Improving Configurability of Unit-level Continuous Fuzzing: An Industrial Case Study with SAP HANA

Hanyoung Yoo*, Jingun Hong†, Lucas Bader‡, Dong Won Hwang†, Shin Hong*
hanyoung0411@handong.edu, {jingun.hong, lucas.barder, dong.won.hwang}@sap.com, hongshin@handong.edu
*Handong Global University, †SAP Labs Korea, ‡SAP SE

***Abstract*—This paper presents industrial experiences on enhancing the configurability of a fuzzing framework for effective continuous fuzzing of the SAP HANA components. We propose five new mutation scheduling strategies for effective uses of grammar-aware mutators in the unit-level fuzzing framework, and three new seed corpus selection strategies to configure a fuzzing campaign to check on changed code in priority. The empirical results show that the proposed extension gives users chances to improve fuzzing effectiveness and efficiency by configuring the framework specifically for each target component.**

***Index Terms*—greybox fuzzing, unit-level fuzzing, regression fuzzing, continuous fuzzing**

I. Introduction

Continuous testing with greybox fuzzing techniques [1], [2], or *continuous fuzzing*, is rapidly emerging as a practical method for open-source projects to guard against security vulnerabilities and software faults [3]. Greybox fuzzing has been extended from system-level testing to unit-level testing [4] for testing an API or internal function for verifying specific target components intensively. Continuous unit-level fuzzing is desirable for developers to identify new failures in a component early to quick start debugging before the buggy component is merged with the others in the same project.

A fuzzing framework for unit-level testing should be much more configurable than conventional fuzzing frameworks for system-level testing because the best fuzzing configuration for a component under test may be largely different other components in the same project, depending on the structure, the functionality, and the usage. However, since the existing unit-level fuzzing framework (e.g., libFuzzer [5]) provide a similar degree of fuzzing configurability, it is challenging to realize effective continuous unit-level fuzzing with the existing framework for their limited configurability.

This paper presents our experiences on constructing a fuzzing infrastructure for SAP HANA to provide effective continuous fuzzing on unit-level fuzzing drivers. HANA comprises a large number of components that constitute a fully-fledged, high-performance database management system supporting various interfaces and runtime environments. To supplement the manually-written unit tests, SAP has integrated libFuzzer [5] to the HANA test infrastructure and conducted a research project to investiage the best practices, and also to find limitations and chances for improvement within the fuzzing framework. Through this project, we observed the following two obstacles of fuzzing effectiveness:

- Despite the users' effort to configure mutation by constructing customized grammar-aware mutation operators, these may not improve the overall performance depending on how these are scheduled.
- Even with a local code change, the fuzzer starts over the whole path exploration from the existing seed corpus, such that it may fail to extensively test the changed behaviors by spending too much effort on irrelevant paths.

We resolve these problems by extending the test infrastructure with libFuzzer to provide more configuration options to fit a fuzzing driver to a target component for continuous fuzzing application along project evolution. The extension is twofold:

- To effectively utilize customized mutation operators, we present new mutation scheduling strategies to offer more options to apply customized mutation operators in a fuzzing campaign (Section III).
- To give the users to options for seed selection, we propose new seed selection strategies that effectively re-use the tests generated with an earlier version (Section IV).

We found that that the proposed technique provides better chances to the engineers to configure suitable options for each target component specifically. The experiment results show that, by using the proposed techniques, the fuzzer can effectively utilize customized grammar-aware mutators to substantially improve code coverage (Section III-B) and also effectively reduces the time to find a failure by leveraging the input corpus generated with a previous version (Section IV-B).

II. Project Overview

A. Background: Fuzzing Infrastructure for SAP HANA

SAP HANA is a high-performance in-memory database system used by numerous IT services worldwide. To ensure high product quality, the developers are putting extensive efforts into constructing regression unit tests along their feature development, and the HANA Quality Engineering team builds and operates a test infrastructure that manages to run a large number of regression tests continuously and systematically.

The HANA Quality Engineering team has integrated libFuzzer [5] into the test infrastructure, so that developers can construct a fuzzing driver as part of the test suite of a component. Once a fuzzing driver is registered, it is designed to periodically conduct fuzzing and report newly found failures automatically to the developers via issue trackers. To further help developers analyze found issues, work is ongoing for performing additional steps within a fuzzing campaign:

- Determine whether a failure is new or redundant, based on the error information (e.g. stack trace) The fuzzing

2643-1572/21/$31.00 ©2021 IEEE
DOI 10.1109/ASE51524.2021.00111

integration can re-use an existing component in the infrastructure that collects error information of all observed failures from regression tests and checks if a given failure is redundant to an existing case. If a failure is found to be redundant, it will not be registered to the issue tracker.
- Since the fuzzer produces an input as fuzzing driver arguments, a fuzzer-generated input is often not human readable or not directly usable as part of regression testing. To resolve this issue, a concrete unit test (i.e. as `gtest`) can then be instantiated by assigning the parameters of the unit test case template with a fuzzer-generated input. This concrete unit test could be automatically pushed to the repository as part of the regular test suite.

B. Unit-level Fuzzing Driver Construction

This project was started to investigate the best practices for writing unit-level fuzzing to supplement manually-written regression unit test cases of the HANA components. To deliver compelling lessons and realistic assessments, we had chosen two representative HANA components as *flagship targets* to apply best unit-level fuzzing practices and to examine the capability of the current fuzzing framework. One target component, hereafter called as Parser, is one of many parsers in HANA, which converts a user-given text to the corresponding abstract syntax tree. The other target component, hereafter called as Deserializer, is a component that reconstructs a complex object from its encoded data. Both target components have developed many years, thus they are well stabilized. As the reliability of both components is crucial, they already have comprehensive regression test suites.

We constructed fuzzing drivers based on the manually-written unit test cases in the regression test suites. First, we grouped the existing unit test cases by test target function. For each target function, we analyzed all concrete inputs to identify different test input aspects, and then assigned each aspect to certain input buffer offsets. Based on the cases of concrete inputs, we merged the execution paths to the target function invocation into a single parameterized unit test that has the common test setup in the main path and the test setup variations in the subpaths. After the target function invocation, we add operations to use the results of the target function in different ways. For example of Parser, the driver executes visitors to traverse the generated abstract syntax tree. Such operations are useful as they diversify path conditions such that more coverage features will be given to greybox fuzzers to exploit. Last, we constructed project-specific mutators and incorporated them in fuzzing to inject syntactically valid mutations for enhancing fuzzing performance.

In our case studies, we had constructed total 10 fuzzing drivers for Parser and Deserializers. Using these fuzzing drivers, we could detect total unknown 9 bugs which were confirmed by the feature developers. Considering the stability and the quality of the target components, these bug findings are considered as promising results and the field developers recognize the effectiveness of unit-level fuzzing.

Through these case studies, we could identify the limitation of the existing fuzzing framework and developed the techniques to use the constructed unit-level fuzzing drivers in continuous testing in effective and efficient ways.

III. Configurability for Mutation Scheduling

A. Motivation and Proposed Technique

Grammar-aware mutation is crucial [6] for exploring various behaviors of a component that receives a structured text as an input argument. Generic default mutation operators are ineffective at exercising grammar-sensitive aspects of the target component since a series of random local text edits hardly induce semantically meaningful and syntactically valid changes on a seed input. To resolve this limitation, test engineers design a grammar-aware mutator which is a text transformer that mutates an identified syntactic aspect of an input, rather than just a word, while keeping the structural validity of the input text. Existing fuzzing frameworks including libFuzzer provide an interface for the engineer to construct grammar-aware mutators accustomed to the input grammar of a specific target component [5], [7].

For effective input mutation for the two flagship components, we constructed 28 customized mutators for Parser and 20 customized mutators for Deserializer, respectively. These mutation operators are systematically constructed based on our investigation on the target input grammars. Each mutator is carefully designed to have unique behaviors (i.e., not redundnat to other mutation operators), and also efficiently implemented not to incur much runtime overhead.

Despite these efforts, we initially had frustrating results. When customized mutators are not properly used together with the generic mutators, the fuzzer failed to improve coverage achievement, and sometime the coverage stucked at a lower point than the fuzzing without using the customized mutators. These results indicates that the fuzzing framework should provide a mechanisms to configure mutation scheduling to leverage both customized mutators and the generic mutators to substantially improve the overall performance of fuzzing.

To enable the users to configure mutation scheduling strategies specifically for a target component, we devised the following five schemes to offer different combinations of generic mutators and grammar-aware mutators:

- *generic-only*: use only the generic mutation operators. All 12 built-in mutators of libFuzzer, including the naïve crossover operator, are employed.
- *grammar-only*: use only grammar-aware mutators.
- *interleaved*: switch between *generic-only* and *grammar-only* every one hour (similar to [8]).
- *intermixed*: at each mutation chance, select either a generic or grammar-aware mutator with predefined probabilities. Based on the pilot study results, we set the probability of choosing a generic mutator as 75% and the other as 25%. that of choosing a grammar-aware mutator as 25%.
- *combined*: switch between two versions of the intermixed fuzzing schemes periodically: one version selects a generic mutator with 75% probability, and the other selects a grammar-aware mutator with 75% probability. This scheme is a hybrid of the interleaved and intermixed schemes.

TABLE I
THE AVERAGE NUMBER OF BRANCHES COVERED AFTER 5 HOURS

	generic -only	grammar -only	inter- leaved	inter- mixed	combined
Parser	10574	10096	10988	11118	10987
Deserializer	5759	4175	6168	5713	6111

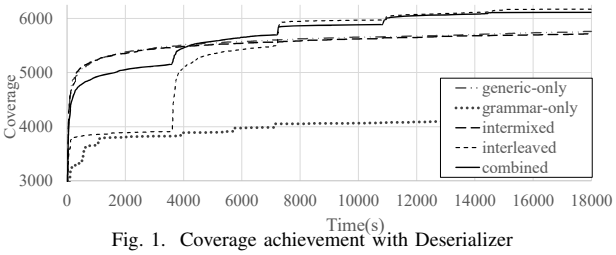

Fig. 1. Coverage achievement with Deserializer

B. Evaluation

We explored the proposed schemes for using generic and grammar-aware mutators to empirically assesses their impacts on test coverage achievements. For each mutation scheduling strategy, we conducted experiments with one fuzzing driver of Parser and one fuzzing driver of Deserializer and measuried the branch coverage along testing time.

We ran a test execution for each fuzzing scheme and target program using 8-cores for 5 hours. All test executions were conducted on a machine running Linux SUSE Server 12 SP3 with Intel Xeon 4116 CPU @ 2.10GHz with 48-cores and 395 GB RAM. The time bound for a fuzzing campaign is set to 5 hours, because, in a pilot study with 12 hours of fuzzing, we found that the branch coverage does not change significantly after 5 hours. We run the fuzzing campaign with the same setting for five times to obtain their average results to limit the risk of sampling bias. To assess the test coverage, we used SantizerCoverage to count the number of branches that are covered at least once in a fuzzing campaign.

Table I shows that the average number of the branches covered after 5 hours of fuzzing. For Parser, the three hybrid schemes (i.e., *interleaved*, *intermixed*, *combined*) achieved higher coverage than *generic-only* and *grammar-only*, and, among the five, *intermixed* shows the best performance. For Deserializer, the result indicates that two interleaving schemes, *interleaved* and *combined*, achieved higher branch coverage than the other three. Unlike the Parser case, *intermixed* achieved less coverage than *generic-only*. Figure 1 shows the coverage increases with the five mutation scheduling schemes. The plots of *interleaved* and *combined* show that the coverage rapidly increased right after each switching point. We conjectured that this is because the branches that the grammar-aware mutators effectively explore and the branches that the generic mutators effectively explore are different.

In overall, the experiment results imply that the grammar-aware mutators should be used together with the generic mutators as they can supplement each other. We also found that the performance of a single mutation scheduling scheme depends on the target component, thus the fuzzer framework should provide the users to experiment different schemes and to configure the fuzzing configuration specifically to a given unit-level fuzzing driver. Finally, among the five proposed schemes, *combined* performs best to provide high coverage achievement reliabily as it conducts intermixing and interleaving of two sorts of input mutation at the same time.

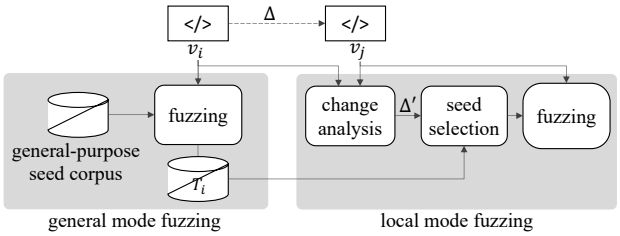

Fig. 2. Seed Corpus Selection Workflow

IV. CONFIGURABILITY FOR SEED CORPUS SELECTION

A. Motivation and Proposed Technique

An initial seed corpus is a key factor that determines the search scope of a fuzzing campaign [9]. A fuzzing driver is conventionally accompanied with a general-purpose seed corpus which aims to target a wide range of the target program behaviors. In contrast to this convention, if the user wants to run a fuzzer to quickly check on a code change in a specific component (e.g., for a pre-commit check), it would be better for the fuzzing to use a seed corpus relevant to the changed code, rather than a geneарl-purpose seed corpus.

To enable users to configure a fuzzing campaign specifically targeted for a code change, we propose a new technique that (1) stores the resulting corpus of an earlier fuzzing campaign together with their coverage information and then (2) produces a change-specific seed corpus by collecting a subset of the stored test inputs that cover the changed code region. The technique provides three selection strategies to control the seed corpus size. We conjecture that, if a code change is not significant, the stored test inputs will position execution paths in the fuzzing queue to effectively explore the changed behaviors of the target program.

Figure 2 describes the overall workflow of the proposed technique. There are two fuzzing modes: general mode and local mode. The general mode uses the general-purpose seed corpus as the same as the conventional fuzzing. A fuzzing campaign in the general mode typically runs for long time to explore as diverse behaviors as possible. Once a general mode fuzzing on an earlier version (v_i) finishes, the generated corpus is passed to the corpus database (T_i). Note that the resulted corpus does not contain all generated inputs, but only a subset of generated inputs each of which covers at least one coverage item uniquely (i.e., interesting input [10]). For each test input, the functions covered in the execution are identified, and then, for each covered function, our framework stores the set of the test inputs covering the function as the *per-function seed corpus* in the corpus database.

The fuzzer starts a campaign in the local mode when a local code change (Δ) is made on the target component and the user wants to quickly check whether the revised version (v_j) has a new failure or not. Under the local mode, the change analysis module identifies all functions ($\Delta^{'}$) whose code was modified from the earlier version on which the latest general

TABLE II
STUDIED CODE CHANGE CASES

study name	component	failure type	# changed lines	# changed functions	# stored tests
C1	Parser	Out-of-mem.	+28, -8	2	7058
C2	Parser	Segfault	+20, -5	2	7058
C3	Deserializer	Segfault	+9, -0	1	3009
C4	Deserializer	Abortion	+10, -0	2	3009

mode fuzzing was conducted (v_i). This module uses `diff` to find all updated source code files and determines all functions where at least one of the code lines was modified, deleted or added. Given information on the changed functions and per-function corpus, the corpus selection module produces a seed corpus targeted for testing the changed functions. The corpus selection offers the following three strategies:

- *select-all*: select all inputs in the per-function corpuses of the changed functions
- *select-coverage*: select a small number of test inputs that cover all functions covered by the per-function corpuses of the changed functions. Initially, random five test inputs are selected from the union of the per-function corpuses of the changed functions. And then, until the selection covers all functions that the per-function corpuses cover, iteratively includes one random test input among the ones that cover a missing function to the selection.
- *select-N*: randomly select N inputs from the per-function corpuses of the changed functions.

We devised three different strategies for the users to control the seed corpus size depending on a fuzzing situtation. After a local mode fuzzing, the proposed technique does not update the corpus database, since the generated test input would be biased toward specific behaviors or code regions at the moment, thus they may not be effective for targeting other parts of the target components.

B. Evaluation

We conducted the experiments to assess to what extent the proposed techniques improve fault detection ability and fault detection rate of our fuzzing framework for checking on code changes. For this evaluation, we designed four studies where we first made a code change by injecting a fault to a target project, and then ran the local mode fuzzing to detect a failure. The code changes used for the study are imitating the historical cases of the actual bugs that were recently fixed within the scope of the unit fuzzing drivers in Parser and Deserializer.

Table II presents the information of the four fault-injecting code changes. Unfortunately, we could not use a previous commit that actually induces a bug since we found that the bug-inducing commit is apart from the latest version and not compatible with the current fuzzer framework intergration. To limit validity threats, each fault injection is carefully constructed in collaboration with the field developers of the target component to reflect the situation of the actual bug.

For each study, we experimented the proposed technique with *select-all*, *select-coverage*, and *select-10* (i.e., select 10 random test inputs that covered the changed functions), independently. In addition, as a baseline technique, we additionally experimented the fuzzer with the general-purpose seed corpus constructed for the corresponding unit fuzzing drivers. We used one fuzzing driver for Parser and another for Deserializer that we constructed for the project. We configured the fuzzer to use all generic default mutation operators. The customized mutation operators were not used in these experiments to simplify the experiment space. Using the same machine as described at Section III-B, we conducted five independent fuzzing campaigns for each strategy using 7 cores with a 5 hour time limit. We measured the probability for the proposed technique with a strategy to find a failure in 5 hours, and the average time to first failure detection.

TABLE III
EVALUATION RESULTS OF SEED SELECTION TECHNIQUES

		general	select-all	select-cov.	select-10
C1	detect. ratio	40%	100%	100%	100%
	average time	16687	1059	1877	3071
	corpus size	61	600	37	10
C2	detect. ratio	40%	80%	40%	40%
	average time	13356	8997	8013	7596
	corpus size	61	600	38	10
C3	detect. ratio	100%	100%	100%	100%
	average time	26	0	0	0
	corpus size	20	117	15	10
C4	detect. ratio	100%	100%	100%	100%
	average time	2973	0	222	990
	corpus size	20	946	49	10

Table III shows the experiment results. Each row with "detect. ratio" shows the ratio of the fuzzing campaigns that detects the failure to the total number of fuzzing campaigns (i.e., 5). Each row with "average time" gives the average time (in seconds) to first failure detection. Each row with "corpus size" shows the average size of the used seed corpus.

The results show that the proposed corpus selection techniques imporves both fault detection abilities and fault detection rates in all four case studies. In C1, the proposed technique always detected the failure, whereas the conventional fuzzing with the general-purpose seed corpus detected the failure only for 40%. The average failure detection time of the proposed technique is only 6.35% (*select-all*) to 18.40% (*select-10*) of the baseline technique. In C2, the failure detection ratio of the proposed technique is greater than (*select-all*) or equal to (*select-coverage*, *select-10*) the baseline technique, and the avarage time to the first failure detection is only 56.9% to 67.4% of the baseline technique result. In C3 and C4, all techniques detected the failures in all cases. In C3, the proposed technique with all strategies immediately found the failure as the stored corpus contains failure-revealing test inputs. In C4, the proposed technique tool only 0.0% (*select-all*) to 33.3% (*select-10*) of the testing time of the baseline.

These results imply that use of the seed corpus specific to code changes improves fuzzing performance, and, among the three proposed strategies, *select-all* showed best results as it always achieved the highest fault detection ratio, and the highest fault detection rate for three cases (except C2 where *select-coverage* showed the best fault detection ratio).

ACKNOWLEDGEMENT

This work is supported by SAP Labs Korea Inc. and the National Research Foundation grants funded by the Korea government (NRF-2020R1C1C1013512 and NRF-2021R1A5A1021944).

REFERENCES

[1] V. J. M. 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 (TSE)*, vol. early access, pp. 1–1, 2019.

[2] P. Godefroid, "Fuzzing: Hack, art, and science," *Communications of the ACM (CACM)*, vol. 63, no. 2, p. 70–76, Jan. 2020.

[3] K. Serebryany, "OSS-Fuzz - Google's continuous fuzzing service for open source software." Vancouver, BC: USENIX Association, Aug. 2017.

[4] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "Fudge: Fuzz driver generation at scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 975–985. [Online]. Available: https://doi.org/10.1145/3338906.3340456

[5] LLVM Compiler Infrastructure, *libFuzzer – a library for coverage-guided fuzz testing*, https://llvm.org/docs/LibFuzzer.html.

[6] 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*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 329–340. [Online]. Available: https://doi.org/10.1145/3293882.3330576

[7] 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. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi

[8] Z. Xu, Y. Kim, M. Kim, and G. Rothermel, "A hybrid directed test suite augmentation technique," in *2011 IEEE 22nd International Symposium on Software Reliability Engineering*, 2011, pp. 150–159.

[9] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. USA: USENIX Association, 2014, p. 861–875.

[10] "American fuzzy lop - a security-oriented fuzzer." [Online]. Available: lcamtuf.coredump.cx/afl/