Beacon: Automated Test Generation for Stack-Trace Reproduction using Genetic Algorithms

Ignacio Slater M.
Computer Science Department (DCC)
University of Chile
Santiago, Chile
ignacio.slater@ug.uchile.cl

Alexandre Bergel
Computer Science Department (DCC)
University of Chile
Santiago, Chile
abergel@dcc.uchile.cl

***Abstract*—This paper describes the experience we have gained by reproducing software failure using genetic algorithm. Our approach is based on reproducing exception for which its associated runtime call stack matches the stack of the error to reproduce. We present two tools for Python. Three simple case studies illustrate the feasibility of our approach.**

I. Introduction

Anyone with experience in programming knows that finding and reproducing errors can be one of the most time expensive activities in the development of a software project. Writing a piece of software (especially when it starts to grow in dimensions) that has no bugs is virtually impossible.

Writing a robust set of tests over our software is a very effective way to prove functionality and find errors in the application, but even then, tests can only show the existence of errors and not the absence of them. There's a very famous quote of Donald Knuth saying "Beware of bugs in the above code; I have only proved it correct, not tried it" [1], implying that even though he gave a proof on the correctness of the code he could not assure that it was free of errors.

The problem is that generating a robust set of tests is a labour-intensive task. To address this issue many tools for automated test generation for crash replication have been introduced in the literature; but even so there's a lack of development in this topic on *dynamically typed languages* such as Python, Ruby, or Smalltalk.

This paper presents two tools:

- *Genyal:* a general purpose evolutionary programming framework written completely in Python focused on being simple to use while maintaining the quality of the results; and
- *Beacon:* a software that uses the previously mentioned framework to reproduce program crashes by generating sequences of instructions that raises a given *exception* and the stack trace associated to it.

The proposal of this tools follows two main objectives:

1) Providing a way to reproduce *Python* crashes.
2) Generalize the ideas presented in previous works to dynamically typed languages to empirically prove that there's no need to provide type information (be it by explicit declaration or by type inference) to apply stacktrace based test generation techniques to a program.

II. State-of-the-art

Most of the previous implementations of using genetic algorithms (GA) for test generation focuses on maximizing the *coverage* [2]–[5].

On the other hand, previous approaches used for crash reproduction do not usually apply evolutionary programming to generate the tests [6]–[8].

Some recent approaches have been using the error's stack trace as the only source of information to replicate software crashes [9]–[11], but the applications of this approach are almost no existent for dynamically typed languages.

One approach that is going to be particularly important in this area is *EvoCrash* [12]; this tool introduces a search-based *Guided Genetic Algorithm* (GGA) to replicate crash stack traces on Java programs, with mostly possitive results, outperforming other approaches with high precision and short execution times [13].

It is important to note that crash reproduction generation does not assure anything about code coverage because we would like the generated code to go through the less amount of lines of code as possible, aiming at finding the minimum amount of instructions to replicate the exception.

III. Genyal

To solve the problem of generating a *minimal crash replication* (MCR) test we introduce *Genyal*, a genetic algorithm framework implemented in pure Python aimed at being effective yet simple to use. This was done because previous approaches to evolutionary programming are either outdated or have a very complex flow that obfuscate the code (PyEvolve [14] was last updated 5 years ago and DEAP [15] requires a series of configurations in order to execute even simple algorithms that ends up complicating the code).

The architecture of Genyal is composed by two main components: a *gene factory* and the *genetic algorithm engine*.

The gene factory (represented by the `GeneFactory` class) is an object that will contain the logic to create new genes from a given generator function (see lines 1-3, 11-12 of listing 1). For a given population, all individuals will share the factory assuring that all generated members of the population are consistent. This also provides flexibility since the generator function can be any function defined by the user of the framework, this makes the algorithm more generic, thus adaptable to any kind of problem (as we will see in the next section when we'll use the factory to generate lines of code).

The engine (defined as `GenyalEngine`) is the runner of the GA, this class is also very flexible allowing almost total control over its parameters and execution, but it also provides default implementations for most of them (tournament selection, single-point crossover, etc) in such way that the manual configurations are done only when absolutely needed. There is only a few parameters that are going to be needed for almost all use cases: the number of individuals in the population, the number of genes per individual (lines 16-17 of listing 1 shows the creation of a population of size 32 where each individual has 3 genes), the function used to calculate an individual's fitness, and, a function to stop the engine (see lines 4-10, 13-15 of listing 1).[1]

```
1   def random_char():
2       return random.choice(
3         string.ascii_lowercase)
4   def fitness_fun(word: list[str]) -> float:
5       return sum([word[i] == "crow"[i]
6         for i in range(0, 4)])
7   def target(genyal_engine: GenyalEngine) \
8         -> bool:
9       return "".join(genyal_engine \
10        .fittest.genes) == "crow"
11  gene_factory = GeneFactory()
12  gene_factory.generator = random_char
13  engine = GenyalEngine(
14    fitness_function=fitness_fun,
15    terminating_function=target)
16  engine.create_population(32, 4,
17    gene_factory)
18  engine.evolve()
19  print(engine.fittest)
20
```

Listing 1. Example of using Genyal to find the word crow

IV. Beacon

The main contribution of this paper is *Beacon*, a tool to generate a MCR based solely on the runtime stack. For this we use a search-based guided genetic algorithm similar to the one presented by Soltani et. al. [12].

The GGA used by *Beacon* follows the same structure as most genetic algorithms (i.e. create a population, crossover, mutation) which objective is to **maximize** a fitness function that represents the similarity of the produced stack trace and the desired one.

The following sections will explain the details of the algorithm.

A. Population

Each individual of the population is defined by a series of statements. To further explain this consider the following definitions:

[1]The specific details of the framework definition can be found at https://github.com/islaterm/genyal

Definition 1. Let P be the population of a GA. The alphabet of the population, α_P, is the set of values used to generate each one of the individuals of the population such that for all individuals $I_i \in P$, and, for each gene γ_j^i of the individual we have that:

$$\gamma_j^i \in \alpha_P$$

Definition 2. We define the signature σ of a function f as a set of pairs (p_n, p_v) such that p_n is the name of a parameter of f and p_v the value of the parameter.[2]

Definition 3. A statement S is a tuple of values (f, n, σ) where:

- f is a function contained in the alphabet α_P
- n is the name of the function f
- σ is the signature of the function f

for α_P the alphabet of a population P.

Note. Depending on the signature of f the alphabet may be an infinite set since p_v could be any value; e.g. $\sigma = (\text{"n"}, p_v)$ where $p_v \in \mathbb{R}$ is a valid signature.

For the first step of the GGA *Beacon* receives the name of a Python module as a parameter. The alphabet of the algorithm is then defined as the set of all functions contained in the given module with random inputs generated from the statement signature and an *input factory* that's part of *Beacon*'s specification.

Next, an initial population is created by generating individuals which genes are a series of N statements.

B. Fitness function

To define the fitness of an individual *Tracer* (the main class of *Beacon*) receives 4 arguments:

1) The previously mentioned Python module's name.
2) The type[3] of the exception involved in the crash; e.g. `AssertionError`.
3) (Optional) A string that should be contained in the arguments of the thrown exception.
4) (Optional) The name of a function that should be present on the runtime stack of the crash.

The following snippet illustrates the usage of *Tracer*:

```
tracer = Tracer("my_module", AttributeError,
    "Arguments should be integers",
    "sum_integers")
```

Listing 2. Example of using Genyal to find the word crow

The main objective of these optional parameters will be to further specify the desired error, this will help the *Tracer* to choose between different statements that may raise the same kind of exception.

[2]In the literature is common for the signature to be defined in terms of the name of the function, its return type (if any) and, the name and type of its arguments. The definition presented in this work is slightly different to focus only on the data that is useful for the algorithm, and, since we are using a dynamically typed language, the type of the parameters should not be a factor for the algorithm.

[3]Note that we need the class of the exception, not an instance of it.

The fitness function definition is heavily inspired by the one presented by Soltani et.al. [12] modified to better adapt to *Beacon*'s approach. The fitness function used by *EvoCrash* depends of three parameters: (1) the location of the crash, (2) the exception class, and (3) the actual stacktrace. Then EvoCrash fitness is defined as

$$f(t) = 3d_s(t) + 2d_{\text{except}}(t) + d_{\text{trace}}(t)$$

for a given test t.

On the other hand *Beacon*'s approach focuses on generating the smallest set of instructions to reproduce a crash. Given that, the resulting stacktrace generated by *Tracer* may not be the same as the one produced by the original crash, so the actual difference between the stacktraces is not as relevant. Instead we focus more on the specific exception that was thrown from the execution, particularly, (1) the type of error, (2) the contents of the error (in this case, the error message), and (3) the location of the crash (given by the presence of a target function on the stacktrace). With that we can define the fitness function.

Definition 4. The fitness of an individual $I = (S_0, \dots, S_N)$ is given by:

$$F_I = 2t_{\text{ex}} + t_{\text{arg}} + 2t_{\text{fn}}$$

where:

$$t_{\text{ex}} = \begin{cases} 1 & \text{if the desired exception type was raised} \\ 0 & \text{otherwise} \end{cases}$$

$$t_{\text{arg}} = \begin{cases} 1 & \text{if the exception's message contains the desired string} \\ 0 & \text{otherwise} \end{cases}$$

$$t_{\text{fn}} = \begin{cases} 1 & \text{if the specified function is present in the stacktrace} \\ 0 & \text{otherwise} \end{cases}$$

Note that, given that every value in the fitness function is either 0 or 1, the maximum fitness of an individual will be 5, in which case the algorithm achieved a MCR. This function, although simplistic produced overall succesful results as can be seen on section VI.

To calculate the fitness of an individual first we need to execute all of its statements to see if an exception is thrown. Everytime the fitness function is called the algorithm will execute each statement sequentially and store the returned value in a variable. Next, for each statement it will feed the function with parameters that can be:

- a randomly generated value (number, string, etc.), or
- the returned value of a previously executed function.

Note. For both cases, we don't worry about checking if the type of the parameters is the ones that the function expects since the function will just raise an exception if the types are wrong, and, as long as it's not the exception that we're looking for, it'll just result in a lower fitness for the individual.

If an exception is raised during the execution of the statements, the algorithm takes a snapshot of the runtime stack's frames and calculates the fitness according to F_I. On the other hand, if no exception is raised the individual's fitness is set to 0.

C. Crossover, mutation and selection

The crossover, mutation an selection all follow a standard implementation, and are handled entirely by *Genyal*. Since there's nothing novel in these steps we're not gonna dwelve deep into details for the sake of brevity. With that said, the applied strategies are:

- **Selection:** a standard tournament selection strategy that selects the individuals with higher fitness.
- **Crossover:** a single-point crossover that takes the first i statements of one of its parents and j from the other (given that $i + j = N$) and creates a new sequence of instructions by appending them; this operation returns only a single child.
- **Mutation:** for each instruction of the individual, it replaces it with a new instruction with random inputs or a reference to an instruction that'll be called before this one, with a small probability.

D. Minimization

After the genetic algorithm has finished its execution *Tracer* reduces the amount of statements of the fittest individual to generate the MCR. This step is done by applying a simple greedy algorithm like the following

```
$S \leftarrow$ The sequence of instructions of the fittest individual
for all $s \in S$ do
    $\text{Candidate} \leftarrow S \setminus \{s\}$
    if $\textit{fitness}(\text{Candidate}) \geq \textit{fitness}(S)$ then
        $S \leftarrow \text{Candidate}$
    end if
end for
```

Then, the final result given by *Tracer* will be the sequence of instructions S given by the application of this algorithm to the fittest individual.

V. Case study

The context of the experiments in the study consists on generating test cases to reproduce 3 common errors in Python. The first one (V-A) only depends on functionalities of the standard library, and the remaining 2 (V-B and V-C) use *NumPy* [16] and *PyYAML* [17], two of the most used Python libraries [18].

To assure the consistency of the experiments all experiments were run under the same conditions (same environment, individual and population lenghts, etc). This conditions be further explained in section VI.

A. Erroneous list extension

For the first experiment consider the following function:

```
def list_reduction(lst):
    def aux(x, y):
        x.extend(y)
    return reduce(lambda x, y: aux(x, y),
                  lst)

```

Listing 3. Wrong use of the `extend` function

This code snippet has an important error that's hard to see when doing a quick look over the code which may lead to very difficult to find bugs.

Consider the following code

```
l = [1, 2, 3]
list_reduction(l)

```

Listing 4. Wrong use of the `extend` function

If executed, the above code will raise an `AttributeError` because the `extend` function has no return value (or to be more precise, it returns `None`). The fact that the function uses both a lambda and a local function makes the code even more difficult to debug.

B. Invalid buffer size

This experiment consists on searching a MCR using only the functions defined in the module `numpy.core`, with no dependence on any code written by us. For this, *Beacon* will retrieve all functions on the given module (with the exception of some built-in functions for which the tools provided by the standard library are incapable to get their proper signature).

In this case, to show that the proposed tool is adaptable to many scenarios, the type of the error will not be provided. Instead, *Beacon* will try to generate a MCR that produces a stacktrace where the crash message should contain the text "buffer size".

For this case there is actually two correct outputs, and we want to reproduce any of those. Listing 5 shows a code that would raise an error with the message "Buffer size, 4, is too small" whilst listing 6 should raise an exception stating that "Buffer size, 20, is not multiple of 16".

```
size = 4
numpy.setbufsize(size)

```

Listing 5. Example of using a buffer size that is too small

```
size = 20
numpy.setbufsize(size)

```

Listing 6. Example of using a buffer size that is not a multiple of 16

C. Wrong anchor on YAML object

The last experiment is the more complex one. In this case *Beacon* will try to reproduce a very specific error on *PyYAML*'s `yaml` module.

The expected result is a series of instructions that produces a `ScannerError` with the mesage containing the string "while scanning an anchor" and where the method `compose` should be contained in the stacktrace.

For this some considerations should be taken into account. Since *Beacon* searches for **functions** on a given module, it can't directly use classes. To overcome this issue, instead of directly running the script on the `yaml` module, a wrapper for the module with auxiliary functions to create and manage the objects was implemented in such a way that the proposed tool is executed over said module.

There are many ways of generating a `ScannerError`, but we are interested in one that is specifically caused by an invalid anchor on a *YAML* document. For this to happen the document must have a string that starts with "&" and doesn't point to a valid reference. Aditionally, we impose that the stack trace must contain a call to the function `compose`. Note that this conditions makes it very unlikely to randomly generate an input that causes this behaviour.

Knowing all this, it is easy to generate this error manually with the following line:

```
yaml.compose("&", yaml.SafeLoader)

```

Listing 7. Generating xd

VI. Experimental results

To evaluate the performance of *Beacon* various criteria were defined:

PC1: The time taken to reproduce an error.
PC2: The number of generations the algorithm needed to reproduce the error[4].
PC3: The number of instructions needed to reproduce the crash; we will call this *MCR length*.
PC4: The accuracy of the result measured by the exception type and, if provided, the MCR length, exception's message and a target method (see experiment V-C).

The tool was also evaluated, in terms of PC2 and PC4 for different population and individual sizes.

Experiment	PC1 (ms)	PC2	PC3 (lines)	PC4 (%)
V-A	105.31	4.09	2.06	93.75
V-B	149.04	3.97	1.13	100
V-C	9043.39	265.66	1.13	71.88

TABLE I
Results with fixed population and individual sizes

Table I show the average results of running each experiment 32 times with a population of 64 individuals, each one composed of 8 genes (statements). The results show very favorable results for experiments V-A and V-B, with low execution times and high accuracy. There's a clear increase on the difficulty of generating the MCR for the last experiment, which is the expected behaviour given that the target error is much more specific than the previous ones, but overall it can reproduce the crash around 70% of the time.

To see the effect of variations the same experiments where evaluated for different sizes of population and individuals, as

[4]With the purpose of reducing the execution time of the experiments the algorithm is terminated when it reaches 500 generations

seen on figures 1 through 6. It's also clear for this results that experiment V-C is the one with the worst preformance. There is not a clear pattern to determine which configuration is the optimal one, specially when comparing the results of using the same configurations across the three experiments. One aspect to note is that even though there's not a clear pattern, population size seems to affect positively the performance of *Beacon*, i.e. greater populations produce a better result. On the other hand, the effect on performance of the individual sizes is not clear; we can see that sizes between 10 and 14 seem to produce an overall good performance but the information is not enough to make a conclusion. For these results, all the results are the average values of running each experiment 32 times for each configuration.

Note that, given the non-deterministic nature of the algorithm, the results of every execution will produce slightly different results. Some examples results generated by the experiments are shown on listings 8-10

```
<class 'AttributeError'>
{
  "x0": "create_list(size_0 = 5, size_1 = 6)",
  "x1": "list_reduction(lst = x0)"
}
'NoneType' object has no attribute 'extend'

```

Listing 8. Example of a correct execution of experiment V-A

```
<class 'ValueError'>
{
  "x0": "ndim(a = 91)",
  "x1": "setbufize(size = x0)"
}
Buffer size, 0, is too small.

```

Listing 9. Example of a correct execution of experiment V-B

```
<class 'yaml.scanner.ScannerError'>
{
  "x0": "compose(stream = &,
    Loader = <class
    'yaml.cyaml.CSafeLoader'>)"
}
while scanning an anchor

```

Listing 10. Example of a correct execution of experiment V-C

VII. Conclusion

This paper presents the experience we have gained by building a simple technique to reproduce software failure. Two tools are presented to that aim, and three case studies shows the usefulness and benefices of our approach.

As a future work, we will improve the fitness function to be able to reproduce complex software failures.

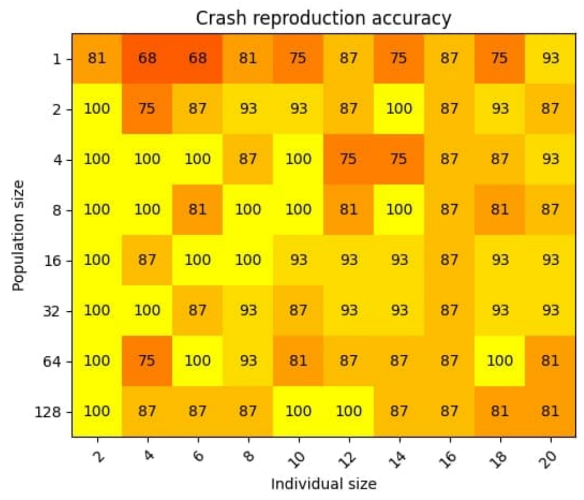

Fig. 1. Accuracy for experiment V-A with variable sizes of population and individuals

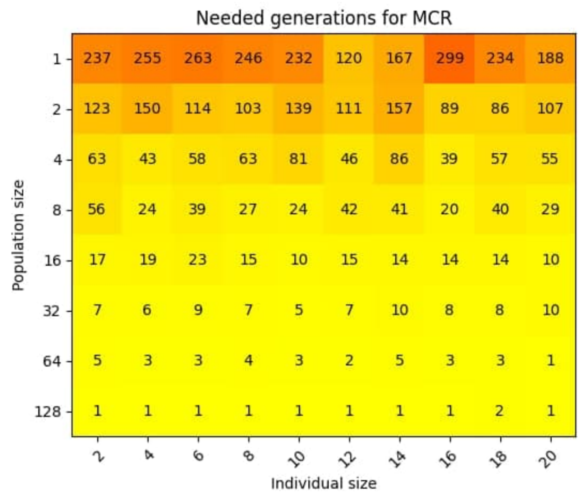

Fig. 2. Generations needed to reproduce the crash for experiment V-A with variable sizes of population and individuals

References

[1] D. E. Knuth, "Notes on the van emde boas construction of priority deques: An instructive use of recursion; private letter to peter van emde boas," 1977.

[2] A. Samarah, A. Habibi, S. Tahar, and N. Kharma, "Automated coverage directed test generation using a cell-based genetic algorithm," in *2006 IEEE International High Level Design Validation and Test Workshop*, 2006, pp. 19–26.

[3] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

[4] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 416–419. [Online]. Available: https://doi.org/10.1145/2025113.2025179

[5] J. Campos, Y. Ge, G. Fraser, M. Eler, and A. Arcuri, "An empirical evaluation of evolutionary algorithms for test suite generation," in *Search Based Software Engineering*, T. Menzies and J. Petke, Eds. Cham: Springer International Publishing, 2017, pp. 33–48.

[6] K. Moran, M. Linares-Vasquez, C. Bernal-Cardenas, C. Vendome, and D. Poshyvanyk, "Crashscope: A practical tool for automated testing of android applications," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 15–18.

[7] T. Roehm, S. Nosovic, and B. Bruegge, "Automated extraction of failure reproduction steps from user interaction traces," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 121–130.

[8] S. Li, J. Guo, M. Fan, J.-G. Lou, Q. Zheng, and T. Liu, "Automated bug

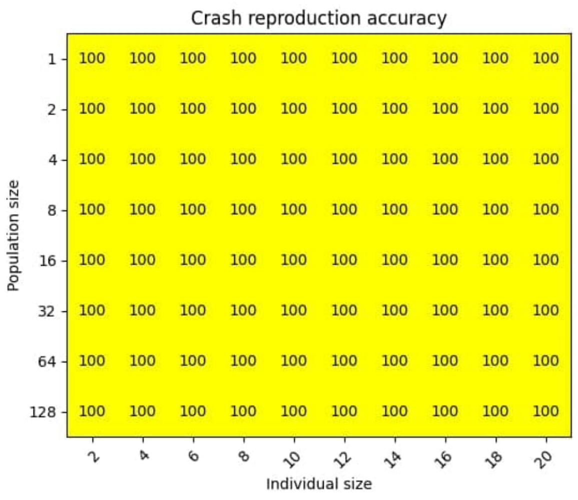

Fig. 3. Accuracy for experiment V-B with variable sizes of population and individuals

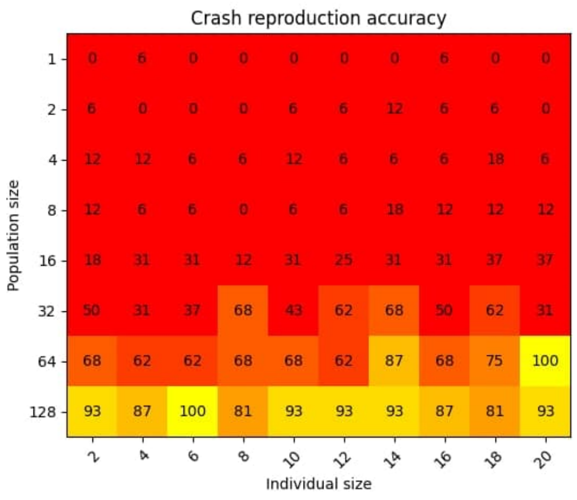

Fig. 5. Accuracy for experiment V-C with variable sizes of population and individuals

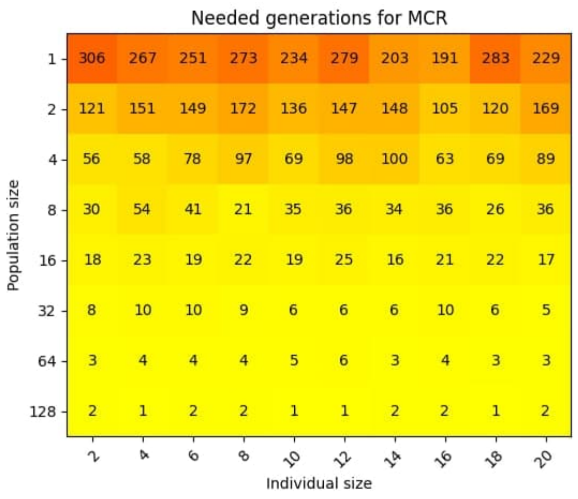

Fig. 4. Generations needed to reproduce the crash for experiment V-B with variable sizes of population and individuals

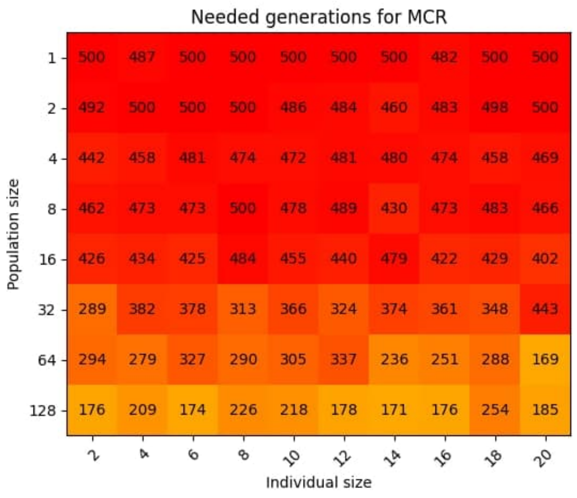

Fig. 6. Generations needed to reproduce the crash for experiment V-C with variable sizes of population and individuals

reproduction from user reviews for android applications," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 51–60. [Online]. Available: https://doi.org/10.1145/3377813.3381355

[9] N. Chen and S. Kim, "Star: Stack trace based automatic crash reproduction via symbolic execution," *IEEE Transactions on Software Engineering*, vol. 41, no. 2, pp. 198–220, 2015.

[10] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, and A. Larsson, "Jcharming: A bug reproduction approach using crash traces and directed model checking," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 101–110.

[11] J. Xuan, X. Xie, and M. Monperrus, "Crash reproduction via test case mutation: Let existing test cases help," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 910–913. [Online]. Available: https://doi.org/10.1145/2786805.2803206

[12] M. Soltani, A. Panichella, and A. van Deursen, "A guided genetic algorithm for automated crash reproduction," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 209–220.

[13] M. Soltani, P. Derakhshanfar, X. Devroey, and A. van Deursen, "A benchmark-based evaluation of search-based crash reproduction," *Empirical Software Engineering*, no. 25, 2020.

[14] C. S. Perone, "Pyevolve: A python open-source framework for genetic algorithms," *SIGEVOlution*, vol. 4, no. 1, p. 12–20, Nov. 2009. [Online]. Available: https://doi.org/10.1145/1656395.1656397

[15] F.-M. De Rainville, F.-A. Fortin, M.-A. Gardner, M. Parizeau, and C. Gagné, "Deap: A python framework for evolutionary algorithms," in *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*, ser. GECCO '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 85–92. [Online]. Available: https://doi.org/10.1145/2330784.2330799

[16] Open Source. Numpy. [Online]. Available: https://numpy.org/about/

[17] YAML Community. Pyyaml. [Online]. Available: https://pyyaml.org/wiki/PyYAML

[18] H. van Kemenade and R. Si, "hugovk/top-pypi-packages: Release 2021.01," Jan. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4409166