Efficient Input Minimization for Kernel Fuzzing via Relation- and Type-Guided Optimization*

HUI GUO†, School of Software Engineering, East China Normal University, Shanghai, China and The Chinese University of Hong Kong, Hong Kong, Hong Kong
HAO SUN, Department of Computer Science, ETH Zurich, Zürich, Switzerland
HUANG SHAN†, East China Normal University, Shanghai, China
TING SU†‡, East China Normal University, Shanghai, China
GEGUANG PU†, East China Normal University, Shanghai, China
SHAOHUA LI*, The Chinese University of Hong Kong, Hong Kong, Hong Kong

Coverage-guided kernel fuzzing is an effective technique for ensuring the reliability and security of operating system (OS) kernels. For example, Syzkaller, a widely used and representative coverage-guided kernel fuzzer, has successfully uncovered over 5000 bugs in Linux to date. Specifically, in kernel fuzzing, *input minimization* is a critical stage for improving fuzzing efficacy by compressing interesting inputs (programs) into shorter and coverage-preserving seeds. If this minimization stage is omitted, the achieved coverage and number of unique bugs found in Syzkaller would decrease by 27.5% and 40.4%, respectively, during a 48-hour fuzzing campaign. However, the minimization stage also takes over half of all fuzzing resources. During the 48-hour fuzzing campaign, 57.5% of program executions are expended in this stage, significantly limiting the potential of kernel fuzzing.

To the best of our knowledge, no prior work has explored or mitigated the problem described above in kernel fuzzing. To bridge this gap, this paper introduces two general and novel optimization strategies — *influence-guided call removal* and *type-informed argument simplification* — to reduce the cost of the minimization stage. The *key idea* of these two strategies is to reduce the number of program executions required to verify whether the new coverage achieved by the inputs is preserved.

*This paper, originally titled "Optimizing Input Minimization in Kernel Fuzzing" [19], was invited by ACM TOCS for extended publication through recommendation by the Chairs of USENIX ATC 2025. We have made substantial extensions in four aspects. (1) We introduced a more aggressive call removal attempt before the original attempt to extend the influence-guided call removal strategy; (2) We implemented the extended strategy on SyzMini, referred to as SyzMini+. The evaluation results show that SyzMini+ achieves higher code coverage and finds more unique bugs compared to SyzMini; (3) We also measured the attempt success ratio and the program length across different stages for both SyzMini (with the original influence-guided strategy) and SyzMini+ (with the extended influence-guided strategy) to explain why the extended strategy leads to higher efficiency during call removal; (4) To study the impact of different attempt orders on the extended influence-guided strategy, we further implemented two additional variants: SyzMini+ I and SyzMini+ II. The results show that the attempt order used by SyzMini+ achieves the highest coverage among all evaluated orders. Hui Guo was a visiting student at The Chinese University of Hong Kong during this work.

†Ting Su and Shaohua Li are the corresponding authors.
‡Hui Guo, Shan Huang, Ting Su, and Geguang Pu are also affiliated with Shanghai Key Laboratory of Trustworthy Computing.

Authors' Contact Information: Hui Guo, School of Software Engineering, East China Normal University, Shanghai, China and The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: guohui.study@gmail.com; Hao Sun, Department of Computer Science, ETH Zurich, Zürich, Switzerland; e-mail: hao.sun@inf.ethz.ch; Huang Shan, East China Normal University, Shanghai, Shanghai, China; e-mail: 52285902006@stu.ecnu.edu.cn; Ting Su, East China Normal University, Shanghai, Shanghai, China; e-mail: tsu@sei.ecnu.edu.cn; Geguang Pu, East China Normal University, Shanghai, Shanghai, China; e-mail: ggpu@sei.ecnu.edu.cn; Shaohua Li, The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: shaohuali@cse.cuhk.edu.hk.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 1557-7333/2026/8-ART
https://doi.org/10.1145/3840388

Just Accepted

We built SyzMini, a prototype based on Syzkaller that implements our two strategies to optimize input minimization. The evaluation shows that SyzMini significantly reduces the cost of the minimization stage by 60.7%, leading to a 12.5% improvement in branch coverage and a 1.7~2X increase in the number of unique bugs found. In addition, SyzMini has found 13 previously unknown bugs in the latest upstream kernel, all of which have been confirmed, with four already fixed. In terms of applicability, our optimization strategies can also improve the effectiveness of other kernel fuzzers (*e.g.*, SyzVegas, SyzDirect, and CountDown).

Additionally, based on the insight that calls with indirect influence on the target call have a low probability of actually affecting its execution, we extended the influence-guided call removal strategy by introducing an additional attempt. This attempt preserves only the direct influences to further reduce the cost during call removal. We also integrated the extended strategy into SyzMini, which we refer to as SyzMini$^+$. Our evaluation results demonstrate that SyzMini$^+$ achieves a coverage improvement of 14.5% to 16.5% and uncovers 1.9 to 2.3 times as many unique bugs as Syzkaller, further enhancing the effectiveness of SyzMini. We have released the implementations of both SyzMini and SyzMini$^+$ as open-source projects, which is available at [1].

CCS Concepts: • **Security and privacy → Operating systems security**; **Software security engineering**.

Additional Key Words and Phrases: Kernel Fuzzing, Input Minimization

1 Introduction

As the core component of an operating system (OS), the reliability and security of the OS kernel are of paramount importance. The bugs in the OS kernel could potentially lead to serious system-wide consequences, such as OS crashes [7, 18, 32], privilege escalation [2, 37] and arbitrary code execution [28, 58]. To ensure the reliability and security of the OS kernel, coverage-guided kernel fuzzing [9, 10, 22, 25, 47, 49, 53] has emerged as one of the most effective techniques for finding kernel bugs. For example, Syzkaller [17], the most popular and representative coverage-guided kernel fuzzer, has been integrated into the Linux kernel testing pipeline and has successfully found over 5000 bugs to date [52].

Generally, coverage-guided kernel fuzzing usually includes two major stages in its core fuzzing loop: (1) *mutation* and (2) *minimization*. The mutation stage aims to generate interesting input programs (in the form of system call sequences) that can achieve coverage in the tested kernel by mutating seeds. Since the interesting input programs are randomly generated, they are likely cluttered with redundant system calls or arguments. Therefore, an important stage, minimization, is adopted. The minimization stage compresses each interesting program into a more streamlined, higher-quality seed containing only essential calls and arguments, while preserving the new coverage. Minimization, in turn, improves the efficacy of subsequent mutations, thus being critical in the fuzzing campaign [43].

Taking Syzkaller as an example, it adopts a *one-by-one minimization strategy* [17] to minimize each interesting program that achieves new coverage. Given an interesting program P in the form of a system call sequence, this minimization strategy attempts to reduce P by (1) removing each call in P one by one, and *later* (2) simplifying each argument in the remaining calls one by one. Crucially, each minimization attempt must be verified that the new coverage achieved by P is always preserved by *dynamically* executing the reduced program; otherwise, the attempt is rolled back. Taking Figure 1 for instance, given a program P that contains six system calls where the last call c_6 achieves the new coverage, the one-by-one minimization strategy first removes c_1 and executes the reduced system call sequence. This attempt is successful as the coverage of c_6 does not change. Then this procedure is applied to each call in P for removing all calls that are irrelevant to c_6, thus five executions are required (Figure 1(a)). The attempt to remove c_3 fails and is rolled back, because the coverage achieved by c_6 is altered. Similarly, the minimization strategy then considers each argument in the remaining calls $[c_3, c_6]$ (Figure 1(b)), where three executions are conducted for a_3, r_1, and $[a_6]$. In total, eight program executions are needed to minimize P.

Just Accepted

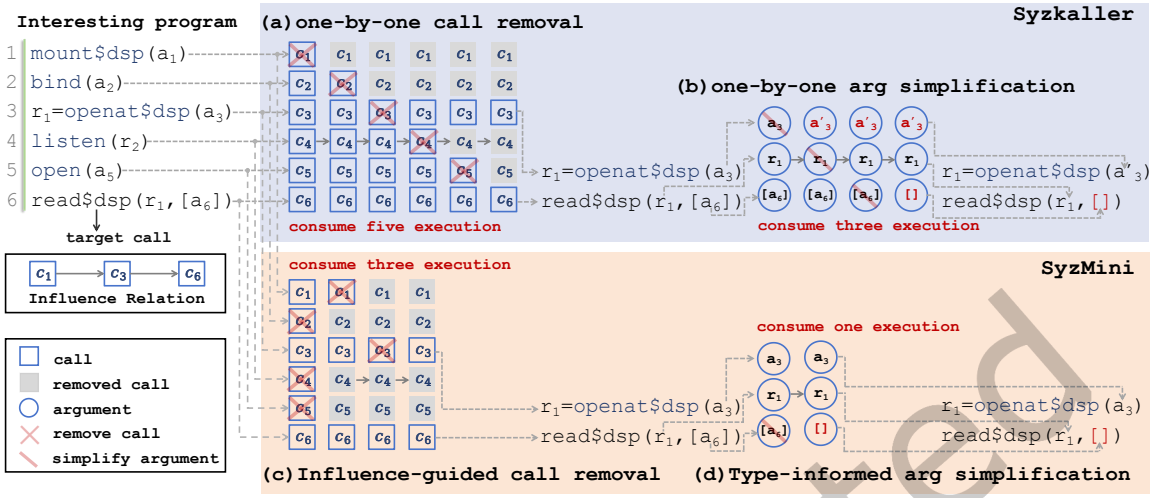


Fig. 1. Illustrative example of input minimization between Syzkaller and our approach SyzMini.

However, we observe that a significant *tension* exists between the benefits and the cost of the preceding minimization in kernel fuzzing. On the one hand, the minimization stage improves the efficacy of subsequent mutations by providing short, coverage-preserving seeds. Our experiment on Syzkaller (detailed in Section 3) shows that omitting the minimization stage, the achieved coverage and the number of unique bugs found would decrease by 27.5% and 40.4% within a 48-hour fuzzing campaign, respectively; moreover, the gap continues to widen with prolonged fuzzing. Indeed, this minimization stage is critical for the effectiveness of kernel fuzzing.

On the other hand, the minimization stage incurs significant fuzzing resource costs. For example, the one-by-one minimization strategy employed by Syzkaller requires *dynamic* program execution for each call and argument in the interesting program to attempt the program reduction. However, in kernel fuzzing, each program execution is expensive because it involves the time-consuming context switching between the user and kernel spaces. Our experiment (detailed in Section 3) reveals that, during a 48-hour fuzzing campaign, 57.5% of program executions are spent on the minimization stage, while the remaining 42.5% of program executions are used by the mutation stage. It is overwhelming that the minimization stage takes over half of the fuzzing resources.

However, to the best of our knowledge, no prior work has explored and mitigated the aforementioned tension in kernel fuzzing. This paper aims to fill this gap. The key challenge is how to significantly reduce the minimization stage cost used for reducing interesting programs. At a high level, our *key idea* is to reduce the number of program executions needed to verify whether the new coverage is preserved—the fewer the program executions, the lower the cost.

To this end, this paper introduces two *general* and *novel* strategies to optimize the minimization stage by reducing, respectively, the number of program executions during call removal and the number of program executions during argumentation simplification. In the scenario of call removal, let P be an interesting program and c be the target call in P that achieves new coverage. Our *key insight* is that removing the irrelevant calls in P, which have no execution influence on c, will not affect the new coverage achieved by c. Therefore, we could substantially reduce the number of program executions needed for verifying the coverage by removing these irrelevant calls *at one time* rather than *one by one*. Indeed, whether a call can affect c's coverage depends on its inherent dependencies. For example, if a call does not share any global kernel state with c, then this call has no execution influence on c_j and can be safely removed. In practice, this form of execution influence–often referred

Just Accepted

to as *influence relation* [9, 15, 20, 43, 49]–can be inferred through a combination of static and dynamic analysis on the system calls. To this end, we propose an *influence-guided call removal strategy* to optimize the call removal.

Figure 1(c) illustrates this strategy. The inferred influence relation (on the left side of Figure 1) shows c_3 has a direct influence on the execution of the target call c_6, and c_1 has an indirect influence on c_6 through c_3 (based on the transitivity of the influence relation). Our strategy first performs a single attempt that preserves both direct and indirect influences by removing calls that do not influence the target call at one time. In this way, the strategy identifies the calls c_2, c_4, and c_5 that have neither direct nor indirect influence on c_6, and then attempts to remove them at one time. If this attempt succeeds (*i.e.*, the new coverage achieved by c_6 is preserved), P will be reduced to $[c_1, c_3, c_6]$ by only a single program execution. Subsequently, we apply the one-by-one minimization strategy to further minimize the remaining calls in P, ensuring that each remaining call is truly associated with the new coverage of the target call and the number of calls in P is minimal. In this example, the remaining calls c_1 and c_3 require two program executions, ultimately reducing P to $[c_3, c_6]$ (assuming c_1 does not affect the new coverage of c_6 but c_3 does). As a result, our strategy requires only three executions to minimize P, whereas the original minimization requires five.

In the scenario of argument simplification, the one-by-one minimization strategy is conservative and will minimize every argument. As a result, it leads to a large number of program executions due to the numerous arguments and their sub-fields involved. Our *key insight* is that we only need to simplify those redundant arguments or their sub-fields that truly matter for the subsequent mutations. For example, removing superfluous elements from an array argument could significantly reduce the space for subsequent mutations, while investing similar effort in simplifying a simple scalar argument leads to marginal benefits. In this way, we could substantially reduce the number of program executions required to verify the coverage without sacrificing the benefits of minimization.

Based on the preceding insight, we design a *type-informed argument simplification strategy* to optimize the argument simplification. Given the interesting program P after the call removal, this strategy analyzes the type information of each argument and its sub-fields in P. If an argument or a field is fixed-size, the strategy skips the simplification; it only attempts to simplify variable-size arguments that may have redundant mutation space. For example, in Figure 1(d), this strategy does not simplify the fixed-size arguments a_3 (*Integer* type) and r_1 (*Resource* type, a 4-byte file descriptor). It would simplify the variable-size argument $[a_6]$ (*Array* type) to an empty array while ensuring the coverage is preserved. As a result, the type-informed strategy only consumes one program execution, while the original minimization requires three executions.

To evaluate the effectiveness of our approach, we integrated our proposed optimization strategies into Syzkaller, resulting in a prototype named SyzMini [1]. Our evaluation demonstrates that SyzMini significantly reduces the cost of the minimization stage: the number of program executions during the minimization stage is reduced by 60.7%. Compared to vanilla Syzkaller, SyzMini achieves a 12.5% improvement in branch coverage and a 1.7× speedup, while discovering 1.7–2× more unique bugs. Furthermore, SyzMini uncovered 13 previously unknown bugs in the latest upstream Linux, all of which have been confirmed, and four have already been fixed. To further illustrate the generality and benefits of our optimization strategies, we enhanced three distinct kernel fuzzing tools by integrating these strategies into the minimization stage: SyzVegas (employing reinforcement learning for scheduling fuzzing tasks), CountDown (accelerating finding memory-related kernel bugs), and SyzDirect (directed fuzzing for reproducing kernel bugs). With our enhancements, SyzVegas increases branch coverage by 14.5% and discovers 50% more bugs, CountDown identifies 66.7% more memory bugs, and SyzDirect reproduces 1.5× more bugs within the same time budget.

We further extended the influence-guided call removal strategy to enhance the efficiency of call removal. Our key insight is that calls with indirect influences on the target call have a low probability of actually affecting the execution of the target call, so these calls can also be removed at one time to save more program executions. Indeed, for a specific program, whether a call with an influence relation on the target call can actually affect

Just Accepted

the target call's execution depends on its runtime conditions (*e.g.*, the global kernel state and specific argument values). In many cases, these conditions are not satisfied, causing such a call — despite having a direct influence on the target call — may not actually affect the target call's execution, and the probability that the calls with indirect influences do so is even lower. For example, in Figure 1, *mount$dsp* ($c_1$) has an indirect influence on *read$dsp* ($c_6$) by *openat$dsp* (c_3). *mount$dsp* can affect the execution of *read$dsp* only if the mount point path of *mount$dsp* matches the file path opened by *openat$dsp* and *read$dsp* uses the file descriptor r_1 returned by *openat$dsp*. However, the argument values for the mount point path of *mount$dsp* and the file path opened by *openat$dsp* differ, meaning that *mount$dsp* ($c_1$) does not actually affect the execution of *read$dsp* ($c_6$) and removing c_1 also does not alter the new coverage achieved by c_6. Therefore, we consider aggressively removing the calls that have even an indirect influence on the target call (*e.g.*, c_1) at one time.

To this end, we introduce an additional attempt before the influence-guided call removal strategy to improve its efficiency. The introduced attempt preserves only the *direct* influences — removing the calls that have an indirect influence or no influence on the target call at one time — to verify whether the new coverage achieved by the target call is altered. Taking the program in Figure 1(c) as an example, c_1 has an indirect influence on c_6 while c_2, c_4, and c_5 do not influence c_6. Thus, these calls c_1, c_2, c_4, and c_5 (except c_3) are identified as irrelevant calls in the first attempt, and they will be removed at one time. If this attempt fails, it indicates that some of the removed calls affect the new coverage of c_6 (*e.g.*, the calls that have an indirect influence on c_6 can actually affect its execution). In this case, the extended strategy will proceed to the attempt in the original influence-guided call removal strategy as the second attempt, preserving both the direct and indirect influences by removing the calls that have no influence on the target call at one time. In this way, the calls c_2, c_4, and c_5 have no influence on c_6, so they will be identified as irrelevant calls and be removed at once in the second attempt. However, in this example, the first attempt succeeds, reducing P to $[c_3, c_6]$ in one program execution. Later, the extended strategy attempts to remove the remaining call c_3, which, however, fails to preserve the new coverage. As a result, the extended strategy only consumes two executions to minimize P, which further reduces one program execution compared to the original influence-guided call removal strategy.

We integrated the extended influence-guided call removal strategy into SyzMini, referred to as SyzMini$^+$. Our evaluation results show that SyzMini$^+$ achieves a 14.5%~16.5% coverage improvement and finds 1.9~2.3× unique bugs compared to Syzkaller, surpassing SyzMini 's 12.1%~13.3% coverage improvement and 1.7~1.9× unique bugs. This highlights that SyzMini$^+$ further enhances the effectiveness of SyzMini. Furthermore, integrating this extended strategy into the minimization stages of SyzVegas, CountDown, and SyzDirect yields greater improvements in their overall effectiveness compared to the original strategy used in SyzMini. We also implemented two variants of SyzMini$^+$ to evaluate the impact of different attempt orders on the effectiveness of the influence-guided call removal strategy. The results demonstrate that the attempt order of SyzMini$^+$— first attempts to preserve only the direct influences and then attempts to preserve both the direct and indirect influences if the first attempt fails — achieves the highest coverage.

Overall, we have made the following contributions:

- To our knowledge, we are the *first* to explore and mitigate the significant tension between the benefits and costs of input minimization in kernel fuzzing.
- We introduce two general and novel optimization strategies, respectively, for call removal and argument simplification in the minimization stage.
- We implement our optimization strategies for minimization in Syzkaller and three distinct kernel fuzzers. The extensive evaluation results demonstrate the effectiveness, general applicability, and benefits of our approach.

Just Accepted

- We further extend the influence-guided call removal strategy and implement the extended strategy on SyzMini, referred to as SyzMini$^+$. The evaluation results show that SyzMini$^+$ reduces more program executions during call removal compared to SyzMini.

2 Background

Fuzzing, first introduced by Miller[40], is an automated software testing technique designed to uncover bugs by repeatedly sending random inputs to the target program [16, 39]. Since the success of AFL, a substantial body of research has emerged that integrates various techniques to enhance fuzzing efficiency. These advancements include taint analysis [6, 35, 36], symbolic execution [33, 42, 44] and static analysis [11, 41, 54].

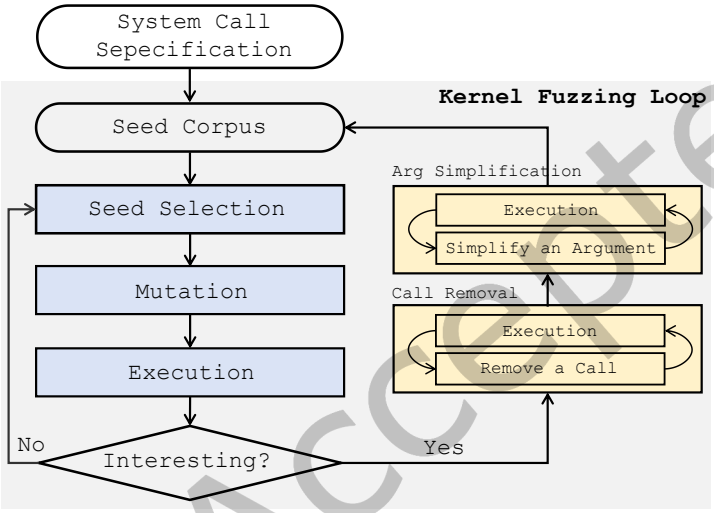


Fig. 2. Overall workflow of Syzkaller.

Kernel fuzzing, a domain-specific fuzzing technique for OS kernels, continuously generates different system call sequences to find kernel bugs. Figure 2 shows the workflow of Syzkaller [17], a state-of-the-art coverage-guided kernel fuzzing tool. Syzkaller first loads the crafted system call descriptions written in the declarative language *Syzlang* [51] as well as the initial seed corpus, and then conducts a fuzzing loop to explore interesting programs and find new bugs. Specifically, the fuzzing loop includes two major stages, *i.e.*, *mutation* and *minimization*.

Mutation Stage. The mutation stage first selects a seed from the corpus (the programs that previously found new coverage), and then performs a series of random mutations on this seed (*e.g.*, inserting a call, removing a call, or changing the value of an argument or its sub-fields) to mutate more diverse programs. Each new mutated program will be executed on the tested kernel, and the coverage achieved by each call in this program is collected independently. For example, given a system call sequence $[c_1, c_2, ..., c_n]$, the coverage achieved by each call is collected separately in n branch arrays (*e.g.*, cov_c_1, cov_c_2, ..., cov_c_n where cov_c_1 saves the coverage of c_1). Additionally, the total coverage achieved so far is maintained in a global set, so the fuzzer can identify the calls that achieve new coverage by simply comparing each branch array with the global set, *e.g.*, comparing cov_c_1 with the global set to determine if the call c_1 achieves new coverage. When one program achieves new coverage, each call contributing to this new coverage is treated as a target call for the next minimization.

Minimization Stage. The minimization stage aims to minimize an interesting program as a new seed for subsequent mutations. It first attempts to remove irrelevant calls and later simplifies the arguments of the

Just Accepted

Table 1. Argument simplification methods in Syzkaller.

Data Types		Simplification Methods
Primitive Type	Integer Flag Protocol Resource	Replace the current argument value with the default value of this type
Derived Type	Array Buffer	Minimize the number of elements in the argument by the binary search
	Pointer	Simplify the object pointed to by the pointer argument
User-defined Type	Struct Union Enum	Recursively simplify each sub-field of this argument

remaining calls in the program. Both call removal and argument simplification employ a one-by-one minimization strategy. We explain these two processes as follows.

① *Call Removal.* Given an interesting program $P = [c_1, c_2, ..., c_n]$, where c_i is the ith system call in P. Let the last call c_n be the target call that achieves the new coverage. In the minimization stage, Syzkaller first removes the first call c_1 and executes the reduced program $[c_2, c_3, ..., c_n]$ to verify whether the new coverage achieved by c_n is preserved. One case is that c_n's coverage is preserved, which means c_1 is an irrelevant call *w.r.t.* c_n that does not affect the execution of c_n. Thus, c_1 will be removed from P and P will be reduced to $[c_2, c_3, ..., c_n]$. In the other case, P remains unchanged to preserve the coverage of c_n. Later, Syzkaller attempts to remove the next call c_2. This process continues until each system call c_i in P (except c_n) has been attempted. Therefore, it requires n-1 program executions to remove the irrelevant calls in P.

② *Argument Simplification.* After the call removal, the interesting program P becomes P' and assuming there are m arguments $[a_1, a_2, ..., a_m]$ in P'. Syzkaller would first attempt to simplify the first argument a_1 to a_1' and execute the simplified program to verify whether the new coverage is preserved. If the coverage is preserved, a_1' replaces a_1; otherwise, a_1 remains unchanged. Each argument will be attempted one by one to minimize P'. Therefore, it requires m program executions to simplify these arguments in P'.

The argument simplification depends on the argument type. Table 1 lists the detailed simplification methods for different argument types in Syzkaller. For the argument of a primitive type (*e.g.*, *Integer*, *Flag*, *Protocol*, or *Resource*), Syzkaller replaces the argument with the default value of this type. For the argument of a derived type (*e.g.*, *Array*, *Buffer*), Syzkaller attempts to minimize the number of elements by binary search. For the argument of a user-defined data type (*e.g.*, *Struct*, *Union*), Syzkaller recursively simplifies each sub-field of this argument one by one.

3 Motivation and Observation

In this section, we demonstrate and characterize the tension that exists between the benefits and the cost of the minimization stage in kernel fuzzing. To this end, we selected Linux 5.15, the widely used and stable long-term Linux kernel version, as the fuzzing target. For the kernel fuzzer, we chose Syzkaller, the most popular kernel fuzzing tool. Specifically, we implemented a version of Syzkaller that omits the minimization stage (named Syzkaller$^-$). For each tool, we allocated 48 hours per fuzzing campaign to fuzz the kernel and repeated each campaign 10 times to mitigate the effects of randomness in our analyses.

Just Accepted

Observation 1: The minimization stage is crucial in improving the effectiveness of kernel fuzzing in terms of the achieved coverage and the number of unique bugs found.

We compared the achieved branch coverage and the number of unique bugs found between Syzkaller and Syzkaller⁻. As shown in Figure 3, compared to Syzkaller, Syzkaller⁻ exhibits a 27.5% decrease in branch coverage and a 40.4% reduction in the number of unique bugs. Furthermore, this gap continues to widen over time. These results show that the minimization stage plays a crucial role in improving the effectiveness of kernel fuzzing.

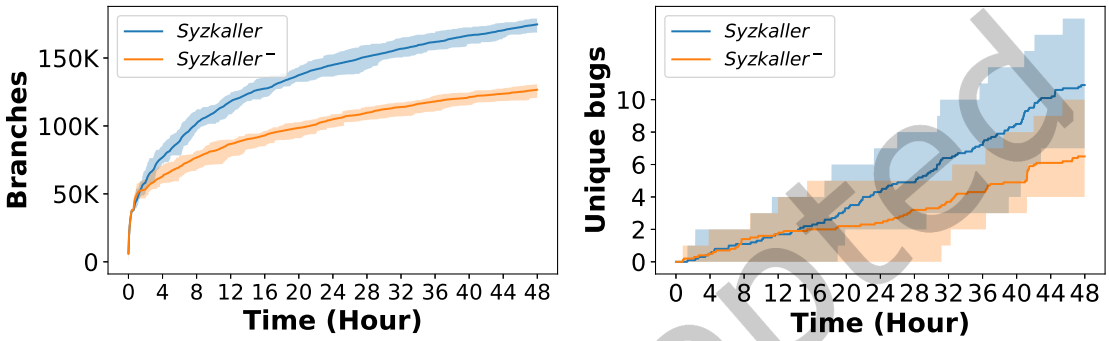


Fig. 3. Syzkaller⁻ *vs.* Syzkaller on average branch coverage and the number of unique bugs (shaded area indicates the min–max range across 10 rounds).

We recorded the number of programs executed during the mutation and minimization stages, respectively, to evaluate the cost. Figure 4 illustrates the proportions of the number of program executions between these two stages. We can see that the minimization stage incurs a higher number of program executions compared to the mutation stage. Specifically, the minimization stage reaches a peak of 68.1% at the time of 4.5 hours. The portion gap gradually decreases over time, but still accounts for 57.5% at the time of 48 hours. In particular, call removal and argument simplification accounted for 34.0% and 66.0% of the executed programs, respectively.

Observation 2: The proportion of program executions during the minimization stage accounts for 57.5%, leading to significant fuzzing resource costs.

The reason for this trend is that the fuzzer initially generates a large number of interesting programs with new coverage. Minimizing these programs requires a considerable number of program executions. As a result, the proportion of program executions in minimization rapidly increases in the first four hours. Over time, although the number of interesting programs gradually decreases, these programs become more complex (*e.g.*, involving more calls and arguments) due to the subsequent mutations. These complex programs require more program executions to conduct the one-by-one minimization. Overall, the proportion of program executions in the minimization stage decreases slowly, but still surpasses that of the mutation stage (after 48 hours), leading to significant fuzzing resource cost.

Summary: The preceding observations highlight that the minimization stage is crucial for enhancing the effectiveness of kernel fuzzing, but also introduces high cost in terms of program executions. Our goal is to reduce this minimization cost, thereby unleashing the potential of kernel fuzzing.

Just Accepted

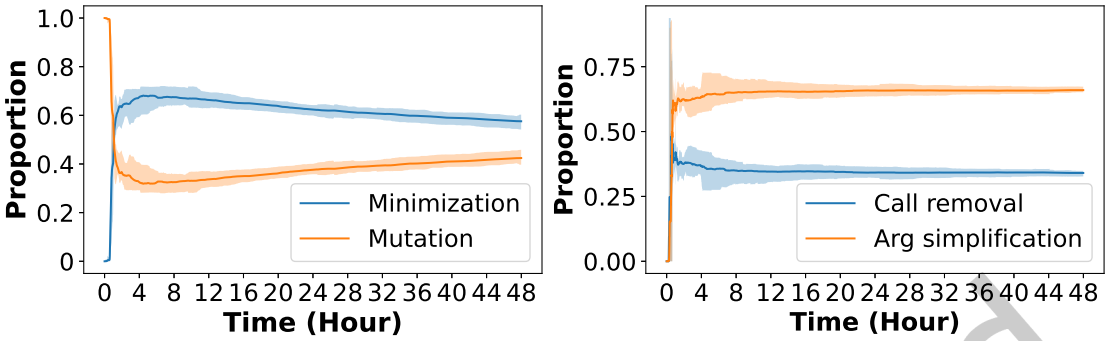


Fig. 4. The proportion of program executions during the minimization and mutation stages over time (shaded area indicates the min–max range across 10 rounds).

4 Approach and Implementation

The overview of our proposed optimization strategies (influence-guided call removal and type-informed argument simplification) is shown in Figure 5. The two strategies extend the default one-by-one minimization strategy adopted by Syzkaller. Specifically, based on the influence relations between system calls, the influence-guided call removal strategy first performs a single attempt. This attempt preserves both the direct and indirect influences to verify whether the new coverage achieved by the target call is altered. Regardless of whether this attempt succeeds or not, the reduced program will undergo the one-by-one minimization to guarantee the number of calls in P is minimal (detailed in Section 4.1).

Additionally, we further extend the influence-guided call removal strategy by introducing an additional attempt, as illustrated by the dashed lines in Figure 5. The introduced attempt aims to remove more irrelevant calls by preserving only direct influences. If this attempt fails, the extended strategy proceeds to the original attempt—which preserves both direct and indirect influences—as the second attempt. Regardless of whether these two attempts succeed, the reduced program then undergoes the one-by-one minimization. The detailed process is described in Section 4.2.

The type-informed argument simplification strategy first analyzes the types of each argument and its subfields to determine whether each argument is fixed-size or variable-size. Then it skips the simplifications of fixed-size arguments and only attempts to simplify the variable-size arguments, as the latter truly matters for the subsequent mutation efficiency (detailed in Section 4.3).

4.1 Influence-guided Call Removal Strategy

We define the influence relation between two system calls:

Influence Relation*: For two system calls c_i and c_j, c_i has an influence relation on c_j if executing c_i can change the execution path of c_j by modifying the kernel's internal state that c_j depends on.*

The definition describes the influence relation between system calls in terms of the impact on the execution path. This influence relation exists when one call c_i can modify the global kernel state that the behavior of another call c_j depends on, thereby changing the execution path of c_j. In practice, the influence relation can be inferred by performing static and dynamic analyses on system calls [49]. The process of collecting the influence relations will be detailed in Section 4.4. In this section, we focus on explaining how to leverage these collected influence relations to reduce the cost of call removal. Thus, we assume the influence relations of n calls have been collected and stored in M—a two-dimensional matrix $R^{n \times n}$.

Just Accepted

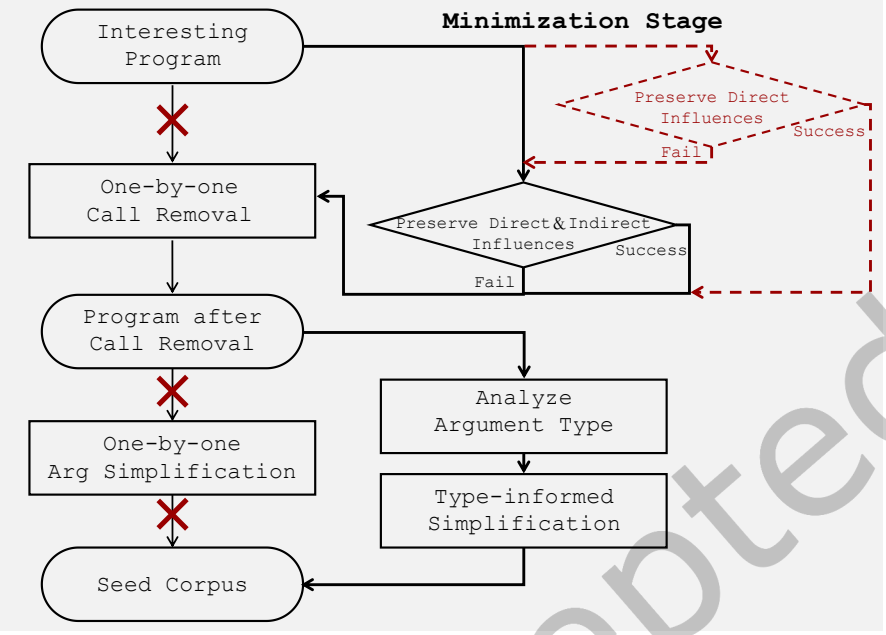


Fig. 5. Overview of the influence-guided call removal and the type-informed argument simplification strategies (the dashed lines indicate the introduced attempt to the influence-guided call removal strategy).

Preserve Direct&Indirect Influences. The strategy first performs a single attempt to remove the calls that have neither direct nor indirect influence on the target call at one time. This attempt consists of two steps and is described by lines 3-6 in Algorithm 1.

Step 1: Identify irrelevant calls. Based on the collected influence relations, this step aims to identify all the irrelevant calls that have neither direct nor indirect influence on the target call (lines 7-20). Let $P = [c_1, c_2, ..., c_n]$ be an interesting program (assuming the last call c_n in P is the target call that achieves the new coverage) and M be the influence relation matrix. The strategy first obtains all the calls that have the *direct* influence relation with c_n (lines 8-11), and then identifies the calls that have the indirect influence relation with c_n based on the transitivity of the influence relation (lines 12-18). Specifically, on line 3, when $M[c_i][c_n]$=1 (denoting that c_i has direct influence on c_n), c_i will be saved into the list *relevant_calls* (which stores all the relevant calls). On line 9, when $M[c_k][c_i]$=1 (denoting that c_k has indirect influence on c_n through c_i), c_k will be also saved into the list *relevant_calls* (line 17). In this way, the strategy uses a worklist algorithm to iteratively identify all the calls that have indirect influence on c_n. Finally, we obtain the set of irrelevant calls (stored in *irrelevant_calls*) *w.r.t.* P by filtering these analyzed relevant calls from the set of the calls from P (line 20).

Example. In Figure 6, the program P is an interesting program and c_6 is the target call. According to the influence relation, c_3 has a direct influence on c_6, and c_1 has an indirect influence on c_6 through c_3. Thus, $\{c_1, c_3\}$ are the relevant calls and the other calls $\{c_2, c_4, c_5\}$ in P are identified as irrelevant.

Step 2: Remove irrelevant calls. When the irrelevant calls are identified, the strategy attempts to remove these calls at one time from P (lines 21-28). If the new coverage achieved by the target call is preserved by P', this attempt is successful, and P is reduced to P' (line 26). Otherwise, P remains unchanged — some calls in *irrelevant_calls* actually influence the execution of c_n.

Just Accepted

Algorithm 1: Influence-guided Call Removal Strategy

Input : Interesting Program: $P = [c_1, c_2, ..., c_n]$, Influence relation matrix: M
Output: Program after call removal P

```
preserve_direct&indirect_influences(&P, M)
one_by_one_call_removal(&P)
Function preserve_direct&indirect_influences(P, M)
    irrelevant_calls ← identify_irrelevant_calls(P, M)
    is_success ← remove_irrelevant_calls(P, irrelevant_calls)
    return is_success
Function identify_irrelevant_calls(&P, M)
    relevant_calls ← {}
    // Identify the calls with the direct influence on the target call
    for i from 1 to n − 1 do
        if M[c_i][c_n] = 1 then
            relevant_calls ← relevant_calls ∪ {c_i}
    // Identify the calls with the indirect influence on the target call based on
       transitivity
    work_list ← all relevant calls
    while work_list is not empty do
        Pick the last call c_i from work_list
        for k from i − 1 to 1 do
            if M[c_k][c_i] = 1 ∧ c_k ∉ relevant_calls then
                relevant_calls ← relevant_calls ∪ {c_k}
                Add c_k to work_list
    irrelevant_calls ← {c_1, c_2, ..., c_n} \ relevant_calls
    return irrelevant_calls
Function remove_irrelevant_calls(&P, irrelevant_calls)
    // Remove the irrelevant calls from P at once
    P' ← P.remove_calls(irrelevant_calls)
    target_call_cov ← execute(P)
    target_call_cov' ← execute(P')
    // this attempt succeed
    if target_call_cov ⊆ target_call_cov' then
        P ← P'
        return true
    return false
```

Example. After identifying $\{c_2, c_4, c_5\}$ as the irrelevant calls, the strategy attempts to remove all these calls at one time. In this example, this attempt succeeds and P is reduced to $[c_1, c_3, c_6]$ by only requiring one program execution.

Just Accepted

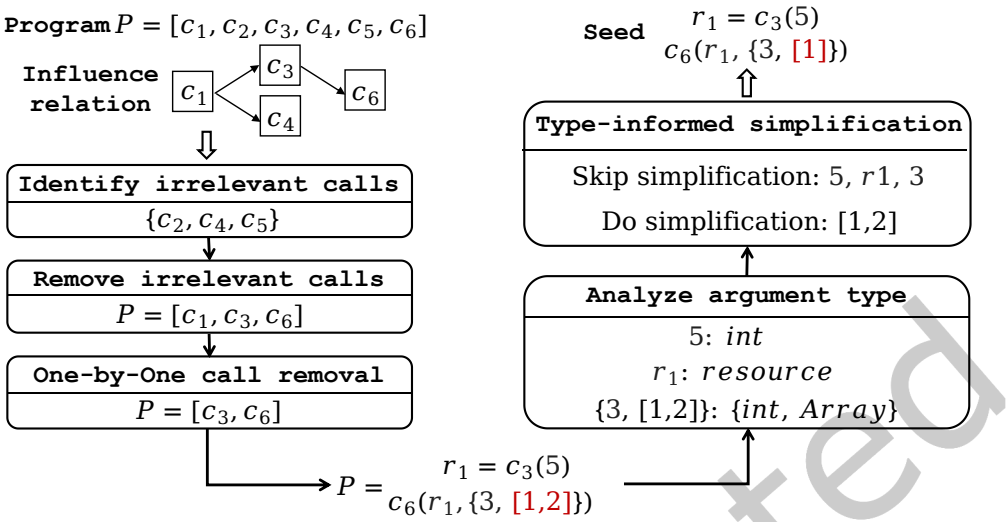


Fig. 6. An Example for Influence-guided Call Removal and Type-informed Argument Simplification.

One-by-one call removal. No matter whether this attempt is successful or not, the remaining relevant calls in P may not contribute to the new coverage. This is because influence relationships are inherently context-sensitive and depend on specific execution states (*e.g.*, call sequences and argument values). Given the enormous state space of the Linux kernel, it is impractical to precisely model and maintain context-sensitive influence relationships for all possible execution states. Therefore, existing approaches, including ours, represent influence relationships at the call level (i.e., call a can influence call b) while ignoring the execution context. As a result, these collected relationships cannot guarantee complete accuracy under the current program context. Thus, P still needs to undergo the one-by-one call removal (*i.e.*, the default one-by-one minimization for removing calls) to ensure that each remaining call is truly associated with the new coverage of the target call and the number of calls in P is minimal (line 2). Compared to the default one-by-one minimization strategy in Syzkaller, this influence-guided strategy is likely to remove many irrelevant calls at one time and thus reduce the cost of the minimization stage.

Example. In Figure 6, the program P becomes $[c_1, c_3, c_6]$ after the attempt removal. Thus, the one-by-one minimization strategy consumes two program executions to check whether c_1 and c_3 would affect the new coverage achieved by c_6, and finally P is reduced to $[c_3, c_6]$ (assuming c_1 actually does not influence the execution of c_6 but c_3 does). As a result, the influence-guided call removal optimization strategy only takes three program executions to remove all the irrelevant calls, while the default one-by-one minimization strategy requires five program executions.

4.2 Extension of Influence-guided Call Removal Strategy

The attempt in the original influence-guided call removal strategy preserves both direct and indirect influences. However, the calls that have an indirect influence on the target call have a low probability of actually affecting the new coverage achieved by that target call. This is because whether an influence relation actually affects execution depends on the runtime conditions (e.g., the global kernel state and specific argument values), which are often not satisfied in practice. In many cases, direct influences may fail to materialize under these conditions, making the likelihood that indirect influences take effect even lower. Therefore, these indirectly influential calls can be removed at one time to save more program executions. To this end, we introduce a more aggressive attempt that preserves only direct influences to extend the influence-guided call removal strategy.

Just Accepted

Algorithm 2: Extended Influence-guided Call Removal Strategy

```
   Input  : Interesting Program: P = [c_1, c_2, ..., c_n], Influence relation matrix: M
   Output : Program after call removal P
1  if preserve_direct_influences(P, M) == false then
2      preserve_direct&indirect_influences(P, M)
3  one_by_one_call_removal(P)
4  Function preserve_direct_influences(&P, M)
       // Step1: Identify irrelevant calls
5      irrelevant_calls ← {}
6      for i from 1 to n − 1 do
7          if M[c_i][c_n] = 0 then
8              irrelevant_calls ← irrelevant_calls ∪ {c_i}
       // Step2: Remove irrelevant calls
9      P′ ← P.remove_calls(irrelevant_calls)
10     target_call_cov ← execute(P)
11     target_call_cov′ ← execute(P′)
       // this attempt succeed
12     if target_call_cov ⊆ target_call_cov′ then
13         P ← P′
14         return true
15     return false
```

Preserve Direct Influences. This new attempt follows the same steps as the attempt in the influence-guided call removal strategy: it first identifies irrelevant calls and then takes one program execution to remove them (lines 5-15 in algorithm 2). The key difference is that in this attempt, calls with indirect influence on the target call are treated as irrelevant as long as they do not have direct influence. Concretely, these irrelevant calls can be identified by $M[c_i][c_n] = 0$ (line 7) and are removed at one time to check whether the new coverage is alerted.

Example. In Figure 6, only c_3 has a direct influence on c_6, so $\{c_1, c_2, c_4, c_5\}$ are identified as irrelevant calls and are removed at one time. As a result, the reduced program P' becomes $[c_3, c_6]$ and is executed. In this example, this attempt succeeds as the new coverage achieved by c_6 remains unchanged. Thus, P is minimized to $[c_3, c_6]$.

If the attempt fails, it indicates that some of the removed calls (*e.g.*, those with indirect influences on the target call) affect the new coverage achieved by c_n. In this case, P remains unchanged, and we will perform the second attempt — which is identical to the attempt in the influence-guided call removal strategy – to preserve both the direct and indirect influences (lines 1-2). Finally, no matter whether these two attempts are successful or not, the reduced program will undergo one-by-one call removal to ensure the number of calls in this program is minimal (line 3). In the example of Figure 6, since the first attempt succeeds (reducing P to $[c_3, c_6]$), the extended strategy does not need to perform the second attempt. It only proceeds to the one-by-one call removal to verify whether the remaining call c_3 affects the execution of c_6. Thus, the extended strategy takes a total of two program executions (one for the first attempt and one for the one-by-one minimization), which further reduces one program execution by one compared to the influence-guided call removal strategy.

Our evaluation results show that the first attempt in the extended strategy has an average success rate of 41.9%, which is comparable to the 50.1% success rate of the attempt in the original influence-guided call removal

Just Accepted

Table 2. Argument type classification in Linux kernel.

Primitive type	Category	Derived type	Category	User-defined type	Category
Integer Flag Protocol Resource	Fixed-size argument	Array Buffer	Variable-size argument	Struct Union Enum	Classify each sub-field
		Pointer	Classify the pointed-to object		

strategy. This indicates that the first attempt can still maintain a similar success rate even while removing more irrelevant calls at one time, confirming that its introduction is effective and beneficial.

4.3 Type-informed Argument Simplification Strategy

We define fixed-size or variable-size arguments as follows.

*For an argument a, a is a fixed-size argument if a only involves fixed-length data types (*e.g., *Integer, Flag, Protocol and Resource); a is a variable-size argument if a involves any variable-length data type (*e.g. *Array and Buffer).*

This definition utilizes memory size to classify an argument, and the memory size of one argument can be inferred from its type. Thus, we analyze the type information to determine whether an argument is fixed-size or variable-size.

Analyze Argument Type. We detail how we analyze an argument type as follows. Given a program P after call removal, the strategy analyzes the type of each argument a from the calls in P based on Table 2 (lines 2-3 in Algorithm 3). Specifically, if a belongs to a primitive type (such as fixed-length data types as *Integer*, *Flag*, *Protocol* and *Resource*), a is classified as a fixed-size argument. If a belongs to a derived type (such as variable-length data types as *Array* and *Buffer*), a is classified as a variable-size argument. If a belongs to *Pointer*, the strategy will recursively analyze the type of the pointed-to object of a to decide a's argument type according to the preceding definition. If a belongs to a *user-defined* type, the strategy will recursively analyze the type of each sub-field of a to decide a's argument type according to the preceding definition. If any point-to-object or sub-field is a variable-length data type, a is a variable-size argument. Otherwise, a is classified as a fixed-size argument. In this way, any argument from the calls in P can be classified into either fixed-size or variable-size.

Example. Taking the program $P = [c_1, c_3]$ after call removal in Figure 6 as an example, assuming c_3 takes an argument 5 as input and returns r_1; c_6 takes r_1 and $\{3, [1, 2]\}$ as input. The strategy can identify 5 as *Integer* type and r_1 as *Resource* type. Since *Integer*, *Resource*, and *Flag* are fixed-length data types, these arguments 5 and r_1 are classified as fixed-size arguments. For the argument $\{3, [1, 2]\}$, the strategy will recursively analyze the type of each sub-field. As a result, 3 is identified as *Integer* type and $[1, 2]$ is identified as *Array* type. Since the field $[1, 2]$ involves a variable-length data type, the argument $\{3, [1, 2]\}$ is classified as a variable-size argument.

Type-informed Simplification. After obtaining the type information of each argument, in Algorithm 3, this strategy skips the simplification of fixed-size arguments (lines 6-7) and only simplifies the variable-size arguments (lines 9-10). For a variable-size argument *arg*, if the type of *arg* is *Array* or *Buffer*, the strategy will iteratively remove elements from *arg* to minimize the number of elements in the array by binary search while ensuring the coverage is always preserved (lines 16-22). When the type of *arg* is *Pointer* or *User-defined*, the strategy recursively checks the pointed-to object ($point_to_obj$ for *Pointer* type) or each sub-field (sub_field for *user-defined* type) in *arg* and adopts the same method to do simplification (lines 11-15). After traversing all the arguments, the simplified program will be returned and added to the seed corpus.

Example. According to the type analysis result of each argument in Figure 6, since 5 of *Integer* type and r_1 of *Resource* type are fixed-size arguments, the strategy will skip their simplifications. For the variable-size argument $\{3, [1, 2]\}$ of *Struct*, the strategy will skip simplifying the sub-field 3 of *Integer* type but simplify the sub-field $[1, 2]$ of *Array* type. It would consume two executions to simplify $[1, 2]$ to $[1]$ (assuming the element 1 in $[1, 2]$ can affect the execution of c_7). Finally, the simplified P becomes a seed. Overall, the strategy only takes two

Just Accepted

Algorithm 3: Type-informed Argument Simplification Strategy

```
Input  : Program after call removal: P = [c_1, ..., c_n]
Output : Final minimized program P
for i from 1 to n do
    for arg in c_i.args do
        Analyze arg's type  // determine if arg is fixed-size or variable-size based on
            Table 1
        type_informed_simplification(P, arg)
Function type_informed_simplification(&P, arg)
    // skip the simplification of fixed-size argument
    if arg is a fixed-size argument then
        continue
    else
        // simplify array/buffer for variable-size arg
        if arg.type is Array or Buffer then
            simplify_variable-size_arg(P, arg)
        // analyze the point-to object
        if arg.type is Pointer then
            type_informed_simplification(P, arg.point_to_obj)
        // recursively analyze its each sub-field
        if arg.type is User-defined type then
            for sub_field in arg.sub_fields do
                type_informed_simplification(P, sub_field)
Function simplify_variable-size_arg(&P, arg)
    P' ← P.remove_element_by_binary_search(arg)
    target_call_cov ← execute(P)
    target_call_cov' ← execute(P')
    // this argument simplification succeed
    if target_call_cov ⊆ target_call_cov' then
        P ← P'
        simplify_variable-size_arg(P, arg)
```

Just Accepted

program executions to simplify these arguments, whereas the default one-by-one minimization strategy requires five program executions. The reason for skipping the simplification of those fixed-size arguments is that such simplification does not help reduce redundant mutation space and improve the mutation efficiency.

4.4 Implementation

We implemented the influence-guided call removal and the type-informed argument simplification in Syzkaller (commit 1759857fa9bd [17]). We name the version of Syzkaller enhanced by these two optimization strategies as SyzMini. We chose Syzkaller because it is the most popular and representative kernel fuzzing tool, and is a

standard baseline used by many recent works [3, 9, 10, 23, 50, 53]. Additionally, we also implement the extended influence-guided call removal strategy on SyzMini, referring to this enhanced version as SyzMini$^+$.

Specifically, the influence-guided strategy needs the influence relations between the system calls as input. To this end, we collected the influence relations by following Healer [49]. In practice, we use both static and dynamic analysis to infer and collect the influence relations. In static analysis, given two system calls c_i and c_j, c_i has the direct influence relation on c_j if (1) c_i returns a resource type r, and at least one of the parameters in c_j is the resource type r, or (2) c_i has a pointer parameter a that points to one resource type r with an outward data flow direction, and c_j has the resource type r with an inward data flow direction. To collect static influence relations, we analyze the type information of each parameter specified by *Syzlang* [51]. In dynamic analysis, we leverage the default minimization process in Syzkaller to infer dynamic influence relations. Specifically, during minimization, if removing a call c_i affects the execution path (*i.e.*, the coverage) of the next call c_{i+1}, then c_i can directly influence c_j. We ran Syzkaller until the branch coverage saturated and collected the dynamic influence relations, which took about four days. We collected a total of 74,865 influence relations, including 44,966 static influence relations and 29,899 dynamic influence relations.

SyzMini uses the default branch coverage supported by Syzkaller to record the execution information of each call. To check whether one minimization attempt is successful, SyzMini verifies whether the branch coverage of a target call is preserved. If the coverage is preserved, this attempt is considered successful.

5 Evaluation

This section evaluates the effectiveness of the three optimization strategies. The influence-guided call removal and type-informed argument simplification strategies are implemented in the minimization stage of SyzMini, while the extended influence-guided call removal and type-informed argument simplification strategies are implemented in the minimization stage of SyzMini$^+$. We aim to answer the following research questions:

RQ1. How effective is SyzMini (which incorporates the influence-guided call removal and type-informed argument simplification strategies for minimization) in terms of code coverage and bug finding, and can SyzMini$^+$ (which is enhanced by the extended influence-guided call removal strategy) further improve the effectiveness of SyzMini?

RQ2. What are the respective contributions of the influence-guided call removal and type-informed argument simplification strategies in improving kernel fuzzing?

RQ3. Why are the influence-guided call removal and type-informed argument simplification strategies effective in improving kernel fuzzing, and why can the extended influence-guided call removal strategy further improve the effectiveness of SyzMini?

RQ4. How effective are the optimization strategies of SyzMini and SyzMini$^+$ when integrated into other distinct kernel fuzzers?

RQ5. What factors affect the effectiveness of the influence-guided call removal and type-informed argument simplification strategies, and can the different attempt orders affect the effectiveness of the extended influence-guided call removal strategy?

5.1 Evaluation Setup

All the experiments were conducted on a machine running 64-bit Ubuntu 20.04 LTS and equipped with an AMD 3995WX 64-core CPU and 128G RAM. We selected Linux 5.15, 6.1, and 6.11 as the target kernels to be tested. Specifically, Linux 5.15 and 6.1 are the stable long-term versions and are the most widely used kernel versions by many distributions. Linux 6.11 is the latest upstream version at the time of our work. All three kernel versions were compiled with the same configurations, *i.e.*, enabling the KCOV and KASAN features to collect code coverage and detect memory-related bugs. During the fuzzing campaign, each kernel runs on a virtual machine instance

Just Accepted

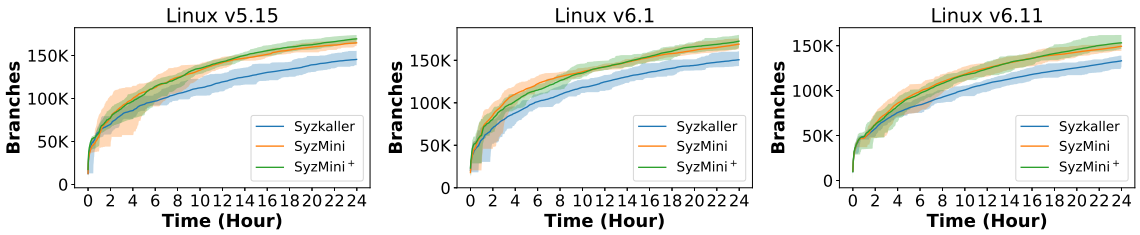


Fig. 7. Average branch coverage growth of SyzMini and SyzMini$^+$ on v5.15, v6.1, and v6.11 over 24 hours compared to those of Syzkaller (shaded area indicates the min–max range across 10 rounds).

allocated with a 4-core CPU and 4GB RAM. The influence-guided strategy is equipped with 74,865 entries of influence relations (detailed in Section 4.4). Each tool uses the same initial seeds as Syzkaller by default across all trials.

5.2 RQ1: Effectiveness of SyzMini and SyzMini$^+$

Setup of RQ1. We aim to evaluate the end-to-end effectiveness of SyzMini that incorporates the influence-guided call removal and type-informed argument simplification strategies. Thus, we compared SyzMini with Syzkaller in terms of the code coverage and the number of unique bugs found. Following the general guidelines for fuzzing evaluation [27, 46], each kernel fuzzing tool was allocated 24 hours as one round of fuzzing campaign, and we repeated such a fuzzing campaign for 10 rounds to mitigate the effects of randomness in our analyses. We also measured the *hitting-round* and *time-to-exposure* (*TTE*) for each found bug. These two metrics are widely used in evaluating fuzzing techniques [4, 8, 50]. Specifically, *hitting-round* represents the number of times a fuzzer triggers a target bug in repeated experiments. *TTE* is the time that a fuzzer triggers the target bug for the first time. We computed the arithmetic average of *TTE* (denoted by μTTE) for repeated experiments. If a fuzzer fails to trigger a target bug in a given time duration (24 hours in our setting), the *TTE* of this bug is treated as 24 hours. Additionally, to study the long-term effectiveness, we ran SyzMini and Syzkaller, respectively, for three days on the latest upstream kernel version, Linux 6.11. We collected branch coverage and the number of bugs found.

To evaluate whether the extended influence-guided call removal strategy (implemented in SyzMini$^+$) can further enhance the effectiveness of SyzMini, we ran SyzMini$^+$ for the 24-hour fuzzing campaign across 10 rounds under the same environment as SyzMini, and compared the branch coverage and unique bugs found by SyzMini and SyzMini$^+$.

Code Coverage. Figure 7 plots the average branch coverage achieved by Syzkaller and SyzMini during the 24-hour fuzzing campaigns. Overall, SyzMini outperforms Syzkaller on the three kernel versions. Specifically, in the first 5 minutes, both tools generate and execute the initial seeds, so they achieve similar coverage. Later, a large number of interesting programs that require minimization are generated; however, performing minimization on these programs does not lead to any coverage increases. Thanks to our optimization strategies, SyzMini avoids a large number of unnecessary program executions during the minimization stage, allowing it to spend more time on the mutation stage. As a result, SyzMini achieves the coverage improvement over Syzkaller after 1.5 hours, and this improvement has been maintained throughout the subsequent fuzzing.

Figure 7 also presents the average branch coverage achieved by SyzMini$^+$ during the 24-hour fuzzing campaigns. In the first 10 hours, SyzMini and SyzMini$^+$ exhibit similar branch coverage. This is because, in the early stage of fuzzing, programs that trigger new coverage are relatively simple, and there are only a small number of irrelevant calls in these interesting programs. Consequently, the difference in effectiveness between SyzMini and SyzMini$^+$ remains minimal. As fuzzing progresses, however, the programs required to achieve new coverage become

Just Accepted

Table 3. Coverage statistics between Syzkaller, SyzMini and SyzMini$^+$.

	v5.15	v6.1	v6.11
Syzkaller	145.371K	150.677K	133.172K
SyzMini	164.724K (+13.3%)	169.015K (+12.2%)	149.311K (+12.1%)
SyzMini$^+$	169.268K (+16.5%)	172.532K (+14.5%)	153.165K (+15.1%)
Speed-up (SyzMini)	+1.83×	+1.61×	+1.62×
Speed-up (SyzMini$^+$)	+1.88×	+1.63×	+1.64×
p-value (SyzMini vs. Syzkaller)	0.00009	0.00014	0.00009
p-value (SyzMini$^+$ vs. Syzkaller)	0.00009	0.00020	0.00009

[1] "+" denotes the coverage improvement achieved by the tool in that row compared to Syzkaller.

increasingly complex (e.g., involving more calls), which also increases the likelihood of containing irrelevant calls. The first attempt in SyzMini$^+$ preserves only the direct influences (omitting the indirect influences), enabling it to remove more irrelevant calls at one time and reducing the number of program executions needed during the subsequent one-by-one minimization. As a result, SyzMini$^+$ can allocate more fuzzing resources to mutation and achieves higher average coverage than SyzMini across all three kernel versions during the later phase of fuzzing. However, the substantial overlap between their coverage ranges indicates run-to-run variability, so this result reflects an average improvement rather than higher coverage in every campaign.

Table 3 gives the detailed coverage statistics. For example, on v5.15, SyzMini covered an average of 164.724K branches in the 10 rounds of a 24-hour fuzzing campaign, while Syzkaller covered an average of 145.371K branches. As a result, SyzMini achieved a 13.3% improvement in branch coverage. Row "speed-up" presents the average speed-up achieved by the tool in that row when achieving the same coverage as Syzkaller. On average, SyzMini achieved 12.5% improvement in branch coverage compared to Syzkaller with a speed-up of 1.69×. The results indicate that the influence-guided call removal and type-informed argument simplification strategies can significantly improve the effectiveness of kernel fuzzing in terms of code coverage. We further perform the Mann–Whitney U-test to assess the statistical significance of the coverage improvements. In Table 3, the p-values for both SyzMini and SyzMini$^+$ compared to Syzkaller are all below 0.05 across the three kernel versions (e.g., 0.00009, 0.00014, 0.00009 for SyzMini, and 0.00009, 0.00020, 0.00009 for SyzMini$^+$), indicating that both approaches significantly outperform Syzkaller.

Additionally, SyzMini$^+$ achieved higher coverage improvements compared to SyzMini. Taking Linux v5.15 as an example, SyzMini$^+$ achieved a 16.5% coverage improvement over Syzkaller, 3.2% higher than that achieved by SyzMini (13.3%). Regarding speed-up, SyzMini$^+$ also reached the same code coverage as Syzkaller faster than SyzMini. These results demonstrate that SyzMini$^+$ further enhances the code coverage compared to SyzMini.

Bug Detection. Table 4 lists the detailed bug statistics. For example, on v5.15, SyzMini found a total of 28 unique bugs in the 10 rounds of 24-hour fuzzing, while Syzkaller found a total of 14 unique bugs. SyzMini found 2.0× more unique bugs than Syzkaller. Specifically, on v5.15, SyzMini found 18 unique bugs that Syzkaller missed, while Syzkaller only found 4 bugs that SyzMini missed. Column "Total" counts the total number of unique bugs found in the three kernel versions. SyzMini found a total of 50 unique bugs, while Syzkaller found only 27 unique bugs. Specifically, SyzMini and Syzkaller found a total of 22 common bugs on the three kernel versions (accumulating the common bugs shown in the three Venn diagrams of v5.15, v6.1, and v6.11 in Column "Bug Distribution").

For SyzMini$^+$, it achieved the highest number of unique bugs among the three tools. Specifically, on v5.15, SyzMini$^+$ discovered 32 unique bugs, which is 2.3× that of Syzkaller and 4 more than SyzMini. Across the three kernel versions, SyzMini$^+$ identified a total of 21 unique bugs that Syzkaller missed, three more than the 18

Just Accepted

Table 4. Bug statistics between Syzkaller, SyzMini and SyzMini$^+$.

Tool	Number of Unique Bugs				Bug Distribution			
	v5.15	v6.1	v6.11	Total	v5.15	v6.1	v6.11	Total
Syzkaller	14	12	6	27				
SyzMini	28 (2.0×)	20 (1.7×)	12 (2.0×)	50 (1.9×)				
SyzMini$^+$	32 (2.3×)	23 (1.9×)	13 (2.2×)	55 (2.0×)				

[1] The blue, green, and yellow regions in the Venn diagrams denote #bugs found by SyzMini, SyzMini$^+$, and Syzkaller, respectively
[2] The common regions denote the number of unique bugs found by both tools.

unique bugs missed by SyzMini. These results demonstrate that the extended influence-guided strategy enables SyzMini$^+$ to achieve higher effectiveness in bug detection.

Table 5. *µTTE* and *hitting-round* of each bug.

ID	*#hitting-round* Syzkaller/SyzMini	*µTTE* (hour) Syzkaller/SyzMini	ID	*#hitting-round* Syzkaller/SyzMini	*µTTE* (hour) Syzkaller/SyzMini
1	3 / 6	22.2 / 12.2	12	3 / 5	16.8 / 14.0
2	1 / 2	23.9 / 19.7	13	7 / 10	12.5 / 3.5
3	4 / 8	15.8 / 10.8	14	1 / 2	23.5 / 20.9
4	1 / 3	23.1 / 20.9	15	2 / 4	19.4 / 15.9
5	5 / 5	17.0 / 13.8	16	1 / 4	22.3 / 18.9
6	7 / 9	19.4 / 15.8	17	2 / 8	22.4 / 13.1
7	3 / 4	22.8 / 19.4	18	1 / 6	23.4 / 13.3
8	3 / 3	21.7 / 20.7	19	1 / 3	22.3 / 19.5
9	1 / 2	22.3 / 20.1	20	10 / 10	4.2 / 2.0
10	1 / 2	22.0 / 20.8	21	1 / 3	23.9 / 19.4
11	9 / 9	11.9 / 7.5	22	8 / 10	5.2 / 2.8

[1] ID 1~10, 11~18 and 19~22 are the common bugs on v5.15, v6.1 and v6.11, respectively.
[2] The green one indicates which tool achieves the higher *hitting-round* or the lower *µTTE*.

Table 5 gives the *hitting-round* and *µTTE* for each common bug found by Syzkaller and SyzMini. On any of these bugs, SyzMini found them faster and more frequently than Syzkaller. Taking the bug with ID 1 as an example, the *µTTE* of Syzkaller is 22.2h, which is 1.8× more than that of SyzMini (12.2h); moreover, SyzMini successfully found this bug in 6 out of 10 rounds of fuzzing, while Syzkaller only found it in 3 rounds. These strong results show that SyzMini not only accelerates bug detection but also improves the reliability of detecting hard-to-trigger bugs.

Long-term Effectiveness. In the 10 rounds of a three-day fuzzing campaign, SyzMini covered an average of 199.4K branches, which is 5.6% more than that of Syzkaller (188.9K). In terms of bug finding, although Syzkaller has been continuously deployed to find bugs in the upstream kernel version, SyzMini has still found 13 previously unknown bugs. We reported each bug (with the corresponding reproducer and kernel configuration) to the kernel maintainers. All these 13 bugs have been confirmed, and 4 have already been fixed. Table 6 lists these bugs. Therefore, SyzMini is also effective in long-term fuzzing for improving coverage and finding bugs.

Just Accepted

Table 6. Previously unknown bugs found by SyzMini.

ID	Subsystem	Function	Risk	Status
1	Block	bio_put_slab	use after free	confirmed
2	Gfs2	gfs2_quota_init	kernel bug	confirmed
3	NTFS3	end_buffer_read_sync	out of bounds	confirmed
4	Reiserfs	update_stat_data	kernel bug	confirmed
5	Net	netlink_unicast	general protection fault	confirmed
6	MM/pat	get_pat_info	inconsistent lock state	fixed
7	Net	cleanup_net	use after free	fixed
8	Bpf	btf_name_valid_section	out of bounds	confirmed
9	Jfs	jfs_syncpt	null-ptr-deref	fixed
10	Usb	usb_disable_device	refcount bug	confirmed
11	Bpf	build_id_parse_nofault	paging fault	fixed
12	Bluetooth	sco_sock_close	use after free	confirmed
13	Jfs	jfs_statfs.cold	out of bounds	confirmed

In summary, the influence-guided call removal and type-informed argument simplification strategies not only improve branch coverage by 12.1~13.3% and find 1.7~2.0× more unique bugs, but also find kernel bugs faster and more stably. Additionally, the extended influence-guided call removal strategy further improves coverage by 2.3~3.2% and increases the number of unique crashes by 0.2~0.3×.

5.3 RQ2: Contributions of Optimizations

Setup of RQ2. To investigate the respective contributions of the two optimization strategies in improving kernel fuzzing, we implemented two versions of SyzMini: SyzMini$^{\alpha}$ and SyzMini$^{\beta}$. SyzMini$^{\alpha}$ only includes the influence-guided call removal strategy, while SyzMini$^{\beta}$ only includes the type-informed argument simplification strategy. We compare the achieved code coverage and the number of found unique bugs between Syzkaller, SyzMini$^{\alpha}$, SyzMini$^{\beta}$, and SyzMini to analyze the contributions of the two optimization strategies. Following the setup of RQ1, we ran each tool for 24 hours and repeated 10 rounds.

Results. Figure 8 shows the number of covered branches and the number of found unique bugs by Syzkaller, SyzMini$^{\alpha}$, SyzMini$^{\beta}$, and SyzMini. For example, on v5.15, equipped with the influence-guided call removal strategy, SyzMini$^{\alpha}$ achieved an average of 4.0% branch coverage improvement (from 145K to 151K) and found 5 more unique bugs (from 14 to 19), compared to Syzkaller. By comparing SyzMini$^{\beta}$ and SyzMini, without this influence-guided call removal strategy, SyzMini$^{\beta}$'s branch coverage dropped by 3.7% (from 165K to 159K) and missed 9 unique bugs (from 28 to 19). Thus, the influence-guided call removal strategy is useful in improving kernel fuzzing.

For the type-informed argument simplification strategy, on v5.15, SyzMini$^{\beta}$ achieved an average of 9.1% more branch coverage (from 145K to 159K) and found 5 more unique bugs, compared to Syzkaller. By comparing SyzMini$^{\alpha}$ and SyzMini, without the type-informed argument simplification strategy, the branch coverage achieved by SyzMini$^{\alpha}$ dropped by 8.2% (from 165K to 151K) and missed 9 unique bugs. These results indicate that the type-informed argument simplification strategy is also important in improving kernel fuzzing. We also note that SyzMini$^{\beta}$ achieved more improvement than SyzMini$^{\alpha}$. We will explain this phenomenon in Section 5.4.

It is worth noting that the optimization effects of the influence-guided call removal and type-informed argument simplification strategies are orthogonal. These two strategies act on two different steps (*i.e.*, removing calls and

Just Accepted

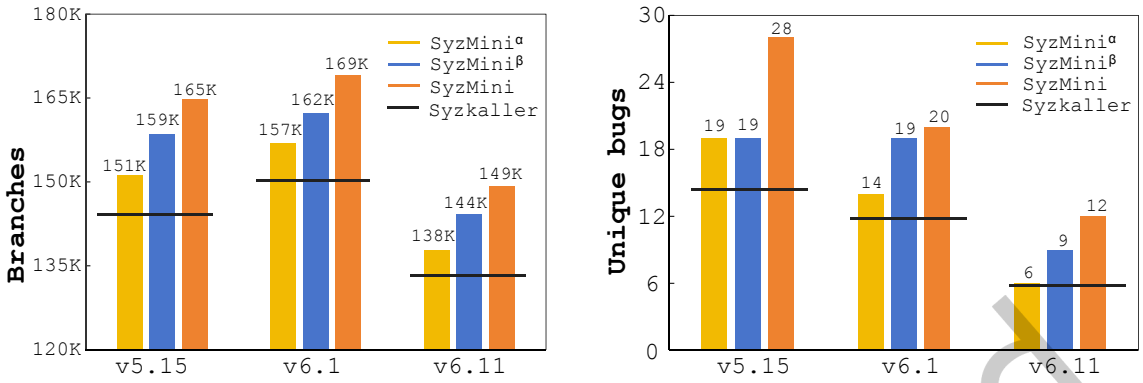


Fig. 8. Numbers of covered branches and found unique bugs by Syzkaller, SyzMini$^{\alpha}$, SyzMini$^{\beta}$ and SyzMini.

simplifying arguments) in the minimization stage. Thus, SyzMini achieved the highest coverage and found the largest number of bugs compared to Syzkaller, SyzMini$^{\alpha}$, and SyzMini$^{\beta}$.

In summary, the influence-guided call removal strategy improves the coverage by 3.5~4.6% and increases the number of found unique bugs by 1~1.35×, while the type-informed argument simplification strategy enhances coverage by 8.3~9.1% and increases the number of found unique bugs by 1.2~1.5×. The optimization effects of these two strategies are orthogonal.

5.4 RQ3: Reasons of Effectiveness

Setup of RQ3. We designed a controlled experiment to explain why the influence-guided call removal and type-informed argument simplification strategies can improve the effectiveness of kernel fuzzing by reducing the cost of the minimization stage. To this end, we ran Syzkaller for 24 hours and found 16,266 interesting programs that need minimization. We applied the default one-by-one minimization strategy in Syzkaller, the influence-guided call removal strategy, and the type-informed argument simplification strategy in SyzMini, respectively, to minimize these interesting programs. We counted the number of program executions needed for verifying the coverage to approximate the cost required by these different minimization strategies. We also counted the number of program executions needed for the extended influence-guided call removal strategy in SyzMini^{+} to explain why the extended strategy can further enhance SyzMini's effectiveness. Additionally, we measured the portions of program executions between the mutation stage and minimization stage for Syzkaller, SyzMini, and SyzMini^{+} during 24-hour fuzzing campaigns, aiming to analyze the cost of the mutation and minimization stage before and after applying the two optimization strategies.

Each successful attempt reduces the number of program executions required for the minimization stage, so the attempt success rate plays an important role in the overall effectiveness of the fuzzer. SyzMini contains only one attempt (*preserve direct&indirect influences*), whereas SyzMini^{+} is composed of two attempts (*preserve direct influences* and *preserve direct&indirect influences*). To this end, we count the average success rate of these three attempts during the 10 rounds of 24-hour fuzzing campaigns, aiming to explain why SyzMini^{+} with the extended influence-guided call removal strategy outperforms SyzMini.

Additionally, as shown in Figure 5, the program length of an interesting program (i.e., the number of system calls it contains) may vary during the call removal. Specifically, such variations manifest across three stages: *before minimization*, *after attempt removal* (where SyzMini and SyzMini^{+} have one and two attempts, respectively), and *after one-by-one call removal*. To further understand why SyzMini^{+} achieves higher effectiveness, we took the 16,266 interesting programs used in the previous controlled experiment, minimized them using both the original

Just Accepted

Table 7. Number of program executions under different minimization strategies in removing calls and simplifying arguments.

Minimization Strategy	Number of Program Executions		
	Removing Calls	Simplifying Arguments	Overall
one-by-one	140,620	260,239	400,859
influence-guided call removal	60,372 (57.1%↓)	–	320,611 (20.0%↓)
type-informed argument simplification	–	96,896 (62.8%↓)	237,516 (40.7%↓)
extended influence-guided call removal	35,983 (74.4%↓)	–	296,222 (26.1%↓)

influence-guided strategy (SyzMini) and the extended strategy (SyzMini$^+$), and measured the program lengths at each stage.

Results. Table 7 gives the number of program executions under the different minimization strategies. Compared to the one-by-one minimization strategy, the influence-guided call removal strategy reduced 57.1% of program executions when removing calls (see Column "Removing Calls"), while the type-informed argument simplification strategy reduced 62.8% of program executions when simplifying arguments (see Column "Simplifying Args"). Overall, the influence-guided call removal and type-informed argument simplification strategies respectively avoided 20.0% and 40.7% of the total program executions required by the one-by-one minimization strategy (see Column "Overall"). Moreover, when both strategies are enabled, they can reduce a total of 60.7% of program executions. These results show that the two optimization strategies can avoid a large number of program executions during the minimization stage. As a result, SyzMini can allocate more time to the mutation stage and execute more diverse programs. It explains why SyzMini can achieve higher coverage and find more unique bugs than Syzkaller.

Additionally, the extended influence-guided call removal strategy in SyzMini$^+$ further improves the efficiency of the minimization stage. It reduces 74.4% of the program executions required for the call removal—an additional 17.3% improvement over the influence-guided call removal strategy used in SyzMini. This greater reduction indicates that SyzMini$^+$ avoids substantially more unnecessary executions during minimization, which contributes to its overall higher effectiveness compared to SyzMini.

It is also worth noting that the type-informed argument simplification strategy reduces 40.7% of program executions, which is about 2X more than that of the influence-guided call removal strategy (20.0%). The former strategy can reduce the cost of minimization more than the latter. This result explains why SyzMini$^\beta$ can achieve more improvement than SyzMini$^\alpha$ for kernel fuzzing in RQ2.

Figure 9 shows the proportion of program executions between the minimization and mutation stages in a 24-hour fuzzing campaign *before* and *after* applying the influence-guided and type-informed optimization strategies on the three kernel versions (*i.e.*, comparing between Syzkaller, SyzMini, and SyzMini$^+$). Compared to Syzkaller, SyzMini and SyzMini$^+$ significantly reduce the proportion of program executions in the minimization stage, while substantially increasing the proportion of program executions in the mutation stage. Taking v5.15 as an example, Syzkaller's minimization occupies 48.2% of program executions, while SyzMini's minimization only occupies 11.5% (reduced by 75.7%), and SyzMini$^+$ further compresses this overhead to a mere 9.8%. On the other hand, Syzkaller's mutation occupies 51.8% of program executions, while SyzMini's mutation occupies 88.5% (increased by 70.5%), with SyzMini$^+$ maximizing the mutation portion to an impressive 90.2%. This consistent trend of SyzMini$^+$ achieving the lowest minimization ratios is also evident in v6.1 (11.4%) and v6.11 (13.6%). The results show that both SyzMini and SyzMini$^+$ successfully reduce the cost of the minimization stage during fuzzing, ensuring that the vast majority of execution resources are allocated to the mutation stage.

Table 8 reports the attempt success ratios for SyzMini and SyzMini$^+$ across three Linux kernel versions. Across all three versions, the single-attempt success ratio of SyzMini is slightly higher than the combined success ratio of the two attempts in SyzMini$^+$ (52.1% *vs.* 50.8% on v5.15, 50.4% *vs.* 48.6% on v6.1, and 47.7% *vs.* 45.5% on v6.11).

Just Accepted

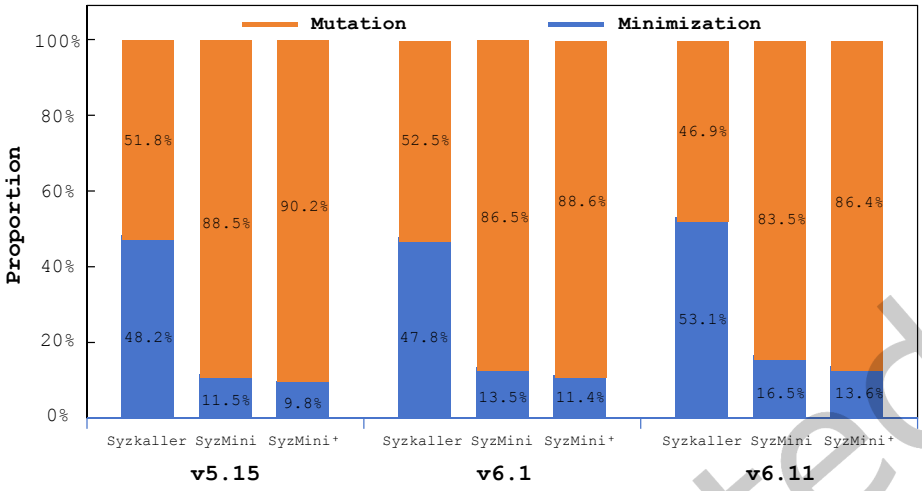


Fig. 9. Proportion of program executions between mutation and minimization in Syzkaller, SyzMini, and SyzMini+.

Table 8. The average attempt success ratio during call removal for SyzMini and SyzMini+.

Version	SyzMini	SyzMini+		
	Preserve Direct&Indirect Influences	*Preserve Direct Influences*	*Preserve Direct&Indirect Influences*	*Both Two Attempts*
v5.15	52.1%	43.1%	7.7%	50.8%
v6.1	50.4%	42.2%	6.4%	48.6%
v6.11	47.7%	40.4%	5.1%	45.5%

[1] The column "*Preserve Direct&Indirect Influences*" denotes the attempt success ratio when preserving both the direct and indirect influences.
[2] The column "*Preserve Direct Influences*" represents the attempt success ratio when preserving only the direct influences.
[3] The column "*Both Two Attempts*" indicates the combined success rate of two attempts in SyzMini+.

Table 9. The average program length across different stages: SyzMini vs. SyzMini+.

Tool	The Average Program Length at Different Stages		
	Before Minimization	*After Attempt Removal*	*After One-by-one Call Removal*
SyzMini	7.3	4.3	2.0
SyzMini+	7.3	3.1	2.0

Nevertheless, previous results show that SyzMini+ achieves better coverage and bug-detection performance than SyzMini. This is because the first attempt in SyzMini+ is more aggressive than the attempt used in SyzMini, while still maintaining a comparable success ratio (43.1% *vs.* 52.1% on v5.15, 42.2% *vs.* 50.4% on v6.1, and 40.4% *vs.* 47.7% on v6.11). As a result, SyzMini+ can remove more irrelevant calls in the first attempt than SyzMini, thereby reducing the number of calls that need to be validated during subsequent one-by-one minimization. Consequently, a higher success ratio of this aggressive attempt allows SyzMini+ to save more program executions and improve overall fuzzing efficiency. Additionally, with more than half of the attempts being successful, these results confirm the effectiveness of the influence-guided call removal strategies.

Table 9 reports the average program length for 16,266 interesting programs at different stages when applying the original influence-guided strategy (SyzMini) and the extended strategy (SyzMini+). Specifically, the average

Just Accepted

program length before minimization is 7.3, and the final program length in both the extended and original influence-guided strategies (after the one-by-one call removal) is 2.0. It means the two strategies eventually achieve the same goal of minimizing the program length. However, after the attempt removal stage, programs minimized by the extended strategy are substantially shorter (3.1) than those minimized by the original strategy (4.3). This is because the first attempt in SyzMini$^+$ also identifies the calls with indirect influence as irrelevant and removes them at one time. Meanwhile, the attempt success rates for SyzMini and SyzMini$^+$ are similar (listed in Table 8). As a result, SyzMini$^+$ shortens the program length even more — using only one program execution for the attempt removal — and therefore requires fewer program executions during the subsequent one-by-one call removal. Consequently, the extended strategy enables SyzMini$^+$ to achieve higher minimization efficiency than SyzMini.

In summary, the influence-guided call removal and type-informed argument simplification optimization strategies reduce 60.7% of programs executed during the minimization stage, and more time is allocated to the mutation stage, thereby enhancing the effectiveness of kernel fuzzing. Additionally, the extended influence-guided strategy can further enhance the original influence-guided strategy by removing more irrelevant calls at one time in the first attempt.

5.5 RQ4: Applicability of Optimizations

Setup of RQ4. We aim to evaluate the generability and applicability of our optimization strategies on other distinct kernel fuzzers. To this end, we surveyed research work on kernel fuzzing that was published in the past five years (2019-2024) in relevant top venues (S&P, USENIX, CCS, NDSS, OSDI, SOSP, Eurosys, ATC). We have collected 11 papers. We excluded 3 papers that did not open-source their tools, and 5 papers whose tools had not been actively maintained anymore. We finally obtained SyzVegas [53], CountDown [3], and SyzDirect [50] as the target fuzzers, all of which are built on Syzkaller but are designed for different purposes.

- **SyzVegas**. SyzVegas aims to improve the effectiveness of kernel fuzzing by applying the reinforcement learning algorithm to three of the most critical decision tasks in Syzkaller: seed selection, seed mutation, and program generation. Specifically, SyzVegas leverages the multi-armed bandit (MAB) algorithms along with a novel reward assessment model to dynamically schedule these three tasks. Each decision of SyzVegas can select the task most likely to improve coverage and find bugs, thereby avoiding ineffective fuzzing.
- **CountDown**. CountDown is a refcount-guided (reference count-guided) kernel fuzzer designed to detect memory-related kernel bugs. It introduces a mutation operator that prioritizes combining syscalls that have previously accessed the same refcounts, thereby increasing the likelihood of triggering complex refcount issues and memory-related erroneous behaviors.
- **SyzDirect**. SyzDirect is a direct kernel fuzzing technique that targets reproducing kernel bugs. Through a novel and scalable static analysis of the Linux kernel, SyzDirect identifies valuable information, such as valid system calls and conditions on their arguments, to guide the generation and mutation of test cases. This approach enables it to reach the target locations more efficiently.

These three tools represent state-of-the-art approaches for different application scenarios. We integrate the optimization strategies of SyzMini (which incorporate influence-guided call removal and type-informed argument simplification) and SyzMini$^+$ (which further extend the influence-guided call removal strategy) into SyzVegas, CountDown, and SyzDirect. This results in two enhanced versions for each tool, denoted as SyzVegas$^+$/SyzVegas^{++}, CountDown$^+$/CountDown^{++}, and SyzDirect$^+$/SyzDirect^{++}, respectively, to investigate whether their performance can be further improved. To ensure fairness, we followed the default setups and evaluation metrics of these tools described in their papers. To compare SyzVegas, SyzVegas$^+$, and SyzVegas^{++}, we measured code coverage and the number of unique bugs found. To compare CountDown, CountDown$^+$, and CountDown^{++}, we counted the number of found KASAN bugs (memory-related errors reported by Kernel Address Sanitizer). To compare SyzDirect, SyzDirect$^+$, and SyzDirect^{++}, we used μTTE and *hitting-round* to evaluate the ability of directed fuzzing.

Just Accepted

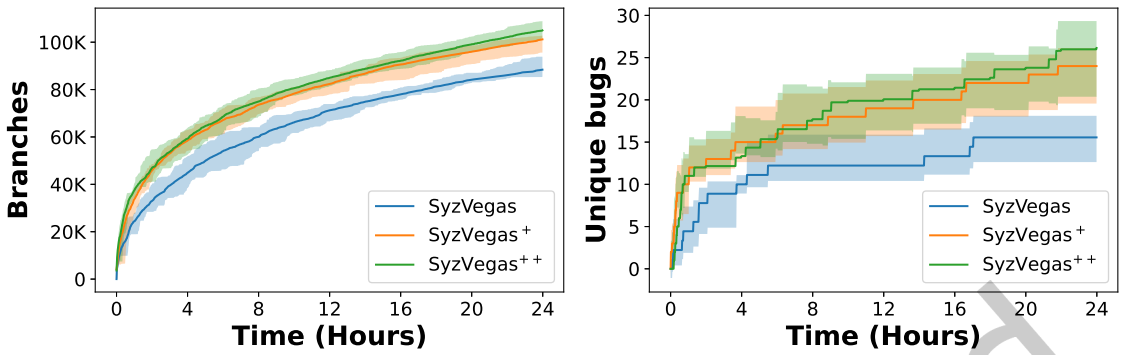


Fig. 10. SyzVegas *vs.* SyzVegas^{+} *vs.* SyzVegas^{++} in terms of branch coverage and number of unique bugs (shaded area indicates the min–max range across 10 rounds).

Specifically, SyzDirect provides a list of 100 kernel bugs as its evaluation dataset. Considering the experimental cost, we selected the first 10 bugs as the evaluation dataset in this paper. To prepare the dataset, we compiled the corresponding 10 Linux kernel versions using the configuration recommended by SyzDirect. We ran each tool for 24 hours and repeated each experiment 10 times to mitigate the effects of randomness in our analyses. Thus, this evaluation for the three kernel fuzzing tools and their respective enhanced versions took 8,640 machine hours.

SyzVegas vs. SyzVegas^{+} vs. SyzVegas^{++}. Figure 10 shows the achieved code coverage and the number of found unique bugs on Linux v5.15. With the help of the influence-guided call removal and type-informed argument simplification optimization strategies, SyzVegas^{+} improves branch coverage by 14.5%. SyzVegas^{+} found 24 unique bugs, while SyzVegas only found 16 unique bugs. Further enhanced with the extended influence-guided call removal strategy, SyzVegas^{++} achieves an additional 3.7% increase in coverage and finds 2 more unique bugs compared to SyzVegas^{+}. These results demonstrate that our optimization strategies improve SyzVegas in terms of both code coverage and bug-finding capability.

CountDown vs. CountDown^{+} vs. CountDown^{++}. Table 10 lists the number of KASAN bugs found by CountDown and its enhanced variants on Linux v5.11. During the 10 rounds of a 24-hour fuzzing campaign, CountDown^{+} found a total of 43 KASAN bugs, while CountDown only found 27 bugs. CountDown^{+} also achieved 4.5% higher branch coverage. Further enhanced with the extended influence-guided call removal strategy, CountDown^{++} found 47 KASAN bugs, representing an additional 9.3% improvement over CountDown^{+}, and achieved further gains in branch coverage (126.8k vs. 124.3k on average). CountDown leverages memory relations between system calls to guide seed mutations for discovering memory-related bugs. However, all seeds in the corpus require minimization to improve subsequent mutation effectiveness. Our optimization strategies reduce the cost of minimization, thereby allowing CountDown more time to perform memory-related mutations for finding KASAN bugs.

SyzDirect *vs.* SyzDirect^{+} *vs.* SyzDirect^{++}. Table 11 gives the μTTE and *hitting-round* for reproducing each target bug. SyzDirect^{+} successfully reproduced 9 bugs, while SyzDirect reproduced 6 bugs. The 6 bugs reproduced by SyzDirect were all reproduced by SyzDirect^{+}. Moreover, for these 6 bugs, SyzDirect^{+} achieved higher *hitting-round*s and lower μTTE compared to SyzDirect. SyzDirect^{++} also reproduces 9 bugs, but further reduces the reproduction time, achieving lower μTTE on 6 bugs compared to SyzDirect^{+}. The results show that our optimization strategies can enhance the directed fuzzing ability of SyzDirect. Given a bug location, SyzDirect prioritizes the mutations of seeds that are close to the bug location. But it still requires minimization to improve

Just Accepted

Table 10. CountDown *vs.* CountDown$^+$ *vs.* CountDown^{++} on the KASAN bugs.

Tool	All KASAN Bugs			Branch Coverage		
	Min	Max	Total	Min	Max	Average
CountDown	0	5	27	116.9k	120.7k	119.0k
CountDown$^+$	1	7	43	119.7k	130.1k	124.3k
Improvement (CountDown *vs.* CountDown$^+$)	1↑	2↑	66.7%↑	2.4%↑	7.8%↑	4.5%↑
CountDown^{++}	1	9	47	120.3k	132.4k	126.8k
Improvement (CountDown$^+$ *vs.* CountDown^{++})	0↑	2↑	9.3%↑	0.5%↑	1.7%↑	2.0%↑

Table 11. SyzDirect *vs.* SyzDirect$^+$ *vs.* SyzDirect^{++} on bug reproduction.

Bug ID	Commit ID	SyzDirect	SyzDirect$^+$	SyzDirect^{++}
		µTTE(hour)~#hitting-round	*µTTE(hour)~#hitting-round*	*µTTE(hour)~#hitting-round*
1	3f2db250099f	×	16.0h~4/10	15.2h~4/10
2	c9a2f90f4d6b	21.6h~2/10	20.0h~4/10	21.1h~3/10
3	b1a811633f73	×	21.2h~1/10	19.9h~2/10
4	2fd10bcf0310	21.2h~2/10	10.1h~8/10	12.3h~7/10
5	b648eba4c69e	×	×	×
6	20aaef52eb08	13.6h~10/10	12.8h~10/10	10.1h~10/10
7	3b0c40612471	18.8h~4/10	17.6h~6/10	19.3h~5/10
8	ffb324e6f874	×	22.1h~1/10	21.0h~2/10
9	01faae5193d6	18.7h~4/10	14.4h~6/10	13.8h~6/10
10	b2a616676839	20.4h~6/10	14.1h~8/10	10.2h~8/10

the quality of seeds. Our optimization strategies save the cost of the minimization stage and give SyzDirect more time to mutate to increase the probability of reproducing bugs.

In summary, the influence-guided call removal and type-informed argument simplification strategies are orthogonal to the optimization mechanisms of existing kernel fuzzers. They improve SyzVegas by achieving 14.5% higher coverage and discovering 1.5× more unique bugs, enabling CountDown to identify 66.7% more KASAN bugs, and enhancing SyzDirect by reproducing 1.5× more kernel bugs. Furthermore, incorporating the extended influence-guided call removal strategy further improves the effectiveness of SyzVegas, CountDown, and SyzDirect.

5.6 RQ5: Factors affecting Optimizations

Factor Affecting Influence-guided Call Removal Strategy. We aim to analyze the factors that could affect the effectiveness of our optimization strategies. Specifically, the influence-guided call removal optimization strategy leverages the influence relations between system calls to reduce the cost of the minimization stage. Thus, the number of influence relations is a critical factor affecting the strategy's effectiveness. To this end, we conducted a controlled experiment to study the impact of different numbers of influence relations on the effectiveness of this strategy. In our experiment, we have 74,865 influence relations in total. We selected the first 10%, 20%, . . ., 100% respectively from the list of these 74,865 influence relations as the input of the influence-guided call removal strategy and observed the number of program executions during minimization.

Results. Figure 11 shows (1) the number of program executions used for removing calls in the minimization stage, and (2) the reduced proportions of program executions compared to the one-by-one minimization strategy (calculated by a similar method in Table 7) under different proportions of influence relations. We can see that when the proportion of influence relations increases, the number of program executions used for removing calls

Just Accepted

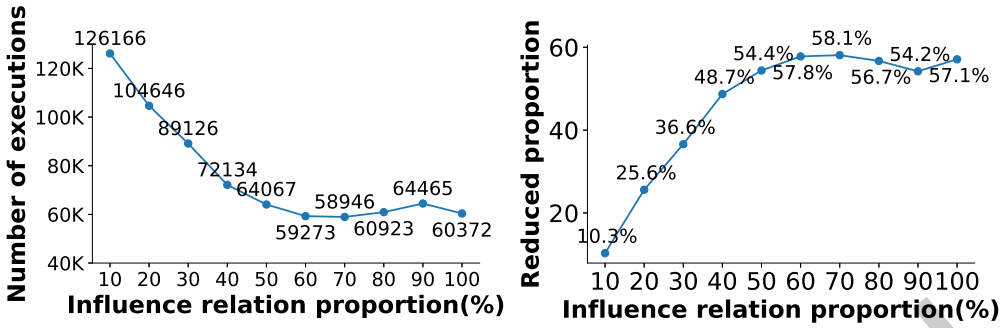


Fig. 11. Number of program executions used for removing calls in minimization, and the reduced proportion of program executions compared to the one-by-one minimization.

Table 12. Number of fixed-size and variable-size parameters in the system calls provided by Syzkaller.

Date	Commit ID	Fixed-size Parameter	Variable-size Parameter
2020-12	79264ae39c1e	101,897 (80.5%)	24,679 (19.5%)
2021-12	36bd2e486525	103,442 (80.1%)	25,643 (19.9%)
2022-12	ab32d50881df	107,575 (80.2%)	26,557 (19.8%)
2023-12	fb427a078200	112,523 (80.6%)	27,145 (19.4%)
2024-12	d7f584ee3c24	285,517 (74.0%)	100,631(26.0%)

gradually decreases (see the left subfigure), and the reduced portion of program executions gradually increases (see the right subfigure). When the proportion of influence relations reaches 60%, the number of program executions and the reduced portion of program executions gradually stabilize. The reason is that, when the number of influence relations increases, the number of irrelevant calls identified by the influence-guided strategy in the program to be minimized gradually saturates. It also indicates that the number of collected influence relations in our experiment is enough for the strategy to take effect.

It is worth noting that this evaluation has also implied the relationship between such precision and optimization effectiveness. The reason is that the proportion adopted essentially approximates the precision of the influence relationship. When a smaller proportion of influence relations is given, the type-informed strategy will erroneously identify more irrelevant calls in the interesting program, reducing the precision of inferring irrelevant calls. The final result in Figure 11 shows that when the proportion of influence relations increases, the optimization effect increases; however, when the proportion reaches beyond 60%, its effect on the optimization strategy is negligible.

Factor Affecting Type-informed Argument Simplification Strategy. The type-informed argument simplification strategy reduces the cost of the minimization stage by skipping the simplification of the fixed-size arguments. Thus, the number of fixed-size parameters (including the pointed-to objects if the parameters are *Pointer* type and the sub-fields if the parameters are *user-defined* type) in the system calls affects this strategy's effectiveness. However, obtaining all the system calls in the Linux kernel is difficult. We note that Syzkaller regularly updates the tested system calls when the Linux kernel has new updates. To this end, we analyzed the tested system calls provided by Syzkaller in the recent five years (2019-2024). We developed a script to scan all the tested system calls based on the *Syzlang* syntax [51], and counted the number of fixed-size and variable-size parameters according to the analysis method in Section 4.3.

Just Accepted

Results. Table 12 shows the number of fixed-size and variable-size parameters in the system calls provided by Syzkaller. Indeed, we can see that the number of fixed-size parameters accounts for a large portion. Taking the version of Syzkaller in Dec. 2020 as an example, the proportion of fixed-size parameters is 80.5%, while the proportion of variable-size parameters is only 19.5%. The results show that the tested system calls in Linux kernels are composed of a large number of fixed-size parameters. Thus, the type-informed argument simplification strategy can avoid simplifying many unnecessary arguments that cannot benefit from reducing the redundant mutation space. As a result, the number of program executions needed for verifying the coverage could be significantly reduced.

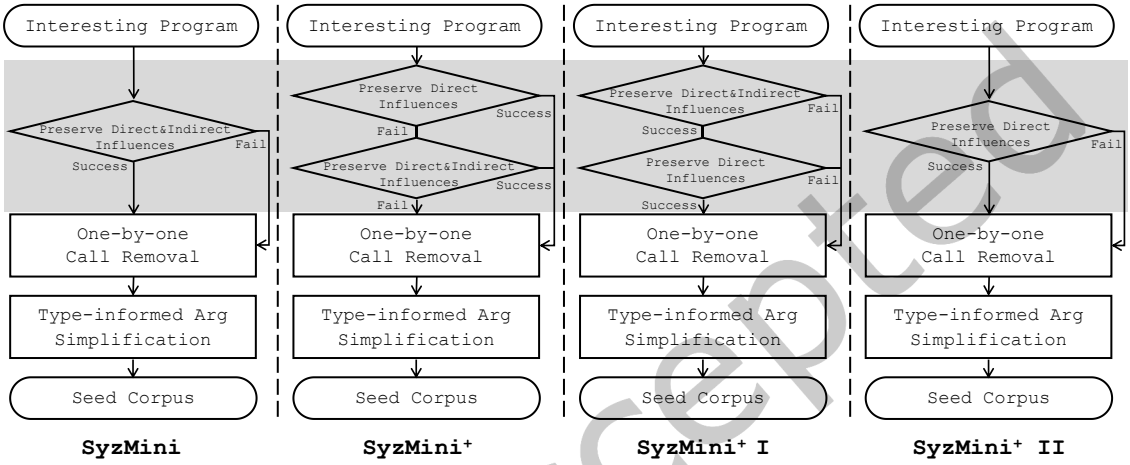


Fig. 12. The overall workflow of different attempt orders on the extended influence-guided call removal strategy.

Factor Affecting Extended Influence-guided Call Removal Strategy. The extended influence-guided call removal strategy in SyzMini+ consists of two attempts: *preserve direct influences* and *preserve direct&indirect influences*. The attempt order may affect the effectiveness of SyzMini+. To investigate this, we enumerate all four possible attempt orders, two of which correspond to SyzMini and SyzMini+. Figure 12 illustrates the overall workflow of the four configurations under different attempt orders. We further implement the remaining two attempt orders, referred to as SyzMini+ I and SyzMini+ II, to evaluate how attempt order impacts the extended influence-guided call removal strategy.

- SyzMini+ I: SyzMini+ I also consists of two attempts. Different from SyzMini+, the first attempt of SyzMini+ I preserves both direct and indirect influences. If this attempt succeeds, SyzMini+ I will perform the second attempt that preserves only the direct influences by further removing the calls that have indirect influences on the target call. Finally, regardless of whether these two attempts succeed, the reduced program will undergo one-by-one call removal and type-informed argument simplification to ensure the number of calls and arguments in the final program is minimal.
- SyzMini+ II: SyzMini+ II only contains one attempt that preserves only the direct influences. In this attempt, all the calls with indirect or no influence on the target call will be removed at one time to verify whether the target call's new coverage changes. Later, the reduced program will also undergo one-by-one call removal and type-informed argument simplification.

For each tool, we ran a 24-hour fuzzing campaign and repeated it 10 times to mitigate the effects of randomness in our analyses. We also measured the achieved branch coverage to observe which attempt order is most effective.

Just Accepted

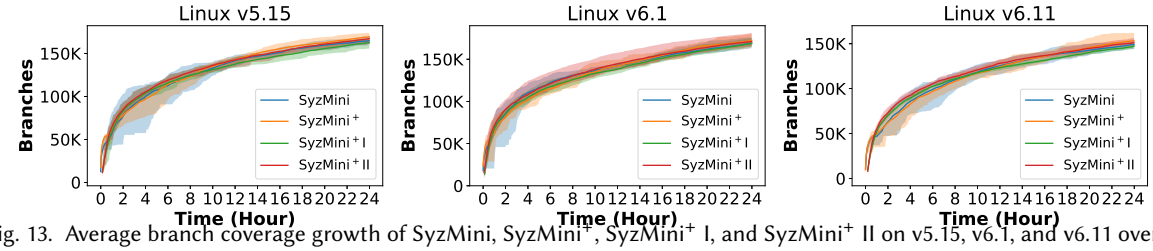


Fig. 13. Average branch coverage growth of SyzMini, SyzMini$^+$, SyzMini$^+$ I, and SyzMini$^+$ II on v5.15, v6.1, and v6.11 over 24 hours (shaded area indicates the min–max range across 10 rounds).

Table 13. Branch coverage statistics of SyzMini, SyzMini$^+$, SyzMini$^+$ I and SyzMini$^+$ II.

Tool	Branch Coverage (*vs.* SyzMini)		
	v5.15	v6.1	v6.11
SyzMini	164.724K	169.015K	149.311K
SyzMini$^+$	169.268K (2.8%↑)	172.532K (2.1%↑)	153.165K (2.6%↑)
SyzMini$^+$ I	162.518K (1.3%↓)	168.702K (0.2%↓)	147.311K (1.3%↓)
SyzMini$^+$ II	167.253K (1.5%↑)	171.014K (1.2%↑)	151.224K (1.3%↑)

[1] ↑ and ↓ denote the coverage change achieved by the tool in that row compared to SyzMini.

Results. Figure 13 plots the average branch coverage achieved by SyzMini, SyzMini$^+$, SyzMini$^+$ I, and SyzMini$^+$ II during the 24-hour fuzzing campaigns. The overall coverage trends of the four tools are relatively similar, indicating that different attempt orders have a limited impact on overall branch coverage. Table 13 also lists the detailed branch coverage results. Taking Linux v5.15 as an example, compared to SyzMini, SyzMini$^+$ achieved the highest branch coverage improvement (2.8%), followed by SyzMini$^+$ II (1.5%), while SyzMini$^+$ I reduced coverage by 1.3%. The difference stems from the order of removal attempts used by these variants. In the first attempt, SyzMini$^+$ and SyzMini$^+$ II preserve only the direct influences, whereas SyzMini and SyzMini$^+$ I preserve both the direct and indirect influences. Because the first attempt of SyzMini$^+$ and SyzMini$^+$ II is more aggressive, they can remove more irrelevant calls (*e.g.*, calls that only have indirect influences on the target call) in a single attempt. This reduces the number of calls that need to be verified during subsequent one-by-one call removal, thereby improving minimization efficiency and achieving higher coverage than SyzMini. In contrast, although both SyzMini and SyzMini$^+$ I preserve both direct and indirect influences in their first attempt, SyzMini$^+$ I performs an additional second attempt even after the first attempt succeeds. The second attempt further removes calls by preserving only the direct influences. However, the additional program execution overhead introduced by this extra attempt outweighs the benefit of removing more indirectly influential calls. As a result, SyzMini$^+$ I achieves lower branch coverage than SyzMini.

Further, when the first attempt in SyzMini$^+$ fails, it performs a second attempt that preserves both direct and indirect influences, whereas SyzMini$^+$ II does not. The second attempt avoids the incorrect removal of calls whose influence is indirect in theory but still affects the target call's new coverage in practice. As a result, SyzMini$^+$ achieves the highest coverage among these four tools. Overall, the results demonstrate that different attempt orders can affect the effectiveness of the influence-guided call removal strategy, and the attempt order used in SyzMini$^+$ is the most effective.

We further perform the Mann–Whitney U-test to compare SyzMini$^+$ and SyzMini in terms of branch coverage. Specifically, SyzMini$^+$ consistently achieves higher coverage than SyzMini across all three kernel versions, with improvements ranging from 2.1% to 2.8%. However, statistical significance is only observed on Linux v6.1

Just Accepted

($p = 0.02575 < 0.05$). The differences on v5.15 ($p = 0.05205$) and v6.11 ($p = 0.07152$) are not statistically significant, as their p-values exceed the conventional $p < 0.05$ threshold. We attribute this to the stochastic nature of kernel fuzzing and the relatively modest improvement achieved by SyzMini$^+$. Although SyzMini$^+$ consistently improves coverage over SyzMini, the gain (around 2%–3%) is relatively small, and the inherent variability of fuzzing results can reduce statistical significance on some kernel versions.

In summary, for the influence-guided call removal strategy, within the threshold of 60% influence relations, increasing the number of relations can improve its effectiveness, and its optimization effect gradually stabilizes as the number of relations continues to grow. For the type-informed argument simplification strategy, the fixed-size parameters occupy a high proportion of tested system calls, which helps reduce the cost of simplifying arguments. For the extended influence-guided call removal strategy, its attempt order (first preserves the direct influence and then preserves both the direct and indirect influences if the first attempt fails) achieves the highest coverage among different attempt orders.

6 Discussion

SyzMini optimizes the minimization stage in kernel fuzzing and does not affect other parts of a coverage-guided kernel fuzzer. Figure 5 shows the extension implemented by SyzMini on top of Syzkaller. In the literature, several research works have been conducted to improve different aspects of kernel fuzzing (*e.g.*, generating system call descriptions [9, 10, 48], mutating the seeds [15, 49], and scheduling the fuzzing tasks [23]). SyzMini is orthogonal to this existing work and would not affect any of these improvements. Thus, our optimization strategies are not limited to Syzkaller. Our extensive evaluation in Section 5.5 has also shown the significant effectiveness improvement when integrating our strategies into three kernel fuzzers (SyzVege, Countdown, and SyzDirect) for different application scenarios. We believe integrating our optimization strategies with other kernel fuzzers will boost their effectiveness as well.

For influence relationships, current approaches (including SyzMini) typically treat influence relationships as binary values (0/1). Specifically, if removing call a changes the coverage of call b, then call a is considered to potentially influence call b. However, these relationships are not truly deterministic. Influence relationships are inherently context-dependent, and different program contexts may change whether one call actually affects another. Therefore, it is possible to model influence relationships using fractional or probabilistic values rather than binary indicators. Such a design may more accurately capture the likelihood of influence, and we plan to further explore this open problem in future work.

7 Related Work

Kernel Fuzzing. Since the success of Syzkaller [17], there has been much research work to improve the effectiveness of kernel fuzzing. A larger number of research works [5, 9, 10, 13, 20, 21, 48, 55] have been devoted to obtaining more comprehensive system call specifications for ensuring the syntactic validity of generated test cases. Moonshine [43] aims to collect the higher-quality initial seeds by distilling the historical execution system call sequences, thereby enhancing the subsequent mutation effectiveness. Other work [12, 15, 20, 26, 49] utilizes the relationship between system calls to narrow the mutation space. SyzVegas [53] leverages reinforcement learning to better schedule the fuzzing tasks, such as program generation, mutation, and minimization. Different from these existing works, our work SyzMini aims to reduce the cost of the minimization stage in kernel fuzzing, which has not been explored before.

Kernel Fuzzing Acceleration. There are some research works that focus on accelerating kernel fuzzing. Horus [38] improves the communication speed between the host machine and the virtual machine running the tested kernel by replacing the remote procedure calls (RPCs) with direct memory access (DMA). Agamotto [14] observes that the kernel driver fuzzers frequently execute similar test cases, so it dynamically creates multiple

Just Accepted

checkpoints for these test cases to skip redundant parts. Different from this work, our work SyzMini accelerates kernel fuzzing by reducing the cost incurred during the minimization stage.

Input Minimization. Zeller *et al.* [57] first proposed the concept of delta debugging to minimize the failure-inducing input. Delta debugging has been adapted in many scenarios. For example, AFL-family tools [31, 34, 56] use afl-cmin [29] and afl-tmin [30] to minimize the seed corpus for improving the effectiveness of subsequent fuzzing. Specifically, afl-cmin takes the seed corpus as input and attempts to remove the seed files that do not contribute to the total coverage achieved by the corpus. On the other hand, afl-tmin focuses on minimizing data within each seed file, trying to remove as much data as possible while preserving the same coverage achieved by the seed file. In compiler fuzzing, C-Reduce[45] is used to reduce the bug-triggering C programs by removing redundant code. In GUI testing, SimplyDroid [24] simplifies crash-triggering UI event traces for Android apps. However, these existing minimization techniques based on delta debugging cannot be directly applied to minimize the system call sequences from OS kernels. Because they do not consider the dependencies between the system calls and the type structures of arguments.

8 Conclusion

We have proposed two general and novel optimization strategies to reduce the cost of the minimization stage in kernel fuzzing. The influence-guided call removal strategy leverages the influence relation between system calls to optimize the call removal, and the type-informed argument simplification strategy utilizes the argument type information to avoid unnecessary simplification. Our evaluation shows that the minimization cost can be greatly reduced, compared to Syzkaller. These two optimization strategies also help improve branch coverage and find more unique bugs. We believe our optimization strategies are general and could benefit many other kernel fuzzers.

Additionally, we further introduce an additional attempt to improve the effectiveness of the influence-guided call removal strategy. We implemented the extended strategy on SyzMini, referred to as SyzMini$^+$. Our evaluation results show that this extension helps SyzMini$^+$ achieve higher code coverage and find more unique bugs than SyzMini.

Concurrent with our research, Syzkaller maintainers independently implemented similar input minimization optimization strategies on August 8, 2024. While the overall ideas are conceptually similar, our approach differs in how influence relationships are leveraged during call removal. Syzkaller performs a single removal attempt that preserves both direct and indirect influences. In contrast, our approach introduced a more aggressive call removal attempt before the original attempt. It first attempts to remove more calls by preserving only direct influences. If this attempt fails, it falls back to the conservative strategy that preserves both direct and indirect influences. As shown in Section 5.6, our aggressive call removal strategy provides additional improvements over the original Syzkaller strategy.

Just Accepted

Acknowledgments

We thank the anonymous ACM TOCS and ATC'25 reviewers for their valuable feedback. This work was supported in part by NSFC Project No. 62572192, National Key Research and Development Program (Grant 2022YFB3104002), Shanghai Trusted Industry Internet Software Collaborative Innovation Center and the Fundamental Research Funds for the Central Universities.

References

[1] 2024. SyzMini and SyzMini+. https://github.com/ecnusse/SyzMini.

[2] Erin Avllazagaj, Yonghwi Kwon, and Tudor Dumitras. 2024. SCAVY: Automated Discovery of Memory Corruption Targets in Linux Kernel for Privilege Escalation. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. https://www.usenix.org/conference/usenixsecurity24/presentation/avllazagaj

[3] Shuangpeng Bai, Zhechang Zhang, and Hong Hu. 2024. CountDown: Refcount-guided Fuzzing for Exposing Temporal Memory Errors in Linux Kernel. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*. ACM, 1315–1329. https://doi.org/10.1145/3658644.3690320

[4] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 2329–2344. https://doi.org/10.1145/3133956.3134020

[5] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. 2023. No Grammar, No Problem: Towards Fuzzing the Linux Kernel without System-Call Descriptions. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. https://www.ndss-symposium.org/ndss-paper/no-grammar-no-problem-towards-fuzzing-the-linux-kernel-without-system-call-descriptions/

[6] Jianchao Cao, Fan Guo, and Yanwen Qu. 2025. JNFuzz-Droid: a lightweight fuzzing and taint analysis framework for native code of Android applications. *Empir. Softw. Eng.* 30, 5 (2025), 113. https://doi.org/10.1007/s10664-025-10671-9

[7] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. 2011. Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In *APSys '11 Asia Pacific Workshop on Systems, Shanghai, China, July 11-12, 2011*. 5. https://doi.org/10.1145/2103799.2103805

[8] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. 2095–2108. https://doi.org/10.1145/3243734.3243849

[9] Weiteng Chen, Yu Hao, Zheng Zhang, Xiaochen Zou, Dhilung Kirat, Shachee Mishra, Douglas Lee Schales, Jiyong Jang, and Zhiyun Qian. 2024. SyzGen++: Dependency Inference for Augmenting Kernel Driver Fuzzing. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. 4661–4677. https://doi.org/10.1109/SP54263.2024.00269

[10] Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. 2021. SyzGen: Automated Generation of Syscall Specification of Closed-Source macOS Drivers. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. 749–763. https://doi.org/10.1145/3460120.3484564

[11] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. 2021. NtFuzz: Enabling Type-Aware Kernel Fuzzing on Windows with Static Binary Analysis. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 677–693. https://doi.org/10.1109/SP40001.2021.00114

[12] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. 2021. NtFuzz: Enabling Type-Aware Kernel Fuzzing on Windows with Static Binary Analysis. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. 677–693. https://doi.org/10.1109/SP40001.2021.00114

[13] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 2123–2138. doi:10.1145/3133956.3134069

[14] Ren Ding, Yonghae Kim, Fan Sang, Wen Xu, Gururaj Saileshwar, and Taesoo Kim. 2021. Hardware Support to Improve Fuzzing Performance and Precision. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. 2214–2228. https://doi.org/10.1145/3460120.3484573

[15] Marius Fleischer, Dipanjan Das, Priyanka Bose, Weiheng Bai, Kangjie Lu, Mathias Payer, Christopher Kruegel, and Giovanni Vigna. 2023. ACTOR: Action-Guided Kernel Fuzzing. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. 5003–5020. https://www.usenix.org/conference/usenixsecurity23/presentation/fleischer

[16] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. https://www.ndss-symposium.org/ndss2008/automated-whitebox-fuzz-testing/

[17] Google. 2015. Syzkaller. https://github.com/google/syzkaller.

[18] Weining Gu, Zbigniew Kalbarczyk, Ravishankar K. Iyer, and Zhen-Yu Yang. 2003. Characterization of Linux Kernel Behavior under Errors. In *2003 International Conference on Dependable Systems and Networks (DSN 2003), 22-25 June 2003, San Francisco, CA, USA, Proceedings*. 459–468. https://doi.org/10.1109/DSN.2003.1209956

[19] Hui Guo, Hao Sun, Shan Huang, Ting Su, Geguang Pu, and Shaohua Li. 2025. Optimizing Input Minimization in Kernel Fuzzing. In *Proceedings of the 2025 USENIX Annual Technical Conference, USENIX ATC 2025, Boston, MA, USA, July 7-9, 2025*. USENIX Association, 1451–1465. https://www.usenix.org/conference/atc25/presentation/guo

[20] HyungSeok Han and Sang Kil Cha. 2017. IMF: Inferred Model-based Fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 2345–2358. https://doi.org/10.1145/3133956.3134103

[21] Yu Hao, Guoren Li, Xiaochen Zou, Weiteng Chen, Shitong Zhu, Zhiyun Qian, and Ardalan Amiri Sani. 2023. SyzDescribe: Principled, Automated, Static Generation of Syscall Descriptions for Kernel Drivers. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. 3262–3278. https://doi.org/10.1109/SP46215.2023.10179298

Just Accepted

[22] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding Kernel Race Bugs through Fuzzing. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. 754–768. https://doi.org/10.1109/SP.2019.00017

[23] Dae R. Jeong, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. 2023. SegFuzz: Segmentizing Thread Interleaving to Discover Kernel Concurrency Bugs through Fuzzing. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. 2104–2121. https://doi.org/10.1109/SP46215.2023.10179398

[24] Bo Jiang, Yuxuan Wu, Teng Li, and W. K. Chan. 2017. SimplyDroid: efficient event sequence simplification for Android application. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. 297–307. https://doi.org/10.1109/ASE.2017.8115643

[25] Dave Jones. 2012. Trinity: Linux system call fuzzer. https://github.com/kernelslacker/trinity.

[26] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. https://www.ndss-symposium.org/ndss-paper/hfl-hybrid-fuzzing-on-the-linux-kernel/

[27] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. 2123–2138. https://doi.org/10.1145/3243734.3243804

[28] Piotr Krysiuk. 2023. Linux Kernel Use-after-free in Netfilter nf_tables When Processing Batch Requests can be Abused to Perform Arbitrary Reads and Writes in Kernel Memory. https://seclists.org/oss-sec/2023/q2/133.

[29] Michal "lcamtuf" Zalewski. 2016. afl-cmin. https://github.com/google/AFL/blob/master/afl-cmin.

[30] Michal "lcamtuf" Zalewski. 2017. afl-tmin. https://github.com/google/AFL/blob/master/afl-tmin.c.

[31] Michal "lcamtuf" Zalewski. 2017. AFLplusplus. https://github.com/AFLplusplus/AFLplusplus.

[32] Kyung-suk Lhee and Steve J. Chapin. 2003. Buffer overflow and format string overflow vulnerabilities. *Softw. Pract. Exp.* 33, 5 (2003), 423–460. https://doi.org/10.1002/spe.515

[33] Penghui Li, Wei Meng, and Kangjie Lu. 2022. SEDiff: scope-aware differential fuzzing to test internal function models in symbolic execution. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. ACM, 57–69. https://doi.org/10.1145/3540250.3549080

[34] Shaohua Li and Zhendong Su. 2023. Accelerating Fuzzing through Prefix-Guided Execution. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 1–27. https://doi.org/10.1145/3586027

[35] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. 2022. PATA: Fuzzing with Path Aware Taint Analysis. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 1–17. https://doi.org/10.1109/SP46214.2022.9833594

[36] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Jianzhong Liu, and Yu Jiang. 2024. Dodrio: Parallelizing Taint Analysis Based Fuzzing via Redundancy-Free Scheduling. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*. ACM, 244–254. https://doi.org/10.1145/3663529.3663844

[37] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. 2022. DirtyCred: Escalating Privilege in Linux Kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*. 1963–1976. https://doi.org/10.1145/3548606.3560585

[38] Jianzhong Liu, Yuheng Shen, Yiru Xu, Hao Sun, and Yu Jiang. 2024. Horus: Accelerating Kernel Fuzzing through Efficient Host-VM Memory Access Procedures. *ACM Trans. Softw. Eng. Methodol.* 33, 1 (2024), 11:1–11:25. https://doi.org/10.1145/3611665

[39] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Trans. Software Eng.* 47, 11 (2021), 2312–2331. https://doi.org/10.1109/TSE.2019.2946563

[40] Bart Miller. 1988. Virtualization without direct execution or jitting: Designing a portable virtual machine infrastructure. In Emerging Technologies for Information Systems, Computing, and Management. http://pages.cs.wisc.edu/~bart/fuzz/CS736-Projects-f1988.pdf.

[41] Aniruddhan Murali, Noble Saji Mathews, Mahmoud Alfadel, Meiyappan Nagappan, and Meng Xu. 2024. FuzzSlice: Pruning False Positives in Static Analysis Warnings through Function-Level Fuzzing. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 65:1–65:13. https://doi.org/10.1145/3597503.3623321

[42] Yannic Noller. 2018. Differential program analysis with fuzzing and symbolic execution. In *Proceedings of the 33rd ACM/IEEE International Conference on AutomatedSoftware Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. ACM, 944–947. https://doi.org/10.1145/3238147.3241537

[43] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 729–743. https://www.usenix.org/conference/usenixsecurity18/presentation/pailoor

[44] Awanish Pandey, Phani Raj Goutham Kotcharlakota, and Subhajit Roy. 2019. Deferred concretization in symbolic execution via fuzzing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19,*

Just Accepted

2019. ACM, 228–238. https://doi.org/10.1145/3293882.3330554
[45] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. 335–346. https://doi.org/10.1145/2254064.2254104
[46] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. 2024. SoK: Prudent Evaluation Practices for Fuzzing. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. 1974–1993. https://doi.org/10.1109/SP54263.2024.00137
[47] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. 167–182. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo
[48] Hao Sun, Yuheng Shen, Jianzhong Liu, Yiru Xu, and Yu Jiang. 2022. KSG: Augmenting Kernel Fuzzing with System Call Specification Generation. In *Proceedings of the 2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*. 351–366. https://www.usenix.org/conference/atc22/presentation/sun
[49] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. HEALER: Relation Learning Guided Kernel Fuzzing. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*. 344–358. https://doi.org/10.1145/3477132.3483547
[50] Xin Tan, Yuan Zhang, Jiadong Lu, Xin Xiong, Zhuang Liu, and Min Yang. 2023. SyzDirect: Directed Greybox Fuzzing for Linux Kernel. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*. 1630–1644. https://doi.org/10.1145/3576915.3623146
[51] Google Team. 2017. Syscall description language. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md.
[52] Syzkaller team. 2020. Syzbot's web dashboard. https://syzkaller.appspot.com/upstream.
[53] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael B. Abu-Ghazaleh. 2021. SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. 2741–2758. https://www.usenix.org/conference/usenixsecurity21/presentation/wang-daimeng
[54] Valentin Wüstholz and Maria Christakis. 2020. Targeted greybox fuzzing with static lookahead analysis. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. ACM, 789–800. https://doi.org/10.1145/3377811.3380388
[55] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2024. KernelGPT: Enhanced Kernel Fuzzing via Large Language Models. *CoRR* abs/2401.00563 (2024). https://doi.org/10.48550/arXiv.2401.00563
[56] Michal Zalewski. 2016. Aamerican fuzzy lop (AFL). https://github.com/google/AFL.
[57] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200. doi:10.1109/32.988498
[58] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. 2022. Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. 71–88. https://www.usenix.org/conference/usenixsecurity22/presentation/zeng

Received 4 December 2025; revised 15 May 2026; accepted 5 August 2026

Just Accepted