SyzSpec: Specification Generation for Linux Kernel Fuzzing via Under-Constrained Symbolic Execution

Yu Hao
University of California, Riverside
USA
yhao016@ucr.edu

Juefei Pu
University of California, Riverside
USA
jpu007@ucr.edu

Xingyu Li
University of California, Riverside
USA
xli399@ucr.edu

Zhiyun Qian
University of California, Riverside
USA
zhiyunq@cs.ucr.edu

Ardalan Amiri Sani
University of California, Irvine
USA
ardalan@uci.edu

Abstract

Fuzzing has become one of the most effective and widely used techniques for discovering bugs and vulnerabilities, particularly in large-scale and complex programs like operating system kernels. A notable example is the kernel fuzzer syzkaller, which has identified over 6,800 bugs in the Linux kernel, with more than 5,500 already fixed. A crucial reason behind the success of the syzkaller is its collection of syscall descriptions, which are typically provided by human experts. Although some methods exist for automatically generating these syscall descriptions for device drivers, they often fall short when dealing with complex user inputs. These existing methods either lack precision or have a limited analysis scope, resulting in incomplete syscall descriptions.

In this paper, we present SyzSpec, a tool designed to address these limitations by performing fully inter- procedural under- constrained symbolic execution on syscall handler functions. This approach enables SyzSpec to explore all possible user inputs and generate syscall descriptions with more precision. The primary innovation in SyzSpec is a novel method to improve symbolic pointer reasoning in under-constrained symbolic execution, working along with the under-under-constrained memory object (UCMO). We compared SyzSpec with existing automated solutions and manually written syscall descriptions from syzkaller. Our results demonstrate that SyzSpec achieves better coverage than other automated tools and offers coverage comparable to that of manually written syscall descriptions. Additionally, we evaluated SyzSpec on the latest stable version of the Linux kernel (v6.10) and identified 86 unique and previously unknown crashes across 11 different categories.

CCS Concepts

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '25, Taipei, Taiwan.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3744811

Keywords

OS kernels security; Vulnerability discovery; Fuzzing; Symbolic execution

ACM Reference Format:
Yu Hao, Juefei Pu, Xingyu Li, Zhiyun Qian, and Ardalan Amiri Sani. 2025. SyzSpec: Specification Generation for Linux Kernel Fuzzing via Under-Constrained Symbolic Execution. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3719027.3744811

1 Introduction

Fuzzing has become one of the most effective and widely used techniques for identifying bugs and vulnerabilities in software, particularly due to its practicality and success in real-world applications. For example, in the domain of large-scale and complex programs like operating system kernels, the state-of-the-art kernel fuzzer, syzkaller [11] has discovered over 6,800 bugs in the Linux kernel over the past several years, with more than 5,500 of these bugs already fixed [10].

A critical component of syzkaller is its collection of syscall descriptions, which are typically provided by human experts. These descriptions, together with the coverage-guided fuzzing algorithm, are essential to generate test cases that thoroughly explore the kernel code. There are currently limited methods available for automatically generating these syscall descriptions for device drivers. Existing solutions like DIFUZE [7] and SyzDescribe [13] rely on static analysis, but they struggle to handle complex user inputs effectively due to the limitations inherent in static analysis techniques.

Symbolic execution presents a promising alternative to overcome these limitations. Two notable tools that use symbolic execution are KSG [30] and SyzGen++ [5]. However, KSG has a limited scope of intra-procedural analysis, which often fails to capture detailed specifications of user inputs. In contrast, SyzGen++ performs inter-procedural dynamic symbolic execution (i.e., Angr [29]) to recover specifications of user inputs in syscall handlers. However, its reliance on memory snapshots still restricts the state space SyzGen++ can explore, e.g., it can only explore one conditional branch (if the condition takes a fixed value) or it will always resolve an indirect call to a specific target (according to its concrete memory value). These will lead to the incomplete generation of user input specifications.

We propose using under-constrained symbolic execution [27] to overcome these challenges, because it allows a more complete exploration of the codebase. However, it is challenging to reason about syscall handler code in an under-constrained fashion. This is because syscall handlers heavily rely on the use of generic pointers (even represented as integers), type casting, and pointer arithmetic. Symbolic pointer reasoning itself is still a challenge, not only for under-constrained symbolic execution but for symbolic execution in general. This makes it difficult to reason about what object a pointer truly points to, which is necessary for syscall description generation.

In this paper, we have designed and implemented a tool called SyzSpec that performs comprehensive under-constrained symbolic execution on system call handler functions. SyzSpec packages a number of novel solutions to effectively generate syscall descriptions. First, we design an important symbolic pointer reasoning technique that ensures both the accuracy and effectiveness of under-constrained symbolic execution, leading to a more complete exploration of user inputs. Second, we generate specifications from the execution paths with UCMO-enhanced. Finally, SyzSpec employs a specification-guided search to prioritize the exploration of execution paths.

We compared SyzSpec with existing automated solutions such as SyzDescribe, KSG, and SyzGen++, as well as manually written syscall descriptions from syzkaller. Our results show that SyzSpec achieves better coverage than other automated tools and comparable coverage to manually written syscall descriptions. Furthermore, we evaluate SyzSpec on the syscall interface of `io_uring`, which is not supported by other automated solutions. Additionally, we evaluated SyzSpec on the latest stable version of Linux kernel (v6.10) and identified 86 unique and previously unknown crashes across 11 different categories. We are currently reporting these crashes to the Linux kernel community and collaborating with them to fix these crashes.

We summarize our main contributions as below:

- We design a tool called SyzSpec based on under-constrained symbolic execution, with improved symbolic pointer reasoning and memory object modeling that enable accurate generation of syscall specifications.
- We implement a prototype of SyzSpec with several additional enhancements tailored for syscall specification generation. We will open-source SyzSpec and the generated syscall descriptions to facilitate replication of our results and to support future research.
- We evaluate SyzSpec on the latest stable version of the Linux kernel, demonstrating its effectiveness in discovering new crashes. We also compare SyzSpec with existing automated solutions and find that it offers better coverage and supports more syscalls.

2 Background

Linux Kernel Fuzzing and Syscall Descriptions. Syzkaller is a state-of-the-art operating system kernel fuzzer, with great success in finding over 6,800 kernel bugs [10]. Its success can be largely attributed to its manually curated syscall descriptions, which are in the syzlang [12] format, and the corresponding fuzzing algorithm [14].

```
io_uring_register$IORING_REGISTER_BUFFERS2(fd fd_io_uring,
        opcode const[IORING_REGISTER_BUFFERS2],
        arg ptr[in, io_uring_rsrc_register], size bytesize[arg])

io_uring_rsrc_register {
        nr        len[data, int32]
        flags     flags[io_uring_rsrc_flags, int32]
        resv2     const[0, int64]
        data      ptr64[in, array[iovec_out]]
        tags      ptr64[in, array[int64]]
}
```

Figure 1: A example of syscall descriptions in syzlang format.

Syscall descriptions primarily include the available syscall interfaces on the kernel and the inputs they require. Since syscalls serve as the main interface between the user space and kernel space, the fuzzer must be aware of the syscalls provided by the operating system kernel. The quality of syscall descriptions is mostly decided by the syscall argument descriptions, *e.g.,* type (*e.g.,* whether it is a pointer, the structure and the fields of the memory), size (*e.g.,* the size of an array), and any additional constraints (*e.g.,* supported values or ranges), as shown in Figure 1. An example is that a syscall argument can even involve pointer fields within a struct that points to other memory, *e.g.,* the `data` and `tags` fields in Figure 1. If the syscall descriptions do not specify this nested structure, it becomes challenging for the fuzzer to randomly generate a valid syscall argument. Specifically, generating a structure with an integer field that correctly points to another memory location of the appropriate size is difficult.

Generally speaking, it is preferable to have detailed descriptions, even if they are not completely accurate all the time. This is because if an inaccurate description does not lead to new code coverage (*e.g.,* wrong value ranges or wrong type of user inputs), syzkaller can still recover from it by ignoring the inaccurate description by its fuzzing algorithm. On the other hand, if some description is missing completely, *e.g.,* lack of type description of a syscall argument, syzkaller will be almost unable to generate a proper input for the argument.

Syscall Description Generation. Currently, the project repository of syzkaller includes a large number of syscall descriptions. These descriptions are manually created and face several challenges [13]. First, these descriptions may be incomplete or even inaccurate due to the labor-intensive nature of writing syscall descriptions. Second, maintaining these descriptions is resource-intensive, requiring continuous monitoring of kernel code changes which require description updates. Third, the approach is not scalable — hard to keep up with a continuous stream of new kernel modules, e.g., Android kernel drivers for OEM-specific devices [7, 32]. These challenges demonstrate the need for methods to automatically generate syscall descriptions. While some recent solutions have been proposed, they have important limitations in different aspects. As shown in Figure 2, we briefly outline these solutions and discuss their advantages and disadvantages.

DIFUZE [7] and SyzDescribe [13] rely on static analysis, with a focus on recovering syscall interfaces. They are less focused

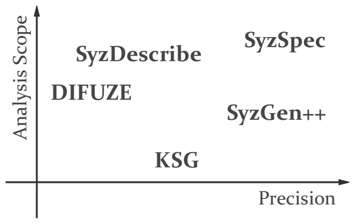

Figure 2: Conceptual comparison of different tools in terms of Analysis Scope and Precision.

on recovering syscall argument descriptions. In particular, they have hard-coded programming patterns regarding how syscall arguments are processed in driver-specific syscalls. Specifically, they focus on analyzing `ioctl()` handler functions through an inter-procedural static analysis, to extract the supported command values through various equality constraints, such as `switch cases` and `if` conditions. It also determines the argument type by examining the pointer types used in functions such as `copy_from_user()`, which transfer data from user space to kernel space. Unfortunately, while they handle the common cases, static analysis is inherently ill-suited to handle complex code constructs such as pointer operations, type casting, and arithmetic. For example, the `ioctl()` handler functions may apply complex operations to user input, including arithmetic or bitwise operations, or may involve custom data structures (*e.g.,* zero-length array). Additionally, syscall handlers might take user inputs to index tables or arrays, which are difficult to handle through static analysis alone.

Symbolic execution is an appropriate solution to address the limitations of static analysis, because it is designed to be precise in reasoning about program behaviors at the path level. Two notable tools that leverage symbolic execution are KSG [30] and SyzGen++ [5]. Unfortunately, KSG limits the symbolic execution scope to a single function, which is insufficient. In contrast, SyzGen++ performs inter-procedural dynamic symbolic execution. But it has its own downsides primarily because it relies on concrete memory snapshots which will limit its exploration space. For example, if the condition of an `if` statement depends on global variables that are fixed in the memory snapshot, this condition cannot be dynamically analyzed. Another example is the challenge of indirect function calls, where SyzGen++ simply takes one particular indirect call target (based on the memory snapshot), which misses other potential targets, leading to incomplete syscall descriptions. Additionally, SyzGen++ is built on Angr [29], a symbolic execution engine for binaries, it inevitably lacks the ability to infer the types of user input. This limitation is significant because accurate type information is crucial for a fuzzer to generate valid test cases.

3 Motivation

Given the limitations of existing approaches in generating syscall descriptions, we propose that *under-constrained symbolic execution* based on KLEE [4] offers a promising solution. First, as discussed earlier, unlike static analysis, symbolic execution is path-sensitive and includes path constraints, enabling it to handle complex operations on user inputs, such as arithmetic and bitwise operations. Second, compared to the dynamic symbolic execution used in SyzGen++, under-constrained symbolic execution not only initiates analysis on handler functions but also examines all possible code paths within these functions, leading to more accurate recovery of user inputs. Third, since KLEE is designed for symbolic execution on LLVM bitcode [19], we can infer the types of user inputs more accurately by performing analysis at the LLVM bitcode level compared to analysis at the binary level. However, symbolic pointer reasoning is complex and presents a major challenge, not only for under-constrained symbolic execution but for symbolic execution in general.

Under-Constrained Symbolic Execution is a variant of symbolic execution, which focuses on analyzing parts of a program where some prior constraints on the execution paths are not available. Unlike traditional symbolic execution, which requires starting from the entry point of the program with fully collected path constraints, under-constrained symbolic execution allows analysis to begin from any point within the program. In the context of the Linux kernel with multiple syscalls (entry points), it allows the analysis to pick any syscall handler and start the analysis, without worrying about dependencies (*e.g.,* needing to call `open()` before `read()`). All the variables, including pointers, that are allocated or initialized outside of the analysis scope will be considered symbolic or "under-constrained" (*i.e.,* the constraints of those variables are incomplete). In essence, the analysis offers an over-approximation compared to standard symbolic execution. This approach allows the analysis to explore a broader state space by eliminating the need for a memory snapshot (used in SyzGen++) and instead assuming that variables can take on arbitrary values. This tradeoff is desirable in the application of syscall description generation due to its need for a complete rather than perfectly accurate description, as alluded to earlier.

Challenges of Symbolic Pointer Reasoning. To accurately perform under-constrained symbolic execution and thoroughly explore all potential user inputs in syscall handler functions, it is crucial to effectively handle symbolic pointer reasoning during symbolic execution. However, dealing with symbolic pointers is a complex task that poses a significant challenge not only for under-constrained symbolic execution but for symbolic execution overall. Specifically, in the context of under-constrained symbolic execution, upon seeing a pointer variable for the first time, one needs to first decide whether the pointer points to a new abstract memory object or any of the existing objects. In current under-constrained symbolic execution work, they prefer to allocate a new memory object every time a new symbolic pointer variable is encountered [2, 5, 9, 27, 37, 38]. In other words, these pointers will be assigned distinct concrete addresses, missing possibilities that some of these pointers may be aliased with each other, causing under-exploration of the state space. This also means that they are unable to handle cyclical data structures such as linked lists (which are common in the Linux kernel). Conversely, if we follow the standard approach to handling generic pointers [4], *i.e.,* considering all possible memory objects, it can lead to a sharp increase in execution paths, a phenomenon known as path explosion, which can make the analysis nearly impossible. This complexity is one of the main reasons why SyzGen++ uses memory snapshots to limit the scope of the code it explores [5]. Nonetheless, to improve the generation of syscall descriptions, we

```
 1. struct A {int a1;  int a2;};
 2.
 3. struct B {struct A b1;  int b2;};
 4.
 5. void handler(long arg1, long arg2, long user_inputs) {
 6.   struct A* var1= (struct A*)arg1;                      // bitcast sym1
 7.   if (var1->a2 == 1) {                                  // load sym1+4
 8.     struct B* var2 = container_of(var1, struct B, b1); // bitcast sym1
 9.     var2-> b2 = 1;                                      // store sym1+8
10.   }
11.   struct A* var3= (struct A*)arg2;                      // bitcast sym2
12.   var3->a2 = 2;                                         // store sym2+4
13.   if (var1->a2 == 2) {                                  // load sym1+4
14.     ... // code related to user_inputs
15.   }
16. }
```

Figure 3: Motivating Example: The value of `arg1` and `arg2` are two symbolic variables, *i.e.,* `sym1` and `sym2`. How can we explore all possible paths with under-constrained symbolic execution?

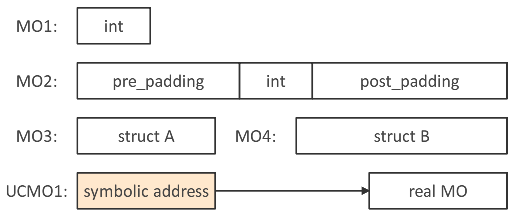

Figure 4: Memory objects of the symbolic pointers in Figure 3

must address the challenges associated with symbolic pointer reasoning in under-constrained symbolic execution.

One intuitive approach is to leverage static alias analysis for symbolic pointer reasoning. While static alias analysis can be helpful, it lacks the precision needed for this specific problem. For example, in loops, static analysis often fails to distinguish between pointers across iterations, whereas symbolic execution inherently differentiates them by assigning distinct symbolic values.

Motivating Example. Figure 3 provides a motivating example. It demonstrates the need for symbolic pointer reasoning. Without it, important code blocks will be missed altogether during the exploration, leading to incomplete syscall descriptions. First, we have `arg1` and `arg2` are symbolic variables, denoted as `sym1` and `sym2`, which are in fact pointers (see line 6 and line 11). As mentioned, when encountering symbolic pointers, (*e.g.,* load operation at line 7), existing solutions prefer to create a new memory object, `MO1`, as shown in Figure 4, for the pointer `sym1+4` and add a constraint `sym1+4 = address of MO1`. However, when we reach the store operation at line 10, we find that the pointer `sym1+8` results in an out-of-bound write because the size of `MO1` was not properly determined. A simple approach would be to create a larger memory object, such as `MO2` as shown in Figure 4, during the load operation at line 7, to avoid the out-of-bound write at line 10. However, this approach has several limitations: 1) The larger memory object may still not be large enough, leading to continued out-of-bound errors. 2) It increases memory usage and requires more time to handle state forking. 3) It places an additional computational burden on the SMT solver, making constraint resolution more time-consuming.

Alternatively, one can infer the size of a memory object early on. For instance, we can deduce that the memory size of the pointer `sym1` is `sizeof(struct A)` given line 6. Accordingly, we can create a memory object `MO3`, as illustrated in Figure 4. However, there is also a possibility that the memory size of `sym1` could be `sizeof (struct B)`, according to line 8. This uncertainty makes it difficult to accurately determine the correct size of the memory object for the symbolic pointer at the beginning.

Furthermore, it is important to determine whether a symbolic pointer can reference an existing memory object or if a new memory allocation is necessary. For instance, we need to understand that the symbolic pointer `sym2+4` at line 12 has the possibility to refer to the same memory location as the symbolic pointer `sym1+4` at line 13. This means the condition at line 13 can be satisfied and we should continue to explore the code in line 14 where `user_inputs` is used for proper syscall argument description generation, *e.g.,* we are yet to see the real type of `user_inputs` and its expected value ranges are missing. As mentioned, previous solutions [2, 5, 9, 27, 37, 38] prefer to ignore this possibility and have always created a new memory object for the symbolic pointer. A straightforward solution to solve this problem is to conservatively assume the symbolic pointer could point to any memory location and then verify each potential location separately to determine whether the path constraints are satisfied [4]. However, this method is impractical because it would result in an overwhelming number of states, leading to excessive computational overhead due to constraint solving.

4 SyzSpec Design

Now we present the workflow of SyzSpec, as illustrated in Figure 5. SyzSpec requires the LLVM bitcode [19] of Linux kernel components, e.g., kernel modules, as well as their entry functions (syscall handlers) as inputs. The output of SyzSpec is a set of syscall descriptions in the syzlang [12] format. As depicted in Figure 5, SyzSpec is composed of following main components:

- **User Input Exploration.** At a high level, SyzSpec initiates its process by analyzing user inputs using under-constrained symbolic execution on the entry functions. It leverages KLEE [4] to perform the under-constrained symbolic execution and accurately collect execution paths. To reduce inaccuracies during this symbolic execution, SyzSpec symbolizes all user inputs and non-constant global memory, including global variables and heap memory referenced from symbolic pointers. This allows the under-constrained symbolic execution to explore all possible branches of `if` and `switch` statements. It conducts a comprehensive inter-procedural analysis, also accounting for callee functions in indirect function calls.
- **Symbolic Pointer Reasoning.** The core of SyzSpec is a novel symbolic pointer reasoning method. This method is specifically designed to manage under-constrained symbolic pointers. It allows for a comprehensive exploration of user inputs, and mitigates the path explosion problem caused by the brute force symbolic pointer reasoning in under-constrained symbolic execution (that will consider a symbolic pointer to point to all

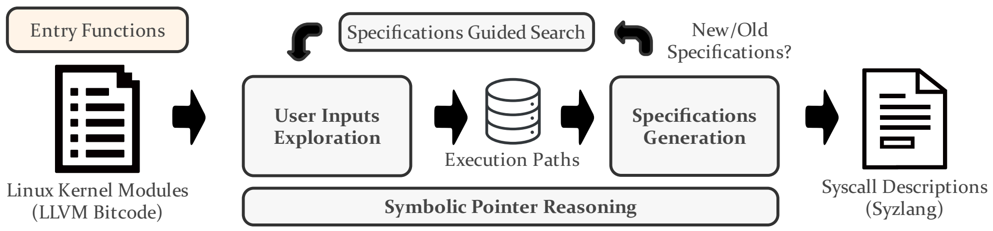
Entry Functions
Linux Kernel Modules (LLVM Bitcode)
Specifications Guided Search
User Inputs Exploration
Execution Paths
Specifications Generation
New/Old Specifications?
Symbolic Pointer Reasoning
Syscall Descriptions (Syzlang)

Figure 5: Workflow of SyzSpec.

possible objects subject to path constraints) [4]. In the end, symbolic pointer reasoning helps generate accurate specifications for the user inputs.

- **Specification Generation.** After analyzing the execution paths, SyzSpec generates the corresponding specifications. For each execution path, SyzSpec generates specifications for user inputs. By examining the complete execution path and its final state, SyzSpec determines the types, sizes, and constraints of user inputs, such as identifying whether they are inputs or outputs and specifying their possible values or ranges (as discussed in §4.2). SyzSpec then merges these specifications to remove redundancies, enhancing the efficiency of the syscall descriptions for fuzzing purposes. Finally, SyzSpec converts the syscall descriptions into the syzlang [12] format.
- **Specification Guided Search.** Furthermore, SyzSpec employs a specification guided search (detailed in §4.3) to prioritize the exploration of execution paths. The key idea is to use feedback on whether an execution path can contribute to new syscall descriptions, guiding the exploration process and mitigating the path explosion problem.

The following sections will provide a detailed explanation of the components of SyzSpec: Symbolic Pointer Reasoning (in §4.1), Specifications Generation (in §4.2) and Specifications Guided Search (in §4.3).

4.1 Symbolic Pointer Reasoning

Our approach provides the first relatively complete solution to overcome the challenges of symbolic pointer reasoning in under-constrained symbolic execution, We first introduce the concept of an under-constrained memory object (UCMO), designed specifically to handle symbolic pointers in under-constrained symbolic execution. As shown in Figure 4, in addition to using general memory objects with fixed, concrete addresses, a UCMO has a symbolic address paired with a corresponding memory object (with a concrete address). The symbolic address uniquely identifies the UCMO, while the real memory object (MO) is the concrete memory location that the symbolic pointer refers to. The key property of a UCMO is that it allows for the dynamic relocation and resizing of memory objects during symbolic execution. This means that if we discover that the size of a memory object is incorrect, we can adjust it as needed. Going back to the motivating example in Figure 3, we initially create a UCMO without a concrete size, named `UCMO1`, for the symbolic pointer `sym1`. The size of `UCMO1` can be adjusted later as needed. At line 6, we can deduce that the size of

Algorithm 1 Under-Constrained Symbolic Pointer Reasoning

```
 1: SymBaseMap ← {}
 2: SymTypeMap ← {}
 3: UCMOMap ← {}
 4: function CollectBaseAndType(Inst)
 5:     if Inst is GEP then
 6:         if GEP.offset ≥ 0 then
 7:             SymBaseMap[GEP.result] ← GEP.source
 8:             UpdateType(SymTypeMap, GEP.source)
 9:         else                                        ▹ For negative offsets
10:             SymBaseMap[GEP.source] ← GEP.result
11:             UpdateType(SymTypeMap, GEP.result)
12:     else if Inst is BitCast then
13:         UpdateType(SymTypeMap, BitCast.ptr)
14: function PointerReasoning(state, symPtr)
15:     basePtr ← StaticBackTrace(symPtr)
16:     while SymBaseMap.contains(basePtr) do
17:         basePtr ← SymBaseMap[basePtr]
18:     for all ucmo ∈ UCMOMap do
19:         if not TypeCompatible(ucmo, basePtr) then
20:             continue
21:         if Sat(state.pc, ucmo, basePtr, symPtr) then
22:             RelocateResize(ucmo, symPtr)
23:             state ← fork(state, ucmo, symPtr)
24:             if state = NULL then
25:                 break
26:     if state ≠ NULL then
27:         state.result ← CreateUcmo(basePtr)
28:         UCMOMap[basePtr] ← state.result
```

`UCMO1` should be `sizeof(struct A)`, allowing us to create the corresponding real memory object `MO3` (as shown in Figure 4). Similarly, at line 8, the size of `UCMO1` should be updated to `sizeof(struct B)`. Consequently, we can relocate and resize `UCMO1`, updating the corresponding real memory object to `MO4`.

Next, we design an efficient solution to check whether symbolic pointers can point to existing UCMOs as opposed to creating new UCMOs all the time. First, our approach resolves the symbolic pointer to a base address with an offset. The key insight is that the base address is crucial in determining whether the symbolic pointer could reference an existing UCMO. If the base address can be resolved to a UCMO, the symbolic pointer (*i.e.,* base address plus offset) might point to the memory before or after the UCMO. This indicates that the current size of the UCMO is not correct and that the UCMO needs to be updated/resized. However, the base address of the symbolic pointer must still fall within the UCMO.

Second, rather than fully relying on the path constraints resolved by an SMT solver, we integrate a basic type-based analysis [8] with constraint-based solution [4] to determine if the base address could reference an existing UCMO. Type analysis is less accurate but computationally cheaper compared to constraint solving.

The exact procedure is outlined in Algorithm 1. First, during the symbolic execution, we maintain two maps: `SymBaseMap` (line 1) and `SymTypeMap` (line 2). `SymBaseMap` stores the base addresses of symbolic pointers, while `SymTypeMap` stores the possible type information associated with each base address. When a GetElementPtr (GEP [24]) instruction is encountered, we extract the base address and update the type information of the symbolic pointer (lines 5 to 11). It is important to consider negative offsets in the GEP instructions, such as `container_of`, because these offsets indicate that the base address is the result of the GEP instruction, not its source (lines 9 to 11). For example in Figure 3, the symbolic expression `sym1` is the base address of both the symbolic pointer `sym1+4` at line 7 and the symbolic pointer `sym1+8` at line 9. If a type cast is encountered, we update the type information of the symbolic pointer accordingly (lines 12 to 13). During this update, we account for multiple potential types that a single pointer might represent. As illustrated in the example in Figure 3, the pointer `sym1` could correspond to `struct A` at line 6 and `struct B` at line 8.

With this information, when we encounter a symbolic pointer, `symPtr`, we first resolve it to its base address, `basePtr`, by performing a basic static backtrace (line 15) based on LLVM. Next, we look up the SymBaseMap to recursively find the possible base addresses (lines 16 to 17). This lookup process is essentially a path-sensitive backward value flow analysis (based on the propagation of symbolic expressions). After identifying the possible base address, `basePtr`, we iterate through all UCMOs to determine if the `basePtr` could reference an existing `ucmo`. Note that we exclude memory objects whose allocation sites have been observed, e.g., stack memory objects or newly allocated heap memory objects, because they are relatively well-constrained (*i.e.,* the execution path includes their allocation) and do not need to be considered potential memory objects for symbolic pointers. For each `ucmo`, we first check if its type is compatible with the `basePtr` (lines 19 to 20). If the type is incompatible, we skip this impossible `ucmo`. Otherwise, we are left with typically a few candidate UCMOs. We then use the expensive but accurate SMT solver to check if the constraints are satisfied (line 21). In addition to path constraints, we also need to add memory constraints for solving. These include: 1) the symbolic base addresses of `ucmo` should match the concrete addresses of their corresponding read memory objects. and 2) the value of the symbolic pointer should be within the range of the UCMO. If all the constraints (including path and memory ones) are satisfied, we consider the `ucmo` to be an object that the symbolic pointer can point to. At this point, we will relocate and resize the `ucmo` if necessary (line 22). We then fork the state to explore new potential UCMOs for the pointer `symPtr` (line 23). If the state is NULL at any time, it means the pointer `symPtr` cannot be resolved to any other objects (lines 24 to 25). It will break the loop and end the analysis. However, if after iterating through all UCMOs, the state is still not NULL, it means that symbolic pointer `symPtr` could still reference a new memory object, we thus would create a new UCMO for the base address `basePtr` (lines 26 to 28).

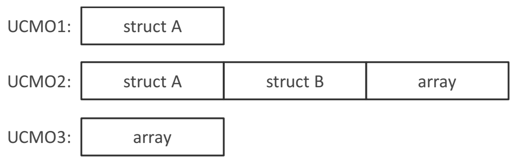

Figure 6: Possible types for `UCMO`.

For instance, in Figure 3, when we encounter the instruction `load sym1+4` at line 7, we recognize that its base address is `sym1`, and its type should be `struct A`. Consequently, we create UCMO1, with its corresponding real MO being `MO3` as shown in Figure 4. Later, when we encounter the instruction `store sym1+8` at line 9, we again determine that the base address is `sym1`. Based on memory constraints, we resolve this to `UCMO1`. Given that the type could now be `strcut B`, the algorithm will relocate and resize the real MO of `UCMO1` to `MO4`. When we reach the instruction `store sym2+4` at line 12, we identify its base address as `sym2`, with its type being `struct A`. Since it shares the same type with `UCMO1`, and the constraints are satisfied (with no current constraints for `sym2`), we first resolve it to `UCMO1`. We then fork the state with the constraint `sym2 = address of UCMO1`. Since the constraint `sym2 ≠ address of UCMO1` can also be satisfied, the forked state is not NULL, allowing the symbolic pointer `sym2` to reference a new memory location. In other words, the algorithm could consider two possible states when analyzing the code: (1) `arg1` and `arg2` being aliased, and (2) `arg1` and `arg2` not being aliased.

In summary, by dynamically relocating and resizing objects and allowing symbolic pointers to reference existing objects, our solution effectively manages symbolic pointers in under-constrained symbolic execution, ensuring that no possibilities are overlooked. In this example, this would allow us to explore important code blocks that would otherwise be missed (line 14) and critical for syscall description generation.

4.2 Specification Generation

This component generates specifications from the execution paths. These specifications include type information and constraints related to user inputs. Currently, our focus is on recovering four types of user inputs: primitive types (*e.g.,* integers with different widths and char), pointers, arrays, and structures (including unions).

For symbolic expressions of primitive types, we use an SMT solver to determine the possible values or range of the expression. Considering type casting, an integer type might represent a pointer. To decide whether the integer is actually a pointer, we feed the corresponding symbolic expression to our symbolic pointer reasoning. If the symbolic expression resolves to a UCMO, it indicates that the integer is a pointer.

For pointers, we first determine their direction, *i.e.,* whether it is an input, output, or input/output pointer. If the associated memory object has been written during the path, it is classified as an output pointer. If the associated memory object has been read from during the path, it is classified as an input pointer. If both, it is classified as an input/output pointer. Regarding the type of the pointer, we record the size and the most precise type for the UCMO during symbolic execution. Then we can recover the type of the pointer from the UCMO. Since we can easily distinguish between integers and pointers, we only record the struct, union, and array types for the UCMO. There are several considerations. First, when recording the type, we prefer the type with a larger size. The reason is that during fuzzing, it is almost impossible to generate valid user inputs when the input size is small. However, if the input size is large, fuzzing can still produce valid inputs by allocating a large memory block and using only a portion of it. Second, when the sizes are equal, we consider structs to be more precise than unions, and unions are more precise than arrays. During fuzzing, the fuzzer can only do random mutation for the array. it can randomly select one structure of the unions. But for structure, it can mutate each field based on its type. So structure is more effective for fuzzing than union and array. Second, when the sizes are equal, structs are considered more precise than unions, and unions are more precise than arrays. The reason is that structs are more effective for fuzzing compared to unions and arrays. During fuzzing, the fuzzer can only perform random mutations on arrays. For unions, it can randomly select one possible strcuts and then mutate it. However, for structs, the fuzzer can mutate each field based on its type. Third, a UCMO may have one or more segments (at different offsets), each with its own type. We consider three possible scenarios: 1) The UCMO has only a single type and its size matches the size of the entire object. In this case, we can directly infer the type information (*e.g.,* `UCMO1` in Figure 6). 2) The UCMO has multiple segments. For example, the object may initially be a large byte array, and only a subset of bytes are cast into more specific types. In such cases, we will construct a new struct type for the UCMO, filling in fields with the most precise type information available at the corresponding offsets and using arrays to fill in the remaining parts (*e.g.,* `UCMO2` in Figure 6). 3) If no type information is available at all, we recover the type of the UCMO as an array of bytes whose size matches the UCMO (*e.g.,* the `UCMO3` in Figure 6).

For arrays, currently, we only recover their overall size without retrieving the constraints for individual elements. This is because syzlang [12] does not currently support specifying constraints for each element within an array.

Finally, for structures, we recursively retrieve the type information and constraints for each field. If a field is a pointer, we also recursively retrieve the type of the memory it points to.

After generating the specifications for each execution path, the next step is to determine whether there are any new specifications from this path. These new specifications could include new types or constraints on user inputs, which would serve as feedback for the specifications guided search (mentioned later in §4.3). If no new specifications are found, we discard the current specifications to avoid redundancy. If new specifications are identified, we handle them in one of two ways: we either merge them with existing specifications of the same type or keep them separate if they differ in type. While the fuzzer can effectively mutate user input values based on comparison feedback from KCOV [17], mutating the type of user input is more difficult. For this reason, we choose to merge the value constraints of specifications that share the same type.

4.3 Specifications Guided Search

Unlike traditional (under-constrained) symbolic execution, which aims to explore all possible execution paths, our goal focuses on exploring all possible user inputs and generating their specifications. To achieve this and mitigate the path explosion issue, we need to prioritize the exploration of execution paths that are likely to lead to new specifications. Otherwise, our under-constrained symbolic execution will not be able to complete the analysis in a reasonable time for most complex modules. Inspired by coverage-guided fuzzing, which uses code coverage as feedback to guide input exploration in fuzzing, we introduce the specifications guided search. The main idea is to prioritize the exploration during the under-constrained symbolic execution based on whether a prior relevant path contributed to new specifications.

Algorithm 2 Specifications Guided Search

```
1:  StateSet ← {}
2:  state ← initState
3:  while state do
4:      while state.pc is valid do
5:          if ForkAt(state.pc) then
6:              RecordFork(state)
7:              if CheckPriority(state) then
8:                  forkedState ← Fork(state)
9:                  StateSet ← StateSet ∪ {forkedState}
10:         state.pc++
11:     spec ← GenerateSpec(state)
12:     UpdatePriority(state, spec)
13:     state ← SelectState(StateSet)
```

The specifications-guided search algorithm is presented in Algorithm 2. The process starts by selecting an initial state for exploration (line 2). Note that we need to generate the specifications of a path first, only then can we determine if any new specifications arise from that path. Thus, for each selected state, the algorithm first executes one path of the state until the end (similar to a depth-first search) to generate its specifications, which is done by the `while` loop from line 4 to line 10.

If the algorithm encounters a fork point during execution (line 5), it records these fork points along with the corresponding branches and execution paths (line 6). This information is used later to update the priority of the forked state. The algorithm then picks a branch with high priority to continue execution (line 7).

When the execution path ends, the algorithm generates specifications from that execution path and merges them as described in §4.2 (line 11). The priority of branches at the fork points and the states forked from those points along the execution path is updated (line 12). If it contributes anything new to the existing specifications, all previously recorded branches whose priorities will increase. The intuition is that the success of an execution path "depended" on all the fork points (*e.g.,* their constraints being useful to the specification generation) along the way. Therefore, we reward all fork points (branches). More specifically, a branch priority is computed as the total number of paths and the ratio of paths containing new specifications to the total number of paths that forked such a branch. Conversely, if an execution path finally does not contribute to any new specification, we will update the total number of paths for the branches and decrease their priority.

Finally, when selecting the next state to explore, the algorithm prioritizes the state associated with a higher-priority branch (line

13). Furthermore, if a branch's priority falls below a certain threshold — the total number of the paths is above a threshold (*i.e.,* 100) and the ratio of paths with new specifications is below a threshold (*i.e.,* 0.01) — that branch can be skipped, and the algorithm continues with the next state. Otherwise, the algorithm forks the state and adds the forked state to the set of states for further exploration (lines 8-9).

5 Implementation

As mentioned before, SyzSpec is built on top of KLEE [4], a symbolic execution engine designed for LLVM bitcode. We have modified approximately 8,600 lines of C++ code to adapt KLEE for SyzSpec. These modifications include adding support for under-constrained symbolic execution [1], specification generation, general symbolic execution improvements, and Linux-specific support. In this section, we will discuss some implementation aspects of SyzSpec in more detail.

LLVM Version. KLEE is now built on top of LLVM 13, while the latest stable version of the Linux kernel (*i.e.,* v6.10) officially supports to be compiled with the LLVM/Clang 14 toolchain [20]. We updated KLEE to work with LLVM 14 to support the latest Linux kernel.

Function Pointer. Unlike a regular symbolic pointer, a function pointer should reference an existing function. To accurately determine the potential targets of a function pointer, we combine a static type-based method analysis [25] with our new symbolic pointer reasoning approach. The static type-based method analysis narrows down possible targets to existing functions by considering type information, while our symbolic pointer reasoning approach further refines the target resolution using path constraints and memory constraints. This combined approach enables more accurate handling of function pointers and allows for a fully inter-procedural, under-constrained symbolic execution.

Assembly Code. The LLVM bitcode of the Linux kernel contains several instances of assembly code. However, KLEE currently does not support the symbolic execution of assembly code. To support the Linux kernel, we need to manage these assembly code segments appropriately. Since our goal is to perform under-constrained symbolic execution, our approach is just to return under-constrained symbolic expressions for assembly code with return values. We primarily handle two types of assembly instructions: the `callbr`[22] instruction and `call asm "inline assembler expressions"`[23]. Additionally, certain assembly code blocks involving memory operations, *e.g.,* `put_user` and `get_user`, are crucial for analyzing user inputs. For these instructions, we implement the corresponding memory operations in KLEE to ensure accurate symbolic execution.

Linked List. Our symbolic pointer reasoning approach effectively handles pointer operations in linked lists, *e.g.,* `list->next = list` and `container_of`. However, to maintain the structure of the linked list, additional constraints are required, *e.g.,* `list->prev->next = list` and `list->next->prev = list`. We ensure these constraints are maintained by adding those additional memory constraints (as mentioned in §4.1) during symbolic execution. In this way, we ensure that the linked list structure remains complete and that the symbolic execution is accurate.

Additional Type Information Collection. Typically, type information for memory can be obtained from the pointer referencing that memory. However, during memory copy operations, although only the value of the memory value is copied, the type information associated with the memory can also be shared with the new location. In the Linux kernel, this scenario is common with user input operations, *e.g.,* `copy_from_user`, `copy_to_user`, and `memdup`. To address this, we maintain a connection between the source and destination memory, allowing type information to be shared between them. This approach enables the collection of additional type information for user input, thereby improving the accuracy of the generated specifications.

Common Functions Modeling. We handle hundreds of common functions in the Linux kernel to ensure compatibility with KLEE operations and enhance performance. These common functions mainly include memory operations (*e.g.,* allocation, free, copy) and lock operations to ensure compatibility with KLEE operations. If these functions have unclear return values, we generate new under-constrained symbolic expressions to represent them.

Loop Unrolling. Although our specifications-guided search (as mentioned in §4.3) can help limit excessive loop unrolling, we also employ a standard method in symbolic execution: setting a maximum depth (*i.e.,* three in our implementation) for loop unrolling. However, since our primary focus is on exploring user inputs, we increase this maximum depth (*i.e.,* 256 in our implementation) if the loop conditions are fully related to user inputs, *i.e.,* all symbolic variables in the loop conditions are from user inputs.

6 Evaluation

In this section, we present our evaluation results. Specifically, we aim to answer the following questions:

- ***RQ1.*** How effective are syscall descriptions generated by SyzSpec in fuzzing compared with others? (§6.1)
- ***RQ2.*** What is the the performance of SyzSpec and our symbolic pointer reasoning solution? (§6.2)
- ***RQ3.*** Can syscall descriptions generated by SyzSpec find new vulnerabilities? (§6.3)

Dataset. As shown in Table 1, we selected 10 drivers in different categories and one non-driver kernel module, `io_uring`, from the Linux kernel as our target. We selected these 10 drivers because, based on the manual inspection of 100 drivers in the evaluation of SyzDescribe [13], there is still room for improvement in the syscall descriptions generated by existing automated tools for these drivers. In contrast, SyzDescribe has shown that for other drivers, especially simpler ones, its results are already close to the ground truth. Moreover, we do not fully include all 8 drivers in SyzGen++ as it primarily focuses on dependency inference, while these drivers involve syscall dependencies. In essence, the two tools are somewhat orthogonal in their objectives and can function complementarily. Additionally, we included a non-driver kernel module to demonstrate SyzSpec's ability to generate descriptions for general syscalls. We specifically chose `io_uring` because it is widely used

[1]UCKLEE [27] was built on an older version of KLEE and was not open source. Therefore, we implemented support for under-constrained symbolic execution in the latest version of KLEE.

Name	Category	Syscalls
capi20	Communication	ioctl, read, write
i2c	Communication	ioctl, read, write
infiniband	Networking	read, write
input_event	Input Devices	ioctl, read, write
mapper_control	Device Management	ioctl
ppp	Networking	ioctl, read, write
ptmx	Pseudo Terminals	ioctl
sg	SCSI Generic	ioctl, read, write
snd_seq	Sound	ioctl, read, write
uinput	Input Devices	ioctl, read, write
iouring	I/O Optimization	io_uring_register

Table 1: Dataset of fuzzing experiments, including 10 device drivers and one non-driver kernel module.

in many applications, and current automated tools do not support it.

Experimental Setup. All fuzzing experiments were conducted on a machine equipped with an Intel(R) Xeon(R) Gold 6248 CPU and 1024GB of RAM, running Ubuntu 18.04 LTS. The version of syzkaller and its syscall descriptions are both based on the *1e9c4cf* commit (dated 07/31/2024) from the syzkaller Git repository [11]. For the Linux kernel of the fuzzing experiments, we used the `syzbot` [10] configuration, which is the same setup that Google uses when fuzzing the Linux kernel with syzbot. And we compiled the kernel using the LLVM/Clang 14 toolchain. For the syzkaller configuration, we used the default configuration provided by syzkaller with QEMU virtual machines. SyzSpec requires the entry functions of kernel modules as input. For device drivers, SyzDescribe effectively identifies these entry functions, so we use its results directly. However, for other kernel modules, we currently collect the entry functions manually.

6.1 Effectiveness of SyzSpec in Fuzzing

In this section, we evaluate the effectiveness of SyzSpec by comparing fuzzing on the syscall descriptions it generates with others. We compare our results with three other tools—SyzDescribe [13], KSG[2] [30], and SyzGen++ [5]—as well as with manually written syscall descriptions from the official syzkaller project. For our evaluation, we fuzz each kernel driver individually for 24 hours, conducting three separate runs per driver. During each fuzzing session, we use 4 CPU cores, split into 2 QEMU instances with 2 cores each. The cpu time is enough for a single module and the coverage increases have plateaued based on our experience. After the sessions, we calculate the average code coverage and the average number of unique crashes for each driver based on these three runs. We selected the Linux kernel v6.6 for our experiments, as it is the latest long-term support (LTS) release as of October 29, 2023. This version was chosen because, while SyzSpec is compatible with the latest Linux kernel versions, the other tools we are comparing it with only work up to v6.6. However, some of their generated syscall descriptions do not have type information or value constraints for certain drivers, resulting in "N/A" entries. Table 2 shows the average code coverage achieved by the syscall descriptions generated by each tool, and Table 3 lists the average number of unique crashes. Overall, for the kernel modules that are supported, the descriptions generated by SyzSpec outperform or at least are comparable to those produced by the other automated tools.

SyzDescribe can generate syscall descriptions for all 10 drivers, and it explicitly indicates that 2 of these descriptions are incomplete (*i.e.,* `infiniband` and `snd_seq`). For the syscall descriptions of the other 8 drivers, SyzSpec achieves higher average coverage and discovers more unique crashes. This is primarily because SyzDescribe performs static analysis to recover specifications of user inputs, which works effectively for most `ioctl` syscalls. However, as shown in Table 1, some modules also use other syscalls like `read`, `write`, and even `io_uring`. For instance, the `infiniband` module only involves `read` and `write` syscalls, without `ioctl` at all. In the case of `snd_seq`, even though it uses the `ioctl` syscall, it involves complex user inputs that SyzDescribe cannot recover. In contrast, SyzSpec accurately handles complex user inputs for different syscalls through its precise under-constrained symbolic execution.

KSG can generate syscall descriptions for all 10 drivers. But for 7 of them, it cannot include any type information or value constraints. Although KSG can recover specifications for the syscall `read` and syscall `write`, it only performs an intra-procedural analysis. This limitation is significant because most kernel modules process user inputs in the callees of the handler function. As a result, KSG cannot recover any type information or value constraints for these 7 kernel modules. In contrast, SyzSpec performs a fully inter-procedural analysis, allowing it to generate specifications for these modules. For the `snd_seq` module, although user inputs are directly used in the syscall handler function, the analysis involves symbolic pointers, which KSG cannot handle effectively. However, SyzSpec can reason about symbolic pointers efficiently, achieving 85% (4835 vs. 2606) more coverage than KSG in `snd_seq` module. For the remaining two modules, SyzSpec performs comparably to KSG.

SyzGen++ successfully generates syscall descriptions for 4 drivers in our dataset. For the `ppp` and `sg` drivers, our tool performs better than SyzGen++. One main reason is that there are conditions whose values come from global memory, as well as symbolic pointers and function pointers. As discussed previously, SyzGen++ directly uses a fixed value from a memory snapshot for them, limiting the possibilities. In contrast, SyzSpec performs an under-constrained symbolic execution, which leads to more complete syscall descriptions.

The official syzkaller provides syscall descriptions for 9 drivers and the `io_uring` module. The syscall descriptions generated by SyzSpec are competitive with those from syzkaller. SyzSpec achieves better coverage in 6 kernel modules, while the official syscall descriptions perform better in 3 modules. For the remaining 2 modules, we observe comparable coverage. Note that since the syzkaller descriptions for these drivers were improved due to the reporting by SyzDescribe authors [13], these descriptions serve as a stronger target for comparison. It is important to note that for the non-driver `io_uring` module, SyzSpec is the only automated tool that supports it, as all prior solutions specialized in driver-related syscall handlers. For example, SyzDescribe hard-code the handling of the `cmd` parameter in `ioctl()` assuming it will never be cast into a pointer. In contrast, SyzSpec treats all parameters the same and does not apply special handling to any syscall or syscall handler

[2]KSG was not open source and we use binary received from its authors for testing.

Name	Syzkaller	SyzSpec	SyzDescribe	KSG	SyzGen++
capi20	2,881.7	3,189.3	2,756.3	2,996.7	N/A
i2c	4,681.3	6,167.3	4,698.0	N/A	N/A
infiniband	7,073.7	4,708.3	N/A	N/A	N/A
input_event	4,749.7	6,382.3	5,286.3	N/A	N/A
mapper	N/A	3,223.0	1,997.7	N/A	N/A
ppp	6,253.7	6,124.3	5,887.3	N/A	5,633.0
ptmx	14,190.3	10,099.0	10,213.7	10,232.3	10,879.0
sg	7,023.0	9,813.7	8,352.3	N/A	7,528.7
snd_seq	4,384.3	4,835.0	N/A	2,606.3	4,878.7
uinput	5,759.0	4,493.7	4,531.0	N/A	N/A
iouring	6,716.0	6,656.7	N/A	N/A	N/A

Table 2: Comparison of the average code coverage between SyzSpec and previous work.

Name	Syzkaller	SyzSpec	SyzDescribe	KSG	SyzGen++
capi20	0.0	0.0	0.0	0.0	N/A
i2c	0.0	0.3	0.0	N/A	N/A
infiniband	0.0	0.7	N/A	N/A	N/A
input_event	0.0	0.0	0.0	N/A	N/A
mapper	N/A	0.7	0.0	N/A	N/A
ppp	0.0	0.3	0.0	N/A	0.3
ptmx	1.3	0.0	0.0	0.0	0.0
sg	3.3	2.7	1.7	N/A	2.0
snd_seq	1.3	0.0	N/A	0.3	0.0
uinput	1.7	1.0	0.7	N/A	N/A
iouring	0.0	0.0	N/A	N/A	N/A

Table 3: Comparison of the average number of unique crashes between SyzSpec and previous work.

parameters. Furthermore, for `io_uring`, SyzSpec achieves coverage similar to that of the official syscall descriptions. We are also working on sharing syscall descriptions generated by SyzSpec with the syzkaller team.

6.2 Performance of SyzSpec and Symbolic Pointer Reasoning

Table 4 shows how long SyzSpec takes to generate syscall descriptions for 11 different kernel modules and the number of times symbolic pointer reasoning is invoked (*i.e.,* how many symbolic pointers are dereferenced) during the execution paths. The execution times range from 0.13 hours to 16.60 hours, depending on the module. We acknowledge the additional computation time required by SyzSpec, compared to existing lightweight solutions such as SyzDescribe. Nevertheless, our solution is designed to handle the complex syscall handlers, e.g., with complex constraints on user inputs, pointer arithmetic, and type casting. We argue that these difficult cases are more likely to contribute to more coverage and bugs. Indeed, Table 2 showed that SyzSpec has a higher coverage and Table 3 showed that SyzSpec can uncover more bugs compared to SyzDescribe and other existing solutions.

Specifications Guided Search. The number of execution paths analyzed by SyzSpec ranges from 209 to 23,297 across different kernel modules. However, we find that there is no direct correlation between the execution time and the number of paths analyzed. Some paths take longer to analyze because they involve more complex constraint-solving processes. For example, the `uinput` module took 8.00 hours to analyze merely 422 paths. The table also includes the number of partially completed paths. These are paths that are terminated early, mainly due to the specification-guided search or loop unrolling limit. Note that even if they did not reach the end of the syscall handler, they may still contribute to new syscall descriptions. That is why we always try to generate descriptions for every terminated path. Generally, as the number of execution paths increases, the proportion of paths that are partially completed also tends to rise. This happens because, initially, each execution path generates new specifications for user inputs. However, as more paths are forked, the new paths eventually stop providing additional useful information about the user inputs.

Symbolic Pointer Reasoning. During these execution paths, the number of times symbolic pointer reasoning is invoked or the number of symbolic pointers are dereferenced (*i.e.,* #Symbolic Pointer Reasoning) ranges from 776 to 33781 across different kernel modules. This indicates that symbolic pointer reasoning is broadly applicable during under-constrained symbolic execution. The number of possible memory objects for symbolic pointers resolved by SyzSpec (*i.e.,* #MO by SyzSpec) ranges from 886 to 34,354, while the average number of memory objects per symbolic pointer resolved by SyzSpec (*i.e.,* #Avg. MO by SyzSpec) falls between 1.00 and 1.14. In contrast, if we resolve the symbolic pointers using the standard method in KLEE, the number of possible memory objects (*i.e.,* #MO by standard KLEE) would range from 11469 to 4541465, and the average memory objects per symbolic pointer (*i.e.,* #Avg. MO by standard KLEE) would range from 1.07 to 232.22. It is important to note that this statistic underestimates the path explosion issue caused by KLEE's standard symbolic pointer reasoning method. This is because we only calculate the number of possible memory objects without forking on pointer reasoning and executing paths based on those objects. Otherwise, the execution would be infeasible to complete due to the intensive use of the SMT solver. Specifically, the analysis of all modules, except `mapper_control`, cannot finish within 24 hours and fails to generate valid syscall descriptions for fuzzing. Despite this underestimation, the data still demonstrates that KLEE's standard symbolic pointer reasoning method results in significant path explosion in most modules, while our method significantly alleviates this issue.

Name	Time	Total Path	Partially Completed Path	Partially Completed / Total	#Symbolic Pointer Reasoning	#MO by SyzSpec	#Avg. MO by SyzSpec	#MO by standard KLEE	#Avg. MO by standard KLEE
capi20	2.29	209	119	56.94%	776	886	1.14	85372	110.02
i2c	1.86	12,628	8,403	66.54%	12885	14570	1.13	2653234	205.92
infiniband	0.13	567	136	23.99%	7363	7478	1.02	132162	17.95
input_event	4.49	17,451	12,532	71.81%	21678	22749	1.05	1345485	62.07
mapper_control	0.27	627	359	57.26%	10750	10751	1.00	11469	1.07
ppp	10.75	4,257	2,410	56.61%	2386	2618	1.10	350817	147.03
ptmx	16.60	16,934	13,832	81.68%	33781	34354	1.02	4541465	134.44
sg	0.54	5,154	3,450	66.94%	27084	27582	1.02	4009177	148.03
snd_seq	7.01	5,197	3,211	61.79%	10426	10808	1.04	547282	52.49
uinput	8.00	422	100	23.70%	5883	6002	1.02	128273	21.80
iouring	7.31	23,297	19,711	84.61%	15395	16826	1.09	3574961	232.22

Table 4: Execution time required by SyzSpec to generate syscall descriptions for those 11 kernel modules and the number of the symbolic pointer reasoning during the execution paths.

Risk	Sum	C repro	Syzlang repro	no repro
out-of-bounds	1			1
use-after-free	4	1		3
use-before-initialization	6	1		5
null-ptr-deref	29	7	3	19
corrupted stack	4	2	1	1
corrupted list	9	3		6
corrupted lock	2	2		0
page fault	4	1		3
deadlock	5	2	1	2
task hung	12	2		10
logic bug	10	2	1	7
Sum	86	23	6	57

Table 5: Overview of the unknown crashes.

6.3 Bug Finding Capability of SyzSpec

To evaluate how well the syscall descriptions generated by SyzSpec can identify bugs, we conducted a one-month fuzzing campaign on the latest stable version of the Linux kernel (v6.10, released on 07/14/2024) using 70 CPU cores. The fuzzing results are shown in Table 5. This effort resulted in the discovery of 100 unique crashes. After cross-validation with bugs in syzbot [10], we find that 86 of which were previously unknown.

As shown in Table 5, there are a total of 86 previously unknown crashes, which can be categorized into 11 different risk types, including out-of-bounds errors, use-after-free errors, and use-before-initialization errors. Notice that the reproducer can be highly beneficial in assisting developers with diagnosing the root causes of bugs. However, only 23 of these crashes have a C reproducer (a test case written in C that can trigger the crashes), 6 have a syzlang reproducer (a test case written in syzlang that can trigger the crashes but a corresponding C test case can not trigger the crashes), and 57 do not have any reproducer at all.

Table 6 list the details of previously unknown crashes in the Linux kernel, identified using syscall descriptions generated by SyzSpec. Table 6 includes 38 crashes that either have a reproducer or involve memory corruption, while **??** contains the remaining crashes. Among the crashes with a reproducer, 8 require multiple threads to trigger, which means those bugs are more difficult to be fixed. We have finished the report process to the Linux kernel security team and the relevant maintainers. Our first focus is on fixing crashes that have a reproducer (a way to reliably replicate the issue) or involve memory corruption. We will provide an update on the status of these crashes in the final version of the paper (some bugs can still be triggered in the latest version of the linux kernel).

```
1. struct A {int a_1;  int a_2;}
2. struct B {struct A b_1; struct A b_2;}
3. struct C {struct A c_1; struct A c_2; struct A c_3}
4. array[struct A]
```

Figure 7: An example of specification merging.

7 Discussion and Limitations

There are a few limitations of SyzSpec, which can be areas for future improvements.

Specifications Merge. Currently, we only merge specifications of the user inputs with the same type. However, there are situations where specifications of different types can also be merged. For example, in Figure 7, there is a case where specifications of different types need to be merged. The structures `struct B` and `struct c` represent the second situation mentioned in §4.2. Although all three structures are of different types, they can be merged into the array on line 4, where the elements are of type `struct A` and without fixed length. However, these potential merges heavily rely on heuristics, and it is challenging to encode all of them.

Entry Functions. Currently, the SyzSpec requires entry functions as input. However, existing tools are limited to automatically recovering entry functions specifically for drivers with hard-coded domain knowledge. For instance, DIFUZE identifies handler functions for syscall interfaces by first locating syscall handler structures using a predefined list of struct types. SyzDescribe recovers syscall handlers by statically reconstructing the initialization process of a kernel driver. KSG and SyzGen++ scan all device files under `/dev`, searching for syscall handler structures triggered when the `open()`

Crashed Function	Risk	Repro	Threaded	Crashed Function	Risk	Repro	Threaded
fast_imageblit	out-of-bounds			stack_depot_save_flags	null-ptr-deref	C	
addrconf_verify_rtnl	use-after-free			update_io_ticks	null-ptr-deref	C	
dup_fd	use-after-free			dup_mmap	corrupted stack	Syzlang	
nsim_fib6_rt_nh_del	use-after-free			sg_ioctl	corrupted stack	C	Yes
rcu_core	use-after-free	C	Yes	worker_thread	corrupted stack	C	
addrconf_ifdown	use-before-init	C		call_timer_fn	corrupted list	C	
calculate_sigpending	use-before-init			dst_init	corrupted list	C	
exit_fs	use-before-init			free_pgtables	corrupted lock	C	
neigh_periodic_work	use-before-init			process_measurement	corrupted lock	C	
schedule_timeout	use-before-init			deliver_ptype_list_skb	page fault	C	
tipc_release	use-before-init			__run_timer_base	deadlock	Syzlang	
__free_object	null-ptr-deref	C		hci_dev_do_close	deadlock	C	
batadv_bla_del...	null-ptr-deref	C		serial8250_console_write	deadlock	C	Yes
batadv_iv_send...	null-ptr-deref	Syzlang		gsm_cleanup_mux	task hung	C	
blk_mq_put_tag	null-ptr-deref	C		kvfree_rcu_bulk	task hung	C	Yes
fib_table_lookup	null-ptr-deref	Syzlang		rss_counter	logic bug	C	
get_mem_cgroup...	null-ptr-deref	C		depot_fetch_stack	logic bug	Syzlang	Yes
mmap_region	null-ptr-deref	C		free_pgtables	logic bug	C	Yes
seccomp_run_filters	null-ptr-deref	C	Yes	input_mt_init_slots	logic bug	C	Yes

Table 6: Parts of previously unknown crashes found by syscall descriptions generated by SyzSpec in the Linux kernel, including the crashes that have a reproducer or involve memory corruption.

syscall is executed on these files, identifying explicitly referenced structures. Therefore, it would be useful to develop a more generalized method to recover handler functions for syscall interfaces. There are several options: one could manually encode the domain knowledge related to other non-driver kernel components, e.g., socket and file system. Alternatively, we envision that it is possible to leverage large language models, which already have significant knowledge about the Linux kernel and syscall internals, to automatically extract the necessary domain knowledge. These are orthogonal to our project and we leave them as future work.

Finally, while our under-constrained symbolic execution approach has been applied to generate specifications for Linux kernel syscalls, this technique is not limited to this particular use case. It can be extended to other types of interfaces, such as library APIs. Specifically, as long as the interfaces of other programs are provided, SyzSpec can be easily adapted to support those programs.

8 Related Work

Linux Kernel Fuzzing. Recent studies have made significant improvements in various aspects of kernel fuzzing, and most of them are building upon the state-of-the-art kernel fuzzer, syzkaller. HFL [18] enhances kernel fuzzing by combining syzkaller with symbolic execution. This approach improves various aspects of kernel fuzzing, *e.g.,* resolving nested argument types and identifying dependencies that involve non-open file descriptors. Although HFL does not focus on directly generating syscall descriptions, the inferences gained through this method can be adapted to enhance certain aspects of syscall descriptions. HEALER [31] improves the quality of test cases and maximizes code coverage by learning the relationships between syscalls. This method relies on existing syscall descriptions provided by syzkaller. A recent study [14] measured the code that remains uncovered after extensive fuzzing and identified gaps in the existing kernel fuzzing. FUZZNG [3] introduces a dynamic approach that hooks all `copy_from_user`-like functions to directly inject data. Instead of generating complex nested input structures, this method simplifies the process by breaking down multi-layer pointers into a series of single-layer buffers.

Under-Constrained Symbolic Execution. There are several works employing under-constrained symbolic execution to finish the analysis task. The work by Ramos *et al.* [27] is one of the most significant contributions to under-constrained symbolic execution in recent years. It presents a relatively complete framework for under-constrained symbolic execution and demonstrates its application in verifying real-world code, such as vulnerability detection and patch verification. Subsequent works such as UBITect [37] and Progressive Scrutiny [38] leverage under-constrained symbolic execution to reduce false positives in static analysis when detecting use-before-initialization bugs in the Linux kernel. LinKRID [21] applies under-constrained symbolic execution to analyze detailed refcount and global reference changes, effectively identifying imbalance reference counting bugs in the Linux kernel. Sys [2] integrates under-constrained symbolic execution with static analysis to identify bugs in web browsers.

Symbolic Pointer Reasoning. Recent research studies have been conducted on symbolic pointer reasoning within symbolic execution. MemSight [6] offers a novel approach to symbolic pointer reasoning. Rather than resolving symbolic pointers in some memory, MemSight focuses on efficiently associating values with symbolic address expressions. It assigns a dereferenced value of the pointer to the most recent value based on timestamps sequentially. However, MemSight is not designed for under-constrained symbolic execution. MemSight is primarily effective in situations where symbolic offsets are applied to specific allocated memory, and the dereferenced value of the pointer should be one of that memory. In contrast, under-constrained symbolic execution requires first determining which memory—potentially even a new one—the symbolic pointer refers to. Past-Sensitive Pointer Analysis [33] differentiates

between objects that were allocated in the past, during the execution path, and objects that will be allocated in the future, which may share allocation sites depending on the sensitivity of the pointer analysis. David *et al.* [35] propose a relocatable addressing model for symbolic execution, aimed at enhancing the segmented memory model and lowering the computational cost of solving large array theory constraints. However, both approaches may not be suitable for under-constrained symbolic execution because the allocation sites might exist before the entry point of the symbolic execution.
Symbolic Execution Optimization. Several studies have focused on optimizing symbolic execution. The work [16] proposes a segmented memory model for symbolic execution. The main idea is to divide memory into multiple segments, so that any symbolic pointer can only refer to memory objects within a single segment. MultiSE [28] introduces a method that allows executing multiple paths simultaneously, which naturally supports state merging. FastKLEE [36] enhances the speed of KLEE by reducing redundant bound checks for type-safe pointers. Chopped symbolic execution [34] aims to skip certain functions and continue symbolic execution. Borzacchiello *et al.* [1] improve constraint solving by fuzzing symbolic expressions He *et al.* [15] apply machine learning to explore paths for symbolic execution. SymQEMU [26] increases execution speed through a compilation-based method for binaries.

9 Conclusion

In conclusion, we introduced SyzSpec, a tool designed to thoroughly explore all possible user inputs and generate more accurate syscall specifications by performing fully inter-procedural under-constrained symbolic execution on syscall handler functions. The primary innovation of SyzSpec is its novel method for enhancing symbolic pointer reasoning within the context of under-constrained symbolic execution, which works together with the under-constrained memory object (UCMO). We compared SyzSpec against existing automated solutions and the manually written syscall descriptions provided by the official syzkaller tool. Our results indicate that SyzSpec achieves superior coverage compared to other automated tools, and offers coverage that is comparable to manually crafted syscall descriptions. Furthermore, we evaluated SyzSpec using the latest stable version of the Linux kernel (v6.10), uncovering 86 previously unknown crashes across 11 different categories.

Acknowledgment

We thank the anonymous reviewers for their insightful comments and valuable suggestions. This material is based upon work supported by the National Science Foundation under Grant No. #2247881 & #2247880 and the United States Air Force and DARPA under Agreement No. FA8750-24-2-0002.

References

[1] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. 2021. Fuzzing Symbolic Expressions. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 711–722. doi:10.1109/ICSE43902.2021.00071

[2] Fraser Brown, Deian Stefan, and Dawson R. Engler. 2020. Sys: A Static/Symbolic Tool for Finding Good Bugs in Good (Browser) Code. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 199–216. https://www.usenix.org/conference/usenixsecurity20/presentation/brown

[3] 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*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/no-grammar-no-problem-towards-fuzzing-the-linux-kernel-without-system-call-descriptions/

[4] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[5] Weiteng Chen, Yu Hao, Zheng Zhang, Xiaochen Zou, Dhilung Kirat, Shachee Mishra, Douglas Schales, Jiyong Jang, and Zhiyun Qian. 2024. SyzGen++: Dependency Inference for Augmenting Kernel Driver Fuzzing. In *IEEE Symposium on Security and Privacy*.

[6] Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2017. Rethinking pointer reasoning in symbolic execution. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 613–618. doi:10.1109/ASE.2017.8115671

[7] 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*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2123–2138. doi:10.1145/3133956.3134069

[8] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. 1998. Type-Based Alias Analysis. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, Jack W. Davidson, Keith D. Cooper, and A. Michael Berman (Eds.). ACM, 106–117. doi:10.1145/277650.277670

[9] Galois. 2024. Under-Constrained Symbolic Execution with Crucible. https://galois.com/blog/2021/10/under-constrained-symbolic-execution-with-crucible/. https://galois.com/blog/2021/10/under-constrained-symbolic-execution-with-crucible/

[10] Google. 2024. Syzbot. https://syzkaller.appspot.com/upstream. https://syzkaller.appspot.com/upstream

[11] Google. 2024. syzkaller. https://github.com/google/syzkaller. https://github.com/google/syzkaller

[12] Google. 2024. Syzlang. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions.md. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions.md

[13] 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*. IEEE, 3262–3278. doi:10.1109/SP46215.2023.10179298

[14] Yu Hao, Hang Zhang, Guoren Li, Xingyun Du, Zhiyun Qian, and Ardalan Sani. 2022. Demystifying the Dependency Challenge in Kernel Fuzzing. In *44rd IEEE/ACM International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, 22-27 May 2022*. IEEE, 1–11.

[15] Jingxuan He, Gishor Sivanrupan, Petar Tsankov, and Martin T. Vechev. 2021. Learning to Explore Paths for Symbolic Execution. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 2526–2540. doi:10.1145/3460120.3484813

[16] Timotej Kapus and Cristian Cadar. 2019. A segmented memory model for symbolic execution. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 774–784. doi:10.1145/3338906.3338936

[17] The kernel development community. 2024. KCOV: code coverage for fuzzing. https://www.kernel.org/doc/html/v6.6/dev-tools/kcov.html.

[18] 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*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/hfl-hybrid-fuzzing-on-the-linux-kernel/

[19] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. doi:10.1109/CGO.2004.1281665

[20] Linux. 2024. Building Linux with Clang/LLVM. https://www.kernel.org/doc/html/latest/kbuild/llvm.html. https://www.kernel.org/doc/html/latest/kbuild/llvm.html

[21] Jian Liu, Lin Yi, Weiteng Chen, Chengyu Song, Zhiyun Qian, and Qiuping Yi. 2022. LinKRID: Vetting Imbalance Reference Counting in Linux kernel with Symbolic Execution. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 125–142. https://www.usenix.org/conference/usenixsecurity22/presentation/liu-jian

[22] LLVM. 2024. Callbr Instruction. https://llvm.org/docs/LangRef.html#callbr-instruction. https://llvm.org/docs/LangRef.html#callbr-instruction

[23] LLVM. 2024. Inline Assembler Expressions. https://llvm.org/docs/LangRef.html#inline-assembler-expressions. https://llvm.org/docs/LangRef.html#inline-assembler-expressions

[24] LLVM. 2024. The Often Misunderstood GEP Instruction. https://llvm.org/docs/GetElementPtr.html. https://llvm.org/docs/GetElementPtr.html

[25] Kangjie Lu and Hong Hu. 2019. Where Does It Go?: Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 1867–1881. doi:10.1145/3319535.3354244

[26] Sebastian Poeplau and Aurélien Francillon. 2021. SymQEMU: Compilation-based symbolic execution for binaries. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021.* The Internet Society. https://www.ndss-symposium.org/ndss-paper/symqemu-compilation-based-symbolic-execution-for-binaries/

[27] David A. Ramos and Dawson R. Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 49–64. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ramos

[28] Koushik Sen, George C. Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 842–853. doi:10.1145/2786805.2786830

[29] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016.* IEEE Computer Society, 138–157. doi:10.1109/SP.2016.17

[30] 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*, Jiri Schindler and Noa Zilberman (Eds.). USENIX Association, 351–366. https://www.usenix.org/conference/atc22/presentation/sun

[31] 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*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 344–358. doi:10.1145/3477132.3483547

[32] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. 2018. Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 291–307. https://www.usenix.org/conference/usenixsecurity18/presentation/talebi

[33] David Trabish, Timotej Kapus, Noam Rinetzky, and Cristian Cadar. 2020. Past-sensitive pointer analysis for symbolic execution. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 197–208. doi:10.1145/3368089.3409698

[34] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 350–360. doi:10.1145/3180155.3180251

[35] David Trabish and Noam Rinetzky. 2020. Relocatable addressing model for symbolic execution. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 51–62. doi:10.1145/3395363.3397363

[36] Haoxin Tu, Lingxiao Jiang, Xuhua Ding, and He Jiang. 2022. FastKLEE: faster symbolic execution via reducing redundant bound checking of type-safe pointers. 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*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 1741–1745. doi:10.1145/3540250.3558919

[37] Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V. Krishnamurthy, and Paul L. Yu. 2020. UBITect: a precise and scalable method to detect use-before-initialization bugs in Linux kernel. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 221–232. doi:10.1145/3368089.3409686

[38] Yizhuo Zhai, Yu Hao, Zheng Zhang, Weiteng Chen, Guoren Li, Zhiyun Qian, Chengyu Song, Manu Sridharan, Srikanth V. Krishnamurthy, Trent Jaeger, and Paul L. Yu. 2022. Progressive Scrutiny: Incremental Detection of UBI bugs in the Linux Kernel. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022.* The Internet Society. https://www.ndss-symposium.org/ndss-paper/auto-draft-249/

A Appendix

Ethical Consideration

All crashes identified by SyzSpec are being reported to the Linux kernel security team and the respective maintainers. We have provided comprehensive descriptions and detailed bug reports for all bugs. Additionally, if we successfully reproduced a bug, we included a reproducer/Proof of Concept (PoC) to facilitate reproduction. We are collaborating with the maintainers to resolve these bugs and develop appropriate patches. Throughout this process, we have followed the responsible disclosure process and are in the process of requesting CVE identifiers for discovered crashes.

Data Availability

We will open source the implementation of SyzSpec and the generated syscall descriptions to facilitate the reproduction of results and future research. The source code of SyzSpec and the generated syscall descriptions would be available at would be available at https://github.com/seclab-ucr/SyzSpec.