Refined Grey-Box Fuzzing with SIVO

Ivica Nikolić[1(✉)], Radu Mantu[2], Shiqi Shen[1], and Prateek Saxena[1]

[1] School of Computing, NUS, Singapore, Singapore
inikolic@nus.edu.sg
[2] University Politehnica of Bucharest, Bucharest, Romania

Abstract. We design and implement from scratch a new fuzzer called SIVO that refines multiple stages of grey-box fuzzing. First, SIVO refines data-flow fuzzing in two ways: (a) it provides a new taint inference engine that requires only logarithmic number of tests in the input size to infer dependency of many program branches on the input bytes, and (b) it employs a novel method for inverting branches by solving a systems of inequalities efficiently. Second, our fuzzer refines accurate tracking and detection of code coverage with simple and easily implementable methods. Finally, SIVO refines selection of parameters and strategies by parameterizing all stages of fuzzing and then dynamically selecting optimal values during fuzzing. Thus the fuzzer can easily adapt to a target program and rapidly increase coverage. We compare our fuzzer to 11 other state-of-the-art grey-box fuzzers on 27 popular benchmarks. Our evaluation shows that SIVO scores the highest both in terms of code coverage and in terms of number of found vulnerabilities.

1 Introduction

Fuzzing is the automatic generation of test inputs for programs with the goal of finding bugs. With increasing investment of computational resources for fuzzing, tens of thousands of bugs are found in software each year today. We view fuzzing as the problem of maximizing coverage within a given computational budget. The coverage of all modern fuzzers improves with the computation budget allocated. Therefore, we can characterize the quality of a fuzzer on its *rate of coverage increase*, the average number of new control-flow edges exercised per CPU cycle.

Broadly, there are three types of fuzzers. Black-box fuzzers do not utilize any knowledge of the program internals, and are sometimes referred to as undirected fuzzers. White-box fuzzers perform intensive instrumentation, for example, enabling dynamic symbolic execution to systematically control which program branches to invert in each test. Grey-box fuzzers introduce low-overhead instrumentation into the tested program to guide the search for bug-triggering inputs. These three types of fuzzers can be combined. For instance, recent hybrid fuzzers selectively utilize white-box fuzzers in parallel to stand-alone grey-box fuzzers. Of the three types of fuzzers, grey-box fuzzers have empirically shown promising cost-to-bug ratios, thanks to their low overhead techniques, and have seen a flurry of improved strategies. For example, recent grey-box fuzzers have

© Springer Nature Switzerland AG 2021
L. Bilge et al. (Eds.): DIMVA 2021, LNCS 12756, pp. 106–129, 2021.
https://doi.org/10.1007/978-3-030-80825-9_6

introduced many new strategies to prioritize seed selection, byte mutations, and so on during fuzzing. Each of these strategies works well for certain target programs, while being relatively ineffective on others. There is no dominant strategy that works better than all others on all programs presently.

In this paper, we present the design of a new grey-box fuzzer called Sivo that *generalizes* well across many target programs. Sivo embraces the idea that there is no one-size-fits-all strategy that works universally well for all programs. Central to its design is a "parameterization-and-optimization" engine where many specialized strategies and their optimization parameters can be specified. The engine dynamically selects between the specified strategies and optimizes their parameters on-the-fly for the given target program based on the observed coverage. The idea of treating fuzzing as an optimization problem is not new—in fact, many prior fuzzers employ optimization either implicitly or explicitly, but they do so *partially* [4,22,30,35]. Sivo differs from these works conceptually in that it treats parameterization as a first-class design principle—all of its internal strategies are parameterized. The selection of strategies and determination of all parameter values is done dynamically. We empirically show the power of embracing *complete parameterization* as a design principle in grey-box fuzzers.

Sivo introduces 3 additional novel refinements for grey-box fuzzers. First, Sivo embodies a faster approximate taint inference engine which computes taint (or sensitivity to inputs) for program branches during fuzzing, using number of tests that are only logarithmic in the input size. Such taint information is helpful for directed exploration in the program path space, since inputs influencing certain branches can be prioritized for mutation. Our proposed refinement improves exponentially over a recent procedure to calculate taint (or data-flow dependencies) during fuzzing [12]. Second, Sivo introduces a light-weight form of symbolic interval reasoning which, unlike full-blown symbolic execution, does not invoke any SMT/SAT solvers. Lastly, it eliminates deficiencies in the calculation of edge coverage statistics used by common fuzzers (e.g. AFL [37]), thereby allowing the optimization procedure to be more effective. We show that each of these refinements improves the rate of coverage, both individually and collectively.

We evaluate Sivo on 27 diverse real-world benchmarks comprising several used in recent work on fuzzing and in Google OSS-fuzz [15]. We compare Sivo to 11 other state-of-the-art grey-box fuzzers. We find that Sivo outperforms all fuzzers in terms of coverage on 25 out of the 27 benchmarks we tested. Our fuzzer provides 20% increase in coverage compared to the next best fuzzer, and 180% increase compared to the baseline AFL. Furthermore, Sivo finds most vulnerabilities among all fuzzers in 18 of the benchmarks, and in 11 benchmark programs finds unique vulnerabilities. This provides evidence that Sivo generalizes well across multiple programs according to multiple metrics. We have released our fuzzer publicly and open-source [25].

2 Problem

Fuzzers look for inputs that trigger bugs in target programs. As the distribution of bugs in programs is unknown, fuzzers try to increase the chance of finding

bugs by constructing inputs that lead to maximal program code execution. The objective of fuzzers is thus to construct inputs, called *seeds*, that increase the amount of executed program code, called *code coverage*. The coverage is measured based on the control-flow graph of the executed program, where nodes correspond to basic blocks (sets of program statements) and edges exist between sequential blocks. Some of the nodes are conditional (e.g. correspond to `if` and `switch` statements) and have multiple outgoing edges. Coverage increases when at some conditional node, called a *branch*, the control flow takes a new edge which is not seen in previous tests—this is called *inverting* or *flipping* a branch.

Grey-box fuzzers assess code coverage by instrumenting the programs and profiling coverage data during the execution of the program on the provided inputs. They maintain a pool of seeds that increase coverage. A grey-box fuzzer selects one seed from its pool, applies to it different operations called *mutations* to produce a new seed, and then executes the program on the new seed. Those new seeds that lead to previously unseen coverage are added to the pool. To specify a grey-box fuzzer one needs to define its seed selection, the types of mutations it uses, and the type of coverage it relies on. All these fuzzing components, we call *stages* or *subroutines* of grey boxes. We consider a few research questions related to different stages of fuzzing.

RQ1: Impact of Complete Parameterization? Fuzzers optimize for coverage. There is no single fuzzing strategy that is expected to work well across all programs. So, the use of multiple strategies and optimization seems natural. Existing fuzzers do use dynamic strategy selection and optimize the parameter value selection. For example, MOpt [22], AFLFast [4], and EcoFuzz [35] use optimization techniques for input seed selection and mutations. But, often such parameterization comes with internal constants, which have been hand-tuned on certain programs, and it is almost never applied universally in prior fuzzers. The first question we ask is what would be the result of complete parameterization, i.e., if we encode all subroutines and their built-in constants as optimization parameters.

The problem of increasing coverage is equivalent to the problem of inverting more branches. In the initial stage of fuzzing, when the number of not yet inverted branches is high, AFL mutation strategies (such as mutation of randomly chosen bytes) are successful and often help to invert branches in bulk. However, easily invertible branches soon become exhausted, and different strategies are required to keep the branch inversion going. One way is to resort to targeted inversion. In targeted inversion, the fuzzer chooses a branch and mutates input bytes that influence it. The following two questions are about refining target inversion in grey-box fuzzing.

RQ2: Efficient Taint Inference? Several fuzzers have shown that taint information, which identifies input bytes that influence a given variable, is useful to targeted branch inversion [2,6,8,12,26,34]. If we want to flip a particular branch, the input bytes on which the branch condition variables depend should be mutated while keeping the other bytes unchanged. The main challenge, however, is to efficiently calculate the taint information. Classical methods for dynamic taint-tracking incur significant instrumentation overheads whereas static meth-

ods have false negatives, i.e. they miss dependencies due to imprecision. The state-of-the-art fuzzers aim for light-weight techniques for dynamically inferring taint during fuzzing itself. Prior works have proposed methods which require number of tests linear in n, the size of the seed input [12]. This is extremely inefficient for programs with large inputs. This leads to our second question: Can we compute useful taint information but with exponentially fewer tests?

RQ3: Efficient Constraint-Based Reasoning? Taint only captures whether a change in certain values of an input byte may lead to a change in the value of a variable. If we are willing to compute more expressive symbolic constraints, determining the specific input values which cause a program branch to flip is possible. The challenge is that computing and solving expressive constraints, for instance first-order SAT/SMT symbolic formulae, is computationally expensive. In this work, we ask: Which symbolic constraints can be cheap to infer and solve during grey-box fuzzing?

RQ4: Precise Coverage Measurement? Grey-box fuzzers use coverage information as feedback to guide input generation. AFL, and almost all other fuzzers building on it, use control-flow edge counts as a common metric. Since there can be many control-flow edges in the program, space-efficient data structures for storing runtime coverage data are important. Recent works have pointed out AFL's hash-based coverage map can result in collisions [13], which has an unpredictable impact on the resulting optimization. How do we compute compressed edge counts with high precision using standard compilers for instrumentation?

3 Overview of SIVO

Grey-box fuzzers instrument the target program to gather runtime profiling data, which in turn guides their seed generation strategies. The objective of SIVO is to generate seeds that increase code coverage by using better and more of the profiling data. SIVO addresses the four research questions with four refinements.

Parametrize-Optimize Approach (RQ1). SIVO builds on the idea of complete parameterization of all fuzzing subroutines and strategies, i.e. none of the internal parameters are hard-coded. SIVO selects strategies and parameter values dynamically based on the observed coverage statistics, using a standard optimization algorithm. Such complete parameterization and optimization inherently makes SIVO adaptable to the target program and more general, since specialized strategies that work best for the program are prioritized. To answer RQ1, we empirically show in our evaluation that this design principle individually helps SIVO outperform other evaluated fuzzers across multiple target programs.

Fast Approximate Taint Inference (RQ2). We devise a fast and approximate taint inference engine TaintFAST based on probabilistic group testing [10]. Instead of testing individually for each input byte, TaintFAST tests for carefully chosen groups of bytes and then combines the results of all tests to infer the taint for each individual byte. This helps to reduce the test complexity of taint

inference from $O(n)$ to $O(\log n)$ executions of the program, where n is the number of input bytes. Thus the fuzzer can infer useful taint dependency even for very large inputs using TaintFAST.

Symbolic Interval Constraints (RQ3). We propose inferring symbolic interval constraints that capture the relationship between inputs and variables used in branch conditions only. Instead of deductively analyzing the semantics of executed instructions, we take an optimistic approach and infer these constraints from the observed values of the inputs and branch conditional variables. The value-based inference is computationally cheap and tailored for a common case where values of the variables are direct copies of the inputs and when branches have comparison operations ($=, \neq, <, \leq, >, \geq$). We show that such a constraint system can be solved efficiently as well without the use of SAT/SMT solvers.

Compressed and Precise Edge Count Recording (RQ4). We tackle both the collision problem and the compressed edge count problem in tracking coverage efficiently during grey-box fuzzing. For the former, we show a simple strategy based on using multiple basic block labels (rather than only one as in AFL) and reduce or entirely eliminate the collisions. For the later, to improve the prospect of storing important edge counts we propose temporary coverage flushing (i.e. resetting the coverage to zero). Although this may appear to be a minor refinement in grey-box fuzzing, we find that it has a noticeable impact experimentally.

4 Design

We present the details of our four refinements in Sects. 4.1–4.4 and then show the complete design of Sivo in Sect. 4.5.

4.1 The Parametrize-Optimize Paradigm

The Sivo grey-box fuzzer aims to increase the code coverage in the fuzzed programs. Two points are central to this goal. First, fuzzed programs come in different flavors, hence the fuzzer should be flexible and adaptive. We tackle the first point with *parametrization*, i.e. by expanding the choice of available fuzzer subroutines. Second, a fuzzer has a few stages (i.e., selection of seeds, choice of mutations and their parameters, etc.), and each one of them can be optimized. To address this point, we apply a complete *optimization* of all available parameters.

Parametrization. The more fuzzing subroutines are available, the higher the chance that some of them may be optimal for fuzzing the targeted program. Thus it is useful to expand the set of available fuzzing subroutines. To do so, we:

– *Add many fuzzing subroutines.* For instance, in addition to the AFL-style *vanilla* mutations that do not require any dependency information (e.g. mutate random bytes), we implement *data-flow* strategies that utilize input dependency of program branches (e.g., mutation of dependent bytes). Besides adding new mutations, we also add more seed prioritization methods that determine how to sample a seed from the pool.

– *Introduce variations in each subroutines.* Often this can be done by varying internal hard-coded parameters in subroutines. For instance, in the mutation of random bytes, instead of changing a single byte, Sivo can change 1, 2, 4, 8, 16, 32, or 64 bytes at once. The exact number of bytes is considered an input parameter; it can take one of the above 7 values (and the choice of value potentially can be optimized). Not all variations in subroutines are effected with changing integer parameters. For instance, the seed selection criterion is based on speed, number of repetitions, length of seed, and so on. These variations are enumerated and serve as an input parameter to the seed criterion. All such parameters to subroutines are optimized per program.

As a result, across the whole fuzzer, there are 17 different fuzzing subroutines with 68 variations. In comparison, the baseline AFL has around 15 different subroutines with around 45 variations[1].

Optimization. The parametrization increases the chance that potentially optimal subroutines are chosen for each program. The next step is to select which subroutines are turned on for a given program. It is critical to understand that we are not dealing with a single optimization problem. Fuzzing is a continuous process, composed of iterations that select a seed and a mutation, apply the mutation to the seed, and check on coverage increase. Thus, in each iteration we need to optimize the selection of fuzzing subroutines several times—for example, the used seed criterion and class, the mutation strategy, (potentially a number of) mutations sub-strategies, the inputs to the mutation strategy, and so on. For this purpose, we use multi armed bandits (MAB), a simple reinforcement learning algorithm. Given a set of choices, each choice providing a certain reward when selected, MAB helps to select the choices such that their accumulative rewards are maximized. The rewards are unknown and stochastic, and the selection process is continuous. Note, after MAB selects a choice, it needs to receive as a feedback the obtained reward to update its choice selection strategy.

Reducing the selection of fuzzing subroutines to MAB problem is straightforward. First, note that we consider each selection as an independent MAB problem, for instance, the optimal number of random bytes to mutate is one MAB problem. Our objective is to maximize the coverage, hence it is natural to use the additional coverage acquired from executing the choice as the MAB *reward*. However, this metric alone may not be accurate because some choices incur higher computational costs. Therefore, we use the *additional coverage per time unit* as the reward. In the conventional MAB, the distributions of rewards are stationary with some unknown mean. In our case, as the fuzzer progresses, it requires more computational effort to reach the remaining unexplored code and increase coverage. In other words, the rewards for the selection choices monotonically decrease over time. Therefore, we model our problem as MAB with *non-stationary* rewards and use discounting to solve it [19]. For more details on application of MAB in Sivo, we refer the reader to Algorithm 1 and Sect. 4.5.

[1] Despite having comparable numbers, Sivo and AFL use mostly different mutations and thus subroutines.

4.2 Fast Approximate Taint Inference

To infer dependency of branches on input bytes, earlier fuzzers relied on the truth value of branch conditions: if changing the value of a particular byte changes the truth value of a branch, then it is inferred that the branch depends on this byte. For instance, in Fig. 1, to correctly infer the dependency of the branch at line 6, the engine first needs to select for mutation the input byte `x[100]` and then to change its value from any other than 40 to 40. GreyOne [12] proposed so-called *fuzzing-driven taint inference* FTI by switching the focus from the truth value of a branch to the value of the variables used in the branch. For instance, FTI determines the dependency of branch at line 6 on `x[100]` as soon as this input bytes is mutated, because this will lead to a change of the value of the variable A that is used in the branch. FTI is sound (no over-taint) and incomplete (some under-taint). Exact reasoning with provable soundness or completeness is not a direct concern in fuzzers, since they only use it to generate tests which are concretely run to exhibit bugs.

```
1 //input is uint8_t x[1024]
2 A = x[100] + 10;
3 B = (uint32_t *)x[200];
4 C = (uint32_t *)x[236];
5 ...
6 if ( A == 50)
7   ...
8 if( B + C < 200 )
9   ...
10 if( C  > 200 ) {
11   ...
12   if( A + C < 400 )
13     ...
14 }
```

Fig. 1. Branches with dependent input bytes.

The prime issue with FTI, which improves significantly over many other prior data-flow based engines, is efficiency. The taint is inferred by mutating bytes one-by-one in FTI. Thus, to infer the full dependency on all input bytes, the engine will require as many executions as the number of bytes. A seed may have tens of KBs, and there may be thousands of seeds, therefore the full inference may quickly become a major bottleneck in the fuzzer. On the other hand, precise or improved branch dependency may not significantly boost fuzzer bug-finding performance, thus long inference time may be unjustified. Hence, it is critical to reduce the inference time.

The TaintFAST Engine. We use *probabilistic group testing* [10] to reduce the required number of test executions for potential full inference from $O(n)$ to $O(\log n)$, where n is the number of input bytes. Instead of mutating each byte individually followed by program execution (and subsequent FTI check for each branch condition if any of its variables has changed), we simultaneously mutate multiple bytes, and then execute the program with the FTI check. We choose the mutation positions non-adaptively, according only to the value of n. This assures that dependency for many branches can be processed simultaneously.

Consider the code fragment at Fig. 1 (here $n = 1024$). We begin the inference by constructing 1024-bit binary vectors V_i, where each bit corresponds to one of the input bytes. A bit at position j is set iff the input byte j is mutated (i.e. assigned a value other than the value that has in the seed). Once V_i is built, we execute the program on the new input (that corresponds to V_i) and for each branch check if any of its variables changed value (in comparison to the values produced during the execution of the original seed). If so, we can conclude that the branch depends on some of the mutated bytes determined by V_i. Note, in all

prior works, the vectors V_i had a single set bit (only one mutated byte). As such, the inference is immediate, but slow. On the other hand, we use vectors with $\frac{1024}{2} = 512$ set bits and select $2 \cdot \log_2 1024 = 20$ such vectors. Vectors $V_{2 \cdot j}, V_{2 \cdot j+1}$ have repeatedly 2^j set bits, followed by 2^j unset bits, but with different starts. For instances, the partial values of the first 5 vectors V_i are given below on the right.

$$
\begin{aligned}
V_0 &= 10101010101010101010... \\
V_1 &= 01010101010101010101... \\
V_2 &= 11001100110011001100 11... \\
V_3 &= 00110011001100110011 00... \\
V_4 &= 11110000111100001111 00... \\
&... \\
\end{aligned}
$$

We execute the resulting 20 inputs and for each branch build 20-bit binary vector Y. The bit i in Y is set if any of the branch values changed after executing the input that corresponds to V_i. For instance, for the branch at line 6 of Fig. 1, $Y = 10100110100101101010$. Finally, we decode Y to infer the dependency. To do so, we initialize 1024-bit vector D that will hold the dependency of the branch on input bytes—bit i is set if the branch depends on the input byte i. We set all bits of D, i.e. we start by guessing full dependency on all inputs. Then we remove the wrong guesses according to Y. For each unset bit j in Y (i.e. the branch value did not change when we mutated bytes V_j), we unset all bits in D that are set in V_j (i.e. the branch does not depend on any of the mutated bytes V_j).

After processing all unset bits of Y, the vector D will have set bits that correspond to potential dependent input bytes. Theoretically, there may be under and over-taint, according to the following information-theoretic argument: Y has 20 bits of entropy and thus it can encode at most 2^{20} dependencies, whereas a branch may depend on any of the 1024 input bytes and thus it can have 2^{1024} different dependencies. In practice, however, it is reasonable to assume that most of the branches depend only on a few input bytes[2], and in such a case the inference is more accurate. For branches that depend on a single byte, the correctness of the inference follows immediately from group testing theory[3]. For instance, the branch at line 6 of Fig. 1 will have correctly inferred dependency only on byte `x[100]`. For branches that depend on a few bytes, we can reduce (or entirely prevent) over-taint by repeating the original procedure while permuting the vectors V_i. In such a case, each repeated inference will suggest different candidates, except the truly dependent bytes that will be suggested by all procedures. These input bytes then can be detected by taking intersection of all the suggested candidates. For instance, for the branch at line 8 (that actually depends on 8 bytes), a single execution of the procedure will return 16 byte candidates. By repeating once the procedure with randomly permuted positions of V_i, with high probability only the 8 actual candidates will remain.

The above inference procedure makes the implicit assumption that *same branches are observed across different executions.* Otherwise, if a branch is not

[2] C-type branches that contain multiple variables connected with AND/OR statements, during compilation are split into subsequent independent branches. Our inference is applied at assembly level, thus most of the branches depend only on a few variables.

[3] The matrix with rows $V_0, V_1, \ldots$ is 1-disjunct and thus it can detect 1 dependency.

observed during some of the executions, then the corresponding bit in Y will be undefined, thus no dependency information about the branch will be inferred from that execution. For some branches the assumption always holds (e.g. for branches at lines 6,8 in Fig. 1). For other branches, the assumption holds only with some probability that depends on their branch conditions. For instance, the branch at line 12 may not be seen if the branch at line 10 is inverted, thus any of the 20 bits of Y may be undefined with a probability of $\frac{200}{2^{32}}$. In general, for any branch that lies below some preceding branches, the probability that bits in Y will be defined is equivalent to the probability that none of the above branches will inverted by the mutations[4]. As a rule of thumb, the deeper the branch and the easier to invert the preceding branches are, the harder will be to infer the correct dependency. To infer deeper branches, we introduce a modification based on forced execution. We instrument the code so the executions at each branch will take a predefined control-flow edge, rather than decide on the edge according to the value of the branch condition. This guarantees that the target branches seen during the execution of the original seed file (used as a baseline for mutation), will be seen at executions of all subsequent inputs produced by mutating the original seed. We perform forced execution dynamically, with the same statically instrumented program, working in two modes. In the first mode, the program is executed normally, and a trace of all branches and their condition values is stored. In the second mode, during execution as the branches emerge, their condition values are changed to the stored values, thus the execution takes the same trace as before. No other variables aside from the condition values are changed. Note that our procedure aims to infer taint dependencies fast and optimistically; we refer readers to Sect. 4.6 for a discussion on these aspects.

4.3 Solving System of Intervals

It was noted in RedQueen [2], that when branches depend trivially on input bytes (so-called direct copies of bytes) and the branch condition is in the form of equality (either $=$ or $\neq$), then such branches can be solved trivially. For instance, the branch at line 1 of Fig. 2, depends trivially on the byte `x[0]` and its condition can be satisfied by assigning $x[0] = 5$ (or inverted by assigning $x[0] \neq 5$).

Thus it is easy to satisfy or invert such branches, as long as the dependency is correctly inferred and the branch condition is equality. Similar reasoning, however, can be applied when the condition is in the form of inequality over integers. Consider the branch at line 3 of Fig. 2, that depends trivially on the input byte $x[1]$. From the type of inequality (which can be obtained from the instruction code of the branch), and the correct dependency on the input byte $x[1]$ and the

```
1 if ( x[0] == 5 )
2   ...
3 if( x[1] < 100 )
4   ...
5 if( x[2]  > 10 ) {
6   ...
7   if( x[2] <=  200 )
8     ...
9     if( foo(x[2]) ==  0 )
10      ...
11 }
```

Fig. 2. Branches and systems of intervals.

[4] This holds even in the case of FTI. However, the probabilities there are higher because there is a single mutated byte.

constant 100, we can deduce the branch form $x[1] < 100$, and then either satisfy it resulting in $x[1] \in [0, 99]$, or invert it, resulting in $x[1] \in [100, 255]$. In short, we can represent the solution in the form of integer intervals for that particular input byte.

Often to satisfy/invert a branch we need to take into account not one, but several conditions that correspond to some of the branches that have common variables with the target branch. For instance, to satisfy the branch at line 7, we have two inequalities and thus two intervals: $x[2] \in [0, 200]$ corresponding to target branch at line 7 and $x[2] \in [11, 255]$ corresponding to branch at line 5. Both share the same input variable $x[2]$ with the target branch. A solution ($x[2] \in [11, 200]$) exists because the intersection of the intervals is not empty.

In general, Sivo builds a system of such constraints starting from the target branch, by adding gradually preceding branches that have common input variables with the target branch. Each branch (in)equality is solved independently immediately, resulting in one or two intervals (two intervals only when solving $x \neq value$, i.e. $x \in [0, value-1] \cup [value+1, maxvalue]$), and then intersection is found with the previous set of intervals corresponding to those particular input bytes. Keeping intervals sorted assures that the intersection will be found fast. Also, each individual intersection can increase the number of intervals at most by 4. Thus the whole procedure is linear in the number of branches along the executed path. As a result, we can efficiently solve these type of constraints and, thus, satisfy or invert branches that depend trivially on input bytes.

Even when some of the preceding branches do not depend trivially on input bytes, solving the constraints for the remaining branches gives an advantage in inverting the target branch. In such a case, we repeatedly sample solutions from the solved constraints and expect that the non-inverted branch constraints will be satisfied by chance. As sampling from the system requires constant time (after solving it), the complexity of branch inversion is reduced only to that of satisfying non-trivially dependent branches. For instance, to reach line 10, we first solve the lines 5, 7 to obtain $x[2] \in [11, 200]$, and then keep sampling $x[2]$ from this interval and hope to satisfy the branch at line 9 by chance.

4.4 More Accurate Coverage

AFL uses a simple and an elegant method to record the edges and their counts by using an array showmap. First, it instruments all basic blocks B_i of a program by assigning them a unique random label L_i. Then, during the execution of the program on a seed, as any two adjacent basic blocks B_j, B_k are processed, it computes a hash of the edge (B_j, B_k) as $E = (L_j \ll 1) \oplus L_k$ and performs showmap[E]++. New coverage is observed if the value $\lfloor \log_2 \text{showmap[E]} \rfloor$ of a non-zero entry showmap[E] has not been seen before. If so, AFL updates its coverage information to include the new value, which we will refer to as the logarithmic count.

Prevent Colliding Edge Hashes. CollAFL [13] points out that when the number of edges is high, their hashes will start to collide due to birthday paradox, and showmap will not be able to signal all distinct edges. Therefore, a fuzzer will

fail to detect some of the coverage. We propose a simple solution to the collision problem. Instead of assigning only one label L_i to each basic block B_i, we assign several labels $L_i^1, \ldots, L_i^m$, but use only one of them during an execution. The index of the used label is switched occasionally for all blocks simultaneously. The switch assures that with a high chance, each edge will not collide with any other edge at least for some of the indices. The number of labels required to guarantee that all edges will be unique with a high chance at some switch depends on the number of edges. Due to space restrictions we omit the combinatorial analysis. In our actual implementation the size of the showmap is 2^{16} and we use $m = 4$ labels per basic block – on average this allows around 8,000 edges to be mapped uniquely (and even 20,000 with less than 100 collisions), which is sufficiently high quantity for most of the programs considered in our experiments. By default, the index is switched once every 20 min.

Improve Compressed Edge Counts. The logarithmic count helps to reduce storing all possible edge counts, but it may also implicitly hinder achieving better coverage. This is because certain important count statistics that have the same logarithmic count as previously observed during fuzzing might be discarded.

For instance, if the for loop in Fig. 3 gets executed 13 times, then AFL will detect this as a new logarithmic count of $\lfloor \log_2 13 \rfloor = 3$, it will update the coverage, save the seed in the pool, and later when processing this seed, the code block $F1()$ will be executed as soon as the condition $C1$ holds. On the other hand, afterwards if the for loop gets executed 14 times, then the same logarithmic count $\lfloor \log_2 14 \rfloor = 3$ is achieved, thus the new seed will not be stored, therefore the chance of executing the code block $F2()$ is much lower. In other words, to reach $F2()$, simultaneously the for loop needs to be executed 14 times and $C2$ condition needs to hold. Hence, $F1()$ and $F2()$ cannot be reached with the same ease despite having similar conditional dependency, only because of AFL's logarithmic count mechanism.

```
1 count = 0;
2 for(i=0; i< x; i++)
3   count++;
4
5 if( 13 == count && C1 )
6   F1();
7 else if( 14 == count && C2 )
8   F2();
```

Fig. 3. The effects of AFL's edge count compression.

To avoid this issue, we propose flushing the coverage information periodically. More precisely, periodically we store the current coverage information, then reset it to zero, and during some time generate new coverage from scratch. After exhausting the time budget on new coverage, we keep only the seeds that increase the stored coverage, and continue the fuzzing with the accumulated coverage.

4.5 Design of the Whole Fuzzer Sivo

Sivo implements all the refinements mentioned so far. It uses the standard grey-box approach of processing seeds iteratively. In each iteration, it selects a seed, mutates it to obtain new seeds, and stores those that increase coverage.

Algorithm 1: OneIterationSIVO (Seeds, Coverage)

```
use_class ← MAB_select( Seed_class )              // choose seed class with MAB
use_crit ← MAB_select( Seed_criterion )           // choose seed criterion
seed ← Sample( use_class , use_crit, Seeds )      // sample seed from the pool
use_strategy ← MAB_select( Fuzzer_strategy )      // choose Data-flow or Vanilla
if use_strategy == Data-flow then
    Taint_inference(seed)                         // if Data-flow then infer dependency
tot_cov_incr ← 0
while time budget left do
    use_mut ← MAB_select( strategy )              // choose one mutation
    use_mut_params ← MAB_select( use_mut )        // choose its params
    new_seed ← Mutate( seed, use_mut, use_mut_params )   // apply mutation
    new_coverage ← ProduceCoverage(new_seed)
    cov_increase ← || new_coverage \ Coverage ||  // new coverage?
    if cov_increase > 0 then
        Seeds ← Seeds ⋃ new_seed                  // add new seed to the pool
        Coverage ← Coverage ⋃ new_coverage        // update coverage
    // feedback cov/sec to MAB to update the effectiveness of the chosen
    mutation and its params
    MAB_update( [use_mut , use_mut_params], cov_increase, while_time )
    tot_cov_incr += cov_increase
// feedback total cov/sec to MAB to update the effectiveness of the chosen
seed class/criterion and fuzzing strategy
MAB_update( [use_class,use_crit,use_strategy] , tot_cov_incr , iter_time)
```

In SIVO (refer to the pseudo-code in Algorithm 1), the seed selection is optimized: first with MAB the currently best class and best criterion are selected, and then a seed is sampled from the pool according to the chosen class and criterion. Afterwards, the fuzzer with the help of MAB decides on the currently optimal fuzzing strategy, either vanilla (apply mutations that do not require dependency information) or data-flow (require dependency). If latter, SIVO first infers the dependency (as a combination of FTI and TaintFAST). Then, according to the chosen fuzzing strategy the fuzzer again uses MAB to select one optimal mutation strategy. The vanilla fuzzing strategy allows a choice of 3 different mutations: 1) mutation of random bytes, 2) copy/remove of byte sequence of current seed, and 3) concatenation of different seeds. On the other hand, data-flow fuzzing strategy consists of 5 mutations: 1) mutation of dependent bytes, 2) branch inversion with system solver, 3) branch inversion by minimizing objective function, 4) branch inversion by mutation of their dependent bytes, and 5) reusing previously found bytes from other seeds to current seed. Most mutations have sub-versions or parameters which are also chosen with MAB. For instance, mutation of random bytes supports two versions: it can use heuristics to determine the positions of the bytes (choice 1), or use random byte positions (choice 2). If choice 1, then it needs to select the number of mutated bytes ($1, 2, 4, 8, 16, 32$, or 64). Both of these selections are determined with MAB. Each mutation is applied to the chosen seed to obtain a new seed, and then the seed is executed.

Algorithm 2: SIVO

```
Seeds ← Initial_seeds
Coverage ← ProduceCoverage(Seeds)
while true do
    OneIterationSIVO ( Seeds, Coverage );
    if time_to_switch_index then
        SwitchIndexInCoverage()
        Coverage ← ProduceCoverage( Seeds )
    if time_to_start_flush then
        Old_coverage, Old_seeds ← Coverage, Seeds
        Seeds ← Initial_seeds
        Coverage ← ProduceCoverage( Seeds )
    if time_to_stop_flush then
        New_coverage ← Coverage \ Old_coverage
        Coverage ← Coverage ⋃ Old_covarege
        Seeds ← Old_seeds ⋃ GetSeedsThatProduceCov(Seeds, New_coverage)
```

The coverage update information is fed back to the MAB, thus assuring that MAB can further optimize the selections.

SIVO runs the iterations and occasionally executes the code coverage refinements – refer to Algorithm 2. We implement the whole fuzzer from scratch in C++ with around 20,000 lines of code [25].

4.6 Limitations of SIVO

The taint engine TaintFAST relies on forced execution, which by definition is not sound, thus the inference is approximate. It means, the engine may introduce false positives/negatives, i.e. it may suggest dependencies of branches on incorrect input bytes. This, however, is not a real concern in fuzzing because later it leads solely to mutating incorrect input bytes, hence potentially it has only impact on efficiency[5], and does not affect the correctness of the fuzzer in any other way. The accuracy of the engine varies between programs. In certain cases (of particular traces), the forced execution crashes the program, and thus the inference has lower accuracy (because the corresponding Y bit is undefined). In our actual implementation of TaintFAST, we prevent some of the crashes by detecting with binary search sequences of input bytes that lead to crashes, and later eliminate them from consideration.

The refinement based on system of intervals is neither sound nor complete. Problems may appear due to incorrect inference of the intervals as well as due to the fact that the system describes only a partial dependency of the target branch on input bytes, i.e. includes only branches that can be presented in the form of integer intervals. Therefore, one may not assume that all of the branches can be properly inverted using this refinement.

[5] The impact can be reduced with various methods, e.g., the MAB-based optimization presented in this paper.

The remaining two refinements do not have apparent limitations, aside from affecting the efficiency in some cases.

5 Evaluation

We show that Sivo performs well on multiple benchmarks according to the standard fuzzing metrics such as code coverage (Sect. 5.2) and found vulnerabilities (Sect. 5.3). We evaluate the performance of each refinement in Sect. 5.4.

5.1 Experimental Setup

Experiment Environment. For all experiments we use the same box with Ubuntu Desktop 16.04, two Intel Xeon E5-2680v2 CPUs @2.80 GHz with 40 cores, 64 GB DDR3 RAM @1866 MHz and SSD storage. All fuzzers are tested on the same programs, provided with only one initial seed, randomly selected from samples available on the internet. To keep experiments computationally reasonable, while still providing a fair comparison of all considered fuzzers, we performed a two-round tournament-like assessment. In the first round, all fuzzers had been appraised over the course of 12 h. This interval is chosen based on Google's FuzzBench periodical reports, which shows that 12 h is sufficient to decide the ranking of the fuzzers usually [15]. The top 3 fuzzers from the first round that perform the best on average over all evaluated programs progress to the second round, in which they are run for 48 h.

Baseline Fuzzers. We evaluate Sivo in relation to 11 notable grey-box fuzzers. In addition to AFL [37], we take the extended and improved AFL family: AFLFast [4], FairFuzz [20], LAF-Intel [1], MOpt [22] and EcoFuzz [35]. Moreover, we include Angora [6] for its unique mutation techniques, Ankou [23] for its fitness function, and a few fuzzers that perform well on Google's OSS-Fuzz [15] platform such as Honggfuzz [32], AFL++ and AFL++_mmopt [11] (version 2.67c). To prevent unfair comparison, we omit from our experiments two categories of fuzzers. First, we exclude popular grey-box fuzzers that do not have an officially available implementation, such as CollAFL [13] and GreyOne [12]. We did not implement these fuzzers from scratch due to the complexity of such a task (e.g. the authors of GreyOne report 20K LoC implementation). Second, we exclude hybrid fuzzers because their approach is basically orthogonal to traditional grey-box fuzzers and thus they can be combined. For instance, the well-known hybrid fuzzer QSYM [36] inverts branches with symbolic execution and is built on top of AFL. With minor modification, QSYM could be built on top of Sivo instead of AFL, and this hybrid may lead to an even better performance.

Programs. Our choice of programs was influenced by multiple factors, such as implementation robustness, diversity of functionality, and previous analysis in other works. Our main goal of the evaluation is comparison of fuzzers according to a few criteria (including discovery of bugs), thus we use versions of programs that have already been tested in prior fuzzer evaluations on similar criteria. Due to limited resources, we did not run the fuzzers on the latest versions to look for actual

CVEs. Our final selection consists of 27 programs including: binutils (e.g.: readelf, nm), parsers and parser generators (e.g.: bson_to_json@libbson, bison), a wide variety of analysis tools (e.g.: tcpdump, exiv2, cflow, sndfile-info@libsndfile), image processors (e.g.: img2txt), assemblers and compilers (e.g.: nasm, tic@libncurses), compression tools (e.g.: djpeg, bsdtar), the LAVA-M dataset [9], etc. A complete list of the programs and their version under test is given in Table 1.

Efficiency Metrics. We use two metrics to compare the efficiency of fuzzers: edge coverage and the number of found vulnerabilities. To determine the coverage, we use the logarithmic edge count because this number is the objective in the fuzzing routines of the AFL family of fuzzers (simple count of unique edges leads to similar results which we omit due to space restrictions). To measure the total number of distinct vulnerabilities found by each fuzzer, first we confirm the reported vulnerabilities, i.e. we take all seeds generated by a fuzzer and keep those that trigger a crash by any of the sanitizers ASAN [28], UBSAN [27] and Valgrind [24]. Then, for each kept seed, we record the program source line where the crash triggers and count each such distinct source line as a vulnerability.

5.2 Coverage

We run all 12 fuzzers for 12 h each, and record the coverage discovered during the fuzzing. The results are reported in Fig. 4. We can see that at the end, Sivo provides the best coverage for 25 out of the 27 programs. On average Sivo produces 11.8% higher coverage than the next best fuzzer when analyzed individually for each program. In direct comparison to fuzzers, Sivo outperforms the next best fuzzer MOpt by 20.2%, and EcoFuzz by 30.6%, and outperforms the baseline AFL by producing 180% increase in coverage. For most of the programs, our fuzzer very soon establishes as the top fuzzer. In fact, the time frame needed to create advantage is so short, that the improved coverage refinement of Sect. 4.4 has still not kicked in, whereas the MAB optimization of Sect. 4.1 had barely any time to feed enough data back to the MABs. Thus, arguably the early advantage of Sivo is achieved due to the parametrize paradigm, as well as the remaining two refinements (TaintFAST and the system solver method).

We test the top three fuzzers Sivo, MOpt, and EcoFuzz on 48-h runs and report the obtained coverage in Fig. 5. We see that Sivo is the top fuzzer for 24 of the programs, with 13.4% coverage increase on average with respect to the next best fuzzer for each program, and 15.7%, and 28.1% with respect to MOpt and EcoFuzz. In comparison to the 12-h runs, the other two fuzzers managed to reduce slightly the coverage gap, but this is expected (given sufficient time all fuzzers will converge). However, the gap is still significant and Sivo provides consistently better coverage.

5.3 Vulnerabilities

We summarize the number of vulnerabilities found by each fuzzer on 25 programs during the 12-h runs in Table 1. (We removed two programs from Table 1, as none

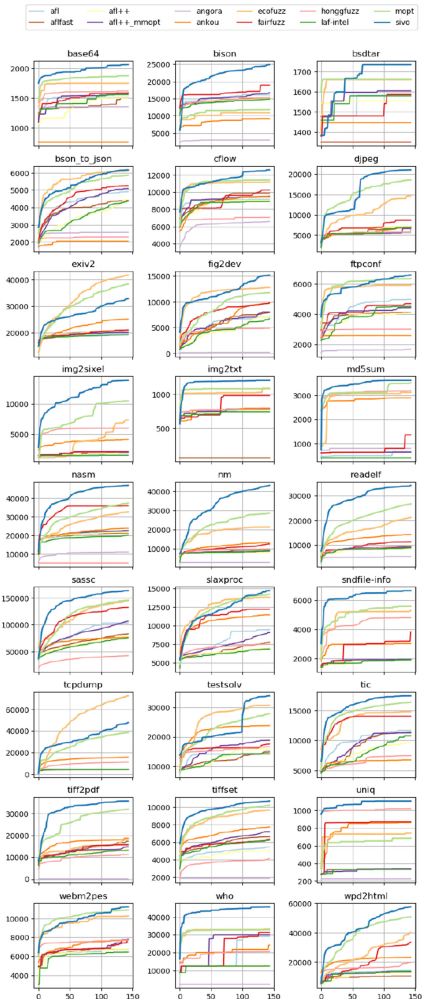

Fig. 4. Coverage for all fuzzers during 12 h of fuzzing (in 5 min increments).

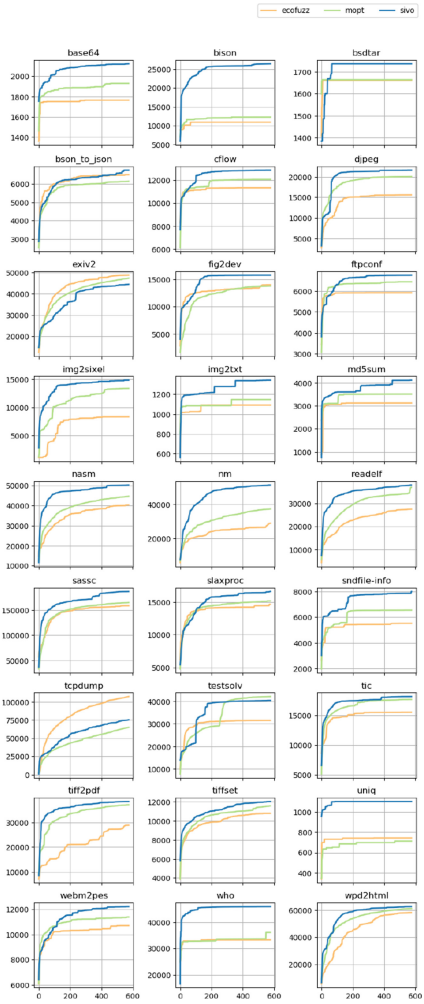

Fig. 5. Coverage for top three fuzzers Sivo, MOpt, EcoFuzz during 48 h of fuzzing.

of the fuzzers finds vulnerabilities for them.) Out of 25 evaluated programs, Sivo is able to find the maximal number of vulnerabilities in 18 programs (72%). For comparison, the next best fuzzer MOpt holds top positions in 11 programs (44%) in terms of vulnerability discovery. This indicates that Sivo is significantly more efficient at finding vulnerabilities than the remaining candidate fuzzers. However, Sivo achieves less top positions in discovery of vulnerabilities compared to code coverage, but this is not unusual as the objective of our fuzzer is code coverage, and the correlation between produced coverage and found vulnerabilities is not necessarily strong [17,18].

Table 1. The number of found vulnerabilities. The number of unique vulnerabilities (when non-zero) are reported after "/". "-" indicates failure to instrument/run the program. "#Vuln." and "#Vuln. uniq" give the number of all vulnerabilities and the number of unique vulnerabilities, respectively. "#Top vuln." shows the number of programs for which the fuzzer finds the maximal number of vulnerabilities. "#Prog. uniq" shows the number of programs for which the fuzzer finds some unique vulnerabilities.

Application	Version	Fuzzer											
		AFL	AFL++	AFL++_mmopt	AFLFast	FairFuzz	LAF-Intel	MOpt	EcoFuzz	Honggfuzz	Angora	Ankou	Sivo
base64	LAVA-M	2	2	2	2	2	2	2	2	2	2	1	2
bison	3.0.5	3	3	3	3	4/1	3	4/1	2	2	1	3	2
bson_to_json	1.8	2	1	1	1	2	2	2	1	1	2	1	2
cflow	1.5	2	1	1	2	2	1	5	3	2	1	3	6/1
exiv2	0.27.3	6	5	6	5	6	6	11/3	0	-	-	8	8
Fig. 2dev	3.2.7a	29/1	24	29	26	30/1	22	35	30/2	43/4	1	40	59/7
ftpconf	3.2.2	2	2	2	2	2	2	2	2	2	2	2	2
img2sixel	1.8.2	1	1	1	1	1	0	16/1	12/1	15/3	-	7	22/6
img2txt	0.99beta19	2	2	2	0	4	2	8/2	5/1	3	-	7/3	10/5
md5sum	LAVA-M	1	1	1	1	2/1	1	1	1	1	1	1	1
nasm	2.14rc15	4	4	5	4	8	4	10	8	2	5/1	9	13/1
nm	2.31	4	3	3	4	4	4	6/1	5	3	0	4	6/1
readelf	2.31	1	1	1	1	1	1	1	1	1	2/1	1	1
sassc	3.5	1	1	1	1	2	1	2	2	1	-	1	5/3
slaxproc	0.22.0	4	3	3	3	3	3	4	3	3	-	6/2	5/1
sndfile-info	1.0.28	0	0	0	0	3	0	8/2	6	13/6	-	1	7
tcpdump	4.10.0rc1	0	0	0	0	0	0	3	1	1	-	1	7/3
testsolv	0.7.2	6	6	6	6	6	6	7	8/1	14/8	-	6	9/2
tic	6.1	2	1	2	1	2	2	3	2	2	-	0	3
tiff2pdf	4.0.9	2	2	1	2	1	2	4	3	1	0	3	4
tiffset	4.0.9	1	1	1	1	1	1	1	1	1	0	1	1
uniq	LAVA-M	1	1	1	1	1	1	2	3	7	1	2	7
webm2pes	1.0.0.27	1	1	1	1	1	1	2	2	1	-	1	3/1
who	LAVA-M	1	1	1	1	1	1	7	3	6	0	3	7
wpd2html	0.10.1	0	0	0	0	0	0	1/1	1	0	-	1	1
#Vuln.		78	67	74	69	89	68	147	107	127	18	113	193
#Top vuln.		4	3	3	3	6	4	11	4	6	4	4	18
#Vuln. uniq		1	0	0	0	3	0	11	5	21	2	5	31
#Prog. uniq		1	0	0	0	3	0	7	4	4	2	2	11

We also measure and report in Table 1 the number of vulnerabilities unique to each fuzzer, i.e. bugs that are found only by one fuzzer, and not by any other. This metric signals distinctiveness of each fuzzer—the greater the number of unique vulnerabilities, the more distinct the fuzzer is on vulnerability detection. Out of 25 programs, Sivo discovers at least one unique vulnerability in 11 programs. In total, Sivo finds 31 unique vulnerabilities, while the next best fuzzer is Honggfuzz [32] with 21 vulnerabilities.

5.4 Performance of Refinements

We evaluate the four refinements individually, in terms of their impact and necessity. To assess the impact of a refinement, i.e. to estimate how much it helps to advance the fuzzer, we compare the performance of the baseline version of Sivo (where all four refinements have been removed) to the baseline version with the one refinement added on. On the other hand, to assess the necessity of a refinement, i.e. to estimate how irreplaceable in comparison to the other three refinements it is, we compare the full version of Sivo to the version with a

single refinement removed. We note that all refinements aside for the Parametrize-Optimize strategy, can be assessed reasonably well because it is easy to switch them on or off in the fuzzer. The same holds for Optimize, but not for Parametrize. As Sivo is built from scratch with many new fuzzing subroutines that are not necessarily present in AFL, it is not clear which fuzzing subroutines and which of their variations need to be removed in the baseline. Therefore, we only assess Optimize, and consider Parametrize to be part of the baseline.

We fuzz the 25 programs (on which Sivo outperformed all other 11 fuzzers) for 12 h, and compare the found coverage to the coverage produced by the complete version of Sivo. In Table 2, we provide the comparisons (as a percentage drop of the coverage) of the versions. We also give the data about the performance of the best non-Sivo fuzzer for each program (see the column Best NoneSivo). In the last row of the table we summarize the number of programs on which the considered version of the fuzzer is able to out-perform all of the remaining 11 none-Sivo fuzzers (for reference, for Sivo this number is 25).

A few observations are evident from the Table 2:

Table 2. Percentage drop in coverage of fuzzers in comparison to Sivo. When no drop occurs, the cells are empty.

Application	Fuzzer									
	Best NoneSivo	SivoBase	SivoBase+Opt	SivoBase+FI	SivoBase+SI	SivoBase+AC	Sivo-Opt	Sivo-FI	Sivo-SI	Sivo-AC
base64	9.1	7.2	2.4	7.2	4.7	7.2	3.3	3.3	8.4	1.7
bison	23.9	23.1		23.1	23.1	23.1	33.4			4.9
bsdtar	4.1						0.4	0.4	0.4	
bson_to_json	0.8	18.9	3.0	18.9	16.1	18.9	17.1		4.6	2.9
cflow	9.5	10.9	4.6	10.9	10.9	10.9	13.4		3.8	5.0
djpeg	11.9	23.9	23.9	23.9	23.9	21.2	33.7		15.4	22.0
fig2dev	15.9	13.3		13.3	13.3	13.3	23.0		1.8	6.1
ftpconf	3.5	10.5	0.6	9.8	9.9	10.5	12.3			1.4
img2sixel	24.7	21.9	8.0	21.5	15.9	21.9	19.9	3.8	7.7	0.6
img2txt	9.8	9.3	9.3	9.3	9.3	8.9	8.6	8.9	7.9	10.3
md5sum	2.9	14.8	14.2	14.8	0.6	6.4	4.6		12.3	
nasm	20.3	27.7	0.5	27.7	27.0	27.7	39.8		3.7	0.6
nm	33.6	11.8	11.8	11.8	6.6	11.8	15.9	43.6	27.4	18.8
readelf	22.0	15.2		7.7	5.0	7.1	7.8	1.9	0.1	
sassc	10.9	25.4		25.4	21.5	23.5	34.6			
slaxproc	3.3	34.9		31.1	28.0	30.3	38.5		1.9	
sndfile-info	16.2	17.2	10.8	17.2	11.7	17.2	6.6		18.6	1.7
testsolv	9.4	43.0	33.1	42.3	10.9	24.6	33.3	34.3	37.2	10.6
tic	6.0	16.9		16.9	16.7	13.7	19.7			0.1
tiff2pdf	10.5	2.4	2.4	2.4	2.0	0.3	3.7			
tiffset	4.4	8.9	7.8		8.9	8.9		0.3		
uniq	7.8	16.4	0.4	16.4	3.1	16.4		0.2	4.8	
webm2pes	3.0	14.0		12.6	14.0	14.0	12.2	7.1	6.6	
who	27.3	29.3	13.6	23.3	17.4	27.6	2.7	9.5	35.9	10.5
wpd2html	11.8	27.1	13.6	27.1	27.1	27.1	48.9	6.4	0.3	5.2
Top positions		9	19	11	13	9	11	22	18	21

- **Parametrize alone is valuable**. The baseline SivoBase, i.e. the version of the fuzzer that does not have any of the four refinements aside from Parametrize, already performs well. It is able to achieve the most coverage for 9 of the 25 considered programs. Hence, just by introducing new fuzzing subroutines and their variations, the fuzzer is able to outperform in terms of coverage the other 11 fuzzers on 36% of the fuzzed programs.
- **Optimize has a strong impact**. Among the four refinements, Optimize has the strongest impact. It helps the baseline fuzzer to add 10 top stops resulting in 19 top positions (refer to SivoBase+Opt column in Table 2), thus leading to most coverage in comparison to the other 11 none-Sivo fuzzers on 76% of the programs. On the other hand, Sivo without Optimize (refer to Sivo-Opt), loses 14 top positions, i.e. the fuzzer loses the top spot for 56% of the programs. Moreover, this refinement effects all of the fuzzed programs, with the exception of a few. The effect is significant—the coverage drop when this refinement is not present is at least 10% and sometimes more than 30%.
- **TaintFAST has a moderate to low impact**. This refinement, denoted as FI in the Table 2, helps the baseline fuzzer to add two top spots. On the other hand, Sivo without TaintFAST, i.e. with only the FTI engine present, loses three top spots. TaintFAST has a strong variance (refer to the Sivo-FI column) in terms of providing additional coverage and most fuzzed programs either benefit largely, or have no benefit at all. This is not unexpected, because the true benefit of TaintFAST is manifested in programs that accept large inputs and that have branches that depend on all of those inputs.
- **Solving systems of interval (SI) has a strong to moderate impact**. It adds 4 top stops to the baseline, and removes 7 top spots from the complete version of Sivo. It provides consistent benefits to the fuzzer – for most of the fuzzed programs SI produces extra coverage. Presumably, this is based on the fact that most programs do have branches based on integer inequalities and that use direct copy of input bytes.
- **Accurate coverage (AC) has a moderate to low impact**. This refinement does not have a strong impact on providing top positions (no jumps after adding it to the baseline, and lost 4 positions when removing it from Sivo), but it gives well balanced improvements in coverage to the fuzzer.

5.5 The Cause of Observed Benefits

It is important to understand and explain why certain fuzzing techniques (or in our case refinements) work well. In Sect. 5.4 we speculate about the type of programs that can be fuzzed well with some of the refinements. Showing this conclusively, however, is difficult. Table 2 shows the percentage drop in coverage observe, per application, obtained by adding and removing one-by-one each of our proposed refinements. However, attributing the cause of improved performance to individual refinements based on such coarse empirical data could be misleading. This is because we are measuring the joint outcome of mutually-dependent fuzzing strategies. We cannot single out the cause of an observed

outcome and attribute it to each strategy, since the strategies mutate the internal state that others use. We thus only coarsely estimate their impact via our empirical findings and speculate that these results extend to other programs.

6 Related Work

Grey-box fuzzers, starting from the baseline AFL [37], have been the backbone of modern, large-scale testing efforts. The AFL-family of fuzzers (e.g. AFLGo [3], AFLFast [4], LAF-Intel [1], MOpt [22], and MTFuzz [30]) improve upon different aspects of the baseline fuzzer. For instance, instead of randomly selecting mutation strategy, MOpt [22] uses particle swarm optimization to guide the selection. MTFuzz [30] trains a multiple-task neural network to infer the relationship between program inputs and different kinds of edge coverage to guide input mutation. Similarly, for the seed selection, AFLFast [4] prioritizes seeds that exercise low-probability paths, CollAFL [13] prioritizes seeds that have a lot of not-yet inverted branches, and EcoFuzz [35] uses multi-armed bandits to guide the seed selection. Common feature for all current fuzzers from the AFL-family is that they optimize at most one of the fuzzing subroutine[6]. In contrast, Sivo first parameterizes all aspects, i.e. introduces many variations of the fuzzing subroutines, and then tries to optimize all the selection of parameters. Even the seed selection subroutines of EcoFuzz and Sivo differ, despite both using multi-armed bandits: EcoFuzz utilizes MAB to select candidate seed from the pool, whereas Sivo uses MAB to decide on the selection criterion and the pool of seeds.

Several grey-box fuzzers deploy data-flow fuzzing, i.e. infer dependency of branches on input bytes and use it to accomplish more targeted branch inversion. VUzzer [26], Angora [6], BuzzFuzz [14] and Matryoshka [7] use a classical dynamic taint inference engine (i.e. track taint propagation) to infer dependencies. Fairfuzz [20], ProFuzzer [34], and Eclipser [8] use lighter engine and infer partial dependency by monitoring the execution traces of the seeds. RedQueen [2] and Steelix [21] can infer only dependencies based on exact (often called direct) copies of input bytes in the branches, by mutating individual bytes. Among grey boxes, the best inference in terms of speed, type, and accuracy is achieved by GreyOne [12]. Its engine called FTI is based on mutation of individual bytes (thus fast because it does not track taint propagation) and can detect dependencies of any type (not only direct copies of input bytes). FTI mutates bytes one by one and checks on changes in variables involved in branch conditions (thus accurate because it does not need for the whole branch to flip, only some of its variables). Sivo inference engine TaintFAST improves upon FTI and provides exponential decrease in the number of executions required to infer the full dependency, at a possible expense of accuracy. Instead of testing bytes one by one, TaintFAST uses probabilistic group testing and reduces the number of executions.

Data-flow grey boxes accomplish targeted branch inversion by randomly mutating the dependent bytes. A few fuzzers deploy more advanced strategies:

[6] This refers to optimization only – some fuzzers improve (but not optimize) multiple fuzzing subroutines.

Angora [6] uses gradient-descent based mutation, Eclipser [8] can invert efficiently branches that are linear or monotonic, and GreyOne [12] inverts branches by gradually reducing the distance between the actual and expected value in the branch condition. Some fuzzers, such as RedQueen and Steelix invert branches by solving directly the branch conditions based on equality (called magic bytes). Sivo can solve more complex branch inversion conditions that involve inequalities, without the use of SAT/SMT solvers. On the other hand, white boxes such as KLEE [5], and hybrid fuzzers such as Driller [31] and QSYM [36], use symbolic execution that relies on SMT solvers (thus it may be slow) to perform inversions in even more complex branches. The hybrid fuzzer Pangolin [16] uses linear approximations of branch constraints (thus more general than our intervals) called polyhedral path abstraction and later it utilizes them to efficiently sample solutions that satisfy path constraints. To infer the (more universal) linear approximations, Pangolin uses a method based on SMT solver. On the other hand, Sivo infers the (less universal) intervals with a simpler method.

The AFL-family of fuzzers as well as many other grey boxes track *edge coverage*. In addition, the AFL-family uses bucketization, i.e. besides edges, they track the counts of edges and group them in buckets that have ranges of powers of two. For practical purposes AFL does not record the precise edges (this will require storing whole execution traces which may be slow), but rather it works with hashes of edges (which is quite fast). The process of hashing may introduce collisions as noted by CollAFL [13]. To avoid such collisions, CollAFL proposes during compilation to choose the free parameters of the hashing function non-randomly, and according to a specific strategy. AFL++ [11] uses a similar idea and provides an open-source implementation based on link-time instrumentation. In addition, AFL++, LibFuzzer [29], and Honggfuzz [32] use so-called sanitizer coverage available in LLVM starting from version 11 to prevent collisions by assigning the free parameters during runtime. On the other hand, Sivo solution is to switch between different hashing functions during the fuzzing (i.e. at runtime). Instead of tracking edge coverage, a few fuzzers such as Honggfuzz [32], VUzzer [26] and LibFuzzer [29] track block coverage. Moreover, the grey-box fuzzer TortoiseFuzz [33] uses alternative coverage measurement metric (assigns different weights to edges based on their potential security impact) to prioritize testcases, and achieves higher rate of vulnerability detection.

7 Conclusion

We have presented four refinements for grey-box fuzzers that boost different fuzzing stages, specifically: (a) a faster dynamic taint dependency inference engine, (b) an integer inequality constraint learner and inference engine, (c) improved coverage tracker, and (d) complete parameterization of the strategies which can be optimized for dynamically. We have implemented the refinements in a fuzzer called Sivo. In comparison to 11 other popular grey-box fuzzers, Sivo scores highest with regards to coverage and number of vulnerabilities found.

Acknowledgments. We thank our shepherd Erik van der Kouwe for his helpful feedback. Abhik Roychoudhury, Zhijingcheng Yu, Shin Hwei Tan, Lu Yan, Andrea Fioraldi, and the anonymous reviewers gave us valuable comments and improvements on this work, for which we are thankful. All opinions expressed in this paper are solely those of the authors. This research is supported in part by the Crystal Centre at NUS and by the research grant DSOCL17019 from DSO in Singapore.

References

1. Circumventing fuzzing roadblocks with compiler transformations (2016). https://lafintel.wordpress.com/
2. Aschermann, C., Schumilo, S., Blazytko, T., Gawlik, R., Holz, T.: Redqueen: fuzzing with input-to-state correspondence. NDSS. **19**, 1–15 (2019)
3. Böhme, M., Pham, V.T., Nguyen, M.D., Roychoudhury, A.: Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 2329–2344 (2017)
4. Böhme, M., Pham, V.T., Roychoudhury, A.: Coverage-based greybox fuzzing as Markov chain. IEEE Trans. Softw. Eng. **45**(5), 489–506 (2017)
5. Cadar, C., Dunbar, D., Engler, D.R., et al.: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI **8**, 209–224 (2008)
6. Chen, P., Chen, H.: Angora: efficient fuzzing by principled search. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 711–725. IEEE (2018)
7. Chen, P., Liu, J., Chen, H.: Matryoshka: fuzzing deeply nested branches. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (2019)
8. Choi, J., Jang, J., Han, C., Cha, S.K.: Grey-box concolic testing on binary code. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 736–747. IEEE (2019)
9. Dolan-Gavitt, B., et al.: Lava: large-scale automated vulnerability addition. In: S&P (2016)
10. Du, D., Hwang, F.K., Hwang, F.: Combinatorial group testing and its applications, vol. 12. World Scientific (2000)
11. Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: Afl++: combining incremental steps of fuzzing research. In: 14th USENIX Workshop on Offensive Technologies WOOT) (2020)
12. Gan, S., et al.: Greyone: data flow sensitive fuzzing. In: 29th USENIX Security Symposium (USENIX Security 20). USENIX Association, Boston, MA (2020). https://www.usenix.org/conference/usenixsecurity20/presentation/gan
13. Gan, S., et al.: CollAFL: path sensitive fuzzing. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 679–696. IEEE (2018)
14. Ganesh, V., Leek, T., Rinard, M.: Taint-based directed whitebox fuzzing. In: 2009 IEEE 31st International Conference on Software Engineering, pp. 474–484. IEEE (2009)
15. Google: OSS-Fuzz - continuous fuzzing of open source software (2020). https://github.com/google/oss-fuzz
16. Huang, H., Yao, P., Wu, R., Shi, Q., Zhang, C.: Pangolin: incremental hybrid fuzzing with polyhedral path abstraction. In: 2020 IEEE Symposium on Security and Privacy (SP), pp. 1613–1627. IEEE (2020)

17. Inozemtseva, L., Holmes, R.: Coverage is not strongly correlated with test suite effectiveness. In: Proceedings of the 36th International Conference on Software Engineering, pp. 435–445 (2014)
18. Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M.: Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 2123–2138 (2018)
19. Kocsis, L., Szepesvári, C.: Discounted UCB. In: 2nd PASCAL Challenges Workshop, vol. 2 (2006)
20. Lemieux, C., Sen, K.: Fairfuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 475–485 (2018)
21. Li, Y., Chen, B., Chandramohan, M., Lin, S.W., Liu, Y., Tiu, A.: Steelix: program-state based binary fuzzing. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 627–637 (2017)
22. Lyu, C., Ji, S., Zhang, C., Li, Y., Lee, W.H., Song, Y., Beyah, R.: MOPT: optimized mutation scheduling for fuzzers. In: 28th USENIX Security Symposium (USENIX Security 2019), pp. 1949–1966 (2019)
23. Manès, V.J., Kim, S., Cha, S.K.: Ankou: guiding grey-box fuzzing towards combinatorial difference. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 1024–1036 (2020)
24. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: PLDI (2007)
25. Nikolic, I., Mantu, R.: Sivo: Refined gray-box fuzzer. https://github.com/ivicanikolicsg/SivoFuzzer
26. Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H.: Vuzzer: application-aware evolutionary fuzzing. NDSS **17**, 1–14 (2017)
27. Ryabinin, A.: Ubsan: run-time undefined behavior sanity checker (2014). https://lwn.net/Articles/617364/
28. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: Addresssanitizer: a fast address sanity checker. In: USENIX ATC (2012)
29. Serebryany, K.: Continuous fuzzing with libfuzzer and addresssanitizer. In: 2016 IEEE Cybersecurity Development (SecDev), pp. 157–157. IEEE (2016)
30. She, D., Krishna, R., Yan, L., Jana, S., Ray, B.: Mtfuzz: fuzzing with a multi-task neural network. In: FSE (2020)
31. Stephens, N., et al.: Driller: augmenting fuzzing through selective symbolic execution. NDSS **16**, 1–16 (2016)
32. Swiecki, R.: Honggfuzz: Security oriented software fuzzer. supports evolutionary, feedback-driven fuzzing based on code coverage (SW and HW based) (2020). https://honggfuzz.dev/
33. Wang, Y., et al.: Not all coverage measurements are equal: fuzzing by coverage accounting for input prioritization. NDSS (2020)
34. You, W., et al.: Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 769–786. IEEE (2019)
35. Yue, T., et al.: Ecofuzz: adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In: 29th USENIX Security Symposium (USENIX Security 20) (2020)

36. Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T.: QSYM: a practical concolic execution engine tailored for hybrid fuzzing. In: 27th USENIX Security Symposium (USENIX Security 2018), pp. 745–761 (2018)
37. Zalewski, M.: American fuzzy lop (2.52b) (2019). https://lcamtuf.coredump.cx/afl/