting of parsing operations: it collapses from a median of 50 % in functions of ten lines or fewer to 1.21 % in functions of more than 250 lines. This is unsurprising: in small functions that parse, parsing tends to be the main purpose of the function; in the largest functions, parsing is but a tiny sliver.

When a function contains several parsers, they do not pile up in one location but spread out across the entire function (fig. 4). Functions with exactly two parsers keep them close (median spread 0.118), but as the number of parsers increases, so does their spread, with particularly parser-rich functions of eleven or more parsers anchoring their first parser near the head (mean offset 0.139) and the last near the end (0.819; median spread 0.779). Heavily parsing functions thread string processing throughout all their operations, rather than concentrating it in a single block.

RQ2. Where and how are ad hoc parsers embedded within programs? Ad hoc parsers are embedded inside ordinary function bodies in a shotgun manner, not confined to their entry points. Measured by the first actual parsing operation, parsers begin at a median offset of 41.2 % through the function, and only 12.69 % parse within the first 5 %. They are also interleaved with surrounding computation and, when functions contain multiple parsers, spread across the function body rather

[1] Although longer functions place parsers far less predictably, as parser dispersion widens with function size (for large functions of 251+ lines, the first-op offset IQR is 0.151–0.784).

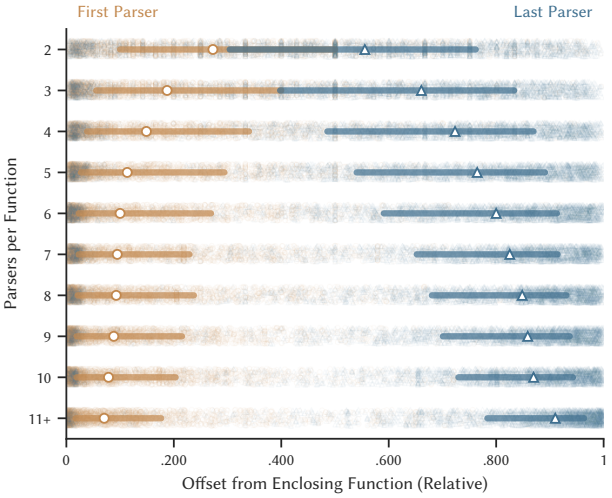


Fig. 4: Relative offsets of first and last parsers in a function, grouped by parser count per function, with highlighted medians and inter-quartile range.

than being collected in a single parsing block.

$\Longrightarrow$ **Fuzzing needs to find well-hidden grammars.** Input parsing acts as a bottleneck to fuzz testing, where programs are bombarded by systematically generated random inputs to see if anything breaks [39, 40, 41]. In order to penetrate deeply into application logic, inputs need to pass all syntactic checks and transformations scattered throughout a program—fuzz inputs need to pass ad hoc parsers. In grammar-based fuzzing, valid inputs are specified with the help of input grammars to do exactly that. However, our findings suggest that the grammar problem is much broader than typically assumed: Hidden grammars are not confined to program boundaries or obvious input modules and reach well beyond file formats or protocols front-ends. Ad hoc parsers hiding input grammars appear all throughout ordinary functions, often beginning well after the function boundary, and frequently overlapping with each other. A fuzzer that only considers grammars at external input interfaces may therefore still fail to reach code guarded by local ad hoc parsers.

D. Ad Hoc Parsers are Small, not Simple

Ad hoc parsers are small. A quarter of all ad hoc parsers (26.94 %) are just one or two lines of code in size, and another third (34.51 %) are between three and five lines; fewer than 7 % of parsers are more than twenty lines long (table IV). The extent of an ad hoc parser within its enclosing function, from the parser's source variable to the last parsing operation, consistently spans roughly $1.7\times$ the lines of code of the parser itself (overall mean parser density 0.58 ± 0.32)—irrespective of the size of the parser (only very small and very large parsers are slightly denser). This aligns with parser coverage shrinking as the enclosing function grows in size (fig. 3): an ad hoc parser is a compact, fairly local, piece of code; it is usually very small, and almost never spread out over large distances.

Parsers appear in three shapes, based on their data flow:

- *Linear* parsers form straight pipelines of successive string transformations, where the source variable is processed into some other string value (e.g., by subscripting), which is passed on to another string operation, whose result is passed on again, and so on, until a final sink operation that returns some non-string value;
- *Branching* parsers pass values on to multiple (parallel) consumers (e.g., by splitting a string into multiple substrings) who can then each continue to pass on their result, tree-like, to one or more further operations, until each branch reaches a sink;
- *Convergent* parsers first branch out but then eventually merge back the results along some of their branches, combining multiple intermediate results into one single result to then be further processed (e.g., concatenating some parts of a previous split operation).

Over the entire corpus, parsers are predominantly linear (48.79 %) or branching (41.75 %); few are convergent (9.45 %). This overall split is dominated by the smallest parsers, however, and shifts markedly with size (table IV). Parsers of one or two lines are naturally almost entirely linear (99.88 %), and account for most of the share of linear parsers. Among parsers large enough to meaningfully branch, branching is in fact most common: at three to five lines, two thirds (65.71 %) of parsers are branching, and only a quarter (26.09 %) are linear; at six to ten lines, half (53.48 %) are branching, a third (30.59 %) are linear, and 15.93 % are convergent. Convergent parsers also grow steadily more common with parser size, accounting for 23.42 % of parsers with fifty to a hundred lines: the larger a parser, the likelier it is to recombine intermediate results. Only the very largest parsers (with more than a hundred lines) revert to a somewhat more linear profile, most likely due to the high prevalence of machine-generated code among parsers of that size.

In addition to internal parser convergence, we can observe convergence between distinct parsers. While each parser uniquely begins by parsing its own input source variable, downstream parsing operations might be shared among parsers by intersecting data flows. In the whole corpus, nearly half of all parsers (47.93 %) intersect with at least one other parser, overlapping on 63.21 ± 25.37 % of operations on average (relative to each intersecting parser). As with internally convergent parsers, between-parser intersections increase with parser size (table IV), although they start at a much higher rate: even among the smallest parsers of up to two lines, a third (33.74 %) already intersect with another parser; this share grows logarithmically with parser size, peaking at 85.64 % for parsers of fifty to a hundred lines. The amount of overlap between intersecting parsers remains relatively similar between parsers of different sizes, from 61.40 ± 20.99 % for very small parsers, to 65.25 ± 28.50 % for parsers of up to 20 lines, to 72.68 ± 26.34 % for parsers of more than a hundred.

TABLE IV: Parsers by size

	Parsers		Extent (LOC)			Density		Shape (%)			Intersecting	Overlap	
Size (LOC)	Count	%	Q_1	Med	Q_3	Mean	SD	Linear	Branch.	Conv.	%	Mean	SD
1–2	8 948 115	26.94	2	3	6	0.68	0.35	99.88	0.09	0.03	33.74	61.40	20.99
3–5	11 460 742	34.51	4	7	14	0.56	0.32	26.09	65.71	8.20	42.49	60.88	22.93
6–10	6 603 424	19.88	10	15	26	0.54	0.29	30.59	53.48	15.93	53.85	62.80	26.57
11–20	4 000 716	12.05	19	29	47	0.52	0.27	34.99	47.00	18.01	67.49	65.25	28.50
21–50	1 859 426	5.60	39	57	92	0.54	0.26	39.39	41.88	18.74	79.74	70.05	29.30
51–100	252 123	0.76	83	119	192	0.57	0.27	34.41	42.18	23.42	85.64	71.71	29.26
101+	90 592	0.27	171	259	423	0.69	0.28	44.01	36.31	19.68	82.94	72.68	26.34
All	33 215 138	100.00	4	10	23	0.58	0.32	48.79	41.75	9.45	47.93	63.21	25.37

Extent is the distance between the parser's input variable and its last operation; density is size/extent.
Overlap is the average proportion (%) of statements shared with an intersecting parser, if there is one.

RQ3. What is the shape and structure of ad hoc parser slices? Ad hoc parser slices are small and local, but not necessarily simple. Most are at most five lines long (61.45 %), and fewer than 7 % exceed twenty lines. Their data flow is usually linear (48.79 %) or branching (41.75 %), with convergence becoming more common as parsers grow. Parser slices also frequently share downstream operations: nearly half (47.93 %) intersect with another parser, often overlapping on much of their parsing logic.

$\Longrightarrow$ **From grammar mining to grammar inference.** Dynamic grammar mining approaches [4, 5, 6, 7] infer input languages from observed executions; thus, they require executable programs, suitable harnesses, and representative bootstrap inputs. Our results suggest that this cannot scale in practice: ad hoc parsers are too numerous, too widely distributed, and too often embedded within ordinary application logic to be comprehensively exposed by dynamic observation. Static grammar inference approaches [8, 9] are more promising for this setting, but they must account for the actual shape of ad hoc parsing in the wild: small parser slices, mid-function parsing, overlapping string flows, calls into common library operations, regular expressions embedded in larger slices, and implicit failure through exceptions.

TABLE V: Most common AST nodes in parsers

AST Node	P	¬P	Δ	×
`Name`	100.00	93.73	+6.27	1.1
`Load`	99.99	94.05	+5.95	1.1
`Call`	92.33	79.63	+12.70	1.2
`Store`	85.29	59.96	+25.33	1.4
`Attribute`	83.21	84.22	−1.00	1.0
`Constant`	81.63	77.25	+4.38	1.1
`Assign`	78.80	56.84	+21.96	1.4
`Expr`	37.99	62.48	−24.49	0.6
`keyword`	29.22	29.75	−0.53	1.0
`If`	29.09	24.90	+4.19	1.2
`Compare`	26.47	26.09	+0.38	1.0
`Subscript`	24.73	24.82	−0.09	1.0
`BinOp`	20.97	18.12	+2.85	1.2
`Tuple`	18.92	20.02	−1.10	0.9
`Return`	17.90	52.43	−34.54	0.3
`Add`	16.69	10.44	+6.25	1.6
`Eq`	13.84	11.95	+1.89	1.2
`JoinedStr`	13.69	4.19	+9.51	3.3
`FormattedValue`	13.51	4.11	+9.40	3.3
`For`	12.09	10.53	+1.56	1.1
`UnaryOp`	9.33	12.51	−3.18	0.7
`List`	9.17	17.19	−8.02	0.5
`BoolOp`	8.82	6.75	+2.07	1.3
`Dict`	7.40	9.10	−1.70	0.8
`Not`	6.39	6.61	−0.22	1.0
`In`	6.23	3.78	+2.45	1.6
`And`	5.84	4.60	+1.24	1.3
`Mod`	5.56	2.86	+2.70	1.9
`Raise`	5.07	5.80	−0.73	0.9

Prevalence (%) in parsers P vs. in non-parser code ¬P (functions without any parsers), with absolute difference Δ and ratio ×. Only nodes with $P \geq 5.00$ are shown.

E. Ordinary Operations Encode Grammars

Direct string manipulation methods such as `split`, `replace`, `strip`, `startswith`, or `lower` are unsurprisingly prominently called in ad hoc parsers. But parsing is not just stringing together a sequence of string primitives; instead, parsers call a variety of functions to combine string construction, path manipulation, type conversion, validation, collection updates, and ordinary program plumbing (table VI).

The most prevalent function call is `format`, appearing in 9.81 % of parsers and accounting for 4.89 % of call occurrences in parsers. Strings are mostly used as arguments to `format` (in 78.58 % of such calls), suggesting the use of formatting before outputting a (partially) parsed string component; however, 8.84 % of `format` calls involve parser-relevant string variables as formatting templates, which has safety implications (cf. section V-F).

Path construction is similarly prominent and suggests that a substantial fraction of parsing appears near file and path boundaries: `os.path.join` is used in 7.27 % of parsers to combine string inputs interpreted as path components, similar to other filesystem-adjacent calls like `open` (4.41 % of parsers) and `os.path.exists` (1.99 %).

Function calls show which APIs parsers invoke, but much of ad hoc parsing is expressed through Python syntax rather than named functions. Looking at raw AST nodes gives us insight into the fundamental operations that make up parsing

TABLE VI: Top Function Calls in Parsers

		Occurence		String Role (% calls)		
	P	Calls	%	Recv.	Arg.	Ret.
`format`	9.81	5 169 975	4.89	8.84	78.58	100
`os.path.join`	7.27	4 124 038	3.90	-	81.41	100
`append`	5.11	2 324 117	2.20	-	31.29	-
`str`	4.75	2 690 972	2.55	-	16.28	100
`len`	4.46	2 236 047	2.12	-	50.93	-
`open`	4.41	1 798 398	1.70	-	77.75	-
`print`	4.11	1 838 399	1.74	-	21.08	-
`join`	3.84	1 535 336	1.45	8.38	16.57	100
`get`	3.75	1 786 549	1.69	-	31.18	-
`split`	3.49	1 533 282	1.45	69.98	9.47	-
`replace`	3.06	1 910 883	1.81	68.49	18.11	100
`int`	2.97	1 618 712	1.53	-	56.66	-
`strip`	2.93	1 327 835	1.26	52.55	0.61	100
`isinstance`	2.45	1 198 064	1.13	-	80.81	-
`super`	2.39	805 017	0.76	-	2.03	-
`startswith`	2.32	1 077 474	1.02	85.27	17.66	-
`lower`	2.14	1 055 484	1.00	68.15	-	100

P indicates prevalence (%) in parsers. String role reports the share of calls in which a parser string is used as a receiver, argument, or return value; roles are not mutually exclusive. Only calls with $P \geq 2.00$ are shown.

idioms (table V).

The most common operation in all Python code, including ad hoc parsing code, is variable referencing, expressed in the Python AST by the node type `Name`, which occurs in $100\,\%$ of all ad hoc parsers—simply because every parser by definition has at least an input variable. Most often, variable references are used for reading values, indicated by the presence of the `Load` context node (occurring in $99.99\,\%$ of ad hoc parsers).[2] Writes are less common than reads in parsing, with the `Store` context node appearing in $85.29\,\%$ of parsers, but much more common than in non-parsing code ($+25.33\,\%$), as are assignments (`Assign`, present in $78.80\,\%$ of parsers, $+21.96\,\%$ compared to non-parsing code)—this is again partly by construction: a parser slice follows partially parsed strings across reassignments. Function calls are likewise more prevalent in parsers than in ordinary code (occurring in $92.33\,\%$ of parsers, $+17.92\,\%$ over non-parsing code), while at the same time bare expression statements (`Expr`) are correspondingly rarer (in $37.99\,\%$ of parsers, $-24.49\,\%$), which indicates that parsers thread values into variables, rather than invoking functions for their side effects.

String construction is the clearest distinguishing trait of ad hoc parsing code: formatted string literals or *f-strings*, which consist of `JoinedStr` and `FormattedValue` nodes, are 3.3× as prevalent in parsers as in non-parser code; and binary operators commonly used for concatenation (`Add`) and `printf`-style percent formatting (`Mod`) are notably 1.6× and 1.9× as common as in non-parsing functions. These elevated AST node frequencies reflect the widespread practice of string construction as part of ad hoc parsing.

Membership and Boolean tests are also notably more common in parser code: `In` (1.6×), `NotIn` (1.7×), `Eq` (1.2×), `NotEq` (1.3×), `And` (1.3×), `Or` (1.3×), `If` (1.2×), `IfExpr` (1.5×) all occur significantly more often during ad hoc parsing.

Curiously, string indexing and slicing is *not* more common among ad hoc parsing code: `Subscript` occurs in $24.73\,\%$ of parsers, pretty much exactly the same prevalence as in non-parsing code; `Slice` occurs in only $4.56\,\%$ of parsers, 0.9× the prevalence in non-parsing code.

Taken all together, the mix of operations found in ad hoc parsers strongly suggests that they are highly amenable to static analysis: they rarely feature the kind of complex and dynamic features identified by Yang et al. [22] that impede static analysis in Python (table VII).

[2] `Load` has only $99.99\,\%$ prevalence due to `Name` sometimes being used in contexts that semantically both read and write a variable (e.g., an augmented assignment +=), which in the Python AST is only assigned a `Store` context.

TABLE VII: Complex Features in Parsers

		All Python*	
Feature	Ad Hoc Parsers	2021	2019
FNL (functional)	4.94	26.42	28.16
DYN (dynamic)	3.17	12.97	16.27
DEC (decorators)	-	12.48	14.56
WTH (with)	4.73	12.46	9.31
ASC (async)	0.12	0.06	0.20

* As reported by Yang et al. [22].

RQ4a. Which programming constructs ... characterize ad hoc parsers? Ad hoc parsers are built from ordinary Python operations, combining string methods, function calls, assignments, string construction, membership tests, Boolean guards, and conversions. String construction is their clearest distinguishing trait: f-strings are 3.3× as prevalent in parsers as in non-parser code, and almost a third of parsers construct strings by concatenation.

$\Longrightarrow$ **Ad hoc parsers are statically analyzable.** Because ad hoc parsers rarely use complex or dynamic features, favoring mostly ordinary local Python idioms, their grammars are not hidden behind exotic language syntax or require dynamic evaluation. Static analyses can make substantial progress by precisely modeling common string, path, formatting, conversion, and Boolean operations.

F. Exception Handling is Rare

Despite processing weakly structured input from all kinds of sources, ad hoc parsers usually do not bother with exception handling: for $91.43\,\%$ of parsers, not a single one of their statements is in the scope of an exception handler; only $8.57\,\%$ have any exception coverage at all, and only $3.30\,\%$ have every one of their statements covered.

Absence of exception handling by itself does not imply fragility: potentially raising statements may be guarded by preconditional checks, constrained by the surrounding control flow to avoid error paths, or operate exclusively on values whose shape is well known. Parsers also might simply not contain that many potentially exceptional operations.

To conservatively estimate exception exposure, we mark parser statements that invoke operations whose documented

failures can plausibly depend on runtime input or the execution environment, such as numeric conversion, indexing, structured-data parsing, file access, or dynamic format templates. We exclude failures caused only by programmer contract violations, such as malformed regular-expression constants or non-dynamic format string literals.

Using this measure, we identify 19.87 % of parsers that contain some operation that can raise an exception (table VIII). The most common exposed exceptions are `OSError` (occurring in 7.35 % of all parsers), `ValueError` (5.81 %), and `IndexError` (4.60 %). Among the parsers exposed to these and other exceptions, only 9.04 % have the relevant statements fully covered by some compatible exception handler.

Coverage varies substantially by exception. Some recognizable anticipated failure modes are handled more often: `CalledProcessError` exposure is covered in 34.33 % of exposed parsers, `ImportError` in 28.97 %, and `JSONDecodeError` in 25.40 %. By contrast, common parser-related operations involving often unknown inputs, such as string indexing or dynamic formatting, are much less often covered: `IndexError` is caught in only 5.14 % of exposed parsers, and exceptions exposed by dynamic `str.format()` operations are covered at a rate of roughly 3 %.

It is not that ad hoc parsers are generally unsafe, but that their exception handling is highly selective. Most parsing proceeds without exception coverage, and many statically identifiable runtime failure sites are not paired with handlers that would catch their documented exceptions. Nearly one in five ad hoc parsers (18.07 %) expose unhandled exceptions.

RQ4b. Which ... failure modes characterize ad hoc parsers? Ad hoc parsers usually leave exceptional parsing failure implicit. For 91.43 % of parsers, no statement is covered by exception handling, and only 3.30 % are fully covered; at least 18.07 % of parsers contain statements that can raise unhandled exceptions.

TABLE VIII: Exception exposure in parsers

	Parsers	%	C%$_\forall$	U%
Any Exception	6 598 755	19.87	9.04	18.07
`OSError`	2 439 874	7.35	8.75	6.70
`ValueError`	1 929 145	5.81	11.43	5.14
`IndexError`	1 528 102	4.60	4.34	4.40
`LookupError`	929 542	2.80	4.73	2.67
`UnicodeEncodeError`	574 649	1.73	5.35	1.64
`UnicodeDecodeError`	406 304	1.22	9.84	1.10
`JSONDecodeError`	345 059	1.04	24.94	0.78
`TypeError`	339 338	1.02	2.71	0.99
`AttributeError`	339 323	1.02	2.64	1.00
`KeyError`	339 323	1.02	3.10	0.99
`SyntaxError`	118 723	0.36	22.04	0.28
`CalledProcessError`	58 216	0.18	33.56	0.12
`ImportError`	47 345	0.14	27.76	0.10
`binascii.Error`	39 423	0.12	16.61	0.10
`struct.error`	33 071	0.10	6.10	0.09
`InvalidFileException`	545	<0.01	17.06	<0.01
`TOMLDecodeError`	223	<0.01	29.15	<0.01

Reports the number and share of parsers exposed to the given exception, i.e., containing at least one operation that can plausibly raise it based on dynamic input data. C%$_\forall$ is the share of exposed parsers in which the given exception is caught by an exception handler in all instances. U% is the share of all parsers containing at least one uncaught raising operation for the given exception.

VI. Threats to Validity

a) Internal Validity: Our analysis is based on a large-scale dataset of open-source projects collected from GitHub. This dataset may not represent Python code in general and thus may not reflect the actual nature of ad hoc parsers in Python. We mitigate this risk by using a large corpus selected through GHS, filtering for intentionally engineered and licensed projects, and relying on the corpus's scale and diversity to reduce the chance that the results are artifacts of a few unusual repositories.

b) External Validity: We only consider ad hoc parsers written in Python. Their characteristics may partly reflect Python's language features and community practices, so the findings may not generalize to other programming languages. Even if they do not replicate across languages, they remain valuable for the Python ecosystem and its program-analysis efforts. We nevertheless conjecture that many observed patterns, such as small local parser slices that begin mid-function, occur naturally in ad hoc parsing more broadly, especially in comparable imperative, dynamically typed languages.

c) Construct Validity: Our method for locating ad hoc parsers (section III-A) may be unsound or incomplete: it may capture irrelevant fragments or miss legitimate parsers. Our operational definition intentionally captures only intra-procedural string data flow, missing parsers spread across functions, opaque library calls, or non-string intermediate representations; conversely, despite filtering, it may include trivial parsers whose accepted language is universal. We mitigate this threat with control-flow and data-dependence analysis rather than surface-level syntactic pattern matching, and with formalization and correctness proofs of the string-flow marking procedure in our reproducibility package. The results should therefore be read as results about parsers under our operational definition, not as a perfect census of every ad hoc parser a human might identify.

VII. Conclusion

Ad hoc parsing and the attendant LangSec concerns are an ecosystem-level challenge. A single two-line parser may be unremarkable, but tens of millions of such parsers, many embedded in ordinary functions and relying on implicit failure behavior, require serious attention. Our work locates these parsers and makes them visible at scale: more than 33 million ad hoc parser slices across 189 393 Python projects; small but complex; local but spread throughout ordinary program logic. By locating millions of real ad hoc parser slices, our work provides a robust foundation to construct evaluation datasets, benchmark suites, and training data for learning-based tools, as well as guidance for the design and evolution of novel grammar analyses [6, 8, 9].

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," *IEEE Security & Privacy*, vol. 12, no. 1, pp. 83–87, 2014. DOI: 10.1109/MSP.2014.1.

[2] S. Bratus et al., "Curing the vulnerable parser: Design patterns for secure input handling," *;login:*, vol. 42, no. 1, pp. 32–39, Spr. 2017. [Online]. Available: https://www.usenix.org/publications/login/spring2017/bratus.

[3] F. D. Momot, S. Bratus, S. M. Hallberg, and M. L. Patterson, "The seven turrets of babel: A taxonomy of langsec errors and how to expunge them," in *2016 IEEE Cybersecurity Development*, Proceedings (Boston, MA, USA, Nov. 3–4, 2016), IEEE Computer Society, pp. 45–52. DOI: 10.1109/SecDev.2016.019.

[4] M. R. Arefin, S. Shetiya, Z. Wang, and C. Csallner, "Fast deterministic black-box context-free grammar inference," in *ICSE'24, Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, (Lisbon, Portugal, Apr. 14–20, 2024), ACM, 2024, 117. DOI: 10.1145/3597503.3639214.

[5] N. Kulkarni, C. Lemieux, and K. Sen, "Learning highly recursive input grammars," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering*, Proceedings (Virtual, Nov. 15–19, 2021), IEEE Computer Society, 2021, pp. 456–467. DOI: 10.1109/ASE51524.2021.9678879.

[6] R. Gopinath, B. Mathis, and A. Zeller, "Mining input grammars from dynamic control flow," in *ESEC/FSE '20, Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, (Virtual, Nov. 8–13, 2020), ACM, 2020, pp. 172–183. DOI: 10.1145/3368089.3409679.

[7] M. Höschele and A. Zeller, "Mining input grammars from dynamic taints," in *ASE'16, Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, (Singapore, Sep. 3–7, 2016), ACM, 2016, pp. 720–725. DOI: 10.1145/2970276.2970321.

[8] M. Schröder and J. Cito, "Static inference of regular grammars for ad hoc parsers," *Proceedings of the ACM on Programming Languages*, vol. 9, pp. 113–143, OOPSLA2 Oct. 2025. DOI: https://doi.org/10.1145/3763054.

[9] L. Bettscheider and A. Zeller, "Look ma, no input samples! mining input grammars from code with symbolic parsing," in *FSE Companion '24, Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, (Porto de Galinhas, Brazil, Jul. 15–19, 2024), ACM, 2024, pp. 522–526. DOI: 10.1145/3663529.3663790.

[10] D. E. Knuth, "An empirical study of FORTRAN programs," *Software: Practice and Experience*, vol. 1, no. 2, pp. 105–133, Apr.–Jun. 1971. DOI: 10.1002/spe.4380010203.

[11] B. Cabral and P. Marques, "Exception handling: A field study in Java and .net," in *ECOOP 2007—Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30–August 7, 2007, Proceedings*, Springer, 2007, pp. 151–175. DOI: 10.1007/978-3-540-73589-2_8.

[12] M. B. Kery, C. Le Goues, and B. A. Myers, "Examining programmer practices for locally handling exceptions," in *13th Working Conference on Mining Software Repositories*, Proceedings (Austin, TX, USA, May 14–15, 2016), ACM, 2016, pp. 484–487. DOI: 10.1145/2901739.2903497.

[13] S. Nakshatri, M. Hegde, and S. Thandra, "Analysis of exception handling patterns in Java projects: An empirical study," in *13th Working Conference on Mining Software Repositories*, Proceedings (Austin, TX, USA, May 14–15, 2016), ACM, 2016, pp. 500–503. DOI: 10.1145/2901739.2903499.

[14] E. Tempero, J. Noble, and H. Melton, "How do Java programs use inheritance? an empirical study of inheritance in Java software," in *ECOOP 2008—Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7–11, 2008, Proceedings*, Springer, 2008, pp. 667–691. DOI: 10.1007/978-3-540-70592-5_28.

[15] M. Orrú, E. Tempero, M. Marchesi, and R. Tonelli, "How do Python programs use inheritance? a replication study," in *22nd Asia-Pacific Software Engineering Conference*, Proceedings (New Delhi, India, Dec. 1–4, 2015), IEEE Computer Society, 2015, pp. 309–315. DOI: 10.1109/APSEC.2015.51.

[16] D. Mazinanian, A. Ketkar, N. Tsantalis, and D. Dig, "Understanding the use of lambda expressions in Java," *Proceedings of the ACM on Programming Languages*, vol. 1, OOPSLA Oct. 2017, Art. no. 85. DOI: 10.1145/3133909.

[17] S. Nielebock, R. Heumüller, and F. Ortmeier, "Programmers do not favor lambda expressions for concurrent object-oriented code," *Empirical Software Engineering*, vol. 24, pp. 103–138, May 2, 2018. DOI: 10.1007/s10664-018-9622-9.

[18] A. E. Rao and S. Chimalakonda, "An exploratory study towards understanding lambda expressions in Python," in *Proceedings of EASE 2020—Evaluation and Assessment in Software Engineering*, (Trondheim, Norway, Apr. 15–17, 2020), ACM, 2020, pp. 318–323. DOI: 10.1145/3383219.3383255.

[19] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen, "Mining billions of ast nodes to study actual and potential usage of Java language features," in *36th International Conference on Software Engineering (ICSE 2014)*, Proceedings (Hyderabad, India, May 31–Jun. 7, 2014), ACM, pp. 779–790. DOI: 10.1145/2568225.2568295.

[20] F. Sihler, L. Pietzschmann, R. Straub, M. Tichy, A. Diera, and A. Dahou, "On the anatomy of real-world r code for static analysis," in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, pp. 619–630. DOI: 10.1145/3643991.3644911.

[21] Y. Peng, Y. Zhang, and M. Hu, "An empirical study for common language features used in Python projects," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering*, Proceedings (Virtual, Mar. 9–12, 2021), IEEE Computer Society, 2021, pp. 24–35. DOI: 10.1109/SANER50967.2021.00012.

[22] Y. Yang, A. Milanova, and M. Hirzel, "Complex Python features in the wild," in *The 2022 Mining Software Repositories Conference*, Proceedings (Pittsburgh, PA, USA, May 23–24, 2022), ACM, 2022, pp. 282–293. DOI: 10.1145/3524842.3528467.

[23] R. Dyer and J. Chauhan, "An exploratory study on the predominant programming paradigms in Python code," in *ESEC/FSE '22, Proceedings of the 30th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, (Singapore, Nov. 14–18, 2022), ACM, pp. 684–695. DOI: 10.1145/3540250.3549158.

[24] K. Underwood and M. E. Locasto, "In search of shotgun parsers in android applications," in *2016 IEEE Symposium on Security and Privacy Workshops*, Proceedings (San Jose, CA, USA, May 23–25, 2016), IEEE Computer Society, 2016, pp. 140–155. DOI: 10.1109/SPW40107.2016.

[25] C. Chapman and K. T. Stolee, "Exploring regular expression usage and context in Python," in *ISSTA'16, Proceedings of the 25th International Symposium on Software Testing and Analysis*, (Saarbrücken, Germany, Jul. 18–20, 2016), ACM, 2016, pp. 282–293. DOI: 10.1145/2931037.2931073.

[26] Anonymous author(s). "Supplement to 'An Empirical Analysis of Ad Hoc Parsers in Python'."

[27] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, Jul. 1987. DOI: 10.1145/24039.24041.

[28] M. Vidoni, "A systematic process for mining software repositories: Results from a systematic literature review," *Information and Software Technology*, vol. 144, Apr. 2022, Art. no. 106791. DOI: 10.1016/j.infsof.2021.106791.

[29] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in github for msr studies," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories*, Proceedings (May 22–30, 2021), IEEE Computer Society, 2021, pp. 560–564. DOI: 10.1109/MSR52588.2021.00074.

[30] K. Grotov, S. Titov, V. Sotnikov, Y. Golubev, and T. Bryksin, "A large-scale comparison of python code in jupyter notebooks and scripts," in *The 2022 Mining Software Repositories Conference*, Proceedings (Pittsburgh, PA, USA, May 23–24, 2022), ACM, 2022, pp. 353–364. DOI: 10.1145/3524842.3528447.

[31] F. Alves, D. Oliveira, F. Madeiral, and F. Castor, "On the bug-proneness of structures inspired by functional programming in javascript projects," Jun. 17, 2022, arXiv. DOI: 10.48550/arXiv.2206.08849, preprint.

[32] W. Lucas et al., "Understanding the adoption of modern javascript features: An empirical study on open-source systems," *Empirical Software Engineering*, vol. 30, May 5, 2025, Art. no. 107. DOI: 10.1007/s10664-025-10663-9.

[33] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *11th Working Conference on Mining Software Repositories (MSR 2014)*, Proceedings (Hyderabad, India, May 31–Jun. 1, 2014), ACM, 2014, pp. 92–101. DOI: 10.1145/2597073.2597074.

[34] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, pp. 3219–3253, Dec. 2017. DOI: 10.1007/s10664-017-9512-6.

[35] H. Borges and M. T. Valente, "What's in a github star? understanding repository starring practices in a social coding platform," *Journal of Systems and Software*, vol. 146, pp. 112–129, Dec. 2018. DOI: 10.1016/j.jss.2018.09.016.

[36] M. Raasveldt and H. Mühleisen, "Duckdb: An embeddable analytical database," in *SIGMOD'19, Proceedings of the 2019 International Conference on Management of Data*, (Amsterdam, Netherlands, Jun. 30–Jul. 5, 2019), ACM, 2019, pp. 1981–1984. DOI: 10.1145/3299869.3320212.

[37] L. Sassaman, M. L. Patterson, S. Bratus, and M. E. Locasto, "Security applications of formal language theory," *IEEE Systems Journal*, vol. 7, no. 3, pp. 489–500, 2013. DOI: 10.1109/JSYST.2012.2222000.

[38] A. King, "Parse, don't validate," Nov. 5, 2019. [Online]. Available: https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/.

[39] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990. DOI: 10.1145/96267.96279.

[40] V. J. Manès et al., "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2021. DOI: 10.1109/TSE.2019.2946563.

[41] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024. Accessed: Jul. 1, 2024. [Online]. Available: https://www.fuzzingbook.org/.