Bootstrapping Automated Testing for RESTful Web Services

Zhanyao Lei, Yixiong Chen, Yang Yang, Mingyuan Xia, and Zhengwei Qi, Member, IEEE

Abstract—Modern RESTful services expose RESTful APIs to integrate with diversified applications. Most RESTful API parameters are weakly typed, which greatly increases the possible input value space. Weakly-typed parameters pose difficulties for automated testing tools to generate effective test cases to reveal web service defects related to parameter validation. We call this phenomenon the type collapse problem. To remedy this problem, we introduce FET (Format-encoded Type) techniques, including the FET, the FET lattice, and the FET inference to model fine-grained information for API parameters. Inferred FET can enhance parameter validation, such as generating a parameter validator for a certain RESTful server. Enhanced by FET techniques, automated testing tools can generate targeted test cases. We demonstrate Leif, a trace-driven fuzzing tool, as a proof-of-concept implementation of FET techniques. Experiment results on 27 commercial services show that FET inference precisely captures documented parameter definitions, which helps Leif discover 11 new bugs and reduce 72% - 86% fuzzing time compared to state-of-the-art fuzzers. Leveraged by the inter-parameter dependency inference, Leif saves 15% fuzzing time.

Index Terms—Fuzz testing, RESTful web service, type inference

1 INTRODUCTION

The REST (Representational State Transfer) architecture [1] nowadays has dominated the design of complex web services, such as public clouds (e.g., AWS and Azure), social networking (e.g., Facebook and Twitter), and code hosting (e.g., GitHub and GitLab). Typically, a RESTful web service exposes a set of RESTful APIs. A client requests an API providing parameter values, and the service responds with data represented in a certain common exchange format (e.g., JSON or XML). According to a recent survey of 40 real-world popular RESTful web services [2], modern services involve an average of 64 APIs and over 20 parameters per API. Testing such an input space of possible parameter value combinatorics is challenging, so automated testing is indispensable.

Since RESTful APIs are intended for applications implemented by different programming languages, API parameters are weakly typed. An investigation on 27 RESTful web services [3] shows that over 67% of the parameters are string-typed, about 32% are number-typed, and the remaining 1% are boolean-typed or object-typed. Overusing primitive data types significantly increases the possible input value space. For example, a string-typed parameter can take values varying from a specific URL to a comment about a YouTube video. Weakly-typed parameters pose difficulties for generating effective test cases. Consequently, many automated REST testing tools are ineffective while RESTful web services suffer from various input-related attacks, such as integer overflow attacks and SQL injection attacks [4]. We call this phenomenon the type collapse problem.

The solution is to bridge the gap for automated testing tools to have a better understanding of parameters. We observe that though parameter types are weak, their values usually have distinct formats. For example, a datetime parameter may require an ISO8601 date string. This motivates us to introduce the FET (Format-encoded Type) which combines data types and value formats to describe parameters in fine-grains. For instance, the SHA1 represents 40-digit-hex string-typed parameters. Furthermore, we introduce the FET lattice, which hierarchically organizes a set of FETs by a partial order, along with the FET inference, which seeks suitable FETs among a FET lattice for parameters in an unambiguous manner.

We implement Leif, a trace-driven fuzzing tool, to manifest how to enhance automated REST testing by FET techniques. Leif gains fine-grained parameter information by performing FET inference on HTTP traffic and then mutates parameter values to mimic real attacks based on the inferred results. Leif also generates the corresponding parameter validation code to help fix potential bugs in the RESTful service under test. We apply Leif to real-world web services, and the experiment results are encouraging. FET techniques provide better bug-finding capability and bring 72% - 86% fuzzing time reduction for Leif compared to state-of-the-art fuzzing tools. With FET techniques, parameter validation code can be generated and embedded in real-world RESTful client/server code. Leveraged by the inter-
parameter dependency inference, Leif saves 15% fuzzing time.

In particular, this paper makes the following contributions:

- We introduce FET techniques, including the FET, the FET lattice, and the FET inference, to remedy the type collapse problem and serve as a cornerstone for high-level automated testing tools.
- We implement Leif, a FET-enhanced fuzzing tool that showcases how to construct a ubiquitous FET lattice for common RESTful APIs and embed FET techniques in an existing testing workflow.
- We evaluate the accuracy of FET inference, and the result is encouraging (67% exact matches, 32% partial matches, and 1% mismatches on average).
- We evaluate Leif’s bug-finding capability (11 distinct bugs detected in 27 commercial web services) as well as its testing efficiency (72% – 86% fuzzing time reduction as compared to existing fuzzing tools).
- We evaluate Leif’s code generation capability by showcasing generated code snippets for different applications and programming languages.
- We explore how the inter-parameter dependency inference help Leif reduce fuzzing time (nearly 15% fuzzing time reduction).

In our previous conference version of this work [5], we introduced FET techniques, implemented Leif, and evaluated the accuracy of FET inference and Leif’s bug-finding capability. In this journal paper, we extend our previous study from the following aspects: (1) We observe and discuss that FET techniques can enhance parameter validation, such as generating parameter validation code for a certain RESTful server or client (Section 4.4). (2) We extend Leif to support parameter validation code generating. In Section 5.5, we choose some real-world bugs Leif found and showcase some code snippets generated by Leif to explore how FET techniques integrates with real-world applications. We also find that FET techniques can enhance standard RESTful specifications and existing specification-based code generators. (3) We discuss the randomness in RESTful servers and fuzzing tools. Moreover, we illustrate the measures we take to eliminate or reduce the influence of randomness to optimize the reproducibility of the experiments. The randomness analysis is given in Section 5.4. (4) We extend Leif to support inter-parameter dependency inference. The design and implementation are given in Section 4.5. (5) We additionally study an important research question (i.e., RQ-4: How much unnecessary fuzzing time can inter-parameter dependency inference help Leif save?) and add an experiment in Section 5.6. (6) We analyze existing studies on inter-API dependencies. Furthermore, we find that most studies detect inter-API dependencies by analyzing the inputs and outputs of APIs. However, they only support primitive types. With the help of FET techniques, we can identify more detailed FETs and naturally match the FETs by matching their minimum acceptance. A clear idea is given in Section 7.1. (7) We discuss another question (i.e., RQ-4: How much unnecessary fuzzing time can inter-parameter dependency inference help Leif save?) and add an experiment in Section 5.6. (8) We also revise the paper in many places, including a detailed description (e.g., detailed descriptions and running examples for the algorithms and detailed fuzzing rules and fuzzing value examples for different FETs), improved figures, and in-depth background for our design and implementation.

The remainder of the paper is organized as follows. Section 2 analyzes the type collapse problem in detail. Section 3 introduces FET techniques to solve the type collapse problem. Section 4 introduces Leif as a proof-of-concept implementation of FET techniques. Section 5 presents the evaluation of FET techniques and Leif. Section 6 discusses related works. Section 7 discusses potential future work and Section 8 concludes.

2 Motivation

Automated REST testing tools generate test cases by filling parameters with automatically generated values. This procedure requires adequate information about parameters. Otherwise, the possible candidate space would become enormous even for one single parameter. Therefore, a majority of state-of-the-art automated testing tools focus on reducing the candidate space by sophisticated methodologies. For instance, RESTler [6] arranges multiple APIs in the producer-consumer order and uses response data gained from the previous APIs to request the next. Chizpurfle [7] and EvoMaster [8] generate optimal candidate values based on evolutionary algorithms.

Nevertheless, the previous works have not focused on the root cause of the candidate space explosion. Since most RESTful APIs are designed for exchanging data between programs implemented by different languages (e.g., Java for mobile applications while Python for the service), only a few common primitive data types can be used to represent API parameters. For example, Amazon’s online shopping web service takes about 2,400 parameters, among which 748 are number-typed (31%) and 1,581 are string-typed (66%) [3]. Besides, the previous works such as RESTier [6], EvoMaster [8], and TnT-Fuzzer [9] parse API specifications such as OpenAPI [10] to test RESTful web services. However, OpenAPI only defines 3 data types [11] (including integer, number, and string). That is, types, which are supposed to be diversified, now collapse into very limited cases in previous studies. Consequently, existing automated testing tools encounter a huge candidate space, e.g., solely knowing a parameter is string-typed spans a nearly infinite candidate space from paragraphs of Shakespeare to specific date-time strings. In addition, it is difficult to pick up effective values that can pass parameter validation, then reach actual business logic, and finally trigger bugs. Fig. 1 shows a code sample of a RESTful API (requires four parameters: string-typed start, string-typed end,
number-typed amount, and number-typed interest). In order to generate an effective value that can reach business logic for the parameter start, a testing tool has to know it is an ISO8601 datetime string. Unfortunately, since parameters are mainly in primitive data types, this information is usually hard to obtain. Therefore, the testing tool may treat it as an ordinary string and generate arbitrary strings that are rejected by the parameter validation and thus useless.

The type collapse problem is the major obstacle to obtaining adequate parameter information and leads to inefficient automated testing. Therefore, our solution provides a fine-grained description method for parameters by exploiting both its data type and value format. Leveraged by such information, we can bootstrap and enhance automated testing techniques to improve efficiency when testing RESTful web services.

3 FET TECHNIQUES

We introduce FET techniques to address the type collapse problem, including the FET (Format-encoded Type), the FET lattice, and the FET inference. A FET models an API parameter’s data type and value format. A FET lattice hierarchically organizes a set of FETs based on a partial order. We design FET inference algorithms to seek suitable FETs among a FET lattice for parameters, and the inferred results are critical for bootstrapping test case generation strategies.

3.1 Type Lattice

The idea of the FET lattice is inspired by the type lattice [12] for programming languages widely used in compilation and program analysis [13], [14], [15]. A type lattice is a complete lattice defined on \((T, \sqsubseteq)\), where \(T\) is a set of data types (e.g., long in C/C++) and \(\sqsubseteq\) is a partial order representing type convertibility. Every two lattice elements have a unique least upper bound and a unique greatest lower bound. An element \(t_i\) is said to cover another element \(t_j\) if and only if \(t_i \sqsubseteq t_j\) but there does not exist a \(t_m\) such that \(t_i \sqsubset t_m \sqsubset t_j\), where \(t_i \sqsubset t_j\) means \(t_i \subseteq t_j\) and \(t_i \neq t_j\). Type lattices can model class inheritance hierarchies for object-oriented languages. In this context, for any two elements \(t_i, t_j\), \(t_i \sqsubseteq t_j\) holds if and only if \(t_i\) inherits from or equals to \(t_j\). Fig. 2 depicts a type lattice for java.util.Collection (each vertex represents a class or an interface, and each directed edge stands for the inheritance relationship).

For example, in Fig. 2, the least upper bound of HashSet and TreeSet is AbstractSet. And the greatest lower bound of them is NoType. Similarly, the least upper bound of HashSet and AbstractSet is AbstractSet. And the greatest lower bound of them is HashSet.

3.2 FET Lattice

A FET lattice is defined on \((\Psi \subseteq T \times F, \triangleleft)\). A FET \(\psi \in \Psi\) is defined by \((t_\psi, f_\psi)\), where \(t_\psi \in T\) is a data type, and \(f_\psi \in F\) is a value format or more specifically a set of values. \(\triangleleft\) is a partial order that for any two FETs \(\psi, \psi'\), \(\psi, \psi' \triangleleft \psi''\) holds if and only if \(t_\psi\) is type-convertible to \(t_\psi'\) and \(f_\psi\) is a subset of \(f_\psi'\), denoted by \(t_\psi \triangleleft t_\psi'\) and \(f_\psi \subseteq f_\psi'\). A FET \(\psi\) covered by \(\psi_i\) implies that \(\psi\) describes parameter features in a finer grain than \(\psi_i\) and \(\psi\) are defined as \((\text{AnyType}, U)\) and \((\text{NoType}, \emptyset)\), where \(U\) is the set containing arbitrary values. Fig. 3 depicts an example FET lattice (a FET’s name describes its value format, and FETs at the same level are identically colored). In Fig. 3, each vertex represents a FET, and the edge from one FET to another refers to covering relation. The Hash FET, for example, denotes a string-typed hexadecimal number with more than 16 digits, while the MD5 FET represents 32-digit-hex, string-typed parameters. The data type of MD5 (i.e., string) is type-convertible to that of Hash (i.e., string). Moreover, the value format of MD5 (i.e., a hexadecimal number with 32 digits) is a subset of the value format of Hash (i.e., a hexadecimal number with more than 16 digits). As a result, \(\psi_{\text{MD5}} \triangleleft \psi_{\text{Hash}}\) holds.

**FET Acceptance for Parameter Values.** Like type lattices, FET lattices help determine FETs for given parameter values. To achieve this, we define that a value \(v\) is accepted by a FET \(\psi\) if and only if \(\text{typeof}(v) \sqsubseteq t_\psi\) and \(v \in f_\psi\), denoted by \(\psi \in \text{acceptance}(v)\). Otherwise \(v\) is said to be rejected by \(\psi\), denoted by \(\psi \notin \text{acceptance}(v)\). Spontaneously, \(\psi_i\) accepts all values while \(\psi_i\) accepts none. A value \(v\) can be accepted by more than one FET, while the greatest lower bound of the acceptances describes the value in the finest grain. We call such an acceptance the minimum acceptance of \(v\). The

![Fig. 2. A type lattice for the java collections framework.](image-url)

![Fig. 3. An example FET lattice.](image-url)
predecessors of the minimum acceptance accept v but describe it in a coarser grain, while the siblings reject v but describe other similar values in the same grain. The minimum acceptance, the predecessors, and the siblings of v compose a tree, denoted by ψ-tree(v). For example, for a SHA1 string v, its minimum acceptance (the SHA1 FET in Fig. 3), the predecessors (Hash, String, and ψ+) and the siblings (MD5, and SHA256) compose the ψ-tree(v).

Avoiding the Ambiguity of FET Lattices. As seen in Fig. 3, if a single value is accepted by two sibling FETs (e.g., MD5 and SHA1), the minimum acceptance will fall into the trivial ψ±. Generally, a FET lattice is said to be ambiguous if there exist two FETs with the same predecessor that can both accept the same value. Therefore, a validation procedure is obligatory after a FET lattice is constructed, which ensures the value formats of every two sibling FETs with the same data type are always disjoint.

In practice, we specify value formats by the regular language and provide a ubiquitous FET lattice [18] to model the most common RESTful parameters. The construction and verification of the FET lattice will be elaborated in Section 4.2.

3.3 FET Inference

Tree-Merging FET Inference. As discussed previously, for a single value v, a unique ψ-tree(v) can always be found in an unambiguous FET lattice. A RESTful API parameter usually involves multiple values in practice. Hence we give the tree-merging FET inference. For a parameter with values v1,...,vn, the tree-merging inference is to compute ψ-tree(v1),...,ψ-tree(vn), and then merge them into one tree. The merged tree is denoted by ψ-tree"n(Vn) where \( V_n = \{v_1,...,v_n\} \). The tree-merging inference can be described as a “find-expand-merge” procedure: (1) find the minimum acceptance for a single value v1 by performing a depth-first searching from ψ+, and add the predecessors along the searching path into the tree; (2) expand the tree by adding the siblings, and then the ψ-tree(v1) is obtained; (3) repeat the step (1) and (2) for every value and merge all the trees. Step (1) and (2) are illustrated in Fig. 4, and step (3) can be reduced to the DNS tree merging [19]. Assuming that the FET lattice has l levels with m FETs, the time complexity is \( O(m) \) for computing one tree and \( O(l) \) for merging two trees. Thus the time complexity of tree-merging FET inference for a parameter involving n values is \( O(n \cdot (m + l)) \).

Bitfield-Boosting FET Inference. In practice, we notice that the number of FETs m in a lattice is a constant while the number of values n is a variate (usually over 1,000). Therefore, we optimize the tree-merging FET inference based on three observations: (1) each FET can be uniquely represented by one bit in an m-bit bitfield, and therefore ψ-trees can be represented by several bits in such bitfields; (2) given a minimum acceptance, its ψ-tree can be uniquely determined, so the ψ-tree for every FET can be computed before inference; (3) merging two ψ-trees is equivalent to performing a bitwise OR operation on their corresponding bitfields.

Hence, we give the forward computation algorithm and the bitfield-boosting FET inference. The forward computation traverses the lattice in a breadth-first order, assigns a unique bitfield ID per FET, and computes the ψ-tree, as shown in Algorithm 1.

Algorithm 1. The Forward Computation

Input: A FET Lattice.
1: \( ID \leftarrow 1; \) queue \( \leftarrow \) Queue(ψ+);
2: while !queue.isEmpty() do
3: \( current \leftarrow queue.pop(); \)
4: \( current.ID \leftarrow ID; \)
5: \( ID \leftarrow ID < < 1; \)
6: for each \( \psi < current \) AND \( \psi \neq \psi_\perp \) do
7: \( queue.push(\psi); \)
8: \( \psi_.pTree \leftarrow 0; \psi_.sTree \leftarrow \psi_.ID; \)
9: \( \psi_.tree \leftarrow \psi_.pTree \lor \psi_.sTree; \)
10: \( queue.push(\psi); \)
11: while !queue.isEmpty() do
12: \( current \leftarrow queue.pop(); \)
13: \( sTree \leftarrow 0; \)
14: for each \( \psi < current \) AND \( \psi \neq \psi_\perp \) do
15: \( sTree \leftarrow sTree \lor \psi.ID; \)
16: \( \psi_.pTree \leftarrow current.pTree \lor current.ID; \)
17: \( \psi.sTree \leftarrow sTree; \)
18: \( \psi_.tree \leftarrow \psi_.pTree \lor sTree; \)
19: \( queue.push(\psi); \)

As an example, consider the FET lattice in Fig. 3. Line 2-7 of Algorithm 1 assigns an ID to each FET in a breadth-first order. There are 21 FETs in the lattice. Thus, each ID is represented by one bit in a 21-bit bitfield (augmented to a 6-digit hexadecimal number). As a result, \( \psi_.ID = (000001)_{16} \), \( \text{Null.ID} = (000002)_{16} \), \( \text{String.ID} = (000004)_{16} \) and so on. After that, Line 11-20 traverses the lattice and computes the ψ-tree for each FET. Specifically, Line 14-15 visits the current FET’s children and adds them to sTree. Given \( \psi_+ \) as current, sTree is assigned \( (000002)_{16} \). Furthermore, Line 16-20 revisits these children and computes the ψ-tree of each child. So we have \( \text{Null.tree} = \text{String.tree} = \text{Number.tree} = (00000F)_{16} \). Other FETs’ ψ-tree is computed in the same way.

Leveraged by the forward computation, the bitfield-boosting inference only needs to find the minimum acceptance by the depth-first searching, yields the bitfield \( \psi_.tree \), and merges it into the \( \psi_.tree^{l-1}(V_{l-1}) \), as shown in Algorithm 2.

Algorithm 2. The Bitfield-Boosting FET Inference

Input: Parameter Values: \( V_n = \{v_1,...,v_n\} \).
Output: \( \psi_.tree^{n}(V_n) \).
1: \( \psi_.tree^{0}(V_0) \leftarrow 0; \)
2: for \( i = 1 \) to \( n \) do
3: \( current \leftarrow \psi_+; \)
4: \( accepted \leftarrow true; \)
5: while accepted do
6: \( accepted \leftarrow false; \)
7: for each \( \psi < current \) do
8: if \( \psi \notin acceptance(v_i) \) then
9: \( current \leftarrow \psi; \)
10: \( accepted \leftarrow true; \)
11: \( \psi_.tree^{l}(V_i) \leftarrow \psi_.tree^{l-1}(V_{l-1}) \lor current.tree; \)
12: return \( \psi_.tree^{n}(V_n) \);
In Algorithm 2, Line 2-11 traverses each collected value for a certain parameter in an API. Line 5-10 traverses the FET lattice in a depth-first order to find the minimum acceptance for the value. For instance, `getDeviceInfo` API contains a parameter `deviceId`, with two possible values: 158980244481 (number-typed) and "f81fa248969650e54f6b291325445b3b" (string-typed), denoted as `v1` and `v2` respectively. For `v1`, with `c` as current, `accepted` is assigned true because the `ψT` FET accepts all values. The algorithm then goes to the `Null` and `String` FETs, with `accepted` assigned false because neither of them accepts `v1`. Next, the `Number` FET is visited, which assigns true to `accepted`. Following that, the `Decimal` FET and `Boolean` FET are visited, both of which reject `v1`. Therefore, the found minimum acceptance for `v1` is `Integer`, and the corresponding `c-tree` is `ð000039Þ16` (computed from Algorithm 1).

Similarly, the minimum acceptance for `v2` is `MD5`, and the corresponding `c-tree` is `ð070045Þ16`. Finally, the merged `c-tree` is `ð07007DÞ16`.

Therefore, the `ψ-tree(v)` can be efficiently computed by a series of bitwise OR operations instead of graph computations, reducing the time complexity from $O(n / C_1(m + l))$ to $O(n \cdot m)$.

4 FET-ENHANCED REST FUZZING

To manifest the utility of FET techniques, we design Leif, a FET-enhanced REST fuzzing tool, and implement it to a command-line tool in 2,796 lines of Python code. This section elaborates the workflow of Leif, along with methodologies for collecting HTTP traffic (Section 4.1), for constructing FET lattices (Section 4.2), for interfacing FET techniques with fuzzers (Section 4.3), and for parameter validation code generation (Section 4.4).

Fig. 5 depicts Leif’s workflow and its interaction with existing systems and tools. Leif assumes that the web service under test is already deployed on a staging server or in a production environment. The developer acquires the Leif program with a built-in FET lattice and traces HTTP traffic between the service and the clients. Then Leif identifies RESTful APIs by parsing the captured traffic and performs FET inference on parameter values. The inferred results are provided for a trace-driven FET-aware fuzzer and a parameter validator. Finally, Leif emits test cases and observes wrongful behaviors of the service. At the same time, the parameter validator can automatically generate parameter validation code and embed the code in RESTful client/server code to help fix potential bugs of the RESTful service under test.

4.1 Collecting and Parsing HTTP Traffic

As introduced in Section 3.3, the inferred result of a parameter is contributed by its different values. Therefore the accuracy of FET inference increases when Leif witnesses more value cases. Thus developers are expected to apply suitable tracing methods. Monkey testing and scripted regression testing, for example, are recommended over unit testing for traffic collection because they focus on real-world application scenarios, resulting in real-world HTTP requests. Leif takes the HAR file (an archival format for HTTP traffic [20]), which is the standard output of network proxies (Fiddler [21], MitmProxy [22], etc.), and browser inspection...
(e.g., Chrome and Safari). For parameter identification, the payload (including the headers, the query string, and the body) of a captured request is parsed to key-value pairs in JSON format. Due to the type collapse problem, only four data types are present: boolean, number, string and object (including array). Non-object-typed parameters are directly provided to FET inference while object-typed parameters are flattened. Since a JSON object is a tree of properties, Leif flattens it by splitting leaf properties to independent non-object-typed parameters and assigning new keys named by their JSONPaths [23], as illustrated in Fig. 6. Then the flattened parameters are also provided to FET inference. Finally, FET inference receives parameters for each API, where each parameter has a unique key and usually multiple values.

4.2 Ubiquitous FET Lattice

Regular Expressions for Value Formats. In Leif’s built-in ubiquitous FET lattice, value formats are specified by regular expressions. We choose to use the regular language rather than creating a new language to define value formats because it has many advantages in this scenario. First, regular expressions are the de-facto descriptions of most string formats. Although regular expressions are context-free, they can still distinguish different value formats. Second, they are already familiar to developers, and therefore they are easy to construct without extra learning costs. Finally, to ensure the unambiguity of a FET lattice is to ensure the regular expression orthogonality of sibling FETs, which can be formally determined by finite automata [24].

FET Lattice Constructing and Updating. We construct the ubiquitous FET lattice by referencing popular RESTful services (e.g., Google Maps, AWS, Twitter, and GitHub):

1) We crawl API documents from these services (including Google Maps, AWS Batch, Twitter, and GitHub), a total of 1268 APIs. Next, we identify potential FETs used in these services. These services contain representative APIs that are complex and frequently used, thus providing abundant information for constructing representative FETs. The crawled APIs are publicly available [25].

2) We construct regular expressions for these FETs by referencing related RFCs (e.g., RFC3339 [26] for ISO8601, and RFC3986 [27] for URI), programming language specifications (e.g., the Java specification [28] for PackageName), and database schema definitions (e.g., the MongoDB data type definition [29] for Hash) to build a base FET lattice which is suitable for generic web services.

3) We apply the Bayesian regular expression generation technique [30] to discover new FETs from traffic and merge them into the base lattice. The Bayesian regular expression generation takes a list of strings (e.g., obtained from HTTP traffic) as inputs and outputs inferred regular expressions that match all the input strings. This approach interprets regular expressions as probabilistic regular grammars. Next, starting with an initial grammar, it generates different candidate regular expressions by enumerating each string and randomly generating new rules or reusing existing rules that produce this string. Each regular expression is scored according to its length (favoring shorter regular expressions) and ability to explain input strings (favoring regular expressions that are more likely to generate input strings). Finally, top-k regular expressions are outputted. Step 3) discovers some unique, in-company FETs (e.g., Base64-encoded arrays in some eBanking services) that cannot be found in step 2).

4) We verify the unambiguity by checking the orthogonality of regular expressions for sibling FETs, using dk.brics.automaton [31]. This library is a DFA/NFA (finite-state automata) implementation and supports some regular expression operations, such as concatenation, union, Kleene star, intersection, complement, etc. Furthermore, we enumerate every two sibling FETs in the FET lattice and use this library to compute the intersection of their regular expressions. If the result is empty, then the two FETs are unambiguous.

The verified lattice has 21 FETs organized in 5 levels, and we believe it is competent to model most of the RESTful services. Once the FET lattice is constructed, it is relatively fixed and does not need to be updated frequently. Suppose a developer has application-specific FETs at the first usage or when major service updates take place. In that case, one can update the lattice by adding FETs via step 3) and repeat step 4) for unambiguity verification.

Twining FET Inference. We notice that some parameters can be represented by multiple data types and are minimally accepted by distinct FETs in different data types. For example, an epoch datetime (elapsed seconds or milliseconds since 1970-01-01 00:00:00) is accepted by the EpochString FET when it is represented by string. However, the epoch datetime is accepted by the Integer...

Fig. 6. An example of object flattening.

(a) The Original Parameter.

(b) The Tree Structure.

(c) The Flattening Result.
FET when it is represented by number. In our experiments, we find many cases where the data type of the same parameter varies in different HTTP traffic (e.g., string and number). To ensure that the inferred \textit{c-tree} fully describes the parameter, we implement the twinning FET inference. Before a value is inferred, Leif generates its twinning value if possible. If the original value is number-typed, Leif generates a twinning string-typed value (e.g., "$1589809244481" \rightarrow "1589809244481") and vice versa ("1589809244481" \rightarrow "$1589809244481"). Then both values are inferred, and the resulting two \textit{psi-trees} are merged as if Leif witnesses two independent values. By doing so, both the \textit{DateTime} and the Integer FETs are included in the final \textit{psi-tree} of an epoch \textit{DateTime} parameter.

### 4.3 FET-Aware Trace-Driven Fuzzing

Trace-driven fuzzing tools generate test cases by replacing parameter values of captured requests with candidate values. Therefore the success of a fuzzer mainly depends on the quality of candidate values. In conventional tools, using a larger candidate dictionary is the basic strategy to increase the opportunity for triggering bugs, yet it lengthens the fuzzing time.

On the contrary, Leif provides a small but targeted dictionary for each FET and we give several examples (corresponding to Fig. 3): Number is tried with integer overflows (8-bit, 16-bit, 32-bit, and 64-bit overflows) with signed and unsigned values; \textit{DateTime} is tried with year overflows (year 2038, and year 10000), invalid dates (e.g., 2019-2-29), and timezone tweaks; \textit{ISO8601} is tried with omitting meta characters ("-", ":", etc.); \textit{URI} is tried with malformed URLs (e.g., doubling "/", stripping "protocol://", and unescaped characters). The dictionary and representative fuzzing value examples are listed in Table 1.

With each parameter tagged by a \textit{psi-tree}, Leif generates test cases by exhausting dictionaries of all the FETs in the tree. Notice that, as discussed in Section 3.2, the predecessors and the siblings of the minimum acceptance describe similar but usually invalid values. Therefore, candidates

#### Table 1: Leif's Fuzzing Rules and Fuzzing Value Examples for Different FETs

| FET       | Fuzzing Rules                                    | Fuzzing Value Examples                                      |
|-----------|--------------------------------------------------|-------------------------------------------------------------|
| Null      | Try language-specific null value formats.        | `null,None,NULL,undefined,nil`                               |
| String    | Fill strings with control characters and SQL; use plain and empty strings. | "\xe0\xd8\xd9\xd3\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xde\xdf\xda\xdd\xde\xdf\xda\xdd\xde\xdf\xda\xdd\xde\xdf\xda\xdd\xde\xdf\xda\xdd\xde\xdf\xda\xdd\xde\xdf\xda\xdd\xde\xdf\xda\xdd\xde\xdf\xda\xdd\xde\xdf\xda\xdd\xde\xdf\xda\xdd\xde\xdf\xda\xdd\xde\xdf\xda\xdd\xde\xdf\xda\xdd\xde\xdf\xda\xdd\xde\xdf\xda\xdd\xde\xdf\xda\xdd\xde\xdf\xda\xdd\xde\xdf\xda\xdd\xde\xdf\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xd8\xd9\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xd8\xd9\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xd8\xd9\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xd8\xd9\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\xdd\xde\xdf\xda\xDD\xda\ddd
from these FETs are the most likely values to pass parameter validation and trigger bugs. Leif exhausts dictionaries for one parameter each time for an API with multiple parameters and tests such API by iterations of exhaustion. In this way, Leif increases the opportunity to trigger bugs and saves fuzzing time. Note that Leif also generates test cases that change the primitive data type of the original parameter (for example, “integer” for “string”).

For instance, iQiyi (a popular online video streaming application from Baidu Inc.) has a private API called GET /book/register that includes three parameters named appVer, soVer, and srcPlatform respectively. appVer is string-typed version number, and the collected values are “10.10.0”, “10.9.0”, and so on. For simplicity, we pick “10.10.0” as an example. According to the FET inference technique presented in Section 3.3, the found minimum acceptance is Version-Tag, and the binary representation of the inferred \( \psi\)-tree \( V_i \) is \((0001045)_{16}\). In Fig. 7, the \( \psi\)-tree \( V_i \) is depicted.

### 4.4 Parameter Validation Code Generation

Popular specification languages have little support in value formats. For example, although the Swagger specification [32] defines 6 data types (including string, number, integer, boolean, array, and object) and provides an optional format (e.g., int64, uri, date-time etc.) keyword which just serves as a hint for the value format of a specific data type, the Swagger specification does not require the value formats to be validated.

However, FET techniques are helpful for parameter validation, and Leif provides an HTTP parameter validator. Once the minimum acceptance of a certain parameter is detected, the validator can automatically generate parameter validation code and embed the code in the RESTful client/server code to help fix potential bugs of the RESTful service. For a RESTful server, the validation will be executed before the business logic. For a client, the validation will be executed before the request is sent. Sample code and their detailed analysis are in Section 5.5.

### 4.5 Inter-Parameter Dependency

For a RESTful API, the presence of inter-parameter dependencies means that the value or the presence of a parameter depends on other parameters, i.e., only some specific combinations of dependent parameters can form valid requests of this API. Inter-parameter dependencies constrain that a group of parameters ought to satisfy presence relation (including “ Requires”, “Or”, “OnlyOne”, “AllOrNone”, and “ZeroOrOne” dependencies), arithmetic relation or other complex relation which combines two or more types of dependencies [2].

For example, for two parameters, the presence relation “ Requires” means that the presence of parameter \( p_1 \) requires the presence of parameter \( p_2 \). As an example, in the Twitter API POST /tweets, when creating a tweet with polls, the presence of parameter poll.duration_minutes requires the presence of parameter poll.options.

However, for trace-driven fuzzers (e.g., Leif), it is difficult to infer the presence relation. For example, in an E-commerce App, premium users have an extra “ discount” parameter for the purchasing API. Premium users should give this parameter in the HTTP requests while non-premium users do not. However, such Apps are usually tested with a non-premium user. Thus, the “discount” parameter is invisible in the traffic. Consequently, all trace-driven fuzzers have nowhere to infer the presence relation. Therefore, we only discuss arithmetic relations because they are explicit.

A survey [2] on 40 commercial RETSful web services with more than 2.5K APIs shows that 85% APIs involve inter-parameter dependencies. Among all dependencies, arithmetic relation is common and important. For example, in the Airbnb API, searching vacation rentals requires a checkin parameter and a checkout parameter. Obviously, checkout must be later than checkin (denoted as checkin < checkout). According to a survey [2], arithmetic relations appear in 50% APIs. Booking services can have checkin and checkout dates. Twitter APIs can have start_time and end_time used for search queries, such as counting tweets. GitHub APIs can have since and before parameters used for statistical purposes, such as listing repositories created over some time. Therefore, we define APIs with two parameters representing an upper and lower boundary, respectively, as range-query APIs. Range-query APIs are widely involved in parameter FETs, such as Datetime, Integer, and VersionTag.

Another example regards the GitHub commit diffing API. It requires two SHA1-FET parameters (i.e., commit hash), base and compare, and returns the changes between these two commits. The inter-parameter dependency of this API is base != compare.

Therefore, the arithmetic relations we consider include GTB (> =), LTB (< =), EQ (= =), and NE (! = =). Moreover, these relations require that the two parameters are comparable. We define two parameters as comparable when they share the same FET. In addition, we define that date FETs (i.e., ISO8601, EpochString, and DateOnly) are comparable because they have similar semantic meaning (i.e., representing dates and times).

Arithmetic relations are usually guaranteed by checking the submitted parameter values on both the client-side and server-side. For example, when a user searches vacation rentals on Airbnb, there is an arithmetic relation where checkin < checkout. This check is done on the client-side, where users cannot select an earlier check-out date than the selected check-in date. This check is also done on the server-side. If the server receives an invalid request, it returns a 200 response where the response data shows the search results with unlimited check-in and check-out dates.
Another example is when GitHub diffs two commits, users cannot select two identical commits in the GitHub application. On the server-side, if the server receives a request where base == compare, it returns a 200 response with an error message in the body and no actual response data (implicit error). Leif prefers explicit 4XX or 5XX responses for errors, so test cases where the dependency is broken are considered useless. Therefore, we can infer arithmetic relations from the applications’ traffic and avoid generating useless test cases (i.e., test cases that break the inferred relations), thus reducing fuzzing time.

Algorithm 3. The Inter-Parameter Dependency Inference

**Input:** Values of Parameter $a$: $V_a = \{v_1, \ldots, v_n\}$; Values of Parameter $b$: $W_b = \{w_1, \ldots, w_m\}$. The Minimum Acceptance of $a$ and $b$: $m_1$, $m_2$.

**Output:** The Infered Dependency between Parameter $a$ and $b$.

1: if $m_1! = m_2$ then
2: if $m_1, isDateFET()$ then
3: return null;
4: if $m_1, isRangeComparable()$ then
5: $lastValidation = v_1, isLessThan(w_i)$;
6: else $lastValidation = v_1, isEqualTo(w_i)$;
7: for $i \leftarrow 2$ to $n$ do
8: if $m_1, isRangeComparable()$ then
9: $validation = v_i, isLessThan(w_i)$;
10: else $validation = v_i, isEqualTo(w_i)$;
11: if $validation! = lastValidation$ then
12: return null;
13: $lastValidation = validation$;
14: if $m_1, isRangeComparable()$ then
15: if $lastValidation$ then return $*<=* else return $*>=*$;
16: else
17: if $lastValidation$ then return $*==* else return $*!=*/

Based on these observations, we introduce FET-enhanced inter-parameter dependency inference. We only consider dependencies between two parameters because they account for 86% of all the dependencies. For an API with more than two parameters, we suppose there is a dependency if the two parameters’ FETs are comparable. Besides, the level of the parameters’ FET ought to be as low as possible because FETs at lower levels describe parameter features in a finer grain than FETs at higher levels. Thus, their relation is more straightforward.

For example, it is straightforward to suppose there is a “LTE or GTE relation” between two ISO8601 parameters since they represent dates and times. It is also straightforward to suppose there is a “EQ or NE relation” between two UUID parameters since they are usually used as identifiers. To be specific, for the FET lattice in Fig. 3, we infer the inter-parameter dependencies of ISO8601, EpochString, DateOnly, PackageName, URI, UUID, VersionTag, MD5, SHA256, SHA1, Integer, Boolean, and Decimal.

We enumerate every pair of parameters in a specific API and validate whether they have a certain dependency. For range-comparable FETs (i.e., ISO8601, EpochString, DateOnly, VersionTag, Integer, and Decimal), we validate whether they have “LTE or GTE relations”. For other FETs, we validate whether they have “EQ or NE relations”. After that, the value pair for these two parameters in each request is enumerated to validate whether the dependency holds, as is shown in Algorithm 3.

In Algorithm 3, the isDateFET() method checks whether the FET is one of Date, ISO8601, EpochString, or DateOnly. We define two parameters as comparable when they share the same Date-FET. Hence, Line 1-3 checks whether the minimum acceptance of parameter $a$ is different from that of parameter $b$ and whether parameter $a$ is a Date-FET. If the conditions are met, the algorithm returns null (Line 3), indicating no inter-parameter dependency between the two input parameters. The isRangeComparable() method checks whether the values of this FET can be compared by greater-than or less-than signs. Moreover, it can be defined by users. For example, our implementation returns true when the FET is one of Date, ISO8601, EpochString, DateOnly, VersionTag, Number, Decimal, or Boolean, and returns false for other FETs. If the parameters are range-comparable, we infer $<=$ or $=>$ relations. Otherwise, we infer $==$ or $!= relation. Hence, Line 4-13 traverses the values of parameter $a$ and $b$, and checks whether each value pair of $a$ and $b$ satisfies $<=$ or $=>$ relation. If the inferred relation of a certain value pair does not match the previous result (e.g., $v_i \lessdot= w_i$ while $v_{i+1} > w_{i+1}$), the algorithm returns null (Line 12). Else, Line 14-19 returns the inferred relation between the two input parameters.

For instance, for a hotel-booking API, parameter $a$ and $b$ are checkin and checkout respectively. There are 3 captured HTTP requests, and the values for checkin and checkout are (“2022-03-20”, “2022-03-20”), (“2022-04-20”, “2022-04-22”), (“2022-05-21”, “2022-05-25”) respectively. Hence, in Algorithm 3, $V_i = \{"2022 - 03 - 20", "2022 - 04 - 20", "2022 - 05 - 22"\}$ and $W_3 = \{"2022 - 03 - 20", "2022 - 04 - 22", "2022 - 05 - 25"\}$. The minimum acceptance of parameter checkin and checkout is the DateOnly FET (computed from Algorithm 1). Therefore, we have $m_1 = m_2 = "DateOnly"$. Each value pair satisfies the $<=$ relation (i.e., “2022-03-20”$<=$ “2022-03-20”, “2022-04-20”$<=$ “2022-04-22”, and “2022-05-22”$<=$ “2022-05-25”). As a result, the algorithm returns $<=$ (Line 15), indicating that there is a $<=$ relation between parameter checkin and checkout.

5 Evaluation

In this section, we evaluate Leif with real-world RESTful web services, and the complete dataset of our evaluation is publicly available [3]. Specifically, we design four experiments to answer the following research questions:

RQ-1 How accurately do FET inference results describe RESTful API parameters of complicated real-world web services?

RQ-2 How many new bugs does Leif find? What are their categories and causes?

RQ-4 How does Leif compare with existing state-of-the-art trace-driven and specification-driven fuzz testing
tools in terms of bug-finding capability and fuzzing time?

**RQ-4** How much fuzzing time can inter-parameter dependency inference help Leif save?

We answer RQ-1 using 50 randomly-picked RESTful APIs from GitHub and Twitter. By studying RQ-1, we will understand whether FET inference is practical in describing real-world RESTful API parameters. As for RQ-2, we apply Leif to 27 popular mobile applications to evaluate the bug-finding capability of Leif. Each selected mobile application:

- has over 1M downloads;
- has a positive rating of 3.0/5.0 at least;
- has 6.7MB bytecode size on average.

Such mobile applications indicate that the corresponding web services are complicated enough to serve many users. Then the found bugs are analyzed to help understand their common patterns and potential causes.

Furthermore, in RQ-3, we compare Leif with other state-of-the-art fuzzers (BurpSuite [33], Fuzzapi [34], Restler [6], and TnT-Fuzzer [9]) to investigate whether Leif has the better bug-finding capability and reduces fuzzing time.

To answer RQ-4, we compare two versions of Leif, one with the inter-parameter dependency inference and the other without, to explore how the inter-parameter dependency inference helps Leif save fuzzing time.

### 5.1 FET Inference Accuracy Evaluation

In this experiment, we assume that API documents provided by the service developers are the *ground truth* and we validate the accuracy of FET inference by comparing the inferred results with the ground truth. We choose GitHub and Twitter, and we randomly pick up 50 RESTful APIs (25 from each). The APIs we chose are not included in the APIs we used to build the FET lattice (Section 4.2). Therefore, the two API sets for creating the FET lattice and evaluating FET inference accuracy are separate. We extract two pieces of information from document text: (1) parameter data types, as explicitly listed in the documents; (2) parameter value formats, as provided in the detailed descriptions (e.g., “This [the parameter since] is a timestamp in ISO8601 format.”). We feed example requests gained from the documents to FET inference, compare the inferred FETs with the ground truth, and observe three levels of matching:

1) **exact match**, the inferred FET is said to be an exact match if it has the same data type and the value format as the ground truth;
2) **partial match**, the inferred FET is said to be a partial match if it has the exact data type, but its value format is a proper superset of the ground truth;
3) **mismatch**, for the remaining cases.

Intuitively, an exact match is the best-case scenario as it precisely describes a parameter such that a fuzzer can exploit it to generate the most targeted values. A partial match includes values that will not appear in practice, and a fuzzer may generate a small set of useless values based on a partial match, which will lead to more fuzzing time. However, a partial match is still quite good because:

- It has the same data type as the ground truth.
- Its value format is a proper superset of the ground truth, which guarantees that the generated values contain the most targeted values. To put it simply, if an exact match helps find a certain bug, so does a partial match.

Overall, partial matches are equal to exact matches in bug-finding capability, but it requires more fuzzing time. A mismatch indicates that the current FET lattice does not support the value format yet.

Fig. 8a exhibits the ratios of matching on GitHub (137 parameters), Twitter (86 parameters) and the weighted average (223 parameters). In total, 149 (67%) inferred results are exact matches, and 71 (32%) are partial matches. And we observe 3 mismatches in two cases: one is a binary-array parameter for file uploading and the other is an array of

---

1. GitHub REST API, [https://docs.github.com/en/rest/reference](https://docs.github.com/en/rest/reference)
2. Twitter Developer Platform, [https://developer.twitter.com/en/docs](https://developer.twitter.com/en/docs)
3. Gists, [https://docs.github.com/en/rest/reference/gists](https://docs.github.com/en/rest/reference/gists)
key-value pairs (e.g., 
["key1", "value1"], 
["key2", "value2"], ...). Binary arrays can be supported by simply adding a FET ([01]* for the value format) to the current lattice, but Leif aims to detect logic-related bugs. In contrast, binaries are usually logic-free but content-sensitive [35]. In other words, a FET tries to describe a parameter in aspects of both its data type and value format. For example, the string "2021-07-23" contains information that implies it is a timestamp in ISO8601 format. Thus, the FET can be constructed. However, if a binary-array parameter (e.g., a binary file stream) is regarded as a string that consists of "0"s and "1"s, the value format of the string is hard to be discovered without pre-given information, such as models or specifications. In addition, current binary fuzzing techniques [35], [36], [37] also require information (e.g., grammar, source code, etc.) that describes the value formats of binary parameters. Unfortunately, in the context of FET techniques, the value format of binary parameters can be obtained from neither API traffic nor specifications. Therefore Leif does not mutate them. As for key-value pairs, they are two-dimensional arrays where the first dimension is immutable since it indicates the actual parameter key. We consider allowing developers to specify which special parameters are immutable in Leif’s future version to support such cases. For the partial matches, we review the documents, and the top cases are application-specified formats, such as comma-separated strings and PGP signatures. These formats are less common, and developers can add application-specific FETs to their lattices by following the steps introduced in Section 4.2. Fig. 8b exhibits the breakdown of exact matches (the inner ring is the distribution of the primitive data types and the outer ring is the inferred FETs) to quantify how FET inference improves parameter information. The coarse-grained number-typed (27%) and string-typed (61%) parameters are divided into much smaller slices (5% - 14%). The breakdown clarifies that FET inference classifies parameters in balance, and therefore restores the collapsed types. This enables a fuzzer to generate more targeted values, which shrinks candidate space and increases the opportunity to find bugs.

Answer to RQ-1: The FET inference describes real-world RESTful API parameters well for the results consist of 67% exact matches, 32% partial matches, and only 1% mismatches.

5.2 Leif Effectiveness Evaluation

In this experiment, we select 27 popular mobile applications to evaluate the effectiveness of Leif. Each of them is backed by a commercial RESTful web service serving millions and billions of users. We monkey-test [38] each application for 20 minutes, capture HTTP traffic and run the full-stack Leif workflow. Table 2 lists the subjects and the services have an average of 133 RESTful APIs with over 19 parameters per API. We collect 46 requests per API on average, which yields adequate request samples for inference. Leif reports 5XX HTTP responses as bugs along with the corresponding traffic. We have reached out to the service owners, reported these bugs, and validated these bugs through analysis of...
traffic (through API URLs, parameter key-value pairs, and response data) and analysis of the involved applications (through reverse engineering and static code analysis of APKs) to eliminate any false-positive or duplicated cases. Table 3 summarizes the 11 distinct bugs found by Leif. The testing process is fully automated, which mimics how developers would use Leif as a black-box fuzzing tool in practice and our following analysis mimics how to classify bugs and locate related code lines based on Leif’s testing results.

**Security Bugs With Information Leakage.** Bug 1, 2, and 10 are security bugs with information leakage problems. They can be reproduced by mutating the parameter appVer (VersionTag), the parameter platform (Identifier), and the parameter c.v (Integer) respectively. These bugs not only cause service crashes but also expose sensitive information to end-users (potential attackers). With the exposed information, attackers can easily design specialized attacks. For example, the response data of bug 10 contains the full Java exception stack trace without any obfuscation. From the stack trace, attackers can obtain the service uses an outdated Spring Framework version which suffers from numerous security vulnerabilities [39], [40], [41], [42], [43], [44]. By exploiting CVE-2020-5421 and CVE-2020-5398 [39], [40], attackers can initiate reflected file download attacks [45] to mislead users into downloading malware. Moreover, by exploiting CVE-2018-1257 [44], attackers can expose STOMP over WebSocket and then initiate denial of service attacks [46]. They can also obtain that the service exposes STOMP over WebSocket and then initiate denial of service attacks [46]. They can also obtain the service exposes STOMP over WebSocket and then initiate denial of service attacks [46]. They can also obtain the service exposes STOMP over WebSocket and then initiate denial of service attacks [46].

**Answer to RQ-2:** Leif finds 11 distinct new bugs in 27 real-world RESTful web services. We observe three major categories of these bugs: security bugs with information leakage, third-party API bugs, and bugs with limited information. The ignorance of sensitive information and the use of vulnerable third-party APIs are their prominent causes.

**Third-party APIs are very common.** However, the bugs of these APIs are often overlooked during developing and testing. Therefore, bugs in third-party code are as important as the application code because they both fail application functionality to billions of end-users. Our results show that Leif can find bugs in third-party APIs. Consequently, we suggest that developers capture application traffic and apply Leif to test untrusted third-party APIs. In addition, developers should design proper exception handling logic for third-party code and timely upgrade to the latest API versions with known bugs fixed. Finally, developers of third-party APIs should spend more time on testing to find more potential bugs.

**5.3 Comparative Evaluation**

**Leif versus Trace-Driven Fuzzers.** We classify Leif as a trace-driven fuzzer and compare it with state-of-the-art trace-driven fuzzing tools. We select BurpSuite [33], a commercial security testing fuzzer for RESTful web services, and Fuzzapi [34], an open-source general-purpose HTTP fuzzer. They provide built-in candidate dictionaries but require a series of manual configurations, including filling the URL for each API and the data type for each parameter. Therefore, we only apply them to Sina News, Toutiao, and Amazon Shopping (518 unique APIs with 15,512 parameters in total). In addition, we implement NaiveFuzzer as a baseline that only understands primitive data types and randomly mutates parameter values solely based on such coarse-grained information. We construct NaiveFuzzer’s candidate dictionaries by combining the dictionaries of BurpSuite and Fuzzapi.

We evaluate the bug-finding capabilities of BurpSuite, Fuzzapi, Leif, and NaiveFuzzer by comparing the number of bugs found by each tool, as reported in Fig. 9a. Besides, we evaluate their fuzzing time by comparing the averaged number of test cases generated per parameter, as exhibited in Fig. 9b. Less generated test cases mean less test execution time, leading to more efficient fuzzing. Note that we use the averaged number of test cases generated per parameter as a metric rather than “the total test case generation time” because: (1) the total test case generation time is too small compared with the execution time; (2) it is unfair to compare different tools’ test case generation time because some are automatic while others require manual configuration. Considering the

---

4. Spring Framework, https://spring.io/projects/spring-framework
5. Fastjson, https://github.com/alibaba/fastjson
subjects are already well-tested before release, we believe the bug-finding capability of Leif is better than BurpSuite and Fuzzapi, for Leif finds extra bugs. Additionally, NaiveFuzzer has the same capability as BurpSuite and Fuzzapi. This is because they share the same candidate space. As for fuzzing time, BurpSuite, Fuzzapi, and NaiveFuzzer generate 5.0/6.7×, 3.6×−4.7×, and 6.3×−7.1× test cases of Leif, respectively, indicating FET techniques bring 72%−86% fuzzing time reduction.

Leif versus. Specification-driven Fuzzers. Generally, trace-driven fuzzers such as Leif need GUI interaction to obtain traffic and generate test cases based on the observed traffic (i.e., the quality of generated test cases depends on observed traffic). On the contrary, for specification-driven fuzzers, GUI interaction is not required, but high-quality specifications (i.e., complete and regularly maintained specifications) are required. Hence, in practice, trace-driven and specification-driven approaches are complementary.

We now compare Leif with existing specification-driven fuzzers, which test RESTful web services based on parsing API specifications. We select RESTler [6], a state-of-the-art research fuzzer, and TnT-Fuzzer [9], an open-source robustness testing tool. They both require OpenAPI specifications [10] as input, but most of the subject services do not provide OpenAPI specifications. Therefore we construct OpenAPI specifications for Sina News, Toutiao, and Amazon Shopping by parsing HTTP traffic and referencing their official API documents.

We intend to run RESTler, but unfortunately, neither the executable program nor the source code is available. According to the paper, RESTler only supports primitive data types and uses a plain candidate dictionary (consisting of 0, 1, ””, and ”sampleString”). Nevertheless, none of the bugs found by Leif can be triggered by these values, indicating that performing RESTler would fail to detect any of the bugs. Regarding TnT-Fuzzer, it generates candidate values simply based on the Python random() function (i.e., purely random fuzzing). We configure it to generate 1,000 test cases per parameter (about 5× of NaiveFuzzer and 30× of Leif). Still, TnT-Fuzzer fails to find any bugs in the three

### TABLE 3
Bugs Found by Leif in the Effectiveness Validation Experiment

| Bug ID | Involved Application | Status Code | API Path | Description |
|--------|---------------------|-------------|----------|-------------|
| 1      | iQiyi               | 500         | /book/register | A private API, served for user registration. |
| 2      | Pinduoduo          | 500         | /cappuccino/splash | A private API, served for first-screen advertising. |
| 3      | Sina News           | 500         | /oauth2/getaid.json | A deprecated public API provided by Sina Weibo, served for user authorization. |
| 4      | Sina News           | 503         | /oauth2/getaid.json | A deprecated public API provided by Sina Weibo, served for user authorization. |
| 5      | Smzdm               | 502         | /integration.php  | A public API provided by Baidu, served for inter-application integration. |
| 6      | Sohu News           | 502         | /sendacc.jsp      | A public API provided by 53KF, served for customer service. |
| 7      | Sohu News           | 502         | /sendacc.jsp      | A public API provided by 53KF, served for customer service. |
| 8      | Toutiao             | 502         | /user/tab/tabs/v3 | A private API, probably served for inter-application redirecting. |
| 9      | Toutiao             | 504         | /user/tab/tabs/v3 | A private API, probably served for inter-application redirecting. |
| 10     | Tuniu               | 500         | /vip/recommend   | A private API, served for content recommendation. |
| 11b    | WUBA                | 502         | /integration.php  | A public API provided by Baidu, served for inter-application integration. |

- **Bug 3 and bug 4 involve the same API but with different HTTP status codes.**
- **Bug 5 and bug 11 involve the same API but different applications.**
- **Bug 6 and bug 7 involve the same API path but different domain names.**

Fig. 9. Bug-finding capabilities and fuzzing time of the evaluated fuzzers.
services. We conclude that the two fuzzers’ effectiveness is limited by the practical hardness of finding well-written OpenAPI specifications and the quality of their candidates. These are also the main shortcomings of all specification-driven fuzzers. Besides, many modern APIs require short-lived session tokens for access control or throttling. Specification-driven fuzzers require manual configuration or even repeated re-configuration for such parameters. In contrast, it is easy for trace-driven fuzzers to achieve this requirement by mutating freshly captured requests.

**Answer to RQ-3:** Leif finds more bugs and reduces 72%–86% fuzzing time compared with state-of-the-art fuzzers.

### 5.4 Randomness Analysis

Leif’s effectiveness evaluation (Section 5.2) and comparative evaluation (Section 5.3) evaluate Leif and other state-of-the-art fuzzing tools’ bug-finding capabilities and fuzzing time. Both experiments have some randomness that lies in:

- **RESTful servers**
- **Fuzzing tools**

Randomness leads to the experiments’ inaccuracy and low level of reproducibility. So in this section, we analyze these two types of randomness and give the steps we take to improve the experiments’ reproducibility.

**Randomness in RESTful Servers.** The randomness of RESTful servers consists of A/B testing and critical server failures. A/B testing (also known as bucket testing or split-run testing) compares two versions of a single variable (e.g., two versions of an algorithm, two types of page layouts, etc.), which is wildly used in web services. By comparing users’ responses to version A versus version B, developers can determine which of the two is more effective. If a specific API is under A/B testing, sending the same request to this API may lead to different responses, which may make the fuzzing tools miss some bugs. For example, suppose two versions of API A, A–α and A–β, are under A/B testing. And a specific request R triggers a bug that is hidden in A–α but not in A–β. If a fuzzing tool sends request R only once and receives an A–β response, the bug hidden in version A–α is never discovered.

As the responses of different versions are randomly assigned by the server, an effective way to reduce the influence of A/B testing is to send the same HTTP request multiple times [49]. Therefore, we use Fiddler [21] to capture the fuzzed traffic when using these fuzzing tools and then reissue the captured requests. Taking execution time into account, we reissue the captured requests for 10 times.

Exceptions are common and uncaught exceptions can cause a server program to fail. Long periods of server program downtime will result in unexpected or unpredictable responses. For instance, if a certain request crashes a server, all the clients (including fuzzers) who send requests may receive 4XX/5XX responses before the server recovers. This leads to randomness because fuzzers cannot know whether the server can process the requests properly.

To tackle this problem, the 27 applications we select are popular commercial mobile applications (listed in Table 2). And each one of them is powered by a large-scale, reliable RESTful web service which is equipped with fault tolerance techniques, such as Docker Swarm or Kubernetes, to guarantee the server’s good accessibility.

**Randomness in Fuzzing Tools.** Random number generation methods (e.g., the `random.randint()` function in Python) mainly cause the randomness in fuzzing tools. For example, TnT-Fuzzer [9] randomly selects a value from a candidate dictionary and randomly replaces a certain parameter in a given API. If the value that triggers a certain bug is never selected, TnT-Fuzzer will never find this bug. However, if TnT-Fuzzer is lucky enough to pick a bug-triggering value at the very beginning, TnT-Fuzzer will find this bug in a short time. Hence, the bug-finding capability and fuzzing time of TnT-Fuzzer depend heavily on the randomly generated number.

In practice, randomness is widespread in fuzzing tools, which will affect the experiments’ reproducibility. Therefore, we modify these fuzzers to generate a deterministic random number sequence. For open source fuzzers, such as Leif, NaiveFuzzer, TnT-Fuzzer, and Fuzzapi, we set a fixed random seed before generating a random number. Furthermore, for closed source BurpSuite, we preload a Java class to set a fixed random seed.

### 5.5 Generated Parameter Validation Code

In this experiment, we showcase some code snippets generated by Leif to show that FET techniques can be used to generate parameter validation code. We take the `POST /integration.php` API (concerning Bug 5 and 11 in Table 3) as examples. The automatically generated JavaScript server code for `POST /integration.php` is listed in Fig. 10.

As shown in Fig. 10, Leif generates a function (Line 27–34) that validates the API’s parameters’ according to a json schema [50]. The corresponding json schema is defined in Line 7–24 which validates the parameters (including “app”, “SdkVer”, “appversion”, “mianliu”, “request_url”, and “lastModified”). The detected FET of each parameter determines the patterns in the JSON schema. Then, the validator is used before the original business logic (Line 39) so that the parameter validation does not interrupt the execution of the original program.

Similarly, the automatically generated Python client code for `POST /integration.php` is listed in Fig. 11.

As shown in Fig. 11, all the parameters in the `POST /integration.php` API are validated before the request is sent. Therefore, the request will not be sent unless all the parameters are valid.

The code generation capability of the FET techniques can also help enhance standard RESTful specifications and existing specification-based code generators. For example, as discussed in Section 4.4, the Swagger specification provides little support in value formats. Moreover, the code generated by Swagger Codegen [51] (a code generator that generates server and client code for any API defined with the Swagger specification) has no parameter validation. Therefore, we can modify the Swagger Codegen to embed
the parameter validation code generated by Leif in the position before the business logic is executed (for servers) or before the HTTP request is sent (for clients).

To conclude, Leif can generate parameter validation code and embed the code in real-world RESTful services to help fix bugs of the services. Furthermore, FET techniques can enhance existing code generators, such as Swagger Codegen, to generation parameter validation code for different programming languages.

### 5.6 Inter-Parameter Dependency Inference Evaluation

Leif infers arithmetic relations, and prunes test cases that do not satisfy the inferred relations, to avoid wasting fuzzing time. Among the detected 11 bugs in Table 3, 2 bugs (Bug 5 and 11) contain arithmetic relations in the related APIs, and we validate that the pruned test cases do not trigger bugs. The other 9 bugs contain no arithmetic relation in the bug-triggering parameters, therefore no test cases are pruned.

To study how much fuzzing time inter-parameter dependency inference can save, we implement two versions of Leif. One is equipped with inter-parameter dependencies analysis (namely Leif-i), and the other is not (namely Leif). We then run Leif and Leif-i on the same captured traffic to generate test cases. For Leif-i, 1,352 dependencies involving 814 APIs are found in 20 out of 27 services. Moreover, we compute reduced fuzzing time by comparing the number of generated test cases by Leif and Leif-i. Some services have no pruned test cases (i.e., test cases that Leif generated but Leif-i did not) because:

- No arithmetic relation is found in this service;
- There are some inferred arithmetic relations, and all test cases generated by Leif satisfy these relations. For example, in an Amazon’s private API /generate_did (probably served for generating an ID used for advertising purposes based on the device’s information), there is a NE relation between parameter sha1_mac and sha1_serial. As Leif only tries boundary values or mixed-case values for SHA1 FETs, sha1_mac and sha1_serial are seldom equal when Leif fuzzes these two parameters.

For services that have pruned test cases, Leif generates 949 test cases, while Leif-i generates 809 test cases (15% reduction on average). Among them, iQiyi gets the highest reduction of 21.2%, and Mafengwo gets the lowest of 8.2%. Moreover, we validate that the pruned test cases do not trigger 5XX responses.

We also notice some interesting findings when observing the reduced test cases:

- No arithmetic relation is found in this service;
- There are some inferred arithmetic relations, and all test cases generated by Leif satisfy these relations. For example, in an Amazon’s private API /generate_did (probably served for generating an ID used for advertising purposes based on the device’s information), there is a NE relation between parameter sha1_mac and sha1_serial. As Leif only tries boundary values or mixed-case values for SHA1 FETs, sha1_mac and sha1_serial are seldom equal when Leif fuzzes these two parameters.

For services that have pruned test cases, Leif generates 949 test cases, while Leif-i generates 809 test cases (15% reduction on average). Among them, iQiyi gets the highest reduction of 21.2%, and Mafengwo gets the lowest of 8.2%. Moreover, we validate that the pruned test cases do not trigger 5XX responses.

We also notice some interesting findings when observing the reduced test cases:
• Some of the EQ/NE dependencies have parameters with similar names. For example, Hash-FET parameters, `device_id` and `device_id`, in a private API which probably served for user settings updating in Toutiao.

• Most LTE/GTE dependencies we found are related to chronological relations. For example, Some of Qunar’s APIs have DateOnly-FET parameters named `startTime` and `endTime`.

Answer to RQ-4: The Inter-parameter dependency inference help Leif reduce an average of 15% fuzzing time.

6 RELATED WORK

Model-Driven Testing. Model-driven testing [52], [53], [54], [55], [56] is usually white-box and requires using a certain specific modeling method (e.g., UML or DSL) through the whole lifecycle of developing, which is human-intensive and technically limited for services across multiple servers and micro-services from different vendors. Essentially, FET techniques are also model-driven (i.e., driven by a lattice-based model) but only intervene in the test phase. Thus FET techniques can be practically employed to test diversified RESTful web services in black-box approaches.

Trace-Driven Fuzzing. Trace-driven fuzzing generates test cases by mutating recorded requests. Fuzzapí [34], BurpSuite [33], AppSpider [57] and Leif all fall into this category. Existing trace-driven fuzzers mainly focus on improving the ability to capture and replay HTTP traffic. However, Leif demonstrates that FET techniques provide fundamental parameter information to fuzzers, bringing the enhanced bug-finding capability and significant fuzzing time reduction.

Specification-Driven Fuzzing. Another main class of fuzz testing techniques is specification-driven fuzzing, such as TnT-Fuzzzer [9], EvoMaster [8], and RESTler [6], which avoids the type collapse problem by assuming developers provide well-defined specifications with detailed parameter information. However, the OpenAPI [10] is the only well-established standard up to now, but it is not widely used. A survey [58] reveals that 71% developers lack the knowledge of the OpenAPI framework. Therefore, the specification-driven fuzzing is still too idealistic for testing real-world RESTful web services. In comparison, instead of asking developers for good specifications, FET techniques generate fine-grained specifications (i.e., ψ-trees of parameters) on their own.

Security Penetration Testing. Fuzz testing techniques are also commonly aimed for security penetration testing. Commercial security penetration tools, such as BurpSuite [33], use values of SQL injections, unescaped HTML characters, XML/JSON external entities, etc., to expose system bugs. FET techniques can also be employed in security penetration testing, as demonstrated in Section 5.2. However, our main goal is not limited to security testing for RESTful web services because FET techniques improve the value selecting strategy for general-purpose REST fuzzing.

Inter-Parameter Dependency Analysis. Currently, several efforts are aiming at the description for inter-parameter dependency in RESTful APIs. Some efforts focus on extending current specification languages. Oostvogels et al. [59] investigate and find 3 classes of inter-parameter dependencies and propose OAS-IP, a specification language focusing on defining and imposing constraints on parameters in RESTful APIs. Martin-Lopez et al. [60] present a DSL called Inter-parameter Dependency Language (IDL) which could describe 7 types of dependencies among parameters. Efforts like OAS-IP and IDL can be considered as extensions for current specification languages.

Some studies aim to infer inter-parameter dependencies based on analysis of current specifications. Xu et al. [61] propose an approach to analyze the OWL-S (Web Ontology Language for Services) and other OWL specifications to extract different dependencies to enable syntax, workflow, and semantic testing. Wu et al. [62] propose an approach called INDICATOR which can automatically infer inter-parameter dependencies of web services by analyzing the service documents, the service SDKs, and the web services themselves. Gao et al. [63] infer inter-parameter dependencies by integrating information of parameters, error messages, and testing results.

Another way to infer inter-parameter dependencies is based on analysis of testing results. For example, Gao et al. [63] combines testing results and API specifications to infer inter-parameter dependencies. By analyzing API traffic, FET techniques can infer the inter-parameter dependencies of a RESTful API.

7 FUTURE WORK

7.1 Inter-API Dependency

For different RESTful APIs, the presence of inter-API dependencies means that the calling of a RESTful API (or some RESTful APIs) depends on the calling of another RESTful API (or other RESTful APIs). For example, a resource included in the response of a certain request (i.e., the output of RESTful API A) is a required parameter in another request (i.e., the input of RESTful API B) [6].

For instance, the Twitter API “GET followers/ids” [64] accepts `user_id` (an Int64-typed parameter which represents the unique identifier for a certain user) or `screen_name` (a String-typed parameter which represents the screen name, handle, or alias that a certain user identifies themselves with) as input, and returns a cursored collection of user IDs for every user following the specified user.

Another Twitter API “GET users/lookup” [65] accepts `user_id` (an Int64-typed parameter which represents the unique identifier for a certain user, which is also in the output of “GET follower/ids”) as inputs, and returns fully-hydrated user objects (an object which contains Twitter User account metadata that describes the Twitter User referenced [66], such as `user_id`, description, `followers_count`) for the specified users.

Therefore, Twitter officially recommends that developers use “GET followers/ids” in conjunction with “GET users/lookup” to get the basic information of some user’s followers [64].

There are several studies focusing on dependencies among different APIs [6], [67], [68], [69]. Current research
aims to infer inter-API dependencies by analyzing or extending specifications. Bai et al. [67] define 3 types of inter-API dependencies: input dependency, output dependency, and input/output dependency. Furthermore, they propose an approach to infer inter-API dependency by analyzing WSDL (Web Service Description Language) specifications, thus generating valid test cases for RESTful APIs. Chaturvedi et al. [69] propose an approach to infer inter-parameter and inter-API dependencies by analyzing WSDL specifications and Web Service source code. Atlidakis et al. [6] propose a RESTful API fuzzer named RESTler, which captures inter-API dependency by analyzing value types of different requests and responses described in OpenAPI specifications.

However, the studies mentioned above only focus on analyzing primitive data types, blind to complex data types like FET. For example, it is arbitrary to assume a dependency between two RESTful APIs if one of them returns string-typed data (e.g., a URI string “www.google.com”) and the other consumes a string-typed parameter (e.g., an ISO8601 datetime string “2021-07-23”).

Consequently, for specification-driven fuzzing tools, such as RESTler, future efforts can make more precise inferences of inter-API dependencies by leveraging FET techniques to infer FETs of input parameters (or output data) of RESTful APIs, thus reducing fuzzing time.

For trace-driven fuzzing tools, the type of input parameters (or output data) can be inferred by analyzing API traffic, thus detecting inter-API dependencies. For example, as discussed in Section 3, for a single value \(v\), a unique \(\psi\)-tree \((v)\) can always be found in an unambiguous FET lattice. The \(\psi\)-tree \((v)\) describes the minimum acceptance of value \(v\) in detail. Therefore, the dependency of two APIs exists if the \(\psi\)-tree of one API’s input parameters match the \(\psi\)-tree of another API’s output data. As a result, Leif can generate test cases that satisfy inter-API dependencies, enhancing Leif’s bug-finding capability. Note that matching \(\psi\)-trees is not a sufficient and necessary condition for the equivalence of two FETs. Instead, it means the two FETs share the same minimum acceptance, i.e., the two FETs are sibling FETs, such as “MD5” and “SHA1”. As sibling FETs are similar, fuzzing among sibling FETs can bring more valid test cases, thus finding more potential bugs of APIs under test. Therefore, input parameter and output data only need to be sibling FETs rather than the same FET.

### 7.2 Updating the FET Lattice

The FET lattice constructed in Section 4 is relatively fixed. Only when major service updates occur or at first usage does the FET lattice need to be updated. Currently, if a developer has to update the FET lattice, one can update the lattice by adding FETs and verifying the FET lattice via steps 3) and 4) in Section 4.2. In the future, an automatic tool that parses RESTful API specifications and automatically detects the corresponding FET lattice is required.

### 8 Conclusion

In this paper, we analyze the type collapse problem and propose FET techniques to remedy this problem. As a proof-of-concept, we design and implement Leif, a FET-enhanced trace-driven fuzzing tool. We demonstrate that using FET techniques greatly improves a fuzzer’s understanding of parameters, resulting in more effective fuzz testing. Our experiment results show that Leif unveils 11 new bugs in application-specific web services as well as general third-party open API platforms with 72% – 86% fuzzing time reduction. The inter-parameter dependency inference, Leif saves 15% fuzzing time.

### Acknowledgments

The authors like to thank the anonymous reviewers for their valuable comments.

### References

[1] R. T. Fielding, “Architectural styles and the design of network-based software architectures,” The Representational State Transfer (REST) Ph.D. dissertation, Dept. Inf. Comput. Sci. Univ. California, Irvine, USA, 2000, pp. 76–85. [Online]. Available: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[2] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortes, “A catalogue of inter-parameter dependencies in RESTful web APIs,” in Proc. 17th Int. Conf. Service-Oriented Comput., 2019, pp. 399–414, doi: 10.1007/978–3-030-37202-5_31

[3] Y. Chen, Y. Yang, Z. Lei, M. Xia, and Z. Qi, “The public dataset of leif evaluation,” figshare, Jan. 2021, doi: 10.6084/m9.figshare.12377150.v1

[4] R. Chandrashekar, M. Mardithaya, S. Thilagam, and D. Saha, “SQL injection attack mechanisms and prevention techniques,” in Proc. Int. Conf. Adv. Comput. Netw. Secur., 2011, pp. 524–533

[5] Y. Chen, Y. Yang, Z. Lei, M. Xia, and Z. Qi, “Bootstrapping automated testing for restful web services,” Fundam. Approaches Softw. Eng., vol. 12649, 2021, Art. no. 46, doi: 10.1007/978–3–030–71500-7_3

[6] V. Atlidakis, P. Godfrey, and M. Polishchuk, “RESTler: Stateful REST API fuzzing,” in Proc. 41st Int. Conf. Softw. Eng., 2019, pp. 748–758, doi: 10.1109/ICSE.2019.00083

[7] D. Cotroneo, A. K. Iannillo, and R. Natella, “Evolutionary fuzzing of android OS vendor system services,” Empirical Softw. Eng., vol. 24, no. 6, pp. 3630–3658, 2019, doi: 10.1007/s10664–019-09725-6

[8] A. Arcuri, “RESTful API automated test case generation with EvoMaster,” ACM Trans. Softw. Eng. Methodol., vol. 28, no. 1, pp. 31–337, 2019, doi: 10.1145/3293455

[9] Trli-fuzzer,” 2018. [Online]. Available: https://github.com/Trebley/Trli-Fuzzer

[10] OAI (OpenAPI Initiative), “The OpenAPI specification,” 2021. [Online]. Available: https://github.com/OAI/OpenAPI-Specification

[11] OAI (OpenAPI Initiative), “The OpenAPI specification data types,” 2021. [Online]. Available: https://spec.openapis.org/oas/latest. html#data-types

[12] P. Cousot and R. Cousot, “Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in Proc. Conf. Rec. 4th ACM Symp. Program. Lang., 1977, pp. 238–252, doi: https://doi.org/10.1145/512950.512973

[13] S. H. Jensen, A. Møller, and P. Thiemann, “Type analysis for JavaScript,” in Proc. 16th Int. Symp. Statist. Anal., 2009, pp. 238–255, doi: 10.1007/978–3-642–84327–0_17

[14] V. Raichev, M. T. Vechev, and A. Krause, “Predicting program properties from “big code”,” in Proc. 42nd Annu. ACM SIGPLAN-SIGACT Symp. Program. Lang., 2015, pp. 111–124, doi: 10.1145/2676728.2677009

[15] D. Schuerer, R. Hähnle, and R. Bubel, “A general lattice model for merging symbolic execution branches,” in Proc. 18th Int. Conf. Formal Eng. Methods Formal Methods Softw. Eng., 2016, pp. 57–73, doi: 10.1007/978–3–319–47846-3_5

[16] A. Møller and M. I. Schwartzbach, “Static program analysis,” Dept. Comput. Sci., Aarhus Univ., Aarhus, Denmark, Oct. 2018. [Online]. Available: http://cs.au.dk/~amoe/liv/spa/

[17] J. Aycock, “A brief history of just-in-time,” ACM Comput. Surv., vol. 35, no. 2, pp. 97–113, 2003, doi: 10.1145/857076.857077

[18] Y. Chen, Y. Yang, Z. Lei, M. Xia, and Z. Qi, “The ubiquitous FET lattice model and verification,” figshare, Jan. 2021, doi: 10.6084/m9.figshare.13622720.v2
Yixiong Chen received the BEng degree from Southeast University in 2019. He is currently working toward the master’s degree in computer science with Shanghai Jiao Tong University. His research interests include focus on program analysis, automated testing, and data visualization.

Yang Yang received the BS degree in information engineering from Shanghai Jiao Tong University in 2017, the MS degree in software engineering from Shanghai Jiao Tong University, in 2020. She is currently working with Tencent as a software engineer. Her research interests include software testing and code coverage.

Mingyuan Xia received the PhD degree from McGill University in 2017. His research interests include focus on mobile systems, program analysis and distributed storage. He received the IBM PhD Fellowship in 2015.

Zhengwei Qi (Member, IEEE) received the BEng and MEng degrees from Northwestern Polytechnical University, in 1999 and 2002, and the PhD degree from Shanghai Jiao Tong University, in 2005. He is a professor with the School of Software, Shanghai Jiao Tong University. His research interests include program analysis, model checking, virtual machines, and distributed systems.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.