Intelligent Code Generation/ Mutation to aid fuzzing of JavaScript engines

Masters Thesis

Amit Andre Menezes
xxxxxxxx

MSc Artificial Intelligence

Spring 2022
Date: 15/05/2022

Supervising Professors:

Prof. Vassil Vassilev

Prof. Karim Ouazzane

School of Computing and Digital Media
London Metropolitan University
166-220 Holloway Road, London, N7 8DB

Abstract

Advances in gray box coverage based fuzzing methods and JavaScript fuzz testing have introduced several JavaScript fuzzers for browser bug hunting. However, existing fuzzers only prioritise the growing of programs in a corpus on coverage discovery. In contrast, there is a lack of guidance to the code generation and mutation tasks of a fuzzer that indirectly influence the growth of programs for the corpus. Recent artificial intelligence research proposes many techniques for heuristic-based decision making. This has led to identifiable similarities in coverage guidance fuzzing and reward-based decision making. Improvements in the decision making of the components in a fuzzer can lead to optimal task selection and potential improvement in coverage discovery. This work attempts to use coverage as a guiding mechanism for potential improvement in fuzzer decision making. The guidance is achieved through intelligence in task selection for fuzzing and is hypothesized to potentially optimal growth in coverage.

To verify this hypothesis potential candidate JavaScript fuzzers were surveyed, along with potential reward-based algorithms to implement the guidance mechanism. The result of the survey selected the fuzzer Fuzzilli and the multi-armed bandit algorithm (MAB) series - Exploration and Exploitation (Exp3). This work implements the integration of the MAB algorithm Exp3 and Exp3.1 with Fuzzilli. The performance of this experimental approach is evaluated, and its optimisation results are demonstrated.

Acknowledgements

I would like to express my gratitude to London Metropolitan University and my supervisors Prof. Vassil Vassilev and Prof. Karim Ouazzane for their support and guidance while performing this project.

I would also like to thank my study skills tutor Esther Efemini for her support work and motivating help during my Masters Course.

Table of Contents

Table of Figures

Table of Graphs

1.Introduction

Web browsers over the years, has evolved to become an essential requirement in performing every-day activities in society. Thus, it is paramount to make them as secure as possible before they are ready for production and dissemination to the public. As a result, a lot of research is focused on techniques to discover vulnerabilities, that includes static testing [1], symbolic execution [2], and dynamic analysis [3].

A predominant insecurity prevalent in browsers is through vulnerabilities introduced via memory leaks, buffer overflows and other logic bugs. It is not often enough to enforce uniformity in secure development practices, when working with many browser components. The first primary pre-emptive action against exploitable vulnerabilities in browsers is achieved through discovery via code review and crash reports. Various test suites are created to analyse potential outcomes of the output generated by components of the browser. However, due to the diversity of modular components in a browser, it is an arduous task to exhaustively eliminate all the vulnerable components in a browser. It is especially difficult to pinpoint when vulnerabilities can be triggered through specific conditions spanning over multiple components. Moreover, the complexity of the components makes it harder for human perception to discover flaws through code analysis; inevitably it leads to lapses even when adopting good coding practices [4]. Therefore, there continues to be re-emergence of old vulnerabilities [5] as newly discovered ones [6] that attackers use for exploitation.

A more thorough approach to discovering vulnerabilities is via Fuzz-testing or fuzzing. In the classical sense, it is a testing technique that specialises in automated discovery of vulnerabilities. The basic strategy is to use variant randomly generated inputs to a target to force it into performing unexpected behaviour. This behaviour exposes faulty conditions at runtime such as memory access violations. As a result, it reveals security flaws in the target that have potential for exploitation. Fuzzing web browser components is an effective way of identifying bugs that can lead to security flaws. The Renderer and Java Script engines within browsers are particularly good target for bug discovery.

Gray-box fuzzing, or coverage-based fuzzing is an optimised approach to fuzzing that has been very promising recently. Various JavaScript and DOM (Document Object Model) fuzzers exist with heuristics and decision-making parameters that are used to guide the direction fuzzing takes. The predominant metric being coverage i.e. The depth reached in the fuzzing target's control flow graph of execution. Coverage aids in directing the fuzzer to seek unexplored code paths exhaustively and trigger difficult to reach edge cases. Most JavaScript and DOM fuzzers employ a corpus of executable programs to generate coverage data. The corpus is grown through evolving programs, this is made possible through the manipulative components of the fuzzer such as code generation and mutations. The choice of programs that end up in the corpus are guided by yield of higher coverage, however this also means that there are programs which are discarded due to not satisfying the coverage requirements. In addition, the semantic correctness of a program must be conserved during the process of growing the corpus.

In recent fuzzers coverage guidance is a heuristic that only guides the growth of corpora but is unable to guide the manipulative components of the fuzzer towards efficiently generating high yielding coverage.

This paper attempts to use coverage guidance in directing the components of a fuzzer code generation, mutation, and seed selection to make intelligent choices while performing fuzzing for faster performance.

1.1. Aim and Objectives

The aim of this project is to implement an intelligent selection mechanism for application of Code Generation/ Mutation tasks to aid fuzzing JavaScript engines and increase performance through higher and faster coverage growth during program execution and preserving the program's semantic correctness.

The objectives for this project are to

1. Identifying a suitable JavaScript Fuzzer
2. Evaluate how Code/Mutation/Seed selection are currently implemented
3. Propose and implement an intelligent selection algorithm that can aid in improved performance
4. Evaluate the results of the fuzzers performance with and without the intelligent selection.

1.2. Methodology

The paper hypothesises an experimental modification to an established JavaScript fuzzer (Fuzzilli) selected after qualitative analysis of JavaScript fuzzing and similar characteristic fuzzers that address optimality in Grey-box coverage-based fuzzing. The performance of this experimental modification is evaluated against the original fuzzer implementation targeting prominent JavaScript Engines that are used in popular web browsers.

The literature review section of this paper identifies key features of JavaScript fuzzing from the perspective of JavaScript language and the JavaScript engines that execute the programs. It then focusses on the existing JavaScript fuzzers and their approaches toward JavaScript fuzzing with a primary focus on Grey-box coverage-based fuzzing. With the acquired information, the rationale for choosing Fuzzilli as a fuzzer for introducing proposed enhancements is established and key decision points for introducing guided intelligence are identified. Various algorithms are then reviewed for potential use in enhancing coverage-based fuzzing via these decision points and the justification of the use of the Multi-Armed Bandit (MAB) algorithm is established.

The next two sections describe Fuzzilli's architecture and the environment for development. The background section provides an overview of Fuzzilli, featuring its key fuzzing components i.e., code generators, mutators, the program corpus and distributed fuzzing. These components are elaborated upon to give context for proposed modifications to the fuzzer including future possibility of enhancement. The Proposed modifications sections detail a high-level overview for changes to the original fuzzer

The Implementation section describes the formulation of a feedback reward mechanism based of coverage discovery to guide the internal fuzzer components and the constructed algorithms to help guide the component selection.

The performance of the implementation of the new modified system against the original fuzzer is evaluated in the evaluation section. This is evaluated against the two most prominent browsers Apple Safari and Google Chrome's JavaScript engines JavaScriptCore (JSC) and V8, respectively.

Finally, discussions of the results and the future development topics are addressed.

1.3. Project Schedule

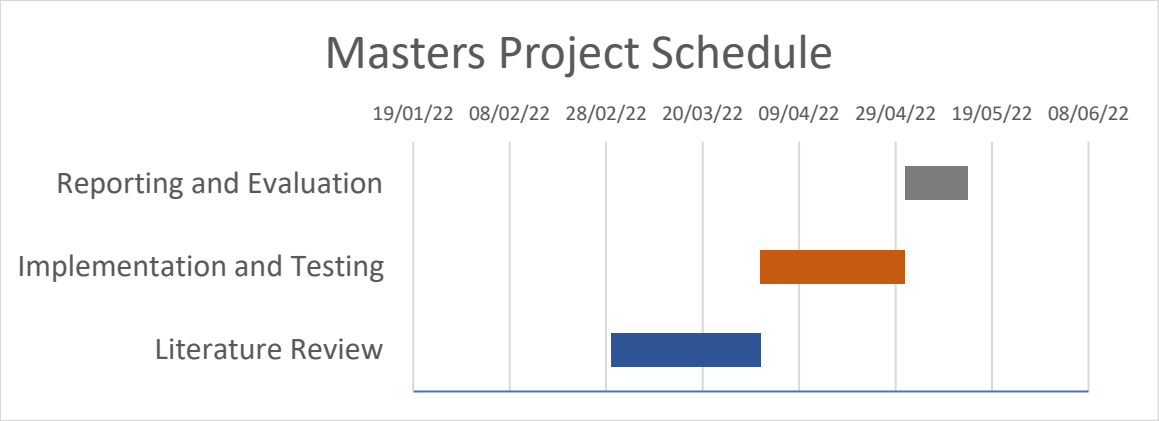


Figure 1: Masters Project Schedule

This project was conducted over a period of 3 months. The first month was spent in identifying JavaScript fuzzers and their architectures. Prominent fuzzers where identified based on success in identifying browser bugs and capabilities of generating and mutating JavaScript for fuzzing. The pool of fuzzers was narrowed down to two fuzzers that needed intelligence to guide fuzzing tasks. Ultimately it was reduced to one (Fuzzilli) after identifying the uniqueness in preservation of semantic and ease of integrating algorithms to guide the fuzzer tasks. The algorithms surveyed for the guidance mechanism varied in complexity of implementation and were hindered by overheads introduced to the fuzzer and challenges dealing with nondeterministic outcome. The algorithm (multi-armed bandit algorithm) that was ultimately chosen accounted for non-determinism and provided a bound on the expected error from the actual recorded outcome. The algorithm also integrated with minimal overhead in performance during fuzzing. The next two months was spent integrating the algorithm to guide the mutation, code generation and seed selection components. Each test for the implementation of the guiding mechanism in the fuzzer was evaluated over a period of 24-hours, to capture enough coverage data for performance measurements. In addition to proper integration with the fuzzing engine the algorithm needed to be numerically stable for long periods of time. This would result in many tests run for 24-hours before the implementation was viable for long term fuzzing. The implementation process started with work on the mutation and code generation engine before moving on to providing intelligent guidance from the programs chosen from the seed corpus. The final week was allotted to report writing and evaluating the statistics gathered from the fuzzer

2.Literature Review

This section addresses the nuances of JavaScript Language and its execution via JavaScript Engines. A brief survey of the components of existing fuzzers that perform JavaScript fuzzing and their characteristics. Subsequently, it delves into addressing the limitations of Fuzzilli and potential approaches to tackle these limitations and feasibility for optimization. Finally, the applicability of the role of Multi-Armed Bandit (MAB) algorithms in Fuzzilli's inherent weightage mechanisms is discussed.

2.1. The Java Script Language and Engine

JavaScript comprises about 97.9% of the used programming language for client-side implementation of web pages in the world [7]. The language conforms to the ECMA standards and therefore ensures interoperability between browsers despite variant JavaScript Engines to interpret the language. All variants of JavaScript engines have the common characteristics of providing a runtime environment for the language and providing optimized performance via Just-in-Time (JIT) compilation.

To execute JavaScript languages, the engine parses the language and creates an abstract syntax tree (AST). The interpreter converts the AST of JavaScript into bytecode for execution directly with the runtime environment. When similar bytecode is repetitively executed, it lowers it with the JIT Compiler into one or more intermediate representations (IR). This IR is then optimized and compiled to machine code for execution. Due to the trivial nature of logic in parsing and interpretation and conformance to the ECMA standards, the frequency of reported issues in the parsing and interpretation are few and far between. [8]

A more prevalent target for bug discovery in a JavaScript engine is in the JIT compiler. During runtime, the engine deems repetitive occurrences of similar control flow and data structure code suitable for the potential of optimization often referred as hot code. To perform optimisation with the code the engine lowers the language level of the interpreted bytecode through one or more lower-level IRs. The JIT compiler then parses the IR and modifies it for speed and storage optimality, ultimately the optimised IR is lowered to native machine instructions for best performance during execution. This is the role of the JIT Compiler, and it is known to have the most variance between JavaScript engines. The differences in JIT compilation are noticeable, especially in terms of IR lowering capabilities and utility of various optimization techniques such as function inlining and redundancy elimination during execution. [9, 10] The significant variance in JIT compilers is attributed to the various implementations of JavaScript engines with the desire of competitively outperforming each other. This dramatically increases the complexity of implementation and as a result the frequency of bugs in this component.
JIT compilers will often add checks, and remove or merge redundant checks to validate assumptions to optimize the IR. With the modified checks, if validation constraints are not satisfied the code falls back to an unoptimized state referred to as bailing out/deoptimization. If the IR is not validated correctly during optimisation, it often leads to discovery of exploitable conditions.
JIT compilers are primarily vulnerable with three kinds of bug classes triggered via failures during JIT optimisation, these are:
Type Safety – this class of bug occurs when a validation for a type is removed during JIT optimization on analysis of the code but then an access is made to the type whose check was dropped. This leads to type-confusion bugs.
Spatial Memory safety – this class of bug is caused when validations are removed for range checking in buffers resulting in illegal memory accesses. This leads to out-of-bounds (OOB) memory accesses bugs.
Temporal Memory safety– this class of bug occurs with dynamic memory allocation when memory is freed for an object that no longer exists and an illegal memory access is made to that freed object. These

bugs are referred to as use-after-free (UAF) bugs and they tend to occur during improper garbage collection.
In addition to implementation strategies along with the complexity and diversity of maintained JavaScript engines across the board, various ideologies exist in approaches to optimisation. These variances practically guarantee introducing flawed handling of validation checks during optimizations. It often results in some class of bugs uniquely specific to engines. This requires that fuzzing JIT compilers for security auditing be adaptive to these variances. [11]

2.2. Java Script Fuzzers

There have been several fuzzers to perform JavaScript fuzzing over the years. These fuzzers are equipped with capabilities to generate and mutate programs via code generators and mutators. They can be described as generative i.e., having the ability to build new programs from the ground up with pre-defined grammar rules or via snippets of code built from a large corpus of deconstructed programs. Prominent generative fuzzers for JavaScript fuzzing are jsfunfuzz [12], Code Alchemist [13], Fuzzilli [14] and DIE [15]. Alternatively, the fuzzer may be mutational i.e., having the ability to recombine various code blocks from various programs with a seed program forming new programs. Prominent mutational fuzzers for JavaScript are Superion [16], LangFuzz [17], Nautilus [18], Fuzzilli and DIE. A good majority of these fuzzers perform grey box fuzzing using coverage as guidance for exploration in some capacity.

Generative fuzzers are usually in complete control of program generation, with the advantage of making every statement in a program generated syntactically and semantically valid. Due to its nature of pre-emptively avoiding syntactical and semantic errors, a considerable amount of these fuzzers manipulate context-free grammars (CFGs) that aid in the construction of semantically correct programs. However, there are limitations to generative fuzzers, primarily while targeting the JIT compiler. For JIT bugs specific conditions are required to be met to trigger the generation of hot code. It is often difficult to produce them in a timely manner due to a large corpus search space for generation.

Aspect. - In this paper describes a feature in a program such as proof of concepts (PoCs) of existing bugs, which is known to generate bugs due to its unique structure and type.

Mutational fuzzers rely heavily on unit test suites and proof of concepts (PoCs) of known bugs as seed inputs. The JavaScript code in this regard is often meticulously constructed towards stressing the JavaScript engine in one or more areas. The key aspects that require preservation for that effect, lie in the unique control flow execution and data dependencies of variables that explore specific paths. A mutational fuzzer, faces challenges in preserving these characteristic aspects after mutations, in addition to producing semantically valid program code.

Generative fuzzers may also benefit from utilisation of imported unit test suites and PoCs but are also at a disadvantage in being unable to construct and trigger the imported data's aspects in reasonable time.

When it comes down to performing mutational and generative fuzzing operations the pool narrows down to two fuzzers - DIE and Fuzzilli. Each of these fuzzers provide features of both generative and mutational capacity.

Fuzzilli uses a unique approach by having the seed programs in an intuitive intermediate language (IL) called "FuzzIL". The IL represents program code at the bytecode level which is closer to a JavaScript engines internal representation. This awards Fuzzilli an advantage in terms of ease of mutations with a

guarantee of syntactic and semantic validity, as the IL assuredly maps to bytecode control flow and data structures with a high rate of correctness.

DIE is a potential fuzzer for integration. However, due to the lack of detailed documentation to aid integration, along with the lack of active development and maintenance. The complexity involved in this fuzzer would require considerably larger effort to overcome than the permissible time for this project.

In this paper Fuzzilli is chosen as the JavaScript Fuzzer for ease of integration of proposed modifications and intuitive control over mutation/generation of JavaScript programs. Additionally, Fuzzilli is open-source and actively in development maintained by browser bug hunting experts enabling adequate peer reviewing for enhancements and improvements.

2.3. Limitations of Fuzzilli

Fuzzilli faces two major limitations in performing fuzzing. 1) Lack of aspect preservation to trigger specific bugs with specific preconditions. 2) A lack of intelligent guided fuzzer components through task selection and seed selection, to improve fuzzer performance and have faster and potentially higher coverage discovery.

The lack of aspect preservation is in respect to preserving context of the unique control flow path and dependency between variables to triggering specific behaviour during fuzzing. Fuzzilli historically works only with programs constructed in FuzzIL, however in recent versions it can perform fuzzing with a corpus of unit test suites and proof of concepts (PoCs). This is possible through Fuzzilli's FuzzIL compiler to convert external JavaScript program corpora to FuzzIL program corpora. With the ability to import programs into the corpus, either generative or mutational fuzzer actions can be performed on a rich corpus of target specific areas. However, as noted about JavaScript fuzzers in the previous section (section 2.2), this partially addresses aspect preservation and results are not always performed in meaningful time. This can be attributed partly due to disassembly of the original program's aspects, wherein mutations may alter the required conditions for triggering the specific outcome. This introduces greater difficulty in reproducing variants of these outcomes such as JIT compilation which requires specific conditions to trigger code for optimisation in the engine.

Therefore, a mechanism is needed to conserve aspects of the program that setup certain conditions for fuzzing.

Fuzzilli has made efforts toward aspect preservation via its hybrid engine. However, this fuzzing engine is still in an experimental stage. The hybrid engine relies on manually configured predefined program templates for use in generating code otherwise needed to setup an environment for specific conditions.

DIE is currently the only known JavaScript fuzzer that performs aspect preservation in an automated manner by considering the structure and type information of a test case from its aspects. Efforts to incorporate similar automated strategy in Fuzzilli is a complex endeavour and beyond the mandate of the paper. However, it remains a potential topic for future research.

The second limitation arises from the large amount of effort lost in fuzzing arbitrary tasks that are unlikely to generate new coverage. This is due to the absence of feedback to prioritize specific code generation tasks, mutation tasks and specific program seed inputs. While in essence the fuzzer is coverage guided as it only preserves programs to the corpus that discover new coverage, it does not maintain the context of a program's viability to discover coverage over time when fuzzing. For e.g., program seed that provided coverage at initial stages of fuzzing are not necessarily productive at late-stage fuzzing. However, due to the arbitrary selection, the last meaningful context of a program having viability for new coverage discovery is not acknowledged. Therefore, programs with obsolete context are reused many times and this slows down the fuzzing performance.

Coverage as a reward metric helps track optimal programs through the context of immediate viability for coverage discovery. Fuzzilli attempts to limit the choice of suboptimal coverage discovery programs through the implementation of its Markov Corpus (further discussed in the Chapter 3). This corpus limits the search space by reducing the size of the corpus with programs performing more exploration through a series of connected edges rather than arbitrarily discovered new edge coverage.

The primary implementations of coverage guidance in JavaScript fuzzing so far, have primarily focused on generating a corpus for higher coverage discovery. The intermediary tasks of mutations and code

generation that evolve the programs in the corpus remain largely unguided. This incurs loss in efficiency by generating suboptimal programs that are eventually discarded due to their inability to find new coverage.

Providing coverage-based guidance to guide the selection of applicable code generators and mutations would intuitively improve the rate of coverage growth. It would minimize suboptimal evolutions to the programs generating a richer corpus.

In the next section algorithms for coverage guided selection of fuzzing components like code generators and mutators along with program selection in corpus management schedulers is addressed.

2.4. Approaches to address limitations

Addressing the limitations in the previous section, the matter of aspect preservation has a lot of emphasis on preservation of the core programs. To create the same conditions that lead to specific outcomes, the theoretical hybrid engine of Fuzzilli can help with preserving aspects throughout fuzzing. However, this requires manual inputs which needs additional extraneous effort. In the autonomous fuzzing process, programs of the initial PoCs and test cases with specific outcomes could potentially undergo mutations to generate variants. However, the essence of the program that generates conditions to achieve those results are not guaranteed to be preserved in the fuzzing process. Therefore, the issue of aspect preservation is only partially resolvable.

The second limitation of the fuzzer with regards to guidance in fuzzer components through task selection and seed selection, can be addressed in an autonomous fuzzing environment through a variety of decision-making algorithms. A common trend for these guidance related algorithms, is a reasonable trade-off between exploring the sample space of outcomes and exploiting the path to the best possible outcome. Thereby, it optimally guides the execution of components in response to a favourable response. This response can be attained, from a performance metric like coverage.

The first approach to guided task selection is through Markov decision processes such as Markov chains [19]. The application of Markov chains relates a task that generates a reward (discovery of coverage) with an outcome (edges discovered), this in turn leads to new outcomes (edges discovering new edges). By modelling the chain of events, and assured reception of rewards it is possible to optimize results through a sequence of high reward yielding tasks. However, this presents a problem as Markov chains relies on deterministic outcomes. Due to the grey box approach of fuzzing, the probability of coverage discovery is nondeterministic in nature. The context for new coverage received, is subject to change with large position shifts between random control flow paths in the fuzzing target through randomly performed mutations. Fuzzilli has a partial application of Markov chains in place, in the form of the Markov Corpus (further discussed in Chapter 3). The corpus creates an association of edges discovered with programs from their execution (in its exploration phase) by only adopting the programs that generate a consistent coverage discovery. Convergence is attained through consistent edge discovery (the exploitation phase) guaranteeing some probability in new edge discovery. Thus, a form of determinism is achieved, and a probability of generating subsequent new connected edges from newfound edges in a chained form is possible with reasonable certainty. Extending the Markov chain approach for fuzzer task selection, like code generation or mutations is impractical. This is considering the large overhead in maintaining states of events for every code generation or mutation task that leads to coverage discovery. Also, no consensus

in convergence can be achieved on the broad influence of mutation and code generation tasks to create chains of coverage discovery.

The second form of guidance that can be given to the fuzzer is through particle swarm optimisation [20]. In this form of optimisation, solutions are mapped through dispersal of tracking objects (particles) in sufficiently large quantities (a swarm) through a search space. The objects, strive to find the most optimal path towards a desired solution (the exploration phase). As each particle improves its trajectory to the desired solution, it informs other particles in the swarm to guide it towards the desired solution, in this instance new edge discovery. When all particles are headed towards a particular solution with a determinable path, it attains convergence. All particles begin "swarming" toward the target at this stage (the exploitation phase). The process is akin to path finding algorithms with heuristics eliminating suboptimal paths. Through inter-communication amongst the multiple agents trying to converge at a destination, there is fast mapping of the search space. During this process, it discovers several unique paths to find the new edges. Although efficient in achieving faster convergence toward coverage discovery, once again it presents complexity in storing the coverage paths of multiple particles for each mutation and code generation task of the fuzzer. Though not particularly useful in guidance for tasks of a fuzzer, it has potential applications in the distributed setup of Fuzzilli where each fuzzer instance in the hierarchical network would be a particle in a swarm communicating the discovery of paths toward new edges. A potential subject for future research and improvement.

This finally brings us to application of sampling algorithms. These algorithms, such as Monte Carlo Sampling [21] and Thompson Sampling [22] , have a heuristic guidance mechanism with lesser complexity. The premise of the sampling algorithms is to randomly attempt a task (the exploration phase) and determine the reward satisfaction achieved from the sampled tasks. This is possible after accurately determining a consistency of the satisfaction on some convergence. When achieved, it is possible to choose the next task for the next best reward (the exploitation phase). Amongst the different kinds of guidance algorithms mentioned, the sampling algorithms requires the least amount of state management to implement. However once again like Markov decision processes a determinism must be achieved on expected outcome. This is apparent in the reward distribution established while exploring tasks for these sampling algorithms; To perform efficient exploitation the mean or variance if known with certainty greatly simplifies the problem and would ideally be the case if the outcomes were deterministic. In Thompson sampling an estimated outcome distribution is created and will constantly be updated with the discovered outcome distribution. As such, it alters the mean and variance of the distribution the more exploration is performed. If the outcomes were deterministic, Monte Carlo and Thompson sampling would home in on the most accurate distribution of outcome and converge on a result quickly.

Unfortunately, the mean and variance are unknowable for fuzzing due to nondeterminism of outcome which would make Thompson Sampling and Monte Carlo operate on ever changing distributions.

The non-determinism however can be overcome, through estimations of an outcome distribution and minimising the difference between an expected outcome distribution and actual outcome distribution. The difference measured is a term known as regret. If the regret is bound to an acceptable manageable range over time trying to bring the value closer to zero, it is possible to determine a near deterministic outcome. This can be achieved through bandit algorithms that build upon the exploration and exploitation foundations of Thompson sampling. The next section discusses the subset of bandit algorithms is a good fit for guiding Fuzzilli's fuzzing task selection.

2.5. Role of Multi-Armed Bandit in Fuzzing

The chosen model for a dynamic weighting system to guide Fuzzilli's decision making is based on the Multi-Armed Bandit (MAB) reinforcement-learning problem.

The MAB problem yet again, presents the classic example of a trade-off between exploration and exploitation. The problem describes a gambler playing several competing slot machine arms (choices) to maximize the expected reward. In the beginning, information on each arm's ability to produce a favourable outcome for a desired reward is partially known and may improve as the arm is played more to acquire more data. The gambler must thus maximize the outcome with information deduced from each arm pulled (the exploration phase), to help make a more affirmative decision on the choice of the next pull (the exploitation phase).

In the analogy used to describe the rewards (gains) from the slot machine, the reward dissemination mechanism may provide a reward which is predicted (expected outcome). The difference between the acquired (actual outcome) and predicted reward is measured as the regret. An accurately predicted outcome would entail zero regret. However, this is not always the case in modelling real world systems. The regret for each reward acquired is unpredictable. Thus, the algorithm tries to bound this regret to an acceptable level that would still generate favourable results and over time try to bring it closer to zero.

Adversarial MAB, introduced in 1995 is a strong generalization for this exact MAB problem. In this model an adversary incurs losses on the gambler by trying to impede its goal of maximizing reward. Thus, rewards for each arm are altered during each play. The gambler in return, adapts a solution that reacts quickly to the changing rewards of each arm.

Fuzzing is arguably like the Adversarial MAB problem. The choices are the different paths in the control flow graph of execution. The fuzzer explores each path and affirms a guidance decision that puts the fuzzer onto various discovered paths. The path chosen overtime is the one which may have the best outcome i.e., acquires new coverage. The gain in this context is measured as the paths discovered in the graph i.e., new coverage discovered. The expected outcome is an estimate of the discovery of new coverage compared to the actual discovery of new coverage. The results of coverage discovered are unpredictable, as the fuzzing target provides impedance to the reach of the fuzzer within the fuzz target. With the target being mostly treated as uncharted domain, the expectance of some regret is quite significant for discovery of coverage. Limiting the bounds of this regret, allows the benefits of a partial guarantee in expected reward and better modelling towards higher gains in fuzzing.

The focus of Fuzzilli's adversarial bandit problem is addressed with the Exponential-weight algorithm for Exploration and Exploitation (Exp3) proposed by Auer et.al. [23] The algorithm exponentially increases weights of well performing arms thereby increasing the probability of playing the arm with the best outcome**.** For the project implementation use case, the variant of the Exp3 algorithm (Exp3.1) [23] is implemented, as it provides a longer time horizon by adjusting upon the changing information from the fuzzer and the decision parameters of the Exp3 algorithm over many epochs. This is desirable as fuzzing evolves over the course of time with initial seed programs eventually becoming inefficient at generating new coverage.

Finally, there is also good evidence of adequate performance in kernel fuzzing, another fuzzing domain with similar task selection optimisation. This can be seen in the kernel fuzzer SyzVegas [24]

3.Background

This section provides context to the overall functionality of Fuzzilli and the key components of the fuzzer that aid in fuzzing. It provides necessary context for the proposed modifications to the fuzzer to have better coverage discovery.

3.1. Fuzzilli Overview

Fuzzilli is a fuzzer, which generates inputs for fuzzing as programs in an intermediate language (IL) called FuzzIL.

After a series of code generation/ mutation tasks performed on an IL program, it can be translated to JavaScript code for execution. This is done through a process called lifting. Lifting is the opposite of lowering which is the process of converting a language from a higher level to a lower-level representation.

The IL is designed with four central tenets:

Ease of Mutations - FuzzIL represents JavaScript code at the bytecode level which is a more truthful representation to the JavaScript engine's internal representation of code. This facilitates ease of mutations, as at the bytecode level it closely resembles control and data flow graph structures. It also generates semantically valid code with high probability. This is because all mutations must conform to a set of basic semantic correctness rules of the IL. These include enforced rules such as defining variables before their usage. The rules and the fact that semantically invalid programs are excluded from the seed corpus, maintains an overall percentage of a low and acceptable level of emitted invalid programs.

Statically Determinable Resolution – This means that FuzzIL instructions are partially immutable in some code generation mutation tasks to avoid problems with variable type resolution. This is because of the nature of the JavaScript language when defining variable types. Ambiguity in type resolution, guarantees the generation of JavaScript code that will have runtime exceptions pre-empting program execution. To overcome this, Fuzzilli employs an abstract interpreter to determine consistently the types of variables against a static model of the JavaScript runtime environment. Reasoning types is not easy if it is not immutable during a generation task. However, as there are code generation independent tasks amongst mutation tasks, it allows the mutations to mutate the variable types.

Ease of lifting to JavaScript - As FuzzIL is a representation of bytecode level code, the algorithms utilized in lifting produce valid JavaScript as all IL instructions have a one-to-one mapping with a JavaScript code.

Correctness Standard - Rules are set in place for generation of the IL. It establishes properties that uphold semantic and syntactic correctness and determinable variable definitions before use.

Fuzzilli like most generative and mutational fuzzers has features for code generation and mutations via its code generation and mutation engines. Fuzzilli also provides corpus management scheduling for importing corpora of test suites and PoCs that can be useful in providing targeted fuzzing. Additionally, Fuzzilli provides means to setup for execution in a distributed environment. This allows for a high level of scalability in fuzzing over multiple instances of the fuzzer.

3.2. Code Generation in Fuzzilli

In Fuzzilli the process of creating input FuzzIL programs from scratch is via Code Generators.

When starting without a corpus, the Code Generators provide Fuzzilli a means to generate a single, arbitrarily chosen initial sample.

The code generators are responsible for generating relevant language features (e.g., object operations, unary and binary operations, etc.) as contents of a program. Apart from being invoked to generate a corpus from scratch, these are invoked whenever mutators that involve code generation are called to mutate programs. The resultant generated programs are lifted to JavaScript executed and are potentially capable of discovering new coverage. Upon finding new coverage the IL program is added to the corpus as new programs that can be retrieved from the corpus to be mutated on again.

Code generators are stored in a weighted list and are thus selected with a weight-based probability distribution. This allows for randomisation in the generated code, and controls how often arithmetic operations or method calls are performed, or how much control flow (if-else, loops, ...) is generated relative to data flow. Adding code generators that generate code fragments that have historically resulted in bugs in the past, such as prototype changes, custom type conversion call-backs (e.g., valueOf), or indexed accessors guides the fuzzer toward certain bug types.

However, Fuzzilli code generators are assigned static weights that prioritize code generators most likely to succeed in creating new coverage growth generating programs in the early stages of fuzzing but produce suboptimal new coverage growth at later stages. This results in longer wait times between programs for notable growth in new coverage generation.

The Code Generation algorithm for Fuzzilli has the following workflow:

The mutation engine is tasked with either a splicing task or a code generation task both initiated through the code generation mutator.

If the task is splicing, a random program is selected from the corpus and instructions from this program are picked for random introduction into any part of the program undergoing the mutation refactoring with appropriate variables and control flow elements.

If the task is code generation, a predefined template for a code block characteristic of the selected generator is selected the scope context is analysed and the available input variables are retrieved.

The generator is then invoked, and the code block is implemented.

The generation process is then performed until a fixed number of instructions are achieved before completing the task of code generation.

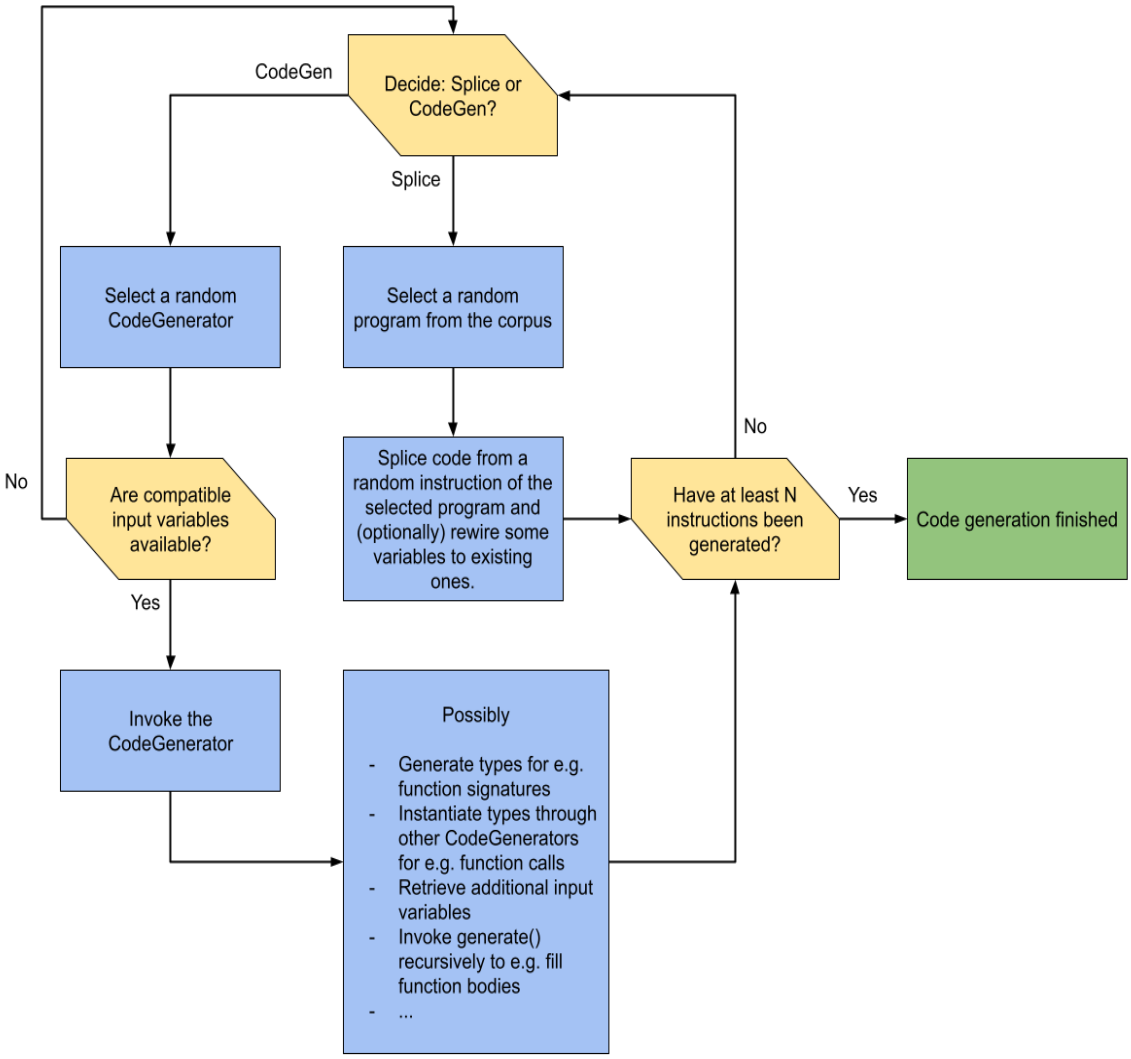


Figure 2:Fuzzilli Code Generation Engine

Source: https://github.com/googleprojectzero/fuzzilli/blob/main/Docs/HowFuzzilliWorks.md#the-generative-engine

3.3. Mutators in Fuzzilli

FuzzIL is designed to facilitate various code mutations. When performing mutator manipulations on the IL programs it is done so on a copy of the original. Thereby maintaining the originals of every program that ends up being mutated.

There are a few core mutator types:

Input Mutator – As the name suggest, it is a central data flow mutation that replaces an input to an instruction with another, randomly chosen one.

Operation Mutator – Again as the name suggests, this fundamental mutation mutates the parameters of an operation e.g., binary operations like addition and subtraction.

Splice Mutators – This mutator copies self-contained parts of one program into the program that is currently undergoing mutation. A simplified variation of a splice mutator is the Combine mutator that inserts a program in entirety.

Code Generation Mutators – These mutators generate new and random code at one or multiple random positions in the mutated program. As explained, these mutators rely on the code generators for the random code generation.

In addition to the core mutators there are other mutators that can be used to perform tasks from duplicating code to refactoring it.

When a Mutation is performed on an IL, its lifted JavaScript program is executed, and the IL program is added to the corpus if its execution results in discovery of new coverage.

Like code generators each mutator is stored in a weighted list and selected with a weight-based probability distribution. Thus, allowing for random mutation changes on programs to generate new ones.

Again, just as code generators weights these mutators are assigned static weights that prioritize mutators most likely to succeed in creating new coverage growth generating programs in the early stages of fuzzing but produce suboptimal new coverage growth at later stages. This further influences the resultant turnover of programs for notable growth in new coverage generation.

The mutator workflow is as follows:

A random IL program sample is selected from the corpus and undergoes a series of mutations to evolve a new program.

This program is lifted to JavaScript to be used for execution in JavaScript Engine.

If the execution is unsuccessful the program mutations are reverted, and the original program is subjected to a new round of mutations.

Upon successful execution the program is evaluated for having discovered new coverage. If this is the case, then the program is prepared for minimisation. Minimisation is an inherent feature of all known JavaScript fuzzers. It is an optimisation step that reduces the number of instructions in a program to one that is capable of still triggering the same new coverage growth as the original. It is primarily achieved through inlining of instructions i.e., reducing unwanted instructions to fewer instructions. Minimisation is

often applied with discretion as it requires multiple executions and may introduce longer wait times between coverage discovery and the next step of adding the program to the corpus.

Either way if the program generates new coverage it is added to the corpus and next round of mutation continues.

Alternatively, if no new coverage is discovered the programs are discarded and next round of mutation continues.

After a fixed number of mutation rounds, the selected program completes one round of fuzzing, and the next fuzzing round begins with a new seed program from the corpus.

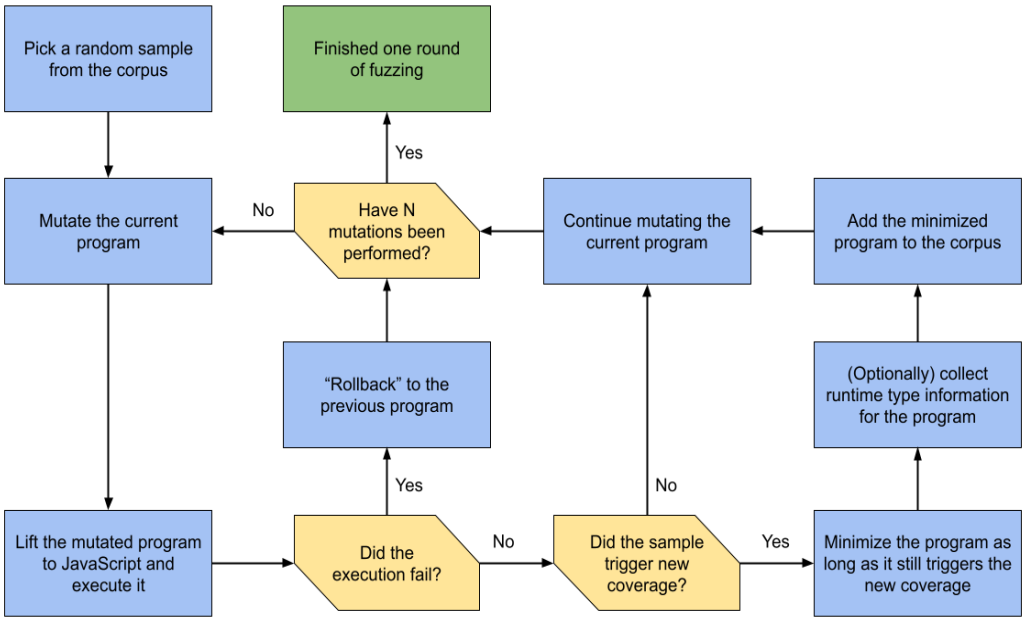


Figure 3:Fuzzilli Mutation Engine

Source: https://github.com/googleprojectzero/fuzzilli/blob/main/Docs/HowFuzzilliWorks.md#the-mutation-algorithm

3.4. Corpora in Fuzzilli

Fuzzilli maintains different kinds of program corpora for future mutations. The central tenet of a program to exist in a corpus is that it is deemed "interesting" i.e., it has discovered new coverage by discovering new edges in the control flow graph of a fuzzing target.

The two key corpora are the Basic corpus and the Markov Corpus

In a basic corpus implementation, samples are mutated a certain number of times before it is executed to assess discovery of new coverage. These programs are assigned an age and eventually "retired" after they have been mutated a pre-determined number of times.

The Markov corpus as the name suggests maintains a corpus that utilises a corpus management algorithm based on Markov Chains. [25, 19]. A key characteristic of the corpus is determinism. The programs in the corpus must converge on the same path of generated coverage every time the program is executed. This characteristic is also central to the corpus management algorithm that utilizes Markov chains. One of the aims of the Markov corpus is to identify the frequency of paths exercised during fuzzing and tune the corpus to utilise programs that exercise lower frequency paths, capable of leading to potentially interesting outcomes. Any new coverage discovering program exercises a path, that unlocks potentially new paths to explore. A mapping of a path i exercised by a program that in turn discovers a program leading to path j with a given probability P_{ij}^{-1} creates a chain of stochastic events that can be modelled as a Markov chain. Utilising an energy parameter to gauge the power schedule of the fuzzer, the basic corpus expends more energy in higher frequency paths due to a bias indicated by its historic discovery rate. This results in loss of potential paths for interesting coverage discovery from the newly discovered paths that are executed less frequently. [19]

3.5. Distributed Setup of Fuzzilli

Fuzzilli implements a simple TCP-based protocol to exchange programs and statistics between fuzzer instances. The protocol consists of messages being sent between instances in both directions to facilitate synchronisation of the corpus between instances. Instances of Fuzilli can either be setup locally in a threaded environment or over a network. Instances adopt a master-worker hierarchy. The master and worker instances perform independently of each other except for having the latest corpus and the overall discovered coverage shared between instances when new coverage is discovered between them. The distributed environment helps scale the rate of coverage discovery.

4.Proposed Modifications

This section addresses modifications to the fuzzing engine of Fuzzilli. A high-level overview description of design modifications adjusting the decision making of the fuzzer towards producing maximum new coverage. More specifically, these modifications are aimed at seeking the best new coverage discovery rate. The details of the coverage discovery rate will be discussed in the implementation section.

4.1. Mutators MAB

To facilitate weight changes for mutator selection a few proposed modifications are introduced in the mutation engine of Fuzzilli. In the original Fuzzilli implementation each mutator is assigned fixed arbitrary weights in a list and prioritizes mutations on a program from the weightage in this list. The new modifications provide additional properties to record the coverage growth in the weighted list and to aid the MAB algorithm to update the weights.

A couple of new mutators are created to isolate specific tasks in existing mutators. The isolated tasks form mutators in accordance with the core mutator types. A very important example of this is the modification to the original code generation mutator. As seen in the code generation engine flow diagram above, this mutator would perform splicing (an infusion of instructions from existing programs into the target program for mutation) and basic code generation (building instructions from scratch) as one mutation task. To have tighter control on the decision of performing splicing or code generation tasks, each task is split into separate mutators with their own assigned weights. This provides the ability of evaluating each tasks individual contribution to coverage growth. Separation of code generation tasks, also allows tighter control over selection of code generators as can be seen later in the proposed code generation modifications.

The coverage information to evaluate weights for each mutator are accumulated over many program mutations. In one fuzzing round (or fuzzer iteration) of Fuzzilli a program undergoes several mutations, after each mutation of the program it is executed and recorded for discovery of new coverage found if any. The coverage for each mutator is accumulated and recorded over several programs spanning many fuzzer iterations. A critical mass threshold of iterations is defined for mutator evaluation. When this threshold is reached, the accumulated coverage data is used to evaluate the updated weights of all participant mutators over the iterations.

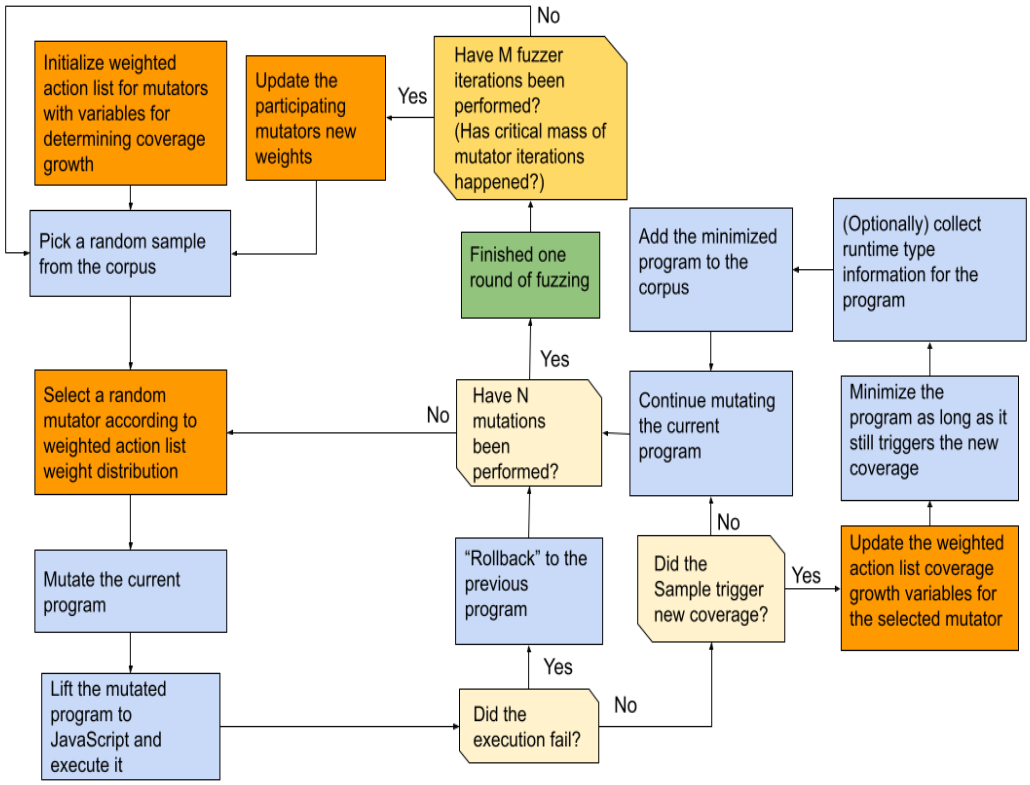


Figure 4:Mutation MAB high-level overview

Original source: https://github.com/googleprojectzero/fuzzilli/blob/main/Docs/HowFuzzilliWorks.md#the-mutation-algorithm

4.2. Code Generators MAB

To facilitate weight changes for code generator selection a few proposed modifications are introduced in the mutation engine of Fuzzilli. Like mutators, in the original Fuzzilli implementation each code generator is assigned fixed arbitrary weights in a list and prioritizes invocation for use in a program from the weightage in this list. The new modifications provide additional properties to record the coverage growth in the weighted list and to aid the MAB algorithm to update the weights of the generators.

The code generation engine is invoked while performing mutations that have code generation tasks to be performed on a program. As explained in the proposed modifications of the mutators, the code generation mutator specifically is separated from its intertwining with a splicing feature. This is done to have better ability in understanding the contribution of a code generator task to the coverage growth of a program.

Like mutations the coverage data is captured for a program over multiple mutations with several participant code generators accumulated over many fuzzer iterations. Like mutators a critical mass threshold of iterations is also defined for code generator evaluation. When the critical mass of iterations is reached, the accumulated coverage data is used to evaluate the updated weights of all participant code generators.

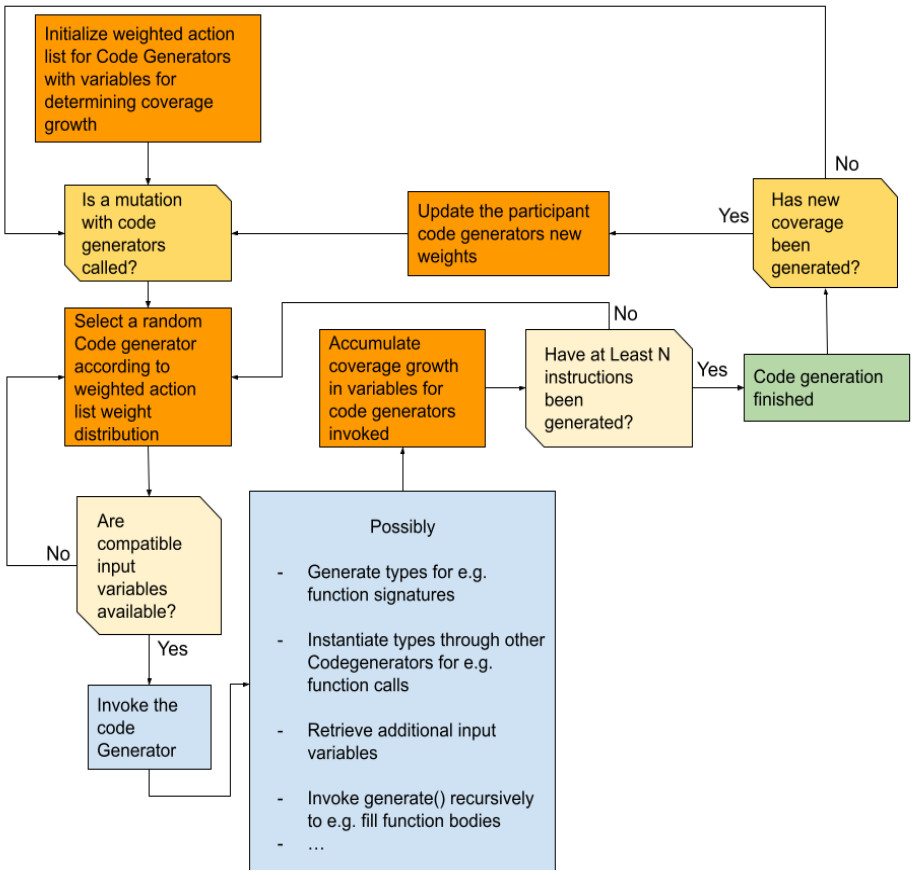


Figure 5:Code Generation MAB high-level overview

Original source: https://github.com/googleprojectzero/fuzzilli/blob/main/Docs/HowFuzzilliWorks.md#the-generative-engine

4.3. MAB Corpus

Fuzzilli works with various corpus management schedulers. A new corpus is proposed that establishes a weighted action list for each of the programs introduced in the corpus. Unlike the mutator and code generator weighted action list, the program weighted action list is grown over time and is not a finite list of actions. To use MAB more efficiently, the corpus employs a caching mechanism creating a finite pool for selection as new programs continue to be added to the corpus. The weights guide the selection of programs on the metrics of discovery of new programs for coverage growth rather than just coverage alone.

The corpus is initialized at the beginning with an empty list it then proceeds to retrieve programs either via an import or via code generation. If the programs are imported, they are executed individually and added to the corpus. To have best fuzzing performance from programs, various pools are available for the corpus. These pools, aid in maintaining a high-performance program tracking and caching mechanism.

The main pools track all the programs in the corpus that have found new coverage during the splicing mutation or other kinds of mutations. The main seed pool and main program pool, separately track programs for the splicing mutation and other mutations respectively.

The availability pools track programs with the highest program discovery to invocation ratio during splicing mutations and other kinds of mutations. The seed availability pool and the program availability pool, separately track programs for the splicing mutation and other mutations respectively.

The main program pool and program cache track programs together that aid in the performance of mutations that do not require preservation of program form. On the other hand, the main seed pool and seed cache together track programs facilitate splicing mutations by preserving a source of the original form of programs. This provides some degree of aspect preservation.

Each program selected from a cache is gradually aged over time. The age of a program is used to retire programs from the availability pool and provide newer programs a chance in the next regeneration of the cache. The regeneration of a cache is triggered when the program selection fails to perform new program discovery within a reasonable number of epochs.

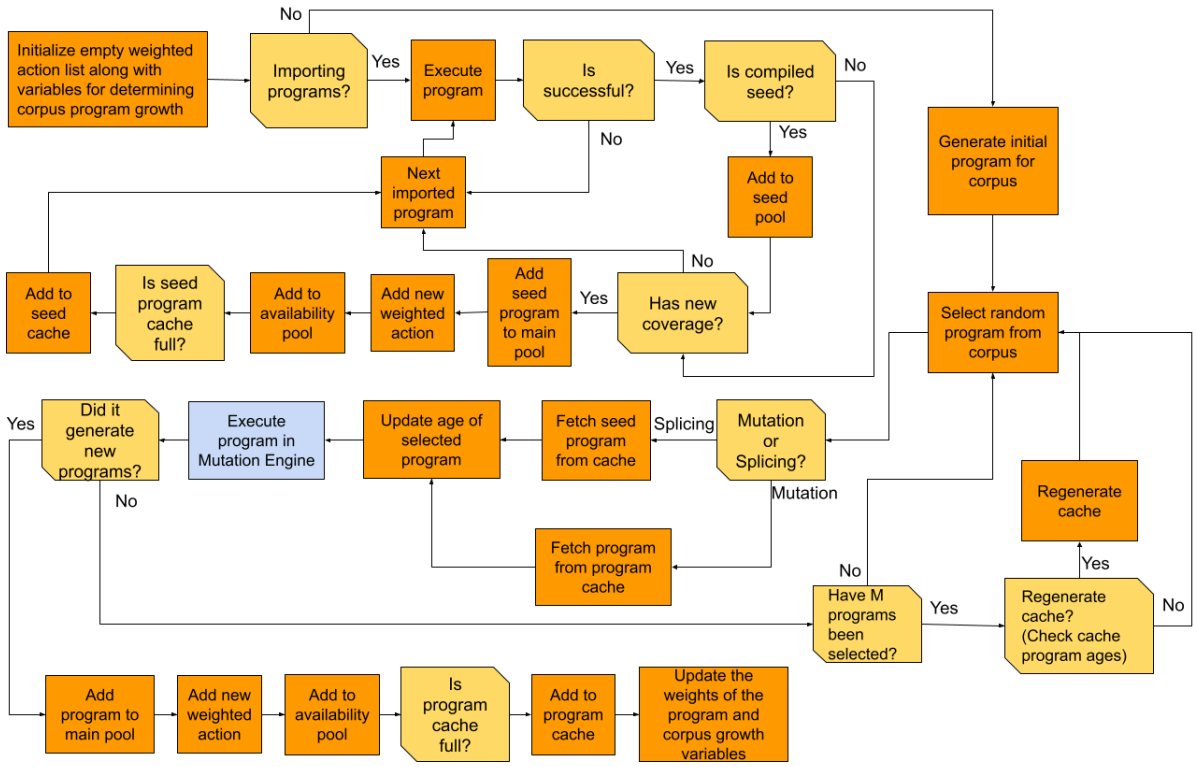


Figure 6: MAB Corpus high-level overview

5.Implementation

This section elaborates on the reward assessment model for coverage evaluation for dynamic rebalancing of the weights. In addition, it discusses the algorithms that update the weights and drive the selection of components to code generators and mutators. Finally, the programs generated from fuzzing are the recipients of changes induced by code generation and mutations. Therefore, it is provided its own algorithm for program selection via the implemented MAB corpus.

5.1. Reward Assessment Model

The following equations are used for evaluating rewards for mutators and code generators are as follows:

The key guiding element in Coverage based grey-box fuzzing is to maximise coverage (i.e., number of edges covered).

The reward model for mutations is modelled as such,

Let t_{mut} be the number of iterations elapsed up to the most recent invocation.

Let c_{mut} be the number of edges found by a single invocation of a mutator

and inv_{mut} the number of invocations of the mutator in a single iteration of fuzzing.

Likewise let C_{mut} be the total number of edges found by the mutator over many invocations since the first iteration of fuzzing and Inv_{mut} be the total invocations of the mutator since the first iteration of fuzzing.

Compute the new average coverage ($c_{mut\ \text{avg}}$) for the mutator within a single iteration

$$c_{mut\ \text{avg}} = \frac{c_{mut}}{inv_{mut}}$$

Compute the global average coverage ($C_{mut\ \text{avg}}$) for the mutator over many iterations

$$C_{mut\ \text{avg}} = \frac{C_{mut}}{Inv_{mut}}$$

The reward for a mutator is thus calculated as the coverage growth rate (g_{m_t})

$$g_{m_t} = g_m(t) = \frac{c_{mut\ avg}}{C_{mut\ avg}} \times iterations$$

The same model is used to evaluate code generators coverage growth rate (g_{cg_t}) calculated with $c_{mut\ cg}$ the total edges in a single invocation, $C_{mut\ cg}$ the total edges discovered for a code generation mutator task over many iterations, inv_{cg} the number of invocations of a code generator on code generation mutation invocations in a single iteration and Inv_{cg} be the total invocations of the code generator since the first iteration of fuzzing.

$$c_{cg\ \text{avg}} = \frac{c_{mut\ cg}}{inv_{cg}}$$

$$C_{cg\,\text{avg}} = \frac{C_{mut\ cg}}{Inv_{cg}}$$

$$g_{cg_t} = g_{cg}(t) = \frac{c_{cg\ avg}}{C_{cg\ avg}} \times iterations$$

For the corpus program rewards the reward (g_{pr_t}) is based on programs discovered instead of coverage.

The reward is g_{pr_t} calculated from the program rate $P_{pr\,\text{avg}}$ with P_{pr} the total programs discovered for a program over many iterations and Inv_{pr} the number of invocations of a program over many fuzzing iterations.

$$P_{pr\,\text{avg}} = \frac{P_{pr}}{Inv_{pr}}$$

$$g_{pr_t} = g_{pr}(t) = P_{pr\,\text{avg}} \times iterations$$

5.2. Normalisation of rewards

The rewards g_{cg_t}, g_{m_t} for code generation and mutation respectively have values that range from $(-\infty, \infty)$. Multi-armed bandit problems such as Exp3 and Exp 3.1 require rewards to be normalised to 0,1. The Logistic function $1/(1+e^{-y})$ is commonly used for a normalisation from $(-\infty, \infty)$ to $(0,1)$. However, it is desired that zero reward be also included in the normalisation and thus the logistic function is shifted to accommodate it and get $(1-e^{-y})/(1+e^{-y})$. This is also desirable for no coverage generating tasks, where the zero reward will have no consequent update to weights.

This scales the range $(-\infty, \infty)$ to $(-1, 1)$ which guarantees that zero reward is always normalised to 0.

To make sure that a positive reward (g) is always normalised to a positive normalised reward a shifted version of standard Z-score ($z = (g - \bar{g})/\sigma_g$) having mean of $\bar{g} = 0$ is used. With this ($y = g/\sigma_g$) substitution in the logistic function.

The normalised reward (x) is

$$x = (1 - e^{-g/\sigma_g})/(1 + e^{-g/\sigma_g}).$$

Due to impracticality of tracking all the rewards for computing the standard deviation of the reward (σ_g) for the Z-score used in the normalisation function it is computed as follows:

$$\sigma_g = \mathrm{E}(g^2) - \mathrm{E}^2(\mathrm{g}) = g^2/\mathrm{n} - (\mathrm{g}/\mathrm{n})^2$$

where the equation relies on the sum of rewards (g), the sum of squared rewards (g^2) and the number of arm invocations (n).

5.3. Mutator Selection Algorithm

The mutator selection algorithm is based on the adversarial MAB algorithm Exp3.1. The algorithm is spread out over many fuzz loops. A fuzz loop corresponds to a single iteration. Several mutations to programs take place in an iteration (i). The data pertaining to mutations is accumulated for each mutator (m) up until a critical mass of iterations ($i_{m\ crit}$) is reached before the algorithm acts upon it.

The algorithm works for trials (t) spread over epochs (r). The rewards gm received are normalised as ($x_{\mathrm{m_t}}$) in the range $(-1, 1)$

An accumulator for total normalised rewards ($x_{m_t}^{i_{m\ crit}}$) over critical mass iterations ($i_{m\ crit}$) accumulates the normalised rewards ($x_{\mathrm{m_t}}$) for each mutator (m) invocation.

During the interim period between critical mass events, rewards ($x_{\mathrm{m_t}}$) are observed after random selection of mutator (m) with probability $p_m(t)$.

The probability is derived from the weights ($w_m(t)$) for the mutator (m) during each trial (t) of the algorithm.

When critical mass is reached, the accumulated normalised reward ($x_{m_t}^{i_{m\ crit}}$) observed for a mutator (m) is randomly selected with probability ($p_m^{i_{m\ crit}}$) influenced with algorithm's decision parameter (γ_r) to facilitate exploration and exploitation of the selected mutator. The probability derived is from the mutator weights ($w_m(t)$). The MAB algorithm's decision parameter (γ_r) changes over epochs (r) allowing the algorithm to theoretically run until maximum coverage is found.

The epochs (r) of the algorithm increase whenever the maximum estimated reward from all mutators ($max_m \hat{\mathrm{G}}_m(t)$) exceeds the difference $g_r - K_m/\gamma_r$ where K_m is the total number of mutators and g_r is a best guess for the resulting reward.

MAB is restarted with a new decision parameter (γ_r) when a new epoch (r) begins. While $max_{\mathrm{m}} \hat{\mathrm{G}}_{\mathrm{m}}(\mathrm{t})$ is less than or equal to $g_r - K_m/\gamma_r$ the estimated accumulated reward over critical mass iterations for the drawn mutator ($\hat{\mathrm{x}}_{m_t}^{i_{m\ crit}} = x_{m_t}^{i_{m\ crit}}/p_m^{i_{m\ crit}}$) is added to the total estimated reward for that mutator ($\hat{\mathrm{G}}_{\mathrm{m}}(\mathrm{t})$) The weight for the mutator ($w_m(t)$) is also exponentially updated with γ_r times the estimated accumulated reward over critical mass iterations for the drawn mutator ($\hat{\mathrm{x}}_{m_t}^{i_{m\ crit}}$). After the weight update the algorithm's trial (t) is complete.

Algorithm 1: Mutator Selection

$\hat{G}_m(0) \leftarrow 0 \quad for\ m = 0,1,2\ldots\ldots K_m$

$t \leftarrow 0$

for all $r = 0,1,2,3\ldots..$

for all $i = 1,2,3\ldots..$

$g_r \leftarrow \frac{K_m\ ln(K_m)}{(e-1)}.4r$

$\gamma_r \leftarrow min\left(1, \sqrt{\frac{K_m\ ln(K_m)}{(e-1).g(r)}}\right)$

if $i = i_{m\ crit}$

$p_m(t) \leftarrow \frac{w_m(t)}{\sum_{j=1}^{K_m} w_j(t)}$

Draw m_t *according to probability* $p_1(t)$ ... $p_{K_m}(t)$

Receive normalised reward $x_{m_t} \in (-1,1)$

$x_{m_t}^{i_{m\ crit}} \leftarrow x_{m_t}^{i_{m\ crit}} + x_{\mathrm{m_t}}$

end if

else if $i \neq i_{m\ crit}$

if $max_m \hat{G}_m(t) \leq g_r - \frac{K_m}{\gamma_r}$

$p_m^{i_{m\ crit}}(t) \leftarrow \frac{(1-\gamma_r).w_m(t)}{\sum_{j=1}^{K_m} w_j(t)} + \frac{\gamma_r}{K_m}$

Draw m_t *according to probability* $p_1^{i_{m\ crit}}(t)$ ... $p_{K_m}^{i_{m\ crit}}(t)$

Receive normalised reward $x_{m_t} \in (-1,1)$

$x_{m_t}^{i_{m\ crit}} \leftarrow x_{m_t}^{i_{m\ crit}} + x_{m_t}$

$\hat{x}_{m_t}^{i_{m\ crit}} \leftarrow \frac{\hat{x}_{m_t}^{i_{m\ crit}}}{p_m^{i_{m\ crit}}(t)}$

$\hat{G}_m(t+1) \leftarrow \hat{G}_m(t) + \hat{x}_{m_t}^{i_{m\ crit}}$

$w_m(t+1) \leftarrow w_m(t).e^{\gamma_r.\hat{x}_{m_t}^{i_{m\ crit}}}$

$t \leftarrow t+1$

end if

end else if

end for

end for

Algorithm 1:Mutor Selection Algorithm

5.4. Code Generator Selection Algorithm

The Code Generator selection algorithm relies on the rewards from mutators g_m, that perform code generation tasks. Several mutations with code generators present in programs take place in an iteration (i). The data pertaining to code generators is accumulated for each code generator (cg) in many programs executed up until a critical mass of iterations $(i_{cg\ crit})$ is reached.

The rewards g_{cg} received is the reward g_m of the mutator that invokes the code generator (cg). The reward is normalised as (x_{cg_t}) in the range $(-1,1)$

An accumulator for total normalised rewards $(x_{cg_t}^{i_{cg\ crit}})$ over critical mass iterations $(i_{cg\ crit})$ accumulates the normalised rewards (x_{cg_t}) for each code generator (cg) invocation.

During the interim period between critical mass events, rewards (x_{cg_t}) are observed after random selection of code generator (cg) with probability $p_{cg}(t)$.

The probability is derived from the weights $(w_{cg}(t))$ for the code generator (cg) during each trial (t) of the algorithm.

When critical mass is reached, the accumulated normalised reward $(x_{cg_t}^{i_{cg\ crit}})$ observed for a code generator (cg) is randomly selected with probability $(p_{cg}^{i_{cg\ crit}}(\mathrm{t}))$. It is influenced with the MAB algorithm's decision parameter (γ_r) to facilitate exploration and exploitation of the selected code generator. The probability derived is from the code generator weights $(w_{\mathrm{cg}}(\mathrm{t}))$.

The algorithm's decision parameter (γ_r) changes over epochs (r).

The epochs (r) of the algorithm increase whenever the maximum estimated reward from all code generators $(max_{\mathrm{cg}}\hat{\mathrm{G}}_{\mathrm{cg}}(\mathrm{t}))$ exceeds the difference $g_r - K_{cg}/\gamma_r$ where K_{cg} is the total number of code generators and g_r is a best guess for the resulting reward.

MAB is restarted with a new decision parameter (γ_r) each time a new epoch (r) begins.

While $max_{\mathrm{cg}}\hat{\mathrm{G}}_{\mathrm{cg}}(\mathrm{t})$ is less than or equal to $g_r - K_{cg}/\gamma_r$ the estimated accumulated reward over critical mass iterations for the drawn code generator $(\hat{x}_{cg_t}^{i_{cg\ crit}} = x_{cg_t}^{i_{cg\ crit}} / p_{cg}^{i_{cg\ crit}}(\mathrm{t}))$ is added to the total estimated reward for that code generator $(\hat{\mathrm{G}}_{cg}(t))$. The weight for the code generator $(w_{\mathrm{cg}}(\mathrm{t}))$ is also exponentially updated with γ_r times the estimated accumulated reward over critical mass iterations for the drawn code generators $(\hat{x}_{cg_t}^{i_{cg\ crit}})$. After the weight update the algorithm's trial (t) is complete.

Algorithm 2: Code Generator Selection

$\hat{G}_{cg}(0) \leftarrow 0 \quad for\ cg = 0,1,2\ldots\ldots K_{cg}$

$t \leftarrow 0$

for all $r = 0,1,2,3\ldots..$

for all $i = 1,2,3\ldots..$

$$g_r \leftarrow \frac{K_{cg}\ ln(K_{cg})}{(e-1)}.4r$$

$$\gamma_r \leftarrow min\left(1, \sqrt{\frac{K_{cg}\ ln(K_{cg})}{(e-1).g(r)}}\right)$$

if $i = i_{cg\ crit}$

$$p_{cg}(t) \leftarrow \frac{w_{cg}(t)}{\sum_{j=1}^{K_{cg}} w_j(t)}$$

Draw cg_t *according to probability* $p_1(t)$ ... $p_{K_{cg}}(t)$

Receive normalised reward $x_{cg_t} \in (-1,1)$

$$x_{cg_t}^{i_{cg\ crit}} \leftarrow x_{cg_t}^{i_{cg\ crit}} + x_{\mathrm{cg_t}}$$

end if

else if $i \neq i_{cg\ crit}$

if $max_{cg}\hat{G}_{cg}(t) \leq g_r - \frac{K_{cg}}{\gamma_r}$

$$p_{cg}^{i_{cg\ crit}}(t) \leftarrow \frac{(1-\gamma_r).w_{cg}(t)}{\sum_{j=1}^{K_{cg}} w_j(t)} + \frac{\gamma_r}{K_{cg}}$$

Draw cg_t *according to probability* $p_1^{i_{cg\ crit}}(t)$ ... $p_{K_{cg}}^{i_{cg\ crit}}(t)$

Receive normalised reward $x_{cg_t} \in (-1,1)$

$$x_{cg_t}^{i_{cg\ crit}} \leftarrow x_{cg_t}^{i_{cg\ crit}} + x_{cg_t}$$

$$\hat{x}_{cg_t}^{i_{cg\ crit}} \leftarrow \frac{\hat{x}_{cg_t}^{i_{cg\ crit}}}{p_{cg}^{i_{cg\ crit}}(t)}$$

$$\hat{G}_{cg}(t+1) \leftarrow \hat{G}_{cg}(t) + \hat{x}_{cg_t}^{i_{cg\ crit}}$$

$$w_{cg}(t+1) \leftarrow w_{cg}(t).e^{\gamma_r.\hat{x}_{cg_t}^{i_{cg\ crit}}}$$

$$t \leftarrow t+1$$

end if

end else if

Algorithm 2:Code Generator Selection

5.5. Program Selection Algorithm

The program selection algorithm relies on the reward g_{pr} acquired after several fuzzer iteration (i). Unlike code generators and mutators the reward acquired is the number of new programs over many fuzz iterations. As the number of programs are infinitely growing a caching algorithm is used simultaneously with the program selection algorithm.

The data pertaining to the program (pr) is acquired for each program executed with multiple mutations and code generators until a critical mass of iterations ($i_{pr\ crit}$) is reached.

The rewards g_{pr} is normalised as (x_{pr_t}) in the range $(-1,1)$

An accumulator for total normalised rewards ($x_{pr_t}^{i_{pr\ crit}}$) over $i_{pr\ crit}$ iterations and accumulates the normalised rewards (x_{pr_t}).

When $i_{pr\ crit}$ fuzz iterations are complete, the accumulated normalised reward ($x_{pr_t}^{i_{pr\ crit}}$) observed for each program (pr) randomly selected with probability ($p_{pr}(\mathrm{t})$). It is influenced with MAB algorithm's decision parameter (γ_r) to facilitate exploration and exploitation of the selected program. The probability derived is from the program weights ($w_{\mathrm{pr}}(\mathrm{t})$).

The algorithm's decision parameter (γ_r) changes over epochs (r).

When the reward $x_{pr_t} > 0$ then a new program is guaranteed to be found. At this point the caching algorithm and the new program (pr) is added to the main pool, the availability pool and is also added with a weight of 1 to the weighted action dictionary. If the cache is not full at this point i.e., $K_{pr} \neq K_{pr\ max}$ where K_{pr} is the total number of programs in the cache and $K_{pr\ max}$ is the total cache size. Then pr is added to the cache and continues to be populated till the cache is full.

The epochs (r) of the algorithm increase whenever the maximum estimated reward for the program ($max_{\mathrm{pr}}\hat{\mathrm{G}}_{\mathrm{pr}}(\mathrm{t})$) exceeds the difference $g_r - K_{pr}/\gamma_r$ where K_{pr} is the total number of programs in the cache and g_r is a best guess for the resulting reward.

MAB is restarted with a new decision parameter (γ_r) when a new epoch (r) begins.

While $max_{\mathrm{pr}}\hat{\mathrm{G}}_{\mathrm{pr}}(\mathrm{t})$ is less than or equal to $g_r - K_{pr}/\gamma_r$ the estimated accumulated reward over $i_{pr\ crit}$ iterations for the selected program ($\hat{x}_{pr_t}^{i_{pr\ crit}} = x_{pr_t}^{i_{pr\ crit}} / p_{pr}(\mathrm{t})$) is added to the total estimated reward for that mutator ($\hat{\mathrm{G}}_{pr}(t)$). The weight for the program ($w_{\mathrm{pr}}(\mathrm{t})$) is also exponentially updated with γ_r times the estimated accumulated reward over critical mass iterations for the drawn mutator ($\hat{x}_{pr_t}^{i_{pr\ crit}}$). After the weight update the algorithm's trial (t) is complete.

When an epoch (r) is reached the caching algorithm checks for the need of a cache reset. A cache reset counter (crc) is updated each time an epoch is reached. If the cache reset counter reaches the threshold value (crc_{thresh}), the cache is cleared and repopulated with random programs from the availability pool. At this point, the MAB program selection algorithm is reset and the trials (t) and epochs (r) reset.

Algorithm 3: Program Caching

for all $r = 0,1,2,3\ldots..$

for all $i = 1,2,3\ldots..$

if $x_{pr_t} > 0$

[main pool] ← pr

{weighted actions} ← $(pr, w_{pr}(t) \leftarrow 1)$

[availability pool] ← pr

if {***cache***} *is not full i.e.,* $K_{pr} \neq K_{pr\ max}$

[***cache***] ← pr

end if

end if

if $i = i_{crit\ pr}$

if $max_{pr}\hat{G}_{pr}(t) > g_r - \frac{K_{pr}}{\gamma_r}$

$crc \leftarrow crc + 1$

if $crc > crc_{thresh}$

[cache] ← [ ]

while $K_{pr} \neq K_{pr\ max}$

Restart program selection and program caching with
$t \leftarrow 1$
$r \leftarrow 0$
Pick random program (pr) *from* $\{availability\ pool\}$
[***cache***] ← pr

end while

end if

end if

end if

end for

end for

Algorithm 3:Program Caching Algorithm

Algorithm 4: Program Selection

$\hat{G}_{pr}(0) \leftarrow 0 \quad for\ pr = 0,1,2\ldots\ldots K_{pr}$

$t \leftarrow 0$

for all $r = 0,1,2,3\ldots..$

for all $i = 1,2,3\ldots..$

$g_r \leftarrow \frac{K_{pr}\ ln(K_{pr})}{(e-1)}.4r$

$\gamma_r \leftarrow min\left(1, \sqrt{\frac{K_{pr}\ ln(K_{pr})}{(e-1).g(r)}}\right)$

if $i = i_{pr\ crit}$

$p_{pr}(t) \leftarrow \frac{w_{pr}(t)}{\sum_{j=1}^{K_{pr}} w_j(t)}$

Draw pr_t *according to probability* $p_1(t)$ ... $p_{K_{pr}}(t)$

Receive normalised reward $x_{pr_t} \in (-1,1)$

$x_{pr_t}^{i_{pr\ crit}} \leftarrow x_{pr_t}^{i_{pr\ crit}} + x_{\mathrm{pr_t}}$

end if

else if $i \neq i_{pr\ crit}$

if $max_{pr}\hat{G}_{pr}(t) \leq g_r - \frac{K_{pr}}{\gamma_r}$

$p_{pr}^{i_{pr\ crit}}(t) \leftarrow \frac{(1-\gamma_r).w_{pr}(t)}{\sum_{j=1}^{K_{pr}} w_j(t)} + \frac{\gamma_r}{K_{pr}}$

Draw pr_t *according to probability* $p_1^{i_{pr\ crit}}(t)$ ... $p_{K_{pr}}^{i_{pr\ crit}}(t)$

Receive normalised reward $x_{pr_t} \in (-1,1)$

$x_{pr_t}^{i_{pr\ crit}} \leftarrow x_{cg_t}^{i_{pr\ crit}} + x_{pr_t}$

$\hat{x}_{pr_t}^{i_{pr\ crit}} \leftarrow \frac{\hat{x}_{pr_t}^{i_{pr\ crit}}}{p_{pr}^{i_{pr\ crit}}(t)}$

$\hat{G}_{pr}(t+1) \leftarrow \hat{G}_{pr}(t) + \hat{x}_{pr_t}^{i_{pr\ crit}}$

$w_{pr}(t+1) \leftarrow w_{pr}(t).e^{\gamma_r.\hat{x}_{pr\ t}^{i_{pr\ crit}}}$

$t \leftarrow t+1$

end if

end else if

end for

end for

Algorithm 4:Program Selection

5.6. Long Term Stability of MAB

To maintain stability for long runs, the weights of the code generator and mutator actions are rescaled at $i^{th}{}_{action\ crit}$ iteration respectively; After each epoch change, the total estimated reward $\hat{G}_{action}(t))$ with the max value that satisfies the epoch change condition ($max_{\text{action}}\hat{G}_{\text{action}}(\text{t})$ less than or equal to $g_r - K_{action}/\gamma_r$) is reset to zero. This ensures that there is no runoff for any actions weight $w_{\text{action}}(\text{t})$ allowing lower weighted actions a chance at moving out from the exploration phase to exploitation phase.

Additionally, there is a restart threshold limit on iterations that restarts the entire MAB process from epoch zero preserving the last observed weights for the code generator and mutator actions. This ensures that the MAB process is recalibrated for later stage fuzzing with fresh exploration and exploitation.

6.Evaluation

In this section, the changes introduced over the last two sections are evaluated. For demonstrating practical performance two of JavaScript engines – JavaScript Core (JSC) and V8, of the most prominent browsers Safari and Google Chrome respectively are used as fuzzing targets.

The evaluation attempts to answer the following questions:

Q1 Can the proposed modification to mutation and code generation generate faster coverage growth? (Section 6.4)

Q2 Can the proposed modification to the program corpus (MAB Corpus) generate faster coverage growth? (Section 6.5)

Q3 Can the combined efforts of the proposed modifications to mutation and code generation and the MAB Corpus generate faster coverage growth? (Section 6.6)

6.1. Fuzzing in Various Setups

The evaluation of the performance of the fuzzer is measured over various setups of the fuzzer.
The first evaluation setup series is for a single node of Fuzzilli and has the following three setups defined for the fuzzer:
Fuzzilli with code generation and mutation modifications only: This setup runs the fuzzer with MAB based coverage guidance to code generation and mutation tasks operating on the default (basic) corpus.
Fuzzilli with MAB corpus only: This setup runs the fuzzer with MAB based coverage guidance to the corpus management scheduler (referred as MAB Corpus).
Fuzzilli with code generation, mutation, and MAB corpus: This setup runs the fuzzer with combined fuzzing components of MAB based coverage guided code generation and mutation along with the MAB Corpus.

The single node series setups are evaluated for the two target JavaScript Engines **Safari / JavaScript Core (JSC)** and **Google Chrome / V8,** respectively**.**

6.2. Testing Setup

The evaluation of the fuzzer was performed using E2 General-purpose machine series of the Google Cloud Platform (GCP). The instance type used was e2-standard-2 with 2 vCPUs and 8GB RAM. [26]

The following are the key configuration parameters of a Fuzzilli instance:

- 400m/s is the fuzzer timeout for execution of a program
- The default arbitrary weights of Fuzzilli's components code generators and mutators are used as the initial weights for MAB instead of a uniform weighting distribution
- The test setup is for a single node setup of Fuzzilli only without any distributed instances (no master-worker network or threading nodes)
- Lastly no compiled/imported seeds were used in the corpus.

The docker images to run the containers with the fuzzer was made available from the Fuzzilli repo at https://github.com/googleprojectzero/fuzzilli/tree/main/Cloud/Docker.

The fuzzing targets are obtained from the following sources:

Safari / JavaScript Core (JSC):
Repository URL: https://github.com/WebKit/WebKit/
Commit hash: 4110e1b44a345737cdb807d36572c8714e90c5d0

Google Chrome / V8:
Repository URL: https://chromium.googlesource.com/chromium/src/
Tag: 10.0.139.15
Commit hash: b2490f598d9debd563bc32191cb3f777a16cbbb0

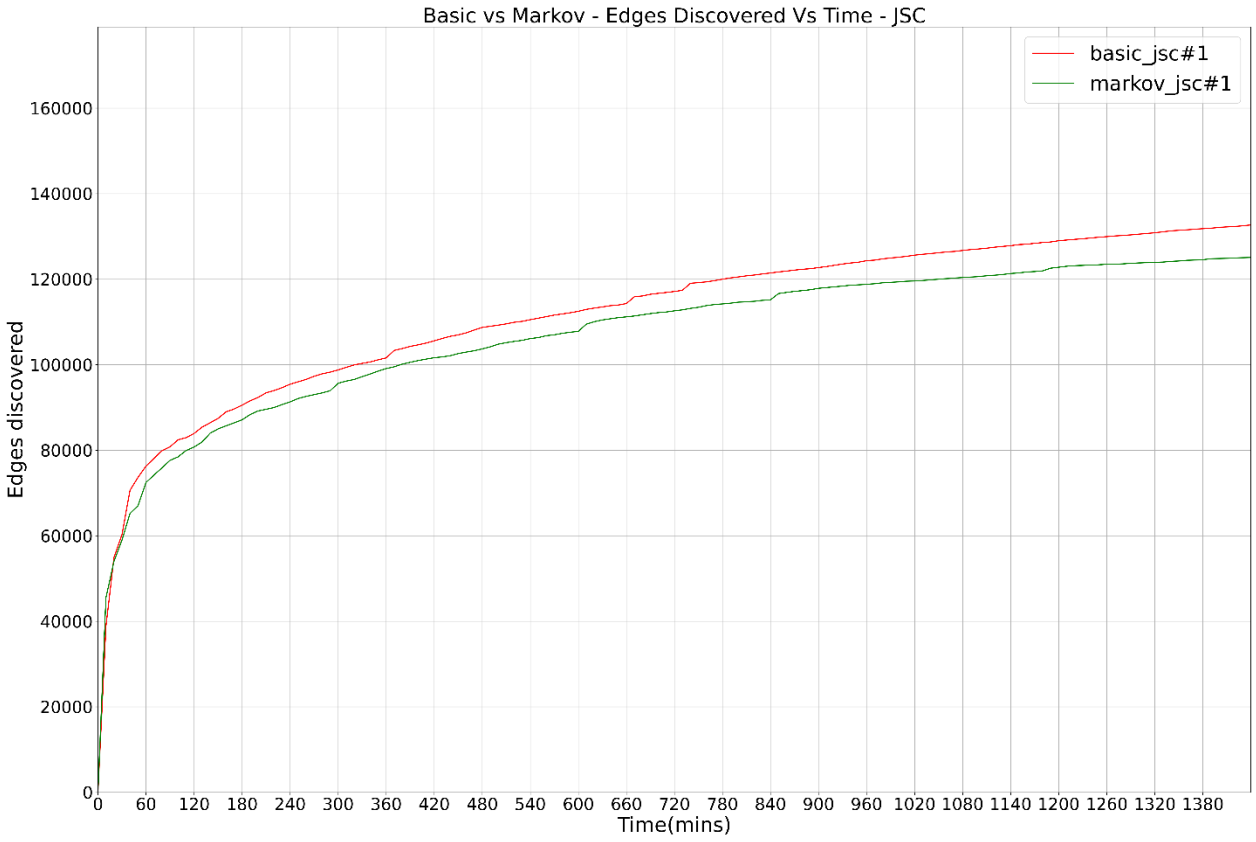
The fuzzer repository is available at https://github.com/googleprojectzero/fuzzilli.git modifications to the fuzzer are introduced over the commit hash 3f0d246a47f39e066ab560f3bb23e2fe47a25850 as its base.

6.3. Evaluation against Baseline Performance

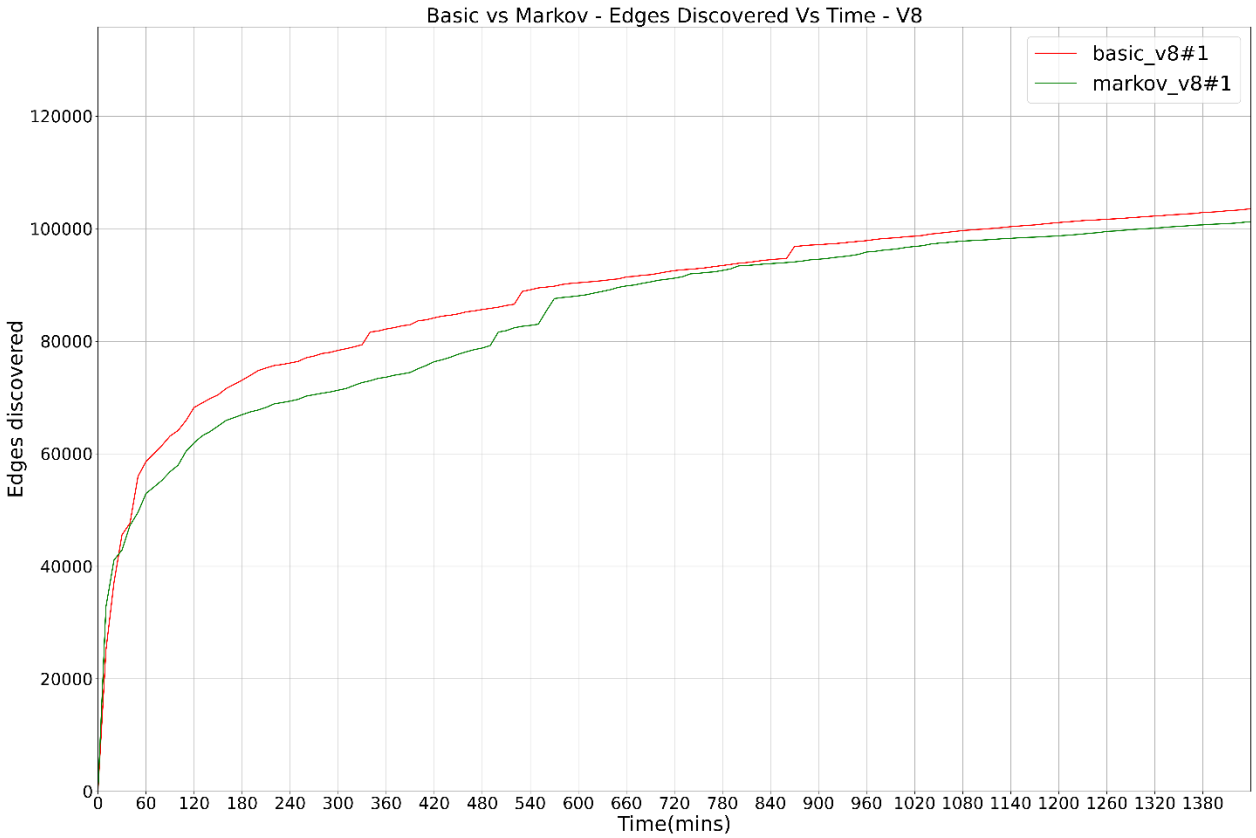
As established in Fuzzilli overview there exist two corpus scheduling mechanisms that are guided by coverage. These to corpora are the default (Basic) and Markov Corpus. The performance evaluation of the proposed modifications is done against the performance of these two existing corpora of Fuzzilli.
A few observations

- The graphs below show the coverage growth found by the default Fuzzilli over 24hrs against the two JavaScript engines under test JSC and V8
- During the first few hours of fuzzing, the fuzzer demonstrates rapid coverage growth but towards the end of the 24-hour period the growth starts to taper off.
- Over longer periods of fuzzing the coverage found over time by both Basic and Markov corpus converges.

It was observed that the performance of both corpus schedulers is near identical if the fuzzer instances are run for longer than 24 hours.



Graph 1:Basic Corpus vs Markov Corpus (JSC)



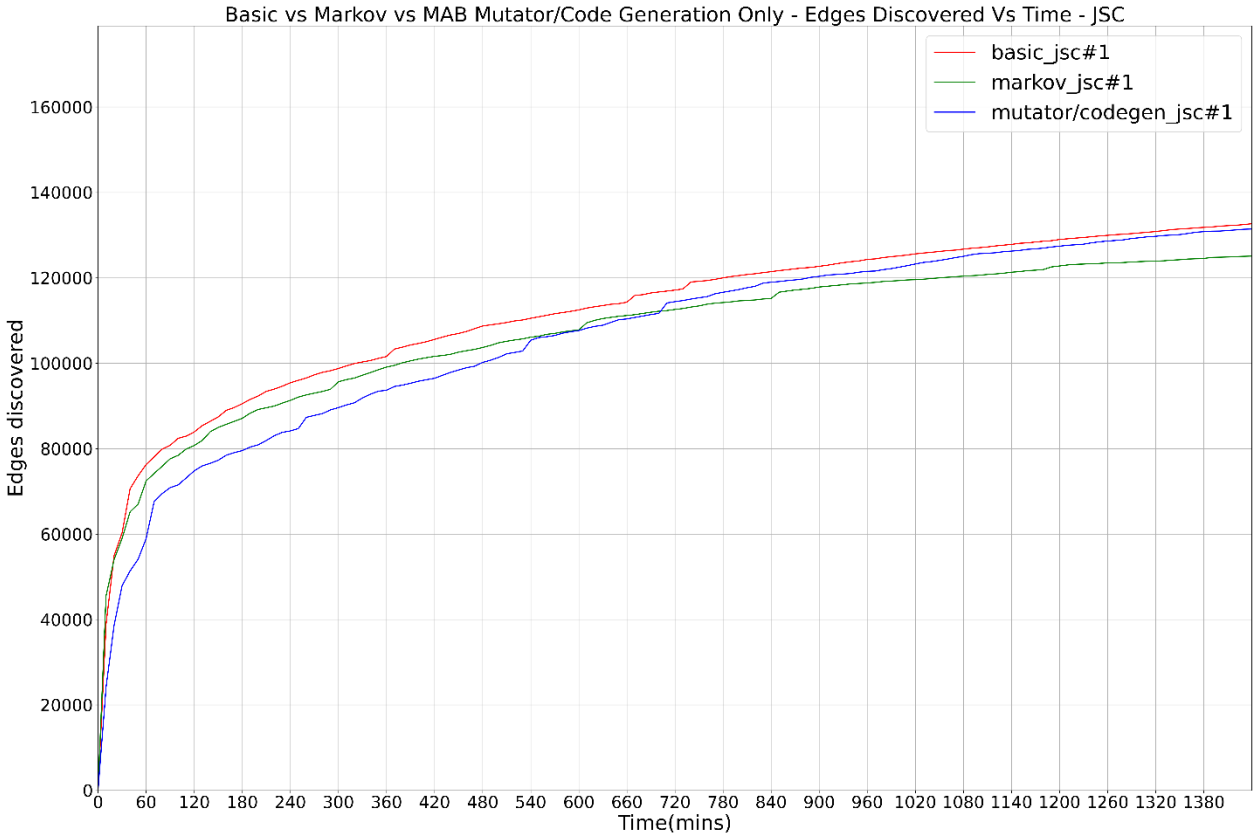
Graph 2: Basic Corpus vs Markov Corpus (V8)

6.4. Code Generation/Mutation Performance

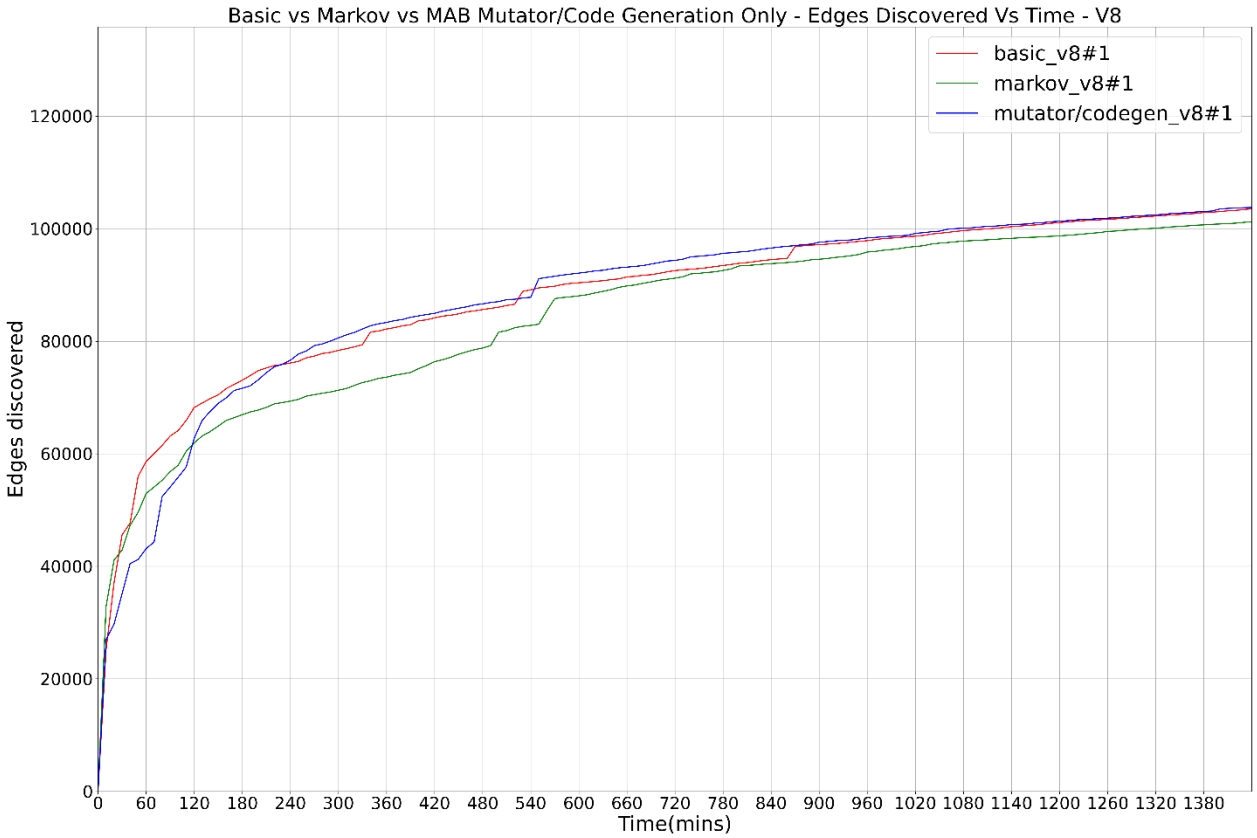
The graphs below show the code generation and mutation enhancements performance on a Basic corpus. The code generation and mutation changes show expected gradual steady growth over time in edge discovery across each engine.
The growth is logarithmic in nature and rapid growth is observed in the initial hours of fuzzing, gradually tapering as edge discovery starts to require more complexity in the type of generated programs.
In comparison to the basic and Markov corpus, it gradually reaches a similar number of paths discovered after 12 hours of fuzzing. Early in the fuzzing process the algorithm under performs against the Basic and Markov corpus but as the algorithm learns from the coverage observed, an optimal weight distribution is found and in the later stages of the 24-hour period, achieves similar performance to the basic corpus and demonstrates an improvement over the Markov corpus.

However, as mentioned in the previous section, the performance of the three variants over time converges and the coverage achieved over time after a 24hr period does not vary significantly.

A side effect of the randomness of grey box fuzzing is that it sets the starting points for each instance at different areas in the fuzzing target. As a result, the complexity of programs because of code generation and mutations can be extremely favourable or unfavourable for a selected starting point of fuzzing.



Graph 3:Basic Corpus vs Markov Corpus vs MAB Mutator/Code Generation Only (JSC)

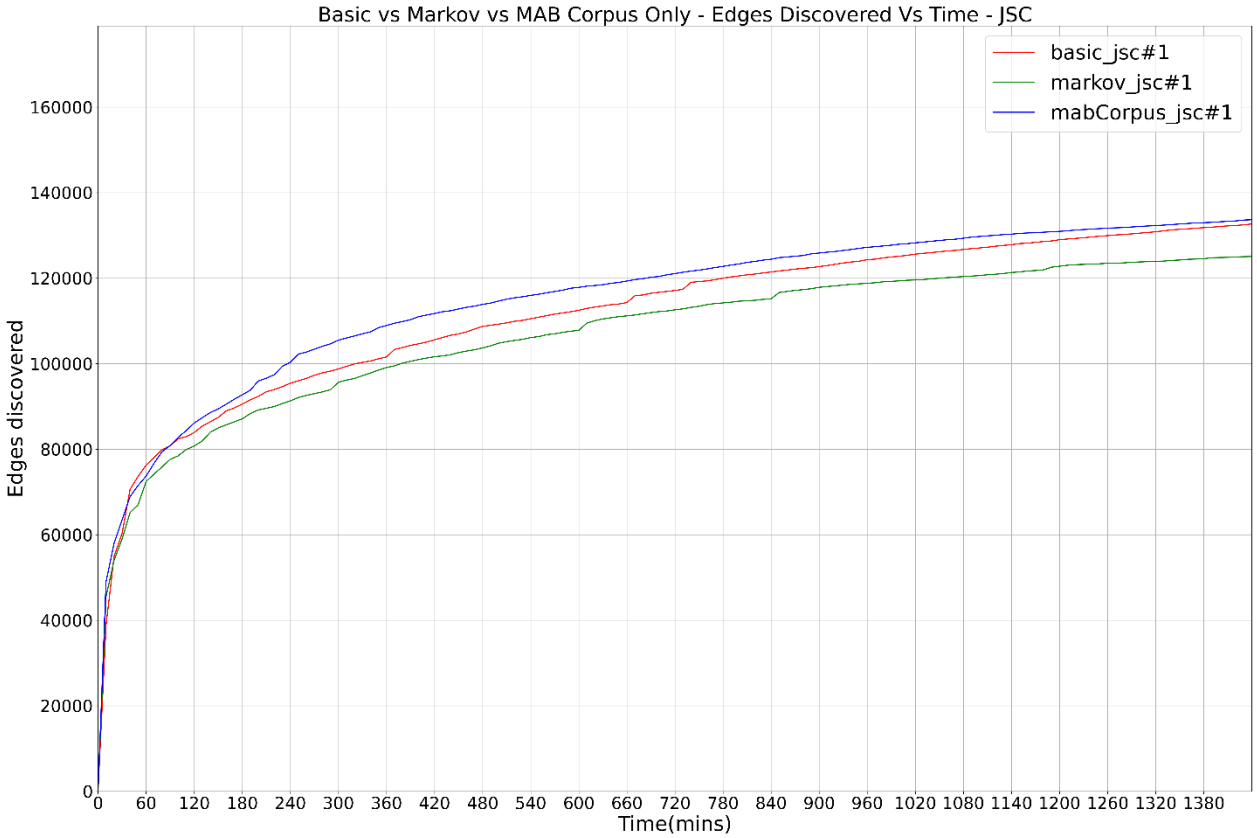


Graph 4: Basic Corpus vs Markov Corpus vs MAB Mutator/Code Generation Only (V8)

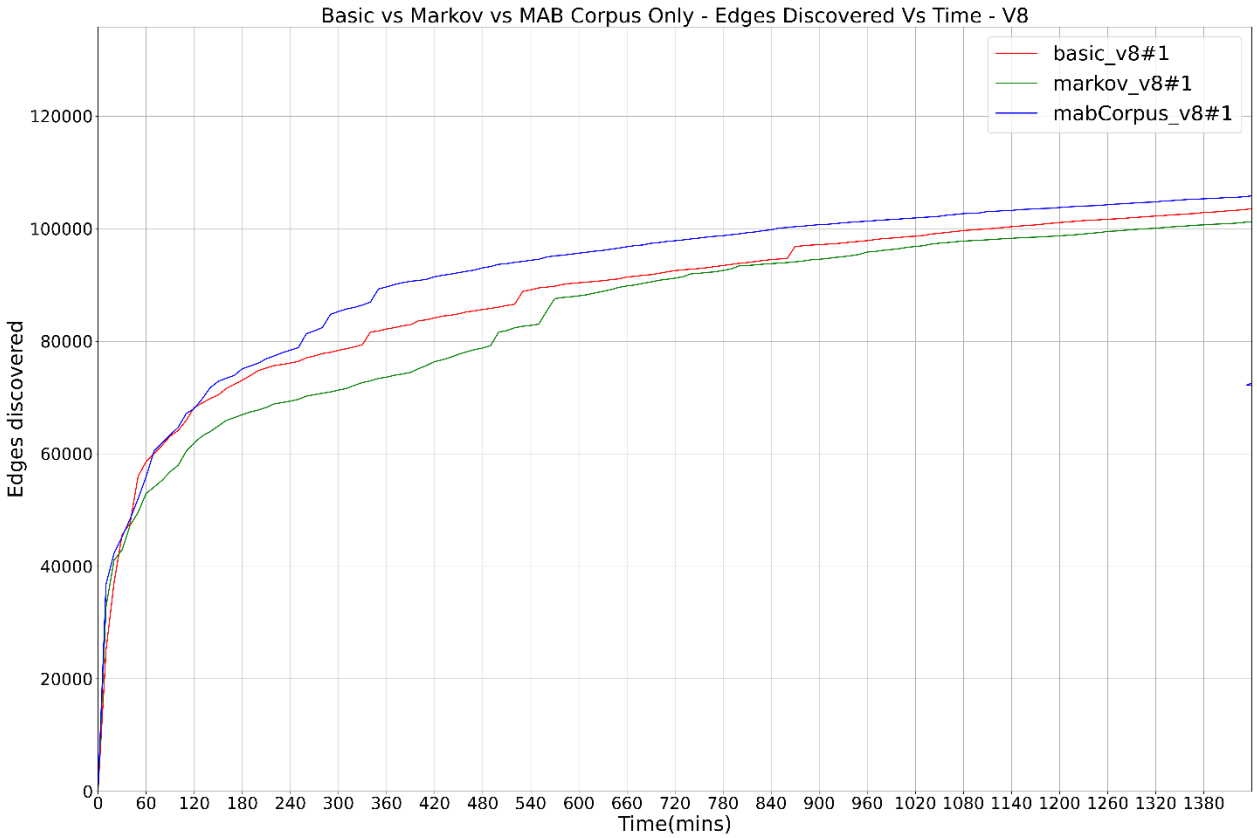
6.5. MAB Corpus Performance

The MAB corpus management scheduler implementation shows once again expected gradual steady growth over time in edge discovery across each engine.
Once again, the growth is logarithmic in nature and steep in the initial hours of fuzzing, and gradual tapering after. However, the convergence is not quickly reached the way it does for code generation and mutations. This is because of the nature of the programs in the corpus, as compared to the choice of mutation or code generation task. The corpus prioritises the use of programs with potential to generate high coverage. This is achieved through exploring the likelihood of new programs generated. The likelihood, that a particular program selected from the corpus for fuzzing will generate coverage finding programs. This has more specific guidance as opposed to just using edges discovered. Additionally, the programs that are picked less often are given fair opportunity to increase their own likelihoods of generating new programs. As a result, there is always an optimised selection pool of programs available for the fuzzer.

In comparison to the basic and Markov corpus, the MAB Corpus reaches a higher number of edges discovered through 24 hours of fuzzing with a faster pace.
Once again, the randomness of grey box fuzzing sets the starting points for each instance at different areas in the fuzzing target. The pace of edge discovery can be extremely favourable or unfavourable for a selected starting point of fuzzing.



Graph 5:Basic Corpus vs Markov Corpus vs MAB Corpus Only (JSC)



Graph 6: Basic Corpus vs Markov Corpus vs MAB Corpus Only (V8)

The corpora in Fuzilli in addition to new coverage discovery is efficient only if it generates “meaningful” programs that lead to coverage discovery.

Meaningfulness of a program goes beyond just discovering new coverage. Simply put, it is the idea that the resulting code does not cause repetitive internal state transitions of the engine regardless of its input types and surrounding code. The minimisation process with the default mutation engine ensures a degree of meaningfulness in programs generated by Fuzzilli.

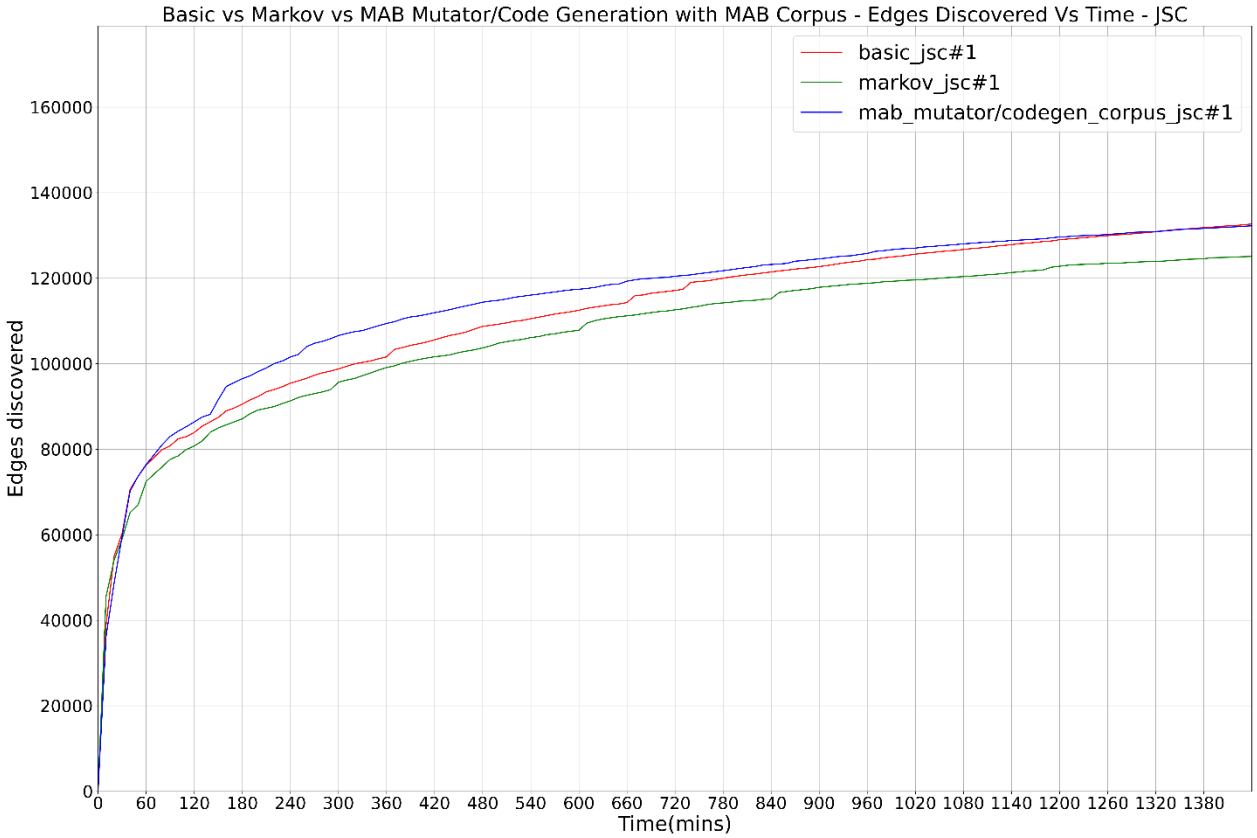
The Markov Corpus in Fuzzilli establishes a chaining of edge discovery through associating programs that establish a connected link of sequential coverage growth discovery across programs. This is in accordance with establishing a Markov chain of sequential edge discovery with deterministic probability from programs that lead to further discovery. The edge associations further improve the odds of meaningfulness being achieved in programs generated. However due to the difficulty of maintaining a corpus state of programs with potential branching links to all connected programs capable of edge discovery. The corpus is restricted to simple chains established through binding of a single discovered edge and a program in the corpus. This results in loss of potential connected link branches to new edges from previously discovered edges.
Unlike the reward systems guided directly by edge discovery and coverage growth, the MAB corpus provides a higher degree of guidance towards the selection of trending programs for new program discovery. This indirectly incorporates a form of chaining, albeit through exploration and exploitation of the next program to generate. Therefore, the observed coverage growth is faster than the Markov or Basic corpus.

6.6. Combined Performance

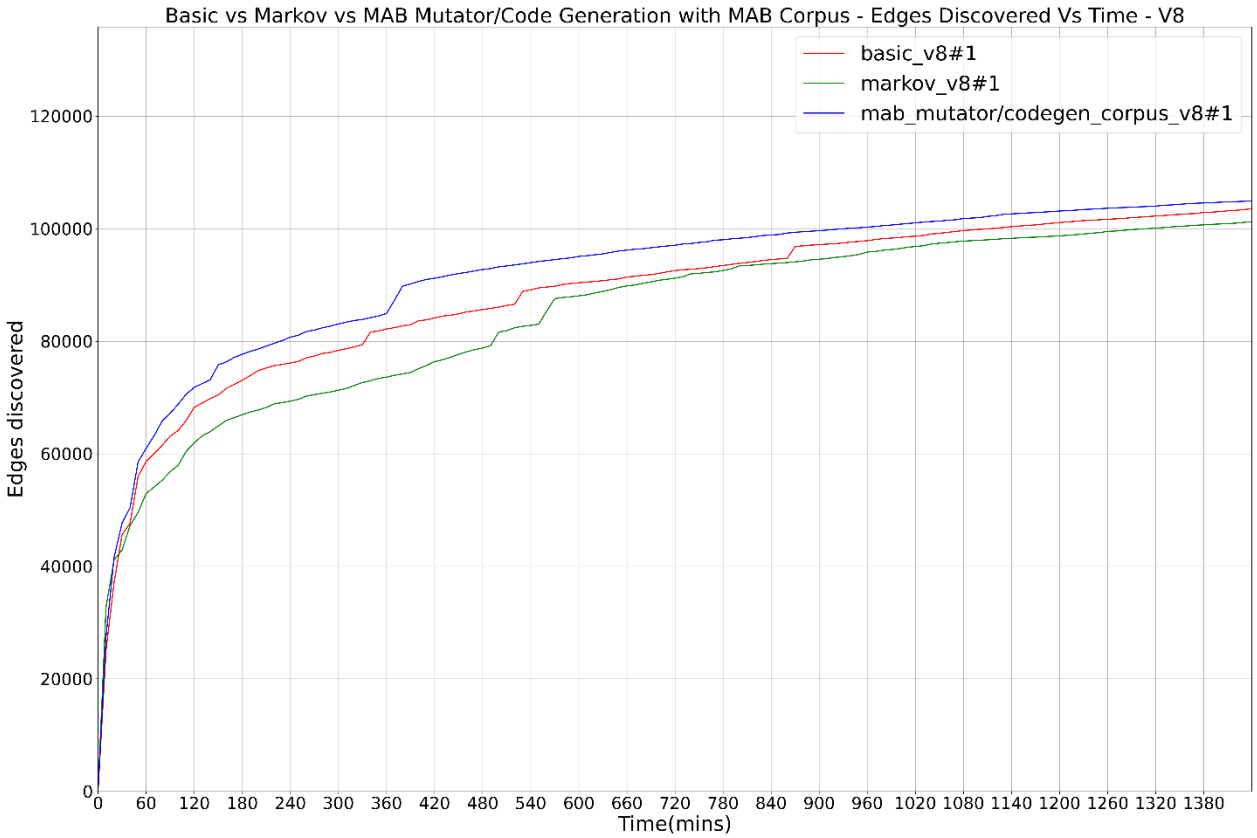
The performance of the combined efforts of code generation and mutation with MAB Corpus is faster than Basic Corpus or Markov Corpus in the first 12-hours of fuzzing. The convergence of coverage with the Basic Corpus however is seen to be faster toward the end of the 24-hours. The results indicate a significant increase in speed of coverage discovery but also a quicker loss of program complexity. This showcases the prioritisation of the MAB Corpus' programs in generating higher coverage programs as well as optimal exploration and exploitation of code generation and mutations in discovering new edges.

The MAB corpus is only able to aid in faster edge discovery when there is consistency of new programs being discovered. The decrease in program complexity eventually leads to a lower program discovery to invocation ratio which exhausts the potential of exploiting programs for coverage quickly. Therefore, there must be external intervention after the 24-hour mark when coverage is no longer increasing.
To improve the discovery for long runs the MAB process for code generation and mutation is restarted after a fixed threshold of iterations to leverage the randomness of ideal starting positions when fuzzing.

The MAB Corpus periodically resets its MAB process when it is unable to discover new programs for coverage. The corpus scheduler will reset if the number of epochs elapsed discovering new programs reaches a threshold beyond which performance faulters. This gives the initial programs in the cache enough time to be explored before regenerating the next set of programs in the cache



Graph 7: Basic Corpus vs Markov Corpus vs MAB Mutator/Code Generation with MAB Corpus (JSC)



Graph 8: Basic Corpus vs Markov Corpus vs MAB Mutator/Code Generation with MAB Corpus (V8)

A significant amount of effort was expended in a new corpus to provide more optimal program choices for enhancing the performance of code generators and mutators. On its own code generation and mutation enhancements take longer to target high coverage with construction of a ‘meaningful’ program taking several hours to obtain from suboptimal programs served up to be mutated. This can be seen by the late-stage performance of the code generators and mutators acting on the Basic corpus.

7.Observations and Limitations of Implementation

This section discusses the performance of the evaluations of the fuzzer in the previous section. Various observations of key performance issues in the implementation are discussed below:

Limitations of code generators. The code generators are inherently limited in their ability to evolve, and the resulting coverage growth has eventual convergence with Basic and Markov corpus. This is expected because the generators are based on generic templates of code blocks. These take longer to create "meaningful" conditions for improved coverage discovery.

Additional guidance needed. It is not enough to guide the code generation and mutations by fuzzing towards coverage growth. It must also be guided towards attaining 'meaningfulness'. Additional guidance to coverage discovery that can identify the context of edges discovered and steer fuzzing towards connected newer edges would potentially improve performance of the fuzzer.

Fuzzilli Hybrid Engine. Fuzzilli's experimental hybrid engine attempts to overcome some of the code generation pitfalls by introducing program templates that preserve coverage discovery conditions and provide preservable aspects on mutation of the program.

However, there is significant manual effort required to make these templates, requiring analysis and identification of patterns within the fuzzing target. The time taken for construction of these templates to be used as generators does not scale fast enough with the fuzzing target of a JavaScript Engine; given that it is constantly updated with newer features.

The coverage-based guidance for mutators/code generators is thus less performative in comparison to the MAB Corpus, as it introduces an unavoidable infusion of meaningless programs created and discarded by the fuzzer. This leads to longer wait times between significant coverage discovery.

Fuzzing with Initial Weights. Initial attempts at implementation of Fuzzilli with MAB utilised uniform starting weights of 1. This was not productive or performative as it would take longer than 24-hours to accumulate an optimized weight distribution for performance. Furthermore, it was exceedingly slow when performing code generation as the incidence of unsuccessful programs at finding coverage was higher. The randomness of starting points while fuzzing, using a staggered weight choice for code generation was found to be the only way to kickstart a faster coverage growth rate.

8.Proposed Optimisations and Related Work

It is observed from the implementation of the guidance mechanism that coverage on its own is an insufficient metric for guidance, since the performance isn't remarkably better than basic or Markov corpus in Fuzzilli. With guidance to code generation and mutations there is still a lot of uncertainty prevalent in the capability of mutated and generated programs being not 'meaningful' enough to generate new coverage after the observed convergence in coverage growth after long runs of Fuzzing.

To address these concerns the following optimisations are proposed and the related work in the field is mentioned

Data Dependency Graphs. Alternative approaches would be to use Data Dependency Graphs in addition to coverage guidance for better edge discovery [27].

Feature Scoring Aspects. Similarly, annotation of interesting aspects in programs could be done. This would act as a feature scoring mechanism to help provide target specific guidance [28, 29, 15].

Running the Fuzzer with Imported Corpus. As mentioned before, PoCs and test suites imported into the corpus may help alleviate the challenges with complexity of code generators to generate programs that are 'meaningful'. This would provide targeting for specific areas that the PoCs and test suites act upon which the code generation fuzzer component is usually not able to reach.

9.Summary and Future Work

The aim of the project was to implement an intelligent selection mechanism for code generation, mutation, and seed selection tasks to aid JavaScript engine fuzzing using Fuzzilli and to evaluate performance in coverage growth. The Multi-armed bandit algorithm was chosen to dynamically rebalance weights for these components and help explore the search space for the selection of these tasks optimally. As a result, the coverage gained over time was faster in comparison to the basic and Markov corpus of Fuzzilli.
However, there was a convergence in coverage growth over time and all the three variants have near identical coverage growth over long fuzzing runs. This convergence was attributed to the limited evolution of code generators that having inadequate complexity to trigger new coverage discovery at late stages of fuzzing. This inadvertently leaks into the MAB corpus and reduces its ability to priorities optimal program selection. Therefore, coverage alone is not adequate at increasing the overall coverage growth rate and additional metrics will have to be needed to better guide the fuzzing. The proposed optimisations and related work (Section 8) suggests future enhancements of the fuzzer which can assist with better guidance along with related works in Java Script Fuzzing.

As the project continues to develop future reported bugs and improvements will be reported at https://deamonspawn.github.io/amenezes.github.io/

References

[1] M. Pistoia, S. Chandra, S. J. Fink and E. Yahav, “A survey of static analysis methods for identifying security vulnerabilities in software systems,” *IBM systems journal,* vol. 46, no. 2, pp. 265-288, 2007.

[2] R. Baldoni, E. Coppa, D. C. D’elia, C. Demetrescu and I. Finocchi, “A survey of symbolic execution techniques,” *ACM Computing Surveys (CSUR),* vol. 51, no. 3, pp. 1-39, 2018.

[3] Y. Wang, W. Cai, P. Lyu and W. Shao, “A combined static and dynamic analysis approach to detect malicious browser extensions,” *Security and Communication Networks,* vol. 2018, 2018.

[4] Y. Shin and L. Williams, “An empirical model to predict security vulnerabilities using code complexity metrics,” in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement,* 2008, pp. 315-317.

[5] J. Lee, “Issue 1438: Microsoft Edge: Chakra: JIT: ImplicitCallFlags checks (CVE-2018-0840),” [Online]. Available: https://bugs.chromium.org/p/project-zero/issues/detail?id=1438. [Accessed January 2022].

[6] J. Lee, “Issue 1565: Microsoft Edge: Chakra: JIT: ImplicitCallFlags check (CVE-2018-8288),” [Online]. Available: https://bugs.chromium.org/p/project-zero/issues/detail?id=1565. [Accessed January 2022].

[7] W3Techs, “Usage statistics of JavaScript as client-side programming language on websites,” [Online]. Available: https://w3techs.com/technologies/details/cp-javascript. [Accessed April 2022].

[8] F. Pizlo, “Speculation in JavaScriptCore,” Webkit.org, [Online]. Available: https://webkit.org/blog/10308/speculation-in-javascriptcore/. [Accessed April 2022].

[9] Webkit.org, “WebKit Blog,” [Online]. Available: https://webkit.org/blog/. [Accessed April 2022].

[10] Google, “V8 Blog,” [Online]. Available: https://v8.dev/blog. [Accessed April 2022].

[11] A. Burnett and S. Groß, “Attacking JavaScript Engines in 2022,” [Online]. Available: https://saelo.github.io/presentations/offensivecon_22_attacking_javascript_engines.pdf. [Accessed April 2022].

[12] MozillaSecurity, “MozillaSecurity - jsfunfuzz,” [Online]. Available: https://github.com/MozillaSecurity/funfuzz. [Accessed April 2022].

[13] H. Han, D. Oh and S. K. Cha, “CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines.,” in *NDSS,* 2019.

[14] Google, “Fuzzilli,” [Online]. Available: https://github.com/googleprojectzero/fuzzilli. [Accessed January 2022].

[15] S. Park, W. Xu, I. Yun, D. Jang and T. Kim, “Fuzzing javascript engines with aspect-preserving mutation,” in *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2020, pp. 1629-1642.

[16] J. Wang, B. Chen, L. Wei and Y. Liu, “Superion: Grammar-aware greybox fuzzing,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 2019, pp. 724-735.

[17] C. Holler, “Grammar-based interpreter fuzz testing,” *Master's Thesis Dissertation, Department of Computer Science, Saarland University,* 2011.

[18] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi and D. Teuchert, “NAUTILUS: Fishing for Deep Bugs with Grammars.,” in *NDSS,* 2019.

[19] M. Böhme, V.-T. Pham and A. Roychoudhury, “Coverage-based greybox fuzzing as markov chain,” *IEEE Transactions on Software Engineering,* vol. 45, no. 5, pp. 489-506, 2017.

[20] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song and R. Beyah, “{MOPT}: Optimized mutation scheduling for fuzzers,” in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1949-1966.

[21] L. Zhao, Y. Duan, H. Yin and J. Xuan, “Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing,” in *NDSS*, 2018.

[22] S. Karamcheti, G. Mann and D. Rosenberg, “Adaptive grey-box fuzz-testing with thompson sampling,” in *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*, 2018, pp. 37-47.

[23] P. Auer, N. Cesa-Bianchi, Y. Freund and R. E. Schapire, “The nonstochastic multiarmed bandit problem,” *SIAM journal on computing*, vol. 32, no. 1, pp. 48--77, 2002.

[24] D. Wang, Z. Zhang, H. Zhang, Z. Qian, S. V. Krishnamurthy and N. Abu-Ghazaleh, “SyzVegaS: Beating Kernel Fuzzing Odds with Reinforcement Learning,” in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2741--2758.

[25] W. Parks, “Fuzzilli - Implements Markov corpus manager#171,” Google, [Online]. Available: https://github.com/googleprojectzero/fuzzilli/pull/171. [Accessed January 2022].

[26] Google, “E2 machine series,” [Online]. Available: https://cloud.google.com/compute/docs/general-purpose-machines#e2_machine_type. [Accessed April 2022].

[27] A. Mantovani, A. Fioraldi and D. Balzarotti, “Fuzzing with Data Dependency Information,” [Online]. Available: https://www.researchgate.net/profile/Andrea-Fioraldi/publication/358931158_Fuzzing_with_Data_Dependency_Information/links/621e3dd2ef04e66eb74d40fe/Fuzzing-with-Data-Dependency-Information.pdf. [Accessed April 2022].

[28] R. Padhye, C. Lemieux, K. Sen, M. Papadakis and Y. Le Traon, “Semantic fuzzing with zest,” in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 329-340.

[29] M. Rajpal, W. Blum and R. Singh, “Not all bytes are equal: Neural byte sieve for fuzzing,” *arXiv preprint arXiv:1711.04596*, 2017.

[30] Google, “Fuzzilli - Mutator Weights,” [Online]. Available: https://github.com/googleprojectzero/fuzzilli/blob/c1a3848b0cdc713b40a326bb76f791066128837e/Sources/FuzzilliCli/main.swift#L367-L374. [Accessed January 2022].

[31] Google, “Fuzzilli - Code Generator Weights,” [Online]. Available: https://github.com/googleprojectzero/fuzzilli/blob/c1a3848b0cdc713b40a326bb76f791066128837e/Sources/FuzzilliCli/CodeGeneratorWeights.swift. [Accessed January 2022].

Appendix

Setup of JSC engine

To build JavaScriptCore (jsc) for fuzzing:

1. Clone the WebKit mirror from https://github.com/WebKit/webkit
2. Apply Patches/*. The patches should apply cleanly to the git revision specified in ./RE-VISION (*Note*: If you clone WebKit from `git.webkit.org`, the commit hash will differ)
3. Run the fuzzbuild.sh script in the webkit root directory
4. FuzzBuild/Debug/bin/jsc will be the JavaScript shell for the fuzzer

Setup of V8 engine

To build v8 for fuzzing:

1. Follow the instructions at https://v8.dev/docs/build
2. Run the fuzzbuild.sh script in the v8 root directory
3. out/fuzzbuild/d8 will be the JavaScript shell for the fuzzer

Git Commit Log for Source code

Source code file:

File name: xxxxxxxxProjMScAI.zip

SHA256: 7ce701b1fcf9bff71081b9907a56684ae76a2cb4ab2827042d4ca6855f5dc18f

Git log:

The commit for the modified changes for this project can be found in the Source code at 00568fa14c74bf79662e128bb21dc81a13cd08d9 on top of the base commit 3f0d246a47f39e066ab560f3bb23e2fe47a25850 form the Fuzzilli Repository

https://github.com/googleprojectzero/fuzzilli