# AFLSmart++: Smarter Greybox Fuzzing

Van-Thuan Pham
*School of Computing and Information Systems*
*The University of Melbourne*
Melbourne, Australia
thuan.pham@unimelb.edu.au

***Abstract*—Model/grammar-based greybox fuzzing has gained attention from both industry and academia due to its capability of discovering bugs/vulnerabilities in programs taking highly-structured inputs. AFLSmart is a specific example. It is a model-based fuzzer that focuses on chunk-based file formats like PNG, PDF and WAV. Its effectiveness is enabled by carefully-designed high-level mutation operators—that work at data chunk levels—and other heuristics such as its validity-based power schedule and deferred cracking mechanism. In this work, we present an extension of AFLSmart in which we explore some design options to (i) support structure-aware low-level mutation operators—that work at bit-byte-word-dword levels—and (ii) improve AFLSmart's usability and applicability with the so-called composite input model. The extension is called AFLSmart++ and it was evaluated independently—along with 11 other fuzzers—on the Google FuzzBench in a large-scale competition setup. The results show that AFLSmart++ secures the 3rd place in terms of bug finding but it ranks 11th based on its code coverage achievement.**

***Index Terms*—fuzzing, structure-aware fuzzing, software security, vulnerability discovery**

## I. Introduction

Fuzzing is an automated process of repeatedly generating (random) inputs (i.e., test cases) and feeding them to the system under test (SUT) to cover more lines of code and discover bugs. For instance, to test the Adobe PDF Reader, a fuzzer would take some sample PDF files, and modify/mutate them to generate million new valid and corrupted files. While feeding those files to the SUT, the fuzzer observes the program behaviours. A program crash indicates a potential security bug in the implementation and the bug-triggering input will be kept for further analyses (e.g., program debugging and fixing). Depending on whether it is aware of program structure, a fuzzing tool can be classified as white-,grey-, or black-box fuzzer. Grey-box fuzzing, as implemented in AFL [1] and libFuzzer [2], is arguably the most popular approach.

Due to its scalability and effectiveness, (grey-box) fuzzing has been applied to discover thousands of bugs/vulnerabilities in large real-world systems including core libraries [3], database management systems [4], web browsers [5], network protocols [6]–[8], web APIs [9], [10]. The technique has also received tremendous attention from the research community, demonstrated by hundreds of papers published at top venues in Computer Security and Software Engineering [11].

Researchers have been working on improvements to grey-box fuzzing in different aspects [11], [12]. One of the active research topics is to improve the quality and diversity of the generated inputs/test cases and leveraging input structure/-grammar is a promising direction (LibProtobuf-Mutator [13], AFLSmart [14], Nautilus [15], Superion [16]).

AFLSmart [14] is a specific structure-aware grey-box fuzzer that has a focus on programs taking chunk-based file formats like PNG, PDF, WAV etc. It leverages a high-level structural representation of the seed files to generate new files and uses higher-order mutation operators that work on the virtual file structure rather than on the bit level. This allows AFLSmart to explore new input domains while maintaining file validity. The fuzzer uses a validity-based power schedule so that it can spend more time generating files that are more likely to pass the parsing stage of the program, which can expose vulnerabilities much deeper in the processing logic. Experimental results show improvements of AFLSmart over its baseline AFL [1].

However, the current implementation of AFLSmart has two limitations:

First, AFLSmart only leverages structural information to do higher-order mutations at the data chunk level such as chunk deletion and chunk splicing, leaving lower-level mutations inside those chunks structure unaware. That is, like the original AFL, AFLSmart still randomly chooses input offsets at which bit-byte-word-dword mutations are applied. We argue that it would be better if AFLSmart could select a data chunk first and then choose the offsets/locations inside that chunk. The benefits of doing that are twofold. First, it could help better maintain the validity of the generated input. Second, more fuzzing energy could be assigned to interesting data chunks that have made better progress in terms of code coverage achievement and bug finding.

Second, AFLSmart only supports fuzzing programs that take one specific input model (e.g., it uses PNG input model to fuzz the LibPNG library). It means that it could not achieve the best performance on programs that handle more than one input format, limiting its applicability. For instance, the Bloaty binary profiler[1] accepts ELF, Mach-O, PE/COFF, and WebAssembly file formats. Moreover, the users must tell AFLSmart which input model it should use even though the information might not be documented in the library/program specification, limiting its usability. To address this issue, we design a new working mode called the composite mode in which all/multiple input models can be bundled into a single input model. When users specify such a composite input

[1]https://github.com/google/bloaty

model, AFLSmart will automatically scan through all the included models and use the most suitable one.

We implemented the proposed solutions into an extension of AFLSmart and called it AFLSmart++. We participated in the SBFT'23 Fuzzing Competition in which AFLSmart++ was independently evaluated along with other 11 fuzzers, including new approaches and popular fuzzers such as AFL [1], AFL++ [17], libFuzzer [2] and Honggfuzz [18]. 12 fuzzers competed in two sets of benchmarks: a coverage-based benchmark of 40 target programs and a bug-based benchmark of 15 target programs.

The results show that AFLSmart++ secures the 3rd place in terms of bug finding but it ranks 11th based on its code coverage achievement. One main reason to explain why AFLSmart++ underperformed in the coverage-based benchmark is that only 25% of the target programs in this benchmark consume AFLSmart++'s supported input formats. In the bug-based benchmark, AFLSmart++ seems to work very well on subjects taking specific file formats like AVI but achieve on-par or worse results on others. This looks consistent with the results reported in the original AFLSmart paper and it opens interesting questions for us to explore further as future work.

The remainder of this paper is structured as follows. In Section II, we give an overview of AFLSmart. In Section III, we describe the design and implementation of AFLSmart++. We discuss fuzzing competition results in Section IV before concluding the paper and sharing future research directions in Section V.

## II. Overview of AFLSmart

Figure 1 depicts the workflow of AFLSmart. Like AFL [1], it takes an instrumented binary of the program under test (PUT) and a seed corpus (a.k.a input queue). In addition to that, it also requires the user to specify an input model/specification describing the structure of the expected input. In each fuzzing round, AFLSmart (i) selects a seed input from the input queue, (ii) extracts the structural information and the validity (i.e., percentage of successfully parsed data) of that input using its File Cracker component, (iii) calculates the fuzzing energy (i.e., number of fuzzs) for that seed, and (iv) mutates the seed with both the original AFL's mutation operators (e.g., bit flippings) and its higher-order ones (e.g., chunk deletion, chunk splicing).

Since the second step (i.e., extracting structural information) is expensive, it is done with a certain probability $p$ that depends on the current time to discover a new path. This is called the deferred parsing/cracking mode.

AFLSmart leverages the collected structural information and seed validity mainly to support the third and the fourth steps. In the third step, it assigns more fuzzing energy to inputs that are more valid with respect to the given input model.

In the fourth step (i.e., mutating the input), mutations are done in a so-called *stacking mode*: several structural (high level) and bit/byte-level (low level) mutation operators are applied one after each other.

In this work, we focus on making the following changes:

- Updating the existing low-level mutation operators to make them structure aware.
- Supporting a new working mode called composite mode in which AFLSmart++ can take composite input models that bundle several or even all available input models. With this mode, the user could use one composite model to fuzz test many programs without worrying about a specific input format. This also includes changes to automatically disable the smart (i.e., structure-aware) fuzzing mode if none of the included models is applicable. This newly introduced mode allows AFLSmart++ to participate in the SBFT'23 Fuzzing Competition where many subject programs were kept secret and hence we did not know their expected input formats.

## III. AFLSmart++: Design and Implementation

### A. Structure-aware Low-level Mutation Operators

From a high-level point of view, this feature allows AFLSmart++ to control the fuzzing energy of different data chunks—more progressive or rarely fuzzed data chunks should be selected. This is motivated by our study of different file formats and their processing libraries. We found that some chunk types control more branches than others.

Once a data chunk is selected, low-level mutation operators—that work at bit-byte-word-dword level—can be applied inside the chunk boundary and hence reducing the chance of breaking the input structure.

To implement this feature, we added a hashset of chunk types to each parsed/cracked seed input and a global hashset of all chunk types extracted from the seed corpus. Each chunk type is associated with properties capturing the runtime information such as their selected times, number of fuzzs, and number of new paths found. Based on these dynamically collected statistics, we calculate the scores for each chunk type—both at the seed level and the corpus level—and use that to decide which chunk type should be selected. In the current implementation of AFLSmart++, its default chunk type selection algorithm is called FAVOR: it prefers rare chunk types or chunk types introducing more paths (i.e., progressive chunks).

AFLSmart++ also supports two other selection algorithms called RANDOM and ROUND_ROBIN. In the RANDOM mode, chunk types are randomly selected. In the ROUND_ROBIN mode, chunk types are given turns, one after the other in a repeating sequence. As a result, all chunk types have (almost) the same chance to get selected.

### B. Composite Input Models

Listing 1 shows an example of a composite input model in which many input models can be combined (e.g., models of ELF, PNG, WAV, PDF, and AVI file formats). When such a composite model is given to AFLSmart++, its File Cracker component will try to select at most one input model that matches the given seed input. If there is no matching input model, the seed is considered invalid (with a validity of 0%). The user can still use individual input models—like what

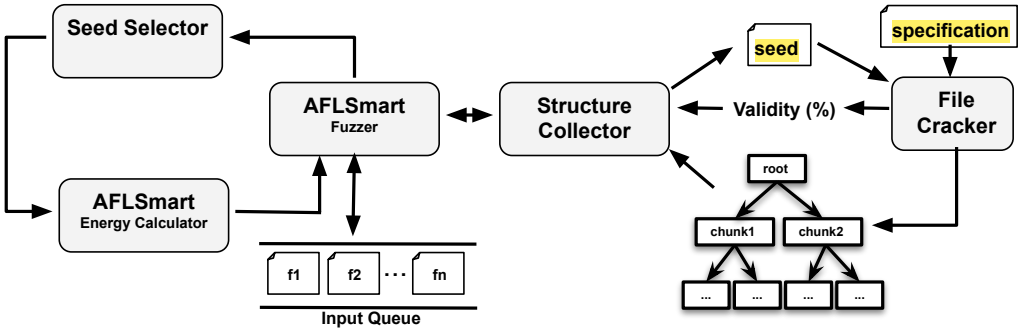


Fig. 1. The main workflow of AFLSmart (the figure is copied from [14]). AFLSmart++ implements structure-aware bit-byte-word-dword mutations and supports composite input models/specifications to improve its usability and applicability.

have been supported by AFLSmart—to fuzz test their targets. For instance, one can fuzz test the LibPNG library by either using the PNG input model (as specified in png.xml) or using this sample composite model. However, using a composite model could allow them to test programs that take several file formats (e.g., Bloaty) or programs with no public input format information.

```
...
<Include ns="elf" src="elf.xml" />
<Include ns="png" src="png.xml" />
<Include ns="wav" src="wav.xml" />
<Include ns="pdf" src="pdf.xml" />
<Include ns="avi" src="avi.xml" />
<Include ns="mp3" src="mp3.xml" />
<Include ns="mp4" src="mp4.xml" />
...

<DataModel name="Composite">
<Choice name="AllFormats" maxOccurs="1">
    <Block name="Elf" ref="elf:ELF"/>
    <Block name="Png" ref="png:PNG"/>
    <Block name="Wav" ref="wav:WAV"/>
    <Block name="Pdf" ref="pdf:PDF"/>
    <Block name="Avi" ref="avi:AVI"/>
    <Block name="Mp3" ref="mp3:MP3"/>
    <Block name="Mp4" ref="mp4:MP4"/>
    ...
</Choice>
</DataModel>
...
```

Listing 1. A sample composite input model

This composite mode introduces a technical issue when the program under test does not take any input format included in the model. The input parsing/cracking process is slow and it could significantly impact the efficiency of the fuzzer if (almost) no input in the corpus adheres to specified formats. To address this issue, we design a simple yet effective algorithm to automatically detect the situation and disable the smart fuzzing mode, letting it run like normal AFL. Basically, it collects the validity information of a configurable number of parsed inputs and if none of them passes a validity threshold, it indicates that the smart mode is likely not useful.

## IV. Evaluation on FuzzBench

Along with other 11 fuzzers, AFLSmart++ was evaluated on two sets of benchmarks on the Google FuzzBench platform: a coverage-based benchmark of 40 target programs and a bug-based benchmark of 15 target programs. These program were selected from the OSS-Fuzz project [3] and the Google Fuzzer Test Suite [19].

AFLSmart++ was run in its newly introduced composite mode and the FAVOR chunk selection algorithm (i.e., rare or progressive chunk types are preferable). The in-used composite input model supports 17 input formats: EFL, MACHO, PNG, JPEG2000, WAV, PDF, AVI, MP3, MP4, GIF, MIDI, OGG, WEBP, XTF, ZIP, PCAP, ICC. These cover media files, binary files, color profiles, compression formats, and font types.

In terms of bug finding, AFLSmart++ secures the 3rd place. However, it ranks 11th in the code coverage benchmark.

There are several reasons that could explain why AFLSmart++ performed pretty well in the bug-based benchmark and underperformed in the coverage-based benchmark. First, in the bug-based benchmark 47% (7/15) of target programs take input formats supported by the chosen input model. However, the figure for the coverage-based benchmark is much lower, just 25% (10/40). Second, AFLSmart++ is still based on the original AFL which does not support recent advanced features like Redqueen [20] and cmplog modes in AFL++ and these features seem to work very well in the coverage-based benchmark.

We also observed that in the bug-based benchmark, AFLSmart++ performed much better in some specific file formats like AVI. It is consistent with the results reported in the original AFLSmart paper. However, it poses an interesting question that is worth further study. Specifically, we would want to see if there are unique properties of the formats making them more suitable for approaches like AFLSmart/AFLSmart++.

## V. Conclusion and Future Work

In this paper, we present AFLSmart++ which is an extension of the structure-aware greybox fuzzer AFLSmart. AFLSmart++ improves AFLSmart in two main aspects. First, it makes low-level mutation operators structure aware. Second, it introduces the so-called composite mode that allows users to fuzz test more types of programs with less effort and a simpler configuration. Experimental results on large-scale benchmarks show that AFLSmart++ is effective in bug finding even though it did not perform well in terms of code coverage achievement.

In our future work, we plan to explore several design options. First and foremost, we will port AFLSmart++ to AFL++ to leverage its advanced and orthogonal features. Second, we will implement and evaluate more systematic data chunk selection algorithms. Specifically, we plan to experiment with multi-armed bandit algorithms [21]. Third, we would consider replacing or simplifying the Peach-based File Cracker component, which is the large source of run-time overhead. Last but not the least, we will add more input models for other input formats.

## VI. Acknowledgement

We thank all the organizers of the SBFT'23 Workshop for their hard work to make the fuzzing competition happen. We also thank Marc Van Hauser Heuse, who is a core developer of AFL++, for his advice on porting AFLSmart++ to AFL++. This work was partially supported by the Amazon AWS Cloud Credit for Research program.

## References

[1] "American fuzzy lop." [Online]. Available: https://lcamtuf.coredump.cx/afl/

[2] "libfuzzer – a library for coverage-guided fuzz testing." [Online]. Available: https://github.com/llvm-mirror/llvm/blob/master/docs/LibFuzzer.rst

[3] "Oss-fuzz: Continuous fuzzing for open source software." [Online]. Available: https://github.com/google/oss-fuzz

[4] R. Zhong, Y. Chen, H. Hu, H. Zhang, W. Lee, and D. Wu, "Squirrel: Testing database management systems with language validity and coverage feedback," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 955–970.

[5] "Domato: A dom fuzzer." [Online]. Available: https://github.com/googleprojectzero/domato

[6] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: a greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 460–465.

[7] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury, "Stateful greybox fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3255–3272.

[8] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz, "Nyx-net: network fuzzing with incremental snapshots," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 166–180.

[9] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Restler: Stateful rest api fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 748–758.

[10] L. Pan, S. Cohney, T. Murray, and V.-T. Pham, "Detecting excessive data exposures in web server responses with metamorphic fuzzing," *arXiv preprint arXiv:2301.09258*, 2023.

[11] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2019.

[12] M. Böhme, C. Cadar, and A. Roychoudhury, "Fuzzing: Challenges and reflections." *IEEE Softw.*, vol. 38, no. 3, pp. 79–86, 2021.

[13] "libprotobuf-mutator." [Online]. Available: https://github.com/google/libprotobuf-mutator/

[14] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1980–1997, 2019.

[15] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars." in *NDSS*, 2019.

[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] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[18] "Honggfuzz." [Online]. Available: https://github.com/google/honggfuzz

[19] "Fuzzer test suite." [Online]. Available: https://github.com/google/fuzzer-test-suite

[20] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence." in *NDSS*, vol. 19, 2019, pp. 1–15.

[21] J. Vermorel and M. Mohri, "Multi-armed bandit algorithms and empirical evaluation," in *Machine Learning: ECML 2005: 16th European Conference on Machine Learning, Porto, Portugal, October 3-7, 2005. Proceedings 16*. Springer, 2005, pp. 437–448.