ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

LUCI: Loader-based Dynamic Software Updates for Off-the-shelf Shared Objects

Bernhard Heinloth, Peter Wägemann, and Wolfgang Schröder-Preikschat
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Abstract

Shared libraries indisputably facilitate software development but also significantly increase the attack surface, and when using multiple libraries, frequent patches for vulnerabilities are to be expected. However, such a bugfix commonly requires restarting all services depending on the compromised library, which causes downtimes and unavailability of services. This can be prevented by dynamic software updating, but existing approaches are often costly and incur additional maintenance due to necessary source or infrastructure modifications.

With LUCI, we present a lightweight linker/loader technique to unobtrusively and automatically update shared libraries during runtime by exploiting the indirection mechanisms of position-independent code, hence avoiding severe runtime overhead. LUCI further adds no additional requirements, such as adjusting the source or interfering with the build chain, as it fully adapts to today's build and package-update mechanisms of common Linux distributions. We demonstrate our approach on popular libraries (like *Expat* and *libxcrypt*) using off-the-shelf (i.e., unmodified) binaries from Debian and Ubuntu packages, being able to update the majority of releases without the necessity of a process restart.

1 Introduction

Third-party libraries are, without doubt, an important part of software development, allowing a programmer to use functionality beyond their domain (or at least to save some time). To prevent outdated source-code copies [49], it is common to dynamically link against libraries distributed in binary format [50] – on system-level so-called shared objects, enabling independent updates. A popular example is *OpenSSL*, providing cryptographic capabilities to numerous projects. However, its infamous *Heartbleed*-vulnerability demonstrated the downside: All of a sudden, millions of systems (yet alone 24 – 55% of all HTTPS-enabled servers [13]) ran the risk of leaking highly sensitive information when the buffer over-read bug was discovered. Even though distributors quickly published fixed versions of the library, all applications directly or indirectly depending on this library had to be restarted, not only requiring immediate manual action but also causing costly downtimes of services [12,22].

Generally, having a new library version to be responsible for the need for a service restart is all but seldom: Default server software in many popular Linux distributions like Debian is dynamically linked and depends on several shared libraries, which are usually the main reason for a service being exposed to known vulnerabilities:

For example, the currently most widely used web server [23], *nginx*, had 11 (out of 34 CVEs[1]) vulnerabilities with high severity[2] since 2010, while its basic (static) dependencies *glibc*, *OpenSSL*, *PCRE*, *libxcrypt*, and *zlib* had at the same time 70 such critical vulnerabilities (of 335 CVEs total).

It is worth noting that these numbers do not take any modules into account: the default configuration of *nginx* includes six core modules, which depend on a total of over 30 external shared libraries themselves – and have at least 98 additional critical vulnerabilities (of 474 CVEs) in the observed period.

A similar picture emerges for the second place: 14 critical vulnerabilities were found in *Apache HTTP Server* including its core modules. In contrast, its required shared libraries in basic configuration (without modules) had 81 such issues due to an additional dependency on *Expat* compared to *nginx*.

Especially for stateful software systems (e.g., database management systems) or systems with active client connections, restarting the service in case of a vulnerability is undesired [36,37]. To address the challenges of avoiding unwanted downtimes and expensive startup costs, dynamic software update (DSU) mechanisms have been developed over the last decades [8,11,15,28,29,42].

Although many interesting DSU techniques are available, and even Linux introduced kernel live-patching in version 4.0 more than seven years ago [20], the requirement for restarting services utilizing updated libraries has not changed since

[1]Common Vulnerabilities and Exposures: www.cve.org.
[2]CVSS (Common Vulnerability Scoring System) score with 7.0 or higher.

then. Unfortunately, common user-space live-patching has not become a reality yet.

This raises the question regarding the reasons for this shortcoming. Most approaches for DSU require modifications in the source [15, 17, 28, 29] or build system [11, 18, 42, 47], programmer-guided patch creation [34, 43, 45], and/or a virtual machine [5, 7, 35] for execution. While the modification of a single project can be rather easy, it would have to be applied to almost every software package in a system for practical usage of dynamic updates – and yet the vast amount of different software prevents any broad application.

That is why we believe that system-wide automatic live updates in user space will only succeed if the requirements are limited to the bare minimum: We argue that *unmodified (i.e., off-the-shelf) binary files* already built and deployed by distributors have to be sufficient input for base and updated versions.

In addition, imposed runtime overhead and stability concerns due to the complexity of existing approaches further hinder any attempts to establish general live-patching in user space. Nevertheless, these penalties are inevitable fallout of the effort to provide a general approach allowing to transform almost any program state – which is usually not required for security fixes: Since the vast majority of system-level software is still written in programming languages not ensuring memory safety (like C and C++) [6], patches for critical vulnerabilities commonly introduce small local code changes like bounds checking (e.g., the fix for the mentioned *Heartbleed* bug). Such patches usually do not alter any function output for valid inputs [24].

The situation for logic errors is similar: Most common bugfix patterns affect only the function scope [32] without side effects beyond its borders. Especially system-level shared objects have to maintain the *application binary interface* (ABI) to which other software binaries are dynamically linked, making structural changes rather seldom. New feature improvements altering the library's *application programming interface* (API) can usually only be used in depending software after modifying its source code as well. Since adjusting projects to the libraries' updated semantics may take time [21], distributors tend to backport bugfix patches while retaining API & ABI compatibility [39].

These insights help us to define the necessary scope required for dynamic software updates based on binary files under practical conditions: First, we focus on supporting the mentioned changes required for error correction instead of enabling updates to introduce arbitrary modifications. Second, we argue that a practical DSU solution has to update off-the-shelf binaries without requiring access to or modifications of the source code or build process.

With Luci, we present a DSU approach for unmodified shared libraries utilizing features already enabled by default in today's build chains without inducing runtime overhead. In this sense, the paper makes the following contributions:

- A lightweight loader-based DSU mechanism targeting security fixes that is based on relinking dynamic ELF binaries by leveraging its metadata
- Design and implementation of an open-source, dynamic linker/loader with *glibc* compatibility for the x86_64 architecture supporting automatic and transparent updates of off-the-shelf shared libraries
- Evaluation of popular, binary-distributed shared libraries of recent Debian and Ubuntu releases to assess the live-patching approach's practicality

The remainder of the paper is organized as follows: After giving an overview of DSU techniques in Section 2, Section 3 describes the details of ELF binaries that we utilize for our approach in Section 4. To verify the results, we back-test Luci with popular shared libraries in Section 5 while classifying the results in Section 6. Section 7 concludes the paper.

2 Related Work

Research of dynamic software updates in user-space dates back four decades, with *DYMOS* [11] presenting the first notable approach beyond manually live-patching machine code. The approach allows updates of programs written in a Modula dialect while using a custom compiler and runtime system. Thereby, this approach involves restrictions with respect to the source-code development and its build system, which correspond to 🅐 and 🅒 in Figure 1. Many sophisticated approaches that evolved since then share these restrictions:

Ginseng [28, 29] allows changes in function prototypes and data representation but is limited to source code written in C while also requiring code adjustments 🅐. Patches are generated by comparing the source files 🅑 in conjunction with analysis information emitted by a custom compiler 🅒 and loaded with a custom runtime system. The approach involves significant performance overhead of up to 30% in updated functions. Although being a powerful approach, *Ginseng* illustrates the massive adjustments required to support generic dynamic updates. Luci avoids these requirements as it hot-swaps code while not supporting data modification.

With *Kitsune* [15] (successor of *Ekiden* [17]), developers have to manually specify update points, add control-flow and data-migration code to their C source 🅐, and use a custom compiler 🅒. After waiting for all threads to reach the update point, it replaces the whole program and performs all migrations. The approach causes a service disruption on update time but reduces overhead costs during normal execution.

POLUS [8] constructs a patch for a successive version using a source-to-source compiler 🅑. This patch can be applied at any time by employing `ptrace`, with old and new versions residing in memory. *POLUS* places redirection instructions

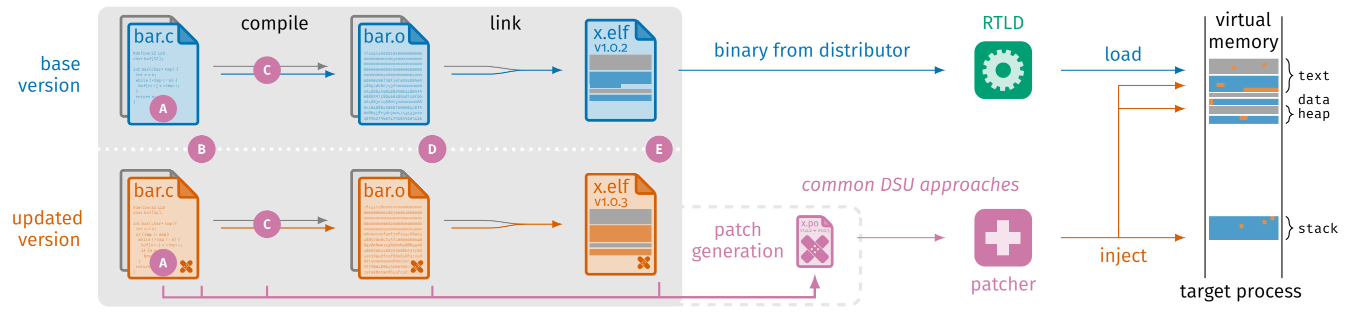


Figure 1: Common DSU approaches require either certain (higher-level) programming languages [11], modification of the code [29], or the changes in the build toolchain [42] (unless the patch is handcrafted from the resulting binaries [5]), as highlighted by the gray area on the left side. Due to the sheer amount of different software projects, it is not feasible to adjust each of them – hence preventing live-patching in user-space for today's real-world software. A generic automatic update mechanism must not interfere with these steps but merely use the resulting binaries, usually built by distributors.

into the old functions and keeps track of global-state modifications. Luci adapts the idea of having multiple versions of a binary in the virtual memory and extends it by shared data.

In contrast to previous approaches, *Katana* [42] is language-agnostic and works on object-file level Ⓓ. While improving applicability, it still requires interfering with the build chain.

For Luci, the feature of coping with off-the-shelf shared libraries is essential – a feature also provided by a few other approaches: Using dynamic binary translation, *DynSec* [34] can patch code of unmodified binaries while having a significant runtime overhead (11% in benchmarks) and requiring programmer-guided patch generation Ⓔ. *Piston* [43] is trying to exploit vulnerabilities in order to fix them. It works on a binary level and can automatically generate repair routines for stack-based buffer overflows, but such routines must be manually provided Ⓔ for other vulnerabilities. With the process virtual machine *DynamoRIO* [5], binary-level code modification Ⓔ is possible at the cost of runtime overhead. Based on this tool, *ClearView* [35] is able to learn normal application behavior and automatically generate patches for certain types of bugs, however, causing massive overheads (depending on the configuration 47% – 303% baseline overhead, and up to 30 000% while learning).

Different methods to update active functions have been proposed: *UpStare* [26] Ⓒ and *StrongUpdate* [51] Ⓑ using stack reconstruction and *ISLUS* [9] Ⓑ with checkpoint-based rollback, all require C source Ⓐ. For Luci, this is not needed since we expect shared library functions to return eventually.

While live updates for Linux [7, 27] and other operating systems [1, 3] were already an important research topic, *kSplice* [2], *kPatch* [38], and *kGraft* [33] ⒸⒹ initiated kernel live-patching in Linux [20]. The latter two can only perform changes on functions, not on data, similar to Luci. In contrast, *kSplice* is not only able to support data changes but can also be used for certain specially prepared user-space libraries [31].

A few other approaches focus on live-patching of shared libraries as well: *LibCare* [47] generates patches from assembly emitted by a compiler wrapper script Ⓒ and applies them using `ptrace`: After acquiring storage for changed code (and new data, if required), relocations in the existing code are updated while using stack unwinding to prevent modifications of currently executed functions.

A rather less-intrusive approach is used by *libpulp* [45], requiring specially prepared/compiled libraries with an additional `nop`-prologue at each function Ⓒ to be able to dynamically insert trampoline code. A manually created description file Ⓔ guides the updater (using `ptrace` in conjunction with a preloaded library) through the symbols to be replaced.

The *libDSU* [30] concept also targets unmodified shared libraries. However, *libDSU* would require an actual implementation of the approach to find and update all locations of pointers in the process' memory – including heap chunks, which is quite complex and error-prone to identify. Luci avoids this requirement by keeping memory locations valid.

All existing approaches either require modifications during building, for both base and updated version, or programmer-guided patch generation. To the best of our knowledge, no approach yet exists that can provide an update on shared libraries without interfering in the build process (gray area in Figure 1). With Luci, we close this gap.

3 Background

On Unix-based systems, a machine-code–generating compiler produces relocatable objects from the source code, nowadays in the *executable and linkable format* (ELF). With ELF being the fulcrum for executables, Luci exploits this format.

The ELF meta information lists available and required symbols (e.g., functions and static variables) accompanied by relocation information, enabling the linker to fix destination-address parameters of memory-access and branching instructions in the machine code. For static (position-dependent) executables, the target addresses of all symbols can and must

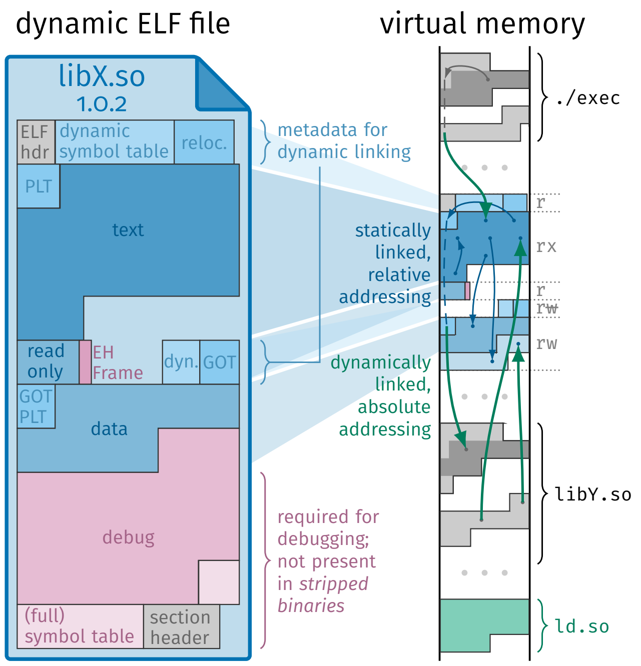


Figure 2: Shared library loaded and relocated in process' virtual memory by the dynamic linker/loader (`ld.so`).

be resolved during static linking, hence, the *executable ELF* contains a complete image of the process's virtual memory. However, for shared objects and dynamic executables, some symbols may remain undefined in the resulting *dynamic ELF* binary. Therefore, the `dynamic` section references their meta information, so symbols can be resolved during execution by the dynamic linker/loader (also referred to as *runtime link-editor*, RTLD) – allowing the use of other shared libraries.

To simplify the execution of such binaries, the compiler (e.g., with `-fPIC` for position-independent code) references local symbols using relative addressing, while emitting instructions and function stubs (like the *procedure linkage table*, PLT) for indirect addressing to access external symbols. The PLT itself is closely tied to a *global offset table* (GOT) section introduced by the static linker, which stores the actual target addresses during runtime. Therefore, the dynamic linker/loader is not required to modify the machine code in the `text` section itself but only the GOT and, if required, data sections. This strategy improves load performance and security since no executable pages are mapped with write permission in the process' virtual memory, while also maintaining a low memory footprint. Additional techniques like *relocation read-only* (`data.rel.ro`) further contribute to security by removing the write permission after the initial linking steps where applicable (e.g., constants referencing external symbols).

The dynamic linker/loader is responsible for finding all required shared libraries of an application on the file system. The tool places libraries in the process' virtual memory, resolves and fixes undefined symbols while retaining a defined search order (to deterministically handle symbols having the same name). Eventually, the dynamic linker/loader performs initialization and passes control to the application's entry point. For lazy binding (i.e., resolving undefined function symbols on the first call), the dynamic linker/loader has to reside in memory during the entire lifetime of the process. The same holds for the dynamic metadata of each shared library (for symbol lookup and relocation). Figure 2 visualizes a mapping of an ELF file into the virtual memory of a process.

Since a POSIX-conform dynamic linker/loader also provides an interface allowing the process to load additional shared objects at runtime, it is itself an executable, self-contained shared object (`ld.so`) with a tight connection to the C standard library (*libc*) used on the target system. In fact, the dynamic linker/loader is usually an integral part of the standard library (e.g., *glibc*, *musl libc*).

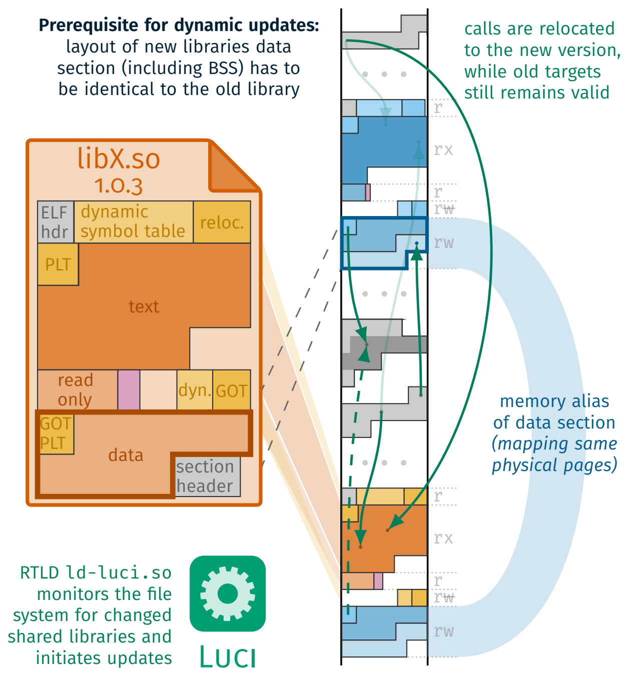
4 Approach

When a modification of a shared object used in a process is detected, we first compare the old and new version. A crucial requirement for our approach is having identical writeable sections: The same symbols have to be stored at the same position having the same size and the same initial content, for both initialized and uninitialized data (`data` and `bss` section). Only then the new version can safely be loaded and linked into the process. We found in practical applications (see Section 5) that this requirement is not a major restriction as modifications on writable sections are rare, as detailed in the following.

While analyzing typical bugfixes for common weaknesses in dominating system programming languages[3], we noticed that newly introduced static variables are rare. Further, modern compilers and linkers work in a deterministic manner (including reordering of variable allocations in an optimization step) and most distributors have optimized their build chains for reproducible builds[4]. Therefore, the probability is high that code changes do not affect the data section of the binary. Consequently, our DSU mechanism checks for alterations in sections containing writeable data, including the *thread-local storage* (TLS), which would prevent an update. While modifications of the writable sections are rare, changes to the read-only data section are more likely, for example, due to introduced strings for error messages. However, these `rodata` updates do not hinder the update process but are inherently supported by our approach, as well as newly introduced automatic variables, which are stored in the stack memory.

[3]The common weakness enumeration (CWE) list at cwe.mitre.org maintains a good overview, including views explicitly focusing on C/C++.

[4]For reproducible builds any indeterminism in the build process is removed, allowing to reproduce a binary-identical file on every build with same source code revision as input. Further information at reproducible-builds.org.



Prerequisite for dynamic updates: layout of new libraries data section (including BSS) has to be identical to the old library
libX.so 1.0.3
ELF hdr
dynamic symbol table
reloc.
PLT
text
read only
dyn.
GOT
GOT PLT
data
section header
RTLD ld-luci.so monitors the file system for changed shared libraries and initiates updates
Luci
calls are relocated to the new version, while old targets still remains valid
r
rx
r
rw
rw
memory alias of data section (mapping same physical pages)
r
rx
r
rw
rw

Figure 3: Luci's core principle is to map the updated shared library to unused space of the processes' virtual memory. Thereby, it exploits *memory aliasing* in order to map the newly introduced data section to the physical memory of the previously active data section.

Segment Layout Requirements In both the old and new version, our approach additionally requires the `data` section to have identical page alignment. However, due to the page-level granularity of permissions in memory-management units, it is already standard in linker scripts to place the writable segment at a page border. The GOT dedicated to the PLT for lazy runtime linking is put at the beginning, followed by the `data` section. The writeable segment is usually preceded by the segment containing the relocation-read-only section, which also includes the standard GOT (for external variables) and is only writeable during the initial dynamic linking stage – but neither its alignment nor its content affects the updateability.

After the updater has ensured the availability of all currently required symbols in the new shared object, it loads all non-writeable data sections into the process' virtual memory, placing it at some previously unused address range. Instead of loading the data section from the ELF file, a *memory alias* of the old version's data section is created (see Figure 3): Both old and new versions' data sections use the same page frames, allowing changes to a variable in the old data section to be immediately visible in the new one and vice versa. Since Linux currently lacks a direct way to create such an alias, we have to use an anonymous in-memory file created by the `memfd_create` system call for the data.

Relinking using GOT Then, all executables and other shared objects utilizing the changed library are updated: The dynamic software updater relocates the affected entries in every GOT to the corresponding symbols in the new library. In addition, it is possible that data sections need relocation as well, for example, when function pointers are used. In this case, the Luci updater must first ensure that the target memory still contains its original value and is compatible before replacing it with the new value.

Since function calls are performed indirectly (using single instruction reads of the GOT PLT) and both old and new functions coexist in the process, there is no need to alter the `text` section. Hence, the update can be initiated at any time without requiring quiescence [8], reaching a certain update point in code, or modifying the process stack. Furthermore, there are no limitations regarding updates of multithreading applications as there are basically no other runtime modifications than the default RTLD performs when lazy binding. Having the code of an old library function be executed at the time of update is not a problem; it continues accordingly until completed (`return` instruction) with the old code, but future calls will be redirected to the new library function, thus leading to a gradual update. Since shared libraries should provide a collection of subroutines and, therefore, frequently return to the caller, proper software engineering prohibits remaining in an endless loop inside the old version.

In order to apply changes, the dynamic software updater needs access to the process' virtual memory. Since the dynamic linker/loader resides in the process, it is a comfortable target to house the update mechanism, not only avoiding the need for additional permissions but also providing easy access to a list of all loaded objects and their relocation information.

RTLD with DSU Capabilities In order to assess the applicability of the approach to real-world libraries, we have implemented our own dynamic linker/loader Luci for the x86_64 architecture with the ability to perform the described update procedure while maintaining binary compatibility (to some extent) to the *glibc* equivalent `ld-linux.so`, allowing loading and live-patching of unmodified binaries from a distributor: Luci can run common executables when passed as a parameter or transparently if Luci is set as the interpreter in the corresponding ELF section of the executable. This fully circumvents `ld-linux.so` but still supports most *glibc* libraries (including `libc.so` and `libpthread.so`) while providing compatible interfaces for RTLD-specific functionality (e.g., `libdl.so` and tunables).

If dynamic updates are enabled (e.g., by the corresponding environment variable), Luci creates an observer thread on start and uses the `inotify` API to detect modifications of all loaded shared objects (or their symbolic links, respectively). In case of a `fork`, Luci intercepts it in order to decouple the data memory alias and create a new observer thread in the child process. Besides the `fork` and thread creation, our live-updating mechanism introduces no runtime overhead.

On detected changes, the observer thread autonomously compares the update compatibility of both versions and, on success, initiates the update process. In addition, LUCI can output status information, for example, notifying about required restarts due to incompatible changes.

A successful update currently does not necessarily prevent the execution of old versions: If the library dynamically stores a pointer to local functions during runtime (e.g., in heap-allocated memory), these could result in reusing old code after an update. LUCI cannot statically identify (and fix) them since there is no relocation information. Nevertheless, it provides an optional method to detect such access at runtime: After a user-defined time following an update, LUCI hides all pages containing executable sections in old libraries and installs a user-space page-fault handler (using `userfaultfd`). If the non-present page is accessed, LUCI makes it available again while using the status output to inform about the failed update and the requirement for a manual restart.

The virtual memory layout of the updated shared object is as intended by the linker and described inside the ELF file, hence not limiting debugging capabilities with standard tools like *GDB*. Furthermore, C++ exception handling (and unwinding in general) works as expected, even in updated libraries, since LUCI provides a version-agnostic interface for dynamic linker introspection: Therefore, requests from exception/unwinding routines will be passed to the exception handling frame (`eh_frame`) of the corresponding library.

Checking Basic Compatibility To safely create a memory alias of the old writeable data section in an updated shared library, the section must match in alignment, layout, and content. The meta information of the ELF file (`data` section address and size, its initial values plus the dynamic symbol table) can be sufficient – unless the writeable data has local relocations: For each static variable pointing to a local symbol, LUCI ensures the target's equivalency in the old and new version. Otherwise, an update could lead to undesirable behavior, for example, because of changed variable semantics.

The dynamic update is straightforward as long as the target is located in the `data` section, because LUCI simply compares the contents and follows the relocations. However, when considering the executable section, we cannot just compare the machine code since changed offsets (e.g., due to added code) likely result in a different byte stream. With the help of the capstone engine [41], LUCI disassembles the code and (using relocation information, instruction-pointer–relative addressing, and branch instructions) creates a dependency graph. This graph is similar to a call graph but also contains references in the data section. To enable fast comparisons, LUCI calculates a fingerprint for each function: Similar to techniques used in malware analysis [10], LUCI creates a hash based on the machine instructions while excluding relocated immediate operands and `%rip`-displacements – they are replaced instead with corresponding symbolic equivalents. This allows LUCI to find identical functions independently from their location.

For dynamic updates, LUCI considers a target symbol to be compatible if the fingerprint (for symbols in the executable section) or contents (for data) in old and new versions match, as do the targets of all references.

After ensuring the update compatibility in principle, LUCI further checks if the process has not altered the memory targeted by the relocation entry. Consequently, LUCI keeps track of previous values used for fixing relocations and aborts the update process on detected changes while notifying the user about the requirement for a manual restart.

Improving Compatibility Detection So far, stripped binaries – containing only essential parts – are sufficient. However, access to binaries' full symbol table and debug information can contribute to detecting whether an updated version can be applied: While its full symbol table contains storage information for local variables, the *DWARF* debug sections allow an even deeper inspection. Not only type information for all variables can be gathered here, but also the fields of records (`struct`) and enumerated values can be compared.

For debug purposes, many distributors like Debian and Ubuntu offer additional debug information for the binaries, due to size considerations usually distributed in separate packages. With `libdebuginfod`, there is even a web service for easy retrieval of debug symbols, using the unique *BuildID* of the binary located in the `note` section.

Although modification of records (e.g., `union`s) and additional `enum` values do usually not interfere with a successful update, they are still quite rare in non-feature-updates. Thus, we prefer a pessimistic approach for the sake of stability: If either variable location or type, any internal record field, or enumerated value has changed, the update is not applied. This limitation has the ability to simplify the version comparison drastically: By suitably hashing the information about the internal structure, the hash value is sufficient to determine compatibility. We have implemented a service (similar to `libdebuginfod`) that LUCI queries. An alternative to LUCI's approach is to include the value directly in the ELF file, for example, in an additional `note` section inserted by the post-processing steps of the packaging toolchain.

Dynamically Loaded Libraries The POSIX function `dlopen` enables one to load libraries during runtime. By default, both functions and global symbols are retrieved as pointers using `dlsym`. To effectively support dynamic updates for such runtime-loaded files, LUCI creates an indirection for function types: Similar to the PLT, the address of a helper function is returned, which redirects the call to the latest version of the symbol – introducing the overhead of an additional `jmp` instruction. Using this trampoline technique, LUCI can even update dynamically loaded libraries.

5 Evaluation

We successfully validate the previously described functionality of our implementation with a custom test suite written in C/C++ consisting of small examples (targeting especially the corner cases) using different compilers and versions (*GCC* v6 – v12, *LLVM/Clang* v11 – v15) on several distributions (including RHEL/AlmaLinux, Fedora, and openSUSE Leap). Other tests demonstrate the ability to update code changes in libraries written in Ada (*GNAT*), Fortran (*GNU Fortran*), Go (`c-shared` using the *GNU Go Compiler*), Rust (*Rust compiler* with `prefer-dynamic` flag), and, with some limitations, Pascal (*Free Pascal Compiler*).

However, to demonstrate the practicability of our approach, as well as its limitations, this evaluation focuses on popular shared libraries without any custom modifications. To provide a realistic scenario [44], we do not perform updates of individual patches/commits but on the level of full official release versions, which usually contain multiple changes.

For each library, we independently build (without using any artifacts of a previous compilation) each version of a reasonable range having a compatible API. As far as possible, we use the suggested toolchains and default configuration for each library according to the corresponding documentation. Neither changes to the source nor custom tools are used during the build process. In order to evaluate their impact on LUCI's compatibility detection, we manually enable debug symbols.

Then, a program – preferably a test suite with high code coverage – linked against this library interface is executed, while a supervisor script subsequently, with a certain delay, exchanges the library version on the file system in the background. Moreover, this supervisor listens to the status interface of LUCI in order to be notified about incompatible or failed updates (the communication is strictly uni-directional) – which will cause a restart of the test program, enabling it to use the latest library version in the traditional way.

In addition to self-compiled vanilla versions, we also test the corresponding binary releases in popular distributions the same way. We choose Debian and Ubuntu as they are considered to be the most widely used Linux distributions (at least for web servers [48]). Additionally, we are able to retrieve outdated packages from previous releases[5]. We focus on their two most recent versions: *Focal Fossa (20.04)* and *Jammy Jellyfish (22.04)* in the case of Ubuntu (LTS) and *Buster (10)* and *Bullseye (11)* for Debian. However, as debug symbols for some versions are missing in the archives, LUCI solely relies on the (stripped) ELF files when evaluating external builds.

It is worth noting that Debian carefully tries to prevent any breaking changes in their stable package releases, often backporting security fixes to the library version used initially in a particular Debian version. Therefore, its library versions may differ from custom builds having the same version number.

Suitable libraries must meet the following criteria:

[5]Using launchpad.net for Ubuntu and snapshot.debian.org for Debian.

- Enough *recent development* to compare different version releases – especially different binary releases in the mentioned distribution versions.
- *Independent libraries* providing distinct functionality rather than just an interface to a service. To simplify testing, it should not be tightly entangled with system components.
- Availability of a *test tool or suite* with reasonable coverage of the library interface and its code. It must not use internals beyond the public/official interface since this might prevent it from running with other releases. To demonstrate the dynamic update, we further need a long-running process (ideally executing the tests in an endless loop) – scripts executing individual tests in subprocesses are unsuitable.
- *High popularity* in both local installations and software depending on it. The Debian popularity contest [40] tracks the installations of their users and can act as an indicator.

Taking those requirements into account, we have selected the libraries *Expat*, *libxcrypt*, *OpenSSL*, and *zlib* for evaluation, all of them within the top 150 packages (out of 70 000) in the Debian popularity contest. This also covers main dependencies of the *nginx* and *Apache HTTP server*.

Environment

We perform all tasks in container environments with a minimal base system installed, running on an x86_64 architecture (Intel Core i5-8400 with four cores and 16 GiB of RAM).

For all tests, LUCI is configured to automatically detect changes in library files or their symbolic links on the file system, check compatibility, and, if applicable, update libraries during runtime. A few seconds after applying an update, the executable section of older library versions is unmapped. Any subsequent access to the old library would trigger the user-space page-fault handler, which marks the update as failed and results in a restart by the supervisor script several seconds later. The delay before a restart allows us to ensure that the program correctly continues even after a failed update, not producing unexpected results or aborting. Incorrect results or abnormal program terminations, regardless of whether the library release or the update causes them, are explicitly noted.

5.1 Expat

Since the *Expat XML parser* is used in numerous applications [46] and hence part of all popular *Linux* distributions, its dozen critical vulnerabilities discovered within the last decade make it a good target and test candidate for LUCI. We focus on major version 2, having 27 version releases since 2006 – with 29 CVEs (including 11 critical vulnerabilities).

	Expat	2.0.0	2.0.1	2.1.0	2.1.1	2.2.0	2.2.1	2.2.2	2.2.3	2.2.4	2.2.5	2.2.6	2.2.7	2.2.8	2.2.9	2.2.10	2.3.0	2.4.0	2.4.1	2.4.2	2.4.3	2.4.4	2.4.5	2.4.6	2.4.7	2.4.8	2.4.9	2.5.0
ELF segment	`init`	–					❍						❍															
	`code`	–	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	`rodata`	–	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	`relro`	–	●	●	●	●	●		●	●	●	●	●	●		●	●	●			●	●	●	●	●		●	●
	`data`	–		●		●	●		●		❍		●					●										
	`bss`	–		●			●		●				●					●										
	symbol tables	–		●		●	●		●		●		●					●										
DWARF	writeable vars	–		●			●		●				●					●										
	internal types	–	●	●		●	●				●						●	●										
	external API	–		●		●	●				●		●				●	●										
	dynamic update	–	✖	✖	✔	✖	✖	✔	✖	✔	✖	✔	✖	✔	✔	✔	✖	✖	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
			restart																									
LUCI results	*start*	*1*	*2*	*3*		*4*	*5*		*6*		*7*		*8*				*9*	*10*										
	# test cases	326	326	329	333	333	340	340	341	341	341	341	341	341	341	341	341	342	342	342	342	342	342	342	342	342	342	343
	# failed (max)	14	14	14	13	13	13	13	13	13	12	11	11	8	8	8	7	7	7	7	7	6	5	4	3	3	2	0
	$\overline{\text{time}}$ (ms)	390	338	371	532	498	578	577	576	577	532	518	535	532	533	532	530	531	530	531	578	578	521	522	660	659	542	522
	$\overline{\text{time}}$ SD (ms)	7	6	6	3	3	4	3	0	3	4	3	4	3	5	3	9	3	1	4	3	9	3	17	3	10	3	4
baseline	*start*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*	*25*	*26*	*27*
	# test cases	326	326	329	333	333	340	340	341	341	341	341	341	341	341	341	341	342	342	342	342	342	342	342	342	342	342	343
	# failed (max)	14	14	14	13	13	13	13	13	13	12	11	11	8	8	8	7	7	7	7	7	6	5	4	3	3	2	0
	$\overline{\text{time}}$ (ms)	382	331	362	523	489	569	572	566	568	523	507	528	526	523	525	519	523	521	522	568	568	513	510	648	648	534	513
	$\overline{\text{time}}$ SD (ms)	5	4	1	6	6	14	1	5	13	4	2	17	3	6	13	5	7	6	13	6	6	7	2	7	7	15	6

Table 1: Successively (every 25s) replacing the *Expat* library on the filesystem with all vanilla version 2 releases in chronological order while running the test suite (in an endless loop). The upper part of the table shows LUCI's internal analysis of each changed library binary compared to the previously loaded version. ● denotes detected changes of symbols in the corresponding segment, while ❍ marks identical segments having symbols with modifications in their dependencies (at other segments). When non-updatable changes (highlighted with red color) are detected, the library version is incompatible (✖) and the test suite gets restarted (marked by a vertical bar: |). Compared to the baseline (using the default RTLD, shown on the bottom), LUCI can prevent two thirds of restarts while providing the same results (as shown in the middle row).

The developers maintain a good and regularly updated test suite in a single program, which we have to slightly modify: Tests causing segmentation faults and double-frees in older releases are dynamically omitted in those vulnerable versions. Further, due to a slight API change (new symbols in 2.1.0 and 2.4.0), the corresponding tests are only enabled if those symbols are available in the currently active library release (using weak linkage). The LUCI-loaded program now executes all eligible tests sequentially in an endless loop while measuring the duration and the number of executed and failed tests[6].

Vanilla The results in Table 1 show that LUCI is able to perform 17 dynamic updates (67%) during runtime – with 11 subsequent updates (starting with version 2.4.0) without requiring any restart. During those patches, we observe a steady decrease in failed test cases due to the bugs fixed in newer releases, identically to manually executing the test suite with the corresponding library version using the default RTLD. The average duration of a test iteration in LUCI is slightly worse (about 2%), but this is caused by our RTLD implementation itself since it is not as optimized as the *glibc* counterpart and needs to use some workarounds/indirections for compatibility with standard `libc.so.6`: The timings with LUCI are consistent regardless of whether the DSU functionality is enabled or disabled. Furthermore, the raw data does not show any notable increase while updating a library to a new version.

Most failed updates have obvious reasons, like changes in the writeable data section, which LUCI automatically detects in both binary and DWARF debug information. However, LUCI rejects the updates to 2.0.1 and 2.3.0 only due to debug information. A detailed look reveals that in both cases only a single enumeration value was added. This causes a different hash of the datatype (and, in the latter case, the API as well) even though an update would be possible – which we validate in additional tests. However, the currently strict setting safely allows certain update directions: It is not only possible to skip several releases (e.g., 2.4.4 → 2.4.9), but rolling back to an earlier release is also possible – as long as the hashes are identical and the requirements of the binary are met.

[6]A use-after-free bug (CVE-2022-40674) causes jitter in the results for versions prior to 2.4.9, as the corresponding test only sometimes fails.

		Binary releases in Debian Buster																		
		development												*stable*						
	Expat	2.2.0-2	2.2.1-1	2.2.1-2	2.2.1-3	2.2.2-1	2.2.2-2	2.2.3-1	2.2.3-2	2.2.5-1	2.2.5-2	2.2.5-3	2.2.6-1	2.2.6-2	+deb10u1	+deb10u2	+deb10u3	+deb10u4	+deb10u5	+deb10u6
ELF segment	`note`	–	●	●	●	●		●	●	●	●		●	●	●	●	●	●	●	●
	`init`	–	🔴						🔴				❍							
	`code`	–	●			●		●	●	●			●	●	●	●	●	●	●	●
	`rodata`	–	●			●		●	●	●			●	●	●	●	●	●	●	●
	`relro`	–	●			●		●	●	●			●	●	●	●	●	●	●	●
	`data`	–	🔴					🔴		🔴										
	`bss`	–	🔴					🔴												
	`dynsym`	–								🔴										
	updatable	–	✗	✓	✓	✓	–	✗	✗	✗	✓	–	✗	✓	✓	✓	✓	✓	✓	✓
LUCI	# tests	333	340	340	340	340	340	341	341	341	341	341	341	341	341	341	341	341	341	341
	# failed	13	13	13	13	13	13	13	13	12	12	12	11	11	11	10	9	7	6	5

		Debian Bullseye									
		development				*stable*					
	Expat	2.2.7-1	2.2.7-2	2.2.9-1	2.2.10-1	2.2.10-2	+deb11u1	+deb11u2	+deb11u3	+deb11u4	+deb11u5
ELF segment	`note`	●	●	●	●	●	●	●	●	●	●
	`init`										
	`code`	●	●	●	●	●	●		●	●	●
	`rodata`	●	●	●	●	●	●		●	●	●
	`relro`	●	●	●	●	●	●		●	●	●
	`data`										
	`bss`										
	`dynsym`	🔴									
	updatable	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
LUCI	# tests	341	341	341	341	341	341	341	341	341	341
	# failed	11	10	8	8	8	7	5	4	3	3

Table 2: *Expat* test suite running with binaries retrieved from the official Debian repository – including development builds.

Backtesting Additionally, we back-test the approach using prebuilt packages from distributors. For Debian, we can retrieve the binaries from the snapshot archive, which is not limited to updates for stable releases but includes all development builds as well. The implications become apparent when considering the Debian workflow: Whereas during the development phase latest library versions are maintained – having the same update incompatibilities as our vanilla builds above – the versions are frozen after the Debian release gets stable. However, that does not mean a standstill at all: The Debian team puts effort into backporting fixes for issues, like the decreasing number of failed test cases in Table 2 shows: While *Debian Buster* uses *Expat* 2.2.6 in its stable release 2019, their latest package `2.2.6-2+deb10u6` fixes several vulnerabilities found in 2022. Since they usually do not include feature changes but only minor changes in the code section, the stable phase is an ideal situation for LUCI: all package releases are eligible for dynamic updates. When also considering development builds, LUCI can prevent 72% of restarts in Buster and 90% in Bullseye.

Although the stable phase used in productive environments is our main focus for LUCI, the development builds offer interesting insights about the applicability of our approach: Several different versions of the build utils were used during that time; however, this does not necessarily cause incompatibilities. For example, package 2.2.9-1 was built with *GCC* 9.2, while 2.2.10-1 used *GCC* 10.2, but a dynamic update is still possible. It can also happen that a newer release build results in identical code and data segments, which may or may not produce a different *BuildID* stored in the `note` section. For example, the unequal *BuildID* between 2.2.5-1 and 2.2.5-2 requires LUCI to analyze the file, while the subsequent update to 2.2.5-3 is binary identical, allowing LUCI to safely skip any further processing quite early.

Ubuntu is similar regarding the version freeze after stable release, as shown in Table 3: Even with development libraries, 83% of restarts can be omitted with LUCI in *Jammy*. Library updates published for the stable release can all be dynamically applied; the same is true for all library releases in *Focal*. In both versions, there is only one update each, simply differing in the *BuildID* compared to its predecessor; all other updates have actual code changes.

These results highlight the effectiveness of LUCI in a real-world scenario, as it can update the majority of all *Expat* versions, even off-the-shelf libraries built with different compilers and without access to debug information.

Build		***Expat***	**updatable**	
custom (vanilla)		2.0.0 – 2.5.0	17 / 26	(65%)
Debian Buster	*all*	2.2.0 – 2.2.6	13 / 18	(72%)
	stable	2.2.6	6 / 6	(100%)
Debian Bullseye	*all*	2.2.7 – 2.2.10	9 / 10	(90%)
	stable	2.2.10	5 / 5	(100%)
Ubuntu Focal	*all*	2.2.7 – 2.2.9	6 / 6	(100%)
	stable	2.2.9	4 / 4	(100%)
Ubuntu Jammy	*all*	2.4.1 – 2.4.7	10 / 12	(83%)
	stable	2.4.7	2 / 2	(100%)

Table 3: Summary of LUCI executing the *Expat* test suite using different library builds (including off-the-shelf binaries).

5.2 libxcrypt

The *extended crypt library* [14] is a modern replacement for *glibc* `libcrypt.so.1`, providing various one-way hashing methods that are frequently used for user authentication. Several test cases are included in the source. Most of them solely

Build	***libxcrypt***		**updatable**	
custom (vanilla)	4.0.0 – 4.4.33	*all*	35 / 47	(74%)
		unqiue	19 / 31	(61%)
Debian Bullseye	4.4.10 – 4.4.18	*all*	7 / 7	(100%)
		unqiue	3 / 3	(100%)
Ubuntu Focal	4.4.10	*all*	8 / 8	(100%)
		unqiue	0	–
Ubuntu Jammy	4.4.18 – 4.4.27	*all*	4 / 4	(100%)
		unqiue	4 / 4	(100%)

Table 4: Program running all *libxcrypt* tests in parallel (using multithreading) on LUCI while exchanging library builds.

use the shared library interface without any knowledge about the internal structure and are therefore suited for our evaluation. However, we have to exclude three test cases because of severe memory leaks and division-by-zero bugs in conjunction with older releases. Our test program endlessly repeats each eligible test case in a distinct thread without any synchronization whatsoever, requiring LUCI to apply updates to a process with several active threads.

When building and testing all 48 releases of version 4, LUCI is able to dynamically update 74% of them. However, 11 builds are binary identical to the previous one (e.g., 4.4.28), and 5 builds only differed in their *BuildID* – still enabling LUCI to update 19 out of 32 unique builds with actual code changes during runtime.

During updates, we also observe a steady decrease in failing tests: While 9 (out of 25) test cases report errors when running with the first release of the library (version 4.0.0), there are no more unsuccessful tests after the latest update.

While *Debian Buster* still retains the *glibc* crypt library, *Bullseye* moved to this replacement library (package name `libcrypt1`), which can be fully dynamically updated as stated in Table 4. In *Ubuntu Focal*, one can find 9 different packages, but their code and data are all identical (having 5 different *BuildID*s). In contrast, *Jammy* has 5 actual different builds that are all compatible. Due to the fact that *Bullseye* and *Jammy* only have a single *stable* release, we consider *all* releases, including *development*.

The approach of LUCI does not restrict the update of a library concurrently employed by several dozen active threads in a multithreaded application, as these results show.

5.3 OpenSSL

Because of its broad application – and several severe vulnerabilities in recent years – the secure communication library *OpenSSL* has achieved a certain degree of brand awareness. We focus on its two main libraries `libssl.so` and `libcrypto.so` and set up a client-server environment using the `openssl` utility for testing.

Of 20 releases in OpenSSL 1.1.1, only 6 versions of `libssl.so` seem to be updatable and none of `libcrypto.so`. Further investigation showed that the `ssl3_undef_enc_method`-structure could be blamed: Its members point to functions that can reach `ERR_raise` using an array containing error messages, which are frequently edited in the source. This would not be a problem if the structure in question, which is – to the best of our knowledge – never modified, were marked as constant. However, since it is currently writeable and hence placed in the `.data` section, LUCI requires all of its references recursively to be identical to the previously loaded library for updates. Although LUCI's decision to reject such changes works just as intended, when temporarily relaxing this constraint, 9 releases in both libraries meet the requirements for an update.

However, after every update during testing, LUCI detects code usage in superseded libraries and hence reports it as failed. Again, the reasons for this shortcoming are function pointers in writable data: Instead of statically initializing a variable, *OpenSSL* does this during runtime[7] – taking over the work originally intended for the RTLD and hence leaving LUCI with no clue about those relocations, unable to correct them to the updated version.

Using `uprobes`, we can verify that only old code referring to unchanged functions was executed and the updates are effectively applied. However, LUCI is currently not meant to handle such code as further discussed in Section 6.

5.4 zlib

The library *zlib* is used in many software dealing with data compression and was not exempted from serious vulnerabilities. We are testing version 1.2 – since the version numbering switches between three and four places, there are 49 releases from 1.2.0 to 1.2.13 (the latest at the time of writing). Since the full code coverage test of the inflate algorithm cannot be used due to its interference with internal structures, we run the various de- and inflation tests of the *zlib* example file in an endless loop while updating the libraries.

Of all releases in the past two decades, only half are suited for dynamic updates, as the results in Table 5 show. The main reasons preventing an update are the 19 interface extensions during that time, which are often accompanied by additional data structures – both creating different debug hashes. The binary builds show an ambivalent picture as well: due to the interface changes and the fix for CVE-2018-25032 modifying internal structures, none of *Debian Buster*'s builds can be updated. The same CVE applies to *Debian Bullseye* and *Ubuntu Focal*, which have around half their packages eligible for dynamic updates. Only for *Ubuntu Jammy* (whose development began after discovering this CVE), all releases can be applied dynamically.

[7]For example, `names_lh` in `crypto/objects/o_names.c` is initialized with `NULL` and assigned once with a fixed value in a custom `RUN_ONCE` function during start.

Build	***zlib***	**updatable**		
custom (vanilla)	1.2.0 – 1.2.13	*all*	24 / 48	(50%)
		unqiue	24 / 48	(50%)
Debian Buster	1.2.8 – 1.2.11	*all*	1 / 3	(33%)
		unqiue	1 / 3	(33%)
Debian Bullseye	1.2.11	*all*	4 / 5	(80%)
		unqiue	3 / 4	(75%)
Ubuntu Focal	1.2.11	*all*	5 / 7	(71%)
		unqiue	2 / 4	(50%)
Ubuntu Jammy	1.2.11	*all*	4 / 4	(100%)
		unqiue	2 / 2	(100%)

Table 5: *zlib* de- and inflations tests using LUCI while exchanging library builds.

6 Discussion

Unlike several other DSU approaches, LUCI does not aim for general updateability and deliberately sacrifices some features: Most notably, the requirement for identical writeable data segments, which allows multiple library versions to coexist concurrently in memory, prevents changes of static variables. In case the library maintains some sort of state, it must not differ between the old and new versions for the same reason. Hence, changes to the initialization functions are prohibited. Consequently, record types have to be equal as well since they might be used in a shared state during the update transition (e.g., `structs` in heap or stack memory). If the lifetime of a pointer to an internal function outlasts the execution time of the function in which it was assigned, old code may be executed after an update. Furthermore, a related case can occur when a library function resides for a long time on the call stack, possibly because of an endless loop.

Accordingly, LUCI statically verifies the compatibility of data and interface before starting the update, while afterward dynamically detecting the execution of old code: To prevent abnormal terminations, LUCI takes a strict course, favoring false positives over false negatives. If a new version of a library does not meet all requirements and therefore is not eligible for dynamic updates, the process remains running in a valid state while LUCI notifies users about the non-updatable version, so they can manually restart the service. However, in case any subsequent update is again compatible with the currently active library, the update will be applied.

For many libraries, this pessimistic approach is sufficient to update most library versions, especially when it comes to minor (patch-level) changes like stable release branches of distributions: The approach's restrictions rarely apply to bugfixes, which most frequently require a timely deployment. The decreasing number of failures in the *Expat* test suite in Section 5.1 demonstrates the immediate effect of bugfix updates. However, extensive bugfixes and feature changes with cross-cutting changes cannot be applied, as seen in major version updates (e.g., *Expat* 2.3.0 $\rightarrow$ 2.4.0).

A notable exemption is *OpenSSL*: Although it is an interesting target for DSU due to its wide distribution, it is notorious for its code quality [4, 19], and hence the failing results are not surprising. While a first examination suggests that a few changes in its code (which we rather categorize as coding-standard fixes) would considerably improve its updateability, there are possibilities for LUCI to handle such libraries: By using a fine-grained code-access–detection method like `uprobe` and `ptrace`, further conclusions about the actually executed symbols in old libraries can be drawn. Accordingly, LUCI can ignore accessed symbols that are identical to their corresponding newer version. However, in contrast to the currently employed coarse-grained user-space page-fault mechanism, this would require additional permissions and induces overhead. Nevertheless, if a function pointer actually refers to a symbol that has been modified in the updated library, countermeasures are possible: LUCI could alter the old library's code in such a way that it redirects the control flow to the corresponding new version of the symbol (e.g., by using a `nop` slide) – but at the same time losing the advantage of simplicity and safety while not having to modify the `text` section.

It is worth mentioning that function pointers to symbols in other shared libraries do not point directly to the target but to the corresponding PLT entry – which LUCI fixes on update. Moreover, the use of virtual inheritance in C++ is not affected as well: Such objects are internally extended by an additional `vpointer` attribute, which references the `vtable` stored in the (relocation-)read-only section and is updatable.

A crucial limitation that LUCI shares with other automatic DSU approaches is the application of structural changes, which can become arbitrarily complex due to the semantic gap between (the intentions at) the source code level and the information available at the binary level after compilation. They are generally unrecognizable on a binary level without further context.

Listing 1 shows a contrived example of a change in a spinlock implementation: The semantic change of the value signaling the holding of a lock from `1` to `0` is only reflected in

```
typedef int lock_t;

void lock(volatile lock_t * var) {
-    while(!CAS(var, 0, 1)) {}
+    while(!CAS(var, 1, 0)) {}
}

void unlock(volatile lock_t * var) {
-    *var = 0;
+    *var = 1;
}
```

Listing 1: A contrived example of a code modification that LUCI cannot automatically reject, due to changes at a higher semantic level.

the functions' machine code. It is indistinguishable from a valid (i.e., bugfix) code change, and it does not change the interface. Consequently, LUCI cannot automatically detect this as an incompatible update. Even restrictions such as *activeness safety* [16], which prevents active functions (on the call stack) from updating, would allow an update point inside the critical section: For example, if one thread can hold the `lock` and the incorrect update is applied, then another thread can incorrectly acquire the `lock` a second time. However, proper software-engineering techniques requiring `lock_t` to be an `enum` with constants `LOCKED` and `UNLOCKED` would enable LUCI to detect the incompatibility using *DWARF* information and reject the update.

Data structures can pose similar issues regarding the detection of incompatible changes: While changes in `structs` are reflected in the debug information, a semantic modification of the access using only pointer arithmetic and casting is not detectable by LUCI. This lack of further information about the data layout is especially true for dynamically allocated memory. Further problems may arise if the modification of function parameters can have an impact on the process environment (e.g., adding a write-protection flag in `mmap`).

The problems described can be tackled with carefully chosen manual update points and hand-crafted (or test-cases–assisted [25]) state transformations. However, this would involve a significant amount of work for each library. For LUCI, we instead propose a more pragmatic solution by analyzing the change set at the source code level and explicitly marking a binary as *compatible* or *incompatible* with the previous version. Maintainers or distributors usually have the knowledge to perform this compatibility review. The resulting information can be included in the metadata of the binary (`note` section) or the package. Alternatively, the compatibility check can be carried out independently by third parties (e.g., by providing the information along with the debug hash).

To facilitate the compatibility review, LUCI has tooling support to automatically identify obviously incompatible versions (e.g., different writeable section). A further LUCI tool for simplifying the review shows the associated source-code lines that belong to the modified code in the resulting binary (using the *DWARF* symbols). With LUCI, we argue that less knowledge is required to decide on the update compatibility compared to manually writing update routines (e.g., introducing update points or writing state transformers).

In summary, Listing 1 illustrates that corner cases exist that circumvent LUCI's automatic compatibility check. Therefore, as mentioned, LUCI's tooling infrastructure assists the user in performing manual compatibility checks. However, all the libraries we have analyzed so far have well-structured code that does not require manual intervention. From this practical observation, we argue that LUCI's approach solves numerous real-world code-patching problems.

Regarding the aspect of LUCI's memory demand, we argue that having multiple coexisting versions of the library in memory is rather unproblematic: Non-writable segments are file mappings and, therefore, do not permanently reside in memory. The data segments exist only once due to LUCI's memory-aliasing technique.

7 Conclusion

Even though dynamic software updates are a well-received research topic and the benefit due to security concerns is undisputed, they barely made their way to user space on our everyday systems due to the required effort for software developers. In this paper, we propose a concept addressing the existing obstacles, hereby focusing on the most frequently reused kind of software: shared libraries.

Analysis of common bugfix patterns, including their effect on the resulting ELF files, allows the conclusion that the metadata is sufficient to enable a dynamic linker/loader to update today's binary-distributed shared libraries without requiring any changes or inducing additional runtime overhead.

To validate this claim, we have implemented our own dynamic linker/loader LUCI, which acts as an evaluation platform for live-patching common binaries: LUCI dynamically updates many versions of popular shared libraries like *Expat*, *libxcrypt*, and *zlib*. But even in the case of incompatibilities, normal execution is maintained: In no case does an update, neither successful nor failed, lead to an abnormal program termination or incorrect behavior during our evaluation.

The presented results suggest that the proposed approach is practicable and can play a part in paving the way for the common use of live-patching of user-space applications.

Although binary-compatible and supporting several of its interfaces, LUCI is not deemed a replacement for the existing default dynamic linker/loader (e.g., from the *glibc* project) but is intended to support further investigation and research for loader-based dynamic software updates. We hope that LUCI will help DSU become a reality in user space – and, for example, be supported by the standard RTLD some day.

Acknowledgments

The author order corresponds to the SDC (*sequence determines credit*) model. This work is part of a project campaign on *dynamic operating-system specialization* (DOSS) and is also funded in parts by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 465958100 and 502947440. We are grateful to the anonymous reviewers and our shepherd for their insightful comments.

Availability

The source code of LUCI and its components is published under the *GNU Affero General Public License, Version 3* at github.com/luci-project/luci.

A Artifact Appendix

Abstract

Our artifact includes the source of the dynamic linker/loader LUCI itself, alongside with a script to evaluate its live-update capabilities on the libraries *Expat*, *libxcrypt*, *OpenSSL*, and *zlib* using suitable tests. For building and testing, container environments (based on *Docker*) are employed. The artifact contains scripts to support automatically building these libraries from their official source and tools to download the corresponding Debian and Ubuntu packages. LUCI is intended for a recent Linux environment on an x86_64 architecture.

Scope

We aim to achieve two main goals with the artifact: Firstly, we want to encourage reproducing the results presented in this paper, the artifact therefore supports

- building the dynamic linker/loader LUCI, which is implemented according to Section 4,
- building several release versions of the libraries *Expat*, *libxcrypt*, *OpenSSL*, and *zlib* using the source from official repositories,
- validating the changes of the test suites, and
- running all experiments described in Section 5. This enables one to reproduce the results referred to in text and listed in further detail in Table 1, Table 2 and Table 3 for *Expat*, Table 4 for *libxcrypt*, and Table 5 for *zlib*.

Secondly, since we are aware that there is a lack of "hackable" dynamic linker/loaders (especially when it comes to *glibc*-compatibility), we provide LUCI for academic purposes, mainly but not limited to research on loader-based DSUs.

Contents

The dynamic linker/loader consists of the following internal sub-projects (each distributed in a separate repository):

`dlh` provides basic functionality similar to libc/STL for creating static freestanding applications (without *glibc*).

`elfo` is a lightweight parser for the Executable and Linking Format, supporting common GNU/Linux extensions.

`bean` binary explorer/analyzer to compare shared libraries and detect changes, which uses the Capstone Engine.

`luci` dynamic linker/loader with DSU capabilities and *glibc* compatibility (`ld-linux-x86-64`).

To build LUCI, it is sufficient to recursively clone the repository with its submodules and run `make` in the main folder. Further details are provided in the `README.md`.

For each shared library used in Section 5, there is a corresponding subfolder in the evaluation repository. With `gen-lib.sh`, the desired version(s) are built, `gen-test.sh` compiles the test program (located in `src-test`), and `run-test.sh` runs the experiments with automatic library exchanging in a containerized environment.

Hosting

Both LUCI's source code and the evaluation environment are available at github.com/luci-project/eval-atc23.

The utilities for building the shared libraries retrieve the source code from the following official repositories:

- github.com/libexpat/libexpat
- github.com/besser82/libxcrypt
- git.openssl.org
- github.com/madler/zlib

To acquire the Debian and Ubuntu packages released for each library, the utilities use the web services launchpad.net, metasnap.debian.net, and snapshot.debian.org.

Requirements

We have written all parts of the dynamic linker/loader in C/C++. A standard GCC (version 9 & 10) is sufficient to compile the project. While LUCI has no further restrictions on its build environment, its execution is currently limited to distinct versions of certain distributions, since LUCI must conform to the corresponding *glibc* interface (see Table 6).

To enable all features of LUCI, a Linux kernel 4.11 or newer with a default configuration is required. We recommend a Debian Bullseye installation using its standard kernel image.

We use *Python 3*, *GNU make*, and *Bash* for helper scripts. The tests are executed in a *Docker* container using the official Debian and Ubuntu images. The hardware platform should be an x86_64 architecture with at least 16 GiB of RAM and 6 GiB of storage.

Distribution	**Release**	**glibc**
Debian	Stretch (9)	2.24
	Buster (10)	2.31
	Bullseye (11)	2.31
	Bookworm (12)	2.36
Ubuntu	Focal Fossa (20.04)	2.31
	Jammy Jellyfish (22.04)	2.35
AlmaLinux Oracle Linux RedHat Enterprise Linux	9	2.28
Fedora	36	2.35
	37	2.36
openSUSE Leap	15	2.31

Table 6: Distributions currently supported by LUCI.

References

[1] Jonathan Appavoo, Kevin Hui, Craig A. N. Soules, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, Marc A. Auslander, David Edelsohn, Benjamin Gamsa, Gregory R. Ganger, Paul E. McKenney, Michal Ostrowski, Bryan S. Rosenburg, Michael Stumm, and Jimi Xenidis. Enabling autonomic behavior in systems software with hot swapping. *IBM Systems Journal*, 42(1):60–76, 2003. DOI: 10.1147/sj.421.0060

[2] Jeff Arnold and M. Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*, pages 187–198, 2009. DOI: 10.1145/1519065.1519085

[3] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *Proceedings of the USENIX Annual Technical Conference (ATC '05)*, pages 279–291, 2005.

[4] Bob Beck. LibreSSL - an OpenSSL replacement. Berkeley System Distribution (BSD), Andrea Ross, 2014. DOI: 10.5446/15347

[5] Derek Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004.

[6] Matthieu Caneill, Daniel M. Germán, and Stefano Zacchiroli. The debsources dataset: two decades of free and open source software. *Empirical Software Engineering*, 22(3):1405–1437, 2017. DOI: 10.1007/s10664-016-9461-5

[7] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live updating operating systems using virtualization. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE '06)*, pages 35–44, 2006. DOI: 10.1145/1134760.1134767

[8] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. POLUS: A powerful live updating system. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, pages 271–281, 2007. DOI: 10.1109/ICSE.2007.65

[9] Zhikun Chen and Weizhong Qiang. ISLUS: an immediate and safe live update system for C program. In *Proceedings of the 2nd International Conference on Data Science in Cyberspace (DSC '17)*, pages 267–274, 2017. DOI: 10.1109/DSC.2017.50

[10] Cory Cohen and Jeffrey S Havrilla. Function hashing for malicious code analysis. *CERT Research Annual Report*, pages 26–29, 2009.

[11] Robert P. Cook and Insup Lee. DYMOS: a dynamic modification system. In *Proceedings of the Symposium on High-level debugging (SIGSOFT '83)*, pages 201–202, 1983. DOI: 10.1145/1006147.1006188

[12] Christophe Cérin, Camille Coti, Pierre Delort, Felipe Diaz, Maurice Gagnaire, Marija Mijic, Quentin Gaumer, Nicolas Guillaume, Jonathan Le Lous, Stephane Lubiarz, Jean-Luc Raffaelli, Kazuhiko Shiozaki, Herve Schauer, Laurent Smets, Jean-Paul Seguin, and Alexandrine Ville. Downtime statistics of current cloud solutions. *The International Working Group on Cloud Computing Resiliency*, 2014.

[13] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The matter of heartbleed. In *Proceedings of the Conference on Internet Measurement Conference (IMC '14)*, pages 475–488, 2014. DOI: 10.1145/2663716.2663755

[14] Björn Esser and Zack Weinberg. libxcrypt – extended crypt library for descrypt, md5crypt, bcrypt, and others. https://github.com/besser82/libxcrypt, 2021.

[15] Christopher M. Hayden, Karla Saur, Edward K. Smith, Michael Hicks, and Jeffrey S. Foster. Kitsune: Efficient, general-purpose dynamic software updating for C. *ACM Transactions on Programming Languages and Systems*, 36(4), 2014. DOI: 10.1145/2629460

[16] Christopher M. Hayden, Edward K. Smith, Eric A. Hardisty, Michael Hicks, and Jeffrey S. Foster. Evaluating dynamic software update safety using systematic testing. *IEEE Transactions on Software Engineering*, 38(6):1340–1354, 2012. DOI: 10.1109/TSE.2011.101

[17] Christopher M. Hayden, Edward K. Smith, Michael Hicks, and Jeffrey S. Foster. State transfer for clear and efficient runtime updates. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *Proceedings of the 27th International Conference on Data Engineering (ICDE '11)*, pages 179–184, 2011. DOI: 10.1109/ICDEW.2011.5767632

[18] Haegeon Jeong, Jeanseong Baik, and Kyungtae Kang. Functional level hot-patching platform for executable and linkable format binaries. In *Proceedings of the International Conference on Systems, Man, and Cybernetics (SMC '17)*, pages 489–494, 2017. DOI: 10.1109/SMC.2017.8122653

[19] Poul-Henning Kamp. Please put OpenSSL out of its misery: OpenSSL must die, for it will never get any better. *Queue*, 12(3):20–23, 2014. DOI: 10.1145/2602649.2602816

[20] The kernel development community. The Linux kernel: Livepatch. https://docs.kernel.org/livepatch/livepatch.html, 2022.

[21] Raula Gaikovina Kula, Daniel M. Germán, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration. *Empirical Software Engineering*, 23:1–34, 2018. DOI: 10.1007/s10664-017-9521-5

[22] Andrew Lerner. The cost of downtime. https://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime/, 2014.

[23] Netcraft Ltd. December 2022 web server survey. https://news.netcraft.com/archives/2022/11/10/november-2022-web-server-survey.html, 2022.

[24] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. SPIDER: Enabling fast patch propagation in related software repositories. In *Proceedings of the IEEE Symposium on Security and Privacy (SP '20)*, pages 1562–1579, 2020. DOI: 10.1109/SP40000.2020.00038

[25] Stephen Magill, Michael Hicks, Suriya Subramanian, and Kathryn S. McKinley. Automating object transformations for dynamic software updating. *ACM SIGPLAN Notices*, 47(10):265–280, 2012. DOI: 10.1145/2398857.2384636

[26] Kristis Makris and Rida A. Bazzi. Immediate multithreaded dynamic software updates using stack reconstruction. In *Proceedings of the Conference on USENIX Annual Technical Conference (ATC '09)*, pages 1–14, 2009.

[27] Kristis Makris and Kyung Dong Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '07)*, pages 327–340, 2007. DOI: 10.1145/1272996.1273031

[28] Iulian Neamtiu and Michael Hicks. Safe and timely updates to multi-threaded programs. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*, pages 13–24, 2009. DOI: 10.1145/1542476.1542479

[29] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*, pages 72–83, 2006. DOI: 10.1145/1133981.1133991

[30] Martin Alexander Neumann, Christoph Tobias Bach, Stefan Kratochwil, Marcel Kost, and Michael Beigl. libDSU: Towards hot-swapping dynamically linked libraries on stock Linux. In *Companion Proceedings of the ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '16)*, pages 41–42, 2016. DOI: 10.1145/2984043.2989223

[31] Oracle. New userspace patching with oracle ksplice! https://blogs.oracle.com/linux/post/new-userspace-patching-with-oracle-ksplice, 2015.

[32] Kai Pan, Sunghun Kim, and E. James Whitehead. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009. DOI: 10.1007/s10664-008-9077-5

[33] Vojtěch Pavlík. kgraft: Live kernel patching. https://www.suse.com/c/kgraft-live-kernel-patching/, 2014.

[34] Mathias Payer, Boris Bluntschli, and Thomas R. Gross. DynSec: On-the-fly code rewriting and repair. In Cristian Cadar and Jeff Foster, editors, *Proceedings of the 5th Workshop on Hot Topics in Software Upgrades (HotSWUp '13)*, 2013.

[35] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP '09)*, pages 87–102, 2009. DOI: 10.1145/1629575.1629585

[36] Luís Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. Mvedsua: Higher availability dynamic software updates via multi-version execution. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, pages 573–585, 2019. DOI: 10.1145/3297858.3304063

[37] Luís Pina, Luís Veiga, and Michael Hicks. Rubah: DSU for Java on a stock JVM. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*, pages 103–119, 2014. DOI: 10.1145/2660193.2660220

[38] Josh Poimboeuf. Introducing kpatch: Dynamic kernel patching. https://www.redhat.com/en/blog/introducing-kpatch-dynamic-kernel-patching, 2014.

[39] Debian Project. Debian security FAQ: Why are you fiddling with an old version of that package? https://www.debian.org/security/faq.en.html#oldversion, 2021.

[40] Debian Project. Debian popularity contest. https://popcon.debian.org/stable/main/by_inst, 2022.

[41] Nguyen Anh Quynh, Tan Sheng Di, Ben Nagy, and Dang Hoang Vu. Capstone engine. https://github.com/capstone-engine/capstone, 2021.

[42] Ashwin Ramaswamy, Sergey Bratus, Sean W. Smith, and Michael E. Locasto. Katana: A hot patching framework for ELF executables. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES '10)*, pages 507–512, 2010. DOI: 10.1109/ARES.2010.112

[43] Christopher Salls, Yan Shoshitaishvili, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. Piston: Uncooperative remote runtime patching. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC '17)*, pages 141–153, 2017. DOI: 10.1145/3134600.3134611

[44] Edward K. Smith, Michael Hicks, and Jeffrey S. Foster. Towards standardized benchmarks for dynamic software updating systems. In *Proceedings of the 4th International Workshop on Hot Topics in Software Upgrades (HotSWUp '12)*, pages 11–15, 2012. DOI: 10.1109/HotSWUp.2012.6226609

[45] SUSE. Libpulp. https://github.com/SUSE/libpulp, 2022.

[46] Expat team. Expat XML parser: Software using Expat. https://libexpat.github.io/doc/users/, 2022.

[47] TuxCare. Libcare – patch userspace code on live processes. https://github.com/cloudlinux/libcare, 2020.

[48] W3Techs. Usage statistics of Linux for websites. https://w3techs.com/technologies/details/os-linux, 2022.

[49] Pei Xia, Makoto Matsushita, Norihiro Yoshida, and Katsuro Inoue. Studying reuse of out-dated third-party code in open source projects. *Information and Media Technologies*, 9(2):155–161, 2014. DOI: 10.11185/imt.9.155

[50] Asimina Zaimi, Apostolos Ampatzoglou, Noni Triantafyllidou, Alexander Chatzigeorgiou, Androklis Mavridis, Theodore Chaikalis, Ignatios Deligiannis, Panagiotis Sfetsos, and Ioannis Stamelos. An empirical study on the reuse of third-party libraries in open-source software development. In *Proceedings of the 7th Balkan Conference on Informatics (BCI '15)*, 2015. DOI: 10.1145/2801081.2801087

[51] Deqing Zou, Hao Wang, and Hai Jin. StrongUpdate: An immediate dynamic software update system for multi-threaded applications. In *Human Centered Computing*, pages 365–379, 2015. DOI: 10.1007/978-3-319-15554-8_30