OCELOT: A Search-Based Test-Data Generation Tool for C

Simone Scalabrino  
University of Molise  
Pesche, Italy

Giovanni Grano  
University of Zurich  
Zurich, Switzerland

Michele Guerra  
University of Molise  
Pesche, Italy

Andrea De Lucia  
University of Salerno  
Salerno, Italy

Dario Di Nucci  
Vrije Universiteit Brussel  
Brussels, Belgium

Harald C. Gall  
University of Zurich  
Zurich, Switzerland

Rocco Oliveto  
University of Molise  
Pesche, Italy

ABSTRACT

Automatically generating test cases plays an important role to reduce the time spent by developers during the testing phase. In last years, several approaches have been proposed to tackle such a problem: amongst others, search-based techniques have been shown to be particularly promising. In this paper we describe OCELOT, a search-based tool for the automatic generation of test cases in C. OCELOT allows practitioners to write skeletons of test cases for their programs and researchers to easily implement and experiment new approaches for automatic test-data generation. We show that OCELOT achieves a higher coverage compared to a competitive tool in 81% of the cases. OCELOT is publicly available to support both researchers and practitioners.

CCS CONCEPTS
• Software and its engineering → Maintaining software; Search-based software engineering.

KEYWORDS
Automated Testing, Test Case Generation, Search-Based Software Engineering

ACM Reference Format:

1 INTRODUCTION

Software testing —and, in particular, test case writing— is one of the most human-intensive and time-consuming activities in software development life-cycle [1]. Therefore, noticeable effort has been devoted toward automated test-data generation, applying and possibly combining approaches like random testing, symbolic, and concolic execution [3, 7, 10, 16, 20]. Search-based approaches have also been proved to be particularly well-suited [8, 13].

While many search-based tools are nowadays available for Java, some of which very mature and used in practice (e.g., EvoSuite [6]), the same is not true for C. Indeed, the tools developed in the past, like TESTGEN [5], QUEST [2], and GADGET [15], are quite dated and not available (neither as executable binaries). To the best of our knowledge, the most recent search-based tools for test-data generation in C are: i) IGUANA, proposed by McMinn [14], developed in Java and relying on JNI to interface with C; ii) AUSTIN, introduced by Lakhotia et al. [12], developed in OCaml and based on the CIL framework; iii) CAVM, recently proposed by Kim et al. [9], developed in C, C++, and Python and based on CLang and gcc for source code instrumentation. Considering that both IGUANA and CAVM are not publicly released, the only available tool for test-data generation in C is AUSTIN. However, despite being open source, this project is not active anymore. In summary, C remains a widely used language (ranked 2nd in the TIOBE Index for May 2018[3]) and, for this reason, we believe that the search-based research community should put some effort into actively developing test-data generation tool for this programming language.

In this paper, we take a step in such a direction and we present OCELOT, a search-based test-data generation tool for C that provides an extensible framework to quickly implement different types of search-based approaches. OCELOT is also able to automatically write test suites, relying on the LibCheck framework. We empirically evaluate OCELOT with respect to AUSTIN on 26 functions from 3 open-source C programs, and we show that OCELOT achieves a higher branch coverage in 81% of the cases. OCELOT is publicly available along with its documentation.

2 BACKGROUND

In the context of white-box coverage-driven testing, the automatic test-data generation consists of the generation of a (possibly minimal) set of inputs (TD) for a given target program P to reach a desired amount of code coverage (e.g., branch coverage). To achieve this goal, a search-based test-data generator (TDC) uses a search algorithm, calling P with many combinations of inputs (test data),
with the aim of maximizing the total achieved coverage. This process continues until the specified search-budget (e.g., number of iterations) is over. Therefore, generating test data requires a continuous interaction between the target program and the test-data generator. When TDG calls $P$ with a given input, $P$ has to provide feedback about the branches covered with that input to TDG.

Since $C$ is a procedural programming language, it is possible to consider functions as the units to test ($P$). This makes test generation conceptually different compared to object-oriented languages (e.g., Java), where the units to test are classes with a state [21]. Furthermore, automatically generating test data for $C$ presents unique technical challenges. The main technical challenge ($C_1$) is the instrumentation of $P$, necessary to register what happens in $P$ when TDG tests a specific input. Indeed, the object language of $C$ depends on the target architecture and it is necessary to find a way to instrument $P$, without tying to a specific architecture/operating system, to grant a certain level of portability. A second technical challenge ($C_2$) regards the integration between TDG and $P^*$ (i.e., the instrumented $P$). It is necessary to devise a mechanism to i) make the input generated by TDG suitable for $P^*$ and ii) to make $P^*$ able to communicate with TDG. A final major research and technical challenge ($C_3$) regards one of the main peculiarities of $C$, i.e., pointers. Consider a function that takes as input $(\text{int}^* a, \text{int} b)$. At some point in the function, there is a condition $1 \text{f } (a == b)$. A test-data generator should be able to generate test data that evaluate such a condition as $\text{true}$. While Java has a more rigid and implicit typing rule (i.e., objects are stored as references, primitive types are stores by values), in $C$ it is possible to have mixed situations, that makes handling data types in $TDG$ more difficult.

3 OCELOT

The architecture of OCELOT is inspired by IGUANA [14]: it is developed in Java and uses JNI as an interface with the target program to try different combinations of test data. Running OCELOT consists of two distinct macro-phases: build and run. Figure 1 shows the phases of test-data generation in OCELOT. In the build phase, the target program is instrumented, wrapped in a JNI library and then compiled. The output of such a phase is a static library. This library is linked by OCELOT and used in the run phase, where a search-based algorithm is exploited to identify a set of inputs that maximize the code coverage. The output of this phase is a set of test data that can be used to test the target program. OCELOT also provides two additional optional phases: minimize and write. During the minimize phase, test data that do not contribute to improve the code coverage are removed, while during the write phase, skeletons of actual test cases are written. Such skeletons lack the assertions, that should be manually added, and they are based on the LibCheck framework.

3.1 Build Phase

The build phase consists of: i) instrumenting the target program (i.e., transforming $P$ to $P^*$), ii) wrapping it in the JNI library, iii) generating the makefile, and iv) compiling the static library. OCELOT directly instruments the $C$ source code, making it able to potentially work regardless of the target architecture. Specifically, it adds probes near the control structures to track the exact steps executed in $P$. Existant tools (e.g., gcov or lcov) are not used, since they do not report the runtime values of the variables, needed to compute part of the fitness function for some approaches. During the wrapping sub-phase, OCELOT adds to the target function a set of pre-defined $C$ functions to integrate $P^*$ in OCELOT through JNI. Moreover, it adapts some of those files to handle the specific inputs required by $P$. Indeed, while the static Java interface to call $P$ remains the same, each target function has its own input types. The wrapping sub-phase handles the information that will be passed by OCELOT during the run phase and translate it into actual $C$ variables that will be passed to $P^*$ for the execution. Finally, OCELOT generates the makefile (depending on the operating system and on the special requirements of the target program) and it compiles the JNI library.

3.2 Run Phase

The run phase consists of: i) generating inputs for $P$, ii) calling the JNI interface to execute the code, iii) grabbing the execution events and simulating them on the Control Flow Graph of $P$, and iv) evaluating the inputs and giving feedback to generate other inputs. The input generation and the feedback mechanism strongly depend on the specific test-data generation strategy used. For example, a strategy based on random search would randomly generate inputs and ignore the evaluation, adding all the generated test data to the resulting test suite. On the other hand, the execution and simulating sub-phases are common to all the strategies. After the execution of $P^*$, a list of events (i.e., the actual evaluations of conditions and fitness function) is generated. Such a list is used to simulate the events on the Control Flow Graph of $P$ to easily compute the fitness function necessary to give feedback to the strategy.

3.3 Features

OCELOT provides some additional features compared to the state-of-the-art tools. First of all, it implements different search-based approaches and it is easily extensible. Indeed, OCELOT is not tied to...
any specific approach. It implements LIPS [19], MOSA [18], AVM [11], and the random approach [17] while, for instance, CAVM [9] is only based on AVM [11], while AUSTIN implements AVM and AVM+ [12]. Furthermore, it supports different budget-handling and minimization strategies that can be combined together with the aforementioned algorithms. Thus, Ocelot is not designed to work only with a specific type of algorithm and, therefore, it is easily extensible with new test-data generation strategies. With respect to other tools (e.g., AUSTIN), Ocelot is based on Eclipse CDT\(^7\). Hence, it is able to automatically detect the data types of the parameters and to fully instrument code.

Ocelot strongly relies on a configuration file, in which all the parameters for all the phases have to be specified. For example, this file contains the maximum number of evaluations, the desired target coverage, the name of the target function, the path to the file with the target function, and so on. Since there are many parameters that can be set, we also developed a user interface for Ocelot, implemented in a CLion\(^8\) plugin. This plugin allows developers to easily use Ocelot for their projects. While, as previously stated, the command line allows to make experiments, the CLion plugin is focused on practitioners and, therefore, it supports by default the write mode able to generate skeletons of test cases. It adds a configuration page for Ocelot in the general configuration of CLion. Moreover, it introduces an item in the “Code” menu (i.e., “Generate TC”). When clicking on this item, the plugin shows a window with all the functions of the current files for which the developer could generate test-data. Finally, the plugin automatically recognizes the signature of each function, allowing the users to specify, for each parameter, the ranges of possible values. Figure 2 shows the Ocelot configuration window in CLion.

4 EVALUATION

In this section we report a comparison between Ocelot and AUSTIN, the only available search-based tool for automatically generating test-data in C, to the best of our knowledge. We were not able to compare Ocelot to CAVM or IGUANA, since these tools are not publicly available. Note that our goal was to compare Ocelot with competitive search-based tools, so we did not consider test data generation tools for C based on other approaches, such as CUTE [20] and DART [7]. This comparison will be part of future work.

As previously shown in Section 3, Ocelot provides a broader number of features compared to other state-of-the-art tools. Moreover, it is more flexible and easily extensible. Amongst the others, Ocelot allows to generate tests relying on different meta-heuristics. Being the goal of the evaluation to check whether Ocelot is a good alternative to AUSTIN, we set Ocelot to run with AVM, i.e., the search-algorithm employed by AUSTIN.

With our study we aim to compare out-of-the-box the performance of these tools. It is worth noting that AUSTIN requires an explicit pointer constraints in the source code of the target function to instantiate any pointer. Thus, a pointer will not be instantiated if the code does not compare it to NULL. We argue that this an important limitation for the usability of the tool. Therefore, as done in previous work [9], we decide to not manually modify the subjects to evaluate the tools in a real-world scenario.

To compare the tools, we run both of them on a subset of 26 functions coming from 3 different C open source projects: GIMP, GLS, and SGLIB. GIMP is the open source GNU image manipulation software; GLS stands for GNU Scientific Library; SGLIB is a library offering generic utilities. It is worth noting that this set of functions has been used both in our previous work and in the paper that introduced AUSTIN [12, 19]. Table 1 lists the set of functions, representing the context of the comparison. In total, we take into account 400 branches for 26 functions.

As done in previous work [9], we set 1000 evaluations for each branch of the function under test as search budget. Moreover, for the same reason, we do not impose any time limit constraints.

We run the tools 20 times for each function; we then compare the achieved branch coverage relying on the non-parametric Wilcoxon Rank Sum Test [4] with significance level \( \alpha = 0.05 \). Significant \( p \)-values allows us to reject the null hypothesis, i.e., that the two tools achieve the same coverage. Moreover, we rely on the Vargha-Delaney (\( \hat{A}_{12} \)) statistic [22] to estimate the effect size, i.e., the magnitude of the difference between the measured metrics. \( \hat{A}_{12} \approx 0.5 \) means that Ocelot reaches a higher coverage than AUSTIN, while \( \hat{A}_{12} > 0.5 \) has the opposite meaning. Vargha-Delaney statistics also classifies such effect size into four different levels: negligible; small; medium; large [22]. For the experiments, we execute both the tools on a virtual machine with 8 cores and 32GB RAM, running Ubuntu 16.04 LTS. We make available two docker images on DockerHub—one for each tool—to foster the replicability of the results\(^9,10\).

4.1 Results

Table 1 summarizes the branch coverage results with the comparison between AUSTIN and Ocelot. The table shows the average coverage achieved over 20 runs along with the \( p \)-values obtained with the Wilcoxon test and the Vargha-Delaney statistic. We type-set the statistically significantly results (i.e., \( p \)-value \( \leq 0.05 \)) in bold, highlighting the tool reaching the highest coverage. From Table 1 we notice that Ocelot performs better than AUSTIN in 21 out 26 cases (\( \approx 81\%)\), while the opposite happens in only one case. In detail, the coverage improvement ranges from a minimum of 43%.

\(7\) https://www.eclipse.org/cdt/
\(8\) https://www.jetbrains.com/clion/
\(9\) https://hub.docker.com/r/giograno/austin/
\(10\) https://hub.docker.com/r/giograno/ocelot/
to a maximum of 100% for the function (e.g., gimp_cmyk_to_rgb). Indeed, in each of the aforementioned cases, the effect size is large.

Qualitatively looking at the results, we can see that Austin stops at 0% of coverage for the subjects having only pointers as arguments (e.g., gimp_hsl_to_rgb). Furthermore, even in case of functions with only primitive values as parameters, Ocelot is able to reach a higher coverage than Austin. For example, on gradient_calc_radial_factor Austin constantly achieves 50% of coverage, while Ocelot hits ≈ 93% on average with a 0.13 standard deviation.

Even if we do not impose any time limit constraints as search budget, it is still worth to discuss this dimension when it comes to compare the two tools. In particular, during our experiment we noticed that, given the same maximum amount of iterations, Ocelot is way faster than Austin. Just to give an idea, the evolutionary search for the function sglib_int_array_quick_sort takes ≈ 6 seconds for Ocelot, while it takes ≈ 4 minutes for Austin.

5 CONCLUSION AND FUTURE WORK

In this paper we presented Ocelot, a search-based tool for automatically generating test-data for C programs. We released the executable of Ocelot, describing the configuration parameters that can be set, to allow both researchers and practitioners to use the tool. Our preliminary study shows that Ocelot is able to achieve higher coverage than Austin. Moreover, Ocelot presents some additional advantages over Austin: i) it is easier to use: on the one hand Austin requires manual code-changes to instantiate pointers or to specify input preconditions; on the other hand Ocelot automatically handles those cases; ii) it is written in Java, a more common programming language compared to OCaml, and it is actively maintained; iii) it offers a CLion plugin to help the practitioners to configure the tool; iv) it supports different search algorithms, generation and minimization strategies that can be combined together; v) it is more effective and efficient.

Future work will be aimed at improving Ocelot in several aspects. We plan to implement a different representation for the test input, to support recursive data structures. We also plan to implement a new instrumentation process with the aim of reducing the time needed for the instrumentation of multiple target functions: this step should improve the scalability of the tool. Finally, we plan to release the code of tool as open source in the near future.

REFERENCES