

# SANDCRUST: AUTOMATIC SANDBOXING OF UNSAFE COMPONENTS IN RUST

---

Benjamin Lamowski, Carsten Weinhold<sup>1</sup>, Adam Lackorzynski<sup>1</sup>, Hermann Härtig<sup>1</sup>

October 28, 2017

<sup>1</sup> Technische Universität Dresden

C is “glorified assembly”:

- lack of memory safety
- concurrency not built into the language
- error handling is optional
- vague standards, different interpretations, undefined behavior
  - even if a programmer manually checks for an overflow, the compiler may optimize the check away (`c + len > c`)

- Traits as interfaces and data type properties (e.g., `Copy`, `Drop`)
- strict rules for memory access:
  - Resource Acquisition Is Initialization (RAII)
  - *Lifetimes*
- concurrency:
  - variables are read-only by default
  - *Ownership & References*
- **checked at compile time**
- mandatory handling of errors / absence of result (`Result<>` / `Option<>`)
- integer overflows checked in debug mode

- `unsafe` keyword to disable compiler checks
- *Foreign Function Interface (FFI)*
- execution in same address space
- nullifies Rust's security guarantees
- porting C software to Rust: safety vs. correctness

SANDCRUST

---

Sandcrust (Sandboxing C in Rust) isolation crate:

- isolates unsafe code in separate protection domain (process)
- integrates into language & development ecosystem
- compartmentalizes applications with minimal code changes
- focus on sandboxing reused C libraries



credit: Alan Levine / CC BY 2.0

- ideal: automatically wrap C library API
  - problems:
    1. containing *unsafe* data type transformations
    2. C by-reference parameters (pointers)
    3. error handling
- commonly Rust wrapper function around C API

solution: sandbox the wrapper for a safe interface

# CHOOSING A KNIFE



credit: Giulia Gasparro / CC BY-NC-ND 2.0

options:

options:

- compiler modification

options:

- compiler modification
- external source-to-source transformation
  - related work: Codejail (Wu et. al. 2012)

options:

- compiler modification
- external source-to-source transformation
  - related work: Codejail (Wu et. al. 2012)
- compiler plugin (*Syntax Extension*) **unstable**

options:

- compiler modification
- external source-to-source transformation
  - related work: Codejail (Wu et. al. 2012)
- compiler plugin (*Syntax Extension*) **unstable**
- macros

options:

- compiler modification
- external source-to-source transformation
  - related work: Codejail (Wu et. al. 2012)
- compiler plugin (*Syntax Extension*) **unstable**
- **macros**

options:

- compiler modification
- external source-to-source transformation
  - related work: Codejail (Wu et. al. 2012)
- compiler plugin (*Syntax Extension*) **unstable**
- **macros**

macro limitations:

- metaprogramming (match/replace on Abstract Syntax Tree)
- isolated modifications of wrapped code
- lack of compiler intrinsics (e.g., type of variable)

# MACRO EXAMPLE

```
1 macro_rules! instrument_function {
2     (fn $f:ident($($x:tt)*) $body:block ) => {
3         fn $f($($x)* {
4             println!("instrumenting $f");
5             $body
6         }
7     }
8 }
9
10 instrument_function!{
11     fn inc_arg(a: &mut i32) {
12         *a += 1;
13     }
14 }
```

# MACRO EXAMPLE

```
1 macro_rules! instrument_function {
2     (fn $f:ident($($x:tt)*) $body:block ) => {
3         fn $f($($x)* {
4             println!("instrumenting $f");
5             $body
6         }
7     }
8 }
9
10 instrument_function!{
11     fn inc_arg(a: &mut i32) {
12         *a += 1;
13     }
14 }
```

# MACRO EXAMPLE

```
1 macro_rules! instrument_function {
2     (fn $f:ident($($x:tt)*) $body:block ) => {
3         fn $f($($x)*) {
4             println!("instrumenting $f");
5             $body
6         }
7     }
8 }
9
10 instrument_function!{
11     fn inc_arg(a: &mut i32) {
12         *a += 1;
13     }
14 }
```

# MACRO EXAMPLE

```
1 macro_rules! instrument_function {
2     (fn $f:ident($($x:tt)*) $body:block ) => {
3         fn $f($($x)* {
4             println!("instrumenting $f");
5             $body
6         }
7     }
8 }
9
10 instrument_function!{
11     fn inc_arg(a: &mut i32) {
12         *a += 1;
13     }
14 }
```

# MACRO EXAMPLE

```
1 macro_rules! instrument_function {
2     (fn $f:ident($($x:tt)*) $body:block ) => {
3         fn $f($($x)*) {
4             println!("instrumenting $f");
5             $body
6         }
7     }
8 }
9
10 instrument_function!{
11     fn inc_arg(a: &mut i32) {
12         *a += 1;
13     }
14 }
```

# MACRO EXAMPLE

```
1 macro_rules! instrument_function {
2     (fn $f:ident($($x:tt)*) $body:block ) => {
3         fn $f($($x)*) {
4             println!("instrumenting $f");
5             $body
6         }
7     }
8 }
9
10 instrument_function!{
11     fn inc_arg(a: &mut i32) {
12         *a += 1;
13     }
14 }
```

```
1  extern crate libc;
2  use libc::.*;
3
4  #[link(name = "snappy")]
5  extern {
6      fn snappy_compress(input: *const u8, input_length: size_t, compressed: *mut u8,
7                          compressed_length: *mut size_t) -> c_int;
8      fn snappy_max_compressed_length(source_length: size_t) -> size_t;
9  }
10 pub fn compress(src: &[u8]) -> Vec<u8> {
11     unsafe {
12         let srclen = src.len() as size_t;
13         let psrc = src.as_ptr();
14         let mut dstlen = snappy_max_compressed_length(srclen);
15         let mut dst = Vec::with_capacity(dstlen as usize);
16         let pdst = dst.as_mut_ptr();
17         snappy_compress(psrc, srclen, pdst, &mut dstlen);
18         dst.set_len(dstlen as usize);
19         dst
20     }
21 }
22
23 fn main() {
24     let d = vec![0xde, 0xad, 0xd0, 0xd0];
25     let c: &[u8] = &compress(&d);
26 }
```

```
1  extern crate libc;
2  use libc::.*;
3
4  #[link(name = "snappy")]
5  extern {
6      fn snappy_compress(input: *const u8, input_length: size_t, compressed: *mut u8,
7                          compressed_length: *mut size_t) -> c_int;
8      fn snappy_max_compressed_length(source_length: size_t) -> size_t;
9  }
10
11 pub fn compress(src: &[u8]) -> Vec<u8> {
12     unsafe {
13         let srclen = src.len() as size_t;
14         let psrc = src.as_ptr();
15         let mut dstlen = snappy_max_compressed_length(srclen);
16         let mut dst = Vec::with_capacity(dstlen as usize);
17         let pdst = dst.as_mut_ptr();
18         snappy_compress(psrc, srclen, pdst, &mut dstlen);
19         dst.set_len(dstlen as usize);
20     }
21 }
22
23 fn main() {
24     let d = vec![0xde, 0xad, 0xd0, 0x0d];
25     let c: &[u8] = &compress(&d);
26 }
```

# WRAPPER EXAMPLE WITH SANDCRUST

```
1  extern crate libc;
2  use libc::.*;
3
4  #[macro_use]
5  extern crate sandcrust;
6  use sandcrust::*;

7
8  #[link(name = "snappy")]
9  extern {
10      fn snappy_compress(input: *const u8, input_length: size_t, compressed: *mut u8,
11      ↪ compressed_length: *mut size_t) -> c_int;
12      fn snappy_max_compressed_length(source_length: size_t) -> size_t;
13  }
14
15  sandbox!{
16      pub fn compress(src: &[u8]) -> Vec<u8> {
17          unsafe { // Implementation omitted
18
19          }
20      }
21  }
22
23
24
25
26
27
28
29  fn main() {
30      let d = vec![0xde, 0xad, 0xd0, 0x0d];
31      let c: &[u8] = &compress(&d);
32  }
```

- challenge: generate global IPC endpoint from local transformation
- IPC via Unix pipe pair
- *Serde* serialization and **deserialization** framework  
implements **Serialize** and **Deserialize** traits
- *Procedural Macros* **derive** traits from struct members

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

- data marshalling via *Bincode* library
- Sandcrust byte vector optimizations

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

```
run_rpc_loop(){  
    sandbox_setup()  
    loop {  
        func_helper = get_func_ptr()  
        func_helper(){  
            get_function_args!()  
            ret = func() ← UNSAFE  
            return_output_parameters!()  
            put_var(ret)  
        }  
    }  
}
```

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

```
run_rpc_loop(){  
    sandbox_setup() ← Seccomp-BPF  
    loop {  
        func_helper = get_func_ptr()  
        func_helper(){  
            get_function_args!()  
            ret = func() ← UNSAFE  
            return_output_parameters!()  
            put_var(ret)  
        }  
    }  
}
```

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

```
run_rpc_loop(){  
    sandbox_setup() ← Seccomp-BPF  
    loop {  
        func_helper = get_func_ptr()  
        func_helper(){  
            get_function_args!()  
            ret = func() ← UNSAFE  
            return_output_parameters!()  
            put_var(ret)  
        }  
    }  
}
```

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

```
run_rpc_loop(){  
    sandbox_setup() ← Seccomp-BPF  
    loop {  
        func_helper = get_func_ptr()  
        func_helper(){  
            get_function_args!()  
            ret = func() ← UNSAFE  
            return_output_parameters!()  
            put_var(ret)  
        }  
    }  
}
```

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

```
run_rpc_loop(){  
    sandbox_setup()  
    loop {  
        func_helper = get_func_ptr()  
        func_helper(){  
            get_function_args!()  
            ret = func() ← UNSAFE  
            return_output_parameters!()  
            put_var(ret)  
        }  
    }  
}
```

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

```
run_rpc_loop(){  
    sandbox_setup() ← Seccomp-BPF  
    loop {  
        func_helper = get_func_ptr()  
        func_helper(){  
            get_function_args!()  
            ret = func() ← UNSAFE  
            return_output_parameters!()  
            put_var(ret)  
        }  
    }  
}
```

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

```
run_rpc_loop(){  
    sandbox_setup() ← Seccomp-BPF  
    loop {  
        func_helper = get_func_ptr()  
        func_helper(){  
            get_function_args!()  
            ret = func() ← UNSAFE  
            return_output_parameters!()  
            put_var(ret)  
        }  
    }  
}
```

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

```
run_rpc_loop(){  
    sandbox_setup() ← Seccomp-BPF  
    loop {  
        func_helper = get_func_ptr()  
        func_helper(){  
            get_function_args!()  
            ret = func() ← UNSAFE  
            return_output_parameters!()  
            put_var(ret)  
        }  
    }  
}
```

## EVALUATION

---

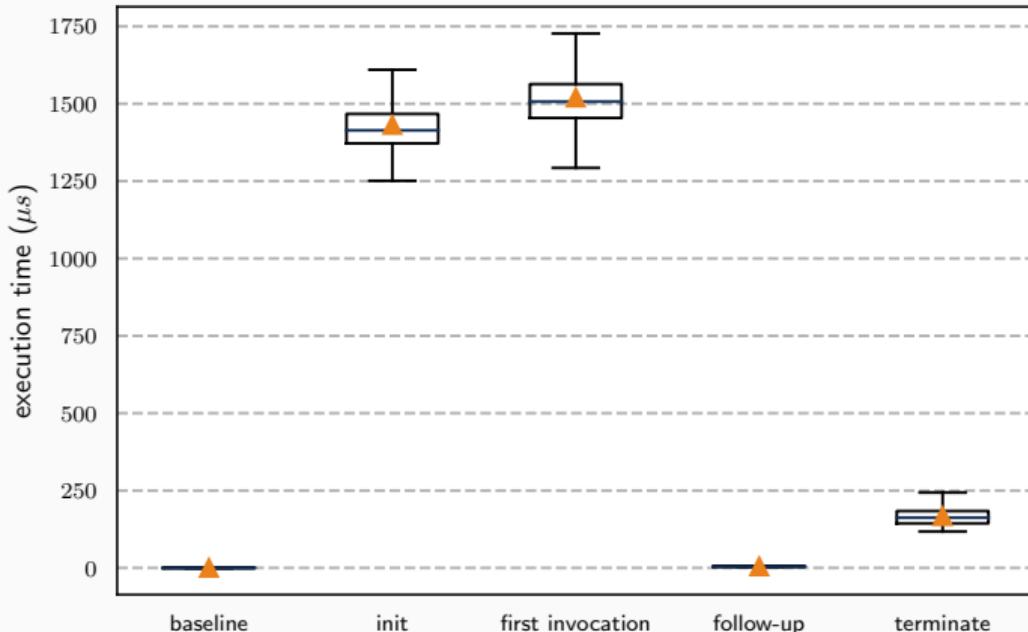
library interactions:

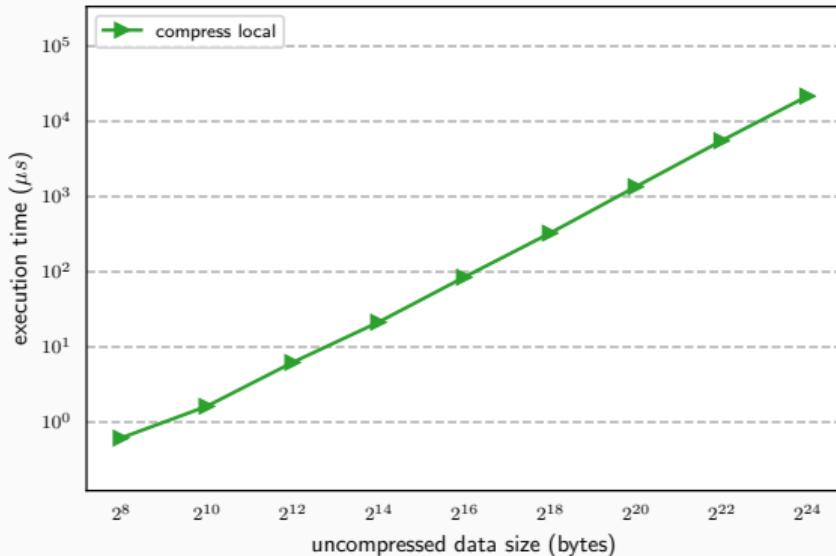
- by-value & by-reference parameter passing and return
- global variables
- function callbacks (future work)
- long jumps **highly unsafe, only in sandboxed process**

data types:

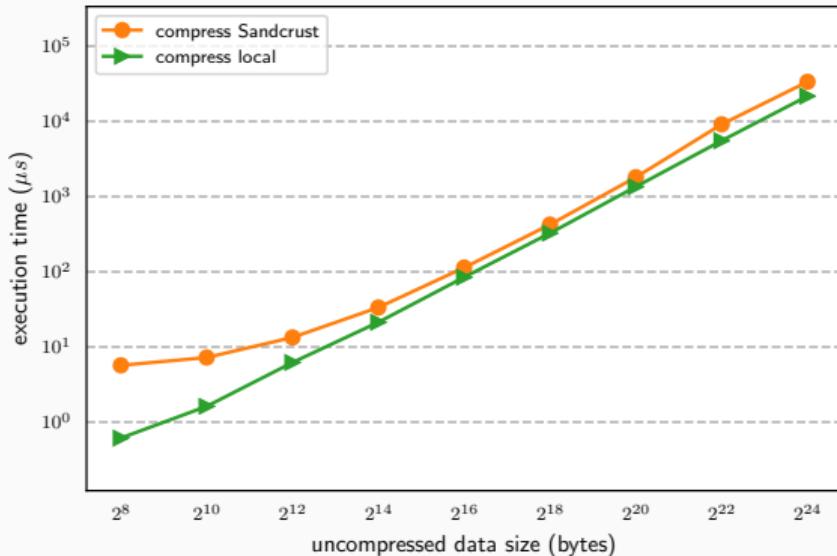
- that implement **Serialize** and **Deserialize** traits
- manual **#[derive]** annotations necessary for composed data types
- (**unsafe**) raw pointers **not** supported

# MICROBENCHMARKS

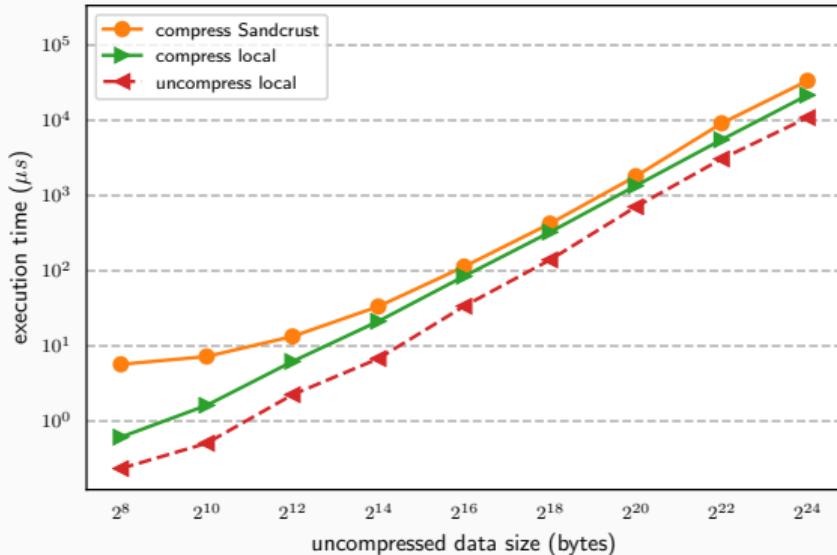




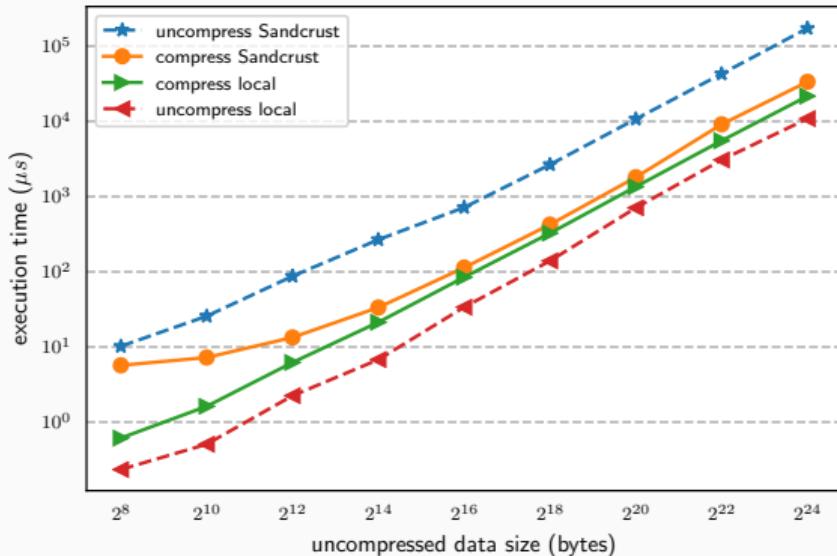
```
pub fn compress(src: &[u8]) -> Vec<u8>
pub fn uncompress(src: &[u8]) -> Option<Vec<u8>>
```



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



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



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

global data structures, multiple wrapper calls to decode PNG:

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

global data structures, multiple wrapper calls to decode PNG:

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

ID	File	Size (KiB)	Sandcrust Slowdown Factor
1	rust-logo-256x256-blk.png	7.3	7.40
2	myblood.png	42	6.98
3	objects.png	378	2.07
4	arithmetic28x11.png	874	3.25

## OUTLOOK & CONCLUSION

---

- improve Bincode performance / replace Bincode
- implement support for callbacks into the trusted process
- remove serializing effect of IPC
- **#[sandboxed]** function attribute via stable syntax extension

- Sandcrust enables automatic sandboxing of unsafe code
- it integrates seamlessly into a Rust project
- writing safe wrapper functions still necessary

Unsafe code sandboxed with Sandcrust runs outside the trusted process and does not diminish Rust's security guarantees.



## SANDBOX LIMITATIONS

- library initialization
- in-memory resources:
  - memory mappings
  - shared memory
  - Unix environment
- solution for memory: early `sandcrust_init()`