Spork: A `posix_spawn` you can use as a `fork`

Manuel Vögele
Ruhr University Bochum

Christopher Thomas
Ruhr University Bochum

Timo Hönig
Ruhr University Bochum

Abstract

In the over 50 years since its introduction, the `fork` system call has evolved from a simple system call into a slow, complex behemoth that pervades many operating system (OS) primitives and hinders the implementation of superior OS concepts. There exists a multitude of issues with the `fork` system call, and programmers are encouraged to use alternatives such as `posix_spawn`. However, migration to new APIs is not happening, as shown by the analysis in this paper.

To pave the way for new OS concepts, we propose Spork, a flexible emulation layer that eliminates `fork`. Spork enables `fork`-free operating systems while maintaining backward compatibility with legacy software that uses the `fork` system call. Without the need for kernel-level `fork` functionality, Spork also solves many of the issues caused by `fork` for 84 % of existing legacy software in modern Linux distributions.

CCS Concepts

• **Software and its engineering** → **Multiprocessing / multiprogramming / multitasking**; **Interoperability**; *Software performance*; *Software post-development issues.*

Keywords

operating systems, system calls, process creation, POSIX

ACM Reference Format:
Manuel Vögele, Christopher Thomas, and Timo Hönig. 2025. Spork: A `posix_spawn` you can use as a `fork`. In *Workshop on Hot Topics in Operating Systems (HotOS '25), May 14–16, 2025, Banff, AB, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3713082.3730396

1 Introduction

In the early 1970s, `fork` was born out of pure necessity to enable multi-program operation on computer systems. As such, the `fork` system call has been a part of Unix V1 in 1970 and has since been included in all Unix variants—including BSD, Linux, and macOS. The original API design for `fork` was driven by the needs of the command-line shell and the need for simplicity with which the system call could be implemented at the time: It only required 27 lines of assembly [10]. With the inclusion of `fork` in operating system standards such as POSIX, operating systems are now required to implement `fork` if they strive for compatibility with the large body of existing legacy software and applications.

However, `fork` has lost its simplicity. Dennis Ritchie already noted back in 1980 how the initial implementation of `fork` caused unforeseen interactions with other parts of the operating system, which then had to be updated to accommodate `fork` [10]. Baumann et al. [3] discuss how, over the years, the increasing complexity of software and hardware has caused more unwanted interactions, making `fork` slow, error-prone and difficult to implement, to the point where some concepts cannot be realized if `fork` is to be supported.

An alternative to `fork` exists within POSIX: `posix_spawn`. `posix_spawn` combines process creation and execution into a single function call. This allows it to forgo many of the disadvantages that `fork` suffers from, such as having to copy the entire memory of the processes. Baumann et al. conclude their work [3] by appealing to programmers and software developers to stop using and teaching `fork` and starting to use available alternatives like `posix_spawn`.

We have analyzed the developments since then [3] and discuss these findings in this paper. Following the methodology for static analysis of Altidakis et al. [2], we analyzed the software repositories of past and current releases[1] of Ubuntu, a popular Linux distribution, by decompiling all binaries in

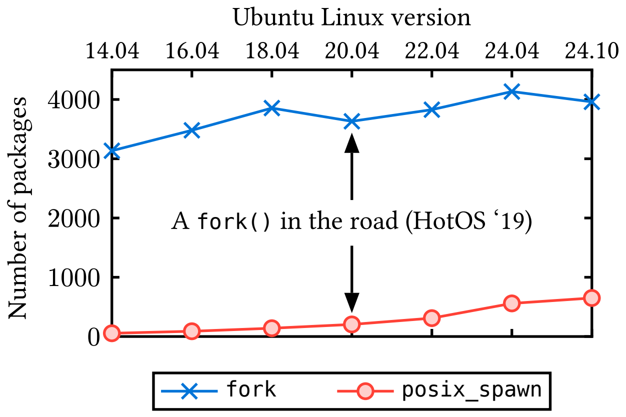

Figure 1: Process creation API prevalence in software shipped with Ubuntu Linux (2014 – 2024).

[1]We analyzed all currently supported LTS versions plus the most recent Ubuntu Linux version, which as of writing is 24.10.

each package and searching their dynamic link table for links to fork and posix_spawn. The results are plotted in Figure 1. While the use of alternatives such as posix_spawn is slowly increasing, and the growing number of packages in Ubuntu is causing the relative number of packages using fork to decrease, new packages using fork are still being added, showing that despite efforts to date, software will continue to depend on fork for a long time to come. If we want to build systems without fork today, we need to take action on the operating-system side.

The contributions of this paper are threefold. First, we provide numbers for usage patterns of fork in current Linux distributions. Second, we classify the revealed usage patterns of fork to better understand and enable us to describe solutions for freeing operating systems of this old cruft. Third, we propose Spork[2] (**sp**awn-f**ork**), an emulation layer that maps fork calls to posix_spawn. We show that contrary to common belief, all calls to fork can be substituted at runtime with specialized calls to posix_spawn, even for processes that perform elaborate setup steps before calling exec or do not call exec at all. All the proposed emulation techniques can be implemented in user space by the libc, allowing OS developers to remove fork from their kernels entirely.

2 The Piercing Tines of fork

The flaws of fork have been known to the operating systems community for years and were detailed by Bauman et al. in their paper "A fork() in the road" [3]. These identified flaws of fork are summarized in Table 1.

fork's API is designed to duplicate the entire state of a process. This is inherently slow, even if copy-on-write is utilized. This causes the Chromium browser to slow down by up to 100 ms when calling fork [9]. As an additional side effect, copying the entire process memory encourages operating systems to overcommit memory on the assumption that the forked process will not modify all of its memory pages. fork also introduces security issues, as the entire process state is shared with the child; some of this state is even retained after an exec. It is the responsibility of the caller to make sure that any state that is not intended to be shared is cleared from the child process after the fork. Frequent spinlock acquisitions during process state copying limit the parallelism of fork. As demonstrated [12], this severely impacts fork's scalability on multicore systems.

Finally, fork is hard to implement, as many operating system components need to consider the possibility of a forking process in their implementation. Even then, applications retain state that is not intended to be copied, such as buffered I/O operations where the buffers contain unflushed data. This means that fork does not compose well with other APIs.

[2]Like the cutlery, Spork is a posix_spawn you can use as a fork.

3 Enter Spork

In order to systematically replace fork, it is first necessary to understand the different usage patterns of fork in detail. To this end, we analyzed the fork patterns of software to determine how programmers use fork. We identified the following three patterns:

P_1 spawning a sub-process by calling exec after fork
P_2 creating worker processes
P_3 creating snapshots of the running process

Spork consists of three components: *Facade*, *Primer* and *uFork*, which are described in detail below. In combination, the components replace fork for all three patterns with posix_spawn[3]. In addition to allowing OS developers to remove fork from their kernels, Spork's Facade can remove several of the showstoppers identified by Baumann et al. [3] for processes using P_1, as shown in Table 1. Since 91 % of applications use P_1 (see Section 4), these improvements are relevant for almost all applications.

Facade does compose and is thread-safe, since the adverse effects of fork only occur in the execution of the duplicated process. Facade never creates such a duplicate (see Section 3.1), preventing these effects entirely. Facade also fixes fork's scaling problem, as copy-on-write is not needed. Similarly, because Facade does not create huge memory areas that may never be used, it does not encourage memory overcommit. Facade avoids memory duplication, likely making it fast, especially when using cached symbolic execution (see Section 5.1), but further experiments have to prove this.

Below, we give an overview over what each of the components of Spork does. A suggested implementation and further discussion of the components is discussed in Section 5 and Section 6, respectively.

[3]While we select posix_spawn as replacement, because it is part of POSIX and therefore broadly available, the presented techniques will work with other process spawning APIs like io_uring_spawn, too.

	fork	Spork
simple to implement	✗	✗
composes	✗	✓
thread-safe	✗	✓
secure API design	✗	✗
fast	✗	(✓)
scalable	✗	✓
no memory overcommit	✗	✓

Table 1: Desirable traits identified by Baumann et al. Spork's Facade offers a solution to most of the identified issues.

3.1 Facade

The goal of applications using fork with the P_1 pattern is to spawn a process that will execute a specified program. They do not care about creating a copy of the currently running process, apart from possibly using it as means to an end to modify the properties of the new process. Facade makes use of this property of P_1 applications. When an application invokes fork, the C standard library libc will not forward this request to the operating system. Instead, it searches for the corresponding exec call. It analyzes which parameters will be passed to the exec call and how the program modifies the process state between fork and exec. Facade then calls posix_spawn in a way that creates the desired process with the desired properties. The fork function will then return the process id of the child process to the parent process. To applications this will appear as if fork, exec and the code in between had been executed, allowing the program to continue as intended. In reality fork has been short-circuited to posix_spawn and no process duplication occurred.

In the case where Facade cannot find a posix_spawn corresponding to the fork, it hands the control over to uFork (see Section 3.3), which can handle such cases.

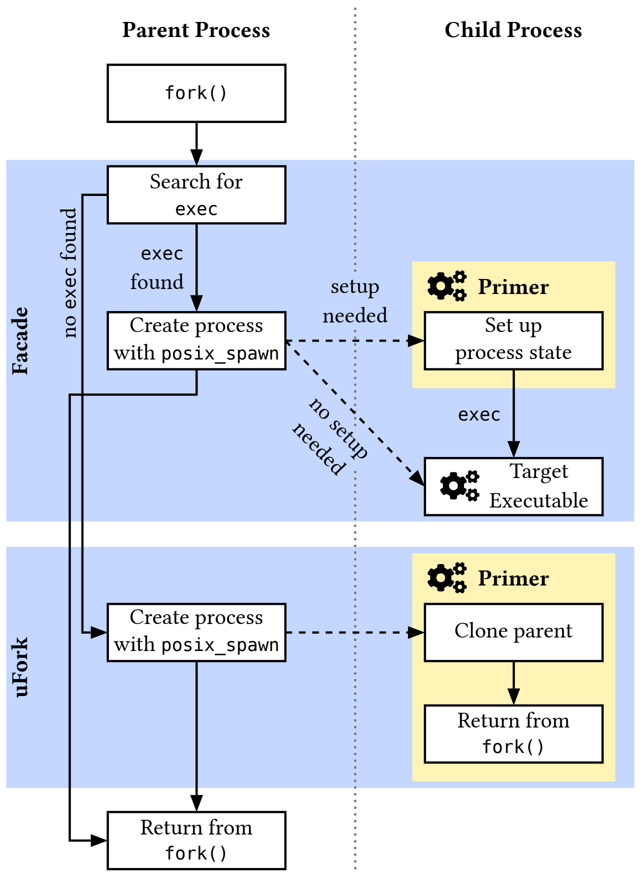

Figure 2: Architecture of Spork. When fork is called, first Facade tries to emulate it with the help of Primer. If this fails, uFork is used as fallback. Gears indicate binary executables.

3.2 Primer

There are some modifications to the process state that are made by P_1 applications that cannot be handled directly by posix_spawn. One example for such a modification is updating the signal handlers of the new process, for which posix_spawn does not provide a parameter. Primer performs these process state transformations whenever Spork needs to change the state in a way that is not natively supported by posix_spawn.

Primer is an executable that is shipped with a Spork-enabled libc. Whenever Facade needs to modify a process in a way that is not supported by posix_spawn, it instructs posix_spawn to execute Primer instead of the desired binary. Primer then acts as a loader for the target executable. Facade informs Primer about the desired process state and instructs it which program shall be executed once the process has been brought into the correct state.

Since Primer is running in the new process, it can execute arbitrary code to put the process in the desired state—similar to how programs modify the new process after a successful fork. It possesses the same capabilities as a forked process, except that it has no access to the memory of the parent process. Facade and Primer thus have to work together closely: Facade must calculate the required process state in the parent process, as access to the parent's memory is likely required as an input for these calculations. It is then Primer's responsibility is to perform the necessary operations to bring the process into the desired state. Once the correct process state is reached, Primer runs exec to hand control over to the target executable.

Primer is being used by both Facade and uFork whenever the native capabilities of posix_spawn are insufficient to reach the desired process state.

3.3 uFork

uFork is being invoked when Facade cannot find the exec corresponding to a fork (see Section 3.1). It is intended as a fallback for the few applications (see Section 4) that use P_2 or P_3. uFork is an implementation of fork in user space. It creates a new process using posix_spawn and uses Primer (see Section 3.2) to transform the child process into a copy of the parent process. Then Primer issues a return statement to resume execution in the child after the fork call.

4 Use-cases for fork

Baumann et al. [3] imply that in the regular case, a fork will be followed by an exec, an assumption that is shared by the design of Spork. To verify whether this conjecture holds true we analyzed the executables installed on a Ubuntu 22.04 installation. We used a static analysis technique, as this allowed us to analyze a large number of binaries without knowledge of their usage or configuration requirements.

The static analysis algorithm leverages the fact that applications usually call a dynamically linked libc, when they invoke fork or a function of the exec-family. A binary that is dynamically linked exposes a list of symbols imported from other libraries. The analyzer can use these imports to identify code locations that call fork. After every fork, the analyzer inspects the code using search_exec from Listing 1. It follows every possible branch until either an exec is found or all branches end in a statement that returns from the function that initially called fork. In the latter case, the analyzer assumes that no exec will follow the fork call.

The results (see Figure 3) show that in 84 % of the executables using fork, all the fork calls lead to an exec call. 7 % of applications call exec only after some of their fork calls, while the remaining 9 % of applications never use exec after fork at all. A manual analysis of applications that invoke fork but not exec revealed that these applications mostly use fork to create worker processes for parallelism. Another use-case for fork without exec that we have been able to identify are applications that use fork for its copy-on-write properties, for example, to quickly create a snapshot of the current process that they can easily return to later.

5 Implementation

This section suggests possible implementations for the components of Spork. The components can be implemented independently and only need to share a common API to communicate with each other. Thus, each of the following chapters details the implementation of each component separately.

```
func search_exec(instruction):
  while (true):
    if (instruction is exec):
      return true
    if (instruction is ret):
      return false
    if (instruction is call):
      if (search_exec(call target)) == true:
        return true
    if (instruction is conditional jump):
      if search_exec(jump target) == true:
        return true
    instruction = next instruction
```

Listing 1: Pseudocode of the algorithm determining if there is a corresponding exec to a fork.

5.1 Facade

The implementation of Facade comes with two main challenges. Facade must be able to locate the exec corresponding to the current fork call and must be able to determine which process state changes occur between fork and exec.

Detecting process state changes. We introduce two implementation variants for Facade: One using symbolic execution and another one employing emulation-based techniques.

In the first variant, when a fork is first encountered, Facade initializes a symbolic execution process, adding an element for each instruction encountered on the way to the exec. Parameters passed to functions that update the process state are marked as output of the symbolic execution. When the exec is reached, Facade has formulas to calculate the input parameters for all the functions that change the state of the process, based on the current process memory. This allows it to infer the target state of the child process. Once gathered for a fork call, these formulas can be cached for faster processing on future calls.

The second variant utilizes a CPU emulator. The CPU emulator executes code that would run in the child. Memory pages accessed by the emulator are copied on demand to avoid duplicating the entire process in the emulator. Calls to the functions that alter the process are recorded together with their parameters. Once exec is executed, the CPU emulator terminates and reports the process changes that occurred during emulation, allowing Facade to derive the desired process state.

Symbolic execution promises much better performance than emulation, especially when a fork call is executed multiple times and caching can be used. However, symbolic execution may not be able to resolve all paths from fork to exec, because there are operations in between that are too complex to capture. In these cases, emulation has the advantage of being able to reliably detect all process changes in any conceivable fork-exec path.

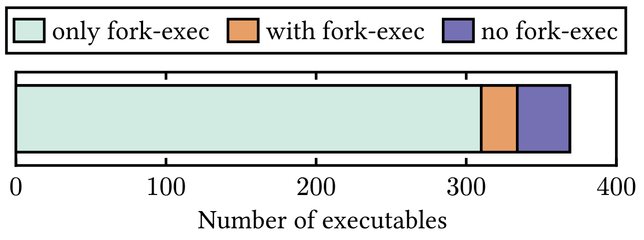

Figure 3: Frequency of fork-exec patterns in Ubuntu executables that contain at least one fork call. In most binaries, every fork leads to an exec (only fork-exec), in some fork never leads to an exec (no fork-exec) and in a few some fork calls lead to an exec while some calls do not (with fork-exec).

Locating exec. The simplest way to locate the exec belonging to a fork is to assume that an exec will follow. Doing so, Facade will start the algorithm that detects the process state changes without prior analysis and imposes a limit on its execution (e.g. a maximum number of instructions or CPU cycles). The algorithm will terminate early, if an exec is found. If the algorithm runs into its execution limit, Facade assumes that no exec will be called.

The method described is easy to implement and fast in cases where an exec will follow the fork—which is true for the vast majority of executables (see Section 4). In cases without exec, however, this simple approach is quite wasteful. To improve performance in cases where there is no exec, Facade could try to detect whether an exec exists. This can be achieved, for example, by an algorithm like the one we used in our analysis in Section 4.

5.2 Primer

Primer is a loader that updates its process to reflect a desired state. It then hands over control to the target program. The main challenge in implementing Primer is to design its API in a way that allows Facade and uFork to transmit all the relevant information about the desired process state that Primer needs to do its job. There are two existing API concepts that Primer variants could be derived from.

One variant is inspired by Windows' CreateProcessA and allows the parent process to specify the desired state for the new process. Primer is then responsible for identifying the operations that need to be executed to transform the process it is running in into the desired state.

The other variant is inspired by the design of fork. Here, the parent process takes care of identifying all the operations that need to be performed. In this design, Primer does not make any decisions, but is merely a gadget that the parent process can use to invoke commands in the child process. The transmission of commands to Primer can happen in a similar fashion to the API of io_uring_spawn: A chain of commands is sent by the parent process that Primer will execute until either exec is called or an error occurs. Alternatively, the parent process can send executable bytecode to Primer, which Primer simply interprets and executes. Using interpreted bytecode over CPU instructions allows Primer to offer a consistent API across different CPU architectures.

Independently of which implementation is chosen for Primer, it will be necessary to open a temporary IPC channel between the parent process and Primer. While some processes will require relatively little setup, other processes will necessitate the transmission of process state information that is impractical to pass via command line arguments, making a more sophisticated communication channel necessary.

5.3 uFork

Using Primer as a tool to manipulate the process state of the child, implementing a uFork becomes feasible. To facilitate this, Primer is started in a special forking mode, where it is instructed not to execute exec after the process setup, but to end its execution with a return statement. During the process setup phase, all necessary information to duplicate the parent process is sent through the IPC channel.

The largest chunk of information that must be transmitted is the memory of the parent process. If offered by the operating system, this will ideally be done by creating copy-on-write mappings. In addition to the memory, other, less obvious process state—like information about open file handles—will be transmitted, too.

Once the process has been copied over, Primer hands the control over to the forked process. After this handover, several conditions must be met: All memory allocated by Primer for its tasks must be unmapped, and all registers must hold the correct values for the execution of the forked process. To accomplish this, Primer needs epilogue code that allows it to clean up after itself. This epilogue code first pushes the register values for the forked process to the stack. Then all memory apart from the epilogue code is being unmapped. Finally, all registers are being popped from the stack. As a final step, Primer executes a return instruction. Since the stack of the forked process is now loaded, this appears to the forked process as a return from the fork function and normal execution can continue.

One last fragment of Primer remains, however: The epilogue code has not been unmapped during this process, since the code cannot unmap itself. One option to solve this is to reserve some free space in the Spork-enabled libc. This reserved space is copied over to the child process during the fork operation and Primer can then write its epilogue code to that location. Once the epilogue calls return, the epilogue code remains in the reserved memory region. However, since this region is never accessed by the forked program, the fork call still continues to function as expected by the application.

6 Discussion

All but two of the identified fork showstoppers (cf. Table 1) can be addressed by Spork's Facade. fork is not a simple operation, and unfortunately there is nothing Spork can do to mitigate this fundamental characteristic. While Spork forgoes the need of process duplication in many cases, the proposed emulation techniques are complex themselves. Especially the use of symbolic execution in Spork's Facade can encounter path explosion issues when working with applications that execute complex code between fork and exec. This problem can be mitigated with techniques like Symbolic Backward Execution [7] and Dynamic Symbolic

Execution [6], which contribute to the implementation complexity of Spork. However, Spork moves the complexity away from the kernel into user space, where it causes less disruption. Moreover, `fork` is insecure, because its API is hard to use correctly. To accurately emulate this API, Spork must reproduce all the pitfalls, and thus Spork inherits the insecurity of `fork`.

The improvements shown in Table 1 do not apply to uFork, unfortunately. uFork resembles a regular `fork` operation, but in user space. As such, it has most of the disadvantages of a traditional `fork`. Since uFork cannot rely on the availability of copy-on-write, it is likely to be (much) slower than a traditional `fork`. However, this is not a major problem. As shown in Section 4, 84 % of binaries only use P_1 and thus, uFork is not required. Also, applications that use P_2 are unlikely to be affected too much by this performance penalty, as worker processes are usually created only once (often when the application is started) and then run for a long time. However, the processes may experience higher memory overhead, as uFork might create copies of memory pages that would exist as a copy-on-write reference indefinitely using a traditional fork. Applications using P_3 will experience the most significant impact from this behavior. Their use of `fork` falls outside of the POSIX specification, which does not make any guarantees about whether `fork` will use copy-on-write or not. The assumption of P_3 applications that invoking `fork` will necessarily trigger copy-on-write behavior will break under Spork, just as it would break if they were to be executed on any operating system that does not implement copy-on-write. These applications will continue to function with Spork, but they will experience a significant slowdown every time they call `fork`.

7 Related Work

Few attempts exist to improve the process spawning situation. Zhao et al. propose a `fork` implementation with improved speed [13]. However, while they address the speed of `fork`, their implementation solves none of the other issues `fork` poses.

Previous works have found ways to implement `fork` on platforms where forking is non-trivial, like on SGX [11] and in microkernels [5, 8], making those platforms compatible with legacy `fork`-based software. These solutions rely on properties of their respective target platforms (e.g. [11] relies on creating a copy of the current enclave, which is an operation that cannot be performed outside SGX). In contrast, Spork aims to be compatible with all platforms that are POSIX compliant.

BUDAlloc [1] is an allocator that will not experience any bugs when being forked in a multi-threaded environment. While this addresses one of the thread-safety issues of `fork`, it cannot compensate for any of the other downsides of `fork`. Moreover, it is very focused on being applied to an allocator and may is not easily transferrable to other components of the operating system.

Finally, Boos et al. [4] propose a new process spawning API that is specifically designed to be used in Rust. While the new API addresses all issues presented by `fork`, it is incompatible with legacy applications.

8 Conclusion

Despite the fact that the downsides of `fork` have been known for a long time and are still being debated in research, our data shows that fork use is still increasing. To enable operating systems to move away from `fork` without breaking legacy applications, we analyze how applications use `fork` and identify three unique patterns.

To finally free operating systems of this old cruft, we propose Spork, an emulation layer that allows applications to use the `fork` system call on operating systems that simply *do not* implement it. To overcome the limitations of `posix_spawn`, we proposed Primer, an application loader that allows fine-grained control over the state of a process that has been created. Using Primer as a foundation, we proposed Facade, an emulation layer that can efficiently map `fork` calls to `posix_spawn`. Finally, Spork provides uFork, a user space implementation of `fork` that allows Spork to cover rare corner cases of `fork` usage in legacy code.

Acknowledgments

This work was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 539710462 ("DOSS"), 502228341 ("Memento") and 465958100 ("NEON") and from the Bundesministerium für Bildung and Forschung (BMBF, Federal Ministry of Education and Research) in Germany for the project SUSTAINET-inNOvAte 16KIS2262.

References

[1] Junho Ahn, Jaehyeon Lee, Kanghyuk Lee, Wooseok Gwak, Minseong Hwang, and Youngjin Kwon. 2024. BUDAlloc: Defeating use-after-free bugs by decoupling virtual address management from kernel. In *Proceedings of the 33rd USENIX Conference on Security Symposium (USENIX Security '24)*. 181–197. https://www.usenix.org/conference/usenixsecurity24/presentation/ahn

[2] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. 2016. POSIX abstractions in modern operating systems: The old, the new, and the missing. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys '16)*. 1–17. https://doi.org/10.1145/2901318.2901350

[3] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. 2019. A fork() in the road. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HotOS '19)*. 14–22. https://doi.org/10.1145/3317550.3321435

[4] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. 2020. Theseus: An experiment in operating system structure and state management. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI '20)*. 1–19. https://www.usenix.org/conference/osdi20/presentation/boos

[5] Haibo Chen, Xie Miao, Ning Jia, Nan Wang, Yu Li, Nian Liu, Yutao Liu, Fei Wang, Qiang Huang, Kun Li, Hongyang Yang, Hui Wang, Jie Yin, Yu Peng, and Fengwei Xu. 2024. Microkernel goes general: Performance and compatibility in the HongMeng production microkernel. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation (OSDI '24)*. 465–485. https://www.usenix.org/conference/osdi24/presentation/chen-haibo

[6] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E platform: Design, implementation, and applications. *ACM Transactions on Computer Systems* 30, 1 (Feb. 2012), 2:1–2:49. https://doi.org/10.1145/2110356.2110358

[7] Peter Dinges and Gul Agha. 2014. Targeted test input generation using symbolic-concrete backward execution. In *Proceedings of the 29th International Conference on Automated Software Engineering (ASE '14)*. 31–36. https://doi.org/10.1145/2642937.2642951

[8] Costin Lupu, Andrei Albișoru, Radu Nichita, Doru-Florin Blânzeanu, Mihai Pogonaru, Răzvan Deaconescu, and Costin Raiciu. 2023. Nephele: Extending virtualization environments for cloning unikernel-based VMs. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys '23)*. 574–589. https://doi.org/10.1145/3552326.3587454

[9] Chromium Project. 2018. Bug 40565873. https://issues.chromium.org/issues/40565873

[10] Dennis M. Ritchie. 1980. The evolution of the UNIX time-sharing system. 79 (1980), 25–35. https://doi.org/10.1007/3-540-09745-7_2

[11] Chia-Che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17)*. 645–658. https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai

[12] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 24th Conference on Computer and Communications Security (CCS '17)*. 2313–2328. https://doi.org/10.1145/3133956.3134046

[13] Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. 2021. On-demand-fork: A microsecond fork for memory-intensive and latency-sensitive applications. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys '21)*. 540–555. https://doi.org/10.1145/3447786.3456258