Codejail: Application-transparent Isolation of Libraries with Tight Program Interactions*

Yongzheng Wu[1], Sai Sathyanarayan[2], Roland H.C. Yap[2], and Zhenkai Liang[2]

[1] Singapore University of Technology and Design
yongzheng_wu@sutd.edu.sg
[2] School of Computing, National University of Singapore
{sathya,ryap,liangzk}@comp.nus.edu.sg

Abstract. Dynamically linked libraries are commonly used in software programs to facilitate code reuse. Once a library is linked into a software program, a bug in the library can lead to compromise of the whole program. Moreover, the library may also contain malicious code. Existing solutions for software component isolation assume simple interactions between a library and the main program, otherwise, they require significant modification of the main program and the library. In this paper, we propose a novel solution, Codejail, which supports a partial isolation of libraries that have tight memory interactions with the main program. Codejail requires no modification to the main program or the library. We demonstrate using a Linux prototype that Codejail can work easily with real-world programs and libraries. The performance is good for a portable implementation with costs commensurate with the degree of tight interaction.

1 Introduction

Software today heavily relies on dynamically-linked libraries. Libraries are usually seen as a necessary step to facilitate code reuse. While the use of libraries can considerably simplify and speedup software development, there is a downside, namely, the libraries can themselves have bugs. In this paper, we distinguish between the code in specific libraries (or simply library) with the code outside the library which we call the *main program*. Once a library is linked into the main program, a bug in the library can lead to compromise of the entire program.

Specifically, there are two main threats posed by third-party libraries to its main program. First, the library may be vulnerable because it contains a bug that can be exploited by an attacker. Typical attacks are through memory corruptions, such as buffer overflow together with code injection or return-oriented programming [14]. Even though the main program contains no vulnerabilities, vulnerabilities in the library may propagate and affect the main program. Second, a library could also be malicious. The extensive use of libraries, i.e. dynamically linked libraries, only exacerbates these problems.

* This work has been supported by a DRTech grant R-394-000-054-232.

To mitigate the threats from dynamic libraries, a range of solutions have been proposed to isolate program components to control their privilege [5, 6, 16, 18]. Most solutions adopt a separated memory model: a component can only access its own memory, which is mutually exclusive with the memory of the main program. While this model works with simple inter-component interactions, it either does not work or is not efficient for libraries that engage in tight interactions with the main program. Rather, it is more common for programs and libraries to be written with tight interactions, such as sharing global variables, passing references to complex data structures, callbacks, `longjmp`, etc. It is not practical to assume that such libraries can be rewritten to eliminate the tight interaction, let alone doing so for all software using such libraries. This is even less practical for close-source libraries/programs.

Our goal is to mitigate the threats from dynamic libraries in a transparent fashion. Ideally, we want to reduce or prevent these threats without the need to modify either the libraries or the main program when there are tight interactions between them. Existing solutions, however, are not transparent to the main program and libraries. Often, significant work is needed to port a library and a main program in order to make library execution safe while preserving the functionality. This usually needs source code, thus, preventing reuse of existing binaries. We argue that while existing solutions can provide security, they do not address transparency and hence are of limited applicability.

We use NativeClient (NaCl) [18] as an example to illustrate the need for a transparent library security mechanism. NaCl is designed for isolating *untrusted native browser plugins*. It adopts a separated memory model that ensures an untrusted component can only access its own dedicated memory and code. As a result, communication between the trusted and untrusted components are in a remote-procedure-call style, i.e. parameters and return values are passed by value and data structures are serialized. However, libraries are not typically designed for this model. Instead, most libraries assume that the library and main program share the same memory and by-reference parameter passing is commonly used for efficiency. There are practical difficulties with porting any code that uses such assumptions. Good software engineering practices mean that details of data structures are encapsulated and (mostly) opaque, and thus porting requires reverse engineering the implementation. Many complex libraries employ tight interactions such as callbacks and `longjmp`, which are not allowed in NaCl because the code segment is isolated, e.g. the popular `libpng` library uses an opaque structure to keep internal states with `longjmp` as the error handling mechanism. Thus, NaCl cannot be used to isolate `libpng`, and `libpng` is used in a number of web browsers. Many security vulnerabilities have been found in `libpng`, rendering such browsers vulnerable.

In this paper, we present Codejail, a framework to isolate untrusted libraries. We assume that libraries have well defined APIs, which specify the extent of the tight interactions with the main program. This is reasonable since it is necessary for the user of a library to understand how to make use of it. To prevent bugs in a library from compromising the whole program, Codejail ensures: (i) memory

access in the library is sandboxed so that side effects are limited to conform to its API; and (ii) system privileges are controlled to be only what is allowed for its designed tasks. The design goal of Codejail is to provide application-transparent solutions for isolating libraries that have tight interaction with the main program. Unlike the separated memory model, Codejail proposes a semi-shared memory model, which allows common tight interactions while ensuring the integrity of main program's data. In this model, the jailed library has full access to its own memory and read access to other memory. In addition, the main program can selectively allow the jailed library to write to any memory. We support callbacks where the jailed library needs to run a function supplied by the main program using the main program's data and the use of `longjmp` to return to the main program from the jailed library. As we work at the binary library API level, library source code is not needed and Codejail works with dynamically linked libraries.

Although the overall goal is to support tight interactions of the main program with an untrusted library transparently, we have some restrictions. Codejail ensures that a jailed library is not able to modify arbitrary memory outside its sandbox. This restriction applies even if the library has no vulnerabilities nor is it malicious, so not every library with tight interaction can work transparently with Codejail. Nevertheless, we believe that a much larger class of libraries and software will function with Codejail than with more strongly separated memory models such as software fault isolation.

We have built a portable Unix prototype implementation of Codejail in Linux. We demonstrate the usability of Codejail to transparently sandbox well-known dynamic libraries using the off-the-shelf binaries of standard programs and libraries. Even though our prototype is portable and works in user-mode, the performance impact is still reasonable. Where there are large numbers of calls to a jailed library (or callbacks) and the library needs to write significant data outside its own memory, there will naturally be more overhead. We have tested `libpng` with the Mozilla Firefox browser and were able to protect against attacks from `libpng` to Firefox. From a performance standpoint, we did not observe any degradation in the user experience when using Firefox.

In summary, our major contribution is the design and prototype implementation of a novel approach, Codejail, that isolates untrusted libraries. Codejail supports tight program interactions required by a signification portion of libraries. It can also be applied transparently, without modifying the software program or the untrusted library.

2 Related Work

Applying the principle of least privilege by partitioning a program into a number of processes with different privilege has been studied by many researchers. Provos et al. [13] partitioned OpenSSH into two parts, a privileged master to only handle authentication and unprivileged slaves to handle the rest of the work. Kilpatrick [8] proposed Privman, a library to help partition privileged UNIX

daemons where the main program talks to a privileged server with the Privman library to perform privileged tasks. This is by replacing privileged function calls to the corresponding Privman wrappers, but a significant amount of manual work is still necessary. Brumley et al. [2] proposed Privtrans to automate the privilege separation work. The programmer manually specifies privileged data and functions. Privtrans automatically separates the program into an untrusted slave and a trusted monitor, each running in a separated process. Access to privileged data and functions only takes place in the trusted monitor. Both Privman and Privtrans adopt a trusted callee model where the main program is untrusted and the privileged operations are performed by a trusted monitor process. In contrast, Codejail addresses the opposite situation with a trusted caller and untrusted callee.

There are other solutions on confining memory access of a software component without separation into different processes. Software Fault Isolation (SFI) [16] ensures all memory accesses of the untrusted component is within the memory dedicated to the component by statically verifying direct memory access instructions and dynamically checking indirect access. Other work [4–6, 11, 18] uses the same idea while using different techniques and hardware features. Vx32 [6] uses the segment register in x86 to confine memory access in hardware. Other solutions [3, 10, 15] provide isolation by confining the untrusted component to a memory region assigned to the component, we call such a memory model as a *separated memory model*. There are two problems with this model. Firstly, existing code typically assumes global memory access and has to be recompiled or manually ported. Secondly, inter-component pass-by-reference function calls need to be changed to pass-by-value, as the callee cannot access the memory of the caller. This is not easy or efficient for complex data structures.

Wedge [1] uses tagged memory to restrict memory accesses of software components, where each memory allocation is explicitly associated with a tag. Software is partitioned into least-privilege components, which can only access memory with specific set of tags. Compared to the separated memory model, Wedge allows memory sharing, such as by-reference function parameters, if the memory regions have compatible tags. However, it needs each component to understand how memory is used by other components. This requires understanding the memory access behavior of all components in the software and memory allocation has to be modified to specify the correct tag. When modification is not possible, Wedge provides a way to specify the default tag for all allocations made by a component. This can lead to the confused deputy problem. Consider a malicious component C_1 and a benign component C_2 both using component C_3. C_2 uses C_3 to allocate memory to store critical data. C_1 may be able to tamper with C_2's data by using C_3.

3 Problem Statement

3.1 Motivating Example

We use the `libpng` library to demonstrate the problem of tight interactions between a main program and a library. It shows the difficulty of supporting such interactions in the separated memory model. Fig. 1 shows a sample main program using `libpng`. We underline the key points in the listing, and we emphasize whether `main` or `libpng` manage the memory of particular data structures, as well as who uses it.

The main data structure for `libpng` is an opaque structure `png_struct` pointed by `png`. It is created at Line 7 with `png_create_read_struct` (similarly, `info` at Line 8) – memory allocated by `libpng` is used in `main`. This structures pointed by `png` and `info` can be thought of as identifying the interaction between `main` and `libpng` but as they are not directly accessed by the main program, the details should be considered as private and implementation specific. If `libpng` is sandboxed using a separated memory model, parameter marshalling of `png_struct` will break the separation between interface and implementation. Rather than marshalling, `png` can be treated as a resource handle rather than pointer. However, this can crash the main program if it tries to dereference it. `png_destroy_read_struct` at Line 10 & 22 is used to free the opaque structure as well as resetting the pointer to NULL – `libpng` changes `png` in `main`.

Due to lack of language-based exception handling in C, `setjmp` and `longjmp` are often used in libraries including `libpng`. At Line 9, `setjmp` is used to create the error handling code in `main`, the `jmpbuf` comes from memory managed by `libpng`. Such library code does not fit with the separated memory model, e.g. the `longjmp` branches outside the allowed code range and stack frame in NaCl.

The function `main` reads the PNG file in a loop. It passes chunks of PNG data to `libpng` using `png_process_data` at Line 19 – `libpng` reads `buff` managed by `main`. The main program also passes the function pointer `row_callback` to `libpng` at Line 14 – the function resides in `main`. In `png_process_data`, `row_callback` is called by `libpng` whenever a row of pixels is decoded. The main program then displays the row through its `row_callback` function. This mechanism is known as *function callback*, where the main program registers a function pointer in the library, which will be called by the library. The callback mechanism will cause a similar problem as `longjmp` in separated memory model approaches.

3.2 Tight Interactions

We now examine the typical interactions between the main program and a library, including those that are challenging to support under the separated memory model, such as the side effects of library functions and function callbacks:

1. **By-Value Parameter Passing and Return:** The parameters are copied from the caller to the callee and vice versa for return values, e.g. `sqrt()`.

```
1  static void row_callback(png_struct *png, png_bytep new_row,
2      png_uint_32 row_num, int pass) {
3    // display the row
4  }
5  int main (void) {
6    FILE *fp = fopen("foo.png", "rb");
7    png_struct *png = png_create_read_struct(...);
8    png_info *info = png_create_info_struct(png);
9    if (setjmp(png_jmpbuf(png))) {
10     png_destroy_read_struct(&png, &info, NULL);
11     close(fp);
12     return 1;
13   }
14   png_set_progressive_read_fn(ptr, ..., row_callback, ...);
15   while (1) {
16     char buff[1024];
17     size_t len = fread(buff, 1, 1024, fp);
18     if (!len) break;
19     png_process_data(png, info, buff, len);
20   }
21   fclose(fp);
22   png_destroy_read_struct(&png, &info, NULL);
23   return 0;
24 }
```

Fig. 1: Using `libpng` to read a PNG file

2. **By-Reference Parameter Passing and Return:** The caller passes pointers of the parameters, and memory is dereferenced by the callee and possibly modified, e.g. `strlen()` and `asctime_r()`.
3. **Global Variable:** Some libraries export global variables that can be directly accessed by the main program or other libraries, e.g. `errno` from `libc`.
4. **Function Callback:** Library functions may need to call the main program in order to read/write data or signal task completion, e.g. `png_process_data()` makes a callback as described in Sec. 3.1.
5. **Long Jump:** Some libraries, e.g. `libpng`, use `setjmp`/`longjmp` as an error-handling mechanism. This can cause the library to transfer control to the main program without using the return mechanism.

The first type of interaction involves no tight memory interactions, which can be easily supported by memory isolation models. However, the other types of interactions are not compatible with memory isolation models. They either have implicit shared memory operands, or involve non-standard control transfer between code of the main program and the library.

3.3 Threat Model and Design Goal

Threat model. In our approach, we aim to mitigate the untrusted library's threat to directly access undesired system resources or memory. Note that the untrusted library can cause indirect threats through the data it returned to the main program, such as returning malicious data to exploit memory errors in the main program. This type of threats is out of the scope of our solution (and the related solutions in Section 2). It can be addressed by the main program through data sanitizing, checking returned data (including updates of by-reference parameters) from the untrusted library.

Design goal. Under this threat model, an untrusted library must be separated from the main program to prevent it directly accessing the main program's resources and memory. However, the library needs tight interactions with the main program. The goal of Codejail is to isolate untrusted libraries into different execution contexts. The contexts share a flexible memory model which supports close interactions. Specifically, our solution guarantees the following properties:

- The untrusted library cannot execute arbitrary code in the trusted context.
- The untrusted library cannot crash the main process through, for example, null pointer dereference, illegal memory access and deadlock.
- The untrusted library cannot make arbitrary system calls, for example, only system calls explicitly specified by the main process are allowed. The specification can include a set of allowed system calls, a set of files and directories and system resource limits such as memory usage, time limits, etc.

4 The Codejail Approach

We describe Codejail's key techniques showing how they meet our design goals. An untrusted library is typically used in the following fashion. The software consists of a trusted main program, an untrusted library, and a trusted library. The main program uses functions from both libraries; the untrusted library uses the trusted library. The main program only interacts with the untrusted library through the library's API, which specifies functions exported from the libraries with their parameter types and calling conventions. For by-reference parameters, the API should specify whether the callee updates the parameter. The API also specifies exported global variables and their data types. For data types that are directly accessed externally, their data structures have to be specified, e.g., in a header file. We do not assume availability of source code of the main program or the libraries. However, we assume the header file of the library to be available. We have no assumption about the binary of the untrusted library, i.e. it can contain arbitrary code including indirect branches and system calls.

4.1 Codejail Overview

Fig. 2 gives an overview of Codejail. Codejail creates contexts to separate the main program and the untrusted library. The main program runs in the trusted

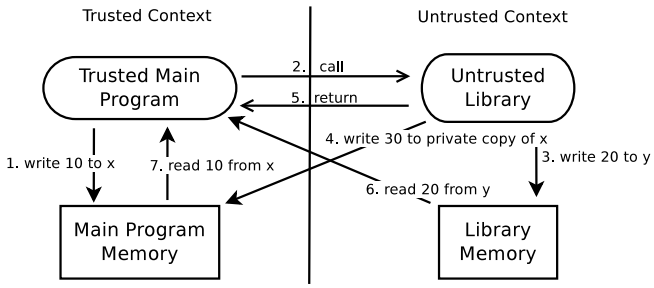


Fig. 2: An overview of Codejail

context, while the untrusted library runs in the untrusted context. When the main program calls (Step 2) a function in the untrusted library, Codejail switches execution to the untrusted context. When the untrusted library function returns (Step 5) to the main program, execution switches back to trusted context. Calling functions in the trusted library does not change the context. Calling any function, including functions in the main program, in the untrusted context remains in the untrusted context. However, callbacks can be set up so that the untrusted library can call them to switch from untrusted to trusted context. The untrusted context is unprivileged, and its system resource accesses are sandboxed.

Memory access in the untrusted context is sandboxed. It can write to its own memory. However, for writes to the main program's memory, changes are only observable in the untrusted context but not in the trusted context. In Fig. 2 Step 3, the untrusted library assigns 20 to variable y, which is in its memory. The main program observes this change in Step 6. However, in Step 4, the assignment of 30 to x, which is in the main program, is not observable by the main program in Step 7, where the old value 10 is read. The main program can selectively "commit" changes made in untrusted context, so that it sees the value 30 in x. Other changes are lost in both contexts.

4.2 Memory Access Policies

Codejail classifies writable memory into three types: M_m, memory of the main program; M_j, memory of the untrusted library; and M_l, memory of the trusted library. The semantics of memory access depends on context and memory type.

Table 1 shows the classification of writable memory types. The static allocated memory typically includes global variables and static local variables. In the ELF binary format, such variables reside in the `.bss` and initialized data segment. The static allocated memory is associated with the code that declares it. In Codejail, we further divided it into three parts: the main program, trusted libraries, and untrusted libraries. Codejail divides the stack into two parts: one used by the trusted context (including the main program and trusted libraries) and the other used by untrusted context. Similarly, the heap is divided into two – different contexts allocate memory in different heaps.

Memory		Type
static allocated memory	main program	M_m
	untrusted library	M_j
	trusted library	M_l
stack	trusted context	M_m
	untrusted context	M_j
heap	trusted context	M_m
	untrusted context	M_j

Table 1: Memory types in Codejail.

We now describe the policies for memory access on the three types of memory:

- M_m **Memory of Main Program:** Initially, M_m memory in both contexts is synchronized, i.e. memory read gives the same value for the same address. After the main context writes to it, the memory is still synchronized. However, after the untrusted context writes to it, the memory is not synchronized. Each context has a different view of the memory, so that the memory writes are only observable in its own context. As a result, after the untrusted library function returns, the main program is not able to observe the change made by the untrusted library. In this way, we prevent the untrusted library from corrupting data of the main program.
  In some cases, we want the library to update some data. For example, we want `memcpy(dest, src, n)` to update `[dest, dest+n]`. Codejail provides the API (`cj_recv(void *ptr, size_t size)`) to copy data from the untrusted context to the trusted context.
  Before each untrusted function call, M_m memory is re-synchronized to the synchronized state, and changes made by the previous untrusted function, unless committed through `cj_recv`, are discarded.
- M_j **Memory of Untrusted Library:** Both contexts can observe the updates made by each other. As a result, the main program should sanitize it before using its value.
- M_l **Memory of Trusted Library:** Initially M_l memory in both contexts is synchronized. However, memory writes made by either context cause it to be not synchronized, thus the other context is not able to observe the changes. M_l memory is never re-synchronized. For example, if we consider `libc` as the trusted library, the random seed used by `rand()` is in M_l. Thus a different copy of the random seed is kept in each context. The untrusted context cannot modify the trusted context's seed.

4.3 Supporting Tight Interactions

Using the memory types and access policies, Codejail supports tight interactions between the trusted context and the untrusted context.

Pass-by-reference: The main program passes pointer p to the untrusted library. Memory pointed by p can be allocated in trusted context (M_m) or untrusted context (M_j). In Line 10 & 22 of Fig. 1, `&png` points to memory in the

```
png_uint_32 wrapper_png_get_text (png_structp png_ptr,
  png_infop info_ptr, png_textp *text_ptr, int *num_text)
{
  png_uint_32 retval = (png_uint_32) cj_jail(
    real_png_get_text, 4, png_ptr, info_ptr, text_ptr, num_text);
  cj_recv(text_ptr, sizeof(png_textp));
  cj_recv(num_text, sizeof(int));
  return retval;
}
```

Fig. 3: An wrapper function for `png_get_text()` in `libpng`

main program's stack, i.e. in M_m. The main program has to call `cj_recv` to "commit" the change made in `libpng`. In Line 19, `png` points to memory allocated in `libpng`, i.e. in M_j. The main program does not need to call `cj_recv` in this case.

Global Variables of the Untrusted Library: The memory model of Codejail allows the main program to transparently read and write global variables exported by the untrusted library. This is because they are in M_j, which allows both contexts to observe changes made by each other.

Callback: Codejail allows the untrusted library to call the main program's functions in a trusted context. To do this, the main program calls Codejail API `cj_reg_callback` to register a callback function. A function pointer f is passed to `cj_reg_callback` and another function pointer f' is returned. f' is then passed to the untrusted library. When the untrusted library calls f' in the untrusted context, Codejail will switch to trusted context and call f. After f returns, execution switches back to untrusted context. Both call and return are transparently handled by Codejail. Callbacks can be nested recursively, i.e. during a callback, the main program can call untrusted library functions, which then make more callbacks.

Long Jump: Codejail allows long jumps between different contexts. To prevent the untrusted library from jumping to arbitrary code in the trusted context, Codejail ensures all long jumps are using a `jmp_buf` registered with `setjmp`.

We handle typical cases of tight interactions. However, there are cases that Codejail does not support. For example, memory allocation in the trusted context is freed in the untrusted context, which is not allowed to protect M_m. When the interface between the library and the main program is not well defined, Codejail cannot support the interaction. This includes undocumented memory write by the library, undocumented library functions called by the main program, and passing opaque data structures to the library. Such not well defined interactions are less common being not good software engineering practices.

4.4 Codejail Primitives

There are two ways to apply Codejail. One is to write a wrapper library exporting the same set of functions as the untrusted library. The wrapper library calls the

Codejail API. In this way, we do not need to modify the main program. The wrapper library is reusable on any program that uses the untrusted library.

Fig. 3 shows an example of the wrapper function for `png_get_text()`, which passes back a number of strings to the caller. `real_png_get_text` is the real `libpng` function. A string pointer and integer are allocated in the trusted context (M_m). `cj_recv` is necessary to pass them from the untrusted context. The actual string is allocated in untrusted context (M_j) but it is not necessary to call `cj_recv` to pass the string. In this example, assuming 32-bits, 20 bytes (one function address and 4 parameters) are passed before the jailed call. 12 bytes (one return value and 2 output parameters) are passed after the jailed call.

The second way is where a general wrapper library is not feasible, or some assumptions in the main program can make Codejail to be more efficient. In these cases, we can modify the main program, to call Codejail API directly.

We list the Codejail API functions which are called from the trusted context:

- `void *cj_jail (void *func, int argc, ...)`
  It switches the context from trusted to untrusted and calls function `func` with `argc` number of integer type[3] arguments and return value.
- `void cj_recv (void *data, size_t size)`
  It synchronizes M_m and M_l memory from the untrusted context to the trusted context. Note that only one address is specified, because the address space layout is the same in both contexts.
- `void *cj_reg_callback (void *mainfunc, int argc)`
  It takes a function pointer in the main program and returns another function pointer which can be called from the untrusted context. When it is called, the context is switched from untrusted to trusted.
- `void *cj_jail_func (void *libfunc, int argc)`
  It takes a function pointer in the untrusted library and returns another function pointer which can be called from the trusted context. When it's called, context is switched from trusted to untrusted.
- `FILE *cj_duplicate_file (FILE *fp)`
  It takes a file pointer opened in the trusted context and returns a *shadow file* pointer to be used in the untrusted context. The purpose of this function is to allow passing `FILE` pointers from the trusted context to untrusted context without understanding the internal data structure of the `FILE` structure. The untrusted context is able to operate on the shadow file which points to the same underling file. The limitation is that the file will be corrupted if both contexts operate on the file after calling this function, because the file pointer in both `FILE` structures will be out of sync.

4.5 Security Analysis

Although Codejail's design achieves the functionality requirement, attackers may launch attacks targeting Codejail. We consider the following potential attacks.

[3] For simplicity, this notation assumes integer type arguments which include `char`, `short`, `int`, `long` and pointer types but can be extended in a straightforward way.

- **Denial-of-Service**: The untrusted library can refuse to perform its expected function by, for example, infinite looping, infinite memory allocation, segmentation fault. This can be caught by timeout or signal handler and handled appropriately.
- **Return-to-Libc, Return oriented attacks**: Some library APIs allow the libraries to pass function pointers in returned data structure and the main process will call the pointed function. A potential attack is for the untrusted library to pass a pointer to a malicious function and hope the function is invoked in the trusted context. This is prevented by not allowing the library's code to be executed in the trusted context. In case the main program intends to call the function, it can call the Codejail API `cj_jail_func` described in Sec 4.4 to wrap the function pointer.

  Another attack is to return pointer to code in the main program or trusted library, similar to return-to-libc attack and return-oriented-programming. This is prevented by wrapping all function pointers.
- **Abusing system privileges**: We assume system privileges requested by the untrusted library to be examined by either the programmer of the main program or system administrator, depending on how Codejail is applied.

5 Implementation

Our prototype Codejail is implemented portably in Linux in user mode. We now discuss the implementation choices and challenges.

Context for isolating libraries. We choose process as the basic mechanism for implementing the two contexts, a main process for trusted context and a jailed process for untrusted context. Communication across contexts is supported by sending and receiving data through a UNIX socket. In the jailed process, we use etrace [7] as a portable user-mode system call interposition mechanism. (However, kernel-based system call sandboxing mechanisms, e.g. Systrace [12], can be used). When etrace finds a suspicious system call, it sends a signal to the Codejail process. Now, the Codejail process can abort the execution and safely pass the control back to the main program. Thus, Codejail is effective against memory corruption and arbitrary code execution attacks as well as side effects from system calls.

Memory sharing across contexts. Memory of M_m and M_j is shared between two processes using the standard `shm_open` and `mmap` API. Codejail creates virtual files to be mapped into both processes. M_j memory is mapped as `MAP_SHARED` in both processes, so that memory writes can be observed by each other. M_m memory is mapped as `MAP_SHARED` in the main process and `MAP_PRIVATE` in jailed process, so that the main process cannot observe jailed process' memory writes. Re-synchronization of M_m memory is done by re-mapping (`munmap` and `mmap`) it in the jailed process.

Codejail hooks the memory allocation routines in order to control memory allocation in the M_m for the main process and M_j for the jailed process. This

works for most programs using the standard library memory allocator or custom allocators which call the standard allocator.[4]

Since Codejail has to maintain a symmetric memory address layout, `mmap` performed in one process has to be performed in the other. For readonly `mmap`, we can simply call `mmap` with the same parameters in the other process. However, writable `mmap` has to be handled properly in order to ensure our memory model. The rule is that `mmap` performed by the main process should be in M_m; while `mmap` by jailed process should be in M_j. For writable anonymous `mmap`, we can consider it as a heap allocation. For writable file-backed and `MAP_PRIVATE` `mmap`, we can allocate on the heap and read in the file. However, this is inefficient as it breaks the sole purpose of `mmap`, which is not to read the whole file.

Implementation of Codejail primitives. The initialization of Codejail is performed after dynamic linking and before calling `main`. It is implemented transparently by hooking `__libc_start_main`. Codejail forks a new process and setups the shared memory and the communication channel. After that, the jailed process reads and waits for a message from the socket.

When the main process calls `cj_jail`, the target function address and arguments are send through the UNIX socket. The jailed process receives them and calls the target function. After the function returns, jailed process sends the return value to the main process through the socket. Callbacks are handled similarly in the reverse direction. To prevent the jailed process from invoking arbitrary internal functions in the main process, a callback table is used (similar to the jump table in related work in control flow integrity [9, 17, 19]). When the main process calls `cj_recv`, the address and size are sent and the memory is received. At the end of the main program, main process sends a termination message and the jailed process exits.

Codejail supports multi-threaded program, but for simplicity we assume only one thread uses the untrusted library at a time. We have one thread in the jailed process servicing multiple threads in the main process. A *pthread* mutex prevents multiple simultaneous library calls.

Application-transparency support. To transparently support existing program and libraries, we use a wrapper library exporting the same functions as the isolated library. The `LD_PRELOAD` environment variable "injects" the wrapper library into a program so that it transparently calls our wrapper functions instead of the real untrusted library functions.[5] Functions in the wrapper library identify the original functions using `dlsym(RTLD_NEXT, name)`, and call the original functions using `cj_jail` then `cj_recv` to receive data from the jailed library when necessary.

[4] Some programs use completely custom allocators, e.g. Firefox uses *jemalloc*. In Codejail, the allocated memory will not be shared, but only valid in the allocating process. If the process passes the memory to the other process, it will cause a segmentation fault when it is accessed. We get around this by re-building Firefox and disabling *jemalloc* in the build configuration.

[5] If dynamic loading with `dlopen()` is used, our wrapper library will be opened when a relative path is used. If an absolute path is used, which is uncommon, the original library will be opened, calling its API has an exception as it is not executable in the main context.

Name	Callback	Shadow File	Modify main's Memory	Pass-by-Reference	longjmp
libpng	Yes	Yes	Yes	Yes	Yes
libexpat	Yes	No	No	Yes	No
libbzip	No	Yes	Yes	Yes	No
libtiff	No	Yes	Yes	Yes	Yes

Table 2: Libraries used in evaluation and their types of program interactions.

Attacks targeting the implementation. When the attackers are aware of the Codejail implementation mechanism, they may launch attacks targeting the implementation. We discuss possible attacks and the defenses.

- *Attacking Codejail's internal states*: In the jailed process, the Codejail's internal routines such as heap allocator, signal handler, and RPC handler, execute in the same memory and privilege state as the jailed library routines. Thus, we consider the Codejail's internal routines only as helper routines rather than trusted routines. The security guarantees are not based on the correctness of these routines, thus attacking the internal routines and states does not break the guarantees.
- *Denial-of-service attacks*: Infinite loops and memory allocation can be dealt with by `setrlimit`, causing an exception in the jailed process. It can then caught by a timeout set in the main process.
- *Controlling the main program using ptrace, /proc/[pid]/mem*: The jail process can use system mechanisms such as `ptrace` and `/proc` interface to modify the main process' memory. This is prevented by the system call policy.
- *Library constructor*: The separation into two processes takes place after library loading and before calling `main()`. Before the separation, the process run with full privilege, thus a malicious library can call system calls in library constructor, which is called before `main()`. One way to prevent this is to delay the library constructor and call it after the separation.

6 Evaluation

We have evaluated the Codejail prototype using a number of real-world programs using complex real world libraries. The experiments are run on an Intel Core 2 Duo 2.80GHz processor with 4GB of RAM in Linux 2.6.35.

We evaluate the following libraries which exhibit a full range of tight program interactions: `libpng` (1.4.2) provides handling of Portable Network Graphics (PNG) images; `libtiff` (3.8.2) provides support for Tag Image File Format (TIFF) images; `libexpat` (2.0.1) is a XML parser library; and `libbzip2` (1.0.4) provides a general purpose compressor/decompressor. The types of program interactions used by the above libraries are shown in Table 2. For example, `libtiff`

does not use callbacks, while `libpng` and `libexpat` do. We also chose these libraries for their popularity and because the particular versions have the following known vulnerabilities: `CVE-2010-1205`, `CVE2009-3720`, `CVE-2008-1372` and `CVE-2010-3087`.

6.1 Functionality and Usability

We wrote wrappers for all the libraries and evaluated them on the command line utilities listed in Fig. 4. We tested transparency by using the wrappers with executables of each program together with corresponding DLL binary. In all cases, we could transparently deploy Codejail for the program and library with the same functional behavior.

In addition, for `libpng`, which exhibits the full range of close interaction in Table 2, we tested with several GUI programs that display PNGs using `libpng`, namely, the `eog` image viewer, the Mozilla `firefox` web browser and the `xfig` and `dia` graphics editors. All these programs are multi-threaded. The programs all worked and displayed images correctly with Codejail. As these are GUI programs, we did not measure performance, rather their overall usability. We did not find any perceptible delays or other differences in the usage.[6]

As Firefox is a complex program, we show some details of how we jail `libpng` in Firefox (3.6.4). This version of Firefox is selected as having modern features but still being single process (due to the existing restrictions of the prototype). Normally, Firefox includes its own `malloc` library, `jemalloc`, and a special version of `libpng`. This is because it supports the Animated Portable Network Graphics (APNG) file format which is an unoffical extension to PNG and thus not supported by `libpng`. However, there is also a patch available to support APNG files with the standard `libpng` library. While it is feasible to hook the internal Firefox code and redirect them to the appropriate wrappers, we want use Codejail to jail untrusted dynamically linked libraries. Thus, we simply recompiled Firefox, to not use `jemalloc` and its own internal `libpng` code so that Codejail can use the APNG patched `libpng` DLL.

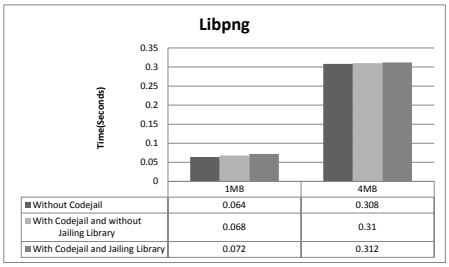
6.2 Performance Evaluation

Fig 4 shows benchmarking Codejail on four command-line programs using the libraries. For each library, we used two input files of different sizes. We measured execution time without Codejail; with Codejail but not jailing any library functions (to see the impact of Codejail on the main program and trusted library); and jailing all library functions. In all the test cases, using Codejail without jailing the library has small overhead.

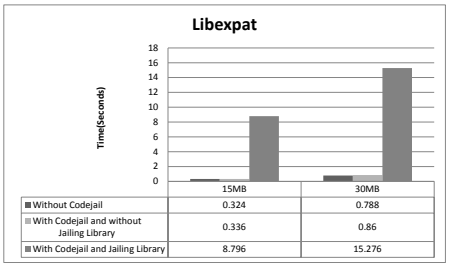
The `libpng` library, although the most complex, has the best performance. The overhead is low as there are only 15 jailed function calls and no callbacks are used by `pngtopnm` as it uses file I/O style of using `libpng` rather than the display function style.[7] The pixel buffer is allocated by `libpng` thus not copied.

[6] The figures in the paper are drawn using `dia` with Codejail.

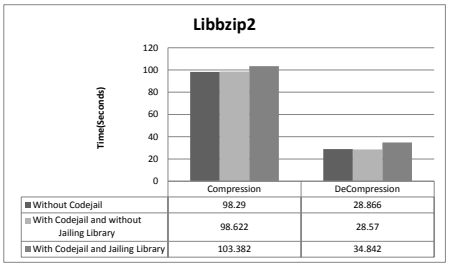
[7] However, `eog` and `firefox` use callbacks with `libpng`.



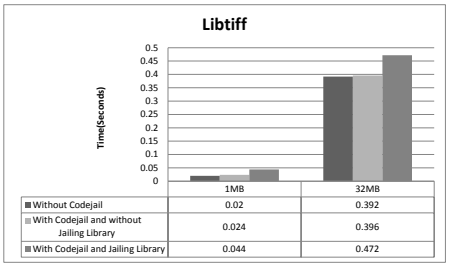
(a) `pngtopnm` using `libpng`



(b) `xmlwf` using `libexpat`



(c) `bzip2` using `libbzip2` (300MB decompressed)



(d) `tifftopnm` using `libtiff`

Fig. 4: Performance evaluation: (i) native; (ii) program with Codejail but the library is not jailed; and (iii) program with Codejail and library being jailed

On the other end of the scale is, `xmlwf` with `libexpat`. It has large overheads due to the large number[8] of callbacks made. The callbacks have more overhead as they incur context switches.

`bzip2` using `libbzip2` shows the advantage of the Codejail memory model where the jail can access the `main` memory. In `bzip2`, the main program handles I/O to do with decompressed file while the library handles I/O of the compressed file. When compressing, the original data can be accessed directly in the jail which handles the compression and writing of the compressed file, thus, the overheads are small ($\sim 5\%$). Decompression has a higher overhead as the main program allocates the output buffer and ask the library to fill it. Thus, `cj_recv` is needed to copy the buffer. In total 298.9MB is copied, which is about the same size of the decompressed file.

The overheads for `tifftopnm` with `libtiff` are due to transferring the pixel buffer processed by the jail process to the main process using `cj_recv`. For a 1MB TIFF file, 1.1MB is copied using `cj_recv`; and 34.3MB for a 32MB file.

[8] It registers callbacks which are called for all XML nodes. 425931 callbacks for a 15MB XML and 960961 for 30MB.

7 Discussion

We discuss limitations and other issues of Codejail and its implementation.

Handling the `fork` *system call.* `fork` needs to be treated specially. Firstly, both the main process and the jailed process have to be forked. They have to remain in the same state of execution. Secondly, the shared memory and communication socket has to be duplicated. In particular, M_m and M_j should be duplicated into M'_m and M'_j. Modification in M_m should not be reflected in M'_m. For simplicity, our current implementation does not support fork. However, we support multi-threading which does not need to handle new jail processes.

Inline function and macros. Inline library functions and macros cause binary code to be generated in the main program which then execute in the trusted context. The simplest solution is to ensure that they stay in the library by turning them into functions and recompiling. This is not totally transparent but is easy to do with source code. Of course, if these functions do not have any security issues, then nothing needs to be done.

Efficient memory sharing. The `cj_recv` function is implemented by reading memory in the jailed process, sending it through socket, and writing it to main process' memory. We remark that there are further optimizations which are possible but are not easily doable in a user-mode implementation. A kernel-based implementation only needs to update memory if it has been changed in the jail and copying costs is also more efficient than socket IPC.

Multiple untrusted libraries. Codejail can be extended easily to have multiple untrusted contexts for multiple untrusted libraries if they do not interact with each other, i.e., they only directly interact with the main program. Otherwise, it is simpler to place them in the same untrusted context.

8 Conclusion

We presented Codejail, a novel solution that achieves partial isolation of untrusted libraries that require tight interaction with the main program. Codejail transparently supports existing software and library binaries, working without the need to rebuild them. The key techniques of Codejail is to use a separate context to confine untrusted libraries with the Codejail memory model. Our Linux prototype shows that Codejail works with real-world programs and libraries and overheads are small except when there is excessive tight interactions.

References

1. A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: splitting applications into reduced-privilege compartments. In *Proc. of NSDI*, 2008.
2. D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proc. of the USENIX Security Symp.*, 2004.
3. M. Castro, M. Costa, J.P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *Proc. of ACM SOSP*, 2009.

4. J.R. Douceur, J. Elson, J. Howell, and J.R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proc. of OSDI*, 2008.
5. Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G.C. Necula. Xfi: Software guards for system address spaces. In *Proc. of OSDI*, 2006.
6. B. Ford and R. Cox. Vx32: Lightweight user-level sandboxing on the x86. In *Proc. of USENIX Annual Technical Conf.*, 2008.
7. K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. *In Proc. of NDSS*, 2000.
8. D. Kilpatrick. Privman: A library for partitioning applications. In *Proc. of the USENIX Annual Technical Conf., FREENIX track*, 2003.
9. R. Kumar, A. Singhania, A. Castner, E. Kohler, and M. Srivastava. A system for coarse grained memory protection in tiny embedded processors. In *Proc. of DAC*, 2007.
10. Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M.F. Kaashoek. Software fault isolation with api integrity and multi-principal modules. In *Proc. of ACM SOSP*, 2011.
11. S. McCamant and G. Morrisett. Evaluating sfi for a cisc architecture. In *Proc. of the USENIX Security Symp.*, 2006.
12. N. Provos. Improving host security with system call policies. In *Proc. of the USENIX Security Symp.*, 2003.
13. N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Proc. of the USENIX Security Symp.*, 2003.
14. Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Info. & System Security*, 2012.
15. M.M. Swift, M. Annamalai, B.N. Bershad, and H.M. Levy. Recovering device drivers. *ACM Trans. on Computer Systems*, 2006.
16. R. Wahbe, S. Lucco, T.E. Anderson, and S.L. Graham. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review*, 1994.
17. Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proc. of IEEE S&P*, 2010.
18. B. Yee, D. Sehr, G. Dardyk, J.B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proc. of IEEE S&P*, 2009.
19. B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proc. of ACM CCS*, 2011.