

# Easier Plugin Design with Rust

---

Clevis in MariaDB  
Trevor Gross

# Background: Why Rust?

---

## Sample Rust

# What is it?

- Compiled to assembly
- No garbage collector
- Prevents the errors you shouldn't make but do
- Compiler guarantee:  
*zero runtime undefined behavior*

```
1 pub struct Foo {
2     a: u64,
3     b: u64,
4 }
5
6 // Methods are "bolted on", more like C than classes
7 impl Foo {
8     // This gets used as `some_foo.multiply()`
9     pub fn multiply(&self) -> u64 {
10         // Last line is the return statement!
11         self.a * self.b
12     }
13 }
```

C Equivalent

```
1 example::Foo::multiply:
2     mov     rax, qword ptr [rdi + 8]
3     imul  rax, qword ptr [rdi]
4     ret
```

```
1 typedef struct Foo {
2     // Or uint64_t, but this is more fun
3     unsigned long long int a;
4     unsigned long long int b;
5 } Foo;
6
7 unsigned long long int foo_multiply(Foo *foo) {
8     return foo->a * foo->b;
9 }
10
```

```
1 foo_multiply:                                # @foo
2     mov     rax, qword ptr [rdi + 8]
3     imul  rax, qword ptr [rdi]
4     ret
```

Tip: **return** isn't needed if the last line is the return value

# Who's Using It?



index : kernel/git/torvalds/linux.git

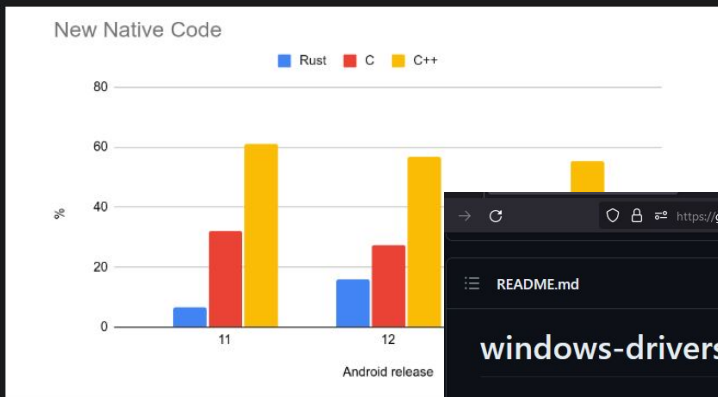
Linux kernel source tree

about summary refs log **tree** commit diff stats

```
Mode      Name
-rw-r--r-- .clang-format
-rw-r--r-- .cocciconfig
-rw-r--r-- .get_maintainer.ignore
-rw-r--r-- .gitattributes
-rw-r--r-- .gitignore
-rw-r--r-- .mailmap
-rw-r--r-- .rustfmt.toml
-rw-r--r-- COPYING
-rw-r--r-- CREDITS
d----- Documentation
-rw-r--r-- Kbuild
-rw-r--r-- Kconfig
d----- LICENSES
-rw-r--r-- MAINTAINERS
-rw-r--r-- Makefile
-rw-r--r-- README
d----- arch
d----- block
d----- certs
d----- crypto
d----- drivers
d----- fs
d----- include
d----- init
d----- io_uring
d----- ipc
d----- kernel
d----- lib
d----- mm
d----- net
d----- rust ←
d----- samples
d----- scripts
```

<https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/>

As we noted in the original announcement, our goal is not to convert existing C/C++ to Rust, but rather to shift development of new code to memory safe languages over time.



As the amount of new memory-unsafe code entering Android decreased, the number of memory safety vulnerabilities fell from 76% in 2019 to 35% in 2022. **2022 memory safety vulnerabilities do not represent a majority of Android's total vulnerabilities.**

<https://security.googleblog.com/2022/12/memory-safe-code/>

README.md

## windows-drivers-rs

This repo is a collection of Rust crates that enable developers to develop Windows Drivers in Rust. It is the intention to support both WDM and WDF driver development models. This repo contains the following crates:

- wdk-build**: A library to configure a Cargo build script for binding generation and downstream linking of the WDK (Windows Developer Kit). While this crate is written to be flexible with different WDK releases and different WDF version, it is currently only tested for NI eWDK, KMDF 1.33, UMDf 2.33, and WDM Drivers. There may be missing linker options for older DDKs.
- wdk-sys**: Direct FFI bindings to APIs available in the Windows Development Kit (WDK). This includes both autogenerated ffi bindings from `bindgen`, and also manual re-implementations of macros that `bindgen` fails to generate.

<https://github.com/microsoft/windows-drivers-rs>

# Lifetimes in C

```
1  #include <stdio.h>
2
3  struct Thing { int* x; };
4
5  struct Thing make_thing() {
6      int local = 100;
7      struct Thing foo = { .x = &local };
8      return foo;
9  }
10
11 int main() {
12     struct Thing bar = make_thing();
13     printf("first x in thing: %i\n", *bar.x);
14 }
15
```

```
tmgross@quince scratch % ./a.out
first x in thing: 100
```

# Lifetimes in C

```
17 #include <stdio.h>
18
19 struct Thing { int* x; };
20
21 struct Thing make_thing() {
22     int local = 100;
23     struct Thing foo = { .x = &local };
24     return foo;
25 }
26
27 int i_use_the_stack() {
28     int local = 5000;
29     return local * 10;
30 }
31
32 int main() {
33     struct Thing bar = make_thing();
34     printf("first x in thing: %i\n", *bar.x);
35     i_use_the_stack();
36     printf("second x in thing: %i\n", *bar.x);
37 }
```

```
[tmgross@quince scratch % ./a.out
first x in thing: 100
second x in thing: 32759
```

```
tmgross@quince scratch % gcc c-ret-bad.c -Wall \
> -Werror --pedantic -O3 -o a-opt.out
tmgross@quince scratch % ./a-opt.out
first x in thing: -1107384008
second x in thing: 73896
```

# Lifetimes in Rust

```
1  #[derive(Debug, Clone)]
2  struct Thing {
3      x: &u32,
4  }
5
6  fn make_thing() -> Thing {
7      let local = 1u32;
8      Thing { x: &local }
9  }
10
11 fn main() {
12     let bar = make_thing();
13
14     // The `#[derive(Debug)]` above lets us print our
15     // struct directly using debug formatting (`:?`)
16     println!("{:?}", bar);
17 }
```

```
error[E0106]: missing lifetime specifier
```

```
→ a.rs:3:8
```

```
3 |     x: &u32,
  |         ^ expected named lifetime parameter
```

```
help: consider introducing a named lifetime parameter
```

```
2 ~ struct Thing<'a> {
```

```
3 ~     x: &'a u32,
```

```
error: aborting due to previous error
```

```
For more information about this error, try `rustc --explain E0106`.
```

# Lifetimes in Rust

A **Thing** will exist for some time. This says: “call a **Thing**’s lifetime ‘**a**”

```
3  #[derive(Debug, Clone)]
4  struct Thing<'a> {
5      x: &'a u32,
6  }
7
8  fn make_thing(x_ref: &u32) -> Thing {
9      Thing { x: x_ref }
10 }
11
12 fn main() {
13     let local: u32 = 100;
14     let bar = make_thing(&local);
15
16     // The `#[derive(Debug)]` above lets us print our
17     // struct directly using debug formatting (`{:?}`)
18     println!("{:?}", bar);
19 }
```

This says:  
**x** must be valid for (at least) ‘**a**’

Fix the issue: make **x** something that lives as long as **Thing** (**x\_ref** does)

```
Thing { x: 100 }
```



# Aliasing & Validity

```
3 fn main() {
4     let mut v: Vec<u8> = vec![1, 2, 3, 4];
5     let item_ref: &u8 = &v[3];
6
7     // dbg! is a handy macro for, well, debugging
8     // references print their value by default (instead of as a pointer)
9     dbg!(item_ref);
10
11    // empty the vector
12    v.clear();
13
14    // try to print again
15    dbg!(item_ref);
16 }
```

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
--> examples/simple-vec-alias-fail.rs:12:5
```

```
5 | |         let item_ref: &u8 = &v[3];
   | |                               - immutable borrow occurs here
...
12 | |         v.clear();
   | |         ^^^^^^^^^^ mutable borrow occurs here
...
15 | |         dbg!(item_ref);
   | |             ----- immutable borrow later used here
```

For more information about this error, try `rustc --explain E0502`.`

# How Does it Know?

Every item has:

- The data itself (owner)
- 0 to  $\infty$  immutable (const) references (&)
- **OR** one mutable reference (&mut)
- Never both!
- References *cannot* outlive the owner

```
3 fn main() {
  created owner let mut v: Vec<u8> = vec![1, 2, 3, 4];
  created & let item_ref: &u8 = &v[3];
6
7 // dbg! is a handy macro for, well, debugging
8 // references print their value by default (ins
used & dbg!(item_ref);
10
11 // empty the vector
used &mut v.clear();
12
13
14 // try to print again
used & dbg!(item_ref);
16 }
```

```
[ ] pub fn clear(&mut self)
```

Clears the vector, removing all values.

Note that this method has no effect on the allocated capacity of the vector.

## Examples

```
let mut v = vec![1, 2, 3];

v.clear();

assert!(v.is_empty());
```

# The Result

## Eliminated:

- Over/underflow
- Segfaults/trap (with the help of bounds checking)
- Data races (aliasing)
- Use after free / double free


```
3 // This attribute disables a compile time bound check
4 // (since arr[20] will fail)
5 #[allow(unconditional_panic)]
6 fn main () {
7     let arr = [1u8, 2, 100, 200];
8
9     // By the way, you don't need to manually track
10    // array / buffer length
11    for val in arr {
12        if val > 50 { println!("{val}: woah, slow down"); }
13    }
14
15    println!("this will fail: {}", arr[20]);
16 }
```

```
[tmgross@quince scratch % ./arr-panic
100: woah, slow down
200: woah, slow down
thread 'main' panicked at 'index out of bounds: the len is 4 but the
index is 20', arr-panic.rs:15:36
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace
```


# Rust & MariaDB

---

# User-Defined Functions

Crate **udf** 

[source](#) · [\[-\]](#)

 A wrapper crate to make writing SQL user-defined functions (UDFs) easy

This crate provides bindings for easy creation of SQL user-defined functions in Rust. See [the readme](#) for more background information on how UDFs work in general.

## Usage

Using this crate is fairly simple: create a struct that will be used to share data among UDF function calls (which can be zero-sized), then implement needed traits for it. `BasicUdf` provides function signatures for standard UDFs, and `AggregateUdf` provides signatures for aggregate (and window) UDFs. See the documentation there for a step-by-step guide.

```
use udf::prelude::*;

// Our struct that will produce a UDF of name `my_udf`
// If there is no data to store between calls, it can be zero sized
struct MyUdf;

// Specifying a name is optional; `#[register]` uses a snake case version of
// the struct name by default (`my_udf` in this case)
#[register(name = "my_shiny_udf")]
impl BasicUdf for MyUdf {
    // Specify return type of this UDF to be a nullable integer
    type Returns<'a> = Option<i64>;

    // Perform initialization steps here
    fn init(cfg: &UdfCfg<Init>, args: &ArgList<Init>) -> Result<Self, String> {
        todo!();
    }
}
```

<https://mariadb.org/writing-user-defined-functions-in-rust/>

# User-Defined Functions

```
CREATE FUNCTION blake2b512 RETURNS string SONAME 'libudf_blake.so';
CREATE FUNCTION blake2s256 RETURNS string SONAME 'libudf_blake.so';
CREATE FUNCTION blake3 RETURNS string SONAME 'libudf_blake.so';

CREATE FUNCTION jsonify RETURNS string SONAME 'libudf_jsonify.so';

CREATE FUNCTION ip_validate RETURNS string SONAME 'libudf_net.so';
CREATE FUNCTION ip_to_canonical RETURNS string SONAME 'libudf_net.so';
CREATE FUNCTION ip_to_ipv6_mapped RETURNS string SONAME 'libudf_net.so';

CREATE FUNCTION lipsum RETURNS string SONAME 'libudf_lipsum.so';

CREATE FUNCTION uuid_generate_v1 RETURNS string SONAME 'libudf_uuid.so';
CREATE FUNCTION uuid_generate_v1mc RETURNS string SONAME 'libudf_uuid.so';
CREATE FUNCTION uuid_generate_v4 RETURNS string SONAME 'libudf_uuid.so';
CREATE FUNCTION uuid_generate_v6 RETURNS string SONAME 'libudf_uuid.so';
CREATE FUNCTION uuid_generate_v7 RETURNS string SONAME 'libudf_uuid.so';
CREATE FUNCTION uuid_nil RETURNS string SONAME 'libudf_uuid.so';
CREATE FUNCTION uuid_max RETURNS string SONAME 'libudf_uuid.so';
CREATE FUNCTION uuid_ns_dns RETURNS string SONAME 'libudf_uuid.so';
CREATE FUNCTION uuid_ns_url RETURNS string SONAME 'libudf_uuid.so';
CREATE FUNCTION uuid_ns_oid RETURNS string SONAME 'libudf_uuid.so';
CREATE FUNCTION uuid_ns_x500 RETURNS string SONAME 'libudf_uuid.so';
CREATE FUNCTION uuid_is_valid RETURNS integer SONAME 'libudf_uuid.so';
CREATE FUNCTION uuid_to_bin RETURNS string SONAME 'libudf_uuid.so';
CREATE FUNCTION uuid_from_bin RETURNS string SONAME 'libudf_uuid.so';
-- `bin_to_uuid` and `uuid_from_bin` are aliases
CREATE FUNCTION bin_to_uuid RETURNS string SONAME 'libudf_uuid.so';

-- `xxhash` and `xxhash64` are aliases
CREATE FUNCTION xxhash RETURNS integer SONAME 'libudf_xxhash.so';
CREATE FUNCTION xxhash3 RETURNS integer SONAME 'libudf_xxhash.so';
CREATE FUNCTION xxhash32 RETURNS integer SONAME 'libudf_xxhash.so';
CREATE FUNCTION xxhash64 RETURNS integer SONAME 'libudf_xxhash.so';
```

# Easy & Hard

The easy parts:

- Interfacing with the existing codebase
- Designing the APIs
- Using the Rust APIs
- Building and testing

The hard part:

- Understanding the existing APIs enough to define guaranteed behavior

# An Example with Encryption

- You need to read 10+ calls deep to figure out destination buffer size
- `encrypted_length` is sometimes ignored
- `src` and `dst` would, in some cases, overlap

```
/**
 * Returns the size of the encryption context object in bytes
 */
unsigned int (*crypt_ctx_size)(unsigned int key_id, unsigned int key_version);
/**
 * Initializes the encryption context object.
 */
int (*crypt_ctx_init)(void *ctx, const unsigned char *key, unsigned int klen,
                     const unsigned char *iv, unsigned int ivlen, int flags,
                     unsigned int key_id, unsigned int key_version);
/**
 * Processes (encrypts or decrypts) a chunk of data
 *
 * Writes the output to the dst buffer. note that it might write
 * more bytes than were in the input. or less. or none at all.
 */
int (*crypt_ctx_update)(void *ctx, const unsigned char *src,
                       unsigned int slen, unsigned char *dst,
                       unsigned int *dlen);
/**
 * Writes the remaining output bytes and destroys the encryption context
 *
 * crypt_ctx_update might've cached part of the output in the context,
 * this method will flush these data out.
 */
int (*crypt_ctx_finish)(void *ctx, unsigned char *dst, unsigned int *dlen);
/**
 * Returns the length of the encrypted data
 *
 * It returns the exact length, given only the source length.
 * Which means, this API only supports encryption algorithms where
 * the length of the encrypted data only depends on the length of the
 * input (a.k.a. compression is not supported).
 */
unsigned int (*encrypted_length)(unsigned int slen, unsigned int key_id,
                                unsigned int key_version);
```



# An Example with Encryption

## Module mariadb::plugin::encryption

[+] Expand description

### Enums

---

**EncryptionError** Errors returned by encryption operations

**KeyError** Error types returned by key managers

### Traits

---

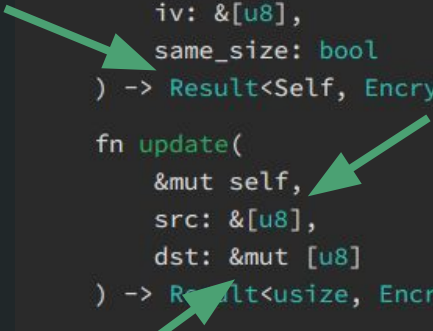
**Decryption** Decryption interface; implement this on decryption context

**Encryption** Encryption interface; implement this on encryption context

**KeyManager** A key management implementation with optional key rotation

# An Example with Encryption

```
pub trait Encryption: Sized {  
    // Required methods  
    fn init(  
        key_id: u32,  
        key_version: u32,  
        key: &[u8],  
        iv: &[u8],  
        same_size: bool  
    ) -> Result<Self, EncryptionError>;  
  
    fn update(  
        &mut self,  
        src: &[u8],  
        dst: &mut [u8]  
    ) -> Result<usize, EncryptionError>;  
  
    // Provided methods  
    fn finish(&mut self, dst: &mut [u8]) -> Result<usize, EncryptionError> { ... }  
    fn encrypted_length(key_id: u32, key_version: u32, src_len: usize) -> usize { ... }  
}
```



```
fn init(  
    key_id: u32,  
    key_version: u32,  
    key: &[u8],  
    iv: &[u8],  
    same_size: bool  
) -> Result<Self, EncryptionError>
```

[source](#)

Initialize the encryption context object.

Parameters:

- `key`: the key to use for encryption
- `iv`: the initialization vector (nonce) to be used for encryption
- `same_size`: if `true`, the `src` and `dst` length will always be the same. That is, ciphers cannot add additional data. The default implementation uses this to select between an AEAD (AES-256-GCM) if additional data is allowed, and a streaming cipher (AES-CBC) when the
- `key_id` and `key_version`: these can be used if encryption depends on key information. Note that `key` may not be exactly the same as the result of `KeyManager::get_key`.

# An Example with Encryption

```
45  /// Our encryption plugin will use the AEAD if a tag can be appended. Otherwise, the fallback is
46  /// the in-place streaming cipher.
    2 implementations
47  enum ChaChaCtx {
48      Stream(ChaCha20),
49      Aead {
50          cipher: ChaCha20Poly1305,
51          nonce: [u8; CHACHA_NONCE_LEN],
52          tag: [u8; CHACHA_TAG_LEN],
53          already_called: bool,
54      },
55  }
56
57  impl Encryption for ChaChaCtx {
58      fn init(
59          _key_id: u32,
60          _key_version: u32,
61          key: &[u8],
62          iv: &[u8],
63          same_size: bool,
64      ) -> Result<Self, EncryptionError> {
65          init_cipher(key, iv, same_size)
66      }
67  }
```

# Clevis & Tang

## Key Generation:

- Remote server (Tang) holds a key
- The client (Clevis or MariaDB) requests a public key
- The client performs ECDH to create an encryption key

## Key Retrieval:

- The client provides the key ID to Tang
- Encryption key can be recovered by performing ECDH with a short-lived key

Information is transmitted using JOSE standards

# Clevis & Tang

```
24
25 register_plugin! {
26     KeyMgtClevis,
27     ptype: PluginType::MariaEncryption,
28     name: "clevis_key_management",
29     author: "Daniel Black & Trevor Gross",
30     description: "Clevis key management plugin",
31     license: License::Gpl,
32     maturity: Maturity::Experimental,
33     version: "0.1",
34     init: KeyMgtClevis,
35     encryption: false,
36     variables: [
37         SysVar {
38             ident: TANG_SERVER,
39             vtype: SysVarConstString,
40             name: "tang_server",
41             description: "the tang server to use for key exchange",
42             options: [SysVarOpt::OptionalCliArg],
43             default: "localhost"
44         }
45     ]
46 }
47
```

## Trait mariadb::plugin::encryption::KeyManager

```
pub trait KeyManager: Sized {
    // Required methods
    fn get_latest_key_version(key_id: u32) -> Result<u32, KeyError>;

    fn get_key(
        key_id: u32,
        key_version: u32,
        dst: &mut [u8]
    ) -> Result<(), KeyError>;

    fn key_length(key_id: u32, key_version: u32) -> Result<usize, KeyError>;
}
```

# What is Working

- Encryption / Key Management plugin interfaces
- Function (UDF) plugin interfaced
- System Variables
- *sql\_service*
- Logging
- CMake build system integration (dynamic, not yet static)
- Unit testing within Rust
- Preliminary integration testing with mariadb-test

# What is Next

- Getting Clevis plugin to a shippable state
- Further plugin support:
  - Authentication
  - Data types (macaddr, macaddr8, cidr)
- Create Rust plugins without building against MariaDB

# Easier UDF Interfaces

```
1  #[mariadb::udf]
2  fn sample_regex<'a>(s: &'a str, re: &str) -> Option<&'a str> {
3      |
4      |     todo!()
5  }
```

```
// Specifying a name is optional; `#[register]` uses a snake case version of
// the struct name by default ('my_udf' in this case)
#[register(name = "my_shiny_udf")]
impl BasicUdf for MyUdf {
    // Specify return type of this UDF to be a nullable integer
    type Returns<'a> = Option<i64>;

    // Perform initialization steps here
    fn init(cfg: &UdfCfg<Init>, args: &ArgList<Init>) -> Result<Self, String> {
        todo!();
    }

    // Create a result here
    fn process<'a>(&'a mut self,
        cfg: &UdfCfg<Process>,
        args: &ArgList<Process>,
        error: Option<NonZeroU8>,
    ) -> Result<Self::Returns<'a>, ProcessError> {
        todo!();
    }
}
```

```
CREATE OR REPLACE FUNCTION plrust.one()
    RETURNS INT
    LANGUAGE plrust

AS
$$
let one_val = 1_i32;
log!("The plrust.one() function is returning: {one_val}");
Ok(Some(one_val))

$$
;
```



# Discussion Period

---