Uncovering Web API Performance Anomalies through Response-Time-Guided Fuzz Testing based on Genetic Algorithm

Ying-Tzu Huang and Shin-Jie Lee

Department of Computer Science and Information Engineering, National Cheng Kung University, Taiwan
yingcihuang133@gmail.com, jielee@mail.ncku.edu.tw

Abstract: Fuzz testing is a software testing technique that involves injecting random and unexpected inputs into a program to identify vulnerabilities and flaws. It has been widely applied in diverse domains, frequently in combination with genetic algorithms. While some research endeavors employ a combination of genetic algorithms and fuzzing techniques to identify vulnerabilities in Web application, there is a relative scarcity of studies directly addressing the detection of web API performance anomalies. In this research, we propose a response-time-guided fuzzing approach that utilizes response times and a genetic algorithm to uncover web API performance anomalies. In our experiment with a real-world targeted web API, the proposed approach revealed a sudden and severe response latency, uncovering a performance anomaly that cannot be detected by a pairwise approach. Furthermore, it was confirmed that the detected crash-inducing input almost consistently leads to longer response times.

Keywords: Genetic algorithm, fuzz testing, web APIs testing

1. INTRODUCTION

Fuzz testing is an automated software testing methodology that entails the deliberate injection of erroneous, unanticipated, or random data as input into a software program. Subsequently, the program is observed and scrutinized for any irregularities, which may include occurrences such as program crashes, failed internal code assertions, or the identification of potential memory leaks. While a substantial portion of the existing research concentrates on uncovering web API bugs [1][2][3][4], relatively limited attention has been directed towards investigations related to web API performance. In this research, our emphasis lies in the identification of anomalies in API performance. To achieve this objective, we have employed a genetic algorithm as a fundamental component of our approach.

The genetic algorithm has found extensive application alongside fuzzing techniques across various domains. For instance, KameleonFuzz has been specifically devised for the purpose of black-box XSS (Cross-Site Scripting) detection [5]. In addition, code-coverage guided fuzzing methods have been directed towards the discovery of desktop application vulnerabilities [6] [7], while others have employed genetic algorithms in the context of Web application vulnerability detection [8].

In the context of our research, we have introduced an approach termed 'response-time-guided fuzzing,' which harnesses response time data obtained from target Web APIs. This approach, in conjunction with a genetic algorithm, is employed to uncover performance-related vulnerabilities with fuzz seeds generated by pairwise testing [9]. In the experiment, we applied our approach to a real-world web API - Course Query API in NCKU Course Information and Enrollment System. The results demonstrate that the proposed approach revealed a sudden and severe response latency, uncovering a performance anomaly that cannot be detected by a pairwise approach. Furthermore, it was confirmed that the detected crash-inducing input almost consistently leads to longer response times.

2. RELATED WORKS

RESTler, introduced by Atlidakis et al. [1], offers a black box testing approach for Web APIs. This method employs generation-based fuzzing to identify software bugs, diverging from the focus of uncovering performance anomalies within Web API. Tsai et al. [4] introduced a method that utilizes test coverage level to guide genetic algorithms in improving the efficiency of automatic Web API testing. Same as RESTler, their emphasis is on identifying bugs. AFL [7], widely recognized for its prowess in gray box fuzzing, employs code coverage to direct genetic algorithms in uncovering vulnerabilities within desktop applications. However, it's worth noting that AFL necessitates access to the source code of the fuzz target, whereas this paper employs black box fuzzing, a method not bound by source code limitations. Zhou et al. [8] utilized genetic algorithms to detect web vulnerabilities, such as SQL injection and XSS. However, their approach does not provide direct support for Web API testing.

3. RESPONSE-TIME-GUIDED FUZZ TESTING

In this section, we present the proposed response-time-guided fuzzing approach based on genetic algorithm. We begin by introducing the target system and API. Subsequently, we outline the key steps, including seed generation using PICT, the fuzzing process, and the integration of response time data with the genetic algorithm to create new entities. These steps are depicted in Figure 1.

3.1 Target System and API

In our experiment, we selected Courses Query API as our primary fuzz target. This API is an integral component of the

NCKU Course Information and Enrollment System. Below, experiments.

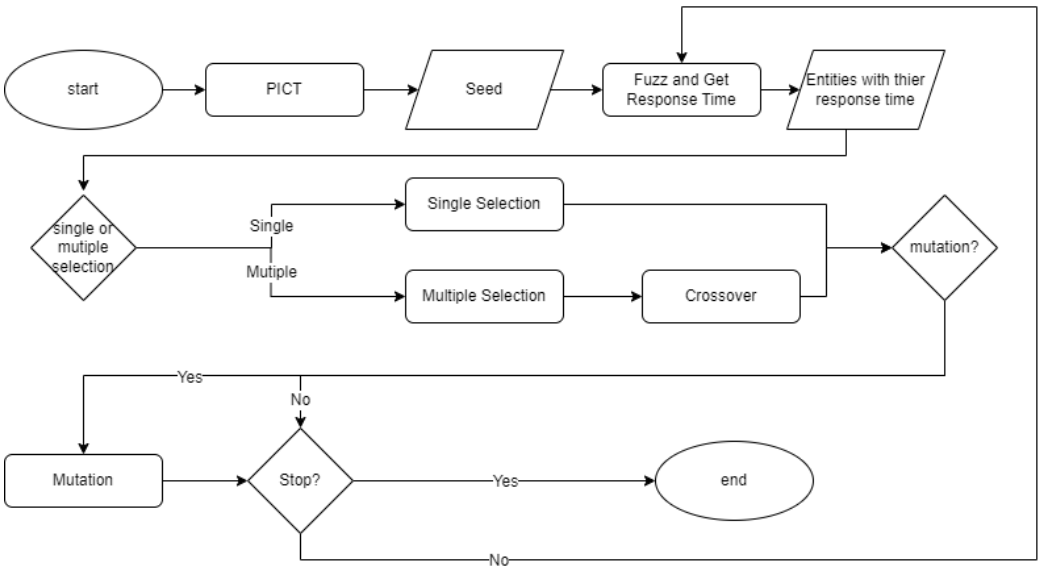


Fig. 1 Operational concept of response-time-guided fuzz testing

we provide an overview of the API parameters:

- cosname (Course Name): A string variable obtained from user input.
- teaname (Teacher Name): A string variable obtained from user input.
- wk (Week): An integer variable ranging from 1 to 7, representing the days of the week.
- dept_no (Department Number): A specific string denoting the department associated with the course.
- degree (Degree): An integer variable ranging from 1 to 7, signifying the degree level.
- cl1 to cl16 (Time Sections for Courses): Boolean variables that can be set to either 0 or 1, indicating the availability of time slots for courses.

3.2 Seed Generation Using PICT

The concept of a "seed" holds significant importance in the realm of fuzzing. Its role goes beyond mere initiation, as the choice of an optimal seed can profoundly influence the effectiveness of the entire fuzzing process. Some delve deeply into the intricate task of selecting the most suitable seed [10]. In our methodology, we harnessed the power of Pairwise Independent Combinatorial Testing (PICT) to systematically generate the initial fuzz seed dataset. PICT's combinatorial approach ensured a comprehensive and efficient representation of seed input combinations for our

3.3 Responed-Time-Guided Genetic Algorithm

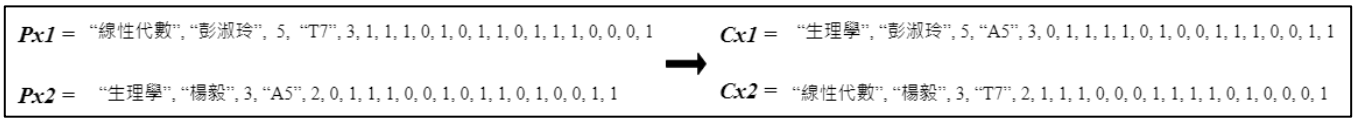
Following the initial fuzzing process using the seed generated in Section 3.2, we obtain the response time for each individual entity. These response times serve as the fitness criteria for our genetic algorithm (GA). The subsequent steps in our approach involve the utilization of GA to enhance our fuzzing strategy. The following outlines the strategy in our genetic algorithm:

1) Selection

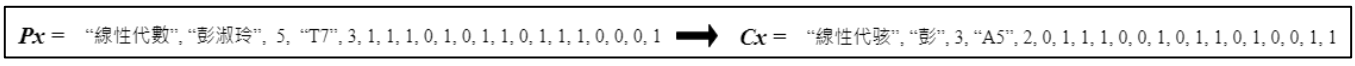
The selection step plays a critical role in determining parent entities for either crossover or mutation. This decision is influenced by a probability factor. When

Table 1 The strategies of mutation

Parameter	Parameter Type	Strategy
Cosname, Teaname	string	Delete, Insert, Flip or Swap characters
wk, dept_no, degree	Number or string from specific range	Random choose one value from the range
Cl1 to cl 16	0 or 1	Flip the bit



(a) Example of crossover



(b) Example of mutation

Fig. 2 Examples of crossover and mutation

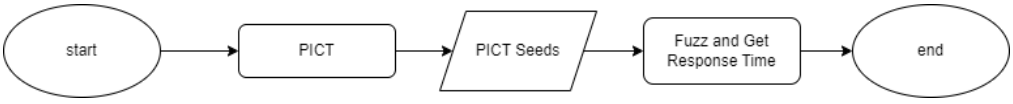


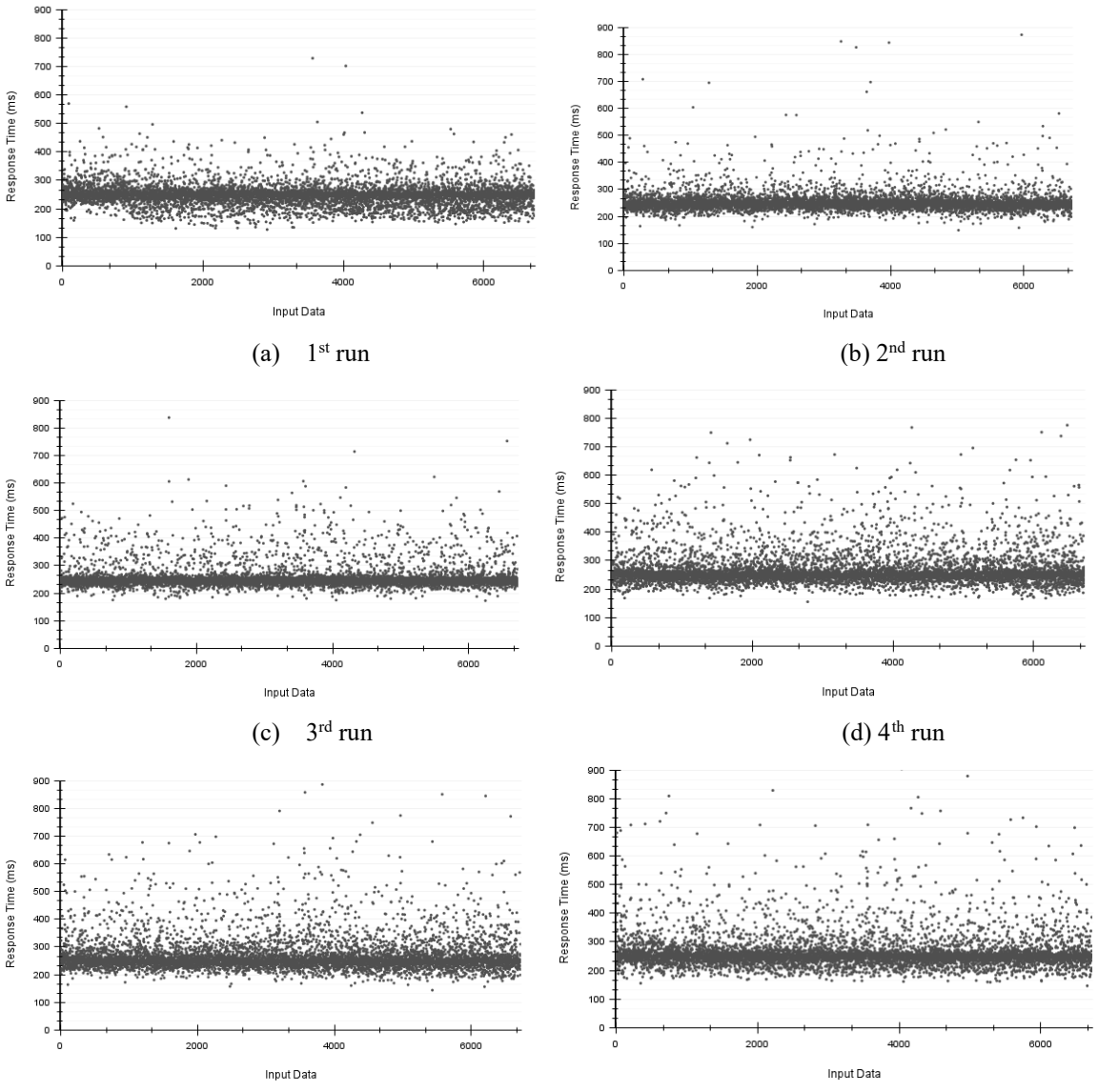
Fig. 3 A comparison subject: fuzz input data generated only by PICT without a genetic algorithm.

crossover is required, multiple selection methods come into play, while in the absence of crossover, single selection methods are applied.

In our approach, we employ two distinct selection methods. Roulette Wheel Selection is applied for multiple selection, which avoids killing poorer individuals too soon but still favoring the better individuals [10]; Rank Selection is our choice for single selection. It is similar to Roulette Wheel selection but allocates probabilities based on entity order.

2) Crossover

In the realm of crossover strategies, the Uniform method has better performance when compared to other common methods [11]. Consequently, we have selected Uniform as our preferred crossover strategy. We select two entities as our parents and for each parameter there is a 0.5 probability of switching their positions. This switching results in the creation of two new child entities. An illustration of this uniform crossover example is presented in Figure 2 (a). We designate the two parent entities as *Px1* and *Px2*. The swapping process involves the first parameter, resulting in the first child entity having "生理學" (denoted as *Cx1*), and the second child entity having "線性代數" (denoted as *Cx2*).



(a) 1st run (b) 2nd run

(c) 3rd run (d) 4th run

(e) 5th run (f) 6th run

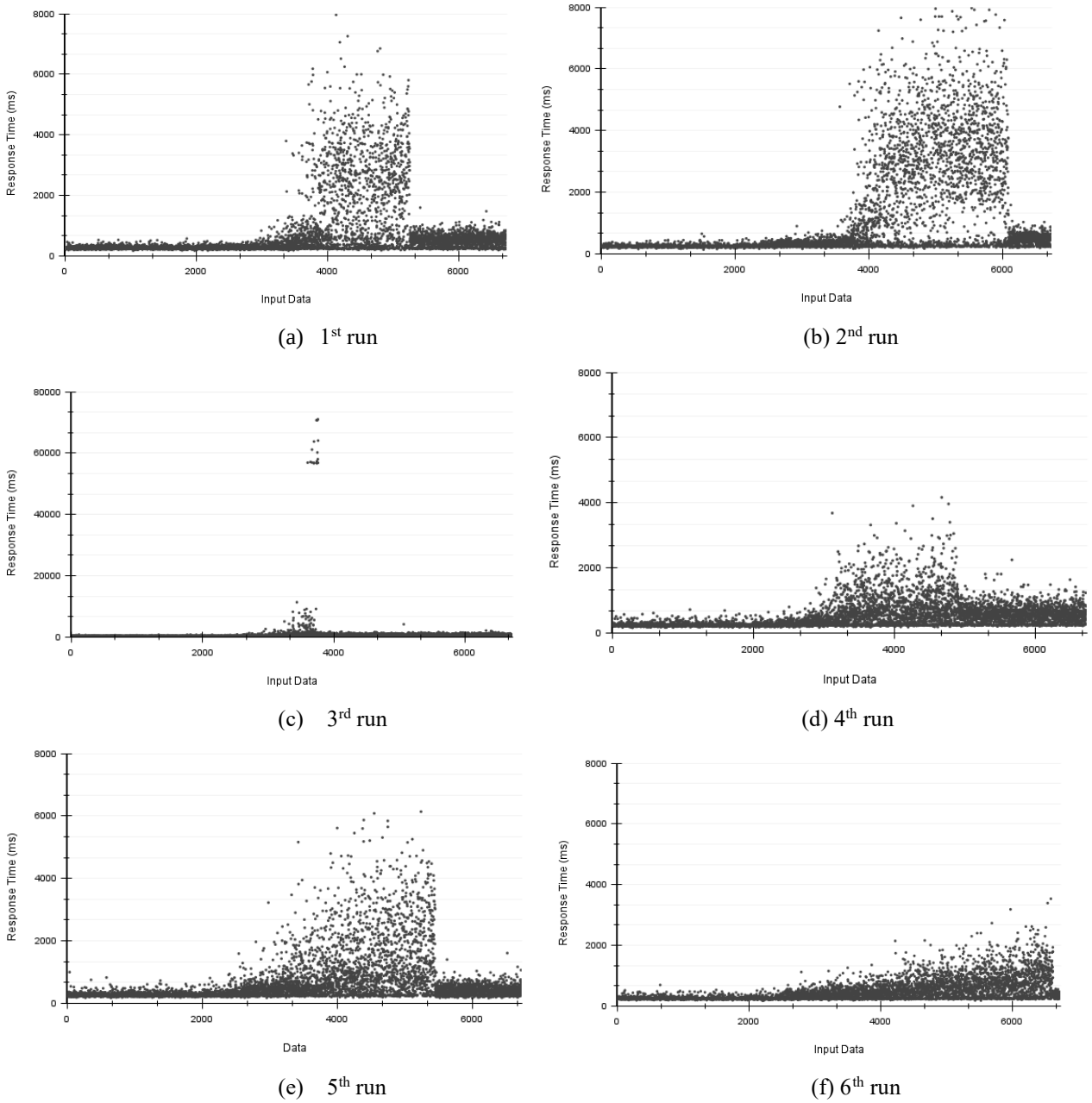
Fig. 4 No web API performance anomalies uncovered by PICT-only fuzzing.

3) Mutation

Mutation is a crucial component for generating novel entities. Through mutation, GA has the ability to create entities distinct from the seed. Our approach accommodates the diversity of parameter types within our target API. To ensure tailored treatment, we employ distinct mutation strategies based on parameter type. For cosname and teaname parameters (string type), we utilize common mutation strategies designed for string variables. Each character within the "cosname" and "teaname" strings has a 0.5 probability of undergoing mutation. If a mutation is triggered, one of the following operations is randomly selected: delete, insert, swap, or flip. This approach allows the input string to incorporate random words.

Conversely, parameters like "wk," "dept_no," and "degree" are characterized by specific ranges. Accordingly, our approach involves the random selection of a single value within these predefined ranges. Conversely, parameters like "wk," "dept_no," and "degree" are characterized by specific ranges. Accordingly, our approach involves the random selection of a single value within these predefined ranges. This strategy ensures that these parameters remain within the specified range, simplifying the mutation process and making the generated inputs more closely resemble realistic values.

Parameters denoted as "Cl1" to "Cl16" are characterized by binary values, limited to either 0 or 1. To accommodate this simplicity, our mutation strategy for these parameters adheres to a straightforward approach. Specifically, there is a 0.5 probability assigned to the operation of value inversion, effectively flipping the binary state of these parameters. The



(a) 1st run (b) 2nd run

(c) 3rd run (d) 4th run

(e) 5th run (f) 6th run

Fig. 5 A web API performance anomaly uncovered by response-time-guided fuzzing

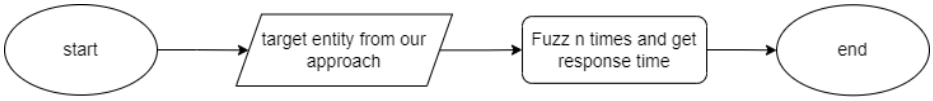


Fig. 6 Fuzzing n times repeatedly with only one specified input

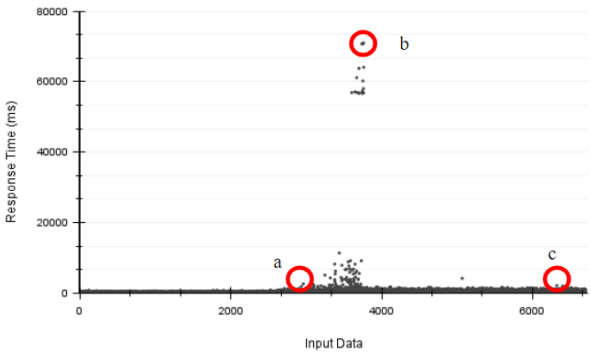


Fig. 7 Point *a*, *b* and *c* from Fig 6 (c)

strategies for each parameter are listed in Table 1. An illustrative example of mutation is presented in Figure 2 (b). We denote the entities before mutation as *Px*. The first parameter has its last character flipped, and two characters from the second parameter are deleted. For the wk, dept_no, and degree parameters, new values are randomly selected from their specific ranges. The last parameters undergo a flip mutation. As a result, we obtain a new entity denoted as *Cx*.

4. EXPERIMENTAL RESULTS

4.1 Experiment 1: Uncovering anomalies

To further investigate the effectiveness of our approach, we undertook a comparative analysis against a set of fuzz input data generated only by PICT without a genetic algorithm (see Figure 3). Figure 4 displays the results of six runs conducted using the PICT-only approach (measured in milliseconds). In each run, we collect response times for all input data. As shown for the six runs, no web API performance anomalies were uncovered by PICT-only fuzzing.

On the other hand, we conducted the proposed approach, involving an iterative process comprising 50 rounds. In each round, requests were sent to acquire response times and subsequently utilize GA to generate a new set of entities. Notably, each round yielded 100 novel entities, while the top 10 entities with longest response time from the previous generation were retained as the elitism. Given the inherent instability of the target system, each entity undergone 10 executions, and the median response time from these runs was utilized as the fitness.

Figure 5 displays the results of six runs conducted using our approach. In each run, we collect response times for all entities generated through our approach, encompassing both the initial seed and subsequent iterations. Entities generated in order by the genetic algorithm are plotted from left to right, reflecting the progression of the iterative process. As shown

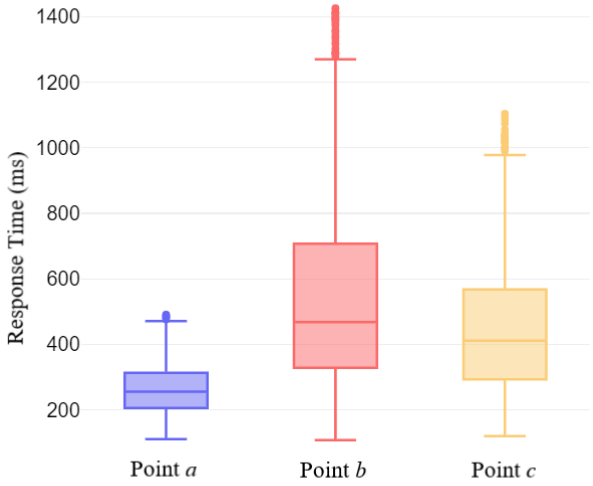


Fig. 8 Response times of the three selected points

on the plot, the response-time-guided fuzz testing reveals a noteworthy observation: **A sudden and severe response latency occurs in every round, uncovering a performance anomaly of the target web API**.

4.1 Experiment 2: Evaluating crash-inducing inputs

In the second experiment, we want to confirm that: if the crash-inducing inputs, triggering prolonged response times, can always cause the long response times? The experimental design is shown in Figure 6. We selected three key points (inputs) from the results of the 3rd run by response-time-guided fuzzing in the first experiment. These selected points were then subjected to 1000 rounds of fuzzing to empirically verify whether the identified inputs genuinely caused prolonged response times. The first point, denoted as *a* in Figure 7, corresponds to the highest fitness value and is situated towards the left end of the plot; notably, this point is also the highest among the initial seed inputs generated by PICT. The second point, labeled *b*, is the input with the longest response time among the overall points. Finally, the third point, marked as *c*, corresponds to the input with the longest response time observed near the end of the experiment.

The experiment results are depicted using as a boxplot (see Figure 8), enabling a clear comparison of the fuzzing outcomes for the three selected points. The parameters associated with the boxplot are detailed in Table 2. It is evident that point *a* exhibits a significantly lower average response time compared to points *b* and *c*. As expected, point *b* demonstrates better performance than *c*. These findings substantiate that the crash-inducing input, *b*, identified by our method indeed almost constantly induces longer response times.

Table 2 Average, min, Q1, median, Q3 and max of response times for the three points (in milliseconds)

Response Time	**Point *a***	**Point *b***	**Point *c***
Average	262	550	456
Min	111	108	120
Q1	206	329	293
Median	255	468	411
Q3	313	707	568
Max	491	1426	1104

5. CONCLUSION

In this study, we introduced a novel response-time-guided fuzzing approach based on genetic algorithm to detect web API performance anomalies. In our experiments with a real-world web API, we revealed its effectiveness in uncovering a sudden and severe response latency. Furthermore, it was confirmed that the crash-inducing input almost consistently leads to longer response times. The experiment also demonstrated the superiority of our method over a naive fuzzing approach that relies solely on pairwise test case generation. This preliminary finding shows its potential to uncover possible web API performance anomalies. In our future work, we will apply this method to more real-world web APIs to further improve its efficiency and effectiveness.

6. ACKNOWLEDGEMENT

This research is sponsored by Taiwan National Science and Technology Council under the grant NSTC 112-2221-E-006-084-MY2 and NCKU under the account number AA10202015.

7. REFERENCES

[1] V. Atlidakis, P. Godefroid and M. Polishchuk, “RESTler: Stateful REST API Fuzzing”, *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, Montreal, QC, Canada, pp. 748-758, 2019.

[2] P. Godefroid, B.-Y. Huang, and M. Polishchuk, “Intelligent REST API Data Fuzzing”, *the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, New York, USA, pp. 725-736, 2020.

[3] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, “RESTest: Automated Black-Box testing of RESTful Web APIs”, *In Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, New York, USA, pp. 682–685, 2021.

[4] C.-H. Tsai, S. -C. Tsai and S. -K. Huang, “REST API Fuzzing by Coverage Level Guided Blackbox Testing”, *IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, Hainan, China, pp. 291-300, 2021.

[5] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, “KameleonFuzz: Evolutionary Fuzzing for Black-Box XSS Detection”, *In Proceedings of the 4th ACM Conference on Data and Application Security and Privacy,* New York, USA, pp. 37–48, 2014.

[6] A. Takanen, J. D Demott, C. Miller and A. Kettunen, “Fuzzing for Software Security Testing and Quality Assurance”, *Artech House*, 2018.

[7] American Fuzzy Lop (AFL). “https://github.com/google/AFL”

[8] X. Zhou and B. Wu, “Web Application Vulnerability Fuzzing Based on Improved Genetic Algorithm”, *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, Chongqing, China, pp. 977-981, 2020.

[9] PICT - Pairwise Independent Combinatorial Testing, “https://github.com/Microsoft/pict”

[10] E. Jääskelä, “Genetic Algorithm in Code Coverage Guided Fuzz Testing”, *University of Oulu,* 2016.

[11] G. Syswerda, “Uniform Crossover in Genetic Algorithms”, *3rd International Conference on Genetic Algorithms,* pp. 2-9, 1989.