Dynamic Linkers Are the Narrow Waist of Operating Systems

Charly Castes
EPFL
Switzerland

Adrien Ghosn
Azure Research, Microsoft
United Kingdom

Abstract

While software applications, programming languages, and hardware have changed, operating systems have not. Widely-used commodity operating systems are still modelled after the ones designed in the seventies. The accumulated burden of backward compatibility with the large software ecosystems that run our workloads prevents systems from embracing more efficient and disruptive designs explored by the system research community.

This paper advocates a fresh approach to operating system research, where innovations are incrementally integrated into operating systems, without disrupting existing software, to gradually reshape our daily-use systems. The dynamic linker emerges as a pivotal element in this transformation process, redefining system behavior. The paper outlines specific use cases, covering performance enhancements, strengthened security measures, streamlined software deployment, and enriched programming language abstractions. Additionally, the paper introduces Spidl, an experimental modular dynamic linker to facilitate the exploration of this promising new research avenue.

CCS Concepts • **Software and its engineering** → **Abstraction, modeling and modularity**; **Runtime environments**.

ACM Reference Format:
Charly Castes and Adrien Ghosn. 2023. Dynamic Linkers Are the Narrow Waist of Operating Systems. In *12th Workshop on Programming Languages and Operating Systems (PLOS '23), October 23, 2023, Koblenz, Germany*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3623759.3624548

PLOS '23, October 23, 2023, Koblenz, Germany

ACM ISBN 979-8-4007-0404-8/23/10…$15.00
https://doi.org/10.1145/3623759.3624548

1 Introduction

The original Unix system was developed more than 50 years ago, initially in assembly and then rewritten in C in 1973 [42], marking the beginning of a tight co-design between the language and the operating system [29]. Since then, C has remained the *lingua franca* for system interfaces and dictates, to this day, how programs are assembled, executed, and isolated. For over five decades, the C and Unix couple acted as a robust foundation, fostering the growth of a complex software ecosystem leading to novel programming languages with improved safety guarantees, new high-level programming models, and principled software development methodologies, all of which led to incredible technological innovations. Ironically, operating systems have reaped the least benefits from the overall progress in software technologies. Only a handful of these innovations have been integrated into their implementation. Operating systems continue to be written in unsafe languages and their APIs have remained largely unchanged. This not only gives a sense that our systems are falling behind but also hamper or slow down software innovation, as demonstrated by *e.g.*, kernel bypass becoming a widely adopted approach for low-latency networking.

In 2000, Rob Pike argued that "*systems software research is irrelevant*", and that it has become "*a sideline to the excitement in the computing industry*" [38]. Indeed, despite prolific results from the research community [9, 11, 18, 27, 35, 40, 43] there has been very little impact on industrial operating systems.

The limiting factor in the adoption of radical new operating systems is their incompatibility with existing software ecosystems. The implementation of a new kernel is only the first challenge, and arguably the easiest, when compared to the necessity to rewrite development tools, compilers, system libraries, and rebuild, from scratch, an entire ecosystem before porting any application. This daunting effort can, in part, explain why disruptive new operating systems, such as Singularity [27] or Plan 9 [40], have not been widely adopted. BeOs [3] shows that even with a mature development ecosystem, porting applications remains a deterring factor [20].

A similar problem arises in the development of new programming languages. To facilitate adoption, programming language developers often provide interoperability with existing languages and gradually reimplement standard libraries in the new language. This approach trades-off guarantees provided by the new language (*e.g.*, memory safety in Rust [7]) for usability and to speed up adoption, until a sufficiently large ecosystem can be developed. Without any surprise, the C language is often the first one to gain support for interoperability.

New operating system designs would benefit from a similar approach, that could gradually replace portions of an existing software ecosystem with new implementations unlocking better performance, stronger security guarantees, or simply more flexible programming abstractions. The challenge in this approach is to pick the right layer to introduce interoperability between existing and new systems.

This paper argues that dynamic linkers are the ideal system interoperability layer to decouple system designs from existing software ecosystems, and allow them to evolve separately. First, dynamic linkers are standard programs that run in user-space and directly interact with the underlying operating system. Second, they are responsible for loading other programs and can thus be transparently adapted to new system abstractions, unbeknownst to the program being instantiated. Third, they already support complex dynamic binary transformations, code patching, and contextual loading of libraries.

This paper is organized as follows: § 2 provides useful background on dynamic linking and the ELF format, highlighting how they can be transparently leveraged to extend our systems. However, standard dynamic linker implementations are hard to extend and modify. § 3 addresses this limitation with a new design for a modular dynamic linker and introduces Spidl, a prototype implementation. § 4 showcase a diverse range of load-time transformations that can be implemented with Spidl. As a whole, this paper seeks to raise awareness of dynamic linkers' ability to transform and progressively enhance operating systems while ensuring full backward compatibility.

2 Background on dynamic linking

This section provides useful background on dynamic linking and the ELF binary format. It emphasizes the dynamic linker's central role in modern systems and how it can be adapted to facilitate seamless system evolution while maintaining software backward compatibility. Next, the section provides an introduction to the ELF format, emphasizing its extensibility. We argue that the dynamic linker's significant control over program execution and interaction with the system, combined with the high extensibility of the ELF format, form a solid foundation to gradually enhance modern operating systems and unlock clean slate revolutionary designs.

2.1 The dynamic linker rules the system

Dynamic linking dates back to the 60s and was introduced as a core feature of the Multics system [16]. With the widespread adoption of Unix and its derivatives, dynamic linking has become a foundational element of modern operating systems and programming languages. Its role in modern software development is critical, as it not only facilitates the creation of flexible, modular, reusable, and maintainable software but also now plays an integral part in enhancing security through measures like address space layout randomisation (ASLR). Although self-relocating programs do exist, modern operating systems predominantly rely on a *dynamic linker/loader* to link and load dynamic programs along with their dependencies. On Ubuntu, for instance, more than 99% of binaries are dynamically linked [15].

When a program is instantiated through a system call, such as execve, the system first checks for the presence of an *interpreter* in the target program's ELF metadata (§ 2.2). For dynamic programs the interpreter is a dynamic linker, also called dynamic loader or simply loader, picked at compile time. Instead of transferring control directly to the target program, the system loads and hands over control to the interpreter. The dynamic linker then proceeds to configure the environment, loading the target program and all its dependencies, and ultimately transfers control to the program's entry point.

In practice, *the dynamic linker governs the system.* On Linux platforms, most programs rely on the single loader provided by the distribution, which runs before almost every program on the machine, holding the complete privileges associated with the process. It is a narrow waist, *i.e.*, a pivotal point of control, ideal for implementing *transformative changes* to our operating systems, that can be replaced and fine-tuned transparently with ease.

Although many systems, similar to Linux, depend on a single dynamic linker, the ELF format offers a straightforward way to choose between multiple linkers for each program. This capability allows for *progressive* enhancement without compromising backward-compatibility with the existing ecosystem.

2.2 ELF as a database format

This paper focuses on the ELF (*Executable and Linkable Format*) binary format [2] for storing programs and libraries. Widely adopted across Unix systems, particularly in Linux, the ELF format benefits from numerous tools and compiler directives to parse, display, manipulate, or extend binaries.

An ELF file is a document that represents the same program with two different views: one for linking, known as *sections*, and another for loading, known as *segments.* Figure 1 depicts both views of an ELF file, along with the resulting in-memory program. The ELF header provides essential details about the file, such as its architecture, operating system ABI,

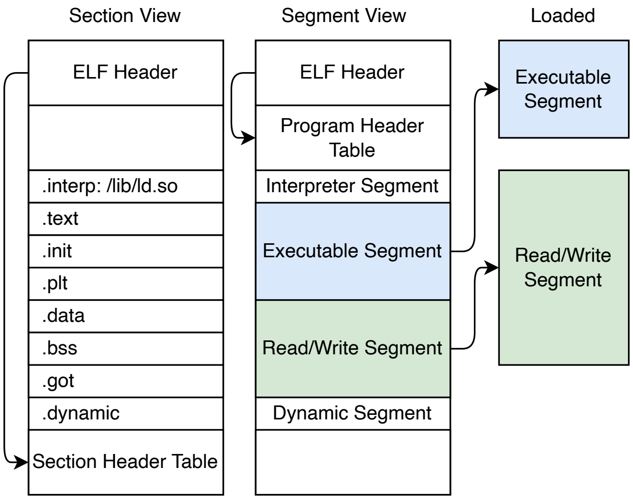

Figure 1. The different views of an ELF file: the sections, segments, and the in-memory layout.

entry point, and the location of other important metadata, including the program header table (for segments) and the section header table.

Sections represent logical units of information that relate to code, data, symbols (references to functions or variables), and relocations within the binary. Sections serve a crucial role during the linking and debugging process. The linker uses sections to resolve symbols, perform relocations, and combine object files to create the final executable.

Segments, on the other hand, are larger contiguous blocks of memory with associated access rights that describe a program's memory layout and content. They are used by the loader to *map* the binary into memory, *i.e.*, allocate virtual memory regions, copy the binary's bytes into them, and set up the program's runtime environment. Each segment contains one or more sections.

The ELF format is designed to be extensible. While the various header formats are fixed, their fields only have a handful of reserved values. For example, program headers (*i.e.*, segments) include a type field of 4-bytes for which only a dozen default values are reserved. Consequently, new loaders can utilize this field to define new segment types which traditional loaders would simply ignore. A similar observation holds for section headers.

The ELF format's flexibility and extensibility is already commonly used to include extra information. For example, the Debugging With Arbitrary Records Format (DWARF) [4] data is inserted inside the binary as a debug section and parsed by debuggers. In practice, ELF files can be extended with arbitrary data, may it be semantic information about the program, text, bytecode, or linker instructions. Some related work [23, 24] even embed full binaries inside dedicated ELF sections, which are then extracted and loaded into isolated nested execution environments by dedicated code in the main program's runtime.

3 A blueprint for a modular dynamic linker

The dynamic linker plays a crucial role in today's operating systems and, combined with the extensibility of ELF, could be used to explore new interactions between operating systems and programs. Unfortunately, dynamic linkers are often considered as arcane pieces of software that are hard to reason about and extend. They are complex, lack modularity, and too-tightly integrated with system software. For instance the GNU dynamic linker and libc are so intertwined that neither of those can function without the other.

In this section, we introduce a novel approach to dynamic linking, characterized by a modular and tunable framework design to extend existing systems while preserving software backward compatibility. We implemented a prototype of this linker design called Spidl. Spidl is written in Rust and targets Linux platforms. At the opposite of existing loaders, it does not make any assumptions about the structure of ELF files beyond the existence of the ELF header, program header, and section tables.

Spidl's design revolves around two core concepts: *phases* (§ 3.2) and *modules* (§ 3.1). As demonstrated on Figure 2, Spidl organizes binary linking and loading as a per-file pipeline of sequential phases. Modules define transformations called *passes*, assigned to Spidl's phases, which together perform the actual linking and loading of the executable and its library dependencies.

To demonstrate Spidl's compatibility with existing linker ecosystems, we developed a System V module. The System V module provides all the required passes to link and load standard executables (as described in the System V specification) and fully replaces a standard Unix dynamic linker. By combining modules, Spidl can augment or modify the interactions between the OS and programs, or even introduce completely different ABIs while retaining cross-compatibility with existing software.

3.1 Modules

A module in Spidl is a Rust `struct` that implements the `Module` trait (Rust's version of interfaces). A module must implement at least the `register` method. When loading a file, Spidl calls `register` on all available modules to populate the loading pipeline. A module participates in the loading of an object by either registering a pass for one or several phases, or by acting as a symbol resolver.

During registration, modules can communicate their intent by returning one of three values: *do not handle*, *handle weak*, or *handle strong*. Spidl provides modules with full visibility over the modules that precede them in the pipeline. The order of module registration is thus critical and must be carefully considered when configuring Spidl. Weak and

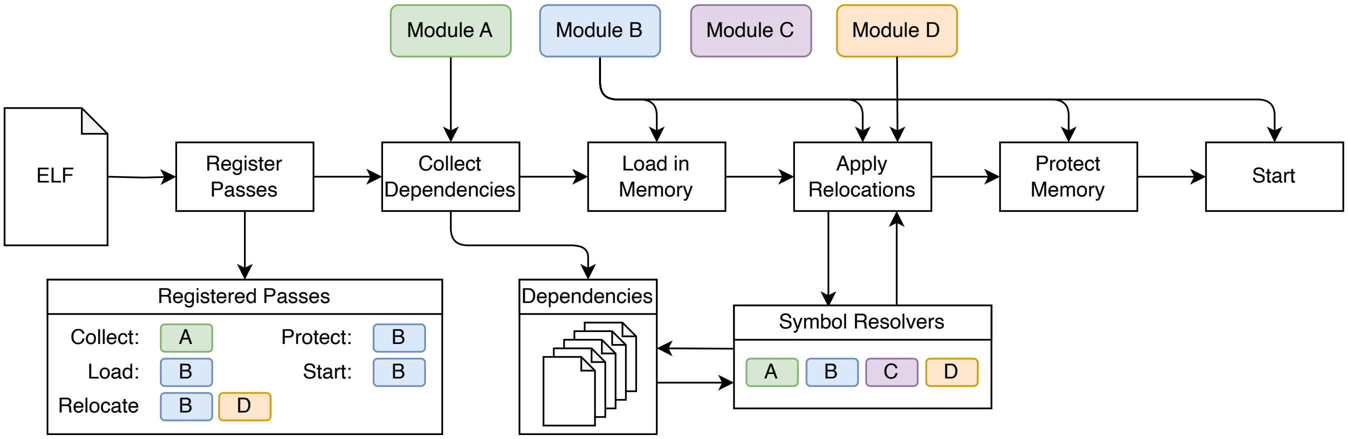

Figure 2. Loading of an ELF file with the Spidl dynamic linker. Modules first register passes for the different phases of the loading of the file that will then be scheduled by Spidl.

strong handling mark the distinction between complementary operations and overriding a pass. For example, implementing an LD_PRELOAD mechanism is straightforward and only requires a module to register a weak pass for the collect phase.

3.2 Phases

The Spidl loader defines a fixed set of phases that act as synchronization points between various modules. During each phase, registered modules execute a series of standard transformations, as outlined in Table 1. Moreover, additional "*post-phase*" phases are available to carry out module-specific tasks. For instance, the System V instantiates the thread local storage (TLS) in the post-relocate phase, as the TLS might be initialized with relocated data. Below, we elaborate on the different phases. In the remainder of this section, we use the term object and file interchangeably to reference an ELF binary (executable or library).

Register: When first opening a file, Spidl calls the register method on all modules, with the file as argument, to register passes to be applied in subsequent phases. Spidl allows modules to register passes for the main ELF program file as well as for any other file that might need to be opened as part of the loading (see collect). During the register phase modules indicate for each subsequent phase whether they intend to execute a pass, and if so whether it is the main pass for that phase or not. As there can be only a single main pass, this enables a module to override another.

Collect: The collect phase gathers the objects that need to be loaded for the proper execution of the binary. Registered modules can return a list of dependencies, either in the form of files to be opened or as new in-memory ELF objects. Spidl then repeats the register and collect phases for dependencies recursively, until all files are collected.

Load: During the load phase, modules perform both memory allocation and the actual loading of the objects into memory. The allocation and loading are done on a per-object basis, like all operations in the different phases. Hence, global optimisations such as those presented in iFed [41] require pre-computing allocation locations, but can still be fitted in the Spidl approach.

Relocate: The relocate phase performs dynamic linking. During this phase, modules apply *relocations*, *i.e.*, patch code or data with the actual memory location of loaded objects. Spidl differs from other dynamic linkers in how relocations are applied, as it enables cooperation between modules. Indeed, during relocation, a module might need to learn about the location of an item in another object (this is called *symbol resolution*), but the module itself might not be able to understand that other object's format. For instance, a C-based object might need to call a JavaScript function, but the System V module does not know about an hypothetical JavaScript object ELF encoding or runtime. For this purpose, Spidl mediates the symbol resolution by querying relevant modules for the requested symbol, thus removing the need for a module to know about all possible ABI used in the system and allowing it, instead, to focus on handling a single type of programs. The relocation process is illustrated in Figure 2.

Protect: During this phase the different memory protections are adjusted. After this, loaded objects can no longer be modified.

Start: The final phase applies only to the original ELF target and transfers control to the linked program's entry point after the last preparations.

3.3 Limitations

Spidl is still in early development and therefore has a few limitations. First, Spidl does not yet offer runtime services.

Phase	Description
register	Register modules for following phases
collect	Collect dependencies
post-collect	
load	Memory allocation and loading
post-load	
relocate	Apply the relocations
post-relocate	
protect	Apply memory protections
post-protect	
start	Transfer control to the program

Table 1. Phases in the Spidl dynamic linker.

Most dynamic linkers, such as GNU ld.so, tightly integrate with the libc to provide runtime services. Those services notably include the `dlopen` family of functions and the ability to create new thread local storage slots, required for multithreading. In the future, Spidl could be extended to register runtime services.

Second, the current implementation lacks support for GNU extensions, heavily used throughout the GNU libc and its ecosystem. As a result, Spidl cannot run programs compiled on GNU systems out of the box. This is not a limitation of Spidl but rather of our System V module. Two approaches to support the execution of glibc programs are possible: (1) creating a GNU module to replace the System V symbol resolution, or (2) adding a module to resolve glibc to another supported libc, such as musl-libc. Note that some of the GNU extensions, such as `IFUNC` and lazy binding, might be detrimental to the system security.

4 A new skin for the kernel

The dynamic linker can be used to disentangle operating system APIs and application development ecosystems. Spidl allows to progressively introduce changes to operating systems, ranging from simple optimisations to API-breaking OS re-designs, while maintaining compatibility with existing software.

In this section, we showcase a range of transformations that a dynamic linker can perform, organized in approximate order of how far they stray from traditional system interfaces. These transformations can all be realized as modules for the Spidl linker introduced in § 3, allowing for the combination and gradual integration of individual system extensions. This section is not meant to be an exhaustive list of transformations supported by Spidl. It rather aims at inspiring the exploration of operating system and programming language designs through a customizable linker.

4.1 Performance optimisations

iFed [41], a pass-based dynamic linker that supports global optimizations, already demonstrated the benefits of load-time program optimizations. iFed leverages a global view of the program, including all shared libraries, to implement two passes: dynamic libraries concatenation and relocation branch eliminations. The former consists in grouping segments from different libraries with similar access rights together to enable the use of huge pages and improve the TLB hit ratio, while the latter eliminates indirections caused by dynamic library calls by patching binaries. Together, these optimisations provide an 8.58% reduction of TLB miss and 3.28% branch miss-prediction reduction on x86_64, and a 13.04% and 1.85% reduction on ARM respectively. Spidl can trivially implement similar optimizations as a module.

4.2 Executable compression

In addition to performance, program size, both in-memory and on-disk, can be a limiting factor in environments with limited resources. To reduce the on-disk binary footprint, a traditional approach involves an executable-packer compression technique, wherein a program is modified to self-extract and decompress at launch time. However, this unpacking process could be simplified by implementing a decompression Spidl module, eliminating the need for complex self-relocation logic in the binary and further reducing both its in-memory and at-rest size.

Another widely deployed optimization is the Android dynamic relocation compression [1], which reduces RAM usage by up to 9%. Loading compressed programs requires special assistance from the linker and is so far only available on Android. Spidl could provide a generic cross-platform module with a relocate phase pass, similar to module D on Figure 2, to export this functionality beyond Android.

4.3 Load-time validation

Load-time is an apt moment to validate properties of programs and libraries. In addition to verifying program and library signatures, the dynamic linker can conduct more sophisticated checks, such as validating proof-carrying code [37] or type-checking interfaces.

Consider the case of type-checking interfaces. Present-day compilers go to great lengths to ensure that if a program compiles, it will execute correctly (*e.g.*, memory and type safety, ABI/API compatibility). These guarantees are, however, limited to the scope visible to the compiler. On current systems, the dynamic linker can link two pieces of code with incompatible interfaces without any remorse. This may occur following a library upgrade for example. Such bugs could be easily detected by checking for ABI-compatibility at load time, provided relevant type information is available. Fortunately, compilers already emit type-level information in the form of DWARF [4]. This information, usually used

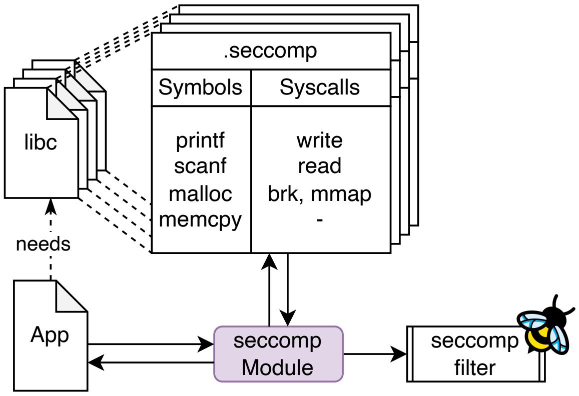

Figure 3. Scalable seccomp sandboxing. A seccomp module aggregates the authorized system calls based on symbol resolution and a per-file .seccomp section.

by debuggers, can also be leveraged by the dynamic linker to ensure ABI-compatibility and, in the case of languages like Rust, validate pointers' lifetime.

4.4 Sandboxing

Sandboxing restricts the resources that a software component or a full program can access. Effective sandboxing requires application-level knowledge to define appropriate policies. Nowadays most approaches rely on expensive whole-program analysis through compiler instrumentation [12, 22, 31]. Those approaches are usually applied to a few applications on the system and do not scale to the whole OS, especially when considering that any library upgrade requires to re-run expensive analysis for all dependent programs.

The dynamic linker is the ideal layer to make existing sandboxing approaches more scalable, by combining per-library analysis dynamically at load-time. To illustrate the benefits of dynamic linkers, consider how to leverage seccomp for sandboxing. Seccomp [30] is a Linux system call that enables a process to gradually reduce its access to the system call API. Building an appropriate seccomp eBPF system call filter requires fine-grained analysis of the entire application. A Spidl module could remove the need for costly whole-program analysis by leveraging pre-computed per-library analysis (mapping from function symbols to required system calls) and combining them into a custom made filter at load time, as illustrated in Figure 3. This approach would allow to: (1) only perform per-library analysis once and reuse the result across applications, and (2) upon a library update, limit the static analysis' scope to the modified component. It could further be extended to support seccomp-based sandboxing at the scale of a Linux distribution.

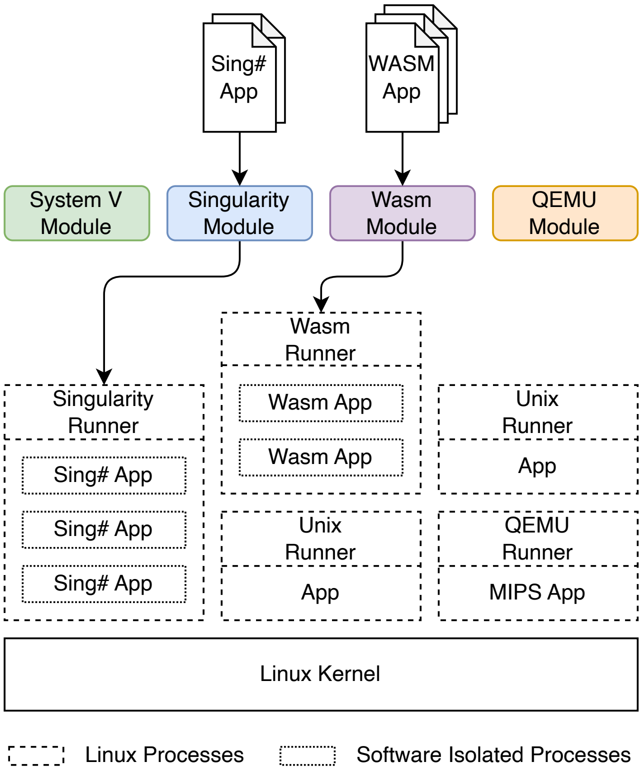

Figure 4. Building a runner abstraction on top of Linux. Each module is capable of loading one type of file and configures or re-uses an appropriate environment prior to program start.

4.5 Compartmentalization

Compartmentalization is a staple of system and security research communities. Proposed solutions include operating system extensions [10, 26, 35], compiler instrumentation [43, 45] and, more recently, support for new hardware security extensions such as Intel MPK [28] and SGX [17] taking various forms [8, 13, 19, 25, 32, 34, 36]. However, compartmentalization solutions can require various modifications to the software stack, from user program modifications to leverage new isolation abstractions [10, 23, 24, 34, 35], to the reimplementation of a loader inside the program itself [23, 24]. Integrating the specialized loading logic into the dynamic linker would be more fitting and could facilitate transparent compartmentalization for libraries, *i.e.*, without application code modification, akin to the FlexOS approach [33].

4.6 Customized Runners

Since Unix was initially rewritten in C, Unix-based systems have consistently exposed a C environment to user programs. However, the C language imposes rigid rules that programs must follow, such as adhering to the calling convention, accounting for arbitrary memory accesses, unbounded stack usage by C functions, dealing with threads, TLS storage, content of system registers, and following libc/Unix abstractions.

A more flexible approach pioneered by Fuchsia [6] is the concept of *runners* [5]. Runners provide execution environments tailored for specific needs and might impose various constraints and ABI. Examples from the Fuchsia documentation include an ELF runner providing a familiar C environment, and a web runner capable of running and sandboxing web applications. The runner abstraction can provide compatibility across systems, for instance one can implement a Singularity [27] or an Oberon [44] runner on top of Fuchsia.

The dynamic linker is uniquely positioned to provide a runner abstraction on top of existing kernels, such as Linux, by transforming the Unix environment provided by the kernel prior to program execution. For instance, loading a JavaScript or WebAssembly [43] program would configure the language virtual machine transparently, while a Sing# [21] program would be loaded into an existing Singularity runner as a *software isolated process* [11, 27].

Figure 4 illustrates how the runner abstraction can be implemented using Spidl. Each runner exposed by the operating system is paired with a Spidl module that understands how to load programs for this runner. Because Spidl enables cooperation between modules during symbol resolution (§ 3.2) such a system would benefit from transparent interoperability across languages and runners, dethroning C as the *lingua franca* of programming languages.

5 Discussion

Despite the dynamic linker's pivotal role in the system, there have been limited efforts to provide an extensible dynamic loading and linking infrastructure, even though the need to modify loading behavior arises frequently [9, 14, 23–25, 39]. To the best of our knowledge, iFed [41] stands as the first attempt at a genuinely modular approach to dynamic linking, offering extensibility and straightforward code reuse.

While Spidl shares similarities with iFed, it distinguishes itself by presenting a truly clean slate approach, facilitating the green field development of operating system extensions and load-time instrumentation/transformation. Unlike iFed, which relies on a fixed intermediate representation built on top of the GNU loader and tightly integrates with libc, Spidl makes no assumptions about the program being loaded and its execution environment. Spidl overcomes the difficulties of linking a wide variety of objects together by mediating cooperation between specialized modules.

We hope that availability of modular dynamic linkers, such as Spidl, will encourage the community to explore new research avenues at the interface between programs and operating systems. We believe that dynamic linkers will be transformative for areas such as security and interoperability across languages, runtimes, and isolation boundaries. Indeed, we see tremendous potential for dynamic linkers to benefit from compiler-generated ABI, types, and invariants information, as well as user-provided isolation requirements. With Spidl as a playground, we plan to explore new ELF extensions and develop an ecosystem of modules that can progressively augment and customize today's operating systems.

6 Conclusion

While the inertia and huge software ecosystem of existing operating systems makes it hard for academia to move innovative ideas from the lab to mainstream systems, we argue that dynamic linkers can ease the transition. Dynamic linkers execute on the initialization path of any program on the system, and can perform extensive transformations to drastically improve performance, harden security, and expose tailored system interfaces. Current commodity dynamic linkers are monolithic and deeply intertwined with system libraries, and therefore hard to modify and extend. We introduce Spidl, a modular dynamic linker designed for extensibility and experimentation that can serve as a basis for the exploration of new interactions between programs and operating systems. Finally, we present a wide variety of transformations that can be performed at load time, and hope to inspire further research along that direction.

7 Acknowledgements

The authors thank the anonymous PLOS'23 reviewers for their constructive and helpful feedback.

References

[1] Android relocation packer. https://android.googlesource.com/platform/bionic/+/f5e0ba94d911ef2622ecfd3f7fabc4432a4806d3/tools/relocation_packer/README.TXT.

[2] Tool interface standard (tis) executable and linking format (elf) specification. https://refspecs.linuxfoundation.org/elf/elf.pdf, 1995.

[3] *Be developer's guide - the official documentation for the BeOS.* O'Reilly, 1997.

[4] Dwarf debugging information format version 5. https://dwarfstd.org/doc/DWARF5.pdf, 2017.

[5] Fuchsia component runners. https://fuchsia.dev/fuchsia-src/concepts/components/v2/capabilities/runners, 2023.

[6] The fuchsia operating system. https://fuchsia.dev/, 2023.

[7] The rust programming language, 2023.

[8] Baumann, A., Peinado, M., and Hunt, G. C. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst. 33*, 3 (2015), 8:1–8:26.

[9] Belay, A., Bittau, A., Mashtizadeh, A. J., Terei, D., Mazières, D., and Kozyrakis, C. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI)* (2012), pp. 335–348.

[10] Bittau, A., Marchenko, P., Handley, M., and Karp, B. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI)* (2008), pp. 309–322.

[11] Boos, K., Liyanage, N., Ijaz, R., and Zhong, L. Theseus: an Experiment in Operating System Structure and State Management. In *Proceedings of the 14th Symposium on Operating System Design and Implementation (OSDI)* (2020), pp. 1–19.

[12] Canella, C., Werner, M., Gruss, D., and Schwarz, M. Automating Seccomp Filter Generation for Linux Applications. pp. 139–151.

[13] Castes, C., Ghosn, A., Kalani, N. S., Qian, Y., Kogias, M., Payer, M., and Bugnion, E. Creating Trust by Abolishing Hierarchies. In *Proceedings of The 19th Workshop on Hot Topics in Operating Systems (HotOS-XIX)* (2023), pp. 231–238.

[14] che Tsai, C., Arora, K. S., Bandi, N., Jain, B., Jannen, W., John, J., Kalodner, H. A., Kulkarni, V., Oliveira, D., and Porter, D. E. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the 2014 EuroSys Conference* (2014), pp. 9:1–9:14.

[15] che Tsai, C., Jain, B., Abdul, N. A., and Porter, D. E. A study of modern Linux API usage and compatibility: what to support when you're supporting. In *Proceedings of the 2016 EuroSys Conference* (2016), pp. 16:1–16:16.

[16] Corbató, F. J., and Vyssotsky, V. A. Introduction and overview of the multics system. In *AFIPS Fall Joint Computing Conference (1)* (1965), pp. 185–196.

[17] Costan, V., and Devadas, S. Intel SGX Explained. *IACR Cryptol. ePrint Arch. 2016* (2016), 86.

[18] Dorward, S. M., Pike, R., Presotto, D. L., Ritchie, D. M., Trickey, H. W., and Winterbottom, P. The Inferno™ operating system. *Bell Labs Tech. J. 2*, 1 (1997), 5–18.

[19] Ferraiuolo, A., Baumann, A., Hawblitzel, C., and Parno, B. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)* (2017), pp. 287–305.

[20] Finley, K. This os almost made apple an entirely different company, 2015.

[21] Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G. C., Larus, J. R., and Levi, S. Language support for fast and reliable message-based communication in singularity OS. In *Proceedings of the 2006 EuroSys Conference* (2006), pp. 177–190.

[22] Ghavamnia, S., Palit, T., Mishra, S., and Polychronakis, M. Temporal System Call Specialization for Attack Surface Reduction. pp. 1749–1766.

[23] Ghosn, A., Kogias, M., Payer, M., Larus, J. R., and Bugnion, E. Enclosure: language-based restriction of untrusted libraries. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVI)* (2021), pp. 255–267.

[24] Ghosn, A., Larus, J. R., and Bugnion, E. Secured Routines: Language-based Construction of Trusted Execution Environments. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)* (2019), pp. 571–586.

[25] Hedayati, M., Gravani, S., Johnson, E., Criswell, J., Scott, M. L., Shen, K., and Marty, M. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)* (2019), pp. 489–504.

[26] Hsu, T. C.-H., Hoffman, K. J., Eugster, P., and Payer, M. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2016), pp. 393–405.

[27] Hunt, G. C., and Larus, J. R. Singularity: rethinking the software stack. *ACM SIGOPS Oper. Syst. Rev. 41*, 2 (2007), 37–49.

[28] Intel. Intel 64 and ia-32 architectures software developer's manual., 2022.

[29] Kernighan, B. W., and Ritchie, D. *The C Programming Language.* Prentice-Hall, 1978.

[30] Kim, T., and Zeldovich, N. Practical and Effective Sandboxing for Non-root Users. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)* (2013), pp. 139–144.

[31] Kirth, P., Dickerson, M., Crane, S., Larsen, P., Dabrowski, A., Gens, D., Na, Y., Volckaert, S., and Franz, M. PKRU-safe: automatically locking down the heap between safe and unsafe languages. pp. 132–148.

[32] Lee, D., Kohlbrenner, D., Shinde, S., Asanovic, K., and Song, D. Keystone: an open framework for architecting trusted execution environments. In *Proceedings of the 2020 EuroSys Conference* (2020), pp. 38:1–38:16.

[33] Lefeuvre, H., Badoiu, V.-A., Jung, A., Teodorescu, S. L., Rauch, S., Huici, F., Raiciu, C., and Olivier, P. FlexOS: towards flexible OS isolation. pp. 467–482.

[34] Lind, J., Priebe, C., Muthukumaran, D., O'Keeffe, D., Aublin, P.-L., Kelbert, F., Reiher, T., Goltzsche, D., Eyers, D. M., Kapitza, R., Fetzer, C., and Pietzuch, P. R. Glamdring: Automatic Application Partitioning for Intel SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)* (2017), pp. 285–298.

[35] Litton, J., Vahldiek-Oberwagner, A., Elnikety, E., Garg, D., Bhattacharjee, B., and Druschel, P. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)* (2016), pp. 49–64.

[36] McCune, J. M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V. D., and Perrig, A. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy* (2010), pp. 143–158.

[37] Necula, G. C. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)* (1997), pp. 106–119.

[38] Pike, R. Systems software research is irrelevant. *Invited talk* (2000).

[39] Porter, D. E., Boyd-Wickizer, S., Howell, J., Olinsky, R., and Hunt, G. C. 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-XVI)* (2011), pp. 291–304.

[40] Presotto, D., Pike, R., Thompson, K., and Trickey, H. Plan 9, a distributed system. *Proc. of the Spring* (1991), 43–50.

[41] Ren, Y., Zhou, K., Luan, J., Ye, Y., Hu, S., Wu, X., Zheng, W., Zhang, W., and Hu, X. From Dynamic Loading to Extensible Transformation: An Infrastructure for Dynamic Library Transformation. In *Proceedings of the 16th Symposium on Operating System Design and Implementation (OSDI)* (2022), pp. 649–666.

[42] Ritchie, D., and Thompson, K. The UNIX Time-Sharing System. *Commun. ACM 17*, 7 (1974), 365–375.

[43] Rossberg, A., Titzer, B. L., Haas, A., Schuff, D. L., Gohman, D., Wagner, L., Zakai, A., Bastien, J. F., and Holman, M. Bringing the web up to speed with WebAssembly. *Commun. ACM 61*, 12 (2018), 107–115.

[44] Wirth, N., and Gutknecht, J. *Project Oberon - the design of an operating system and compiler.* Addison-Wesley, 1992.

[45] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy* (2009), pp. 79–93.