ProFIPy: Programmable Software Fault Injection as-a-Service

Domenico Cotroneo, Luigi De Simone, Pietro Liguori, Roberto Natella
Università degli Studi di Napoli Federico II, Italy
{cotroneo, luigi.desimone, pietro.liguori, roberto.natella}@unina.it

***Abstract*—In this paper, we present a new fault injection tool (*ProFIPy*) for Python software. The tool is designed to be *programmable*, in order to enable users to specify their software fault model, using a *domain-specific language* (DSL) for fault injection. Moreover, to achieve better usability, *ProFIPy* is provided as *software-as-a-service* and supports the user through the configuration of the faultload and workload, failure data analysis, and full automation of the experiments using container-based virtualization and parallelization.**

***Index Terms*—Software Fault Injection, Python, Software-as-a-Service, Bug Pattern**

I. Introduction

Fault injection is a key technique for assessing fault-tolerant systems, ranging from embedded and mobile systems [1] to distributed systems [2]. To perform a fault injection campaign, it is important to define a *fault model*, which describes the faults to be emulated in the experiments. The fault model entails the definition of three main aspects, namely *what* to inject (i.e., which kind of fault), *when* to inject (i.e., the timing of the injection), and *where* to inject (i.e., the part of the system targeted by the injection) [3]–[7]. The *what* can be represented by bit-flips [1]; program exceptions for amplifying unit- and integration-tests [8], [9]; node crashes, network partitions and latency for networked and distributed systems [2], [10]. The *when* and *where* to inject are sampled from a (large) space of possibilities across time and program locations.

The problem of defining a fault model becomes more difficult when injecting *software faults* (i.e., design and/or programming defects [11]), since they depend on a variety of technical and organizational factors, including the programming language, the software development process, the maturity of the system, the expertise of developers, and the application domain [12], [13]. Despite the variability of software faults across systems, the existing software fault injection tools are based on a predefined, fixed software fault model, that cannot be easily customized by users. Most of the existing tools adopt the *Orthogonal Defect Classification* (ODC), proposed in the '90s (e.g., bugs in initialization, algorithm, interfaces, etc.), or derived the fault model from bug samples of third-party open-source and commercial projects [3], [14].

We believe that a modern software fault injection tool has to be able to modify the fault model for the following reasons. First, a typical necessity in industry, which arises when a critical failure occurs, is to introduce regression tests against the fault that caused the failure, to assure that the same failure cannot occur again [15]. Second, to preserve the efficiency of the fault injection campaign, it is important to avoid injecting bugs that are unlikely to affect a system; e.g., some classes of faults may be prevented by testing and static analysis policies adopted by the company [16]. Third, as the scale and the complexity of systems increase, the need for a more sophisticated fault model grows. For instance, modern distributed systems, such as cloud applications, have to integrate a variety of components, including third-party and open-source ones, and they have to deal with high volumes of traffic. For these systems, the user needs to inject more variants of design/programming defects than those reported in the literature, including performance bottlenecks, resource management issues, lack of interoperability between components, security issues, failed updates, etc., and to adapt these faults to their projects. In general, the potential users of software fault injection want to tune the fault model so that it reflects their experience and expectations about failures. All these use cases require a greater degree of control over the fault model than what provided by existing fault injection tools.

In this paper, we present a new fault injection tool (*ProFIPy*) designed to be *programmable*, enabling users to add and to customize a software fault model. By using our tool, users can specify new software fault models using a *domain-specific language* (DSL) for fault injection. The tool compiles the specification into an automatically-generated fault injector. Finally, the generated fault injector is applied to the software-under-test to generate fault-injected versions and to execute experiments. To achieve better usability, *ProFIPy* is provided as *software-as-a-service*, and includes a workflow for configuring the faultload and the workload to i) fully automate the execution of experiments using container-based virtualization and parallelization, and to ii) perform failure data analysis. The tool has been designed for the popular Python language, which has recently arisen as one of the most widespread languages (e.g., among the GitHub and StackOverflow communities [17], [18]), and has found applications in several areas such as systems software (e.g., the OpenStack cloud platform is one of the largest projects in Python [19], [20]), enterprise and web applications and data science [21]. We present *ProFIPy* in the context of a Python project, by performing three fault injection campaigns in which we define three different faultloads.

In the following, Section II discusses related work; Section III presents a new domain-specific language; Section IV describes the workflow of the tool; Section V shows the application of *ProFIPy* on a Python project; Section VI concludes the paper.

II. Related Work

The idea of software fault modeling for fault injection purposes was initially investigated by Chillarege et al. [22], who analyzed a dataset of failures of IBM OS and DBMS products at users' sites [23], [24], to identify recurring patterns in the faults that caused them, and to inject the same patterns by corrupting program data and code, e.g., as in the *FINE* tool [25]. In the same period, they also introduced the *Orthogonal Defect Classification* (ODC) [26], [27], where one the goals was to classify software fault data into orthogonal categories, including *Initialization*, *Algorithm*, *Interface*, *Checking*, and *Synchronization* defects. Christmansson and Chillarege [3] proposed to inject software faults by following the statistical distribution of OS faults across these categories, such that the injected faults are representative of faults experienced by the users of the OS in the field. Similarly, Chen and colleagues [28], [29] defined a software fault model for OSes based on data for the IBM MVS and Tandem GUARDIAN90 OS products [23], [30], and used this fault model to emulate realistic OS and DBMS crashes, to assess crash recovery mechanisms. This fault model was later merged in the well-known fault injection tool of the *Nooks* project [31].

The work on the *G-SWFIT* fault injection technique by Madeira and colleagues [14], [32] aimed to define a *generic* software fault model (i.e., not tailored for a specific system) that could go beyond specific OS and DBMS products, and that could be used for injecting faults even without any field failure data for the specific system under testing. To define such a generic fault model, they analyzed a sample of bugs in several open-source projects in C [14], [32] and Java [33], [34], and looked for bug-fixes (e.g., program elements that were changed to fix the bug, such as new assignments, control flow constructs, function calls, etc.) which were recurring more than the norm, and which occurred consistently across all of the projects. Based on this analysis, they defined a software fault model with 13 fault types, covering 60% of the sample of bugs in the open-source projects [14]. This fault model was used in several other tools, including *SAFE* [35], *HSFI* [36], and *FastFI* [37]. However, these tools focus on a fixed software fault model, with no ability to customize the injected faults according to the specific needs of a project or company.

Winter et al. [38] and Giuffrida et al. [39] showed that implementing a new fault model in a tool takes both significant programming effort, e.g., in terms of SLOC and other metrics, and considerable expertise in program analysis and transformation, e.g., to implement a software fault injection tool using the *LLVM* compiler suite, which are not affordable for the average user of a fault injection tool.

Some tools provide a limited ability to customize the fault model with a lower effort: among them, the *FIDLFI* tool [40] provides the user with a configuration language to control the *trigger* of fault injection (i.e., instructions and paths that trigger the injection), *target* (i.e., instruction source and destination registers to inject), and *action* (e.g., corruption, freeze, delay, etc.). The *FAIL-FCI* tool [41] provides a fault injection language tailored for grid systems, which specifies protocol states and nodes to inject (e.g., node crashes). *PreFail* [2] and *FATE* [10], which inject crashes and I/O API errors, allow the user to write *policies* in Python to select the location and timing of potential injections by considering the allocation of processes across nodes and racks (e.g., network partitions between different racks), and the coverage of injectable points in the software-under-test. *LFI* [42], which injects errors at C library calls, allows the user to configure what functions and error codes should be injected, and when to trigger the injection (e.g., when a specific function appears in the stack frame) using an XML configuration file. The commercial tools *QA Systems Cantata* [43] and *Razorcat TESSY* [44] provide user-friendly GUIs to select a source-code statement to inject, similarly to breakpoints in a GUI debugger.

It is important to note that these tools do not support rich software fault models as in *G-SWFIT* and derivatives, as they only provide limited control on *what* to inject, e.g., they focus on API and library calls, register accesses, nodes, etc., but do not allow to create new fault types for injecting arbitrary changes to the software. The proposed *ProFIPy* tool provides a new language to gain a higher degree of control, where the user can specify transformation rules about which parts of the program to inject, in terms of program elements (e.g., assignments, expressions, control flow directives, and combinations of thereof), and how to transform these program elements into faulty ones.

III. Fault Injection Domain-Specific Language

The *ProFIPy* allows the user to enter a *bug specification* using a high-level and easy-to-use DSL language, which is close to the Python language. The bug specification describes how the source code of the program should be transformed to introduce a software bug. It consists of two parts:

- **Code pattern**: a description of which parts of the program should be fault-injected. The fault injection tool parses the source code of the software and will generate a fault for every match of the code pattern.
- **Code replacement**: a description of the code that should be injected, which will replace the original source code that matched the code pattern.

The code pattern describes a combination of program entities (variables, expressions, blocks, control flow constructs, etc.) that will be searched for in the software-under-injection. The code pattern can either consist of a Python snippet of code; or, it can be a mix of Python code and DSL directives. In the former case, *ProFIPy* will look for *exact* matches between the Python snippet in the code pattern and the Python code in the software-under-injection. In the latter case, the DSL directives will make the pattern to match several different variants of the Python snippet of code. Similarly, the code replacement can either be Python-only code, i.e., the injector will insert a fixed snippet of buggy code; or, it can contain a mix of Python and DSL directives, i.e., the injected buggy code can vary depending on what matched the code pattern.

Fig. 1 shows three examples of bug specifications. These specifications inject three fault types from G-SWFIT [14]: the omission of a function call (MFC); the omission of a small block of statements surrounded by an IF construct (MIFS); and a wrong parameter in input to a function call (WPF). Differing from the G-SWFIT technique, we modified the definition of the fault types, to point out the features of the DSL language, and to emulate more accurately some of the bugs that we found in the OpenStack project [45], [46].

```
change {
   $BLOCK{tag=b1; stmts=1,*}
   $CALL{name=delete_*}(...)
   $BLOCK{tag=b2; stmts=1,*}
} into {
   $BLOCK{tag=b1}
   $BLOCK{tag=b2}
}
```

(a) Missing function call fault (MFC).

```
change {
   if $EXPR{var=node} :
      $BLOCK{stmts=1,4}
      continue
} into {
}
```

(b) Missing IF construct with statements (MIFS) fault.

```
change {
   $CALL#c{name=utils.execute}(..., $STRING#s{val=*-*}, ...)
} into {
   $CALL#c(..., $CORRUPT($STRING#s), ...)
}
```

(c) Wrong parameter in function call (WPF) fault.

Fig. 1. Examples of fault specifications.

The MFC fault type from G-SWFIT looks for function calls in the software-under-injection, where there is no return value from the function call, or where the return value is ignored by the caller [14]. By targeting this kind of function calls, the injector can emulate a function call omission by removing these function call statements, and yet to obtain a syntactically-correct program, as the removal does not break any dependency with the rest of the program. Moreover, the G-SWFIT study [14] recommended that the function call should only be removed when the function call is not the only statement in its block, to better reflect the real bugs from open-source projects that were analyzed in that study.

In Fig. 1a, the code pattern (i.e., the *change* { . . . } part of the specification) looks for any function or method call, by using the `$CALL` directive of the DSL. The {`name=delete_*`} syntax after `$CALL` means that we are targeting calls where the function name starts with "*delete_*" string, in order to inject faults in calls to the OpenStack Neutron APIs `delete_port`, `delete_subnet`, `delete_network`, etc. This is an example of how a user may want to customize fault injection according to domain knowledge: these APIs are prone to omissions (e.g., the Neutron bug #1028174 [47]), and users may want to simulate these faults to assess solutions for resource leak detection. The rest of the specification implements the rules of the MFC fault type. `$CALL` only matches statements where the function or method call is the outermost part of the statement: thus, a statement like `x = mycall()`, where the assignment is the outermost expression, would not match the code pattern of Fig. 1a. The *(. . .)* syntax means that we are targeting function calls with any number of input parameters (zero, one, or more). The directives `$BLOCK` directives require that the function call must be both preceded and followed by one or more statements. Finally, the code replacement (i.e., the *into* { . . . } part of the specification) means that we want to transform the matched code by replacing it only with the blocks that precede and follow the function call. The {`tag=...`} syntax after `$BLOCK` allows the user to give a label (e.g., `b1`, `b2`) to the parts of the code pattern that matched the software-under-injection, and to reuse these parts in the code replacement.

In the second example (Fig. 1b), the MIFS fault type matches an IF construct with its statements (up to 4), and removes them, i.e., the code replacement part of the specification is empty. The specification mixes fragments of Python code (i.e., the `if` construct and `continue` keywords) and DSL directives (`$EXPR`, `$BLOCK`). Again, we refined the original fault type from G-SWFIT by leveraging domain knowledge, to inject into more specific targets. We emulate another recurring issue in OpenStack, in which metadata of resources (e.g., the `UUID` of instances) must have been initialized to allow operating on the resource, but a check on the validity of the metadata has been omitted (e.g., the Nova bug #1096722 [48]). To emulate this real bug, we target `if` constructs that check specific variables (e.g., variables called `node`, which are used throughout the OpenStack Nova codebase) and that skip an operation if the check fails (e.g., by issuing a `continue`).

In the third example (Fig. 1c), the WPF fault type injects an invalid parameter to a function call. The bug specification replaces a `$CALL` statement with the same `$CALL` statement, but modifying one of the input parameters. We use again a `tag` to reuse code from the code pattern in the code replacement, by means of the `#c` syntax after `$CALL`, i.e., the matched function call is labeled as "*c*". We tailored the bug specification to match another recurring issue in OpenStack, in which an external utility (e.g., `iptables`, `dnsmasq`, `e2fsck`) is invoked at the host OS level, but with incorrect or missing parameters (e.g., the Nova bug #732549 [49]). Thus, we target the `utils.execute()` library function (the `name` attribute in `$CALL`), and look for a string literal (`$STRING`) among the input parameters of the function, where the string contains the character used by UNIX utilities to denote parameters. In the code replacement, we inject the same function call, but the string literal (labeled *s*) is wrapped by a function call that corrupts the string with random contents, using the `$CORRUPT` DSL directive.

In addition to these examples, we have been using the DSL to define several fault models in an industrial context, in cooperation with Huawei Technologies Co. Ltd.. The DSL provided us a fine-grain control over the injections, by combining DSL directives with Python code fragments. Other fault types include: the injection of exceptions within `try` blocks, in order to increase the test coverage of error handlers [2], [42]; the injection of `None` values from library function calls, in order to test error handlers in which the returned value is checked by an `IF` construct after the call; the omission of optional input parameters to function calls; the omission of AND/OR clauses in `IF` conditions; wrong or missing initialization of data, such as key-value pair literals in Python dictionaries, using the `$CORRUPT` directive; high resource consumption (CPU, memory, storage), using the `$HOG` directive.

The DSL can be used to inject more complex fault types, by: using regular expressions for specifying search patterns; using the tagging syntax in the *change* block, to change the order of statements in the *into* block; mutating any arithmetic, boolean, and control flow expression of the Python grammar; injecting algorithmic bugs by removing entire portions of code (e.g., patterns with multiple nested loops and control flow constructs), and by injecting artificial time delays using a `$TIMEOUT` directive. More examples are presented in § V.

IV. The *ProFIPy* workflow

ProFIPy provides a complete fault injection workflow, which assists test engineers at applying software fault injection in Python systems. The *ProFIPy* workflow generates a set of mutated versions of the target software, according to user-defined bug specifications. These mutated versions are executed in a controlled environment, and further analyzed for drawing insights about the system behavior under failure. Fig. 2 summarizes the workflow, which consists in a sequence of three main phases, that is, *Scan* (see § IV-A), *Execution* (see § IV-B), and *Data Analysis* (see § IV-C and § IV-D). The following sub-sections provide details for each phase.

A. Scan

In the *Scan* phase, the user interacts with the *ProFIPy* tool to define the *fault injection plan*, which is the set of fault injection experiments to be run. Each experiment specifies a fault to be injected. *ProFIPy* takes in input the source code of the target software, and the bug specification described by using our DSL (section III). The fault model is stored in a JSON file, and users can save and import fault models of previous fault injection campaigns. *ProFIPy* provides pre-defined fault models based on previous fault injection studies (section II).

The *Scan* phase identifies **fault injection points** in the software, i.e., a statement (or group of statements) in the source code where *ProFIPy* can inject the software bug according to the user-defined specification. *ProFIPy* looks for arithmetic/boolean expressions, method and function calls, variable initializations, and other kinds of statements.

ProFIPy processes the target code using its Abstract Syntax Tree (AST) representation, which is commonly by program analyzers to represent the structure of a piece of code. The *DSL compiler* component takes the bug specification written using the DSL and generates a meta-model, which consists of a small AST that reflects the structure of the code in the code pattern. The meta-model will be used by the *source code scanner*, which visits the program's AST to find matches against the code pattern (i.e., portions of the program's AST that match the AST of the meta-model). The meta-model is also used by the *source code mutator* to generate fault-injected versions of the program (see § IV-B).

After obtaining a set of fault injection points, the user can select a subset of such locations according to their needs. For example, the user may want to perform experiments only for a specific component (e.g., class or file); the user may want to inject a sample of randomly-chosen faults (e.g., to enforce a limit on the number of experiments); or, the user can inject faults in all of the injection points. The set of injections defines the fault injection plan, which is used in the *Execution* phase.

In this paper, the proposed DSL is tailored for the Python language. It is possible to define a similar DSL to support other languages, such as C/C++ and Java. Several of the bug patterns for Python could be re-used (i.e., patterns not involving special Python syntax). The porting would mostly affect the DSL compiler and the source code scanner and mutator.

B. Execution

In this phase, *ProFIPy* iterates over the fault injection plan. In each experiment, the *original* Python source code is transformed into a *mutated* version, which is identical to the original except for a few mutated statements. The mutation emulates a residual bug in the software. For example, to inject a wrong parameter bug in a method call, *ProFIPy* modifies the method call statement by replacing it with a call to the same method but with different or corrupted input parameters; to emulate an omission by the developer, *ProFIPy* deletes the method call in the mutated version. The set of mutated versions are the *faultload* that will be executed in the experiments. At the end of every experiment, *ProFIPy* collects logs from the target system for data analysis (§ IV-C).

The user also configures a *workload*, i.e., a set of directives to exercise the target software during the experiments. The workload emulates the operating conditions of the system and triggers the injected fault. Moreover, the workload serves to detect service failures and recovery abilities, e.g., by looking for crashes and timeouts of the workload (e.g., due to stalled service calls), or by performing consistency checks with test assertions on the outputs of the workload (e.g., after a resource has been modified by the workload, the behavior of the system should reflect the new state of the resource).

The user defines the workload by providing command-line directives. For example, the user can use UNIX shell commands to start the target software, e.g., to launch a UNIX daemon such as a network server. Command-line directives can be used both to invoke the command-line interface of the target Python program or to indirectly launch the software by running automated test scripts. These scripts can be uploaded by the user along with the target Python source code (Fig. 2). Additionally, the user can specify command-line directives to launch workload generator tools, such as HTTP and RPC traffic generators, which in turn exercise the target software.

ProFIPy runs the fault injection experiments within a *container-based* experimental environment, by using the Docker virtualization system [50]. The tool first creates a container image, in which it copies the Python source code uploaded by the user. The user can customize the container image by adding configuration directives in *Dockerfile* format [51], such as, to install within the container external dependencies to run the Python software under test (e.g., using the *pip* command), and to install external tools (e.g., HTTP and RPC traffic generators). Then, for each fault to be injected, *ProFIPy* deploys a new container, by copying into it the mutated source code with

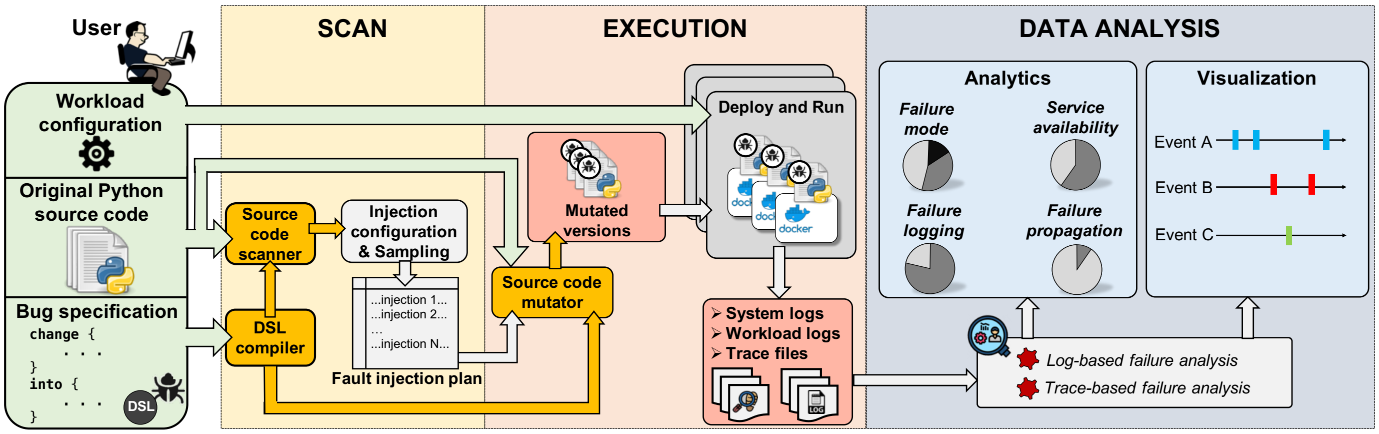

Fig. 2. Workflow of the *ProFIPy* tool.

the fault, and runs the workload directives defined by the user. The experiment ends when the workload completes, or when a user-defined timeout expires. Finally, *ProFIPy* cleans-up the experimental environment by deallocating the container. In this way, the tool can also clean-up any resource leaked or corrupted because of the injected fault (e.g., stale processes or files). Using containers also allows the tool to run several parallel experiments on independent sandboxes, to take advantage of multi-core CPUs. *ProFIPy* tunes the number of parallel experiments according to run at most $N - 1$ parallel containers at the same time, where N is the number CPU cores in the host system [52]. To avoid interferences in memory and I/O bandwidth, the tool further reduces the number of parallel containers if it hits a threshold for memory and I/O utilization.

ProFIPy can enable and disable the injected faulty code at any time during the execution of the target software. The mutated source code retains a copy of the original statements of the fault injection point, similarly to the EDFI fault injection tool [39]: *ProFIPy* mutates the source code by inserting an `IF ... ELSE ...` construct, where the two branches include respectively the original statements and the faulty ones. Then, the tool can control which of the two branches to execute, by writing a control variable (a "*trigger*") allocated in a shared memory area between the tool and the target software. This ability enables additional analyses of the effects of failures and recovery. The tool executes the workload for two times ("*rounds*"), without restarting the target program between the two executions. In the first round, the injected fault is enabled, so that it infects the target software with error states, possibly causing service failures. The workload is executed again in the second round, but the injected fault is disabled. Of course, if the target program fails and is unable to recover, the second workload execution will fail. The second round allows us to analyze the *scope* of the error states [53], [54]. In the best case, the error state is confined to service requests that were issued during the first round, and the requests during the second round are not affected by any error (e.g., the target software recovers a correct state with a restart). In the worst case, the error states are persistent even after that the faulty code is disabled, causing further failures during the second round. This analysis provides additional feedback to the user about the failure behavior of the target software.

During the experiments, *ProFIPy* saves the output of the target program (*stdout*, *stderr*) and the output of the workload directives (e.g., the commands for launching a workload generator, which reports service failures). Moreover, the tool can be configured to save log files that may be generated by the target software or by the workload. These outputs and logs are analyzed in the last phase of the *ProFIPy* workflow (*data analysis*), as discussed in the following.

C. Data Analysis

The *data analysis* evaluates the target software in terms of service failures, logging, and recovery. *ProFIPy* classifies the experiments into a set of **failure modes**, which include the crash and the timeout of the target software, and user-defined failure modes. The user can specify patterns (e.g., using keywords and regex) that the tool will look for among the outputs and the logs produced by the experiments. For example, failure modes can include failures of the workload (e.g., the workload stops due to a service API exception) and of the target software (e.g., the software detects an error state with an internal assertion, and reports it with a high-severity log message). The tool reports the statistical distribution of failure modes. The user can drill-down the individual classes of failures, to further inspect logs of experiments in that class. The user can also drill-down with respect to fault types and injected components, to identify the critical areas (e.g., components that are most prone to failures) where failure mitigations are most needed.

ProFIPy can analyze failures with respect to workload rounds. It computes a *service availability* metric, i.e., the percentage of experiments in which the software was (un)available in the second round of execution (injected fault disabled), because of error states generated during the first round (injection fault enabled) that persisted and were not recovered. These cases deserve a deeper analysis, e.g., to identify resource leaks that may occur in error handling paths, and that may cause more failures over time [55], [56].

D. Advanced Features

ProFIPy includes more, optional features for deeper analysis of the large amounts of data produced by fault injection experiments.

We briefly report here on these features.

■ **Coverage analysis**. To reduce the time needed to run the fault injection experiments, *ProFIPy* performs a preliminary analysis to avoid injecting faults in *program paths* that are not *covered* by the workload. Most likely, the workload will not cover all of the paths in the program, and injecting into non-covered paths causes a waste of time since the fault would not cause any effect. Before executing the experiments, *ProFIPy* conducts a *coverage analysis*, by running a "fault-free" execution (i.e., no fault injected) using the same workload that will be used for the experiments. It generates coverage information by adding logging statements at every fault injection point in the target program discovered by the *scan* phase (see § IV-A). After the fault-free run, *ProFIPy* generates a *reduced* fault injection plan, by only including the covered fault locations.

■ **Failure logging**. *ProFIPy* checks whether the target system can detect error states and report diagnostic information on *log files*. The tool computes a *failure logging* metric, i.e., the percentage of experiments in which the target software both experienced a workload failure and logged at least one error message. Failures and error logs are identified with user-provided keywords and regex. This metric gives feedback about the logging abilities, and non-logged failures are opportunities for improving telemetry. An example of this analysis can be found in a previous study [45].

■ **Failure propagation**. *ProFIPy* checks if the fault in the injected component propagated across other components. The tool computes a *failure propagation* metric, i.e., the percentage of injected faults that impacted on more than one component. This metric is applicable for larger software with a component-based architecture, where each sub-system generates a distinct log file, or where logs of the sub-systems can be separated with keywords and regex. The user configures a list of sub-systems, their source code files (e.g., a sub-folder of the source code), and their log files or patterns. The experiments that exhibit propagation are worth further investigation, e.g., to develop more robust interfaces between sub-systems to prevent the propagation and make recovery easier. Examples of this analysis can be found in previous papers [45], [57].

■ **Failure visualization**. *ProFIPy* provides a graphical representation of an experiment, to help the user to understand what happened during a failure. The tool instruments selected RPC APIs in the target software, and records their invocations during the experiment using the *Zipkin* distributed tracing framework [58]. These API calls are visualized as *events* on timelines as interactive plots. An example of visualization can be found in a previous study [59].

V. Case Study

We present an application of *ProFIPy* in the context of *Python-etcd* [60], which is a library that provides Python bindings for the *etcd* distributed key-value store [61]. Huawei uses *Python-etcd* in their systems and asked for three fault classes to be evaluated using our fault injection tool (Table I): (i) call failures when invoking APIs from external libraries (wrong response, timeouts, etc.), (ii) wrong inputs to the *Python-etcd* APIs, and (iii) resource management faults. We implemented these fault types using the *ProFIPy* DSL language.

TABLE I
Injected fault types.

Fault Category	Injection Target	Examples of Injections
Failures when calling external library APIs	API calls to the `urllib` and `os` Python modules	Exceptions, `None` objects, omitted call, wrong call
Wrong inputs in Python-etcd API	`set(key, val)`, `get(key)`, `test_and_set(key, val, old)`, ...	String corruptions, `None` values, negative integers
Resource management bugs	`set(key, val)`, `get(key)`, `test_and_set(key, val, old)`, ...	Hog threads inside methods of Python-etcd

We performed three fault injection campaigns on *Python-etcd* version 0.4.5. The workload used deploys the *etcd* server, and it uploads and queries several key-value pairs of a different kind (e.g., with directories, sub-keys, TTL, etc.) that we derived from *Python-etcd*'s integration tests. In the following subsections, we present the injected fault types and analyze failure modes using *ProFIPy*.

A. Errors from external APIs

In the first campaign of experiments, we injected faults at method calls in Python-etcd external modules, targeting the methods of `urllib` (a Python package for working with URLs) and from `os` (e.g., Python methods for file I/O). The injected fault types include:

- **Throw Exception**: The `raise` of the exception on a method call, according to pre-defined, per-API list of exceptions (e.g., `ConnectTimeoutError`);
- **Missing Function Call**: A method call is entirely omitted (e.g., replaced with the python statement `pass`);
- **Missing Parameters**: A method call is invoked with omitted parameters (e.g., the method uses a default parameter instead of the correct one).

For this faultload, *ProFIPy* identified 26 points where to inject faults. In 13 cases, the workload covered the injected faulty code. We found failures in 12 experiments.

▷ **Reconnection failure**. In half of the cases, we found failures in both rounds of execution, as denoted by the *service availability* metric. The experiments did not complete within the timeout, and `etcd` was unable to reconnect even after the fault removal. We found that the `etcd` server was unable to bind to a TCP/IP port. Thus, restarting `etcd` does not suffice to recover from the fault, but the port needs to be explicitly freed. We need additional exception handlers to catch exceptions caused by network connections, such as time-outs.

▷ **Critical errors about `'member has already been bootstrapped'`**. In a few experiments, Python-etcd was unable to perform operations on `etcd` in the first round, due to an inconsistent state of the server caused by the fault. To recover from this failure, the system needs a more elaborated exception handling: it should explicitly remove the affected member by using the dynamic configuration API of `etcd`, and it should restart `etcd` by reverting to a previous consistent state.

▷ **Client process crash due to an exception**. In the remaining cases, the client process crashed during the first round due to an unhandled exception. Moreover, the system was not available after disabling the fault. In these cases, Python-etcd should provide exception handlers to catch these exceptions or to raise another kind of exception (such as *EtcdException*) to be managed by Python-etcd client process.

B. Wrong Inputs

In the second campaign of fault injection experiments, we injected faults in input parameters of Python-etcd API methods. We configured *ProFIPy* with fault types for injecting corrupted inputs, such as strings with random characters, None object references, negative integers, etc. For example, let us consider the `method test_and_set(key, value, old_value)` taking in input three parameters: A fault consists in injecting a corrupted input in the first parameter (string type) by randomly replacing the characters of the string.

The *ProFIPy* tool identified 66 locations where to inject these faults. In all of the cases, the injected faulty code was covered by the workload, and in 29 experiments we found the following failures in the first round of execution:

▷ **AttributeError: 'NoneType' object has no attribute 'startswith'**. This failure is due to an issue of Python-etcd. It happens when the tool injects a `None` value instead of a string (e.g., a *key* string). Python-etcd does not check whether the input strings are valid. Therefore, when a `None` value is passed in input, Python-etcd uses the `startswith` attribute on a `None` reference. To avoid this failure, Python-etcd should sanitize null strings in inputs.

▷ **EtcdKeyNotFound exception**. This failure happens when a wrong key or value is injected. In this case, the workload failed because it is not able to find the expected key or value in the `etcd` datastore. The caller (in this case, the workload) needs to get/set the correct keys and values. Thus, the Python-etcd client should handle these exceptions.

▷ **EtcdException: Bad response: 400 Bad Request**. This failure happens when *ProFIPy* injects a wrong key or value that is not valid (e.g., a non-ASCII string). When this value is passed to `etcd`, the server rejects the request with the *HTTP Error 400 Bad Request*. Python-etcd should be fixed to check and sanitize non-ASCII strings.

C. Resource Management Bugs

In the last campaign of experiments, we injected CPU hogs to overload Python-etcd. We used *ProFIPy* for injecting stale threads that generate a high CPU load. We targeted the same methods of the second campaign of experiments, by injecting a resource hog after the method call. The tool found 37 injectable locations, and the faulty code was always covered during the workload execution. In 14 experiments, the system experienced a service failure in the first round of execution. Most of these failures forced a process termination with the exception *"UnboundLocalError: local variable ... referenced before assignment"*. In other cases, the workload also failed because of inconsistent values read from the `etcd` datastore. The high CPU usage triggered race conditions in Python-etcd, and in the Python interpreter itself. Since it is hard to find and to fix these issues, the failure should be mitigated, by cleaning-up stale threads that may cause high CPU consumption. This should be pursued by monitoring at run-time the CPU utilization of Python processes, and by killing or restarting stale threads if CPU utilization is too high.

D. Performance evaluation

ProFIPy can quickly inject faults even for large projects since the scan and mutation can be parallelized across several CPUs (it is an "embarrassingly parallel“ task). It took less than one minute to scan and mutate *Python-etcd* on an 8-core Intel Xeon with 16 GB RAM. We also evaluated performance on the OpenStack project, by targeting the three most important modules (Nova, Neutron, and Cinder) accounting for about 400K lines of Python code. Using the same hardware, *ProFIPy* takes about 20 min to identify 17488 injectable locations using 120 different DSL patterns, which is reasonable for practical purposes given the large size of this project. The duration of the *execution* phase is beyond the control of our tool since it depends on the time to deploy the target system and run the workload. It took between 10s and 120s (worst case of a "hang" failure) to run a single experiment on *Python-etcd*, and about 30 min to run all of the tests of this section. For OpenStack, an experiment takes several tens of minutes, since it is a complex system that deploys VMs, loads large storage volumes, initializes databases, etc. We were able to execute experiments on OpenStack through nightly parallelized runs.

VI. Conclusion

ProFIPy is designed to be programmable and highly usable, by performing fault injection campaigns with customized faultloads in Python software.

The analysis of results pointed out several failure modes, which were acknowledged as valid threats by our industrial partners. The programmability of the tool through a DSL was useful to easily and quickly customize fault injections to comply with the fault classes requested by the company, based on their internal software resiliency requirements. We discussed in the paper potential strategies to mitigate the failure modes.

We plan to extend the tool with more features for failure analysis and to use it as a basis for research on software fault tolerance strategies in modern applications, such as cloud software.

Acknowledgments

This work has been done in the framework of the R&D project of the multiregional investment programme "REINForce: REsearch to INspire the Future" (CDS000609) with Hitachi Rail S.p.A., supported by the Italian Ministry for Economic Development (MISE) through the Invitalia S.p.A. agency.

References

[1] M. Hsueh, T. Tsai, and R. Iyer, "Fault injection techniques and tools," *IEEE Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[2] P. Joshi, H. Gunawi, and K. Sen, "Prefail: a programmable tool for multiple-failure injection," *ACM SIGPLAN Notices*, vol. 46, no. 10, pp. 171–188, 2011.

[3] J. Christmansson and R. Chillarege, "Generation of an Error Set that Emulates Software Faults based on Field Data," in *FTCS*, 1996.

[4] H. Madeira, D. Costa, and M. Vieira, "On the Emulation of Software Faults by Software Fault Injection," in *Proc. DSN*. IEEE, 2000.

[5] A. Johansson and N. Suri, "Error propagation profiling of operating systems," in *Proc. DSN*. IEEE, 2005, pp. 86–95.

[6] A. Lanzaro, R. Natella, S. Winter, D. Cotroneo, and N. Suri, "An empirical study of injected versus actual interface errors," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 397–408.

[7] D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa, "Experimental analysis of binary-level software fault injection in complex software," in *IEEE EDCC*, 2012.

[8] C. Q. Adamsen, G. Mezzetti, and A. Møller, "Systematic execution of Android test suites in adverse conditions," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 83–93.

[9] Z.-M. Jiang, J.-J. Bai, J. Lawall, and S.-M. Hu, "Fuzzing error handling code in device drivers based on software fault injection," in *Proc. IEEE Intl. Symp. on Software Reliability Engineering*, 2019.

[10] H. Gunawi, T. Do, P. Joshi, P. Alvaro, J. Hellerstein, A. Arpaci-Dusseau, R. Arpaci-Dusseau, K. Sen, and D. Borthakur, "FATE and DESTINI: A Framework for Cloud Recovery Testing," in *USENIX Proc. NSDI*, 2011.

[11] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE TDSC*, 2004.

[12] F. Huang, B. Liu, and B. Huang, "A taxonomy system to identify human error causes for software defects," in *18th Intl. Conf. on Reliability and Quality in Design*, 2012, pp. 44–49.

[13] F. Huang, "Human error analysis in software engineering," *Theory and Application on Cognitive Factors and Risk Management: New Trends and Procedures*, p. 19, 2017.

[14] J. Durães and H. Madeira, "Emulation of Software faults: A Field Data Study and a Practical Approach," *IEEE TSE*, 2006.

[15] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *11th {USENIX} Symposium on Operating Systems Design and Implementation*, 2014, pp. 249–265.

[16] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[17] GitHub Inc., "The state of the octoverse," 2019. [Online]. Available: https://octoverse.github.com/

[18] Stack Overflow, "Developer survey results," 2019. [Online]. Available: https://insights.stackoverflow.com/survey/2019

[19] OpenStack project, "User stories showing how the world #RunsOnOpenStack," 2019. [Online]. Available: https://www.openstack.org/user-stories/

[20] ——, "The OpenStack marketplace," 2019. [Online]. Available: https://www.openstack.org/marketplace/

[21] Python Software Foundation, "Applications for Python," 2019. [Online]. Available: https://www.python.org/about/apps/

[22] R. Chillarege and N. Bowen, "Understanding Large System Failures–A Fault Injection Experiment," in *Digest of Papers, Intl. Symp. on Fault-Tolerant Computing*, 1988, pp. 356–363.

[23] M. Sullivan and R. Chillarege, "Software Defects and their Impact on System Availability: A Study of Field Failures in Operating Systems," in *Digest of Papers, Intl. Symp. on Fault-Tolerant Computing*, 1991, pp. 2–9.

[24] ——, "A comparison of software defects in database management systems and operating systems," in *Digest of Papers, Intl. Symp. on Fault-Tolerant Computing*, 1992, pp. 475–484.

[25] W.-I. Kao, R. Iyer, and D. Tang, "FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults," *IEEE TSE*, vol. 19, no. 11, pp. 1105–1118, 1993.

[26] R. Chillarege, W. Kao, and R. Condit, "Defect Type and its Impact on the Growth Curve," in *Proc. ICSE*. IEEE, 1991, pp. 246–255.

[27] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, and M. Wong, "Orthogonal Defect Classification–A Concept for In-Process Measurements," *IEEE TSE*, vol. 18, no. 11, pp. 943–956, 1992.

[28] W. Ng, C. Aycock, G. Rajamani, and P. Chen, "Comparing disk and memory's resistance to operating system crashes," in *Proc. ISSRE*. IEEE, 1996, pp. 185–194.

[29] W. Ng and P. Chen, "The Design and Verification of the Rio File Cache," *IEEE TC*, vol. 50, no. 4, pp. 322–337, 2001.

[30] I. Lee and R. Iyer, "Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System," in *Digest of Papers, Intl. Symp. on Fault-Tolerant Computing*, 1993, pp. 20–29.

[31] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, "Recovering device drivers," *ACM TOCS*, vol. 24, no. 4, pp. 333–360, 2006.

[32] J. Duraes and H. Madeira, "Emulation of Software Faults by Educated Mutations at Machine-Code Level," in *Proc. ISSRE*. IEEE, 2002, pp. 329–340.

[33] T. Basso, R. Moraes, B. Sanches, and M. Jino, "An Investigation of Java Faults Operators Derived from a Field Data Study on Java Software Faults," in *Workshop de Testes e Tolerância a Falhas*, 2009.

[34] B. Sanches, T. Basso, and R. Moraes, "J-SWFIT: A Java Software Fault Injection Tool," in *Proc. LADC*. IEEE, 2011.

[35] D. Cotroneo and R. Natella, "Fault injection for software certification," *IEEE Security & Privacy*, vol. 11, no. 4, pp. 38–45, 2013.

[36] E. van der Kouwe and A. S. Tanenbaum, "HSFI: Accurate fault injection scalable to large code bases," in *Proc. DSN*. IEEE, 2016.

[37] O. Schwahn, N. Coppik, S. Winter, and N. Suri, "FastFI: Accelerating software fault injections," in *Proc. PRDC*. IEEE, 2018, pp. 193–202.

[38] S. Winter, C. Sârbu, N. Suri, and B. Murphy, "The impact of fault models on software robustness evaluations," in *Proc. ICSE*. IEEE, 2011, pp. 51–60.

[39] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Edfi: A dependable fault injection tool for dependability benchmarking experiments," in *Proc. PRDC*. IEEE, 2013, pp. 31–40.

[40] M. R. Aliabadi and K. Pattabiraman, "FIDL: A fault injection description language for compiler-based SFI tools," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2016, pp. 12–23.

[41] W. Hoarau, S. Tixeuil, and F. Vauchelles, "FAIL-FCI: Versatile fault injection," *Elsevier FGCS*, vol. 23, no. 7, pp. 913–919, 2007.

[42] P. Marinescu and G. Candea, "LFI: A practical and general library-level fault injector," in *Proc. DSN*. IEEE, 2009.

[43] B. Sampson, "How to automate software fault injection testing, without changing source code," Aug. 2018. [Online]. Available: https://www.aerospacetestinginternational.com/opinion/how-to-automate-software-fault-injection-testing-without-changing-source-code.html

[44] J. Happich, "Automated fault injection without source code change," Jan. 2018. [Online]. Available: http://www.eenewseurope.com/news/automated-fault-injection-without-source-code-change

[45] D. Cotroneo, L. De Simone, P. Liguori, R. Natella, and N. Bidokhti, "How bad can a bug get? an empirical analysis of software failures in the openstack cloud computing platform," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 200–211.

[46] D. Cotroneo, L. De Simone, A. K. Iannillo, R. Natella, S. Rosiello, and N. Bidokhti, "Analyzing the context of bug-fixing changes in the openstack cloud computing platform," in *Proceedings of the 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, 2019, pp. 334–345.

[47] OpenStack Launchpad, "Bug #1028174 "tenant cannot delete network when dhcp-agent is running"," Jul. 2013. [Online]. Available: https://bugs.launchpad.net/neutron/+bug/1028174

[48] ——, "Bug #1096722 "inconsistent nova-bm state will prevent launching new instances"," Jan. 2013. [Online]. Available: https://bugs.launchpad.net/nova/+bug/1096722

[49] ——, "Bug #732549 "execvp fallout"," Mar. 2011. [Online]. Available: https://bugs.launchpad.net/nova/+bug/732549

[50] Docker Inc., 2019. [Online]. Available: https://www.docker.com/

[51] ——, 2019. [Online]. Available: https://docs.docker.com/engine/reference/builder/

[52] S. Winter, O. Schwahn, R. Natella, N. Suri, and D. Cotroneo, "No PAIN, no gain?: the utility of PArallel fault INjections," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 494–505.

[53] T. Yoshimura, H. Yamada, and K. Kono, “Using fault injection to analyze the scope of error propagation in Linux,” *Information and Media Technologies*, vol. 8, no. 3, pp. 655–664, 2013.

[54] M. Sugimoto, T. Kubota, and K. Kono, “Short-liveness of error propagation in kernel can improve operating systems availability,” in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks–Supplemental Volume (DSN-S)*. IEEE, 2019, pp. 23–24.

[55] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, “Software rejuvenation: Analysis, module and applications,” in *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*. IEEE, 1995, pp. 381–390.

[56] M. Grottke and K. Trivedi, “Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate,” *IEEE Computer*, vol. 40, no. 2, pp. 107–109, 2007.

[57] D. Cotroneo, L. De Simone, P. Liguori, R. Natella, and N. Bidokhti, “Enhancing failure propagation analysis in cloud computing systems,” in *Proceedings of the 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, 2019, pp. 139–150.

[58] Zipkin. [Online]. Available: https://zipkin.io

[59] D. Cotroneo, L. De Simone, P. Liguori, R. Natella, and N. Bidokhti, “Failviz: A tool for visualizing fault injection experiments in distributed systems,” in *2019 15th European Dependable Computing Conference (EDCC)*. IEEE, 2019, pp. 145–148.

[60] Python-etcd, 2019. [Online]. Available: https://pypi.org/project/python-etcd/

[61] etcd HomePage, 2019. [Online]. Available: https://etcd.io/