Cali: Compiler-Assisted Library Isolation

Markus Bauer
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany
markus.bauer@cispa.saarland

Christian Rossow
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany
rossow@cispa.saarland

ABSTRACT

Software libraries can freely access the program's entire address space, and also inherit its system-level privileges. This lack of separation regularly leads to security-critical incidents once libraries contain vulnerabilities or turn rogue. We present Cali, a compiler-assisted library isolation system that *fully automatically* shields a program from a given library. Cali is fully compatible with mainline Linux and does not require supervisor privileges to execute. We compartmentalize libraries into their own process with well-defined security policies. To preserve the functionality of the interactions between program and library, Cali uses a Program Dependence Graph to track data flow between the program and the library during link time. We evaluate our open-source prototype against three popular libraries: *Ghostscript*, *OpenSSL*, and *SQLite*. Cali successfully reduced the amount of memory that is shared between the program and library to 0.08% (*ImageMagick*) – 0.4% (*Socat*), while retaining an acceptable program performance.

CCS CONCEPTS

• **Security and privacy → Software and application security**; *Systems security*; • **Software and its engineering →** *Automated static analysis.*

KEYWORDS

Library Isolation, Memory Isolation, Privilege Separation, Program Dependence Graph, Compiler, LLVM, Cali

ACM Reference Format:
Markus Bauer and Christian Rossow. 2021. Cali: Compiler-Assisted Library Isolation. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (ASIA CCS '21), June 7–11, 2021, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3433210.3453111

1 INTRODUCTION

Programs extend their own logic with external libraries, which ease the life of developers by offering APIs that abstract from common tasks. Whereas convenient and common practice, linking third-party libraries imposes a significant security risk. Libraries execute in the context of the main program, and thereby can freely access the program's entire address space and inherit the program's system-level privileges. At the same time, libraries often contain risky functionality, such as parsers, that do not really need to use the full program's privileges and address space. This lack of privilege separation and memory isolation has led to numerous critical security incidents.

We can significantly reduce this threat surface by isolating the library from the main program. In most cases, a library (i) neither requires access to the entire program's address space, (ii) nor needs the full program's privileges to function properly. In fact, even complex libraries such as parsers require only limited interaction with the program and/or system. Conceptually, there is thus little need to grant an untrusted library access to the program or to critical system privileges. Basic compartmentalization principles thus help to secure a program from misuse by untrusted code. First, *memory isolation* shields the program's sensitive memory from untrusted code parts (e.g., libraries). Second, *privilege separation* reduces the set of privileges of untrusted code parts.

Two recent attempts have proven that library isolation fosters memory isolation *and* privilege separation. Sandboxed API [15] assists developers in isolating the library into its own process (memory isolation) by providing isolation primitives that can be adapted to program-to-library interfaces. To enforce privilege separation, Sandboxed API allows the developer to configure seccomp-BPF [11] filter rules. Similarly, RLBox [30] lets developers split Firefox's libraries into different processes and again uses seccomp-BPF for confinement. Both systems provide primitives to ease library isolation for developers, but still require significant manual code changes ("*Migrating a library into RLBox typically takes a few days [...]*" [30]). Given the plenitude of programs and libraries, even such reduced manual effort will severely hinder the wide deployment of library isolation.

The core challenge of automated library isolation is the inherent historic assumption that programs and libraries share the same address space. Any attempt to split this address space (e.g., into different processes) breaks the underlying semantics, if not dealt with accordingly. Passing data across contexts is trivial with primitive data types, but quickly becomes challenging for complex objects. PtrSplit [24] is a first general attempt to tackle this issue by marshalling complex data types whenever they cross boundaries. Methodologically, PtrSplit tracks bounds of complex objects to learn necessary size information that is required to copy complex objects. While such an analysis allows for automation, (i) deep copies create significant memory and performance overhead (essentially duplicating objects passing to/from libraries), (ii) parallel computation on copies will lead to data inconsistencies, and (iii) PtrSplit's analysis cannot handle type casts. Especially the latter restriction is problematic in practice, as libraries commonly make use of type casts which undermine PtrSplit's analyses. For example, LLVM's `memcpy` cannot be tackled because operands are casted to `void*`, the same holds for generic callback arguments found in various common libraries.

ASIA CCS '21, June 7–11, 2021, Hong Kong, Hong Kong

In this paper, we present Cali, a compiler-assisted library isolation system that uses *shared memory* to allow a secure and efficient interaction between a program and its libraries. Using shared memory makes the use of pointer marshaling obsolete, dropping all its associated disadvantages. The core challenge of Cali is preserving the full functionality of the library invocations, while minimizing the program parts that are exposed to the library. A naïve solution could place the entire program memory in shared memory that is accessible to the program *and* the library, which gained little security as the library had full write access to the main program's data (including pointers). Instead, Cali leverages a Program Dependence Graph (PDG) [12] to infer which memory allocations will (potentially) be passed from the program to the library. To this end, during link time, the PDG observes and propagates data flows crossing the security contexts. Cali then places the according memory regions in shared memory, and isolates the remaining memory in the application and library processes, respectively.

Cali is the first system that *fully automatically* shields a program from libraries. Cali is compatible with mainline Linux, does not require supervisor privileges, and has an acceptable space and performance overhead. Cali compartmentalizes untrusted libraries into their own contexts with well-defined security policies. We provide both privilege and memory isolation by placing libraries in their own process that operates in per-compartment kernel namespaces. We apply seccomp filters to minimize the system interface of the library.

We implemented Cali based on LLVM and published its source code.[1] We evaluate the functionality, space and performance overhead of our prototype against three popular libraries: *Ghostscript* (tested with *ImageMagick*), *OpenSSL* (tested with *Socat*), and *SQLite* (tested with *Filezilla*). Cali reduced the amount of memory that is shared between the program and library to 0.08% (*ImageMagick*) to 0.4% (*Socat*). Not a single function pointer is shared, such that the threat surface is greatly reduced. Cali's compartmentalization has an acceptable performance overhead when limiting libraries to their least privilege in both memory and system access.
To summarize, our contributions are as follows:

- We present a fully-automated compiler-based separation between trusted and untrusted program logic using compartmentalization and data flow tracking.
- Cali transforms any program to a protected version that can be deployed on mainline Linux, without additional requirements about the hypervisor, OS or hardware.
- The strict security boundaries of the resulting programs greatly reduce the threat surface, while our three examples show that the performance overhead is acceptable.

2 BACKGROUND / RELATED WORK

Program compartmentalization is the foundation of two related research directions with orthogonal goals. *Memory isolation* hides certain secrets (e.g., keys) in program memory from other parts/-compartments. Typically built on top of memory isolation, *privilege separation* [34] also limits system access of certain code parts (compartments) to the least privilege. In either way, compartmentalizing an application is a two-step process: First, an application needs to be separated into two or more distinct code parts (cf. Section 2.1, "Compartmentalization"). These code parts must be able to communicate with each other and keep the functionality of the original application. Second, the now distinct code parts need to be isolated from each other and execute in different contexts (cf. Section 2.2, "Isolation Primitives"). Isolation must guarantee the integrity of the trusted context even if the untrusted context acts maliciously.

We will provide background information and discuss related work in the following. To assist in this discussion, Figure 1 provides an overview of key related works, categorized into the two dimensions of degree of automation (y-axis) and their provided security guarantees (x-axis). Cali aims to fill a gap by providing strong security guarantees (both memory isolation *and* privilege separation), and at the same time offering an unprecedented level of automation.

2.1 Compartmentalization

The few past manual efforts to split program into compartments (e.g., Google Chrome, OpenSSH) have shown that splitting a program into isolated parts is a tedious and error-prone task. This motivated research on several compartmentalization libraries that aid developers in this process. For example, Privman [21] can be used to split applications in a privileged server and an unprivileged client, requiring changes to the source code: Any library interaction needs to be rewritten manually, any memory transfer needs to be performed by hand. Privman is merely a library providing the isolation primitives. In addition, Google recently published their framework "Sandboxed API" [15], that can be used in C++ programs to isolate C libraries in a compartment process that has only limited capabilities. To apply Sandboxed API, each and every usage of the library needs to be rewritten manually, which is neither trivial nor convenient. A similar framework is RLBox[30], with a much stronger type system enforcing additional security properties. Thus, summarizing, while all these libraries aid developers significantly, they still do not entirely eliminate the manual analyses and efforts.

In order to lower the manual effort of compartmentalization, researchers proposed various assistance tools that give additional information where and how to separate the application. These tools are based on annotation-guided program analysis [3, 18, 25, 48] or dynamic analysis [2, 26]. Unfortunately, the strict dependence on source code changes (or annotations) implies that software can only be compartmentalized by experienced developers with deep knowledge of the source code and library interactions. This lack of automation does not scale for the wide variety of open-source off-the-shelf software.

Researchers have already identified this urgent need for automation. For example, Codejail [47] can separate program privileges *without* forcing developers to rewrite or know the program code *in the program*. However, to separate a program, Codejail requires that every library function needs to be "described" by a developer. To this end, wrapper functions need to be written by hand, and they must specify every single memory transfer between program and library. Furthermore, Codejail's memory isolation is much weaker than related systems. In particular, the untrusted library can read arbitrary memory from the trusted program (hence bugs like Heartbleed cannot be contained). While these wrappers can

[1] https://github.com/cali-library-isolation/Cali-library-isolation

be written once for a library and then be used in many different projects without hassle, Codejail still leaves open the hard work for developers.

Two existing approaches come close to the degree of automation we envision, PtrSplit [24] and SOAAP [18]. Yet there is one fundamentally hard challenge for automating compartments: data flows of non-primitive objects (e.g., pointers, structs) between the compartments. "PtrSplit" [24] uses static analysis to separate annotated variables and related code. In contrast to prior work, PtrSplit can—assuming code annotations by a skilled developer—infer a separation boundary automatically. Furthermore, as PtrSplit tracks data flows, it marshals complex objects for IPC communication between the compartments. While this boosts automation significantly, it still requires code annotations, which in turn require program knowledge. And while PtrSplit provides memory isolation, it cannot limit privileges of compartments. Facing these challenges, SOAAP even completely left open isolation (and the required automation) open for future work.

Cali automates compartmentalization to the highest degree possible. Given a program and its libraries, Cali fully automatically splits the program from (a developer-specified subset) of libraries. Developers do not have to care about library interfaces, nor have to know at which parts of their program a library has been integrated. The resulting compartments provide memory isolation *and* privilege separation. Cali only assumes a policy that guides privileges of each program part, which could be derived automatically [10, 13, 14, 42].

2.2 Isolation Primitives

Once a program has been compartmentalized, we have to use certain isolation primitives to protect the program parts from each other. That is, assuming well-defined compartment boundaries and interfaces, how can we efficiently enforce isolation between the compartments? To this end, multiple isolation principles can be used, most of which focus only on memory isolation. The conceptually simplest approach is *Software Fault Isolation* (SFI) methods like NaCl ("upro" [32]) or WebAssembly (RLBox [30]), which compile untrusted program parts into sandboxes. SFI requires source code of the program and all libraries, and it comes with major restrictions in functionality—not all programs can be compiled to SFI schemes (e.g., JIT compilers). To solve this problem, other systems used existing OS functionality like executing compartments using different Unix users [21]. To further enhance capabilities, the OS kernel can be modified [2, 19, 23, 38, 43], however sacrificing compatibility with unmodified kernels. In the same spirit, researchers leveraged virtualization extensions of modern CPUs and introduce hypervisors for memory isolation (e.g., "SeCage" [26], "TrustVisor" [27], and "Libsec" [35]) Recently, researchers further boosted isolation primitives with hardware-specific features (CPU extensions, etc) [4, 17, 22, 36, 41], which again reduce wider applicability. All these works demonstrate that special OS or hardware features can be elegantly use to further boost memory isolation, which is important when aiming to protect certain regions of sensitive data. Yet very few of these approaches are compatible to (e.g., process-based) privilege separation. Even if, they require certain features and/or adoptions, which hinders a wider isolation deployment in the wide world of resource-constrained devices (think of IoT).

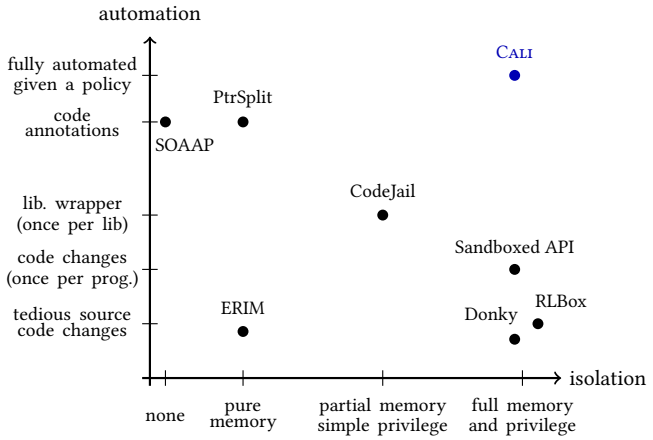


Figure 1: Overview of recent program isolation schemes

3 GENERAL OVERVIEW

3.1 Compiler-Assisted Library Isolation

We now describe Cali, a compiler-assisted compartmentalization privilege separation solution. Our main use case are developers or package maintainers who link common libraries into a (Linux-based) application, written in a native language like C or C++. The library is given in binary form only, source code is only available from the main program—a quite common scenario if a third-party library is proprietary, or closed-source drivers.

We envision that one of these libraries contains a severe vulnerability, giving attackers full control over the entire program. The developer wants to limit the damage that can occur from such a vulnerability. In particular, we want to protect *private information* (from application memory or files, like stored passwords) and *system integrity* (no system modifications like backdoors etc). We do not want to protect from *Denial-of-Service* (DoS) attacks that crash the application—*fault recovery* is out of scope. We aim, however, to mitigate resource exhaustion attacks that block the whole system (e.g., memory exhaustion, fork bombs) by limiting the compartment's computation and memory resources.

The application should not lose portability, it must continue working on any system where it worked before modification. No system modification is desired, and super-user permissions are neither available at installation time nor at runtime.

Threat model: We envision that a library contains a severe vulnerability that gives attackers full control over the entire process running library code, including arbitrary code execution. We do not consider attacks against the kernel, the hardware or micro-architectural attacks. We also assume a sound permission configuration ([10, 13, 14, 42]). We will discuss our assumptions on and mitigation of cross-compartment vulnerabilities in Section 6.4.

3.2 Overview

Cali performs privilege separation by isolating less-privileged code parts into their own context. While this concept is generic, in the following, we will stick to our main use case of library isolation. Cali automatically handles the interaction between the application and library, which now requires inter-context communication. Cali then reduces the permissions of the library context to the minimally required privileges and memory.

Our protection is applied at application build-time. We build the application using the LLVM toolchain with *link-time optimization* (LTO). When linking the application, all source code files are available as LLVM bitcode. In a first step, we perform a static, inter-procedural analysis over the whole application. We determine where and how the library is used and what resources like memory regions it needs from the main application. In a second step, we rewrite the main application. We replace all library calls with calls to stub functions that take care of the transition between application and library context. To this end, we make the minimal necessary program memory regions accessible for the library context.

After linking, the result is a normal Linux binary that runs on any Linux system without additional dependencies. The required interaction of the developer is minimal: Switch the compiler toolchain to Clang/LLVM with LTO, add a compiler flag to enable our system, and specify the permissions applied to the library context in a simple format (see Appendix A for a concrete example of a permission configuration). Furthermore, our design guarantees compatibility. Our context implementation uses primitives (processes, namespacing, seccomp and semaphores) that are readily available in mainline Linux.

4 SHIELDING COMPARTMENTS

Cali creates a compartment for all libraries that should be isolated. A compartment must fulfill three criteria: First, it must provide *privilege isolation*. To this end, we must be able to constrain the access on files, network and other resources (including computation power, memory, etc.) to successfully contain vulnerable libraries. Second, a compartment must provide *memory isolation* to protect the privileged main application (e.g., its pointers) from an attacker in the library compartment. Third, a compartment must preserve the functionality of a program, without requiring modifications to the library. Memory chunks that are passed from the program to the library must be accessible from the compartment and vice versa.

4.1 Basic Compartment Structure

For each library compartment, we fork a new library process from the main process just before the libraries are initialized. This process is restricted in its permissions and only shares selected memory regions with the main program. Communication between compartments happens over shared memory, semaphores and an anonymous socket. Library compartment processes sleep until a library function is invoked. Once woken up, they execute the called library function on behalf of the main process and return the result. They terminate when the main process terminates.

Such library compartments do not undermine memory deduplication, the isolated library is still a *shared* library mapped copy-on-write. Multiple processes sharing the library will require one copy in memory, only. Also nested libraries are not negatively affected by this scheme. If a library loads further libraries, these will be executed in the context of the loading library, inheriting its reduced privileges.

In principle, we can create compartments for any library. Having said this, we do not isolate standard libraries (e.g., libc). They are the usual interface to access the underlying system. Therefore, standard libraries would need all privileges the main program needs. In our design, each context has its own standard library. It does not matter if a library uses raw syscalls or libc—both execute from the library context with identical privileges.

4.2 Shared Memory

We create a segment of shared memory for each compartment and map it in both the main and library process. This memory is mainly used to allocate memory chunks that are accessed by both processes. To organize this memory, we build drop-in replacement versions for `mmap`, `mremap` and `munmap` handling memory from this shared memory pool page-wise. These functions, called `shm_mmap` etc., also synchronize memory mapping between both processes.

We support dynamic memory allocation (e.g., `malloc`) by using a modified version of glibc's heap implementation working with shared memory. In this modified version, we remove the main arena (which cannot be shared) and replace all calls to the `mmap` family with calls to the `shm_mmap` family. Next, we utilize glibc's per-thread heaps to build per-process heaps, preventing concurrency issues between main and library process. Following this principle, we have drop-in replacements for all basic memory-allocating functions. If a chunk of memory needs to be shared with the library, we only need to replace its allocation with the appropriate shared allocation. This way, no memory needs to be copied between processes, improving the efficiency of memory transfers. Any memory outside of this shared memory pool is not accessible by the library.

4.3 Library Calls

For each library function called by the main program, we create a replacement function in the main program and a handler in the library compartment. All calls to library functions are passed to the library process using a custom IPC-based protocol. We rewrite all calls to the library with calls to this replacement function. This function stores all call-by-value arguments (numbers, pointer values, but not the memory pointed to) in shared memory and signals the library process using a semaphore. The library process invokes the handler which loads arguments from shared memory and invokes the original function. Once the function returns, the result is stored back to shared memory and the main process receives a signal using a second semaphore. Finally, the replacement function in the main process loads the return value from memory and returns. This design is completely transparent to program and library, as long as all pointer arguments point to shared memory (which we will ensure in Section 5).

4.4 Callbacks, Signals and File Descriptors

Sometimes libraries expect a callback that they will execute once an event occurs, or programs receive function pointers from a library that they will call later on. In our context, this is dangerous. Callbacks allow one process (e.g., the library) to trigger the execution of code in another process (e.g., the main program), inheriting the program's privileges. To keep up the isolation between processes, we employ a strict policy: Callbacks are executed in the process they come from (where they have been defined).

Whenever a function pointer is transfered by a library call, we create a replacement function on the fly, and store the original pointer in a lookup table. When invoked, the replacement function invokes the call in the other process.

This design complies with the usual structure of code containing callbacks. Functions passed from the program to the library were typically written in the program's code base and require access to the program's internal data. Executing them in program context thus preserves compatibility. Functions returned and defined by the library might depend on library-internal data and should stay in the library compartment, for compatibility and security reasons. The library can not invoke arbitrary code, function pointers it passes execute in the library process.

Signals are handled using this callback mechanism: When a signal handler is registered in one compartment, the handler is synchronized with all other compartments. When a signal is caught, the handler executes in the compartment that registered it. If an uncaught signal terminates one process, the others also terminate.

File descriptors are handled in a different way. We detect them using static analysis (Section 5.7), not by type. When passed as a function call argument or return value, the other side gets full access on the descriptor. A duplicate is handed over on a shared socket, FD numbers are adjusted between the processes. When the descriptor gets closed in one process, the descriptor is also closed in the other one. In Linux most system resources are represented by file descriptors, correct synchronization ensures a synchronized view of the system.

4.5 Isolation

Isolation between processes is provided by OS primitives present on any up-to-date Linux system. The exact isolation can be specified by an isolation policy and might depend on runtime data (environment, arguments, etc.). The policy is given by the developer at compile-time, Appendix A shows an example policy. In the following, we describe several isolations mechanisms that we deploy using a modified version of `nsjail` [16].

We put each compartment process in a new *mount namespace*, mount all accessible directories to an empty folder and finally use *chroot* to jail the compartment into this directory. We utilize a *user namespace* to execute chroot without requiring higher privileges or capabilities. As a result, the library compartment process sees a file structure similar to the real system, but it contains only folders if access has been allowed by the policy. If only read access is desired, we mount the folder with the read-only attribute.

We use a *network namespace* to prevent a library compartment from communicating with other machines, if not allowed by the isolation policy. Next, we use a *PID namespace* to protect other processes running on the system. After forking the compartment process, we drop Linux capabilities or superuser rights the main program might have, according to the isolation policy.

The isolation policy can specify constraints on the computing resources used by the library compartment, enforced using *rlimit*. These constraints prevent DoS attacks on the system, like consuming all available memory, blocking all CPU cores, or "fork-bombs". They are not meant to prevent application DoS (e.g., program crashes).

Finally, we apply a *seccomp* policy to restrict the set of system calls the library compartment can call.

4.6 Threading, Forks and Concurrency

Our prototype has limited support for concurrency: While it does not break the semantics of threading and forking processes, and in fact, also works for multi-threaded or multi-processing programs, our locking mechanisms serializes all threads at the library interface. Therefore, concurrency is not an issue, as only one thread can call a library at a time. To get the full performance of concurrent execution, the library compartment process must be enabled to spawn its own threads, mirroring the threads in the main program process. For new threads, the communication structure needs to be cloned as well. Forking programs can be handled by the same structure. In contrast to other work (that uses thread-like primitives for isolation), this extension does not impact security. However, we refrained from extending our prototype to multiple threads because this is not required to show the general feasibility of our approach.

Concurrently running code from multiple compartments opens up another problem: An attacker in one compartment could modify shared memory while another compartment uses this memory, possibly leading to memory corruption in other compartments, effectively weakening the introduced isolation. These issues are called *double-fetch bugs* [37, 44]. Other systems like PtrSplit or Sandboxed API avoid this problem, they copy memory instead of sharing it. But this approach breaks existing software with legitimate use cases of shared memory: for example, most implementations of synchronized collections or spinlocks rely on shared memory.

Cali solves this issue by providing three modes of operation: In the *default mode*, concurrent access is allowed (to not break existing software). We consider the security impact of concurrent access rather low, manual inspection of the shared memory usages in our three example programs did not show vulnerable memory usage. Related work [37] can be used to counter potential double-fetch bugs.

If the program is clearly not concurrently accessing memory, Cali can be used in *mprotect mode*. After a library function has finished executing, the shared memory is set read-only in the library process. Before the next library function call, the shared memory is set writable again. A custom seccomp filter is installed to prevent attackers from changing the protection of shared memory manually. With this extension, an attacker in the library process can only modify shared memory while a library function is being executed. Even if the attacker has started additional threads, the security level is equal to memory-copying solutions. The downside of this mode is that only one thread can execute a library function at a time.

If the library is not concurrent itself (e.g. does not spawn threads or processes), Cali can be used in *non-concurrent mode*. The library process uses a seccomp filter to prevent forks, clones or thread-spawning syscalls. Without these privileges an attacker can not run code outside of library calls: After a library call returned and the main compartment continued execution, the single library thread is blocked until the next library call is requested. This mode does not have additional overhead but prevents libraries from using concurrency.

5 COMPILER-ASSISTED SEPARATION

To automatically split an application into two parts, we have to know which memory chunk is used both by the program and the library, and which memory is used exclusively by the main program. We call chunks *common memory* if they are used by both compartments. Ideally, all other program memory should not be taken from the shared memory pool, as it otherwise might leak data to the library compartment. Having said this, sharing "too much" memory only weakens security guarantees and does not break functionality.

One core challenge is that we need to know *at allocation time* if a memory chunk is *common memory*. After memory has been allocated, moving the chunk might interfere with legacy code, e.g., pointers to the chunk scattered over the program would need to be updated. Similarly, we also have to identify all (potentially indirect) calls to library functions. They determine which memory is going to be accessible by the library.

To gather all this information, Cali uses *inter-procedural static analysis* on the compiled LLVM bytecode of the main program. In the first step, we use a *program dependence graph* (PDG) with similarities to the one proposed by Liu et al. [24], tracking data flow of memory chunks. We detect all calls to library functions and tag all memory allocations that might reach library functions. In a second step, we rewrite all these memory allocations to use shared memory, we generate replacement functions for used library functions, and finally, rewrite all calls to library functions with calls to these replacements.

5.1 Background: Call Graphs and SCCs

To schedule our analysis operations on the program, we use the *strongly connected components* (SCC's) of the call graph. A strongly connected component of a graph $G = (V, E)$ is a maximal subset of vertices V' where a path between all vertices exists ($\forall v_1, v_2 \in V'.\ v_1 \rightarrow^* v_2$). The graph formed by all strongly connected components in a call graph has nice properties: First, functions that can call each other in a recursive way (nested recursion) are contained in the same SCC. Every other function is in its own SCC of size 1. Second, because all recursive functions are contained in joined SCCs (one per recursive function group), the call graph of all SCCs is acyclic (circles in a call graph indicate recursion, which only happens inside SCCs). Third, traversing the SCC callgraph bottom up traverses all functions in a callee-before-caller order. Traversing the SCC callgraph top down traverses all functions in a caller-before-callee order. To reduce the average SCC size, we only consider calls that are able to transfer memory by reference (not only by value). We ignore calls if all parameters and the return value are constant or of primitive type (`int`, `char`, etc.), because they are not relevant for our following analysis.

Our analysis traverses a SCC callgraph of the whole program in a bottom-up fashion and analyzes all functions in a SCC at once. If we encounter a function call, it either targets a function we already analyzed or a recursive function in the same SCC which we are just analyzing. At a later stage, we will traverse the SCC callgraph top-down, to propagate information from calls to called functions.

5.2 Analysis Phase: Overview

In the analysis phase, we mainly need to determine which memory allocations generate *common memory* (that later needs to be shared). The analysis consists of three phases:

(1) Creation Phase: We construct a PDG containing information about intra-procedural data flow. We mark memory allocations and locations of common memory.
(2) Reachability Phase: For each function group (callgraph SCC), we determine the reachability between memory allocations, common memory expectations, the function's arguments and return value. We store the result in a function summary in the PDG, which is used when calls to this function are analyzed (inter-procedural data flow).
(3) Specialization Phase: We check for functions that should return pointers to common memory depending on their usage. For example, wrappers around `malloc` like `calloc` or `new` should only generate common memory if their result is passed to the library later on. However, these functions are called from many other functions. To keep a precise result, we thus clone these functions (including their containing SCC). The cloned functions will return common memory, while the original functions will not. Call sites are adjusted and reachability analysis is repeated.

Figure 2 shows an example program that we will use to illustrate the analysis. It consists of two struct instances, where one of these structs is used in a library, and the other is not. The reference to that struct passes multiple functions before being used as a library function argument. Figure 8 in Appendix C shows the LLVM translation of the program. The full PDG with all analysis results is given in Appendix C, an excerpt illustrating the most important aspects is shown in Figure 3.

5.3 PDG Construction

In contrast to other PDGs [12, 24], we do not need control dependence in our graph. The construction of our graph is based on the LLVM bytecode of the program. This bytecode is in *static single assignment* (SSA) form, meaning that every LLVM value gets assigned only once (at definition time, typically as result of an instruction) [33]. Our PDG is based on LLVM values, every value is represented by a PDG node. Nodes are additionally tagged with the type of the value and the function where it is contained, so every node is a three-tuple: $(value, type, function)$. We add similar nodes for all global variables and function arguments.

To trace actual memory chunks, we inspect the values further. We disassemble every complex data type (e.g., pointers, structs) into its basic data types. These so-called "subnodes" represent single members of structs or memory referenced by a pointer type in

```c
struct X { long one; long two; };

// Library function we need to compart
void libfunc(int *err, char *input, long *output);

// Main program
void main() {
    struct X *x1 = new_struct(13L);
    struct X *x2 = new_struct(37L);
    struct X *x3 = update(x2);
    char *buffer = malloc(1024);
    lib_wrapper(buffer, x3);
}

struct X *new_struct(long init) {
    struct X *s = malloc(sizeof(struct X));
    s->one = init;
    return s;
}

struct X *update(struct X *x) {
    x.one = 18;
    return x;
}

int lib_wrapper(char *buffer, struct X *x) {
    int err;
    libfunc(&err, buffer, &x->two);
    return err != 0;
}
```

Figure 2: Example program passing memory to a library

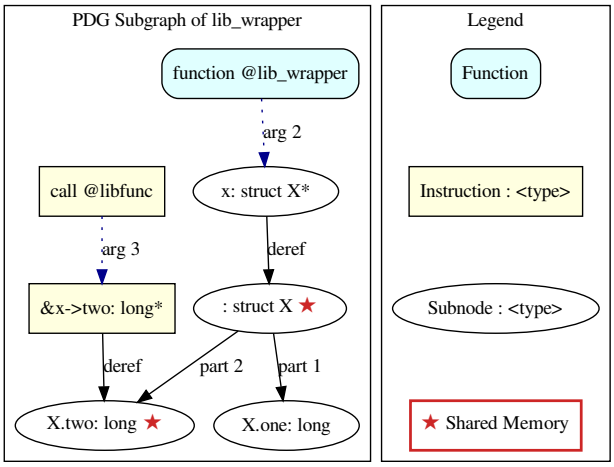


Figure 3: Excerpt of the PDG from the example program. The full graph is given in Appendix C / Figure 9.

the graph. The rules to create subnodes are given in Figure 4: For every pointer type, we create a subnode representing the memory pointed to, and connect it with a "pointer $\xrightarrow{\text{Deref}}$ memory" edge. For every field in a struct or union type, we create a subnode and connect it with a "struct $\xrightarrow{\text{Part}_i}$ field" edge (where i is the index of the field). We repeat this algorithm on all new subnodes up to a configurable recursion depth of 5. Limiting the depth avoids that the analysis is trapped in endless recursion induced by recursive structs. In theory, this limitation might lead to a loss of precision (when dataflow is hidden deep in the subnodes). However, in all our example programs, a recursion depth of 3 was sufficient to cover all necessary information. We also limit the number of struct members to 32 for performance reasons. In our examples, we did not see any impact on the analysis precision by this restriction. Figure 3 shows an example subgraph for the second argument of `lib_wrapper` (the struct pointer `x: struct X*`). The subnodes represent the dereferenced struct of that pointer (`:struct X`), further dissected into its fields (`X.one`, `X.two`).

5.4 Data Flow in PDGs

We next extend the PDG with edges representing intra-procedural data flow. If the LLVM value of a node or subnode n might carry over to another node n' in any possible execution of the program, we assume a data flow from n to n' and add an edge $n \xrightarrow{\text{data}} n'$. A typical example is a load from memory: Reading `err` for the return statement in `lib_wrapper` is `%6 = load i32* %3` in LLVM, we summarize the load as a data edge from the subnode `err` of `%3` (representing the referenced memory) to the output value `%6`.

As a major difference to previous PDGs [24], we use is *data equality* to capture pointer aliases. If two (sub)nodes represent the same value storage (i.e., the same memory location is referenced by two pointers), we consider them to be *data-equal*. If one node's associated value gets updated, the other node's value will also change. We could handle this situation with bi-directional $\xrightarrow{\text{data}}$ edges, but merging significantly reduces the size of the graph and improves the runtime of all further operations. Merging nodes might introduce cycles into the subnode graph if subnodes connected to each other are merged (e.g., linked list structure). The subnodes of a value no longer form a tree, which is vital for handling recursive data structures. Merging nodes also eliminates nodes introduced by type casts.

We use six rules for LLVM instructions that describe data flow between inputs and output of an instruction (Figure 5). To be on the safe side regarding functionality, our rules might over-approximate data flow, but should not under-approximate it.

Data flow between nodes x and y ($x \xrightarrow{\text{data}} y$) is propagated recursively to its subnodes, by two simple rules:

(1) If there is data flow between two structs x and y, then there is also data flow between their members:
$x \xrightarrow{\text{data}} y \wedge x \xrightarrow{\text{Part}_i} x' \wedge y \xrightarrow{\text{Part}_i} y' \implies x' \xrightarrow{\text{data}} y'$

(2) If there is data flow between two pointers x and y (pointers might alias), then we consider the referenced memory data-equal: $x \xrightarrow{\text{data}} y \wedge x \xrightarrow{\text{Deref}} x' \wedge y \xrightarrow{\text{Deref}} y' \implies x' = y'$

Rule 1 expresses that copying a struct from one location to another implies that all struct members are copied, too. Rule 2 can best be explained with the running example. In function `main`, data flows from `%2` to `%3`. Therefore, the referenced memory is the same

5.5 Reachability Analysis

After these inferences, the PDG contains all values in a program and their intra-procedural relations. Equipped with this PDG, we can determine if a memory allocation must produce common memory for a library call in the same function. If there is a data flow path from the memory allocation output (*source*) to a library call's arguments (*sink*), then the memory must be shared. Such data flow problems can be modeled as reachability over $\xrightarrow{\text{data}}$ edges in the PDG.

$$
subnodes\,(n,t) \mapsto \begin{cases} \emptyset & \text{for primitive } t \text{ like } \texttt{int} \text{ etc.} \\ \{node\,(t')\} \cup subnodes\,(node\,(t'),t') \;;\; n \xrightarrow{\text{Deref}} node\,(t') & \text{for } t = t'* \\ \bigcup\limits_{k \le n} (\{node\,(t_k)\} \cup subnodes\,(node\,(t_k),t_k)) \;;\; n \xrightarrow{\text{Part}_k} node\,(t_k) & \text{for } t = \texttt{struct}\{t_1, t_2, ..., t_n\} \\ subnodes(n,t') & \text{for } t = t'\texttt{[k]}, k \in \mathbb{N} \end{cases}
$$

Figure 4: Recursive algorithm to generate subnodes of a value node n with type t. $node(t)$ creates a new subnode with type t.

(1) `x = Load y`: The memory referenced by y is loaded into x.
$\forall y' : y \xrightarrow{\text{Deref}} y' \implies y' \xrightarrow{\text{data}} x$.
(2) `Store x, y`: The value x is stored to memory location $*y$.
$\forall y' : y \xrightarrow{\text{Deref}} y' \implies x \xrightarrow{\text{data}} y'$.
(3) `x = GetElementPtr y c1 c2 ...`: Compute address $\&y[c_1].c_2$
- If instruction computes the pointer to a valid field of $*y$:
$\forall z, x' : y \xrightarrow{\text{Deref}} y' \xrightarrow{\text{Part}_{c_2}} z\,,\; x \xrightarrow{\text{Deref}} x' \implies x' = z$
(z is the struct field in memory, x' is the memory pointed to by the output pointer).
- Otherwise: $x = y$ (output pointer is equal to the input pointer, handles dynamic array access etc.).

(4) `x = BitCast y` and other casts: $x = y$.
(5) `Call` is ignored, inter-procedural analysis will happen later.
(6) All other instructions which output a value are handled similar: All operands data-flow to the output value of the instruction.

Figure 5: Rules to determine data flow for LLVM instructions

First, we use data flow to determine which indirect calls might call a library function (find sinks). Then, we determine which nodes are common memory (reach a sink) within a function. Finally, we extend this to an inter-procedural analysis.

Determine indirect library calls: To decide if a call needs common memory, we need to decide if it could potentially call a library function. While this is trivial for direct calls, it is not obvious for indirect calls. Some programs, for example, build a `struct` of library function pointers as an exchangeable "interface" against a library. We search for paths backward in the PDG from the called function address (sink) to a library function (source). The backward path from sink to source must only consist of $\xleftarrow{\text{data}}$, $\xleftarrow{\text{Part}_i}$ and $\xrightarrow{\text{Deref}}$ edges. From the indirect call address (sink), we follow $\xleftarrow{\text{data}}$ edges backwards that lead us towards the origin of the address value. We also follow $\xrightarrow{\text{Deref}}$ edges in forward direction (pointer to memory) to get from function addresses to actual functions (sources). $\xleftarrow{\text{Part}_i}$ is used to cope with compiler optimizations.

This analysis might over-approximate which call might invoke a library function to preserve program functionality. Having said this, we did not observe such an over-approximation in the three real-world programs in our evaluation.

Intra-procedural Reachability Analysis: At this level we know which function calls invoke library functions. All their call arguments must point to shared memory later, so we first mark everything referenced by an argument of these calls as *common memory*. More formally, we mark everything that can be reached from a call argument using only (and at least one) $\xrightarrow{\text{Deref}}$ edge. We will use this mark as taint and propagate it backwards, marking all nodes that might reach a library call. If that mark reaches the memory produced by a memory allocation, this allocation needs to be rewritten. Formally, an allocation instruction x needs to return shared memory iff $\exists x' : x \xrightarrow{\text{Deref}} x' \wedge \text{marked}(x')$.

In our example in Figure 2 (and Figure 8), we first mark the dereferenced arguments of the call to `libfunc` in `lib_wrapper` with a star, which includes `err: int`, the memory referenced by parameter `buffer` and `x->two`. The struct behind pointer `x` must be shared, but detecting this requires further analysis:

To propagate the common memory mark, we follow all $\xleftarrow{\text{data}}$ and $\xleftarrow{\text{Part}_i}$ backward edges and mark every node we can reach. Formally, if $x \xrightarrow{\text{data}} y$ and y is marked, then x needs to be marked, because there is a path from x to a library call. Marks are also propagated to and from global variable nodes. We do not need to follow $\xrightarrow{\text{Deref}}$ edges here, as all possible dereferences have already been marked by the initial marking step. In our example, we need to mark the `struct X` node, given that struct member `two` needs to be shared and the struct members resides together in memory (Figure 3).

Inter-procedural Reachability Analysis: The analysis so far handles all cases where allocation and library call are in the same function. We extend the analysis first to a group of functions, then to the whole program. To cover recursion, we analyze all functions in a SCC (see Section 5.1) together at once. We resolve all `Call` instructions targeting functions within the same SCC. We connect the function's argument nodes with the parameter values from all actual calls (with a $\xrightarrow{\text{data}}$ edge), and we connect the value of the `Ret` instruction in the callee with the result value of all `Call` instructions in the callers. Once we rerun the data flow analysis, it covers data flow between all functions in this SCC, possibly over-approximating (because no call context sensitivity is given).

Analyzing functions SCC-wise is a good trade-off between full program analysis and function-wise analysis. Analyzing all functions at once is usually not feasible in acceptable time, and we cannot afford losing context sensitivity on all functions. Function-wise analysis is much faster, but cannot handle nested recursive functions. SCC-wise analysis picks the best of both worlds: For non-recursive functions, it boils down to function-wise analysis, only in case of nested recursive functions SCC-wise analysis is slower (but much more precise).

To extend SCC-wise analysis to full inter-procedural analysis, we traverse the SCC call graph bottom-up and run the SCC-wise

analysis on each SCC. Due to the properties of a SCC call graph we visit callees before callers and recursion only occurs within a SCC. That is, if we encounter a `Call` instruction, it points either to a function within the same SCC or it points to a function in an already analyzed SCC.

When analyzing a SCC, we create a *summary* for each contained function, similar to parameter trees from [24]. A summary captures all possible data flow and indirect function invocations between arguments, return value and global variables used in a function, including shared memory markers. When we later see a call to that function, we insert the precomputed summary, ignoring the full graph for the function itself.

In our example graph (Appendix C), we have built a summary edge for `update`. In `main`'s call to `update`, we copy the summary edge between `x2`/`%2` and `x3`/`%3` (and unify the dereferenced structs). We also copy the three memory markers from the `lib_wrapper` to the arguments of its call. With this information we can reason that the `malloc` call in `main` must be shared because its memory will be passed to a compartment (`buffer`).

5.6 Function Specialization

At this point our analysis cannot handle functions allocating (potentially shared) memory and return its reference. Examples are `calloc` and the `new` operator from C++. Both internally use malloc to allocate memory and return a reference to initialized memory. In our example, `new_struct` allocated memory that must (`x2`) or shouldn't (`x1`) be shared. The naïve solution (propagating the marks in both directions) would mean that *all* calls to `new_struct` would return shared memory. If that happens to `calloc`, the majority of memory used in the program might be affected—a prohibitive over-approximation.

We tackle this problem using *function specialization*, which is executed after the reachability analysis. We first determine if a function could potentially create and output memory chunks. Next, we check if any calls to this function requires these chunks to be common memory (i.e., if the function must return shared memory for some calls). If so, we clone the function. The original function is unmodified, the clone (the specialized function) will create shared memory. To this end, we traverse the SCC call graph top-down: For each SCC, we identify the memory output nodes. A memory output node is a subnode of the return value that is reachable over at least one $\xrightarrow{\text{Deref}}$ (function returns a pointer), or a subnode of an argument node reachable over multiple $\xrightarrow{\text{Deref}}$ edges (function stores a pointer in a reference-passed variable). For each call to a function in this SCC, we relate the actual call argument subnodes with the function argument subnodes and relate the call result subnodes with the function return subnodes, based on the subnode graph structure. If any of the related nodes is marked as common memory, we specialize (clone) the whole SCC. We copy all marks from all calls to the argument nodes of the specialized version and re-run the reachability analysis, forcing the function to output common memory. All calls that contributed markers are pointed to the respective specialized function.

In our example, the function `new_struct` is called once with tagged memory output (`x2` from the second call in `main`). We create a copy `specialized__new_struct` and copy two markers to the specialized function. We see that `malloc` in the specialized version must return shared memory. We thus update the second call in `main` to call `specialized__new_struct`. As the end result, just one struct in `main` is in shared memory (`x2` / `%2`).

Updating calls inside a specialized function might require further callees to be specialized. To this end, we iterate the SCC call graph in caller before callee order, including cloned SCCs.

Function specialization potentially increases the program size. To reduce the space overhead, we schedule two LLVM passes after specialization: "Dead Global Elimination" removes the old function if all calls get specialized, "Merge Functions" unifies cloned functions that have not been changed.

5.7 Tracing File Descriptors

We trace file descriptors along with memory chunks, but with much simpler rules. We use a list of known functions that return new file descriptors (`open`, `socket`, etc.). Calls to these functions are the *sources* of our data flow analysis and tagged with Fd. Fd tags propagate forward along $\xrightarrow{\text{data}}$ edges only. During the reachability analysis phase, they are copied from callees to callers. During the specialization phase, they are copied from callers to callees. Function arguments of library functions are the *sinks* of this analysis, if a library function argument is marked with Fd, it denotes a file descriptor that needs to be handled separate. This algorithm detects all file descriptors passed from the program to the library. A similar algorithm can be used to detect file descriptors that are passed from a library function to the program.

5.8 Rewriting Memory Allocations

In LLVM programs, we have three types of memory allocation: global variables, stack variables (`Alloca` instruction), and calls to memory-allocating functions. For each of these allocations we can easily check if it must be shared: If the return value node of the instruction (a pointer) has a $\xrightarrow{\text{Deref}}$ edge to a subnode marked as common memory, then the allocation must be shared. We share calls to memory-allocating functions by replacing them with their shared counterpart (`shm_malloc`, see Section 4.2). We provide these replacements for all primitive memory-allocating functions. Higher ones will be resolved using function specialization. We share global variables by moving them to a special page-aligned section in the ELF binary which will be mapped shared at runtime. We move shared stack variables to our shared heap. They are initialized and freed in the function prologue and epilogue, respectively.

6 EVALUATION

We implemented our Cali prototype in C++ and evaluated it on three sample applications. We chose these applications to cover many different aspects: Different languages (C and C++), user interfaces or console, local and networking applications, and different code sizes. Furthermore, these programs link all libraries during the compilation phase (i.e., no dynamic linking), and thus perfectly suit the link-time passes of Cali. All applications use different, widely used libraries that contained severe vulnerabilities in the past:
ImageMagick is a large (453,000 LoC) image processing tool suite written in C. ImageMagick uses *Ghostscript* to read/write postscript and PDF files, which had some serious bugs in the past [5, 40]. We

protect ImageMagick's `convert` utility, which is used to convert between file formats by isolating *Ghostscript*.

socat is an all-round utility for networking. Its C code base is rather small (29,000 LoC). Socat can establish encrypted TLS connections using the *OpenSSL* library, which had several severe vulnerabilities in the past [7].

Filezilla is a popular FTP client with an wxWidgets GUI. Its large codebase (190,500 LoC) is written in C++ and scattered over different projects. Filezilla uses *SQLite* (with critical vulnerabilities in the past [6, 8]) to manage download queues and store known servers.

We have chosen these example programs because they are widely known, use libraries with vulnerabilites in the past and cover different areas (computation, networking and user interfaces). We evaluate the functionality of Cali on more programs taken from the most popular Debian packages [1].

6.1 Correctness Evaluation

We apply Cali on each of these applications and check if the resulting binary is still fully functional. We additionally instrumented each library interface to catch more subtle bugs.

ImageMagick's functionality can be verified using the provided integration tests (that call the protected `convert` binary). After the protection with Cali, we repeated all provided tests 50 times and found no difference in behavior. Next, we added additional tests: We chose 8 popular image formats (including all formats handled by Ghostscript), prepared sample files for each format and converted each format into each other one. All converted images were identical to the ones produced by an unmodified `convert` program.

socat does not provide integration tests. We therefore combined several socat instances using different types of connections, transferred large amounts of data between them (1000 connections, up to 1 GB per connection), and verified the transfer was working and no data got changed. In detail, the socat "client" configuration reads data from a file, sends it over a TLS connection to a "server". This server is another socat configuration listening for TLS connections and using echo to send incoming data back. A third socat, our "proxy" configuration, was sitting in the middle, using a TLS server to read connections from the client and using a TLS connection to proxy incoming data to the real server. No socat configuration showed a different behavior after being protected by Cali.

Testing *Filezilla* is tricky because no official tests are provided and its GUI is hard to automate. We resorted to manual testing, using a protected Filezilla to connect to various servers and testing all the functions. We specially focused on the parts that used SQLite: the download queue and the server configurations. Again, we could not see any behavioral differences.

To evaluate Cali on an unbiased selection of binaries, we analyzed the most popular binaries taken from Debian Popularity Contest [1] (top 300 packages). We select every binary that (1) links dynamic libraries beside the standard libraries that can run with reduced privileges, (2) can be compiled with Clang/LLVM 7, (3) comes with a working set of integration tests. We rebuild these binaries with Cali enabled and use their integration tests to verify that Cali did not break anything. We additionally instrumented the library interface to reliably detect errors in memory sharing, no errors occured. We confirm that Cali works on all tested binaries, which are: dpkg-deb, dbus-daemon, man, mandb, accessdb, whatis, gpg, gpgv, gpgsm, scdaemon, xz, xzdec, fc-cache, fc-list. We isolated the most important libraries, namely libbz2, liblzma, libz, libexpat, libgdbm, libksba, libsqlite3 and libfontconfig. Each binary used up to 4 of these libraries. Depending on the quality of the provided integration tests, 35%-80% of all library call locations have been covered during these tests. Manual inspection revealed that the uncovered library callsides were mostly error handling or dead code. Additionally, some binaries had tests that did not require any library calls: sudo, gpg-agent, dirmngr, file, shared-mime-info and pstree. Cali did not break any binary.

6.2 Usability Evaluation

Cali is designed to be easily deployable in real-world systems. To apply Cali, we just need to enable link-time optimization (`-flto`) and add our linker. In most build systems, it is sufficient to add Cali using a common environment variable: `LDFLAGS="-fuse-ld=cali -Wl,--cali-config=permissions.yaml"`. Next, we specify which libraries should be separated and which privileges they should have, in a simple text-based config file (see Appendix A).

Only in exceptional cases, Cali needs additional information to handle corner cases. In our examples, ImageMagick uses a custom memory allocator instead of `malloc`. Here we need to configure which function allocates and deallocates the memory (2 lines in the configuration file, no source code changes required, see Appendix A). Socat and Filezilla did not require any annotation. Overall, integrators thus do not need to know application internals, Cali inferred all other information automatically.

This high degree of automation is a major benefit of Cali, which addresses an important aspects left open by others. To evaluate if this promised automation also holds in practice, we gave our prototype and documentation to two students: an undergraduate and a grad student, both in Computer Science. They were tasked to isolate four programs (dpkg, xz, socat, Filezilla) without further assistance. We made sure that the students did not know the program internals (source code, etc.) before handing out the tasks. They correctly isolated previously unknown programs in $\leq$ 45 min per program, and in about 32 min on average. That is, after obtaining the source code, they obtained a well-isolated compiled program in about half an hour. The vast majority of this time was spent on developing and testing a sound permission set, which can be completely automated [10, 13, 14, 42].

6.3 Compilation and Size Overhead

Next, we evaluate the compilation overhead induced by the graph analyses of Cali, as shown in Appendix B / Figure 7 for our three sample applications. We compiled every application 10 times from a clean source directory and measure the median compilation time of a pure LLVM-based build and a Cali-protected build. Cali adds 12–36% compilation time, usually just a few seconds, up to a few minutes even for large projects such as Filezilla. In times where build servers are common, this overhead does not impede wide deployment.

The size of the protected binaries increases compared to the original version. After stripping, the protected binaries are around

Table 1: Remaining shared memory allocations and the number of specialized functions in the main program

Program	Shared Mem.	Mem. Chunks (shared/all)	Specialized functions	Increased code size
ImageMagick	0.078%	28 / 36101	67 / 5395	225 KB (+ 4.7%)
Socat	0.396%	15 / 3787	3 / 748	187 KB (+ 56.2%)
Filezilla	0.255%	48 / 18798	15 / 12348	186 KB (+ 2.8%)

186 KB to 225 KB larger (see Table 1), which is mainly because of our statically linked IPC library (up to 212 KB). For typical x86 architectures, a few hundred KB are no issue, in theory we could also use a shared library.

6.4 Security Evaluation

To assess the degree of security Cali provides, we have to answer two questions: (1) Is the compartmentalization and its compartment privilege system strong enough? (2) Under which circumstances can an attacker escape from a compartment?

The compartment privilege system is strong enough to prevent any influences of an attacker beyond the runtime of the program. The PID namespace ensures all processes spawned by the library compartment are killed on program termination. Containment policies can usually either revoke file system access of libraries, or confine access to subparts only. Hence, neither sensitive data can be accessed, nor persistent backdoors can be installed. Furthermore, attackers can only leak data if network operations are allowed. We enforce strict security policies on the isolated libraries:

Ghostscript inside ImageMagick gets write access only to the folder with the input/output files (as named in the command line parameters) and the temporary directory `/tmp` (see Appendix A). Additional read-only access is granted on its installation directory and `/dev/urandom`. No network communication is permitted. This isolation is quite close to the minimal required privileges: an attacker exploiting a library vulnerability can only tamper with files in the same folder as the output file. Our permission configuration file is shown in Appendix A.

OpenSSL inside socat can only read files given in command line parameters (certificate, private key, etc.) and the randomness devices. Nothing is writeable, an attacker exploiting OpenSSL cannot trigger any permanent changes on the system. We can even block general network access for this library, because the program passes the file descriptor of an open socket to the library. The provided socket is the only network communication possibility of OpenSSL. Attackers can still access the certificate's private key (which the library must know in order to work), but have only very limited possibilities to leak it.

SQLite inside Filezilla can only access files in Filezilla's configuration folder. Network access can even be fully forbidden.

Attackers are further tightly bounded when they aim to leak information from shared memory or modify critical data structures in shared memory. Table 1 shows that Cali greatly reduces the number of memory allocations in the program that actually produce memory shared between the program and the library. Only a small fraction (< 0.4%) of all memory allocations produce chunks that are accessible to the library. Most of these memory allocations are essential to keep the functionality of the program, and thus, the information would have been passed to the library anyways.

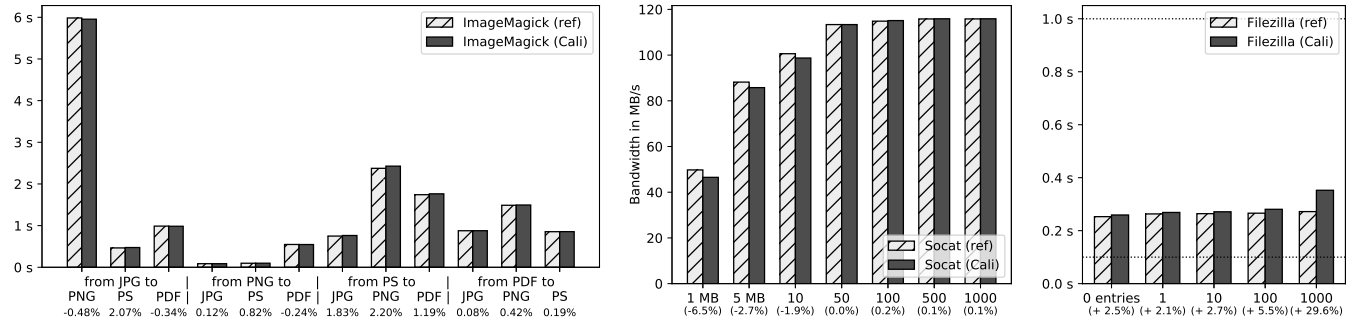
A limitation of our fully-automatic approach is the remaining risk of cross-compartment exploits: The exposed interface between privileged and unprivileged context might be vulnerable, e.g., if an attacker could store invalid values in shared memory or trigger callbacks with unexpected parameters or in an unexpected order. RLBox [30] approached this risk by proposing a restrictive C++ type system and requiring the developer to rewrite his source code to adjust to this type system. While protecting from cross-compartment exploits, this tedious work (multiple days) impedes usability, especially if users (e.g., repository maintainers) are largely unfamiliar with the source code. In contrast, Cali provides fewer security guarantees against cross-compartment exploits. Having said this, Cali already handles code pointers like callbacks as function arguments or return value (see Section 4.4), such that they cannot be abused to invoke arbitrary code execution in the privileged process. Furthermore, Cali detects and warns about function pointers in shared memory. All evaluated programs (including programs from popularity contest) have either no function pointers in shared memory, or these pointers are only called from within the unprivileged context—which is uncritical. In general, CFI can be used to harden against cross-compartment exploits, including a viable protection for C++ objects [45, 46]. Cali also provides two optional protections against double-fetch bugs (see Section 4.6). Other work [20, 37, 44, 45] already suggests protections against different other similar bugs, which are compatible to our approach. We thus see Cali on the sweet spot between usability/automation and security guarantees. Cali already minimizes the risked interface (< 0.4% of memory is shared) and detects critical function pointers in the shared area—other improvements are left open to future work.

6.5 Performance Evaluation

Wider adoptions of a protection can only happen when its runtime overhead is negligible (typically below 5%-10% [28, 39]). We evaluate if Cali is fast enough for wider deployment.

All experiments were performed on an Intel Core i7-9700K CPU (8×3.6 GHz, no hyperthreading) with 64 GB of RAM. We use Ubuntu version 18.04 LTS with an unmodified Linux 5.4 kernel. Our compiler toolchain is clang/LLVM version 7. To avoid any influences on the benchmark, we set the CPU governor to "performance" and disable "turbo" CPU power states as well as ASLR. We force each program to use only one CPU core, to prevent any unfair advantage our protected version might get, we use cpuset ot ensure this CPU core is reserved exclusively for the program. If not stated, the standard deviation of the results was below 1% of the median.

Microbenchmarks: We attribute the runtime overhead that Cali introduces mainly to two factors. First, at startup, the protected program needs to set up additional data structures and start the library compartment process. Second, every call to the library forces the kernel to switch between the two processes twice (call and return). We measured these two overheads in a minimal program as microbenchmark. Initialization at startup takes 2.2 ms on average. In these micro benchmarks, our compartment handles around 323,000



(a) Runtime of ImageMagick `convert`. The isolated libraries are called to read or write PS/PDF files.

(b) `socat` file transfer bandwidth over TLS (note: higher is better in this case)

(c) Filezilla startup time, depending on transfer queue size.

Figure 6: Performance impact of Cali. Lines in graph c): noticeable delay (100 ms) and interrupting delay (1000 ms).

calls / second (3.1 µs overhead per library call). With the mprotect-based concurrency protection enabled, our overhead is 8 µs per library call.

Application benchmarks: We now evaluate the runtime overhead of Cali on our real-world applications. We measure the runtime of ImageMagick `convert` while converting four different image formats into each other. We chose JPG, PNG, PDF and PostScript. As input files, we picked a camera picture (JPG), a website background (PNG), a test pattern (PostScript) and a sample PDF from W3C. When converting to pixel-based formats, we use a density of 300 dpi, a default value for printing. We run each conversion 100 times. Figure 6(a) shows the median conversion time. The runtime overhead is between 0% and 2.2% (geometric mean: 0.65%).

Next, we measure the network throughput of an encrypted TLS connection using `socat`, where OpenSSL is put in a compartment. Every packet is encrypted or decrypted in the isolated library, while the data is processed by the main program (resulting in ~42,000 library calls per second). We transmit files with varying sizes over a common 1 Gbps local network. The TLS server answers with the content it receives (`echo`), so we test with symmetric up- and downstream. We run each experiment 100 times and take the median throughput, i.e., file size divided by transmission time. Our network throughput showed a higher standard deviation (around 3%) for short connections (files up to 10 MB). To avoid imprecise values, we repeated the affected experiments 1000 times. Figure 6(b) shows the achieved throughput. For files up to 10 MB, the connection cannot be fully saturated, neither with protected nor unprotected `socat`. Reasons are the TLS handshake, the TCP slow start algorithm and application startup time. Cali adds ~2 ms to the application startup, resulting in a throughput degradation of 1.9% to 6.5%. For large files or long-lived connections that face the initialization overhead just once, Cali's impact on network throughput is almost negligible (less than 0.1%).

Finally, we benchmark the impact of Cali on Filezilla, the only GUI program in our test setting. The runtime performance overhead at startup is highest, since every single entry in the database is read. This leads to many library invocations, as SQLite induces one library call per table row, and for each row, one call per cell. We enqueue 0-1000 downloads in Filezilla and measure the startup time until the main window is fully displayed (`CMainFrame::OnActivate`). We repeat each experiment 500 times, as Filezilla's startup time shows a higher standard deviation (up to 20 ms). Figure 6(c) shows the median start times. Filezilla takes around 256 ms to start, and Cali adds 5 ms–80 ms depending on the queue size. With typical queue sizes up to 100 entries, the overhead is ≈3.2%. For an empty queue, the overhead is slightly higher, mainly due to the fact that there is less file I/O, and thus the initialization overhead becomes more prominent.

The user experience does not change, as the Filezilla startup response delay is already above 100 ms (lower dotted horizontal line) and thus noticeable, and even the protected case is way below the time that interrupts workflows (1000 ms, upper dotted horizontal line) according to literature [29, 31].

While it is common to choose a standardized benchmark like SPEC CPU [9] to allow for comparative evaluations, we cannot do so in our case. No program in the CPU benchmark uses third-party libraries.

7 CONCLUSION

Cali protects applications from vulnerabilities and backdoors in third-party libraries. Cali does not assume *a priori* expert knowledge of the program's source code and does not require source code changes. Its compartmentalization can be easily integrated into common build processes. Programs compiled with Cali are fully portable and do not require additional CPU features, OS modifications or superuser privileges. Isolated libraries can only access small, non-sensitive portions of the main program's memory (up to 0.4% in our examples), and only selective system access permissions remain.

Next to its main use-case, Cali can also separate different components *within* a program. Developers can use this feature to split their application into least-privileged components—fully transparently by recompiling their program with Cali. In fact, we support more source code languages than just C and C++. The underlying LLVM bytecode is independent of the source code language, and programs in other languages with LLVM frontend (Delphi, Rust, Go, Swift and many more) could be separated by Cali with minimal adaptations.

AVAILABILITY

Our prototype has been released as Open-Source Software, it is avaliable on Github:
https://github.com/cali-library-isolation/Cali-library-isolation

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their valuable feedback. Also we thank Benedikt Birtel and Leon Trampert for their help to evaluate the prototype, and Fabian Schwarz for his paper draft review.

REFERENCES

[1] Bill Allombert. 2020. Debian Popularity Contest. https://popcon.debian.org/stable/by_vote

[2] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (San Francisco, California) *(NSDI'08)*. USENIX Association, Berkeley, CA, USA, 309–322.

[3] David Brumley and Dawn Song. 2004. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13* (San Diego, CA) *(SSYM'04)*. USENIX Association, Berkeley, CA, USA, 5–5.

[4] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu. 2016. Shreds: Fine-Grained Execution Units with Private Memory. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, San Jose, CA, USA, 56–71. https://doi.org/10.1109/SP.2016.12

[5] MITRE Corporation. 2019. Artifex Ghostscript : Security Vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-10846/product_id-36469/Artifex-Ghostscript.html

[6] MITRE Corporation. 2019. CVE-2019-5018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5018

[7] MITRE Corporation. 2019. Openssl : Security Vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-217/product_id-383/Openssl-Openssl.html

[8] MITRE Corporation. 2019. Sqlite : Security Vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-9237/Sqlite.html

[9] Standard Performance Evaluation Corporation. 2017. SPEC CPU® 2017. https://www.spec.org/cpu2017/

[10] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. 2020. sysfilter: Automated System Call Filtering for Commodity Software. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 459–474.

[11] The Linux Kernel documentation. 2019. Seccomp BPF (SECure COMPuting with filters). https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html

[12] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349. https://doi.org/10.1145/24039.24041

[13] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. 2020. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 443–458. https://www.usenix.org/conference/raid2020/presentation/ghavanmnia

[14] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal System Call Specialization for Attack Surface Reduction. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1749–1766.

[15] Google. 2019. google/sandboxed-api. https://github.com/google/sandboxed-api

[16] Google. 2021. nsjail. https://nsjail.dev/

[17] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. 2015. Protecting Private Keys Against Memory Disclosure Attacks Using Hardware Transactional Memory. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, Washington, DC, USA, 3–19. https://doi.org/10.1109/SP.2015.8

[18] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. 2015. Clean Application Compartmentalization with SOAAP. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. ACM, New York, NY, USA, 1016–1031. https://doi.org/10.1145/2810103.2813611

[19] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. 2016. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. ACM, New York, NY, USA, 393–405. https://doi.org/10.1145/2976749.2978327

[20] Hong Hu, Zheng Leong Chua, Zhenkai Liang, and Prateek Saxena. 2015. Identifying Arbitrary Memory Access Vulnerabilities in Privilege-Separated Software. In *Computer Security – ESORICS 2015*. Springer International Publishing, Cham, 312–331.

[21] Douglas Kilpatrick. 2003. Privman: A Library for Partitioning Applications. In *2003 USENIX Annual Technical Conference (USENIX ATC 03)*. USENIX Association, San Antonio, TX, 273–284.

[22] Hojoon Lee, Chihyun Song, and Brent Byunghoon Kang. 2018. Lord of the x86 Rings: A Portable User Mode Privilege Separation Architecture on x86. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. ACM, New York, NY, USA, 1441–1454. https://doi.org/10.1145/3243734.3243748

[23] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-weight Contexts: An OS Abstraction for Safety and Performance. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, Berkeley, CA, USA, 49–64.

[24] Shen Liu, Gang Tan, and Trent Jaeger. 2017. PtrSplit: Supporting General Pointers in Automatic Program Partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. ACM, New York, NY, USA, 2359–2371. https://doi.org/10.1145/3133956.3134066

[25] Shen Liu, Dongrui Zeng, Yongzhe Huang, Frank Capobianco, Stephen McCamant, Trent Jaeger, and Gang Tan. 2019. Program-Mandering: Quantitative Privilege Separation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1023–1040. https://doi.org/10.1145/3319535.3354218

[26] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. ACM, New York, NY, USA, 1607–1619. https://doi.org/10.1145/2810103.2813690

[27] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. 2010. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP '10)*. IEEE Computer Society, Washington, DC, USA, 143–158.

[28] Microsoft. 2012. The BlueHat prize contest official rules. http://www.microsoft.com/security/bluehatprize/rules.aspx

[29] Robert B. Miller. 1968. Response Time in Man-computer Conversational Transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I* (San Francisco, California) *(AFIPS '68 (Fall, part I))*. ACM, New York, NY, USA, 267–277. https://doi.org/10.1145/1476589.1476628

[30] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 699–716.

[31] Jakob Nielsen. 1993. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[32] Ben Niu and Gang Tan. 2012. Enforcing User-space Privilege Separation with Declarative Architectures. In *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing* (Raleigh, North Carolina, USA) *(STC '12)*. ACM, New York, NY, USA, 9–20. https://doi.org/10.1145/2382536.2382541

[33] LLVM Project. 2020. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html

[34] Niels Provos, Markus Friedl, and Peter Honeyman. 2003. Preventing Privilege Escalation. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12* (Washington, DC) *(SSYM'03)*. USENIX Association, Berkeley, CA, USA, 16–16.

[35] Weizhong Qiang, Yong Cao, Weiqi Dai, Deqing Zou, Hai Jin, and Benxi Liu. 2017. Libsec: A Hardware Virtualization-Based Isolation for Shared Library. In *19th IEEE International Conference on High Performance Computing and Communications; 15th IEEE International Conference on Smart City; 3rd IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2017*. IEEE Computer Society, Bangkok, Thailand, 34–41. https://doi.org/10.1109/HPCC-SmartCity-DSS.2017.5

[36] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1677–1694.

[37] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. 2018. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs Using Modern CPU Features. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security* (Incheon, Republic of Korea) *(ASIACCS '18)*. Association for Computing Machinery, New York, NY, USA, 587–600. https://doi.org/10.1145/3196494.3196508

[38] Raoul Strackx, Pieter Agten, Niels Avonds, and Frank Piessens. 2015. Salus: Kernel Support for Secure Process Compartments. *ICST Trans. Security Safety* 2 (2015), e1.
[39] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, Washington, DC, USA, 48–62.
[40] Carnegie Mellon University. 2018. Ghostscript contains multiple -dSAFER sandbox bypass vulnerabilities. https://www.kb.cert.org/vuls/id/332928/
[41] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1221–1238.
[42] Z. Wan, D. Lo, X. Xia, L. Cai, and S. Li. 2017. Mining Sandboxes for Linux Containers. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, Tokyo, Japan, 92–102.
[43] Jun Wang, Xi Xiong, and Peng Liu. 2015. Between Mutual Trust and Mutual Distrust: Practical Fine-grained Privilege Separation in Multithreaded Applications. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA) *(USENIX ATC '15)*. USENIX Association, Berkeley, CA, USA, 361–373.
[44] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. 2017. How Double-Fetch Situations turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1–16.
[45] Wenhao Wang, Xiaoyang Xu, and Kevin W. Hamlen. 2017. Object Flow Integrity. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1909–1924. https://doi.org/10.1145/3133956.3133986
[46] Yu-Ping Wang, Xu-Qiang Hu, Zi-Xin Zou, Wende Tan, and Gang Tan. 2019. IVT: An Efficient Method for Sharing Subtype Polymorphic Objects. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 130 (Oct 2019), 22 pages. https://doi.org/10.1145/3360556
[47] Yongzheng Wu, Sai Sathyanarayan, Roland H. C. Yap, and Zhenkai Liang. 2012. Codejail: Application-Transparent Isolation of Libraries with Tight Program Interactions. In *Computer Security – ESORICS 2012*. Springer Berlin Heidelberg, Berlin, Heidelberg, 859–876.
[48] Yongzheng Wu, Jun Sun, Yang Liu, and Jin Song Dong. 2013. Automatically Partition Software into Least Privilege Components Using Dynamic Data Dependency Analysis. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering* (Silicon Valley, CA, USA) *(ASE'13)*. IEEE Press, Piscataway, NJ, USA, 323–333. https://doi.org/10.1109/ASE.2013.6693091

A PERMISSION CONFIGURATION

This is our configuration file for ImageMagick `convert`. The file contains all configuration we applied to the build process next to the parameters from Section 6.2.

```
---
contexts:
  main:
    selectors:
      - "*.o"
      - "libMagick*.a"
    function_behavior:
      AcquireMagickMemory: malloc
      RelinquishMagickMemory: free
  library:
    selectors:  # a list of libraries
      - "libgs.so"
    permissions:
      readonly:
        - /var/lib/ghostscript
        - /dev/urandom
      readwrite:
          # folder containing input/output files in argv
        - "$ARGV_FOLDERS"
        - /tmp
      network: none
```

Ghostscript is limited to access only the folder where the input and output files reside, `/tmp` (where ImageMagick might but additional files) and its installation directory (where color profiles etc are located). Network is not available in this compartment. User and PID namespacing is enabled by default. The selectors describe which code belongs into which context: At link time, ImageMagick consists of all `.o` files (and some static libraries). Not in a specific context (not named in any selector) is the standard library (libc), it can be called from both contexts and executes always in the calling context. As stated in Section 6.2, we need to mark ImageMagick's custom heap implementation (line 5-7).

Our prototype is able to auto-generate most parts of the necessary configuration file, for example the selectors.

B COMPILATION TIME

Figure 7 shows the time necessary to compile different programs, with and without Cali.

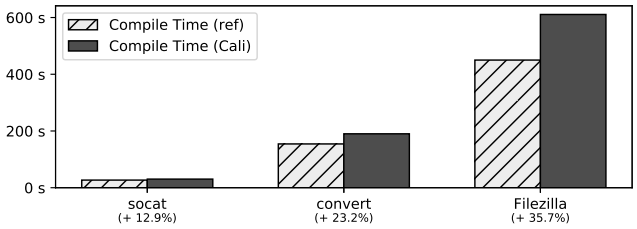


Figure 7: Compile time without and with Cali.

C PROGRAM DEPENDENCE GRAPH

Figure 8 shows the LLVM instructions of our example code in Figure 2, Figure 9 shows its full program dependence graph, after all analyses of Section 5 have been applied.

```
%struct.X = type { i64, i64 }

define void @main() {
  %1 = call %struct.X* @new_struct(i64 13)     ; x1
  %2 = call %struct.X* @new_struct(i64 37)     ; x2
  %3 = call %struct.X* @update(%struct.X* %2) ; x3
  %4 = call i8* @malloc(i64 1024)              ; buffer
  %5 = call i32 @lib_wrapper(i8* %4, %struct.X* %3)
  ret void
}

define %struct.X* @new_struct(i64) {
  %2 = call i8* @malloc(i64 16)
  %3 = bitcast i8* %2 to %struct.X*            ; s
  %4 = getelementptr %struct.X* %3, i64 0, i32 0
  store i64 %0, i64* %4, align 8
  ret %struct.X* %3
}

define %struct.X* @update(%struct.X*) {
  %2 = getelementptr %struct.X* %0, i64 0, i32 0
  store i64 18, i64* %2, align 8               ; one=18
  ret %struct.X* %0
}

define i32 @lib_wrapper(i8*, %struct.X*) {
  %3 = alloca i32, align 4                     ; err
  %5 = getelementptr %struct.X* %1, i64 0, i32 1
  call void @libfunc(i32* %3, i8* %0, i64* %5)
  %6 = load i32, i32* %3, align 4
  %7 = icmp ne i32 %6, 0                       ; err!=0
  %8 = zext i1 %7 to i32
  ret i32 %8
}
```

Figure 8: Simplified LLVM code of the example in Figure 2.

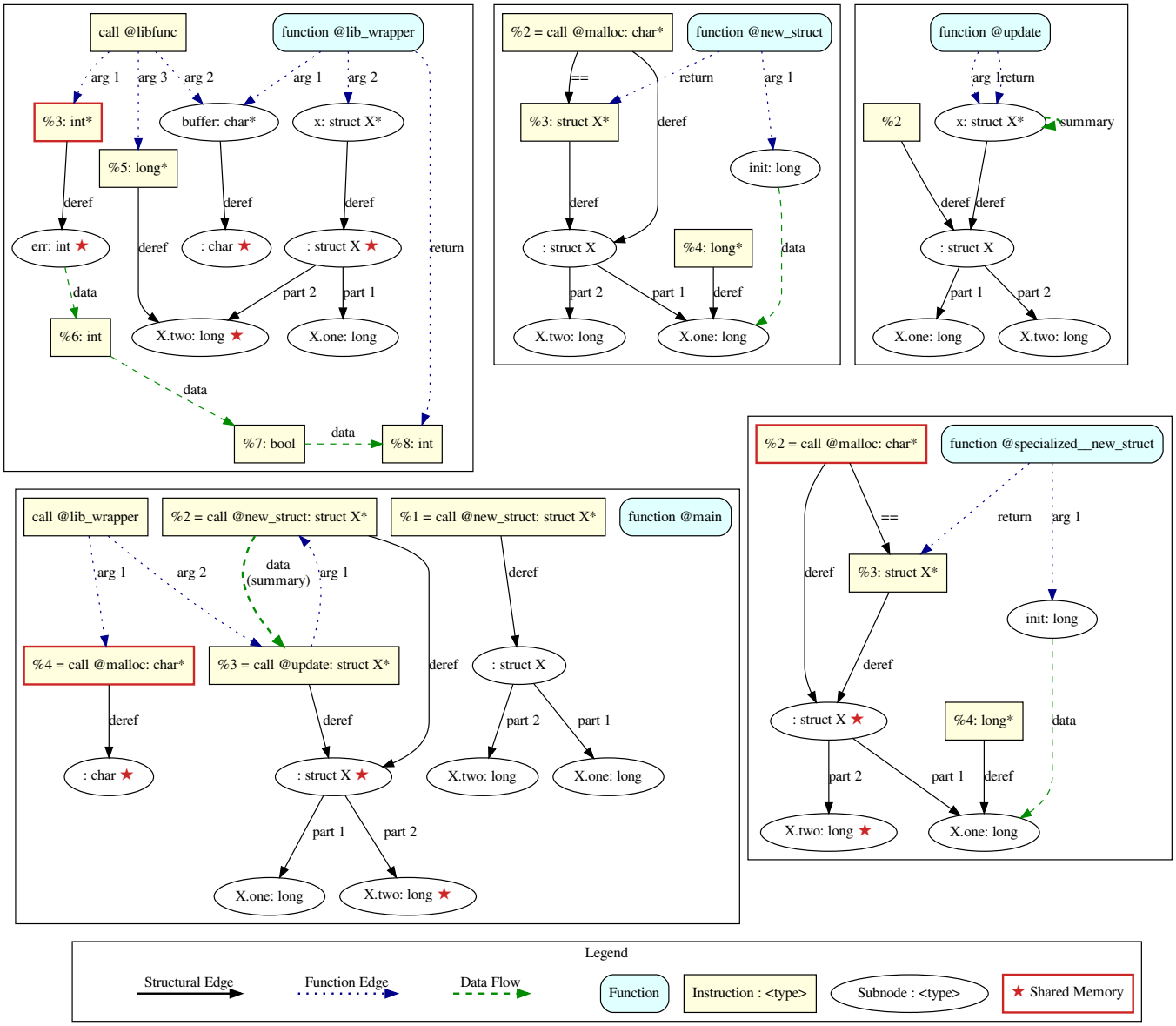


Figure 9: The full Program Dependence Graph from the example program.