Sandcrust: Automatic Sandboxing of Unsafe Components in Rust

Benjamin Lamowski
benjamin@lamowski.net

Carsten Weinhold, Adam Lackorzynski,
Hermann Härtig
Department of Computer Science
TU Dresden, Germany
{firstname}.{lastname}@tu-dresden.de

Abstract

System-level development has been dominated by traditional programming languages such as C and C++ for decades. These languages are inherently unsafe regarding memory management. Even experienced developers make mistakes that open up security holes or compromise the safety properties of software. The Rust programming language is targeted at the systems domain and aims to eliminate memory-related programming errors by enforcing a strict memory model at the language and compiler level. Unfortunately, these compile-time guarantees no longer hold when a Rust program is linked against a library written in unsafe C, which is commonly required for functionality where an implementation in Rust is not yet available.

In this paper, we present Sandcrust, an easy-to-use sandboxing solution for isolating code and data of a C library in a separate process. This isolation protects the Rust-based main program from any memory corruption caused by bugs in the unsafe library, which would otherwise invalidate the memory safety guarantees of Rust. Sandcrust is based on the Rust macro system and requires no modification to the compiler or runtime, but only straightforward annotation of functions that call the library's API.

CCS Concepts • **Software and its engineering → Software reliability**; **Software safety**; *Reusability*;

1 Introduction

Traditional systems programming languages such as C and C++ cannot enforce memory safety and control-flow integrity during program execution. As a result, programs written in these languages are often vulnerable to buffer overflows, use-after-free bugs, or dangling pointers, all of which may compromise security and safety properties of a system. In contrast, modern programming languages like Rust provide a strict memory model that is enforced at compile time. Whole classes of bugs that plague C and C++ programs are eliminated, because the compiler can guarantee that memory accesses at run time are safe under that model. Nevertheless, Rust is still young and, for the foreseeable future, software stacks for real-world use cases will rely on legacy code bases written in unsafe languages, because there is no Rust-based equivalent yet. Rust bridges this functionality gap with its foreign function interface (FFI), which allows for calls into existing libraries written in languages such as C. This integration of legacy functionality is seamless, as code and data structures of these libraries reside in the same address space as the main program written in Rust. The downside of this approach is that compile-time guarantees about memory safety become void as soon as a programming error in the C-based library causes a buffer overflow, out-of-bounds read, or other memory access violation.

In this paper, we present *Sandcrust*, a non-invasive and easy-to-use system to automatically sandbox unsafe C code in an isolated process. Our solution is based on the macro system of Rust. It translates, at compile-time, annotated functions wrapping the C library's API into remote procedure calls (RPC) to a library instance running in a sandboxed process; Sandcrust also keeps global variables in sync. As macro support is part of the stable branch of Rust, our solution does not require modifications to the compiler or runtime system. The contribution of Sandcrust is that the safety advantage of Rust over C or C++ remains, even if parts of the program rely on unsafe and potentially buggy C libraries.

The paper is structured as follows: The next section discusses basics of the Rust programming language. In Section 3, we describe the design of Sandcrust and discuss important parts of its implementation. We evaluate usability and performance in Section 4. In Section 5, we discuss related work before we conclude with avenues for future work.

2 Background

This section gives a brief overview of the basics of the Rust language and how they relate to memory safety. We further

PLOS'17, October 28, 2017, Shanghai, China

ACM ISBN 978-1-4503-5153-9/17/10.
https://doi.org/10.1145/3144555.3144562

describe Rust's Foreign Function Interface (FFI) and its macro system.

Memory Safety Model Rust is a statically typed language which uses Resource Acquisition Is Initialization (RAII) to prevent uninitialized variables. *Lifetimes* of objects are calculated based on the *scope* of a binding or explicit annotation. At the end of their lifetime, resources are deterministically freed in reverse order of their construction, thereby preventing use-after-free errors. A cornerstone of the Rust concurrency model is that variables are read-only by default. Furthermore, when a variable is passed into a different scope (e.g., as an argument in a function call), the binding of this variable *moves* into that scope, depriving the original scope from access. However, this strict notion of *Ownership* can be weakened by lending variables to functions using *References*. By explicit declaration, references can also be mutable. These restrictions enable the Rust compiler to detect reader/writer race conditions at compile time and support automatic memory management that does not require garbage collection.

Traits *Traits* specify that certain operations are supported on a data type; this is similar to a class implementing an interface in an object-oriented programming language. Rust has a number of built-in special traits that provide generic behavior of data types. For example, most primitive types implement the `Copy` trait, which enables variables to be passed by value instead of moving them into the scope of a called function. This way of handling function arguments is both convenient and an optimization for all types whose memory representation does not exceed the pointer size of the platform. As of Rust version 1.15, a `struct` as a composite of types can *derive* traits via *Procedural Macros*, if those traits are implemented for the members of the `struct`. Rust traits in conjunction with procedural macros are a powerful way to implement serialization and deserialization of complex data structures. As we will discuss in Section 3.2, these operations are essential in order to pass function arguments and return types during RPCs.

Foreign Function Interface Rust provides the `unsafe` keyword to denote code blocks that will not be subject to the safeguards described in the preceding paragraphs. For example, the `unsafe` keyword is necessary to dereference raw pointers and to call unsafe functions. These two operations are necessary to interact with external libraries through Rust's foreign function interface (FFI). The Rust C FFI is represented by `extern` blocks with additional attributes; external libraries are specified in the code via link attributes such as `#[link(name = "owfat")]`, which is equivalent to adding the `-lowfat` flag when linking a C program. It is common practice to wrap calls to unsafe C functions inside a safe Rust function. This wrapper is responsible for translating Rust data types into C data types for all function arguments, and vice versa for the return type of the function (if any).

Rust Macros During compilation, the Rust compiler walks the abstract syntax tree (AST) and replaces each node that matches a macro with the contents specified in the body of this macro. Matching macros is done recursively. Listing 1 shows an example: When matched on an identifier in line 2, the `double` macro assumes the identified object implements the `Add` trait and doubles its value. When invoked with a sequence of statements separated by "`;`", the macro in line 3 prints the value of a newly declared variable `a` before executing each statement twice.

```
 1 macro_rules! double {
 2   ($i:ident) => { $i + $i };
 3   ($($s:stmt);+) => {
 4     let a = 42;
 5     println!("macro a is {}", a);
 6     $( $s; $s;)+
 7   };
 8 }
 9
10 fn main() {
11   let a = 1;
12   let two = double!(a);
13   println!("double {} is {}", a, two);
14   double!(println!("main a is {}", a); println!("!"));
15 }
```

Listing 1. Macro Example

The output of this example program is:

```
double 1 is 2
macro a is 42
main a is 1
main a is 1
!
!
```

The conflicting output for variable `a` is a result of Rust's *hygienic* macro system, where variables introduced in macro expansion are tagged with a distinct *syntax context*. Therefore, the `println!()` in line 5 sees variable `a` as defined in line 4, whereas the expanded statements see variable `a` as defined in line 14.

3 Design and Implementation

Sandcrust is targeted at applications written in Rust that must – for functional reasons – also link against libraries developed in unsafe languages like C. We assume that the developer chose Rust over C or C++, because it is a much safer programming language and one that does not sacrifice performance for this property. Consequently, the goal of Sandcrust is to protect the Rust-based main program from unsafe code in libraries, which would otherwise void the memory safety and control-flow integrity guaranteed by the Rust compiler.

3.1 General Design

When using standard toolchains, the safe code generated by the Rust compiler shares a single address space with the unsafe and potentially buggy libraries. Therefore, Sandcrust must isolate both code bases and their data from each other. Isolation can be achieved by moving the unsafe library code to a separate process, where it can be sandboxed.

Sandboxing A sandbox is an instantiation of the Principle of Least Privilege [14]. It aims to minimize access permissions and isolate certain software components to limit the potential harm those components can do to the rest of the system. Sandbox implementations of modern monolithic operating systems employ a combination of per-process access control at the system-call level and hardware-based isolation [5]. FreeBSD's Jails [8] pioneered the concept; Linux offers several solutions, including *Seccomp-BPF* [3], a rule-based system-call filter that is complemented by *Namespaces* to virtualize resources such as the file system and user IDs.

The specific details of a sandbox implementation are orthogonal to the problem of splitting a monolithic program into two parts; they are therefore out of scope for this paper. The design and implementation of Sandcrust as discussed in the following is targeted at Seccomp-BPF and Linux namespaces, but could be adapted to other sandbox solutions, too.

Making the Cut To be practical, Sandcrust's method of separating safe and unsafe code must be non-invasive and easy to use. A sensible boundary is the API of the unsafe library. Thus, process-local function calls must be replaced by RPCs into the sandboxed process. As described in Section 2, each foreign function implemented in a C library is usually wrapped by a Rust function. This function transforms Rust data types passed as function arguments into the Rust representations of the C data types declared in the library's header file. Some of these transformations must be placed in `unsafe` code blocks, like those involving raw pointers. Furthermore, C APIs often blend error codes and return values, a practice that is discouraged in Rust and therefore translated by the wrapper into idiomatic use of `Option` and `Result` objects.

We decided to apply Sandcrust's macro transformations to the Rust wrappers, and not the original, unsafe C function. Due to this design choice, we isolate all unsafe operations in the sandboxed part of the program, thereby upholding Rust's safety guarantees for the main program.

Program Transformation The transformation of local function calls into RPCs should be transparent to the application developer. Ideally, she should not have to do anything except for enabling Sandcrust for specific (or all) libraries. Most prior work on automatic software compartmentalization is based on source-to-source translation or requires modification of the compiler. These approaches are highly invasive and rarely make it into production toolchains. Fortunately, the Rust toolchain enables access to the AST in simpler ways: via compiler plugins and through macros. Compiler plugins come as *Lint plugins* and *Syntax extensions.* Lint plugins extend the correctness checks of the Rust compiler, but are unable to manipulate the AST. In contrast, syntax extensions can manipulate the AST by running Rust code; they do not just match against patterns. Both syntax extensions and macros can be packaged in a *crate* and used in any program. But to function, syntax extensions need access to Rust's internal compiler API, which is only available in the *unstable* version of the Rust compiler and runtime. Macros are available in stable versions of Rust.

To maximize maintainability and availability, we decided against syntax extensions and instead built Sandcrust on the Rust macro system. Macros are a form of monomorphization: the generation of specialized code from a generic template. A limitation of this approach is that each transformation of the AST happens in the context of the matched node (i.e., during transformation of a function, we have no access to other functions in the AST). However, we found that this form of AST transformation is sufficient to create a sandbox in a separate process and forward function calls to it.

3.2 Process Model and Inter-Process Communication

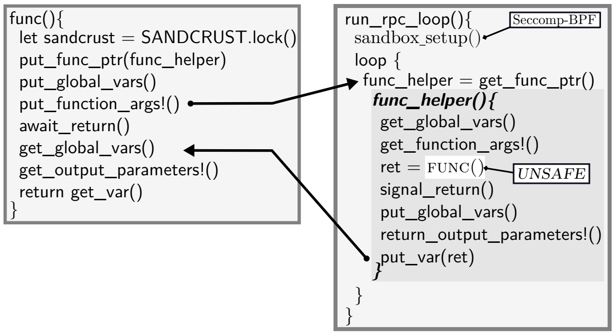
We will discuss the details of the macro-based program transformation and how developers use it in Section 3.3. But first, we explain how Sandcrust creates sandboxed processes and how its implements RPC.

Process Model A program that uses Sandcrust links against unsafe libraries in the same way as an ordinary Rust program. Thus, the binary includes the unsafe code from the libraries, but also code that is injected by Sandcrust's RPC wrapper macros. Before making the first RPC request, this code creates a *child* process using the `fork` system call. The child will immediately branch into a service loop, whereas the *parent* process resumes as the main program. The service loop is generated from macros as well; it waits for RPCs to execute unsafe library functions as requested by the function wrappers in the main program. This approach is different from other application sandboxes like the one used by Google Chrome: it re-executes the Chrome binary with a special command-line switch to spawn new sandboxed render processes [13]. Our approach has three advantages: (1) the application's build process need not be adapted to create a separate binary (e.g., one that only contains the unsafe library), (2) it is not necessary to add additional command-line parsing to the main program, and (3) code and global variables are at identical addresses in both processes.

Inter-Process Communication Our Sandcrust prototype uses two Unix pipes to send RPC requests and receive responses. We also prototyped a shared-memory based solution, but it showed no substantial performance advantage as it still needed memory copying and synchronization-related system calls. We therefore do not discuss it in this paper.

To forward function arguments and return values to another process, Sandcrust must be able to serialize and deserialize any data type used by the unsafe library's API. Rust does not offer built-in support for this task, but with the popular *ser*ialization and *de*serialization framework *Serde*[1],there is a

[1] https://serde.rs/



```
func(){
  let sandcrust = SANDCRUST.lock()
  put_func_ptr(func_helper)
  put_global_vars()
  put_function_args!()
  await_return()
  get_global_vars()
  get_output_parameters!()
  return get_var()
}
```

```
run_rpc_loop(){
  sandbox_setup()
  loop {
   func_helper = get_func_ptr()
    func_helper(){
     get_global_vars()
     get_function_args!()
     ret = FUNC()
     signal_return()
     put_global_vars()
     return_output_parameters!()
     put_var(ret)
    }
  }
}
```

Figure 1. Sandcrust Workflow

third-party solution available. Serde implements `Serialize` and `Deserialize` traits for all data types of the Rust standard library. Using procedural macros, the developer can derive Serde-compatible implementations of `Serialize` and `Deserialize` for arbitrary compound data structures:

```
#[derive(Serialize, Deserialize, PartialEq)]
struct Entity {
  x: f32,
  y: f32,
}

#[derive(Serialize, Deserialize, PartialEq)]
struct World {
  entities: Vec<Entity>,
}
```

If a C library uses complex data structures, the Sandcrust annotations must include `derive` attributes as shown above. Additionally, Sandcrust uses *Bincode*[2] to encode the Serde output in a byte stream such that it can be transmitted via the two Unix pipes. We will discuss potential performance implications of Bincode in Section 4.

3.3 Sandbox Management via Rust Macros

Figure 1 summarizes how Sandcrust transforms a process-local function call into an RPC. The left side of the figure shows pseudo code of a macro-generated replacement for `func`. This code runs in the main process and forwards global variable state and function arguments through the Unix pipe to the sandboxed process. The request loop in the sandboxed process receives the global variable state and function arguments, calls the real, unsafe version of `func`, and then relays back the updated contents of global variables and the result of the function call (output parameters and return value). When signaled by the sandboxed process, the replacement of `func` in the main process receives the state updates and returns the result value to the caller.

The difficulty of implementing this RPC scheme using macros is rooted in the way macro expansion works in Rust: Whenever a macro is referenced in the source code, the code in its body has access only to the small part of the AST that represents the macro's matching pattern. A consequence of this limitation is that macros that transform, for example, a function definition, cannot manipulate any functions outside this scope of said function. In the remainder of this section, we explain how we implemented Sandcrust's annotation macros and how code snippets generated by isolated macro instantiation can cooperate as shown in Figure 1.

Transforming Function Wrappers Sandcrust's transformation macros work by annotating each wrapper of an unsafe C function with `sandbox!{}`. The function signatures matched by this macro include the argument types in the signature (e.g., `func(a: i32, b: Vec<u8>)`). To generate a function-call statement for this function, the types need to be stripped away, resulting in `func(a, b)`. However, care must be taken that elements of the argument list do not become *un-destructible*, which impedes recursive matching of the argument list as a list of `expressions`: “`a: i32`“ would appear as a single AST node that is no longer matched by the `sandcrust_strip_types!{}` macro.

Listing 2 shows the implementation from our prototype. A function without any arguments is matched by line 6. Line 5 starts, using *Push Down Accumulation*, the recursion that builds up the function-call statement without type info (after “`->`“ in the macro). The first match in line 2 performs the strip and recursive invocation, if there are still more arguments in the `$($tail:tt)` list. Finally, line 3 matches a function with one remaining “`&mut`“ argument, strips away the type and outputs a function invocation in place of the macro, with the added argument. To support other types of parameters (i.e., non-mutable, reference), three pairs of match arms structurally similar to lines 2 and 3 are necessary.

Managing the Sandbox Process Many libraries maintain internal state across multiple API calls. To preserve this hidden state, Sandcrust must keep the sandboxed process alive for the entire run time of the program. To this end, Sandcrust encapsulates process ID and communication channels in a global library object. This object is initialized on first use by help of the *lazy_static* crate. Additionally, the RPC service loop must know about all transformed wrapper functions in order to call them. As individual macro transformations do not have access to the global AST, calls to wrapper functions cannot be added to the service loop directly. However, stable versions of Rust do not provide a way to generate a new identifier like `func_`*wrapped*`()`. We worked around this limitation by adding for each transformed function a trait to an otherwise empty struct `SandcrustWrapper`, which is also defined in the library crate. A carefully programmed block of unsafe code obtains a function pointer to the generated trait method, which is then passed to the RPC loop in the sandboxed process in order to call this method.

3.4 Features and Limitations

Sandcrust can automatically handle any native or derived data type that implements the *Serialize* and *Deserialize* traits

[2] https://crates.io/crates/bincode

```
macro_rules! sandcrust_strip_types {
(($head:ident : &mut $var_type:ty, $($tail:tt)+) -> ($f:ident($($body:tt)*))) => (sandcrust_strip_types!(($($tail)+) -> ($f($($body)* &mut
     $head,))));
(($head:ident : &mut $var_type:ty) -> ($f:ident($($body:tt)*))) => ($f($($body)* &mut $head));

($f:ident($($tail:tt)+)) => (sandcrust_strip_types!(($($tail)+) -> ($f())));
($f:ident()) => ($f());
}
```

Listing 2. Macro implementation

of the Serde framework. By design, we exclude *raw pointers* because they are unsafe. Handling C pointers, which are also unsafe, is the responsibility of the function wrapper, but is done in the sandboxed process. Sandcrust also synchronizes *mutable foreign globals* (i.e., mutable global variables defined in the C library) via the `sandcrust_wrap_global!` macro.

Function callbacks from within a library are supported within the sandboxed process, as demonstrated in the *libpng* case study in Section 4. Our prototype does not support callbacks into the main process, but this feature could be added. However, the `setjmp`/`longjmp` pair of C functions is restricted to the sandboxed process; our *libpng* example makes use of this feature. We intentionally do not support `longjmp`ing into the main process, because these “non-local gotos“ jump upwards in the call stack, the outcome of which is undefined, if the calling function has already returned at the time of the `longjmp`; it is also prone to memory leaks.

Finally, Sandcrust cannot detect semantically incorrect results produced by unsafe code in the sandboxed process. This is a fundamental limitation of any sandboxing solution.

4 Evaluation

We evaluate Sandcrust based on two unsafe C libraries: a compression library that served as the FFI example in Chapter 5.9 of the Rust Book and the image codec libpng.

```
#[macro_use]
extern crate sandcrust;
extern crate libc;
use sandcrust::*;
[...]
sandbox!{
  pub fn compress(src: &[u8]) -> Vec<u8> { [...] }
}
sandbox!{
  pub fn uncompress(src: &[u8]) -> Option<Vec<u8>> {
    unsafe {
      [...]
      if snappy_uncompress(psrc, srclen, pdst, &mut dstlen) == 0 {
        dst.set_len(dstlen as usize);
        Some(dst)
      } else {
        None // SNAPPY_INVALID_INPUT
      }
    }
  }
}
```

Listing 3. Instrumentation of Snappy Compression Library

Sandboxing Snappy Listing 3 shows how to import the Sandcrust crate and how to annotate the wrapper functions `compress` and `uncompress`, requiring nothing more than encompassing them with the `sandbox!{}` macro. Although not shown in this example, Sandcrust annotations with this macro can be packaged in the same library crate that provides the FFI function wrappers.

Sandboxing libpng In the second, more complex case study, we sandboxed *libpng*, which suffered from an impressive number of security issues in the past. This library has a highly complex API, for which Listing 4 lists only the wrappers that encapsulate *libpng* functions used in our example. The `callback` function is used to read PNG image data from a buffer and is called from within the sandbox.

```
fn read_file(path: &str) -> Vec<u8>;
fn png_init() -> Result<(), String>;
fn is_png(buf: &[u8]) -> bool;
extern "C" fn callback(callback_png_ptr: *mut png_struct,
                       buf_ptr: *mut u8, count: png_size_t);
fn decode_png(png_image: &[u8]) -> Result<Vec<Vec<u8>>, String>;
```

Listing 4. Function Signatures for libpng

Benchmark Setup We measured Sandcrust’s overhead on an Intel Core-i7 5600U CPU, running a Linux 4.9 kernel and Rust 1.16. We instrumented our test programs with a macro that runs several warm-up rounds before measuring the run time of a sandboxed function call using `clock_gettime`. We determined the time that elapsed between two consecutive calls of `clock_gettime` to be between 83 *ns* and 85 *ns*, which we subtracted from all measurements. Since all measurements are in the order of microseconds or more, the system clock’s resolution is sufficient. Unless stated otherwise, all figures represent the median of several benchmark runs.

Microbenchmarks We sandboxed the *abs* function of the standard C library, to determine the overhead of Sandcrust. Calling this function directly takes in the order of one to two nanoseconds. Creating and initializing the sandbox process took 1469 μs, which contributed the majority of the overhead of 1506 μs for calling the sandboxed instance `abs` for the first time. Subsequent invocations took 4.54 μs. Explicit termination of the sandbox process took 162 μs.

Snappy Performance The graph in Figure 2 shows run times of both unsafe and sandboxed invocations of `compress` and `uncompress` from the Snappy test case, with data sizes ranging from 4 bytes up to 16 MiB. There is a significant difference in overhead between the compress and the uncompress functions: The former goes down from a slowdown factor of 9.28 to a minimum of 1.3 at 2^{18} bytes before stabilizing at a factor of approximately 1.5. The overhead of the sandboxed uncompress function goes down steadily from a

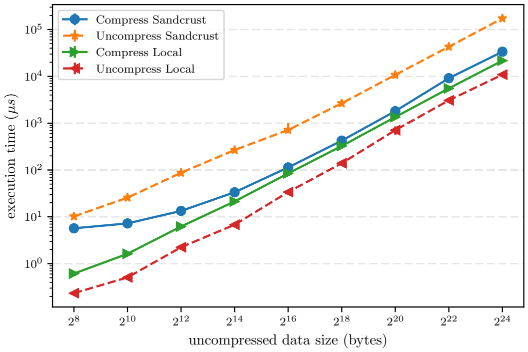


Figure 2. Snappy Overhead

File	Size (bytes)	Slowdown factor
rust-logo-256x256-blk.png	7460	7.40
myblood.png	42693	6.98
objects.png	386814	2.07
arithmetic28x11.png	894174	3.25

Table 1. PNG decoding test data

factor of 44 to 16. To explain this unintuitive result, we have to refer to the signatures of the wrapper functions:

```
pub fn compress(src: &[u8]) -> Vec<u8>
pub fn uncompress(src: &[u8]) -> Option<Vec<u8>>
```

The result of `compress` is a potentially large byte array of type `Vec<u8>`, which Sandcrust transmits through the Unix pipes using optimized read and write functions: they transfer the complete memory that backs a `Vec<u8>` via a single read or write call on the file object. In contrast, the result of `uncompress` is wrapped in an `Option<T>` enum, which is a complex type that must be handled by the third-party package Bincode. However, Bincode is extremely inefficient for byte arrays and, unfortunately, we did not have the time to optimize it like we did for unwrapped `Vec<u8>` objects.

libpng Performance Table 1 shows overhead factors for decoding four different PNG images using sandboxed libpng. As the uncompressed image returned by `decode_png` must be handled by Bincode, decoding in the sandboxed process takes 2 to 7.4 times as long as in the same process, with a sweet spot between cross-process function-call latency and data transfer overhead for "objects.png".

5 Related Work

In 1974, ICOPS [16] pioneered the use of annotated procedures that trap into an RPC runtime to enable distributed computing. It encountered issues that remained hard problems in later work such as handling global variables and pointers. Critical software packages like OpenSSH [12] were split up manually to prevent privilege escalation; the Privman [9] library aided other developers with this task.

Another line of work is centered around microkernels like Mach [1] and L4 [10] and aims at reducing the Trusted Computing Base (TCB) of security-critical components. For example, the VPFS file system [18] based on the Nizza architecture [6] isolates cryptographic components from legacy software running in L^{4}Linux [7]. Similarly, Proxos [15] routes sensitive system calls to a *Private OS* using a Virtual Machine Monitor (VMM). Due to its focus on seamless integration into an existing development ecosystem, Sandcrust is ill-suited for such complex re-architecting of systems. However, it may provide a starting point for future research into better language support for componentized applications.

Hardware-based support for compartmentalization within the same address space is proposed by the CHERI system [17, 19]. It enhances the MIPS CPU architecture with byte-granular protection through memory capabilities, which could in principle replace processes as Sandcrust's means of isolation.

As a first implementation of Decentralized Information Flow Control (DIFC), JFLow [11] restricted data access between different components of a Java program. Privtrans [4] first applied the idea to C. However, the most comprehensive implementation in Wegde [2] shows that the approach requires too much manual work for existing software.

The work closest to Sandcrust is Codejail [20]. It isolates a C program from its libraries without any modifications, but developers must write new function wrappers, which in the case of Sandcrust already exist in form of easily-annotated FFI wrappers. Sandcrust synchronizes global variables and output parameters seamlessly via Rust macros, whereas Codejail requires manual instrumentation to commit changes by the sandboxed process to the main program.

6 Conclusion

For the foreseeable future, programs written in safe programming languages such as Rust will have to rely on components written in unsafe languages like C. In this paper, we introduced Sandcrust, a solution to automatically sandbox unsafe C libraries with minimal annotation and without any modification to the development toolchain. Sandcrust shows that it is possible to contain unsafe behavior in another address space, thereby upholding safety guarantees of the Rust language and compiler. To achieve this, we modify the program via Rust macros, which can be packaged into a library crate to provide a turnkey solution. Sandcrust leverages Rust's strict typing system to automatically derive information about data types, while prior work had to rely on manual annotation.

Acknowledgments

The authors would like to thank the anonymous reviewers for their help on improving this paper. This work has been supported by research grants from DFG via German priority program 1648 and the Cluster of Excellence "Center for Advancing Electronics Dresden" (*cfaed*).

References

[1] Michael J. Accetta, Robert V. Baron, William J. Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Young. 1986. Mach: A New Kernel Foundation for UNIX Development.. In *USENIX Summer*. USENIX Association, 93–113.

[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 (NSDI'08)*. USENIX Association, 309–322.

[3] Ma Bo, Mu Dejun, Fan Wei, and Hu Wei. 2013. Improvements the Seccomp sandbox based on PBE theory. In *Advanced Information Networking and Applications Workshops (WAINA), 2013 27th International Conference on*. IEEE, 323–328.

[4] 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 (SSYM'04)*. USENIX Association.

[5] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. 1996. A Secure Environment for Untrusted Helper Applications.. In *USENIX Security*. USENIX Association.

[6] Hermann Härtig, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter. 2005. The Nizza secure-system architecture. In *CollaborateCom*.

[7] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. 1997. The Performance of μKernel-Based Systems.. In *SOSP*. 66–77.

[8] Poul-Henning Kamp and Robert NM Watson. 2000. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, Vol. 43. 116.

[9] Douglas Kilpatrick. 2003. Privman: A Library for Partitioning Applications.. In *USENIX Annual Technical Conference, FREENIX Track*. USENIX, 273–284.

[10] J. Liedtke. 1995. On Micro-kernel Construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM, 237–250.

[11] Andrew C Myers. 1999. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 228–241.

[12] Niels Provos, Markus Friedl, and Peter Honeyman. 2003. Preventing Privilege Escalation. *12th USENIX Security Symposium* (Aug. 2003), 11.

[13] Charles Reis and Steven D. Gribble. 2009. Isolating Web Programs in Modern Browser Architectures. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*. ACM, 219–232.

[14] J. H. Saltzer and M. D. Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (Sept. 1975), 1278–1308.

[15] Richard Ta-Min, Lionel Litty, and David Lie. 2006. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *OSDI*, Brian N. Bershad and Jeffrey C. Mogul (Eds.). USENIX Association, 279–292.

[16] A. van Dam, G. M. Stabler, and R. J. Harrington. 1974. Intelligent Satellites for Interactive Graphics. *Proc. IEEE* 62, 4 (April 1974), 483–492.

[17] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. 2015. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 20–37.

[18] Carsten Weinhold and Hermann Härtig. 2008. VPFS: Building a Virtual Private File System with a Small Trusted Computing Base. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (Eurosys '08)*. ACM, 81–93.

[19] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. IEEE, 457–468.

[20] 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: 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, Sara Foresti, Moti Yung, and Fabio Martinelli (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 859–876.