Loupe: Driving the Development of OS Compatibility Layers

Hugo Lefeuvre
The University of Manchester
Manchester, UK

Gaulthier Gain
University of Liège
Liège, Belgium

Vlad-Andrei Bădoiu
University Politehnica of Bucharest
Bucharest, Romania

Daniel Dinca
University Politehnica of Bucharest
Bucharest, Romania

Vlad-Radu Schiller
The University of Manchester
Manchester, UK

Costin Raiciu
University Politehnica of Bucharest
Bucharest, Romania

Felipe Huici
Unikraft.io
Heidelberg, Germany

Pierre Olivier
The University of Manchester
Manchester, UK

Abstract

Supporting mainstream applications is fundamental for a new OS to have impact. It is generally achieved by developing a layer of compatibility allowing applications developed for a mainstream OS like Linux to run unmodified on the new OS. Building such a layer, as we show, results in large engineering inefficiencies due to the lack of efficient methods to precisely measure the OS features required by a set of applications.

We propose Loupe, a novel method based on dynamic analysis that determines the OS features that need to be implemented in a prototype OS to bring support for a target set of applications and workloads. Loupe guides and boosts OS developers as they build compatibility layers, prioritizing which features to implement in order to quickly support many applications as early as possible. We apply our methodology to 100+ applications and several OSes currently under development, demonstrating high engineering effort savings vs. existing approaches: for example, for the 62 applications supported by the OSv kernel, we show that using Loupe, would have required implementing only 37 system calls vs. 92 for the non-systematic process followed by OSv developers.

We study our measurements and extract novel key insights. Overall, we show that the burden of building compatibility layers is significantly less than what previous works suggest: in some cases, only as few as 20% of system calls reported by static analysis, and 50% of those reported by naive dynamic analysis need an implementation for an application to successfully run standard benchmarks.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0372-0/24/04.
https://doi.org/10.1145/3617232.3624861

CCS Concepts: • **Software and its engineering → Operating systems**.

Keywords: Operating Systems

ACM Reference Format:
Hugo Lefeuvre, Gaulthier Gain, Vlad-Andrei Bădoiu, Daniel Dinca, Vlad-Radu Schiller, Costin Raiciu, Felipe Huici, and Pierre Olivier. 2024. Loupe: Driving the Development of OS Compatibility Layers. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3617232.3624861

1 Introduction

An operating system is only as useful as the applications it can run. Thus, developers of new OSes seeking to gather early performance numbers, to attract open source contributors, early investors, or to transition to real-world use [40, 55] need to provide support for existing applications. Manually porting software [4, 17] is only viable in the short term [51, 52], hence developers of new and existing OSes must provide support for unmodified software by building compatibility layers [5, 6, 10, 11, 18, 20, 28, 30, 31, 43, 45, 50–52, 57, 61, 62, 64] that present applications with interfaces similar to that of popular OSes such as POSIX or the Linux kernel ABI.

Building a compatibility layer represents a non-negligible engineering effort [31, 40, 41, 45, 46, 51, 52, 55] and involves 1) identifying the OS features (system calls, pseudo-files) required for a target application and 2) implementing these features. This process is iteratively repeated for each application to support. In this paper we focus on streamlining 1), the latter being generally OS-specific [31].

We observe that, despite their cost, compatibility layers are often developed in an ad-hoc fashion [31]: there is no systematic approach to determine and prioritize what OS features to develop and when, which applications to support, or to what extent used system calls should be implemented

to achieve a desired degree of support. This results in a significant amount of unnecessary engineering.

Past attempts at streamlining that process leverage static analysis [63] and suffer from its drawbacks, heavily overestimating the set of OS features required to support an application. For instance, while binary-level static analysis identifies that >100 system calls are required to conservatively support the entire superset of operations, configurations, and error handling code in Redis (much of which can be quite rarely used in practice, or simply irrelevant for an early prototype), we find that only 42 are actually needed to reliably pass its entire test suite, and just 20 to run `redis-benchmark`.

Hence, OS designers often fall back to naive dynamic analysis, e.g., using `strace`. These tools fail to take into account common practices used in early OS development to save engineering effort: feature stubbing (returning `-ENOSYS` [3] upon invocation, without implementing the feature), faking feature success (returning a success code without implementing the feature), and partial implementation of complex features [40, 51]. Indeed, in early development, the goal is not to support every feature but rather core functionalities of target applications [31]. For example, we find that more than half of the system calls invoked by Redis running the `redis-benchmark` can be stubbed or faked, and do not need to be implemented to support that application and workload.

We propose a systematic methodology based on dynamic analysis, centered around a novel tool called *Loupe*. Loupe measures, for an application and a given input workload (e.g., a benchmark, test suite), which OS features really need to be implemented and which ones can be faked, stubbed, or partially implemented. Loupe also computes, given an OS under construction and a set of applications and workloads, an optimized development plan to support as many applications as possible with as little engineering effort as possible.

Dynamic analysis comes with its own challenges, in particular the difficulty to scale to numerous applications. This is tackled by designing Loupe to require as little effort as possible to integrate a new application, letting us present results for more than 100 applications in our evaluation. Another challenge is how to detect OS features that can be stubbed, faked, or partially implemented. This is addressed by leveraging Linux's `seccomp` [23] and `ptrace` [19] tracing and interposition facilities to measure what OS features' implementation can be avoided with these techniques.

We run Loupe on 100+ popular applications, and present examples of optimized Linux compatibility layer development plans obtained with Loupe for three OSes under construction [5, 10, 45] with various levels of existing support for the Linux system call ABI. We further measure the engineering effort savings obtained by using Loupe to drive the development of compatibility layers. Taking half the applications supported by OSv [43], Loupe reports that only 37 system calls are required to run them, vs. 92 for our estimation of the non-systematic process followed by OSv developers, and 142 for a process driven by `strace`-based dynamic analysis.

We study Loupe's Linux API usage measurements for our set of applications. This analysis brings many new insights. We demonstrate that the minimal effort needed to provide compatibility is significantly lower than that determined by previous works using static analysis [63]. Our study shows that as much as 40-60% of system calls found in application code do not need implementation to successfully run meaningful workloads, including full test suites. We also find that many applications are resilient to stubbing, faking, and partial implementation of OS features. We investigate the reasons behind it, and the impact of such practices on application performance and resource usage. Finally, we study how the C library influences OS feature requirements.

In all, this paper makes the following contributions:

- A novel methodology to measure the minimum set of OS features that need implementation for a compatibility layer to support a set of applications and workloads, with the aim of minimizing development effort.
- Loupe, a tool able to derive, for a given OS and target applications, an optimized OS feature support plan to run as many apps as possible, as early as possible.
- A demonstration of the engineering effort savings obtained with Loupe, with examples of optimized feature implementation plans for 11 OSes under development.
- An analysis, using Loupe, of the OS features required by a set of applications showing the lack of precision of past approaches and investigating common development practices in compatibility layer development.

Loupe is actively used in Unikraft [45], an open-source commercial OS, and has attracted the attention of several others. Overall, this study brings a message of hope: contrary to what past work seems to suggest, a good degree of compatibility with existing applications can be achieved without immense engineering, provided we follow a focused and methodical approach. Loupe and our results are available online[1] under an open-source license.

2 Motivation and Approach

Building Compatibility Layers for New OSes. Compatibility layers can be found in mature OSes for interoperability reasons [11, 18, 28, 50, 61], but also in a plethora of new/prototype/research OSes [5, 6, 10, 20, 30, 31, 43, 45, 51, 52, 57, 62, 64]. Providing support for existing applications in these OSes is generally crucial [45, 47, 51, 52] to gather early performance numbers, to attract open source contributors, early investors, or transition to real-world use. Manually porting software [4, 17] is not sustainable in the long run, nor does it scale to a large amount of applications [51, 52]. Hence, the developers of many new OSes resort to implementing compatibility layers. Even considering OS models

[1] https://github.com/unikraft/loupe / https://github.com/unikraft/loupedb

that choose to drop application compatibility for other gains (e.g., performance), it is not uncommon to see Linux versions of these models appear a few years after the seminal paper, with claims of stronger compatibility, e.g., Popcorn Linux [33] for the multikernel [34] or Graphene, Lupine and UKL [46, 58, 62] for the unikernel [48, 49].

Building a compatibility layer is seen as a non-negligible engineering effort [31, 40, 41, 45, 46, 51, 52, 55, 63]. We investigated the compatibility layers present in several open-source OS projects [5, 6, 10, 20, 30, 31, 43, 45, 51, 52, 62, 64]. Based on this study, and on our multiple years of experience providing Linux/POSIX compatibility in research OSes, we observe that compatibility layers are built in an ad-hoc, non-systematic ("organic") way: developers select an application to support, determine the OS features it requires, and implement them [31]. That process is repeated for each target application. Because so many projects undergo the task of building compatibility layers [5, 6, 10, 11, 18, 20, 28, 30, 31, 43, 45, 51, 52, 61, 62, 64], there is a need for tools to streamline that process. The corresponding effort consists in 1) identifying OS features required by target applications and 2) implementing these features. The latter task is known to be very specific to the new OS considered [31], and can hardly be streamlined. We show in this paper that the former task, identifying and prioritizing what OS features to implement, can be systematized and optimized. Next, we motivate our method by explaining how past and current approaches are suboptimal.

Limitations of Static Analysis. Existing approaches measuring the usage of OS features by applications often rely on static analysis [32, 35–38, 51, 52, 54, 63, 65]. Static analysis is comprehensive: the set of features identified for an application includes all the ones that *may* be invoked at runtime, under any possible workload, operation, or configuration, and traversing any possible error path. Alas, static analysis is also conservative and yields many false positives: it overestimates OS features that will actually be invoked at runtime.

Static analysis can be performed on application sources or binaries. Binary analysis [32, 36, 37, 51, 52, 63] scales well to a large number of applications because it targets a common format (e.g., ELF binaries). However, it suffers from a lack of precision due to the difficulty of extracting information from a binary [37]. Such issues may be alleviated with source-level analysis [38, 65], which is however not a panacea: it is language-specific, making it difficult to scale to many applications written in different languages.

Tsai et al. [63] measure, using static binary analysis, the system call usage of the entire set of applications from an Ubuntu distribution. The study concludes that to support 100% of the distribution's packages, 272 system calls need to be implemented. That number goes down to 81 system calls for the 10% most popular applications. These results suggest that a large implementation effort would be required for an OS aiming at supporting even a few applications. As we demonstrate in the evaluation, both source- and binary-level approaches significantly overestimate the OS features required by an application to run popular workloads. This is due to dead or unexecuted code, and the difficulty or impossibility to statically determine runtime-level information (e.g., memory content such as function pointers). Although all of these system calls would likely need to be implemented in a production-grade general-purpose OS, these numbers remain an upper bound of limited usefulness for OS designers in earlier development stages.

Limitations of Naive Dynamic Analysis. Dynamic analysis too has well-known drawbacks. Its precision depends on the coverage of the input workload run during the analysis: if it is too low, some required OS features may not be identified. It is also harder to fully automate, as there is a variable amount of manual effort required for each application to analyze (e.g., selecting an input workload). In this paper we refer to using a tool such as `strace` [24] to trace OS features invoked by an application, as *naive* dynamic analysis. The main drawback of naive dynamic analysis is its failure to consider two techniques commonly used in early OS development [40]:

- Feature *stubbing*: not implementing the feature and returning an error code (`-ENOSYS`: "Not Implemented" [3]) to the application when it invokes the feature.
- *Faking* feature success: not implementing the feature and returning a success code (typically system-call specific) to the application upon invocation.

The two examples below are extracted from the source code of the HermiTux unikernel [51], where the `sigaltstack` system call is stubbed, and the `mprotect` system call is faked:

```
long sys_sigaltstack(const stack_t *ss, stack_t *oss) {
  return -ENOSYS; // stubbed: not supported
}

long sys_mprotect(size_t addr, size_t len, uint64_t prot) {
  return 0; // faked: pretending success
}
```

Many applications are resilient to the failure of OS features [40, 59] and will run correctly when stubbing and faking. In this study, we show that many invoked OS features can avoid being implemented through these practices in the development stages of an OS. This highlights the importance of faking and stubbing as an engineering practice: without it, showcasing a particular application use-case for a new OS concept would take significantly longer, or even be unattainable for a small-scale research project. Despite of this, naive dynamic analysis does not typically consider stubbing and faking. Naive dynamic analysis traces all features and sub-features invoked by an application, independently of the fact that they can be stubbed/faked or not for a given workload. Thus, OS designers typically rely on trial and error to determine which features they need to implement first, and which ones they can fake or stub.

When to Stub or Fake and When not To? The reliance on stubbing and faking as a development practice in transitional OS development stages introduces a pivotal question: *when to stub and fake, and when not to?* This question is driven by two sources of concern:

- Impacting stability. Although guaranteed stability of entire applications is not a primary goal in the early development stages of an OS, faking and stubbing must not impact the stability of relevant application features. Failing to do so would negate the benefits of faking and stubbing by creating an additional debugging cost.
- Impacting performance metrics. Early OS prototypes must be comparable to full-fledged mainstream OSes; this is especially true for research OSes. Impacting performance metrics by faking or stubbing would defeat the purpose of the OS prototype by making it impossible to fairly evaluate its performance advantage or cost. For instance, stubbing or faking an expensive and relevant security feature may provide an unfair advantage to an early OS prototype vs. a full-fledged OS that implements it.

Non-systematic, trial-and-error-based approaches are especially prone to fall into stability and performance pitfalls. Although important, these concerns have been little discussed by works which rely or relied on faking and stubbing.

Breaking the Status Quo with Loupe. We aim to propose a systematic and adaptive method to determine which OS features to implement first. Our goal is to help OS designers transition from *no support* towards *full support* to run as many applications as early as possible.

Overall, dynamic analysis is better suited to the problem we aim to solve, being able to evaluate the concrete impact of both stubbing and faking, and providing fine-grain, per-workload results. The coverage issue of dynamic analysis is a nonproblem in our context: in early development phases of an OS, the goal is not to support every feature but rather core functionalities of target applications [31], which are easily exercised by standard benchmarks and test suites. Dynamic analysis is precisely suited because it is adaptive: as support progresses, workloads can be extended to cover more and more application features. We are left with two challenges: the difficulty to 1) scale to numerous applications, and to 2) detect OS features that can be stubbed, faked, or partially implemented without impacting stability or performance for relevant application features. As we detail in §3, we solve the former by designing Loupe to require as little effort as possible to integrate a new application (at most writing a Dockerfile and test script), letting us present results for 100+ applications in §5. We address the latter challenge by leveraging Linux's `seccomp` [23] and `ptrace` [19] facilities to measure what OS features can be stubbed, faked, or partially implemented for a given application workload. To maintain stability and performance metrics for the evaluated workload, Loupe replicates the analysis several times in containerized environments, and offers a framework for identifying performance regressions on various generic and application-specific metrics. We further discuss stability in Sections 5.2 and 6, and performance impact in Section 5.3.

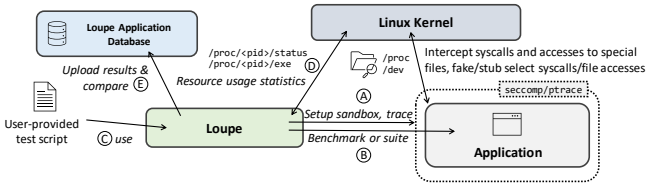


Figure 1. Loupe architecture diagram.

3 Loupe: Accurate Run-time Analysis of OS Feature Usage

To accurately quantify what OS features are needed to gradually support a set of popular Linux applications, and to measure to what extent static and naive dynamic analysis overestimate these requirements, we built *Loupe*. Loupe is a dynamic analysis tool that hooks into each OS feature used by applications at runtime, analyzing the application's behavior as it simulates different degrees of compatibility. Unlike existing naive dynamic analysis tools (e.g., `strace`), Loupe is built as a framework specifically meant for OS feature support analysis. It supports identifying what system calls and pseudo-files are used by a given application, and determining which can be faked, stubbed, or partially implemented. Loupe focuses on reliable and reproducible results, and supports easy integration into existing build systems and complex test suite systems. Finally, Loupe can process measurement data for a set of applications and output targeted OS feature support plans. We implemented a prototype of Loupe in 2.5K LoC of Python, and 500 LoC of C (used for `seccomp` and `ptrace` hooks). In this section, we summarize the functioning and architecture of Loupe (§3.1), detail our approach to evaluate the success and performance of application runs (§3.2), and conclude with details on various aspects of Loupe (§3.3).

3.1 Loupe Overview

Using Loupe to measure the OS feature usage of a new application is straightforward. Users provide Loupe with 1) the application binary whose OS feature usage needs to be measured, and 2) a per-application *test script*, responsible for providing external input to the application, and measuring the performance and success of each run. Loupe operates on binaries, so there are no language or compiler restrictions on either the application or the test script. Provided this, Loupe evaluates the OS feature usage of the application feature by feature. Each used system call and pseudo-file is tested for one of two modes in separate runs: 1) *stubbing* the system call, i.e., do not run the system call and return `-ENOSYS` or 2) *faking* it, i.e., return a success code without running the system call.

Once all OS features have been tested, a final run confirms that the analysis performed on a per-feature basis holds when all features are considered. In the event of a failure, users can use Loupe to alter subsets of system calls to find the culprits, a process which could be automated in future works.

We now detail the behavior of Loupe *for each run*, as visualized in Figure 1. Loupe first simultaneously sets up tracing and sandboxing (Ⓐ on Figure 1) and starts the application (Ⓑ) using the `seccomp` [23] and `ptrace` [19] Linux tracing and interposition facilities. Once the application has been started, Loupe uses the test script to feed the application with inputs (e.g., generating client requests for a server application) and gather performance numbers (Ⓒ), all the while recording data on resource usage via `/proc` (Ⓓ). Using the hooks set up in Ⓑ, Loupe intercepts each system call invoked by the application, and tests it for one of the two previously described modes. At the end of the run, Loupe determines the success of the application using the return code of the test script (more in §3.2). Accesses to pseudo-files are hooked and disabled, stubbed, or faked similarly by catching system calls from the `open` family (see §3.3).

In order to maximize the reliability and reproducibility of the results, each analysis is performed multiple times in containerized replicas, and the result of the analysis is conservatively updated to take all results into account. The number of replicas (*3* by default) and whether they run in parallel (*no* by default) can be configured to suit different applications, accuracy needs, and available hardware.

Finally, OS developers can specify the system calls supported by their OS in CSV form, and Loupe will recommend which OS features to implement, stub, or fake, to support a set of applications selected among those measured by the tool. Loupe will prioritize the list of features to indicate which should be implemented first in order to support as many applications, as early as possible. Loupe's measurements can optionally be shared in an online database (Ⓔ).

3.2 Evaluating Success and Performance

Loupe builds on the premise that users are able to describe a workload that they want to support for a given application. Loupe then tells the user which precise set of system calls they have to support (and how) to be able to run that workload reliably, i.e., over multiple runs without observable functional and non-functional issues.

Describing Workloads. Workloads describe the feature set that must be supported in a given application. Loupe users express workloads in *test scripts*, responsible for supplying external input, if required by the application, and detecting the success of a run.[2] Test scripts may materialize any type of workload: simple health checks (e.g., for a web server: can the application process a simple HTTP query?), benchmarks, test suites, or even fuzzing. If specific error cases or application features must be supported, then the test script must also exercise them as part of the run. In this paper we explore health checks, benchmarks, and test suites. Each workload may be relevant at different stages in the development life cycle of a new OS. Workloads correspond to different levels of guarantee of application stability; they can be evolved as support progresses, until complete compatibility can be provided to ensure stable application behavior in all circumstances.

Defining "Success". A run is considered successful when the application terminates and the test script exit code indicates success. Crashes, or unresponsiveness are considered as generic failure signs. The notion of generic failure can be extended to unusual resource usage, or even unusual filesystem or network usage, which Loupe can observe without understanding application semantics. Generally however, the notion of success or failure is application-specific and inseparable from the workload itself: e.g., outputs on the standard output/error channels or logs that do not correspond to normal application behavior, or altered performance (e.g., throughput, latency, packet loss rate). Application-specific success criteria must be evaluated by the test script. An example of a test script for Nginx benchmarked with `wrk` is shown below:

```
#!/bin/bash
# [...] omitted helpers (including is_failed and grep_req_per_sec)

b=$(wrk http://localhost:8080 -d10s | grep_req_per_sec)
if [[ $(is_failed $b $?) ]]; then exit 1;
else echo $b; fi
```

`is_failed()` is responsible for detecting failures, left out above for space reasons. When performing a simple health check, the function verifies that the throughput is non-zero.

We implemented detection of unusual resource usage and performance in our prototype. Loupe records application resource usage (maximum resident size and open file descriptors) via `/proc` and compares results over multiple runs when stubbing or faking. Similarly, when performing a performance benchmark, test scripts return the relevant performance number (which can be any application-specific performance metric), and Loupe ensures that the performance does not incur a statistically significant variation from the full-fledged baseline. Together, resource usage and performance checks can provide insights into the impact of stubbing or faking features, and particularly increase the confidence on the correctness (or incorrectness) of faking and stubbing. We further discuss performance and resource usage in §5.3.

3.3 Loupe in Detail

We now discuss various aspects of Loupe that are relevant in this paper: supporting vectored system calls and pseudo-files, making Loupe easy to use in many applications, how long Loupe analyzes take, and sharing analysis results.

[2] Some programs do not require input and determine success by themselves or via a wrapper script (e.g., test suites). If so, the test script is *practically included* in the application and need not be passed separately. Loupe supports this. Since this is similar to the general case, we do not further discuss it here.

Vectored System Calls. Identifying OS features at the granularity of an entire system call is sometimes too coarse, considering vectored system calls (e.g., `ioctl`, `fnctl`) and system calls with several functionalities that may be partially implemented in a compatibility layer (e.g., `mmap`, or `madvise`). In such cases, Loupe can also disable, stub, and fake system calls based on *individual system call parameters*, allowing users to easily explore partial implementations at a fine granularity. The output is a list of system calls along with their used sub-features, and whether they can be faked or stubbed.

Pseudo Files. Part of the Linux API is offered through pseudo-files such as `/dev/random`. Loupe is able to detect usage of such special files by pattern matching the arguments of certain system calls (e.g., `open`, `openat`) against paths (e.g., `/dev`, `/proc`). Loupe can also fake or stub system calls accessing these files, enabling users to track which special files require an implementation for applications to run.

Testing Framework Integration. Dynamic analysis tools can be difficult to integrate in application testing frameworks. Test suites, for instance, may start the application multiple times, from complex scripts, from different call points [8, 22]. Calling a naive analysis tool like `strace` requires manual changes, along with additional logic to gather and merge results obtained from the multiple runs triggered by the test suite. Calling the tool on the test suite itself (e.g., `strace make test`) is not effective either, as the test suite may call external tools whose OS feature usage is not part of the application's. For instance, the Ruby test suite makes extensive calls to `git` to set up test environments; the OS feature requirements of `git` should not be included into the application's. We tackle this problem with a whitelist system: when run on a wrapper (e.g., a test suite), users can specify which binaries are that of the application and should be considered in the analysis. Loupe then tracks all children processes, checking the binary path upon `exec`, to ignore any system call originating from a binary that does not correspond to the specified one(s). This allows, for instance, unmodified analysis of test suites run via `make test`; Loupe simply executes the Makefile and only considers system calls executed by the appropriate binary.

Debhelper Integration. To further simplify running Loupe on many applications, we integrated Loupe into the Debhelper [1] Debian package build system. Loupe can build Debian packages and run on the package's `dh_auto_test` [2] rule which, if provided by the package, executes the target application's test suite. Combined with the previous technique, which Loupe can leverage by listing the package's binaries, we can significantly reduce the cost of testing applications. Running Loupe on the Lighttpd, Memcached, and webfsd test suites, for instance, is fully automated this way.

Loupe Run Time. The runtime of a full Loupe analysis is $(2+(2*t*s))*\lceil\frac{r}{p}\rceil$ with: t the application workload runtime, s the number of distinct system calls (and pseudo-files, if enabled in the analysis) executed by the application under the specific workload, r the number of replicas, and p the number of replicas executed in parallel. 2+ corresponds to the initial run to discover executed system calls, and to the final run to confirm the analysis. 2∗ corresponds to the "stubbing" and to the "faking" run for each system call. The overall runtime is therefore dominated by the length and complexity of the application workload; it varies from about 4 minutes for a fast Nginx health check, to 50 minutes for the Lighttpd test suite, and 1-1.5 days for the SQLite test suite (by far the largest we encountered, running *millions* of tests [8]). These run times are reasonable: porting cost for a single application often reach multiple weeks or months in early OS development stages [45] and, as we expand next, this is a one-time cost.

Sharing Loupe Results. Thanks to the techniques described previously, Loupe test scripts are easy to write; 2-30 minutes on average according to the expertise of the user, most of it spent on understanding how to run and test the application. The main barrier to running Loupe on a large number of applications is runtime. Nevertheless, as we described previously, the results are final for a fixed build of the software, its workload, dependencies, kernel, and test script. To leverage this, we have set up a shared online database that can be populated and looked up by any individual running Loupe or interested in its results. Loupe can automatically submit results to the database along with metadata (Ⓔ in Figure 1). We envision that in the long run, this database will contain results for a wide range of applications, helping OS and application developers to study OS features usage patterns, build compatibility layers, and more, without even the runtime cost mentioned previously.

4 Loupe: OS Feature Support Guide

For space reasons, we set aside pseudo files and focus on system call support, as it represents the majority of the engineering effort in building compatibility layers [10, 45, 51].

4.1 Examples of Support Plans

We ran Loupe on a total of 116 applications with various workloads including standard benchmarks (e.g., `wrk` for web applications, `redis-benchmark`). We choose a selection of representative applications from OpenBenchmarking.org [16], as well as various other sources [4, 17, 25][3]. Leveraging these measurements, Loupe guides the process of developing a compatibility layer by giving a prioritized list of system calls to implement/stub/fake. Specifically, given (1) the state of a partially Linux-compatible OS in terms of system calls supported (a simple text file with one line per supported system call) and (2) a set of target applications to support, Loupe can output an incremental support plan listing the order in

[3]Our artifact includes a list of all applications and support plans for 11 OSes: https://github.com/unikraft/loupedb/blob/staging/ASPLOS24-supp.pdf

Table 1. Step-by-step support plans for 3 OSes.

Step	Implement	Stub	Fake	Support for…
Unikraft (commit `7d6707f`, supports 174 syscalls)				
0	-	-	-	(12 apps)
1	290	273, 218, 230	-	+ Memcached
2	218	-	-	+ H2O
3	283, 27	186	-	+ MongoDB
Fuchsia (commit `5d20758`, supports 152 syscalls)				
0	-	-	-	(10 apps)
1	33	273, 302, 105	-	+ Lighttpd
2	302	230	-	+ Memcached
3	-	99, 222, 223	-	+ HAProxy
4	105	40	128, 99, 27	+ Nginx
5	128, 99, 27	-	-	+ MongoDB
Kerla (commit `73a1873`, supports 58 syscalls)				
0	-	-	-	(4 apps)
1	56, 257, 54	(17 syscalls)	47	+ Httpd
2	10	-	302	+ Weborf
3	8, 21, 87	-	25	+ SQLite
4	232, 233, 302	(9 syscalls)	288, 213	+ HAProxy
5	17, 213, 262	95	-	+ Redis
6	291	105, 106, 116, 293	-	+ Lighttpd
7	288, 290	32	102	+ H2O
8	46	230	-	+ Memcached
9	105, 18, 53, 106	40	92, 130, 107, 273, 116, 157	+ Nginx
10	104, 107, 108, 102	-	-	+ Webfsd
11	128, 99, 229, 27, 73, 202, 283	131	137	+ MongoDB

which missing system calls should be implemented/faked/stubbed in order to enable compatibility with a maximum of applications as early as possible.

We enabled Loupe to generate support plans for all 116 applications we measured, for 11 OSes under development: Unikraft [45], Google Fuchsia [5] and Zephyr [30], Kerla [10], HermiTux [51], Google gVisor [6], Graphene/Gramine [7, 62], FreeBSD Linuxulator [11], Browsix [56], OSv [43], and Linux nolibc [60]. To illustrate this functionality, we present here a subset of these results (for space reasons): we consider recent versions of 3 OSes: Unikraft, Fuchsia and Kerla, and a target set of 15 popular cloud applications. The support plans are presented in Table 1. The number of steps to reach support for all 15 apps is directly linked to the maturity of the OS: Unikraft for example has initial support for 12 applications and requires only 3 steps to reach full support, while Kerla, with initial support for only 4 applications, requires 11 steps. Loupe's incremental support plans optimize the development of compatibility layers by breaking down the effort into small steps (>80% of which requiring to implement 1-3 system calls), unlocking support for an application after each step. The support plans in Table 1 target a small set of applications for space reasons. Full support plans for each of the 11 OSes we target, for all 116 applications in our database, are larger: 35 steps for Fuchsia, 32 for Unikraft, and 79 for Kerla.

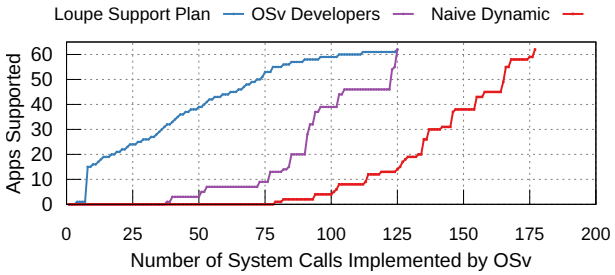


Figure 2. Evolution of the number of applications and system calls supported by OSv assuming 1) a support plan generated with Loupe, 2) organic development based on git history, 3) measurement with naive dynamic analysis without stubbing/faking. Higher values indicate more applications supported for the same effort.

4.2 Engineering Effort Savings

To estimate the engineering effort savings that an OS project would enjoy while building a compatibility layer with Loupe rather than in an ad-hoc, organic fashion, we designed the following experiment: we select a large set of 62 applications supported by a popular experimental OS, OSv [43], from the OSv-Apps repository [17]. We then estimate the order in which these applications were organically supported by the OS. For that we use git metadata to track the creation date of the folder corresponding to each app in the repository. We then derive from the order in which applications were supported, the organic order in which system calls had to be implemented by OSv developers. Because stubbing/faking OS features are well-known practices [40], and because there are traces of their usage in OSv's codebase [53], we assume that OSv developers used stubbing and faking as much as possible. We can then derive, in chronological order, the number of system calls that were implemented by OSv developers, and the evolution of the number of supported applications. We also compute these numbers for a hypothetical optimized compatibility layer development process that would be guided by Loupe's support plan, which would also take stubbing/faking into account, as well as a naive approach that would implement every system call traced by dynamic analysis, without stubbing/faking.

These results are presented on Figure 2. As one can observe, Loupe would have heavily optimized the process of implementing OSv's support for the target application set, leading to more applications supported earlier and with less engineering effort vs. our estimation of the organic process undertaken by OSv's developers. For example, to support half (31) of the applications, with Loupe only 37 system calls need to be implemented, vs. 92 for the organic process. The naive method relying on dynamic analysis without stubbing/faking requires even more engineering effort: to reach 31 applications, 142 system calls would need to be implemented.

Our method to estimate engineering efforts makes a few simplifications. The real order in which applications were supported by OSv is likely not exactly that of folder creation in the OSv-Apps repository. We repeated the study using the date of the *last commit* in each application's folder to determine the order; results were similar. The effort to implement system calls is also variable according to which system call is targeted: the x-axis in Figure 2 is non-uniform since not all system calls have the same implementation cost. However, we believe these results provide a sufficiently solid estimation of the engineering effort reduction that Loupe can bring to demonstrate its usefulness.

5 Analyzing the Linux API with Loupe

Here we study the Linux API usage results obtained using Loupe for the 116 applications considered in our study. We aim to answer the following research questions:

- How important is the accuracy gap between Loupe's method vs naive dynamic analysis (`strace`) and static analysis?
- When building a Linux compatibility layer, which system calls must be implemented, and which ones can be commonly faked or stubbed? What is the absolute minimum set of system calls that must be implemented for a test suite to correctly run?
- What are the most important system calls, i.e., the ones whose implementation is required by most applications?
- Why can some system calls be faked or stubbed? Does it impact performance or resource usage metrics?
- How much do the system call requirements of applications and standard libraries evolve over time?

For space reasons, we concentrate on system calls and set aside results regarding special files and vectored system calls.

5.1 Analysis Method: Static vs. Dynamic

Loupe vs. Naive Dynamic Analysis. We computed the *API importance* of each system call as reported by Loupe and by naive dynamic analysis. API importance [63] represents the probability that in our 116 applications data set, a system call is required by at least one application in that set. A system call is defined as required for an application if it is traced with dynamic analysis, and if it is traced and can neither be stubbed nor faked with Loupe.

Figure 3 visualizes our results. They show that naive dynamic analysis severely overestimates the amount of system calls required to support applications. Loupe reports a total of 148 system calls requiring implementation to support 100% of our 116 applications, vs. 180 system calls for a naive analysis. The 25 most commonly required system calls are present in more than 80% of the applications with Loupe, and in less than 50% with naive dynamic analysis.

Loupe vs. Static Analysis. We faced scalability issues when trying to apply binary- and source-level static analysis tools to our large data set of 116 applications. There exists

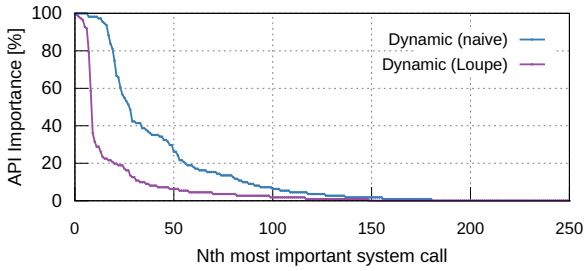


Figure 3. API importance for dynamic analysis with Loupe and a naive approach (= no stubbing/faking).

no source-level tool able to identify system calls for all the relevant programming languages. We also attempted to run several binary-level tools and experienced a high level of failures (close to 50%) skewing the results. Hence, we fall back on selecting a subset of applications from our data set for comparison between static analysis and Loupe.

We select 7 popular cloud applications that support standard benchmarks and ship with comprehensive test suites: Redis, Nginx, Memcached, SQLite, HAProxy, Lighttpd, and Weborf. To gather results for static analysis we use the source- and binary-level tools made available by Unikraft [26, 27]. Figure 4 details the amount of system calls identified in each application by each method. Both static analysis techniques severely overestimate the number of system calls actually needed to run the benchmarks and test suites. The minimum number of system calls identified by Loupe as required for these applications varies around 20 for benchmarks, and 20-40 for test suites. Both static binary and source analysis methods report numbers that are generally between 5x and 2x higher. For example, on Redis, binary-level static analysis identifies 103 system calls vs. 68 dynamically traced ones from the test suite, and Loupe further indicates that more than a third of these can be stubbed/faked. This observation can be generalized to all other applications. Overall these results show that the effort to provide comprehensive support of core features and even full test suites is much lower than suggested by previous work based on static analysis [63].

Figure 5 details which system calls are detected by the various analysis techniques when applied to the 7 applications running benchmarks. Once again the overestimation of static and naive dynamic analysis is clear, compared to the results obtained with Loupe. Regarding static analysis, operating on the binary only yields more system calls compared to targeting the sources. Concerning dynamic analysis, a non-negligible amount of system calls can be stubbed/faked, confirming the benefits of Loupe vs. naive dynamic analysis. We investigate faking/stubbing more in details next.

Insight: Static and naive dynamic analysis both highly overestimate the engineering effort needed to build a compatibility layer for a target set of applications.

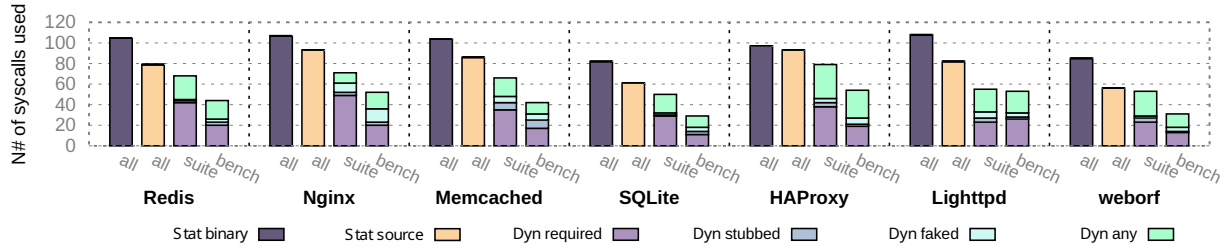
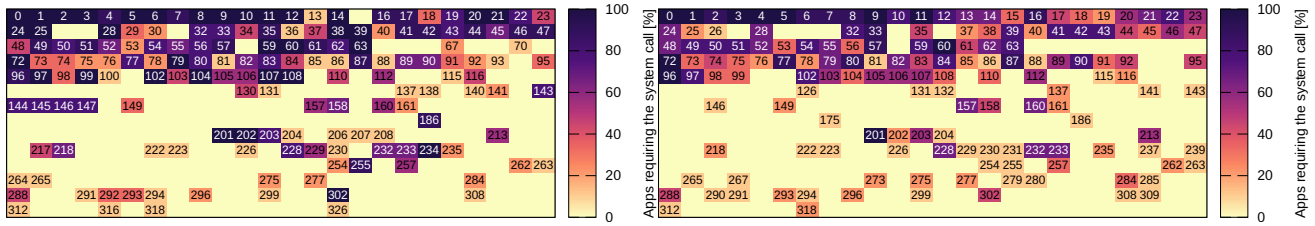


Figure 4. Number of system calls statically identified and dynamically traced by Loupe for applications running standard benchmarks (*bench*) and test-suites (*suite*). Traced system calls are broken down into those that can be stubbed, faked, either faked or stubbed (*any*), and those that can neither be faked nor stubbed (*required*).



(a) Static analysis, binary level.

(b) Static analysis, source level.

(c) Dynamic analysis, traced.

(d) Loupe's dynamic analysis, required (excluding stubbed/faked).

Figure 5. System calls identified by static binary, static source, naive dynamic *traced* (all system calls detected), and Loupe's dynamic *required* (those that cannot be stubbed/faked). Each box represents a Linux system call and its number.

5.2 Resilience to Stubbing and Faking

As visualized in Figure 4, we find that, on average, the proportion of invoked system calls that can be stubbed or faked is 46% for test suites (ranging from 31% for Nginx to 58% for Lighttpd), and 60% for benchmarks (from 51% for Lighttpd to 65% for HAProxy). This shows that the effort required to provide strong support of core features (i.e., those covered by test suites) for these popular applications is certainly lower than suggested by previous work, and is even lower when considering support for benchmarks only (needed for evaluation in research papers). The difference between Figure 5c and 5d clarifies this, highlighting which system calls can commonly be stubbed and faked. We observe broadly two categories:

- **Low range system calls (system call ID ~< *150*)**, representing the majority of system calls detected by all analysis methods. This is unsurprising as these system calls correspond to core services that have been present in the Linux feature set for a long time, such as basic network system calls (`bind`, `accept`, etc.).
- **Higher range system calls (ID ~> *150*)**, where a small set of popular system calls are invoked corresponding to more modern but prominent functionality concerning multithreading (`futex` – 202, etc.), scalable I/O (`epoll` family – 213, 232, 233), as well as new variants of core system calls (`openat` – 257, `prlimit64` – 302, etc.)

Though system calls from both categories can be stubbed or faked, system calls with higher numbers are better candidates: out of the lower half of used system calls (46 system calls with number < *63*), 13 system calls can always be stubbed vs. 30 for the upper half (46 system calls with number > *63*). This is because these map to more recent, generally less critical functionalities; we expand on this next.

Insight: Though applications may invoke many system calls, many of them can be stubbed or faked to run popular workloads.

```c
if (getrlimit(RLIMIT_NOFILE, &limit) == -1) {
  serverLog(LL_WARNING, "Unable to obtain the current NOFILE limit,"
  "assuming 1024 & setting the max clients config accordingly.");
  server.maxclients = 1024 - CONFIG_MIN_RESERVED_FDS;
}
```

(a) Stubbing-resilient Situation (Redis).

```c
if (prctl(PR_SET_KEEPCAPS, 1, 0, 0, 0) == -1) {
  ngx_log_error(NGX_LOG_EMERG, cycle->log,
  ngx_errno, "prctl(PR_SET_KEEPCAPS, 1) failed");
  exit(2); /* fatal */
}
```

(b) Faking-resilient Situation (Nginx).

Figure 6. Real world code snippets where it is effective to stub (left) and fake (right) system call implementations.

Why are Programs Resilient to Stubbing and Faking? Applications are able to detect and react to the failure of a system call. Often, system call failures are non-critical and programs can take action to circumvent them. These actions are the enabling factor of system call stubbing. They include, among others (cf. Figure 5d):

- **Ignoring the issue.** Not all failures are consequential, and programs can simply decide to not take further action. For instance, Redis ignores when `sysinfo` (99) fails to return the maximum memory size and when `ioctl` (16) fails to return the resident size, as this information is only used for output to the debugging logs.
- **Using other system calls**. The system call API is redundant in features: a same means can often be achieved through different system calls. For instance using `mmap` (9) instead of `brk` (12) – a pattern from the glibc early allocator, or reallocating mappings with `mmap` (9) when `mremap` (25) fails, as we observe in SQLite.
- **Falling back to safe default values**. Applications query the OS for various values to tune their behavior (max stack size and file descriptor count, processor affinity and scheduling importance, etc.). When this fails, a safe default can often be adopted. Figure 6a shows an example with `getrlimit` (97) and `prlimit64` (302) in Redis. Another example is using `ioctl` (16) to query the terminal width: when this fails, Redis assumes a safe value of 80 characters.
- **Disabling program functionalities**. Programs may also decide to simply disable the functionality that makes use of the system call; in certain cases, this may not even have observable consequences. For example, many applications only make use of `connect` (42) through the glibc for the NSCD cache socket [15]. When `connect` fails, name caching is simply disabled.

In other cases, programs may interpret the failure conservatively and decide to abort, making stubbing impossible. Still, in a subset of these cases, programs are overly conservative and the failure of the system call is in reality non-critical: if so, faking a successful return value for the system call, without *actually* doing the work of the system call in question, will work. Figure 6b presents a concrete example in Nginx, where `prctl` (157) fails to force the retaining of capabilities upon UID transition; in the context of an OS that does not have a user/kernel separation, like a unikernel, capabilities make little sense and so it is fine to fake success: faking the

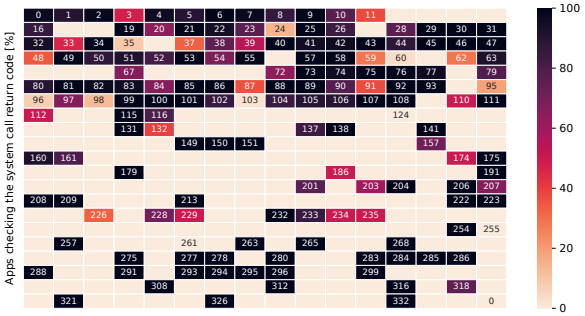


Figure 7. Apps checking system calls return values.

system call here will have strictly no impact on the correct execution of the software. Similar examples are `get/setgroups` (115-116), or `setsid` (112) which have, once again, no meaning in the context of a unikernel. Still, faking OS features may also result in breaking program functionalities, e.g., `pipe2` (293) in Redis (see §5.3). If the functionality is not part of the target set of application features, faking may remain a reasonable approach to achieve a first level of compatibility.

Inversely, certain system calls can (almost) never be stubbed nor faked without breaking core program functionalities. Though generalization is difficult, these system calls typically represent fundamental OS features: executing programs with `execve` (59), opening and writing to connections with `bind` (49), `listen` (50), `socket` (41), and `writev` (20), allocating memory with `mmap` (9). We also find vectored system calls like `fcntl` (72), motivating our discussion in §5.4.

System Call Return Value Checks. In addition to identifying system calls issued by applications, we performed a manual inspection of these applications' source code in order to gather ground truth about which system calls had their return values checked. Is there a link between the presence or absence of checks, and the ability to stub or fake? Note that we are interested here in user-written code, so we look at whether C standard library system calls wrappers – not system calls themselves – have their return value checked.

We choose manual inspection; building an automated static analysis method for this task is non-trivial and rather out of the scope of this paper: some programs directly check the return value, others store it in a variable which is later checked, directly or through auxiliary functions, while yet others rely on macros to do the checking. We semi-automated

the process by building scripts scanning sources for system call wrapper invocations and displaying their corresponding location in source files; we then manually checked this output to determine if the return code was checked or not.

Figure 7 shows, for each system call wrapper, the number of programs that check its return value. The majority have their return value checked. Studying the small set of system calls for which no application has checks, we identify system calls that always succeed, e.g., `alarm` (37), `getppid` (110), but also several that can actually fail: `getrusage` (98), `utime` (132), `inotify_rm_watch` (255) and `futimesat` (261). For those invoked and traced by Loupe, we observe that all can be stubbed/faked for this set of applications. Nevertheless, it would be incorrect to conclude that the ability to stub and fake is induced by the absence of checks: inversely, numerous system calls that are always checked can themselves often be stubbed/faked, such as `ioctl` (16), `uname` (63), or `geteuid` (107). There is also a set of system calls for which only some applications feature checks. These include system calls that are generally assumed to always succeed (even if they can fail) such as `clock_gettime` (228), or freeing resources: e.g., `close` (3), or `unlink` (87). Generally, these can be stubbed/-faked only in some applications. Overall, we conclude that the ability to stub or fake is not a factor of the presence (or absence) of checks, but rather of the semantics of individual system calls and applications.

5.3 Impact on Performance and Resource Usage

An important concern when stubbing and faking system calls is whether doing so would have an effect on performance or resource usage. Both detrimental and *positive* effects are undesirable, as unintended improvements on these metrics may skew comparisons with a full-fledged baseline. To study the question, we use Loupe's ability to record performance and resource usage metrics while performing its analysis. As described in Section 3.2, Loupe gathers performance metrics through user-defined scripts, and resource usage information (peak file descriptor and memory usage) through `/proc`. For the sake of conciseness, we provide detailed results for a subset of three representative, performance-focused applications: Nginx (web server), Redis (key-value store), and iPerf3 (TCP benchmark framework). Nginx is benchmarked with `wrk` [29] (HTTP requests/s), Redis with `redis-benchmark` [21] (SET requests/s), and iPerf3 with an official iPerf client [9] (TCP throughput). All numbers are provided as averages of 10 runs. Our results are visible in Table 2.

Impact on Performance. For the majority of system calls, the variation in performance when stubbing or faking is within the error margin. For the applications considered here, 3/45 system calls trigger a performance change when faked or stubbed. For Nginx, stubbing/faking `write` increases performance as it prevents writing to access logs [14] (something that test scripts do not check – access logs are usually disabled in performance-focused settings as they are written to once per request). It does not, however, prevent payloads from being written to, as this is done via `writev` (which, when stubbed or faked, prevents Nginx to answer requests correctly, and is detected by the test script). Still for Nginx, stubbing or faking `rt_sigsuspend` hurts performance, as it turns the master process' notification-based behavior into busy-waiting. None of these alters the well-functioning of Nginx's core features as tested by the Loupe test script. Conversely, in the Redis case, faking `futex` results in synchronization issues, manifesting as a performance degradation. This alters the core functioning of Redis, clearly indicating that faking `futex` is not a correct path to follow for compatibility, which matches intuitive expectations. As for iPerf3, no system call results in performance degradation when faked or stubbed.

When such variations occur, Loupe notifies the user that further investigation is needed to understand the implications (e.g., on stability or scientific soundness) of stubbing or faking a particular OS feature for a given application. This further emphasizes the need for a tool like Loupe to avoid pitfalls which may cause debugging costs down the line, or skew comparisons with a full-fledged baseline.

Impact on Resource Usage. Similarly to performance, we find that faking or stubbing most system calls does not result in statistically significant variations in resource usage. For the three applications considered, 4/45 system calls result in memory usage variations, and 3/45 in file descriptor usage variations, with one (`brk`) being caused by the libc and thus common among all three applications.

In the general case, as discussed earlier, system calls that allocate resources cannot be stubbed or faked: this is the case for memory allocation services such as `mmap` (9), but also for those that allocate file descriptors such as `openat` (257) (see Figure 5d). In particular cases, the claim is more nuanced; alternatives like `open` (2) do not need to be implemented (e.g., because `openat` is used instead, see Section 5.6). Similarly, `brk` *can* be stubbed or faked in a significant number of cases: for instance, the program exclusively uses `mmap`, and the only usage of `brk` is in the glibc initialization sequence, which is itself capable of falling back to `mmap` if `brk` does not function (at the cost of a slight memory usage increase, see Table 2). Another case is `pipe2`, which creates pipes at the process' demand. Stubbing or faking it results in pipes not being created, which in turn results in an observable reduction in file descriptor count. In the case of Redis, this breaks the persistence feature (which is often disabled in performance-focused experiments), but not the key-value store's core functionalities.

The situation is different for APIs that free resources. In general, `munmap` and `close` can be stubbed or faked without functional impact, though resource usage will increase. For Redis, faking or stubbing `munmap` and `close` leads to a 20% increase in memory usage, and an 8x increase in open file descriptors under a `redis-benchmark` workload (cf. Table 2).

Table 2. Performance and resource usage (file descriptors: FD, memory usage) impact of stubbing and faking for Nginx, Redis, and iPerf3 (=*App*.lications). Only systems calls with impact outside of the error margin (>3%) in either category are displayed. "-" means *no impact*; *+X%* means *X% faster* or *more* resource usage; *-X%* means *X% slower* or *less* resource usage.

App.	System Call	Perf. Impact	FD Usage	Mem. Usage	Explanations of Stubbing/Faking Impact	Breaks…
Nginx	`write`	+15%	-	-	Access logs are not written anymore, increasing performance.	Access Logging
	`brk`	-	-	+17%	Triggers a fallback to `mmap` in the glibc early allocator.	∅
	`clone`	-	-	+10%	Results in master process executing the worker loop.	Core functioning
	`sigsuspend`	-38%	-	-	Results in master process polling (busy-waiting) for events.	∅
Redis	`close`	-	x8	-	FDs are not closed anymore.	∅[1]
	`munmap`	-	-	+19%	Regions are not disposed anymore.	∅[2]
	`brk`	-	-	+2%	Triggers a fallback to `mmap` in the glibc early allocator.	∅
	`sigprocmask`	-	-	-15%	Prevents creation of jemalloc background threads, resulting in memory being freed synchronously and/or at an earlier point.	∅
	`futex`	-66%	+94%	-	Inconsistent synchronization results in incorrect behavior.	Core functioning
	`pipe2`	-	-25%	-	Pipes are not created anymore, resulting in less FDs.	Persistence
iPerf3	`brk`	-	-	+11%	Triggers a fallback to `mmap` in the glibc early allocator.	∅

[1]Within the maximum number of FD limits, core functioning is altered beyond this point. [2]Within the limits of available memory.

Still, although these features can be stubbed or faked without sacrificing stability (as long as resources suffice), we note that the incentives to do so are lower than for other API elements; if the algorithm was developed to allocate resources, it should not be a problem to develop one that frees them.

Lastly, similarly to performance, variations in resource usage turn out to be good indicators of instability caused by stubbing or faking. In the case of Nginx, faking `clone` results in the master process executing the worker event loop, which itself manifests as an increase in memory usage (likely because resources are left dangling). Although functional in practice, it is not a reliable path to take for compatibility and meaningful performance comparison. In the case of Redis, faking `futex` results in inconsistent synchronization, which itself translates into an increased number of allocated file descriptors (see Table 2).

Beyond system calls that (de-)allocate resources, and those that indicate underlying instability, we identify two more classes of system calls which may impact resource usage (or performance):

- **Optimizing system calls:** by giving semantic indications to the kernel regarding e.g., memory management policies, system calls such as `madvise` [12] should influence performance and resource usage. This behavior is not visible when faking/stubbing in Table 2: kernel hints are used rather sporadically in applications, and for those that use them (e.g., Redis), the kernel did not perform actions that impacted our metrics. Impact may be observable in other settings, e.g., multi-process scenarios.
- **System Limit Setters/Getters:** by getting/setting system defaults (e.g., max stack size, number of FDs), getter/setter system calls like `prlimit64` (or part of `ioctl`) may also result in resource usage or performance variations. For instance, with system defaults different from the ones in Table 2, stubbing `prlimit64` in Redis results in 30% lower memory usage under a `redis-benchmark` workload because the libc (stack size) and Redis (FD limits) default to values conservatively lower than the system limits.

Impact on Stubbing and Faking Policy. Overall, we stress the importance of evaluating the impact of stubbing and faking on performance metrics as part of the process of deciding what to support and how. Though most system calls do not impact performance metrics, some do: when the underlying reason is instability, the OS feature should never be faked; otherwise, whether or not to stub or fake should be an *explicit factor* of the experimental setup and expectations on the OS prototype. It is critical that the (positive or negative) impact of stubbing and faking must not be mistaken for that of the system's design. Overall, we encourage authors of future systems research works to explicitly list features that they stub or fake for reproducibility and future analysis.

Insight: Stubbing/Faking does not impact performance and resource usage in the general case. Still, there are edge cases which may or may not indicate correctness issues. Impact on either metric must call for special care when stubbing/faking.

5.4 Partial Implementation of System Calls

In the previous sections, we considered system calls as monolithic API elements. This consideration shows its limits when investigating vectored system calls (e.g., `ioctl`) or complex system calls like `mmap` (usable for memory allocation and file mapping, two very different purposes). To clarify this point, we use Loupe to determine the precise set of sub-system call features applications require.

Our insights are twofold. First, applications execute surprisingly few features from complex or vectored system calls. For example, almost all applications require `arch_prctl` (158) (see Figure 5d). However, they are far from requiring a full implementation: in fact, in all applications that we considered,

Table 3. Nginx 0.3.19 system call usage with different glibc versions. System calls that vary because of the architecture (32/64-bit) are in *italics*; other variations are in **bold**.

GLIBC 2.3.2 / 32-bit (*48 system calls*)	GLIBC 2.31 / 64-bit (*51 system calls*)
_llseek, accept, access, bind, brk, clone, close, connect, epoll_create, *fcntl64*, epoll_ctl, epoll_wait, execve, exit_group, dup2, *fstat64*, *geteuid32*, mkdir, **mmap2**, *setuid32*, **old_mmap**, *setgroups32*, **uname**, **open**, prctl, *pread*, **pwrite**, read, rt_sigaction, rt_sigprocmask, rt_sigsuspend, **set_thread_area**, *setgid32*, setsid, setsockopt, **recv**, socket, socketpair, *stat64*, bind, munmap, umask, getpid, getrlimit, ioctl, write, writev, gettimeofday, listen	read, write, close, *stat*, *fstat*, lstat, *lseek*, brk, rt_sigaction, **mmap**, ioctl, rt_sigprocmask *pread64*, setsockopt, writev, access, sendfile, socket, munmap, accept, connect, epoll_wait, **mprotect**, **recvfrom**, listen, socketpair, *pwrite64*, *prlimit64*, epoll_create, clone,execve, *fcntl*, mkdir, umask, *setuid*, *setgid*, *geteuid*, setsid, rt_sigsuspend, dup2, *setgroups*, **_sysctl**, prctl, **arch_prctl**, getpid, **set_tid_address**, exit_group, epoll_ctl, **openat**, **set_robust_list**

this system call was exclusively called by the libc, which requires one single feature (`ARCH_SET_FS`, out of 6 in total) related to thread local storage setup. The situation is similar for `prlimit64` (302), required by many applications: out of 16 features, only 3 are used, `RLIMIT_CORE`, `_NOFILE`, and `_STACK`, the latter one being used almost exclusively as part of the libc initialization. This is also the case for `ioctl` (16): with a benchmark load, Redis, weborf, and h2o use one single feature (`TCGETS`), Nginx two (`FIONBIO` and `FIOASYNC`), and Lighttpd none. All of them can be stubbed.

Second, when looking at required features of system calls, we find that certain system calls such as `fcntl` typically exhibit a mix of required and fakeable/stubable features, and the required set is typically common among applications. For instance, `F_SETFL` is required to put file descriptors in non-blocking mode in all applications except Nginx, a critical operation for most codebases. On the other hand, `F_SETFD` is widely executed across applications but can always be stubbed as it is used to enable *close-on-exec* on file descriptors, a non-critical operation. In these cases, taking a look at the required system calls at the granularity of a system call would make the situation appear worse than it is in practice.

Insight: Several complex system calls do not require a full implementation to support a large number of applications.

5.5 Stability of System Call Usage Over Time

Once an OS prototype supports an application, how likely is it that, as the program evolves over time, additional or different system calls will be required, breaking the initial support? Here we study the stability of system call usage by applications and libcs.

Evolution: C Standard Library. We first study the libc, from which most system calls invocations generally originate. We compiled Nginx v0.3.19 against an old version of glibc (2.3.2, from 2003) and a modern one (2.31, from 2020). Since we were unable to run Nginx 0.3.19 with glibc 2.3.2 in 64-bit mode, we compiled and run this configuration in 32-bit. This is likely due to these versions featuring unstable AMD64 support (the first AMD64 CPUs were released in 2003 [42]). The results in Table 3 show that the number of used system calls is more or less unchanged, 48 vs 51. Moreover, we see that most of the change in system call usage is caused by the deprecation of old system calls. Still, there is some evolution in the types of system calls invoked, which we classify into two categories. First, the recent libc uses a different version of some system calls due to a change of architecture (e.g., it uses `pread64` instead of `pread`). Second, the recent libc uses additional system calls, e.g., `arch_prctl` to set up TLS. Setting aside the issue of supporting a new architecture (orthogonal to compatibility), we assume that it is the second category that would require supporting effort for updating a given compatibility layer as applications evolve. However, we consider this effort to be low: we only count 8 new system calls in 17 years for this libc/application combination.

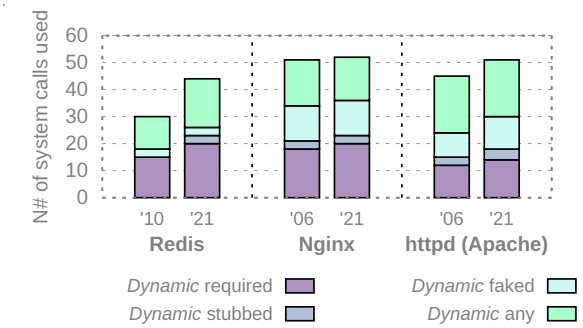


Figure 8. System call usage and capacity to be stubbed/faked for recent (2021) and older (2005-2010) applications releases.

Evolution: Application. We are now interested to see how the system call usage of an application changes over the years. For this experiment, we used a modern glibc/compiler. We explore the difference in system call usage of Nginx, Apache and Redis through the years and list the results in Figure 8. We observe that, although the number of Linux system calls has increased, all applications are using roughly the same amount of system calls; the number of system calls that can be stubbed or faked also remains almost unchanged.

In all, we find the usage of system calls by applications and libcs to be fairly stable over time. This is further encouragement to OS prototype developers: once you provide support for an application, you are likely to be able to keep it with minimal work for a long while.

Insight: Application and libc system call usage patterns tend to be stable over time: support is a one-time effort.

5.6 C Library Impact on System Call Usage

Typical applications perform the majority of their system call invocations through the C standard library (libc). Bypassing the libc using direct system call invocation happens only for functionalities rarely called by user code (e.g., `futex`) or newer system calls for which libcs do not provide a wrapper: we counted around 51 system calls (58 including removed/unimplemented system calls) that do not have a wrapper as of glibc 2.33. In this case, applications wishing to invoke them use the `syscall` function. Setting aside these special cases, we find that the libc implementation greatly influences the system call API usage of applications. This is due to two main factors: (1) the libc initialization sequence and (2) the choice of system call alternatives (e.g., `openat` vs. `open`).

Libc Initialization Sequence. The initialization sequence is the libc code executed from the program entry point until the user's `main` function is invoked. The system calls invoked by that code will be by construction present in any binary linked against that libc and constitute the minimum set of system calls an OS should implement to support this libc. To study initialization sequences, we recorded the system call usage of a trivial application printing "Hello, world!" across two libcs, glibc (version 2.28) and musl (version 1.2.2), for a dynamically- and a statically-linked executable. Results in Table 4 show that the number and types of system calls executed vary: glibc's initialization sequence invokes for dynamically compiled binaries 2.5x more system calls vs. musl, and 1.8x more for statically compiled programs. The system calls invoked also change: glibc is not a strict superset of musl and out of 18 system calls in total, only 6 are common to both libcs for dynamic, 3 for static (and 3 overall).

System Call Alternatives. Some discrepancies are due to the libcs choosing different system call alternatives to perform the same task. For example, glibc uses `write` for `printf`, vs. `writev` for musl. Similarly, musl uses `ioctl` to check that the TTY is writable, while glibc uses `fstat`. Finally, glibc uses `openat`, `read`, `mmap`, and `mprotect` to map the libc into the address space, an operation that musl achieves by embedding the libc into the linker itself, avoiding these system calls entirely. Other differences are caused by libc-specific initialization and debugging features. For example, even in single-threaded programs, musl will call `set_tid_address` during TLS initialization, something that glibc does not. Glibc, on the other hand, uses `uname` to ensure that the kernel is recent enough, `readlink` to expand `$ORIGIN` with statically compiled binaries, and `access` for a debugging feature; none of these used by musl's initialization sequence.

Insight: The *choice of libc* and *linking type* strongly influences system call usage: as much as 4.5x fewer system calls between dynamic glibc and static musl.

Table 4. System call API usage of a hello world application across glibc (2.28) and musl (1.2.2). Apart from `exit_group` and `write`/`writev`, this set corresponds to the libc initialization sequence. Differing system calls are in bold.

glibc	musl
28 system calls (dynamic binary)	*11 system calls (dynamic binary)*
execve (1x), brk (3x), arch_prctl (1x), exit_group (1x), **access** (1x), **openat** (2x), **fstat** (3x), mmap (7x), **close** (2x), **read** (1x), mprotect (4x), **munmap** (1x), **write** (1x)	execve (1x), brk (2x), arch_prctl (1x), exit_group (1x), **writev** (1x), mmap (1x), mprotect (2x), **ioctl** (1x), **set_tid_address** (1x)
11 system calls (static binary)	*6 system calls (static binary)*
execve (1x), arch_prctl (1x), exit_group (1x), **brk** (4x), **fstat** (1x), **write** (1x), **uname** (1x), **readlink** (1x)	execve (1x), arch_prctl (1x), exit_group (1x), **writev** (1x), **ioctl** (1x), **set_tid_address** (1x)

6 Discussion: Pitfalls & Future Works

As discussed throughout this work, there are pitfalls to developing OS compatibility layers with dynamic analysis, stubbing, faking, and partial support techniques.

Impact on Stability. Dynamic analysis, stubbing, faking, and partial support techniques, bring the concern of stability: *do we trade off correctness to reduce porting time?* Loupe assumes that users are able to evaluate the functionality of application features they aim to support by specifying a set of tests (§3.2). The tool ensures that this set of tests can be passed reliably, over multiple runs, when applying stubbing, faking, and partial support techniques. Loupe can also ensure that performance, resource usage, and any other metric, remains stable (§5.3). Under this assumption, stability issues outside users' target feature range are not in the problem scope of Loupe, or our study. Still, perfect correctness cannot be guaranteed, and compatibility bugs may hide in incomplete or buggy tests, varying test environments, etc. We believe that these are reasonable trade-offs to be made in transitional development stages of a new OS.

Impact on Evaluation Metrics. Assuming stability, another concern remains: *do we trade off (or simply influence) performance, resource usage, or any other metric for porting time?* This is most relevant as early OS prototypes must be able to compare, in a sound manner, properties with full-fledged baseline OSes. We show that, although the majority of system calls do not influence performance metrics when stubbed, faked, or partially supported, there *are* pitfalls: even when reliably passing tests, these techniques can result in visible performance or resource usage variations (§5.3). Loupe improves on the state of the art, which does not consider this problem, by evaluating these costs systematically and early, to provide strong evidence that achieved support does not impact chosen metrics. Still, it remains impossible to formally guarantee that these metrics will be unaffected in all cases. We believe that this too constitutes a reasonable trade-off in development stages.

Overall, dynamic analysis, stubbing, faking, and partial support should not be seen as end-goals for production-ready compatibility, but as a transitional, "necessary evil" in development phases. The takeaway of this paper should not be that most of the system call API is irrelevant, or that static analysis is impertinent in engineering compatibility layers; each corresponds to distinct life cycle phases in the development of new OS. As we show, static analysis is not appropriate in earlier stages, however its output should decisively be a target in later stages of development, and full support should eventually come to achieve high levels of correctness assurance.

Looking forward, we plan to improve Loupe with support for other analysis metrics, such as identifying standard application-specific logs and error message formats, or network and file system usage statistics, to better detect silent faults and effects of stubbing, faking, and partial support techniques. We believe that there remain many interesting research opportunities in application analysis for compatibility that should be explored in future works. Future research avenues include exploring speeding up the analysis by transferring knowledge across applications, and generally using machine learning techniques to identify patterns over the data set, at scale, and generating application-specific workloads.

7 Related Work

OS Compatibility Layers. Many research and prototype OSes have implemented compatibility layers to transparently support legacy software. An early example [31] presents a compatibility layer for Linux applications implemented in the K42 [44] OS. Similarly to our work, the authors note that to be widely adopted, an OS must provide good support for existing applications, and that emulating the Linux API is the best way to achieve this goal without requiring modification of target applications. In another study [40], researchers propose a POSIX compatibility layer for the Embassies [39] system. This work presents the construction of the compatibility layer, which is realized in a fully ad-hoc way. As we demonstrate, this process can be highly optimized with Loupe. Still, the authors make some observations similar to ours, in particular the fact that some system calls are "failure-oblivious" (i.e., they can be stubbed) and others are "neutered" (they can be faked). Other works proposed compatibility layers for new monolithic [10, 11, 30], libOS [6, 43, 45, 46, 51, 62, 64] or micro-kernels [5, 20], web browsers [56], for running applications within the Linux kernel [60], as well as various OS interoperability layers for existing kernels [11, 18, 28, 50, 61]. To the best of our knowledge, all these compatibility layers have been developed in an organic way.

Libc-Based Compatibility Layers. Some works [13, 43] approach compatibility at the libc level, instead of the system call API. Though most system calls are performed through the libc, prior works have shown that interfacing at the libc level leads to weaker degrees of compatibility [52] because many programs do issue system calls outside the libc (500+ ELF Debian 10 executables fall into that category [52]). Thus, we focus on compatibility at the system call level.

Linux & POSIX APIs Studies. Past work studied the usage of the Linux [63] and POSIX [32] APIs by applications. Tsai et al. [63] use binary static analysis to measure the system calls and pseudo files required by a large set of binaries from the Ubuntu 15.04 archive. Even for the most minimal Ubuntu installation, the study reports that 224 system calls, 208 `ioctl`/`prctl`/`fcntl` codes and 100+ pseudo files require support. Our results demonstrate that static binary analysis is overly pessimistic. Using dynamic analysis, Loupe shows that the amount of OS features required to run standard benchmarks or even full test suites is actually much lower.

Another study [32] leverages both static and dynamic analysis to measure applications' POSIX API usage. Unlike this work, the authors' goal is not to determine and optimize compatibility efforts, but to study the evolution of the POSIX interface and identify emerging/missing abstractions. Though the study provides valuable insights for building a compatibility layer at the POSIX (i.e., libc) level [43, 46], past studies showed that the Linux API (mainly system calls) provided a higher degree of compatibility [52]: the authors themselves [32] note that many applications (e.g., Go apps) circumvent POSIX to use OS specific APIs.

8 Conclusion

We propose Loupe, an efficient method to determine and prioritize OS features new compatibility layers should implement to provide support for as many applications as possible, as early as possible. Applying Loupe to 100+ applications, we provide examples of support plans, demonstrate high engineering effort savings, and study our measurements in depth. A significant number of system calls identified as needed by previous works are actually not required for those applications to run. These results bring a message of hope to the level of compatibility a new OS must provide in order to support mainstream applications, and should provide encouragement to ongoing and future research OS development efforts.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Donald E. Porter, for their insights. This work was funded by a studentship from NEC Labs Europe, a Microsoft Research PhD Fellowship, UK's EPSRC grants EP/V012134/1 (UniFaaS), EP/V000225/1 (SCorCH), and the EPSRC/Innovate UK grant EP/X015610/1 (FlexCap), as well as EU H2020 grants 825377 (UNICORE), 871793 (ACCORDION) and 758815 (CORNET). UPB authors were supported by VMWare gift funding.

A Artifact Appendix

A.1 Abstract

This artifact contains the source code of Loupe, the proof-of concept of our OS feature analysis method, along with the OS feature usage data generated for the paper. The goal of this artifact is to allow readers to understand and re-use Loupe in their experimental OS development workflows.

A.2 Artifact Check-List (Meta-Information)

- **Program:** the Loupe feature analysis method, along with various scripts used in the paper.
- **Binary:** Mostly Python and Bash scripts, C `seccomp`/`ptrace` core automatically built from source.
- **Data set:** Generated with Loupe over the course of its development, and provided separately from the code of Loupe.
- **Run-time environment:** Tested on GNU/Linux Debian 12. Should work on any Debian-based distribution. Installation of Docker, git, Python 3, `python-git` is needed.
- **Hardware:** Any x86-64 CPU – Loupe does not not have custom hardware requirements. We recommend (but do not require) >8 CPU cores to obtain stable performance numbers.
- **Output:** OS feature usage data; Loupe support plans.
- **Experiments:** E1-E3, all described under §A.6.
- **How much disk space required (approximately)?:** <10GB, excluding dependencies. Most of the disk space is taken by Docker containers.
- **How much time is needed to prepare workflow (approximately)?:** <20 minutes, depending on internet link speed, most of it spent installing dependencies.
- **How much time is needed to complete experiments (approximately)?:** About 1 hour, incl. 40 minutes of computation.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** BSD-3-Clause.
- **Data licenses (if publicly available)?:** CC-BY.
- **Workflow framework used?:** Docker, scripts.
- **Archived (provide DOI)?:** `10.5281/zenodo.8386116`

A.3 Description

A.3.1 How to access. The latest version of the Loupe source code[4] and data set[5] can found on GitHub. Alternatively, individual releases can be downloaded from our Zenodo archive[6].

A.3.2 Hardware dependencies. Any modern x86-64 CPU is suitable. Loupe does not require custom hardware. We recommend (but do not require) >8 CPU cores to obtain stable performance numbers.

A.3.3 Software dependencies. Loupe was developed and tested on GNU/Linux Debian 12, but should work on any Debian-based distribution. Loupe is Linux-specific and requires a kernel that supports seccomp and ptrace (this should match all stable kernels). Installation of Docker, Python 3, `python-git` is needed and documented below. Loupe is known to work with at least Python 3.10.5 and `python-git` 3.1.27.

A.3.4 Data sets. The Loupe data set is provided separately from the source code as documented above. It consists of the Loupe database, a human-readable, text-based database based on a git repository with a custom layout.

A.4 Installation

We will first install dependencies required to build and run Loupe. This may take up to 20 minutes.

Step 1: install Docker, e.g., following official instructions[7]. Then, install git, Python 3, and `python-git`:

```
$ sudo apt install python3-pip git
$ pip3 install gitpython
```

Step 2: clone our AE repositories with appropriate tags:

```
$ git clone -b asplos24-ae-v1 \
    https://github.com/unikraft/loupe.git
$ git clone -b asplos24-ae-v1 \
    https://github.com/unikraft/loupedb.git
```

Step 3: build dependencies and base Docker containers:

```
$ pushd loupe
$ make all
$ popd
```

The system is now set-up to run Loupe.

A.5 Experiment workflow

This artifact appendix describes three experiments (E1-E3) which allow evaluators to assess that the artifact is consistent, exercisable, and complete, i.e., it allows users to generate the main results of our paper:

- **Main result 1**: *fine-grain OS feature usage data.* This result is evaluated by experiment E1, showing how Loupe can be used to *generate* it, and *reproduce* existing results.
- **Main result 2**: *OS support plans.* This result is evaluated by experiment E2.
- **Main result 3**: *stubbing and faking performance impact.* This result is evaluated by experiment E3.

This artifact does not cover reproducing all the paper's results; re-generating all results for the 100+ applications covered by Loupe (particularly test suites) would require an involvement in time and resources that goes beyond what can be expected from artifact evaluators.

A.6 Evaluation and expected results

A.6.1 E1: Gathering OS feature usage (Nginx `wrk`).

Time Cost: 25 minutes, incl. 12-15 minutes computation.

[4] https://github.com/unikraft/loupe
[5] https://github.com/unikraft/loupedb
[6] https://doi.org/10.5281/zenodo.8386116
[7] https://docs.docker.com/engine/install/

Experiment Result: We will generate OS feature usage data for Nginx under a `wrk` workload, showing which features can be stubbed and/or faked, and which ones must be implemented. We will show how to reproduce these results.

Step 1: Write a test script using `wrk` (`nginx-test.sh`), and create a Loupe container (`Dockerfile.nginx`):

```
$ cd loupe
$ cp -r examples/E1/* .
$ # we encourage readers to inspect and understand the files
$ # they just copied (nginx-test.sh and Dockerfile.nginx)
```

Step 2: Run the Loupe analysis. When asked whether or not to create a new Loupe database (`Create it?`), select `y`:

```
$ ./loupe generate -b -db ../loupedb-ae -a "nginx" \
                   -w "wrk" -d ./Dockerfile.nginx
```

Step 3: Commit the analysis data:

```
$ pushd ../loupedb-ae
$ git add nginx && git commit -m "Nginx wrk analysis."
$ popd
```

Step 4: Retrieve the analysis data and manually inspect it:

```
$ ./loupe search --show-usage -db ../loupedb -a "nginx" \
                 -w benchmark
$ # this will return the results of the analysis; it should be
$ # roughly in line with the results of Figure 4, application
$ # Nginx, bar "bench": ~19-20 required, and about 50 in total.
```

Step 5: Reproduce the same run:

```
$ cd ../loupedb-ae/nginx/benchmark-wrk/${hash}/
$ # the hash should be 126e629dc544b4695f12ed602f6902aa
$ ../../../../loupe/loupe generate -b \
                   -db ../../../../loupedb-ae -a "nginx" \
                   -w "wrk" -d ./Dockerfile.nginx
```

Step 6: Check that the results are the same:

```
$ git status
$ # this should report no differences apart from cmd.txt
$ # and explore.logs (which are just log files)
$ cd ../../../../ # go back to the root directory
```

A.6.2 E2: Generating OS support plans with Loupe.

Time Cost: 1-5 minutes of human time.

Experiment Result: We will generate an OS support plan using Loupe and the pre-existing Loupe database (`loupedb`).

Step 1: Generate the support plan for the Kerla OS, and manually inspect it.

```
$ cd loupe && ./loupe search -db ../loupedb --guide-support \
                ../loupedb/Kerla.syscalls \
                --applications "*" --workloads bench
$ # This will not match Table 1 because the set of apps
$ # is different. Just make sure it makes sense.
$ cd ../ # go back to the root directory
```

A.6.3 E3: Evaluating the Performance Impact of Stubbing and Faking.

Time Cost: 30 minutes, incl. 25 minutes computation.

Experiment Result: We will evaluate the performance and resource usage impact of stubbing and faking on Nginx under a `wrk` benchmark using Loupe.

Step 1: Build the Loupe Nginx Docker container:

```
$ cd loupe && git clean -xdf
$ cp -r examples/E3/* . # we encourage readers to inspect
$ # and understand the files they just copied
$ ./loupe generate -b -db ../loupedb -a "nginx" -w "wrk" \
                -d ./Dockerfile.nginx --only-build-docker
```

Step 2: Start the Loupe Nginx Docker container:

```
$ docker run -it docker.io/library/nginx-loupe bash
```

Step 3: Run the performance analysis and inspect results:

```
$ /root/explore.py --perf-analysis -t /root/nginx-test.sh \
  -b /root/nginx/objs/nginx -- -p /root/nginx -g "daemon off;"
$ # inspect results, should be roughly similar to Table 2
$ # dissimilarities may be due to an unstable experimental
$ # setup - for the paper results we CPU pinned, averaged, etc.
```

A.7 Experiment customization

We encourage readers to run a new application of their own choice in the Loupe framework. The artifact provides rather thorough documentation to achieve this (see §A.8).

A.8 Notes

Loupe ships with additional documentation, all described in the main `README.md` and provided in the documentation folder[8]. The documentation expands on the structure of the artifact, on the interpretation of performance metrics, as well as on Loupe Dockerfiles and database. Overall, we recommend readers to read the main `README.md` before installing or using Loupe as it contains more documentation (and possibly more up-to-date documentation) than this appendix.

A.9 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

References

[1] debhelper(7) — linux manual page. https://www.man7.org/linux/man-pages/man7/debhelper.7.html, accessed 08/17/23.

[2] dh_auto_test - automatically runs a package's test suites. https://manpages.debian.org/testing/debhelper/dh_auto_test.1.en.html, accessed 08/17/23.

[3] errno - number of last error (including a description of -ENOSYS). https://www.man7.org/linux/man-pages/man3/errno.3.html, accessed 08/01/23.

[4] Github – Rumprun packages: Ready-made packages of software for running on the Rumprun unikernel. https://github.com/rumpkernel/rumprun-packages, accessed 08/01/23.

[5] Google Fuchsia website. https://fuchsia.dev/, accessed 08/01/23.

[8] https://github.com/unikraft/loupe/tree/staging/doc

[6] Google Gvisor Github webpage. https://github.com/google/gvisor, accessed 05/03/2018.

[7] Gramine: a library OS for unmodified applications. https://gramineproject.io, accessed 08/10/23.

[8] How sqlite is tested. https://www.sqlite.org/testing.html, accessed 08/17/23.

[9] iPerf - the ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/iperf-doc.php, accessed 08/01/23.

[10] Kerla GitHub repository: A new Operating System kernel with Linux binary compatibility written in Rust. https://github.com/nuta/kerla, accessed 08/01/23.

[11] Linuxulator (Linux emulation): running unmodified Linux binaries under FreeBSD. https://wiki.freebsd.org/Linuxulator, accessed 08/01/23.

[12] madvise(2) — Linux manual page. https://man7.org/linux/man-pages/man2/madvise.2.html, accessed 08/01/23.

[13] Newlib: a c library intended for use on embedded systems. https://sourceware.org/newlib/, accessed 12/12/2017.

[14] Nginx docs: Configuring logging. https://docs.nginx.com/nginx/admin-guide/monitoring/logging/, accessed 08/17/23.

[15] nscd - name service cache daemon. https://www.man7.org/linux/man-pages/man8/nscd.8.html, accessed 08/17/23.

[16] OpenBenchmarking.org software repository. https://openbenchmarking.org/, accessed 08/01/23.

[17] OSv application repository. https://github.com/cloudius-systems/osv-apps, accessed 08/01/23.

[18] Proton (Valve Software) GitHub repository. https://github.com/ValveSoftware/Proton, accessed 08/01/23.

[19] ptrace(2) - process trace. https://man7.org/linux/man-pages/man2/ptrace.2.html, accessed 07/31/2023.

[20] ReactOS Github page: A free Windows-compatible Operating System. https://github.com/reactos/reactos, accessed 08/01/23.

[21] Redis benchmark: Using the redis-benchmark utility on a Redis server. https://redis.io/docs/management/optimization/benchmarks/, accessed 08/01/23.

[22] Redis test suite. https://github.com/redis/redis/tree/unstable/tests, accessed 08/17/23.

[23] seccomp(2) - operate on secure computing state of the process. https://man7.org/linux/man-pages/man2/seccomp.2.html, accessed 07/31/2023.

[24] strace - linux syscall tracer. https://strace.io/, accessed 08/17/23.

[25] Unikraft application repository: Applications supported by the Unikraft libOS. https://github.com/orgs/unikraft/repositories, accessed 08/01/23.

[26] Unikraft static binary analysis tool (part of the Loupe artifact). https://github.com/unikraft/loupe/tree/staging/src/static-binary-analyser, accessed 08/17/23.

[27] Unikraft static source analysis tool (part of the Loupe artifact). https://github.com/unikraft/loupe/tree/staging/src/static-source-analyser, accessed 08/17/23.

[28] Wine HQ – a compatibility layer to run Windows applications on POSIX. https://www.winehq.org/about, accessed 08/01/23.

[29] wrk - a HTTP benchmarking tool. https://github.com/wg/wrk, accessed 08/01/23.

[30] Zephyr Project: A proven RTOS ecosystem. https://www.zephyrproject.org/, accessed 08/01/23.

[31] Jonathan Appavoo, Marc Auslander, Dilma Da Silva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert Wisniewski, and Jimi Xenidis. Providing a Linux API on the scalable K42 kernel. In *Proceedings of the 2003 USENIX Annual Technical Conference, FREENIX Track*, ATC'03, 2003.

[32] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. 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, 2016.

[33] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. Breaking the boundaries in heterogeneous-ISA datacenters. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'17, 2017.

[34] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP'09, 2009.

[35] Alexander Bulekov, Rasoul Jahanshahi, and Manuel Egele. Saphire: Sandboxing PHP applications with tailored system call allowlists. In *Proceedings of the 30th USENIX Security Symposium*, USENIX Security'21, 2021.

[36] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. Automating seccomp filter generation for Linux applications. In *Proceedings of the 2021 on Cloud Computing Security Workshop*, CCSW'21, 2021.

[37] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. Sysfilter: Automated system call filtering for commodity software. In *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses*, RAID'20, 2020.

[38] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. Confine: Automated system call policy generation for container attack surface reduction. In *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses*, RAID'20, 2020.

[39] Jon Howell, Bryan Parno, and John R. Douceur. Embassies: Radically refactoring the web. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'13, 2013.

[40] Jon Howell, Bryan Parno, and John R. Douceur. How to run POSIX apps in a minimal picoprocess. In *Proceedings of the 2013 USENIX Annual Technical Conference*, ATC'13, 2013.

[41] Antti Kantee. The rise and fall of the operating system. *USENIX login*, 40(5), 2015.

[42] C.N. Keltcher, K.J. McGrath, A. Ahmed, and P. Conway. The AMD Opteron processor for multiprocessor servers. *IEEE Micro*, 23(2), 2003.

[43] Avi Kivity, Dor Laor Glauber Costa, and Pekka Enberg. OSv - optimizing the operating system for virtual machines. In *Proceedings of the 2014 USENIX Annual Technical Conference*, ATC'14, 2014.

[44] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, et al. K42: building a complete operating system. In *Proceedings of the 1st European Conference on Computer Systems*, EuroSys'06, 2006.

[45] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ștefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: Fast, specialized unikernels the easy way. In *Proceedings of the 16th European Conference on Computer Systems*, EuroSys'21, 2021.

[46] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. A Linux in Unikernel Clothing. In *Proceedings of the 15th European Conference on Computer Systems*, EuroSys'20, 2020.

[47] Hugo Lefeuvre, Gaulthier Gain, Daniel Dinca, Alexander Jung, Simon Kuenzer, Vlad-Andrei Badoiu, Razvan Deaconescu, Laurent Mathy, Costin Raiciu, Pierre Olivier, and Felipe Huici. Unikraft and the coming of age of unikernels. *USENIX; login*, 2021.

[48] A. Madhavapeddy, R. Mortier, C. Rotsos, DJ. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: library operating systems for the cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'13, 2013.

[49] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, 2014.

[50] Christopher McLellan. Docker desktop for Mac - support for running x86-64 binaries with Rosetta 2, 2022. https://github.com/docker/roadmap/issues/384, accessed 08/01/23.

[51] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A binary-compatible unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE'19, 2019.

[52] Pierre Olivier, Hugo Lefeuvre, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A syscall-level binary-compatible unikernel. *IEEE Transactions on Computers*, 2021.

[53] OSv Contributors. Stub of `io_setup`, 2021. https://github.com/cloudius-systems/osv/blob/317d259ab5b0b49a1a114bc837147746e471abc9/core/libaio.cc#L17, accessed 08/21/2022.

[54] Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. Automated policy synthesis for system call sandboxing. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), 2020.

[55] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the Library OS from the top down. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'11, 2011.

[56] Bobby Powers, John Vilk, and Emery D. Berger. Browsix: Bridging the gap between UNIX and the browser. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'17, 2017.

[57] FL. Rawson. Experience with the development of a microkernel-based, multiserver operating system. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, 1997.

[58] Ali Raza, Thomas Unger, Matthew Boyd, Eric B Munson, Parul Sohal, Ulrich Drepper, Richard Jones, Daniel Bristot De Oliveira, Larry Woodman, Renato Mancuso, Jonathan Appavoo, and Orran Krieger. Unikernel Linux (UKL). In *Proceedings of the 18th European Conference on Computer Systems*, EuroSys'23, 2023.

[59] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*, OSDI'04, 2004.

[60] Willy Tarreau. Nolibc: a minimal C-library replacement shipped with the kernel, 2023. https://lwn.net/Articles/920158/.

[61] Deepu Thomas and Seth Juarez. Windows Subsystem for Linux (WSL) overview, 2016. https://learn.microsoft.com/en-us/archive/blogs/wsl/windows-subsystem-for-linux-overview.

[62] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the 9th European Conference on Computer Systems*, EuroSys'14, 2014.

[63] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. A study of modern Linux API usage and compatibility: what to support when you're supporting. In *Proceedings of the 11th European Conference on Computer Systems*, EuroSys'16, 2016.

[64] Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference*, ATC'17, 2017.

[65] D. Wagner and R. Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy.*, S&P'01, 2000.