Byron: A Fuzzer for Turing-complete Test Programs

Byron: A Fuzzer for Turing-complete Test Programs

Dimitri Masetta
s306130@studenti.polito.it
Politecnico di Torino
Torino, Italy

Marco Sacchet
s295033@studenti.polito.it
Politecnico di Torino
Torino, Italy

Giovanni Squillero
giovanni.squillero@polito.it
Politecnico di Torino
Torino, Italy

Alberto Tonda
alberto.tonda@inrae.fr
UMR 518 MIA-PS
INRAE, Université Paris-Saclay
Palaiseau, France
Institut des Systèmes Complexes Paris-Ile-de-France
Paris, France

ABSTRACT

This paper describes Byron, an evolutionary fuzzer of assembly-language programs for the test and verification of programmable devices. Candidate solutions are internally encoded as typed, directed multigraphs, that is, graphs where multiple edges can connect the same pair of vertexes, with an added layer that defines the type of information vertexes can hold, and constraints the possible kinds of edges. Multiple genetic operators and a self-adaptation mechanism make the tool ready to tackle industrial problems.

CCS CONCEPTS

• **Theory of computation** → **Evolutionary algorithms**; • **Hardware** → *Test-pattern generation and fault simulation.*

KEYWORDS

Graph-based GP, Turing-complete GP, Fuzzing, Test-program, Assembly, Real-world application

ACM Reference Format:
Dimitri Masetta, Marco Sacchet, Giovanni Squillero, and Alberto Tonda. 2024. Byron: A Fuzzer for Turing-complete Test Programs. In *Genetic and Evolutionary Computation Conference (GECCO '24 Companion), July 14–18, 2024, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3638530.3664136

1 INTRODUCTION

Fuzz testing consists in creating pseudo-random sequences of input data with the goal of exposing problems or triggering undesired behaviors. In the past decades, *fuzzing* made it possible to detect thousands of bugs and vulnerabilities in various applications, and it is nowadays regarded as a necessary step in all security-related contexts [2, 3, 7, 8, 13].

The origin of fuzzing may be traced back to "a dark and stormy night" of 1990, when Barton Miller, Lars Fredriksen, and Bryan So sought an alternative to formal verification for assessing the reliability of Unix utilities [9]. Incidentally, in the very same year, John Koza wrote a 131-page technical report describing *Genetic Programming* (GP), his new "paradigm for genetically breeding populations of computer programs to solve problems" [6].

As fuzzing is based on simulation and random mutations, it is not surprising that evolutionary algorithms (EAs) have been successfully exploited for this task. Scholars reports smart approaches to create sensible inputs and to link the fitness value to the simulation, such as Zhu et al. [13] or Eberlein et al. [4]. The ability of GP to generate novel, original code by creatively combining and evolving structures has been used by Veggalam et al. [12] to generate high-level code for validating JavaScript interpreters.

Just like software engineers, electronic engineers designing microprocessors and microcontrollers need to check their work too. However, differently from most software products where specific data inputs may uncover bugs, programmable devices require specific *programs* to check their functionality. Such *test programs* do not perform human-recognizable functions: their specific purpose is to make problems apparent, either in the high-level design (verification) or in the physical device (test). Thus, such test programs need to be able to stress the different hardware features of the devices, such as the addressing modes, the registers, or the alternatives ways to call subroutines, low-level details that are usually masked by high-level languages.

It is thus apparent that, for example, fuzzing assembly-language test programs for hardware devices requires highly structured constraints. Assembly languages are complex and often non orthogonal, that is, only certain registers and operands may be involved in specific operations. While the fuzzer must be able to explore all possible corner cases of the language, it is essential to reduce the number of syntactically incorrect test programs, in order to avoid wasting computational time.

This paper describes *Byron*, a tool designed to be an effective fuzzer of assembly-language test programs. In Byron, the candidate solutions are encoded as typed, directed *multigraphs*, that is, graphs with an added layer of structure that defines the type of information each vertex can hold and where multiple edges can connect the

GECCO '24 Companion, July 14–18, 2024, Melbourne, VIC, Australia

ACM ISBN 979-8-4007-0495-6/24/07
https://doi.org/10.1145/3638530.3664136

same pair of vertexes. Byron is Free and Open Software (FOSS), and it is distributed under the permissive Apache 2.0 license[1].

The rest of paper is organized as follows: section 2 introduces Byron, detailing the internal encoding of individuals; section 3 describes the genetic operators, and the self-adaptive mechanisms; section 4 shows the results of preliminary experiments; section 5 concludes the paper.

2 BYRON

Byron is an optimizer based on an EA: as its main purpose is to generate Turing-complete assembly programs, it may be ascribed to the family of GP. Byron is a successor of the C++ tool MicroGP (μGP) [10], as they share the same goal and several key ideas. The tool, however, has been redesigned to maximize usability, flexibility, and expandability; and re-implemented from scratch in Python. Byron is available as a package from PyPi[2], while the development is hosted on GitHub[3].

In Byron, candidate solutions are encoded as typed, directed multigraph. The *type system* defines what data can be stored into vertexes and what the valid kind of edges are, while the *multigraph* data structure encodes the program's control and data flow.

2.1 The Multigraph

Individuals are encoded as a typed, directed multigraph G, that is an ordered tuple $G = (V, T, E, e_s, e_t, t)$, where V is a set of vertexes; T is a set of types; E is a set of edges; $e_s : E \to V$ and $e_t : E \to V$ are two functions that assign to each edge respectively its source and its target vertex; $t : V \to T$ is a function that associates each vertex with its type. Similarly to a programming language, vertexes in individuals can be seen as instances of specific classes, that is, the type of the vertex completely specifies the possible attributes of the vertex.

In more detail, a solution is encoded as a forest, that is a set of disjoint trees, with additional edges connecting leaves both inside the same tree or belonging to different trees (Figure 1a). Edges in the former set, that is the ones defining the forest, are called *structural*; referring to structural edges only, it is possible to use the standard terminology of *successor*/*predecessor* and *parents*/*descendants*. Edges in the latter set are called *references*, and are used to store parameters in macros (see section 2.2). The *structural successors* of a given vertex inside a tree are ordered, while vertexes connected through *reference edges* are not; however, each reference edge has its own unique label.

Each individual contains at least one tree; this special tree represents the standard entry point of the program, like the procedure *main* in some programming languages. Other trees are created when needed, according to the type system.

2.2 The Type System

The type of each vertex in a Byron multigraph is either a *macro* or a *frame*, and users can define their own different macros and frames.

A *macro* is the element used to generate programs: each macro represents a parametric fragment of code, that is, a fixed part composed of one or more lines of text with variable parameters, along with the specification of the type of parameters.

When the type of a vertex in the graph is a given macro, its attributes stores the parameters of the macro according to the definition. For instance, a simple assembly instruction may be represented by:

```
mov reg, imm16        (1)
```

in macro 1, the **reg** is a categorical attribute: a constant string taken from an unordered set of possible values, such as {ax, bx, cx, dx}; while **imm16** is a 16-bit integer, that is, a value between 0 and 65, 535. For example, the vertex could contain **reg** ← ax and **imm16** ← 42 as attributes.

References to other elements in the source code, such as the target of a *call* or the use of data at a specific memory location, are encoded by an edge in the multigraph. For instance, calling a subroutine may be represented by:

```
call proc             (2)
```

in macro 2, the attribute **proc** stores the target vertex. The frames make it possible to restrict the candidate targets according to the valid syntax, by classifying the types of vertexes. It is important to note that a macro may also specify a variable or a memory location.

Frames are like empty containers that provide a hierarchy. Vertexes of these types do not generate any code, but their purpose is to group their successors, assign them a label, and possibly enforce constraints. In version 0.8, Byron supports 3 types of frames:

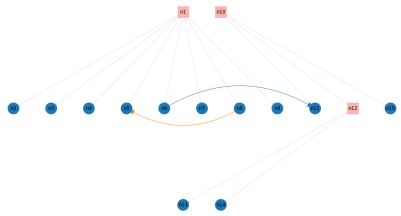
- **Macro bunch**. A vertex of this type is the predecessor of a variable number of vertices whose types are selected from a given set of macros. For example, the body of a subroutine may be a sequence of math instructions with two register operands and math instructions with one 16-bit immediate parameter. Both the minimum and the maximum number of successors of a macro bunch may by specified.
- **Frame sequence**. A vertex of this type is the the predecessor of a fixed number of vertexes of the types specified in a list, in the specified order. For instance, a subroutine may be described with a sequence of header, body, return statement, while the body may be described by a *macro bunch*.
- **Frame alternative**. A vertex of this type is the the predecessor of a single vertex whose type is randomly selected from a set of valid types.

Hence, the vertexes of type *macro* are the leaves of the trees composing the forest, while the vertexes of type *frame* are their internal nodes. The extra edges connecting the leaves are the parameters of type *reference* that can be found in the macros.

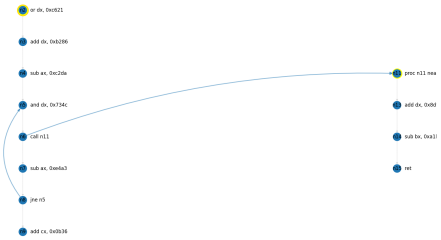
2.3 The Evaluator

As Byron is designed to be used as a test-program generator in industrial contexts, it allows for considerable flexibility during individual evaluation. Candidate solutions are written as text strings, and then evaluated by calling a user-defined Python function or by invoking a shell script that may use external proprietary tools. Different

[1] https://cad-polito-it.github.io/byron/
[2] https://pypi.org/project/byron/
[3] https://github.com/cad-polito-it/byron



(a) Internal representation of a candidate solution as a multigraph. Square vertexes are frames, round vertexes are macros. The frame representing the subroutine *n10* is a sequence containing the header *n11*, that is the target of the reference from *n6*, the macro bunch *n12*, and the return statement *n15*.



(b) Same candidate solution, seen as a linear GP representation.

Figure 1: Candidate solution containing 12 macro nodes and 2 references.

types of parallelization are supported out of the box, from simple multithreading to the creation of temporary directories where multiple subprocesses are concurrently spawned.

The individual is linearized into a text file by writing all the leaves, that is, all the nodes of type *macro*, following a depth-first visit from the root of the trees in the forest. Visualizing only the macros, in the correct order, linked by arbitrary edges, generates a result resembling an individual generated with Linear GP [1] (Figure 1b); however, while this view may be user-friendly for engineers, it ignores a part of the internal structure and might impair the interpretability of the constraints.

3 GENETIC OPERATORS

Byron v0.8 supports both mutation and crossover operators.

3.1 Mutation Operators

- **Single parameter mutation**: generates a new individual by randomly modifying a single, randomly chosen, parameter of a random vertex of the parent. Byron also implements specific versions of this operator to handle specific data types, such as arrays.
- **Add macro to bunch mutation**: generates a new individual by adding random vertexes as successors of a randomly selected macro bunch of the parent, respecting the constraints.
- **Remove macro from bunch mutation**: generates a new individual by removing random vertexes from the successors of a randomly selected macro bunch of the parent, respecting the constraints.

3.2 Crossover Operators

- **Array parameters uniform crossover**: generates a new individual by performing a uniform crossover between two randomly selected compatible arrays inside the two parents.
- **Node crossover**: generates a new individual by swapping randomly selected compatible nodes between the parents. Several versions of this operator have been implemented, differing on how the nodes are considered to be *compatible*.
- **Leaf crossover**: generates a new individual by swapping compatible leaves. Several versions of this operator have been implemented, differing on how the leaves are considered to be *compatible*.

3.3 Self Adaptation

Byron is a self-adaptive evolutionary algorithm, attempting to tweak its internal parameter during the execution to maximize efficiency. The *temperature* is used to determine, in the case of a fitness target, the proximity of the best individual in the population to the desired value. The temperature influences the *operators' strength* $\sigma \in [0, 1]$, that is, a qualitative indication of how much parent and offspring should differ after a mutation (e.g., the average number n_f of bits flipped in an array by a mutation is $n_f = (1 - \sigma)^{-1}$ when $\sigma < 1$, while a new random array is generated from scratch when $\sigma = 1$). Plainly, a higher strength pushes towards exploration, while a lower ones push towards exploitation.

Moreover, as different genetic operators are available, it is useful to detect under-performing operators and use them more sparingly, while applying more often the ones that deliver the best individuals. Byron keeps track of the performance of every operator, recording:

- Number of calls to the operator: how many time the operator was used
- Number of complete failures of the operator, i.e., every time no valid offspring has been generated (abort)
- How many valid offspring the operator has created in total
- Number of new individuals whose fitness value was worse than the fitness values of both parents (failure)
- How many new individuals have a better fitness than at least one of their parents (success)

To measure the performance, operators receive a reward r_1 for every success and a reward r_2 for every valid offspring, with $r_1 > r_2$. No explicit penalization is given for failures and aborts, since the lack of rewards is a form of penalization *per se*.

The concept of *regret* was introduced by Slivkins et al. [11]. Let $\mu(o_p)$ be the mean reward of o_p, where $o_p \in O$, with O being the set of available operators, with cardinality $|O| = K$. It is possible to define $\bar{\mu} = \max(\mu(o_p) : \forall o_p \in O)$ as the best mean reward. Randomly selecting T times an operator from O and using it, it is possible to compute the regret

$$R(T) = \bar{\mu} \cdot T - \sum_{t=1}^{T} o_{p_t} \tag{3}$$

This makes it possible to compare the cumulative reward obtained by the algorithm against the reward obtained by using a hypothetical optimal strategy.

To deactivate the under-performing operators, Byron adopts the *Successive Elimination Algorithm* by Even-Dar et al. [5] with regret $R(T) \leq O(\sqrt{KT \log T})$; a *confidence radius* of $r_t(o_p) = \sqrt{\frac{2 \log T}{n_t(o_p)}}$, where $n_t(o_p)$ is the number of times an operator has been selected.

The *Confidence Interval* $[\mu_t(o_p) - r_t(o_p), \mu_t(o_p) + r_t(o_p), U_t(o_p)]$ represents the interval where it is assumed that the mean reward of an operator will fall.

If, at a given time, the best operators start to fail, the overall L margin will decrease. Therefore, other operators previously discarded could be considered valid again. The ratio is the following: if too many failures start to suddenly appear, the active operators may be not suited to the current structure of the genome anymore, and previously discarded operators could become more successful. In order to try to anticipate this kind of event, every quarter of execution, every operator is tested, regardless if it is active or not.

3.4 Aging

By default Byron adopts a steady-state strategy ($\mu + \lambda$), that is, offspring and parents compete for survival. However, this behavior can be tweaked using the *age* parameter: all individuals die at the age of A_T, but the best A_B individuals do not age. Tuning A_T and A_B can provide the full spectrum of possible behaviors: either $A_T = \infty$ or $A_B = \infty$, corresponding to a pure steady-state approach; $A_T = 1$ and $A_B = 0$ corresponds to a pure generational approach, (μ, λ); while $A_T = 1$ and $A_B = 1$, to a classical *elitist* strategy.

4 EXPERIMENTAL EVALUATION

As Byron is still in alpha and under active development, it has not been used in any commercial projects, yet. It has, however, been tested on benchmarks, loosely based on the classical One Max problem. Details are available in the GitHub used for the development[4].

- **Classical One Max**: The classical test problem, with the individual defined as a frame of type macro bunch with exactly N macros, each one with a single parameter encoding 1 bit; alternatively, with exactly N/m macros, each one with a single parameter encoding an array of m bits. Individuals are evaluated by a pure Python function, with the best possible individual being $\{1, 1, ..., 1\}$.
- **Assembly One Max (desktop)**: Byron is asked to generate an assembly program that set all the bits of a given register to 1, both using a 64-bit version of the x86 assembly and the newer ARM 64-bit assembly. Individuals are evaluated on the host computer by assembling and linking the code, and eventually executing it, respectively, on an Intel i7 CPU (Windows) and on an Apple M1 CPU (OSX).
- **Assembly One Max (simulator)**: like in the previous bench, Byron is asked to generate an assembly program that set all the bits of a given register to 1. Individuals are evaluated by calling an architectural simulator of the RISC-V ISA[5].
- **One Max Go**: Byron is asked to generate a high-level Go[6] function using sub functions, global and local variables, that returns precisely 18, 446, 744, 073, 709, 551, 615, that is, an unsigned 64-bit integer with all bits set at 1. Individuals are evaluated by compiling and running the code on the host computer.

The tool managed to succeed in all these simple benchmarks, finding the optimal solution. Indeed, performances where not relevant, as the purpose of the evaluation was to check the ability to handle different structures, from a flat bit string to a complex high-level program; to generate different types of assembly languages, the IA64, ARM64, and RISC-V; and to test different types of evaluators, from a Python function to an external simulator, on different operating systems.

5 CONCLUSIONS

This short paper described Byron, a GP-based tool designed to be an efficient assembly-language fuzzer, which can be used for test-program generation by electronic engineers designing, verifying, or testing programmable devices such as microprocessors and microcontrollers.

REFERENCES

[1] 2007. *Linear Genetic Programming.* Springer US, Boston, MA. https://doi.org/10.1007/978-0-387-31030-5

[2] Marcel Boehme, Cristian Cadar, and Abhik ROYCHOUDHURY. 2021. Fuzzing: Challenges and Reflections. *IEEE Software* 38, 3 (May 2021), 79–86. https://doi.org/10.1109/MS.2020.3016773

[3] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. 2018. A Systematic Review of Fuzzing Techniques. *Computers & Security* 75 (June 2018), 118–137. https://doi.org/10.1016/j.cose.2018.02.002

[4] Martin Eberlein, Yannic Noller, Thomas Vogel, and Lars Grunske. 2020. Evolutionary Grammar-Based Fuzzing. In *Search-Based Software Engineering*, Aldeida Aleti and Annibale Panichella (Eds.). Springer International Publishing, Cham, 105–120. https://doi.org/10.1007/978-3-030-59762-7_8

[5] Eyal Even-Dar, Shie Mannor, and Yishay Mansour. 2002. PAC bounds for multi-armed bandit and Markov decision processes. In *Computational Learning Theory: 15th Annual Conference on Computational Learning Theory, COLT 2002 Sydney, Australia, July 8–10, 2002 Proceedings 15.* Springer, 255–270.

[6] John R Koza. 1990. *Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems.* Vol. 34. Stanford University, Department of Computer Science Stanford, CA.

[7] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: A Survey. *Cybersecurity* 1, 1 (June 2018), 6. https://doi.org/10.1186/s42400-018-0002-y

[8] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the Art. *IEEE Transactions on Reliability* 67, 3 (Sept. 2018), 1199–1218. https://doi.org/10.1109/TR.2018.2834476

[9] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44. https://doi.org/10.1145/96267.96279

[10] Ernesto Sanchez, Massimiliano Schillaci, and Giovanni Squillero. 2011. *Evolutionary Optimization: The µGP Toolkit.* Springer Science & Business Media.

[11] Aleksandrs Slivkins et al. 2019. Introduction to multi-armed bandits. *Foundations and Trends® in Machine Learning* 12, 1-2 (2019), 1–286.

[12] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. IFuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming. In *Computer Security – ESORICS 2016*, Ioannis Askoxylakis, Sotiris Ioannidis, Sokratis Katsikas, and Catherine Meadows (Eds.). Springer International Publishing, Cham, 581–601. https://doi.org/10.1007/978-3-319-45744-4_29

[13] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *Comput. Surveys* 54, 11s (Sept. 2022), 230:1–230:36. https://doi.org/10.1145/3512345

[4] https://github.com/cad-polito-it/byron/tree/alpha/examples/onemax

[5] An open standard based on established reduced instruction set computer (RISC) principles. See https://en.wikipedia.org/wiki/RISC-V

[6] A statically typed, compiled high-level programming language designed at Google , see https://go.dev/