From Dynamic Loading to Extensible Transformation: An Infrastructure for Dynamic Library Transformation

Yuxin Ren, Kang Zhou, Jianhai Luan, Yunfeng Ye, Shiyuan Hu, Xu Wu, Wenqin Zheng, Wenfeng Zhang, and Xinwei Hu,
Poincare lab, Huawei Technologies Co., Ltd, China

https://www.usenix.org/conference/osdi22/presentation/ren

This paper is included in the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-28-1

Open access to the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

From Dynamic Loading to Extensible Transformation: An Infrastructure for Dynamic Library Transformation

Yuxin Ren, Kang Zhou, Jianhai Luan, Yunfeng Ye,
Shiyuan Hu, Xu Wu, Wenqin Zheng, Wenfeng Zhang, Xinwei Hu
Poincare lab, Huawei Technologies Co., Ltd, China

Abstract

The dynamic linker and loader has been one of the fundamental software, and more than 99% of binaries are dynamically linked on Ubuntu. On one hand, vendors are going to break production software into more and more dynamic libraries to lower the maintenance cost. On the other hand, customers require the dynamic loader to provide rich functionalities to serve their isolation, security, and performance demands. However, existing dynamic loaders are implemented in a monolithic fashion, so they are difficult to extend, configure and optimize.

This paper presents iFed, an infrastructure for extensible and flexible dynamic library transformation. We design iFed in a pass-based architecture to compose various functional and optimization passes. iFed uses a runnable in-memory format to represent libraries and coordinate among multiple transformation passes. We further implement two optimization passes in iFed, which efficiently leverages hugepages and eliminates relocation overhead. iFed is implemented as a drop-in replacement of the current system default dynamic loader. We evaluate iFed and its optimization passes with a wide range of applications on different hardware platforms. Compared to the default `glibc` dynamic loader, iFed reduces an order of magnitude of TLB miss. We improve the throughput of a dynamic website by 13.3%, along with a 12.5% reduction of tail latency without any modifications to the applications.

1 Introduction

Since the 1990s, dynamic linkers and loaders have been one of the most critical software tools for computer programs and applications [11, 15, 23]. Opposite to static linking, which generates a single big application binary, dynamic loading [1] permits complex software to be shipped, delivered, and distributed as a collection of libraries, modules, or components. For low-level languages, such as C/C++ and Rust, these components are implemented as dynamic libraries, also called dynamic-link libraries (`.dll` in Windows) or shared objects (`.so` in Linux). Only when a program starts will its dynamic libraries be integrated to form a runnable application by the dynamic loader. In this way, each dynamic library can be distributed and patched individually without modifying the entire application. As a result, software maintenance cost is greatly reduced while it gains much more flexibility. A study shows that more than 99% of binaries are dynamically linked on Ubuntu [46].

While the dynamic loader's functional structure has been mature and stable for more than one decade, we found it cannot meet the requirements of rapidly developing software and complicated architectures today. Two primary driving factors call out a new infrastructure for extensible and modular transformation on dynamic libraries: (1) the massively increasing number of dynamic libraries used in an application and (2) the emerging diversity of manipulation and operations on dynamic libraries.

Complex commercial software heavily relies on dynamic libraries to decompose a single huge binary into many loosely-coupled, fine-grained modules. This is particularly motivated by two considerations. First, some open source license requires all statically linked code should also be open-sourced. This is so-called "license contamination". GPL license [12] (used by `glibc`) is one such example. Consequently, production software has to use dynamic libraries to avoid "license contamination".

Second, modern software needs frequent updates because of CVE fixes, bug fixes, or adding new features. However, it is painful for vendors to re-compile or link the whole software, and ask customers to reinstall the entire application. Therefore, vendors always break up software into many fine-grained dynamic libraries, and each library can be maintained,

[1] Dynamic loading is also referred to as run-time loading, a mechanism that an application opens, loads, and executes a library by explicitly calling loader interfaces during program execution. As run-time loading shares almost the same backend technology with dynamic loading, throughout this paper, we use dynamic loading to refer to the integrated linking and loading phase when programs are launched.

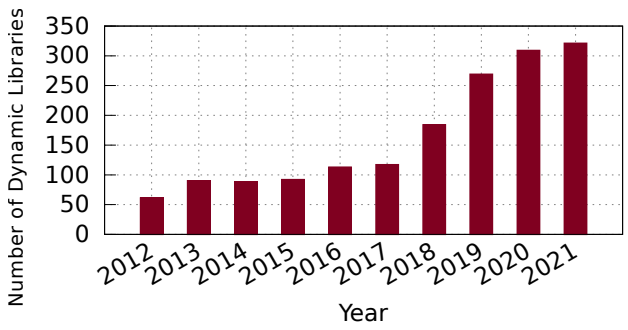

Figure 1: The number of dynamic libraries included in the CUDA Toolkit over the past decade.

updated, or replaced independently. For instance, Figure 1 lists the number of dynamic libraries shipped in the CUDA Toolkit. As it shows, the number has grown rapidly over the last decade. Based on our observation from the industry, this trend will continue in the future.

Along with the growing dynamic library count, the dynamic loader is required to provide more features to make better use of emerging hardware and software technologies. For example, when using recent hardware memory protection (*e.g.* Intel MPK [17] and SGX [16]) to achieve in-process isolation, the dynamic loader has to perform more work. It loads isolated libraries into different memory regions, setups up memory protection and permission properties, and optionally verifies the signature of loaded binaries [7, 14, 37, 38, 40]. Load time code randomization and binary rewriting, provided by the dynamic loader, are widely adopted for profiling, security hardening, and architectural adaptation [51, 53, 54]. Library debloating relies on the dynamic loader to examine and eliminate unused library code from program memory [33, 35]. Control-Flow Integrity (CFI) and Sandbox also require miscellaneous modifications to the dynamic loader, such as analyzing relocation entries and overwriting the entry point [24, 47, 58].

However, the current dynamic loading infrastructure is insufficient and inefficient to offer rich functionalities over a large number of dynamic libraries. This leads to ad-hoc changes to the dynamic loader to satisfy various requirements from different productions. Such customized modifications are incompatible with each other, and cannot be integrated or reused, causing enormous development and maintenance costs. Even worse, the fundamental infrastructure of dynamic loader has been kind of ignored by academia and industry. Thus neither research nor open source community proposes systematic solutions to deal with these issues. For instance, while there are 100+ commits in `glibc` related to the dynamic loader in the last two years, they are almost bug fixes or cleanup without new features developed.

According to our many years' industry experience and realistic production requirements, intrusive and customized modifications cause unacceptable maintenance cost. On the one hand, a large number of source code patches are hard to be accepted by upstream. This also happens to academia work listed above. On the other hand, production departments do not have enough source code level knowledge to maintain patches. Therefore, it is painful for the OS department to maintain many ad-hoc patches and sometimes it has to release different OS distributions with different loaders (along with `glibc`). As a result, it motivates a new infrastructure which satisfies following requirements:

- It offers more functionalities than existing loader.
- It can be flexibly configured for different trade-off and extended to adopt future enhancements.
- Its modifications can be implemented in a modular way that minimizes the effort to align with upstream and fix conflicts due to patch maintenance.

In summary, the issue of current loader design is that it has no interface to allow extensions, thus intrusive modifications cannot be avoided. The loader is historically designed for a few simple functionalities and acts as a "translator". However, now it has to be redesigned, instead of re-engineer, to adopt emerging functionalities and allow future updates in a modular and flexible way, and becomes another platform for application optimization.

We address these challenges by designing iFed, a new infrastructure that achieves extensibility, modularity, and flexibility for dynamic library operations. Our key idea is to organize the iFed as a pipeline of distinct transformation passes instead of a monolithic tool. Each pass only implements some specific manipulation on dynamic libraries to realize its desired functionality, such as security enhancement, memory isolation, or performance optimization. We also design a runnable in-memory format (RiMF) to describe the runtime status and properties of an application and its dynamic libraries (§3.4). RiMF serves as an intermediate representation that every pass operates on, thus different passes are decoupled. By including complete status and information of all dynamic libraries, RiMF further enables iFed passes to do global and aggressive analysis and optimizations. A pass manager orchestrates the series of passes to be applied upon program launch (§3.5). Combined, these features produce the first infrastructure, as far as we know, that satisfies diverse functional requirements without loss of extensibility, flexibility, and modularity.

With various transformation passes plugged in, iFed is able to support much richer features beyond existing dynamic linking and loading. We demonstrate this by implementing two performance optimization passes. The first pass combines the same type of sections from different dynamic libraries into a continuous one, and then leverages hugepages to load the combined section (§3.6). The second one converts relocation branches into direct function calls, thus reducing the overhead of cross-library function calls (§3.7). iFed and its optimization passes are implemented to replace the GNU dynamic loader. We evaluate iFed with a large range of application

benchmarks on different architectures. The results illustrate how iFed optimization passes offer better throughput, latency, and predictability than current dynamic loaders. Without any modifications to the applications in a dynamic website, iFed improves the throughput by 13.3% and reduces the average end-to-end response time by 12.5%.

Our contributions are not only enhancing current loader with some specific optimizations, but also proposing a new infrastructure that is capable to host many other loader features in production. Concretely, the contributions of this paper include:

- We introduce iFed, a pass-based infrastructure for extensible, flexible, and modular transformation on dynamic libraries during load time.
- We design two performance optimization passes in iFed. One pass enables efficient utilization of hugepages by rearrangement and concatenation of multiple libraries. The other pass aggressively eliminates the overhead of cross-library invocations resulting from inefficient relocation.
- We implement iFed infrastructure with the above optimization passes as a drop-in replacement of the default dynamic loader in Linux (`ld.so` in `glibc`). iFed is fully compatible with ld.so and its all interfaces.
- An exhaustive evaluation of iFed on different architectures with a wide range of applications.

The rest of this paper is organized as follows. §2 provides background and motivation for the redesigned dynamic loaders. §3 introduces iFed and discusses its design, while §4 details the implementation of iFed. In §5, we present the performance evaluation of iFed for a wide range of applications. §6 discusses the related work, and §7 concludes.

2 Background and Motivation

2.1 Insufficient Functionality

The basic functionalities of dynamic loading include three parts: (1) library lookup and collection; (2) memory layout preparation; and (3) symbol resolution and name binding. The core jobs to implement these functionalities in existing dynamic loaders are simple. The loader allocates memory and maps libraries into the address space with the given layout specified in library object files. Then it resolves external symbols by populating some lookup tables with the actual memory address. While these steps are just enough to execute programs with dynamic libraries, they are not able to further transform libraries to meet diverse isolation, security, and execution requirements. Thus, many projects have to customize the loader to fulfill their system objectives. We list a few examples here.

- CubicleOS [38] is a library OS that isolates components in MPK protected memory regions, called cubicles. It implements a new cubicle loader who acts as the dynamic loader. The loader is responsible for cubicle creation and component loading. It additionally scans binaries to ensure that there are no any MPK-related operations, and resoles cross-cubicle calls with special trampolines.
- BlankIt [33] is a dynamic loading framework that predicts and loads only the set of library functions that will be used by the application. At load time, BlankIt iterates over all executable's dynamic libraries, wipes out unused functions it predicates, and overwrites these functions with a mispredication trampoline.
- Shuffler [53] patches the loader to support continuous code re-randomization. The modified loader implements constructor prioritization in multiple libraries, and employs binary rewriting to track and update all code pointers.

In summary, while many projects illustrate the necessity and benefit of loader modification, they have to do some redundant work, yet their own work cannot be easily integrated by others. Hence, a new infrastructure for extensible and modular dynamic loading is necessary.

	TLB miss	IPC	99th percentile latency (cycle)	Execution time (s)
`glibc`	1,231,950	1.96	318	6.01
iFed	117,782	2.43	232	4.86

Table 1: Performance comparison between `glibc` and iFed on x86 machine.

2.2 Inefficient Performance

Even worse, current dynamic loaders fail to effectively utilize modern hardware capabilities and global system resources, resulting in sub-optimal performance. A representative case is ineffective hugepage usage.

The current loader loads each dynamic library individually, and within each library, maps code and data section randomly. Thus sections are likely loaded into fragmented memory which only uses small pages (4K) for physical memory. This leads to more TLB miss, slower library function calls, and unpredictable execution time. A better loading strategy is combining the same sections of all libraries into a big one, and loading it into hugepage memory. We study performance penalties incurred by the current loader from `glibc`. On an Intel machine, we conduct a micro-benchmark that simply invokes 100 dynamic libraries, and each library contains only one function accessing memory (full details in §5). Table 1 depicts the micro-architecture impact of (instruction) TLB miss and instruction per cycle (IPC), as well as benchmark results of 99th percentile library function call latency and total execution time. Due to loading libraries with small pages, the benchmark suffers frequent TLB miss, which further leads to slow and unpredictable execution. In contrast, dynamic library concatenation pass in iFed effectively loads libraries

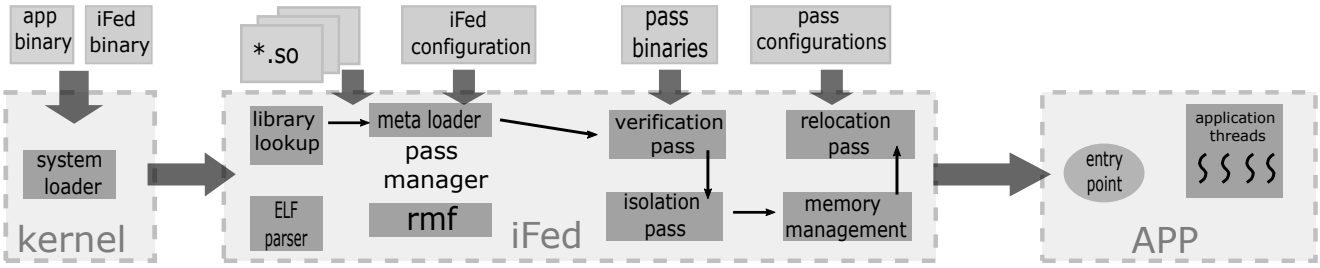

Figure 2: iFed architecture and workflow. The pass manager loads and invokes a series of transformation passes, which interact with RiMF. The workflow of program launch starts from the operating system kernel, which loads both application and iFed binary. After iFed gets the control, it discovers, parses and transforms dynamic libraries and finally boots up the application.

into hugepages, providing an order of magnitude reduction on TLB miss and 23.6% improvement on execution time. Next, we discuss how iFed enables more optimization and transformation of dynamic libraries in a modular and flexible way.

3 iFed Design

3.1 Design Principles

iFed integrates the lessons we learned from the experience of supporting diverse production demands on the dynamic loader. We below outline the key principles, the guidance throughout iFed design.

Extensibility and Modularity (P1). Due to different security or performance considerations, different production always requires a distinct subset of loader features. Therefore, various functionality should be organized in a loosely-coupled way instead of a monolithic implementation. Additionally, iFed should allow applying new features easily without intrusive modification to the loader itself.

Flexibility and Customizability (P2). It is desirable that iFed capabilities can be customized on per-application, customer, or even per-run basis. Such flexibility is important for system managers and end customers to have more control over running applications, opposite to accepting everything from the current loader passively.

Compatibility and Transparency (P3). Compatible with the existing loader interface is critical for iFed to be production-ready. Changes to the loader should be transparent to application developers, and require minimal modification of legacy code. Thus, we aim to design iFed as a drop-in replacement for the existing loader from the beginning.

3.2 iFed Functionality and Usage

As discussed in §2.1, current dynamic loaders cannot keep up with application demands on new functionalities. According to these demands, we summarize the desired features a loader should provide beyond existing ones.

- **Memory management.** The loader should be responsible for memory allocation, library address space layout, and content initialization. This has a large impact on application performance or memory consumption. Some examples of load time memory management are library debloating [33, 35], replaying the profiled hot regions [28], and hugepage optimization (§3.6).
- **Isolation.** The loader is the first place to partition and load different libraries into isolated regions. The customers' strong demand to isolate untrusted or vulnerable third-party libraries paired with the emerging MPK and SGX technologies, motivate the loader to offer more isolation capabilities [7, 14, 40, 50] beyond the traditional read/write/execute permission restrictions.
- **Security enhancement.** The loader is convenient to perform transparent security hardening regardless of running applications. For instance, we can enable CFI or sandbox [5, 24, 47, 58], apply code randomization [26, 51] or perform binary encryption/decryption or signature verification [25, 56].
- **Binary rewriting and execution control.** In addition to traditional relocation, the loader is feasible to perform more advanced binary rewriting and control program execution, such as Shuffler [53] and Egalito [54]. Furthermore, load time transformation is also necessary to migrate applications among heterogeneous environments or offload execution to smart devices [8, 52]. We will discuss a relocation elimination pass in iFed in §3.7.

Current usage of dynamic loader is a mass of interplay among build toolchains, such as compiler and linker. Some configurations and functionalities are scattered in various parts. For example, to prevent GOT overwrite attack [18], the following `gcc` options are widely used: `-Wl,-z,relro,-z,now`. `gcc` passes these options down to the linker, but these options do not take effect until the dynamic loader marks the corresponding memory region as read-only. However, existing usage is not appropriate. We argue that the dynamic loader should be hidden from application developers, but configured and controlled totally by end users or system administrators. This is because customers do not trust that developers properly

build the software to meet their requirements. Thus, iFed consolidates all loader-related operations in one place, and gives the control to end users who actually run the application.

3.3 iFed Architecture

When designing a loader, we should separate functional modules from low-level infrastructure. Functional modules will impact the application run-time behavior and the infrastructure orchestrates these modules. Furthermore, functional modules can be easily replaced or combined without intrusive modifications to the infrastructure and other modules.

We choose pass-based architecture for the loader design. As a result, source code patches are no longer needed, and independent modules with enough semantics can be developed, configured and maintained. The overall iFed architecture is shown in Figure 2. The core component in iFed is a series of transformation and optimization passes that manipulate and transform dynamic libraries for various purposes related to security, isolation, and performance. Each pass is a separate module which can be enabled or disabled independently. Such pass-based modular architecture gives great flexibility and extensibility to users to customize iFed functionality according to their own demands. All passes are managed and controlled by a pass manager (§3.5). Users configure the pass manager to instruct it to construct and execute the pass pipeline. The pass manager also maintains all libraries' in-memory status, and organizes them using RiMF format (§3.4). RiMF is an intermediate representation that is shared by all passes. In this way, passes are able to retrieve global information scatted in many libraries and to perform advanced inter-library transformations. Same as the existing loader, iFed offers other utility components as well, such as library discovery and `elf` parser.

3.4 Runnable In-memory Format

A main goal of iFed is splitting the current monolithic dynamic loader into extensible passes. On the one hand, it is desirable that a pass does not rely on another, thus enabling different passes to be developed and evolve independently. On the other hand, when multiple passes run together, they should be aware of how others transform libraries. Hence, we need a kind of intermediate representation that captures all libraries' status originating from library objects and generated by iFed passes on the fly. Runnable in-memory format (RiMF) is intended to coordinate iFed passes by providing a central place to hold library information at load time.

Passes in iFed do not communicate with each other directly, instead, the shared RiMF is the only interface for library transformation any pass can use. In this way, RiMF hides iFed internal complexity and other pass's implementation details to pass developers. Currently, whenever modifying the dynamic loader to add new features, a developer has to understand most of its codebase, even though much of them are irrelevant. In contrast, all a developer needs to know to write a transformation pass in iFed is the format and properties inside RiMF, and the operations it exposes. iFed maintains a single RiMF image which includes all dynamic libraries a program requires, instead of a separate object file for every library as today. Thus, iFed pass has more opportunities to apply global analysis and optimization. Our dynamic library concatenation pass demonstrates the power of global RiMF. Different from ELF object file which is designed for the dense on-disk format, RiMF rather focuses on load time in-memory representation, such as isolation constraints, memory placement and attributes, and code interposition.

The first-class object in RiMF is the isolation domain, which composes a subset of libraries within the same protection boundary. The actual isolation domain implementation depends on the iFed pass. It could be implemented by MPK, SGX or even device offloading. At the top level, RiMF consists of a list of isolation domains, inter-domain invocations that need to be resolved specially and a global application entry point. Inside each isolation domain, similar to an ELF file, RiMF provides sections, exposed symbols, and relocation records. These information are organized in a set of tables. Primary tables provided by RiMF are: (1) memory-mapping tables which describe library address space layout and memory attributes; (2) symbol tables dealing with symbol definition, binding, reference, and so forth; (3) section metadata tables that associate RiMF sections to original ELF object files. A RiMF section does not contain the actual binary, but maps to one or more ELF sections initially. RiMF varies throughout the iFed transformation pipeline. RiMF exports multiple interfaces to query, insert, modify and commit its internal tables. For example, a pass can update section metadata tables to combine different ELF sections into a new RiMF section. By manipulating symbol tables, a pass is able to remove unused code or override a function call with a customized trampoline. The commit interface is used to apply table modifications to the actual binary, such as interposing them in the library code and loading sections to memory.

3.5 iFed Pass Manager

The iFed pass manager orchestrates transformation passes to operate on RiMF sequentially. The pass manager takes a user-provided configuration file and invokes each pass accordingly. In essence, the pass manager is mainly responsible for two tasks. First, the pass manager maintains the RiMF image and provides interfaces to various passes to query and modify RiMF. Second, the pass manager acts as a meta loader, which loads and executes each transformation pass. Consistent with iFed overall design principle, each transformation pass is also implemented as a dynamic library, which needs to be loaded before execution as well. For simplicity, we reuse the existing `glibc` loader for this minimal meta loader, so any transformation trick is not applied to pass libraries.

Current iFed does not contain a sophisticated scheduling policy for running passes nor supports parallel pass execution. We leave these as future work. Thus, the user has to explicitly deal with pass dependency and pass confliction in the configuration file.

Pass Dependency. In general, passes are not aware of each other because they only use RiMF as the communication medium. However, the order of passes impacts the runtime overhead a lot in some use cases. For example, a binary verification pass is preferred to run as early as possible, so following passes will not waste time on bad libraries. It is also beneficial to place one pass behind another, if it can reuse the analysis result from the previous pass, avoiding repeated work.

Some special cases must be handled carefully. vDSO is one such tricky example. vDSO is a virtual dynamic library (*e.g.* `linux-vdso.so`) inserted into the application by the kernel, but still uses the standard dynamic loading mechanisms. Popular usage of vDSO is mapping some kernel regions into the application's address space, thus some system calls can directly execute on these regions. As a consequence, vDSO libraries must be loaded earlier than any pass that will issue vDSO related system calls. Otherwise, iFed pass itself will fault due to incomplete vDSO even before the application starts running. Similarly, if a pass relies on `malloc` from `libc`, it has to make sure that `malloc` is working properly ahead of the pass execution.

Pass confliction. With more passes integrated together, they are possible to introduce conflict transformations on libraries. Different passes may partition libraries into different isolation domains, or they have opposite optimization objectives. Currently, iFed relies on users to construct the transformation pipeline properly. Automatic dependency extraction and confliction detection will be supported in the future.

Figure 2 demonstrates a potential iFed transformation pass pipeline. All libraries are verified first using security signatures in the first verification pass. Then an isolation pass divides libraries into several isolation domains. Libraries in each domain are loaded into memory, where the memory management pass allocates and sets up memory permissions appropriately. The last binary rewriting pass completes symbol resolution, relocation, and other intent manipulations.

3.6 Dynamic Library Concatenation

Hugepages (superpages) can greatly reduce the address translation overhead, because it eliminates one level page table hierarchy and occupies fewer TLB entries. However, the current loader does not explicitly leverage hugepages. As shown in Figure 3 (a), the current loader individually maps every section in each library into the process's address space. If these sections use a small amount of memory (*i.e.* smaller than the size of a hugepage), the operating system is unlikely to allocate hugepages for them automatically. As a result,

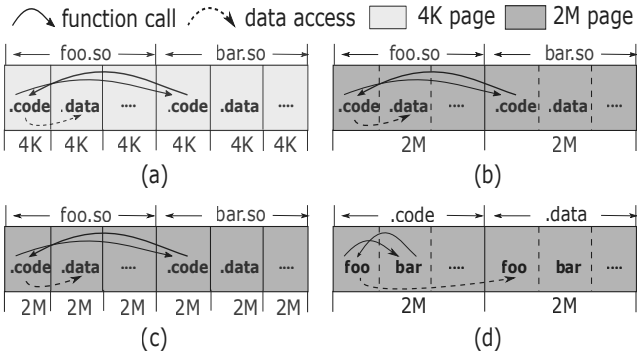

Figure 3: Different hugepage usage schemes for dynamic libraries.

frequent inter-library function calls will trigger more TLB misses, causing expensive page table walk, stalling the CPU instruction pipeline and slowing down applications.

The industry has two approaches to mitigate the impact of high TLB miss, but neither of them is ideal. Figure 3 (b) depicts the first approach, which allocates hugepages to hold all sections in the same library. While this approach reduces the number of used TLB entries, it brings many security vulnerabilities. Since all sections are in the same hugepage, that page should have all read/write/execute permissions required by different sections. For example, `.code` section becomes writeable and `.data` section is executable. Thus, this approach is only used in some closed environments. This, once again, indicates that the loader is capable to alter any policies designated during the development phase, making those policies unreliable. Therefore, the loader should provide capabilities to enforce security policies at load time.

The second method is illustrated in Figure 3 (c), such as the transparent hugepages for file systems proposed in the Linux kernel [27]. In this case, hugepages are used for large sections in each library. While it works well for applications using only a few large libraries, it cannot scale to a larger number of libraries. However, as we discussed in §1, using more and more libraries is the trend for production software, which leads to that such method will be less effective.

In iFed, we design a different approach and implement using in a iFed pass called dynamic library concatenation. The basic idea is intuitive as Figure 3 (d) shows. We collect the same sections, such `.code`, from all dynamic libraries and concatenate them one by one to form a big section. This combined section is large enough to fit in hugepages. More importantly, all the sections share the same memory permissions, so it is safe to place them in the same hugepage. Thanks to RiMF holding all libraries' information, the dynamic library concatenation pass is able to disassemble and rearrange libraries easily.

By combining all libraries `.code` sections into a big one, we might reduce the possible address range used by address space layout randomization (ASLR). To mitigate this security

concern, we have some options. (1) We can concatenate these libraries in random order. [2] (2) Hugepages do not have to be continuous in the virtual address space as long as the original section does not cross two hugepages. (3) We can leverage other code randomization techniques at load time [51] or run time [53], which is easier to employ in iFed.

Another potential negative impact introduced by dynamic library concatenation is library sharing. Dating back to the early days of computing, the motivation for using dynamic libraries is to save limited memory. When multiple running processes require the same library, they only share a single in-memory copy of the libraries. Library concatenation makes the sharing more difficult, as different processes have to use the same set of libraries. However, from our experience, this issue is acceptable for the following reasons. (1) The shared region is mainly the immutable code section. However, the code size of libraries is negligible compared to today's memory capacity. For instance, glibc has around 1.3 million lines of code and its un-stripped binary is only 17 MB, while a common server in the data center has 500 GB memory. (2) Thanks to the customizability of iFed, we can apply library concatenation only to key applications, while other utility or background processes still use the default memory management policy to share dynamic libraries. (3) In some cases, such as edge computing or micro-service, the same process will fork multiple times to serve different customers [36]. Since the forked process has the same address layout, they can share the concatenated library without any problem. (4) In the extreme case where the library must be shared, we align sections from different libraries at the 4K boundary. Thus, the 4K page in the middle of a hugepage can still be mapped to other applications at the cost that others are unable to utilize hugepages.

3.7 Relocation Branch Elimination

An important job accomplished by dynamic loaders is relocation, because the compiler cannot statically resolve cross-library function calls due to lack of address information. After the dynamic loader maps all libraries into process address space, it populates the actual address of unresolved functions in a lookup table. Then every call to a function in a dynamic library first retrieves the address from the lookup table and jumps to that destination. These extra actions result in a trampoline code, which is stored in another table.

Figure 4 (a) shows a simplified execution flow of relocation. The table used to serve address lookup is usually called global offset table (.got) and the procedure linkage table (.plt) saves the trampoline code. When functions in `foo.so` (*e.g.* `foo1` and `foo2`) call the function `bar` in `bar.so`, they call the trampoline (`bar@plt`) instead. The trampoline issues an indirect jump instruction, whose destination address is fetched

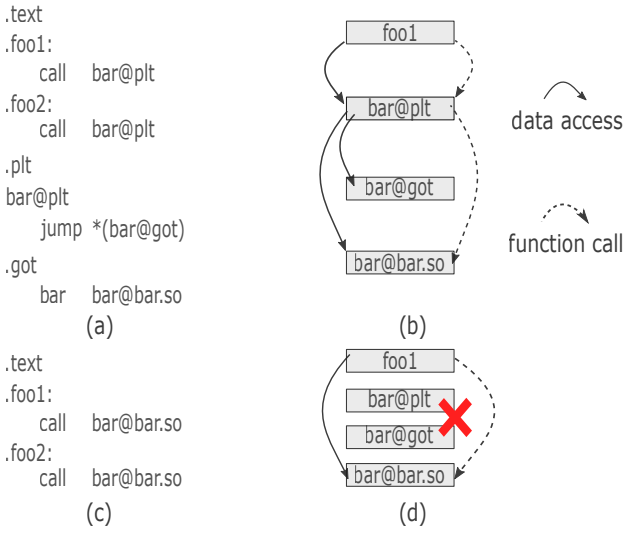

Figure 4: Function call relocation for dynamic libraries. Function `foo1` and `foo2` in `foo.so` call function `bar` in `bar.so`. In (a), function calls are first redirected to `.plt`, and consult `.got` entries to get the destination address, and finally branch to the destination. (b) depicts that the current relocation method incurs three memory access and two code branches. As shown in (c), the relocation branch elimination pass in iFed rewrites the function call sites so that they directly jump to the destination. As a result, only one memory access and code branch is needed in (d).

from an entry in `.got` (`bar@got`). Thus, the execution finally branches to the real address of `bar` (`bar@bar.so`). [3] The above relocation mechanism is applied to every function calls across dynamic libraries, thus incurring pervasive performance overhead. Figure 4 (b) depicts the performance cost in detail.

More executed instructions. Obviously, the single call instruction is expanded to multiple trampoline instructions, consuming more CPU cycles. Even worse, the additional indirect jump puts more challenge on the branch predictor. This is exacerbated by the fact that the trampoline code is not densely packed and `.plt` is often sparsely accessed, leading to more branch misses.

Extra memory access. The existing relocation approach also introduces more memory access. First, `.plt` asks for more memory to store the trampoline. Second, the trampoline needs to load from the extra `.got` memory. More memory access compete for the TLB and cache more frequently. Worse still, they are likely to be evicted from TLB and cache by other data access within the applications, especially in data-intensive scenarios, causing increased function call latency and unpredictability.

[2] Existing loader (*e.g.* `ld.so`) loads libraries in a deterministic way, which is decided by its internal library discovery algorithm according to the dependency information from application binaries.

[3] This simplified execution flow omits some complexities. .got entries are initially populated with a pointer to a loader's own resolver function. So when a library function is invoked at its first time, it branches to the resolver function, which then updates the `.got` entry using the actual address. This also requires additional instructions to be patched into the trampoline.

However, there are no practical solutions to eliminate these performance penalties. Switching to statically linked libraries is not always feasible as discussed in §1, and some hardware methods [1] are not available in production due to architectural modifications. It is also difficult to replace the relocation mechanism in the current dynamic loader with little effort.

Thanks to iFed, we have a chance to insert an optimization pass to reduce the relocation cost in an extensible manner. We design the relocation branch elimination pass for this purpose. The key idea inside relocation branch elimination is pretty intuitive. As shown in Figure 4 (c), we can directly rewrite the `call` instructions to replace their target address using the address of library functions, instead of the address of the trampoline in `.plt`. The performance gain is obvious. We eliminate the extra two memory access and one instruction branch as shown in Figure 4 (d). As a result, we essentially achieve the performance of static linking on top of dynamic libraries. Despite its simple idea, we have to deal with instruction decoding, relocation sites management, and other implementation issues carefully. Implementation details are discussed in §4.

Rewriting instructions causes it more difficult to share libraries among applications, since they have to be organized in the exact same address space layout. Thus, the relocation branch elimination pass is preferred to be used in environments with sufficient memory. Another challenge that needs to be overcome is the distance restriction of a relative branch. When using relative addressing mode, the CPU has restrictions on the distance between the call site and the target address. [4] Therefore, only rewriting the target address is not always possible if the library functions are loaded far from the call sites. This issue can be handled in multiple ways. (1) When combined with the dynamic library concatenation pass, it is rare that the distance exceeds the architectural constraint. (2) We can change the relative addressing to absolute addressing mode at the cost of an extra instruction to load the address into a register. This change can be done by recompiling the code or rewriting the instructions by the loader. For instance, the Linux kernel module loader rewrites the instructions when detecting the constraint violation. (3) For the call sites that are far away from the target function, we can fall back to the existing relocation method using `.plt` and `.got`.

3.8 Discussion and Summary

The pass-based architecture enables iFed to accommodate much more load time technologies and functionalities. However, we do not argue that our architecture is the only or best way to design a loader. Other methods are possible, such as "Linux kernel module" or "systemd service unit" approach. This is an open and new research area, and researchers are welcome to investigate more. iFed also brings side effects to program launch time and binary size, and we discuss these trade-offs below.

Loading Time. While iFed infrastructure itself does not introduce additional overhead to program launch, boot time will increase as more iFed transformation passes are enabled. End users have to make the judgment on the trade-off between longer loading time and securer or faster application in run time. According to our experience so far, the increased loading time in iFed is acceptable. This is because (1) For applications that already require a modified loader to provide new functionalities, they do not suffer more extra launch costs after switching to iFed; (2) For long-running services, such as web server and database, the one-time overhead during the startup is always negligible; and (3) For short-lived tasks in high churn environments, we can explore process template and in-memory caching technology [36] to fork processes from an initialized template, thus all forked processes will bypass iFed loading phase and its associated overhead. We study how our dynamic library concatenation and relocation branch elimination passes impact loading time in §5.

Binary Size. As some iFed transformation passes may need extra binary information to perform in-depth analysis, it is likely to bloat the application binaries. For example, the relocation branch elimination optimization requires the linker to retain all relocations in the executable file, resulting in larger binaries. While it is possible to scan the binary to re-generate these information, it is not wise to waste time on these redundant work. So far, the bloated binaries are not a big deal given the current massive persistent storage, but we argue the ELF-based binary scheme can be improved in the following senses. First, developers should keep relevant binary information (*e.g.* data generated by static analysis or bitcode of LLVM IR) as much as possible to reflect more comprehensive semantics close to the source code, instead of throwing them away at build-time and hiding them from the users. It is the user who makes the decision whether these information should be stripped at deploy- or install-time. Second, while iFed uses ELF-based objects for compatibility now, it is better to have a different object file format in iFed to match the pass-based structure and RiMF image. Particularly, object files can be disassembled into per-pass pieces, and these pieces can be fetched, trimmed, or analyzed through per-pass configuration. These improvements are left as future work.

Summary. We summarize how iFed resolves the issues discussed in §1 based on our design principles. Organizing iFed with a collection of transformation passes inherently achieves modularity (P1). Passes do not directly interact with each other, but rely on the pass manager to mediate and operate on RiMF image as the only interface for collaboration. New passes are easily plugged into iFed, which significantly improves extensibility in iFed (P1). Therefore, vendors do not need to randomly modify the loader nor maintain multiple

[4] This is because only a subset of bits in the branch instruction is available to encode the address. For example, x86 limits the range as $\pm$2 GB, while ARM has a limitation of $\pm$128 MB.

versions to satisfy customers' different demands, and in the meantime, customers are able to enjoy more features for free. Since users can choose which pass to be used in iFed via iFed configuration, they can flexibly construct transformation pipelines to customize the application at load time (P2). Paired with the iFed's capability to transform libraries with a global view, customers are flexible to determine the trade-off among security, isolation, and performance. iFed is implemented to be compatible with the current loader, so no application modification is required (P3). More importantly, iFed enables another level of transparency for system administrators. For example, managers can insert a default security enhancement pass to iFed, regardless of if applications are built with security options.

4 iFed Implementation

Figure 2 depicts the typical workflow of iFed during program launch. A program's binary is first loaded by the operating system, which then loads the dynamic loader's binary if necessary. Next, the kernel returns to user space and hands over the control to iFed. After discovering all required libraries, iFed invokes the pass manager with an initial RiMF image which simply contains all libraries in a single isolation domain. Based on the iFed configuration, the pass manager loads and executes each pass in sequence. Finally, iFed invokes the application's entry point and completes the loading phase.

Compatibility. Current iFed is implemented on top of `glibc` 2.28. We reuse some utility components, such as library discovery and ELF parser from the `glibc`. As a result, iFed is able to load unmodified ELF binaries and supports common loader extensions, such as `LD_PRELOAD`. For compatibility, the existing dynamic loader (*i.e.* `ld.so`) is organized as a special fake pass in iFed. Linux allows an application to specify the dynamic loader it will use. Thus, we use this facility to enable the usage of iFed within applications.

Dynamic Library Concatenation. In this pass, we collect the same sections from all libraries and pack them into continuous memory backed by hugepages. To save memory, the last page is converted to small pages if less than 64 KB memory is occupied. While the implementation is intuitive, we must fixup the global variable access. Global variables are always accessed via offset, which is the difference between the address of the accessing instruction and the variable itself. For example, in Figure 3 (a) and (d), the offset between the `.code` and `.data` section within the `foo.so` is changed due to the rearrangement. Thus accessing variables in the `.data` section is broken. Our current solution is to instruct the compiler to emit all the symbol access information (*e.g.* using `-emit-relocs` options in `gcc`), and to fix the offset during the pass execution. The book-keeping infomation inside iFed is also updated according to the finalized address, so as to serve run-time loader interfaces, such as `dlsym()` and `dladdr()`. We only rearrange the libraries which are position independent.

Relocation Branch Elimination. This pass rewrites the branch instructions so that they do not need indirect jump based on `.plt` and `got`. First, we identify all branch instructions from the relocation records. Each record saves the position of the instruction and the remote symbol it references. The symbol could be either a function or a variable. Then, we find the actual address of the symbol and modify the instruction to use the address instead. Modifying instructions is architecture-dependent. We further optimize the function pointer invocations. In case of the function address can be determined at the loading time, we substitute the function pointer with the actual function.

5 Evaluation

Our evaluation goals include:

- Illustrate the effectiveness of dynamic library concatenation and relocation branch elimination pass using micro-architecture statistics.
- Understand the applicability of iFed along with our optimization with a wide range of applications.
- Assess the generality when running iFed on different hardware architectures.

Setup. We evaluate iFed on two architectures. The first one is two 26-core sockets Intel(R) Xeon(R) CPU @ 2.3GHz, with hyper-threading enabled, resulting in 104 cores in total. The other is ARM Kunpeng-920 CPU @ 2.6GHz with four NUMA nodes, and each node has 24 cores. All experiments run on openEuler 20.03 [30] based on Linux 4.19 kernel. We compare iFed with the system default dynamic loader, `ld.so` in `glibc 2.28`.

5.1 Micro-benchmarks

We conduct a set of micro-benchmarks to evaluate the performance improvement of library concatenation and relocation branch elimination passes in iFed. Each test calls functions provided by a configurable number of dynamic libraries, and each function accesses a certain amount of memory. Figure 5 and Figure 6 study the impact of different library counts and working set sizes in the library function, respectively. All tests are run 500K iterations on the Intel machine. We compare four different implementations, (1) `glibc` – the system default dynamic loader. (2) iFed-`hugepage` – iFed with only dynamic library concatenation pass. (3) iFed-`relocation` – iFed with only relocation branch elimination pass. (4) iFed– iFed with both optimization passes.

Micro-architecture Impact. Figure 5 (a) shows the number of misses in instruction TLB. With more libraries involved, the total iTLB miss grows rapidly. `glibc` incurs the most iTLB miss because it uses 4K pages to load libraries and runs out of the limited number of iTLB entries. iFed-

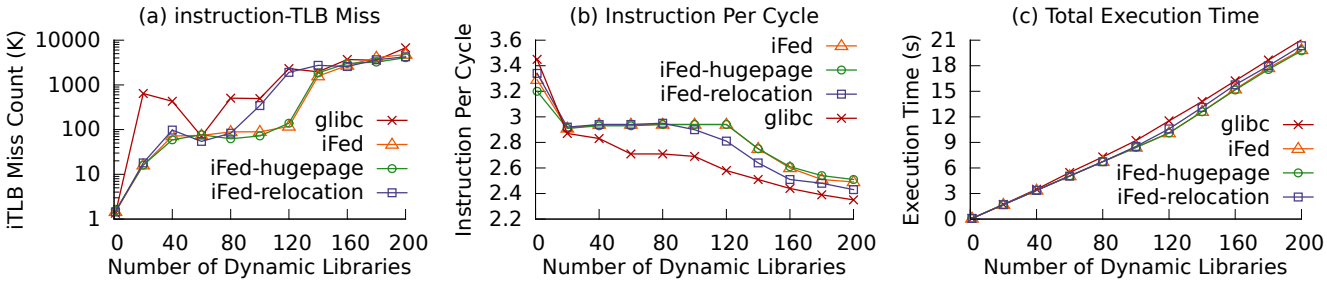

Figure 5: Micro-benchmarks: the working set is fixed at 256 KB.

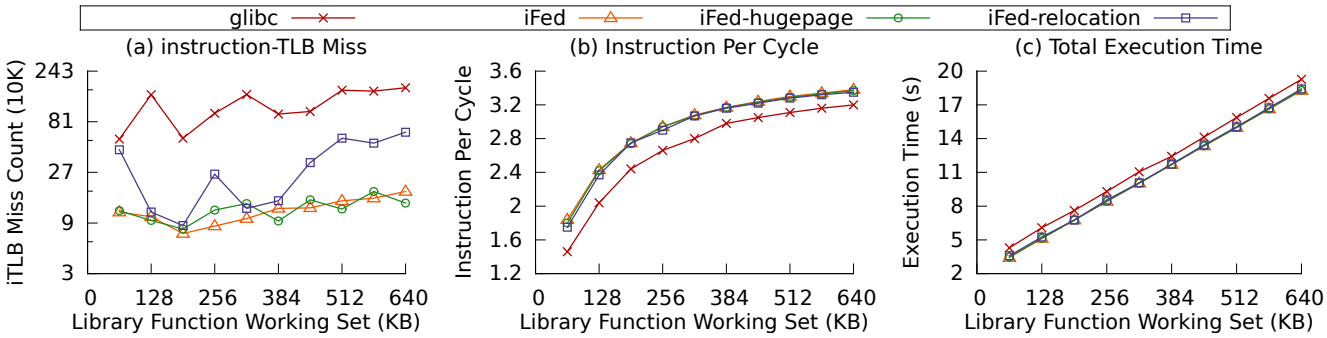

Figure 6: Micro-benchmarks: the number of dynamic library is 100.

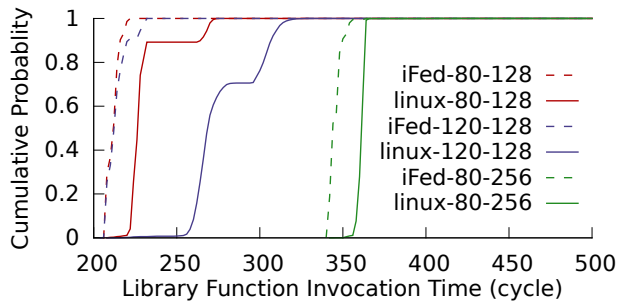

Figure 7: Library function invocation latency distribution.

`relocation` has little difference with `glibc` as it uses 4K pages, too. However, iFed-`hugepage` and iFed perform much better than `glibc` (note the log scale of y axis). Thanks to the usage of hugepage, they reduce the iTLB miss by an order of magnitude when the library count is smaller than 160, and by 40% with larger library counts. Less TLB miss leads to higher IPC as shown in Figure 5 (b). In addition to TLB miss, fewer code branches also decrease IPC. Thus, `glibc` has lower IPC than iFed-`relocation`, and iFed performs the best after integrating both optimizations. While the purposed optimizations almost work on `.code` sections, they also get benefits with a varied amount of data access as shown in Figure 6. With more data access, they compete for the cache and TLB when shared with `.code`, `.plt` and `.got` sections. This

`glibc`	iFed-`hugepage`	iFed-`relocation`	iFed
1.42 ms	5.96 ms	7.02 ms	9.07 ms

Table 2: Loading time overhead comparison. These are the cost to load a redis server which has 36195 relocation sites.

is illustrated in Figure 6 (a) where iFed-`relocation` triggers less TLB miss than `glibc`. In Figure 6 (b), IPC increases with the larger working set, as the memory access dominates the program execution. However, `glibc` performs worse than all iFed variants.

Latency Analysis. The improvements on micro-architecture further lead to the reduction in total execution time as depicted in Figure 5 (c) and Figure 6 (c). All execution time rise linearly with the test scale, but iFed runs faster than `glibc` in all cases. For instance, with 200 libraries and a 256 KB working set, iFed is 6% faster than `glibc`. More importantly, due to less TLB miss and branch, the predictability of library function invocation is improved a lot. To better understand the latency of library function calls, Figure 7 presents a CDF of function call latencies under different configurations. From the results, we observe that `glibc` has higher tail latency than iFed. For the 99th percentile latencies under the three configurations, iFed has improvements of 19%, 27%, and 3%, respectively.

Loading Time Discussion. Since iFed incorporates more functionalities, it inevitability slows down the time to launch a

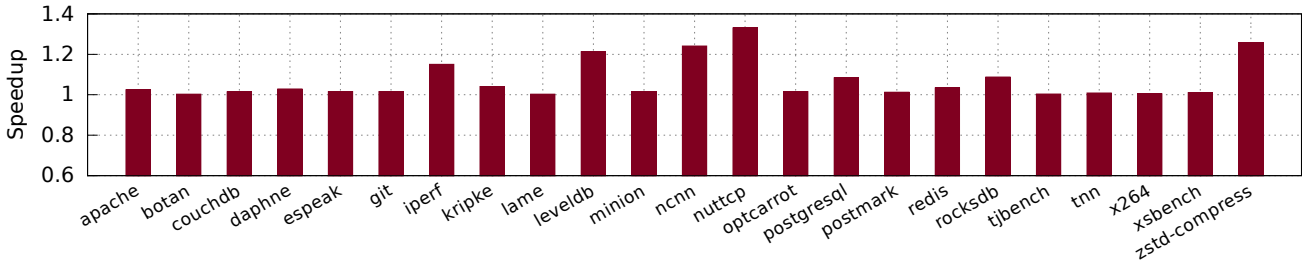

Figure 8: Phoronix test suite on ARM physical machine.

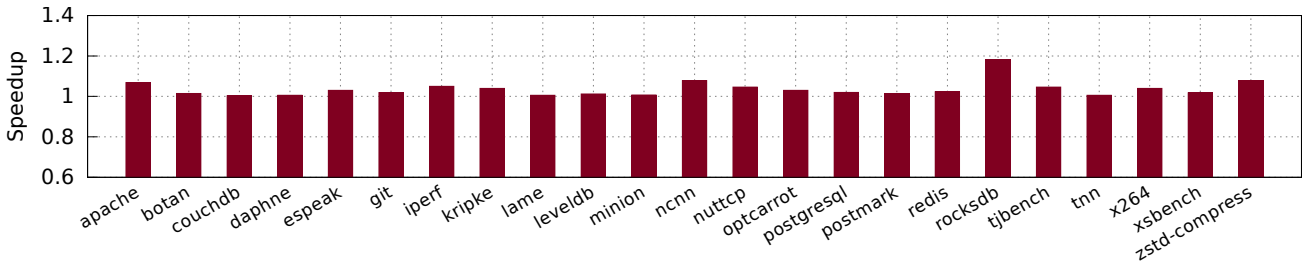

Figure 9: Phoronix test suite on x86 virtual machine.

program. Table 2 reports the loading time spent in the interval from the `exec` system call to the program's `main` function, when loading a redis server. As discussed in §4, the current iFed contains the `glibc` loader for compatibility, thus the result differences indicate the overhead of iFed optimization passes. Dynamic library concatenation overhead mainly comes from memory movement. The cost of relocation branch elimination depends on the number of relocation sites that has to be rewritten, thus it may incur a larger overhead.

5.2 Application Benchmarks

We evaluate iFed with Phoronix test suite v10.4.0 [42], which has a wide range of common applications and realistic workloads. We selected 23 applications from multiple domains that stress different components in the system, such as memory (`zstd-compress`), processor (`botan`), disk (`postmark`) and network (`iperf`). These tests also cover different running forms of multi-process, single- and multi-thread. We run these tests in two environments, the ARM Kunpeng server and a 60-core KVM-based virtual machine hosted in the Intel machine, because VMs are popularly deployed to hold applications today. Hardware virtualization is enabled, and the VM is configured with 128G memory. The guest OS is also openEuler 20.03 based on Linux 4.19 kernel. We enable all optimization passes in iFed, and report the average performance speedup compared to `glibc` in Figure 8 and Figure 9. The data is gathered from the built-in performance comparison tool in Phoronix. Since better hugepage usage and eliminated `.got/.plt` indirection in iFed will improve many tightly correlated micro architecture factors, we use `perf` to measure some typical CPU events for each benchmark. Table 3 lists the percentage of TLB miss reduction, branch miss reduction, and IPC improvement compared to `glibc` on both ARM and Intel testbeds.

Whether an application can get benefits from iFed depends on its bottleneck. For computing intensive applications who do not suffer from TLB miss or branch mispredictions, iFed keeps the same performance with `glibc`. For example, `botan` is a C++ crypto library and the benchmark measures the performance of many cryptographic algorithms. iFed has less than 1.5% performance difference with `glibc` in all test cases. As shown in Table 3, iFed has a negligible impact on IPC. `xsbench` tests a key computational kernel of the Monte Carlo neutronics application OpenMC. iFed does not reduce branch misses on ARM, thus the performance difference between iFed and `glibc` is less than 2%.

When the application is memory bound and its data compete for the shared TLB and cache with the code, iFed is able to mitigate the interference and improve the performance. For instance, the `zstd-compress` benchmark compresses and decompresses a 1 GB Linux kernel image. iFed reduces the number of TLB misses by 16.56% and 19.4% on Intel and ARM machine, respectively. Please note that our dynamic library concatenation deals with both `.code` and `.data` section, thus iFed does not only reduce iTLB misses. As a result, iFed speedups the benchmark by 7.3% on Intel and 25.7% on ARM.

For complicated applications that have complex function call patterns across libraries or use many dynamic libraries, iFed can boost their performance. For example, `leveldb` from Google gets 1.1% and 21.2% better performance on Intel and ARM platform, respectively. On ARM, `glibc` in-

curs 10^9 instruction TLB misses, while iFed just incurs 10^5 iTLB misses! `ncnn` is a mobile neural network inference framework developed by Tencent. Its IPC is improved by 3.04% and 6.36% on Intel and ARM platform respectively, and correspondingly iFed gets 7.7% and 24% overall better performance. `iperf` and `nuttcp` have a large performance boost because both benchmark server and client are loaded by iFed.

In some cases, iFed shows large relative improvements on perf events while has little impacts on the benchmark performance. That is because the event's absolute numbers are so small that slight variations of the event result in large percentage difference. For example, on the Intel machine, `optcarrot` shows 10.45% less TLB misses while its performance is only 2.9% better. After examining the absolute number of TLB misses, we found there are only 1.8 million misses with `glibc` and iFed lowers it to 1.6 million. Those numbers are several orders of magnitude smaller than those in other memory intensive benchmarks. Another example is branch miss reduction in `botan` benchmark on the ARM platform. `botan` experiences around 323K and 311K branch misses under `glibc` and iFed respectively. Despite 3.65% reduction in branch misses, iFed does not have speedup over `glibc`.

To further validate our results, we analyse the `rocksdb` benchmark on the Intel VM in depth. The benchmark contains 3857 `.got` entries and 8153 `.plt` entries, and 94327 relocation sites point to these entries. With `glibc`, 15.6% of total cycles are spent on page table walk due to TLB misses, while this ratio is reduced to 10% after iFed optimization. Relocation branch elimination pass contributes 6% improvement, and dynamic library concatenation pass continues to improve 10%, leading to an overall improvement of 18%. We also tested a statically linked version which performs 9% better than the dynamic one with `glibc`. This improvement is less than iFed with the hugepage optimization, but is better than iFed with relocation elimination since static linking has more chance to apply link-time optimization.

In general, we do not observe the loading time overhead causing performance degradation even for the benchmarks which need to frequently boot up and shut down the test programs. On the Intel virtual machine, compared to `glibc`, the average TLB miss is reduced by 8.58%, the average branch miss is reduced by 3.28%, and the average IPC is improved by 3.02%. iFed is 3.7% better than `glibc` on average and achieves 18% maximum improvement. On the ARM physical

benchmark name	x86			ARM		
	tlb miss	branch miss	instruction per cycle	tlb miss	branch miss	instruction per cycle
apache	12.27%	8%	4.98%	5.44%	1.82%	0%
botan	8.01%	3.13%	0.09%	0.05%	3.65%	0%
couchdb	3.86%	0.79%	4.64%	4.94%	0%	0%
daphne	8.25%	5.56%	2.22%	2.15%	4.25%	4.47%
espeak	12.6%	1.33%	0.26%	32.04%	0.07%	0.37%
git	3.85%	6.71%	2.54%	1.36%	0.46%	3.09%
iperf	7.58%	5.03%	5.73%	27.95%	3.91%	19.44%
kripke	4.56%	10%	4.31%	17.79%	1.12%	5.41%
lame	7.4%	11.52%	1.17%	18.1%	0.7%	1.19%
leveldb	3.15%	1.37%	4%	34%	5.29%	32.43%
minion	13.66%	0.71%	1.63%	1.01%	0.37%	1.54%
ncnn	7.98%	5.03%	3.04%	37.05%	2.87%	6.36%
nuttcp	3.12%	3.03%	5.92%	34.55%	6.95%	56.52%
optcarrot	10.45%	1.39%	3.85%	0.24%	1.65%	1.49%
postgresql	5.93%	2.21%	1.56%	10.33%	2.4%	4.05%
postmark	3.3%	0.7%	-0.47%	9.97%	0.63%	0.85%
redis	6.4%	1.01%	1.54%	12.78%	2.12%	2.23%
rocksdb	35.9%	4%	13%	13.52%	2.71%	8.16%
tjbench	14.82%	-0.34%	3.59%	2.83%	0.1%	0.61%
tnn	1.4%	1.09%	0.57%	2.69%	1.11%	1.26%
x264	1.99%	0.64%	1.28%	1.88%	0.51%	0.62%
xsbench	4.21%	1.27%	1.78%	9.96%	-0.58%	0%
zstd-compress	16.56%	1.32%	2.25%	19.4%	0.48%	18.52%
average	8.58%	3.28%	3.02%	13.04%	1.85%	7.33%

Table 3: Application benchmarks: percentage of TLB miss reduction, branch miss reduction, and IPC improvement..

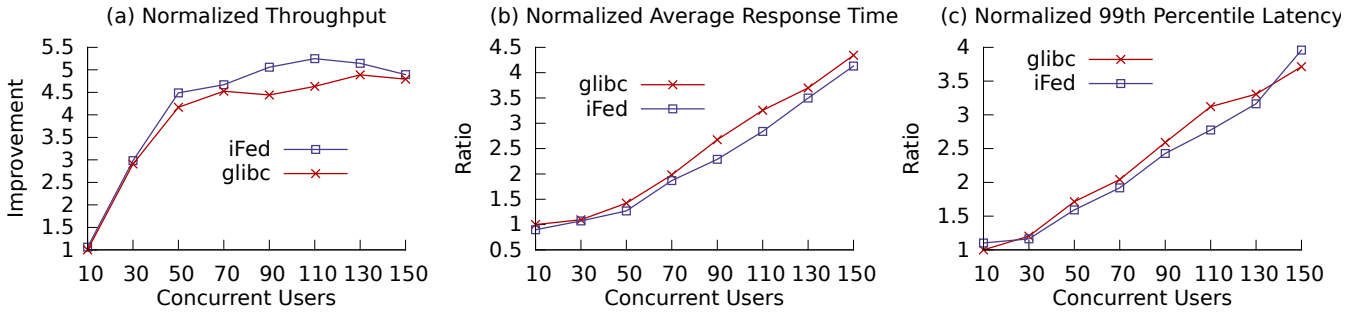

Figure 10: Dynamic web serving performance. All data are normalized to the result of 10 concurrent users with `glibc`. (a) shows the throughput across all operations, the higher the better; (b) shows the average response time of `postwire` operation (similar to posting a tweet), the lower the better; (c) shows the 99th percentile latency of `postwire` operation, the lower the better.

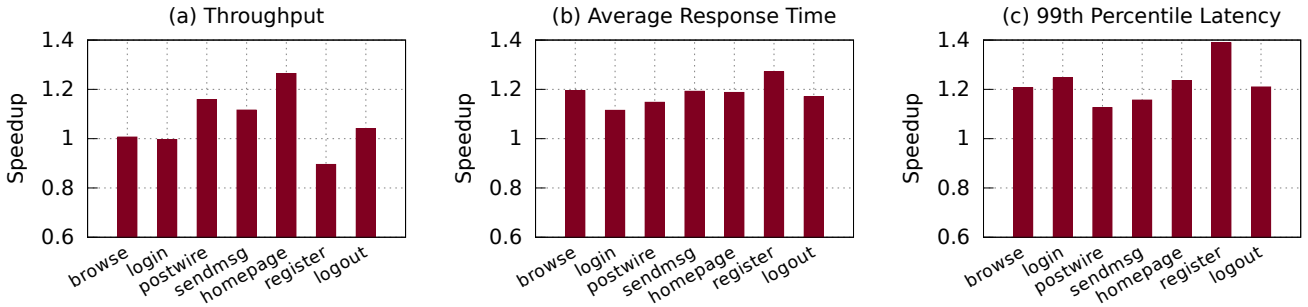

Figure 11: Performance of each operation with concurrent 110 users. All data are the speedup ratio normalized to the `glibc`, the higher the better.

machine, iFed reduces 13.04% TLB miss, lowers the branch miss by 1.85%, and improves the IPC by 7.33%, on average. The average speedup is 7% and the largest improvement is 33%. In most cases, iFed achieves a larger improvement on the physical machine than the virtual machine. This is because VMs have an extra address translation layer. Thus if the host OS allocates small pages to the guest OS, iFed will get less benefit by enabling hugepage in the guest OS.

5.3 Web Serving

Finally, we evaluate iFed in a system-wide scenario with a web serving benchmark from Clousuite [10]. This benchmark is a dynamic website hosting a production-quality social networking engine. Since the current Clousuite is not supported on ARM architecture, we port it to our ARM machine, and upgrade its components to newer versions. Particularly, we use nginx 1.16.1, mysql 8.0.17, PHP 7.2.10, and elgg 3.0.7. We run the client and server on two ARM machines under the same ToR switch. The client simulates multiple users who browse the website and issue different operations, such as register, login, and send messages to a friend. These operations are mixed in a distribution that favors common operations (*e.g.* send a message and post a tweet), while containing fewer login/logout operations. Each test case runs 5 minutes, and Clousuite collects the throughput and response time.

Figure 10 shows the normalized performance with various simulated concurrent users. The efficiency is seen in the improved throughput, reduced response time, and tail latency. The performance keeps increasing with more users until the system is saturated. For the peak performance, iFed has 13.3% higher throughput, 14.7% smaller average response, time and 12.5% lower 99th percentile latency. Figure 11 shows the detailed performance statistics of each operation with 110 concurrent users. iFed is better than `glibc` in most cases. For the throughput of `register` operation, iFed is lower because the client issues less `register` operation due to the probabilistic workload distribution. This is also confirmed by the reduced response time from Figure 11 (b) and (c). To summarize, these results demonstrate that optimizations in iFed are effective in the realistic multi-application environment.

6 Related Work

Loader modification and improvement. Many projects have to modify the loader to achieve their specific goals, even though the loader is not of their research contribu-

tion. However, the current dynamic loading infrastructure is neither flexible nor extensible to accommodate those modifications, causing their research to have limited applications in the industry. When utilizing MPK [14, 38, 50] or SGX [7,13,32,37,40,44,60] for stronger isolation or security, most works have to modify the loader to be aware of such isolation facility. Besides hardware-assistant isolation, software implementation also require to coordinate with loader, such as sandbox [5,6,24,55] and CFI [22,25,47,58]. It is necessary to change the loader to support program migration and execution on remote, heterogeneous, or smart devices [3, 8, 52]. Kard [2] leverages MPK for per-thread memory protection to implement a dynamic data race detector, which uses a custom loader to handle global variables. Shuffler [53] continuously re-randomizes code locations in a separate thread, but requires a small loader patch for bootstrap. With iFed, these modifications will be made easily and further reused across different projects.

Agrawal et al. propose a speculative hardware mechanism to avoid executing relocation trampolines [1], while we provide a pure software approach to eliminate relocation overhead in iFed. Stephen Kell et al. describe the formal semantics for static linking [19]. As iFed decouples a monolithic dynamic loader into smaller pieces, we expect a similar formal method can be applied to dynamic linking as well.

Load time technologies. There is a large body of research focusing on load time technology. Paschalis Mpeis et al. introduce a capture and replay mechanism [28] that detect and profile hot code regions, and optimize them offline. Instead of the original code from binary, these captured and optimized hot regions are fed into the loader to replay. Egalito [54] is a binary transformation framework that supports dynamic analyses or code-generation at load time. Load time binary stirring [51] randomly reorders some code sections and repairs code pointers accordingly. ASLR-Guard [26] contains a dynamic loader, which decouples code sections from data sections and encrypts some sensitive regions. Library debloating [33,35] is a type of load time optimization that loads only the set of library functions that will be used at each library call site within the application at runtime. iFed provides a platform to explore and integrate broader load time technologies. Wei Dong et al. propose a holistic dynamic linking and loading mechanism in networked embedded systems to generate minimal code size [9].

Loader on new system and architecture. Since dynamic loader is a basic toolkit, it has to be rewritten whenever a new system or hardware comes. For example, RedLeaf [29] is a rust-base OS with a new abstraction, called Domains, for lightweight isolation, and supports dynamically loaded Domains. CARAT [41] allows programs to run efficiently in a physical address space and needs a loader to collaborate properly. Different loaders are also implemented within different system architectures, such as microkernel [20,49], unikernel [43] or LibOS [4,34,45,59]. Similarly, the loader is always needed to be updated to explore new hardware features for isolation [39], security [31], communication [48], container [57], and embedded device [21]. With the help of iFed's modular design, we are able to extract the system agnostic or architecture independent parts and reduce the porting effort.

7 Conclusions

We introduce iFed, an infrastructure for dynamic library transformation. While iFed is compatible with the current dynamic loader, its function goes beyond the traditional dynamic linking and loading. By a pass-based architecture and RiMF, iFed can provide much richer functionalities over isolation, security, and optimizations in a flexible, extensible, and modular way. We demonstrate the extensibility of iFed by implementing two optimization passes. One pass reduces TLB miss and improves IPC because of the effective usage of hugepages. The other pass rewrites the call sites to eliminate function relocation overhead. Modularity and extensibility are crucial to reducing the development, deployment, and maintenance costs of today's complicated system software. We believe it is an open research area to investigate modular design in many other monolithic system software, not just in the loader.

Our evaluation shows optimizations in iFed improve performance and predictability for a wide range of applications on multiple architectures and platforms. On an ARM physical machine, iFed achieves up to 33% speedup, and on an Intel virtual machine, iFed gets a maximum improvement of 18%. In a complex dynamic website that requires collaboration among multiple applications, iFed improves the throughput by 13.3% and achieves a 12.5% reduction of end-to-end 99th percentile latency. More importantly, iFed boosts the performance transparently with no application changes. Building on both customers' demands from industry and load time technology advances from academia, the dynamic library manipulation infrastructure is a promising area of research. We believe that iFed paves the first example of a new generation of dynamic loaders for integrating research advancement of load-time transformations and technologies.

Acknowledgments

We sincerely thank our shepherd Andreas Haeberlen for his insightful feedback. We are grateful to the OSDI anonymous reviewers for their valuable comments and suggestions. We thank the EulerOS team at Huawei for their contributions to this work, including but not limited to, Zixian Liu, Bin Wang, Sirui Liu, Pan Zhang, Lin Fu, Xiangyang Yu, Yanchao Yang, Chao Liu, Danni Xia, Jiaqi Yang, Yining Shen, Tianxiong Lu, Haomin Cai, Wei Du, and Guiping Zhang.

References

[1] Varun Agrawal, Abhiroop Dabral, Tapti Palit, Yongming Shen, and Michael Ferdman. Architectural support for dynamic linking. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*, 2015.

[2] Adil Ahmad, Sangho Lee, Pedro Fonseca, and Byoungyoung Lee. Kard: Lightweight data race detection with per-thread memory protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, 2021.

[3] 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 Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, 2017.

[4] Andrew Baumann, Dongyoon Lee, Pedro Fonseca, Lisa Glendenning, Jacob R. Lorch, Barry Bond, Reuben Olinsky, and Galen C. Hunt. Composing os extensions safely and efficiently with bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)*, 2013.

[5] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *10th USENIX Symposium on Operating Systems Design and Implementatio (OSDI'12)*, 2012.

[6] Mihai Bucicoiu, Lucas Davi, Razvan Deaconescu, and Ahmad-Reza Sadeghi. Xios: Extended application sandboxing on ios. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (Asia CCS'15)*, 2015.

[7] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (ATC'17)*, 2017.

[8] Shenghsun Cho, Han Chen, Sergey Madaminov, Michael Ferdman, and Peter Milder. Flick: Fast and lightweight isa-crossing call for heterogeneous-isa environments. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA'20)*, 2020.

[9] Wei Dong, Chun Chen, Xue Liu, Jiajun Bu, and Yunhao Liu. Dynamic linking and loading in networked embedded systems. In *2009 IEEE 6th International Conference on Mobile Adhoc and Sensor Systems*, pages 554–562, 2009.

[10] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*, 2012.

[11] Michael Franz. Dynamic linking of software components. *Computer*, 30(3):74–81, March 1997.

[12] Free Software Foundation (FSF). Gnu lesser general public license, `https://www.gnu.org/licenses/lgpl-3.0.html`.

[13] Lukas Giner, Andreas Kogler, Claudio Canella, Michael Schwarz, and Daniel Gruss. Repurposing segmentation as a practical lvi-null mitigation in sgx. In *31st USENIX Security Symposium (USENIX Security'22))*, 2022.

[14] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 USENIX Annual Technical Conference (ATC'19)*, 2019.

[15] W. Wilson Ho and Ronald A. Olsson. An approach to genuine dynamic linking. *Software-Pratice and Experience*, 21(4):375–390, April 1991.

[16] Intel. Intel(R) Software Guard Extensions, `https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html`.

[17] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual.*

[18] Seunghoon Jeong, Jaejoon Hwang, Hyukjin Kwon, and Dongkyoo Shin. A cfi countermeasure against got overwrite attacks. *IEEE Access*, 8:36267–36280, 2020.

[19] Stephen Kell, Dominic P. Mulligan, and Peter Sewell. The missing link: Explaining elf static linking, semantically. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*, 2016.

[20] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood.

Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*, 2009.

[21] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys'14)*, 2014.

[22] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*, 2014.

[23] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1999.

[24] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. Minibox: A two-way sandbox for x86 native code. In *2014 USENIX Annual Technical Conference (ATC'14)*, 2014.

[25] Yan Lin, Xiaoyang Cheng, and Debin Gao. Control-flow carrying code. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications (AsiaCCS'19,)*, 2019.

[26] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. Aslr-guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, 2015.

[27] LWN.net. Transparent huge pages for filesystems, `https://lwn.net/Articles/789159/`.

[28] Paschalis Mpeis, Pavlos Petoumenos, Kim Hazelwood, and Hugh Leather. Developer and user-transparent compiler optimization for interactive applications. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'21)*, 2021.

[29] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. Redleaf: Isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, 2020.

[30] openEuler. `https://openeuler.org`.

[31] Meni Orenbach, Andrew Baumann, and Mark Silberstein. Autarky: Closing controlled channels with self-paging enclaves. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)*, 2020.

[32] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Safebricks: Shielding network functions in the cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, 2018.

[33] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. Blankit library debloating: Getting what you want instead of cutting what you don't. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)*, 2020.

[34] 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 Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*, 2011.

[35] Anh Quach, Aravind Prakash, and Lok Yan. Debloating software through piece-wise compilation and loading. In *27th USENIX Security Symposium (USENIX Security'18))*, 2018.

[36] Yuxin Ren, Guyue Liu, Vlad Nitu, Wenyuan Shao, Riley Kennedy, Gabriel Parmer, Timothy Wood, and Alain Tchana. Fine-grained isolation for scalable, dynamic, multi-tenant edge clouds. In *2020 USENIX Annual Technical Conference (ATC'20)*, 2020.

[37] Vasily A. Sartakov, Daniel O'Keeffe, David Eyers, Lluís Vilanova, and Peter Pietzuch. Spons & shields: Practical isolation for trusted execution. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'21)*, 2021.

[38] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. Cubicleos: A library os with software componentisation for practical isolation. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, 2021.

[39] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys – efficient in-process isolation for risc-v and x86. In *29th USENIX Security Symposium (USENIX Security'20)*, 2020.

[40] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and efficient multitasking inside a single

enclave of intel sgx. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, 2020.

[41] Brian Suchy, Simone Campanoni, Nikos Hardavellas, and Peter Dinda. Carat: A case for virtual memory through compiler- and runtime-based address translation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)*, page 329–345, 2020.

[42] Phoronix Test Suite. `https://www.phoronix-test-suite.com/`.

[43] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-unikernel isolation with intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'20)*, 2020.

[44] Jörg Thalheim, Harshavardhan Unnibhavi, Christian Priebe, Pramod Bhatotia, and Peter R. Pietzuch. rkt-io: a direct I/O stack for shielded execution. In *Sixteenth European Conference on Computer Systems (EuroSys'21)*, 2021.

[45] 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 Ninth European Conference on Computer Systems (EuroSys'14)*, 2014.

[46] 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 Eleventh European Conference on Computer Systems (EuroSys'16)*, 2016.

[47] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive cfi. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, 2015.

[48] Lluís Vilanova, Marc Jordà, Nacho Navarro, Yoav Etsion, and Mateo Valero. Direct inter-process communication (dipc): Repurposing the codoms architecture to accelerate ipc. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys'17)*, 2017.

[49] Qi Wang, Yuxin Ren, Matt Scaperoth, and Gabriel Parmer. SPeCK: a kernel for scalable predictability. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'15)*, 2015.

[50] Xiaoguang Wang, SengMing Yeoh, Pierre Olivier, and Binoy Ravindran. Secure and efficient in-process monitor (and library) protection with intel mpk. In *Proceedings of the 13th European Workshop on Systems Security (EuroSec'20)*, 2020.

[51] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*, 2012.

[52] Yaron Weinsberg, Danny Dolev, Tal Anker, Muli Ben-Yehuda, and Pete Wyckoff. Tapping into the fountain of cpus: on operating system support for programmable devices. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'08*, 2008.

[53] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and deployable continuous code re-randomization. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, 2016.

[54] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, 2020.

[55] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *30th IEEE Symposium on Security and Privacy (S&P'09)*, 2009.

[56] Hansen Zhang, Soumyadeep Ghosh, Jordan Fix, Sotiris Apostolakis, Stephen R. Beard, Nayana P. Nagendra, Taewook Oh, and David I. August. Architectural support for containment-based security. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, 2019.

[57] Hansen Zhang, Soumyadeep Ghosh, Jordan Fix, Sotiris Apostolakis, Stephen R. Beard, Nayana P. Nagendra, Taewook Oh, and David I. August. Architectural support for containment-based security. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, 2019.

[58] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *22nd USENIX Security Symposium (USENIX Security'13)*, 2013.

[59] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. Kylinx: A dynamic library operating system for simplified and efficient cloud virtualization. In *USENIX Annual Technical Conference (ATC'18)*, 2018.

[60] Wenjia Zhao, Kangjie Lu, Yong Qi, and Saiyu Qi. Mptee: Bringing flexible and efficient memory protection to intel sgx. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)*, 2020.