# Giving Rust a chance for in-kernel codecs

Daniel Almeida
Consultant Software Engineer
Collabora

COLLABORA

Open First

## Brief recap

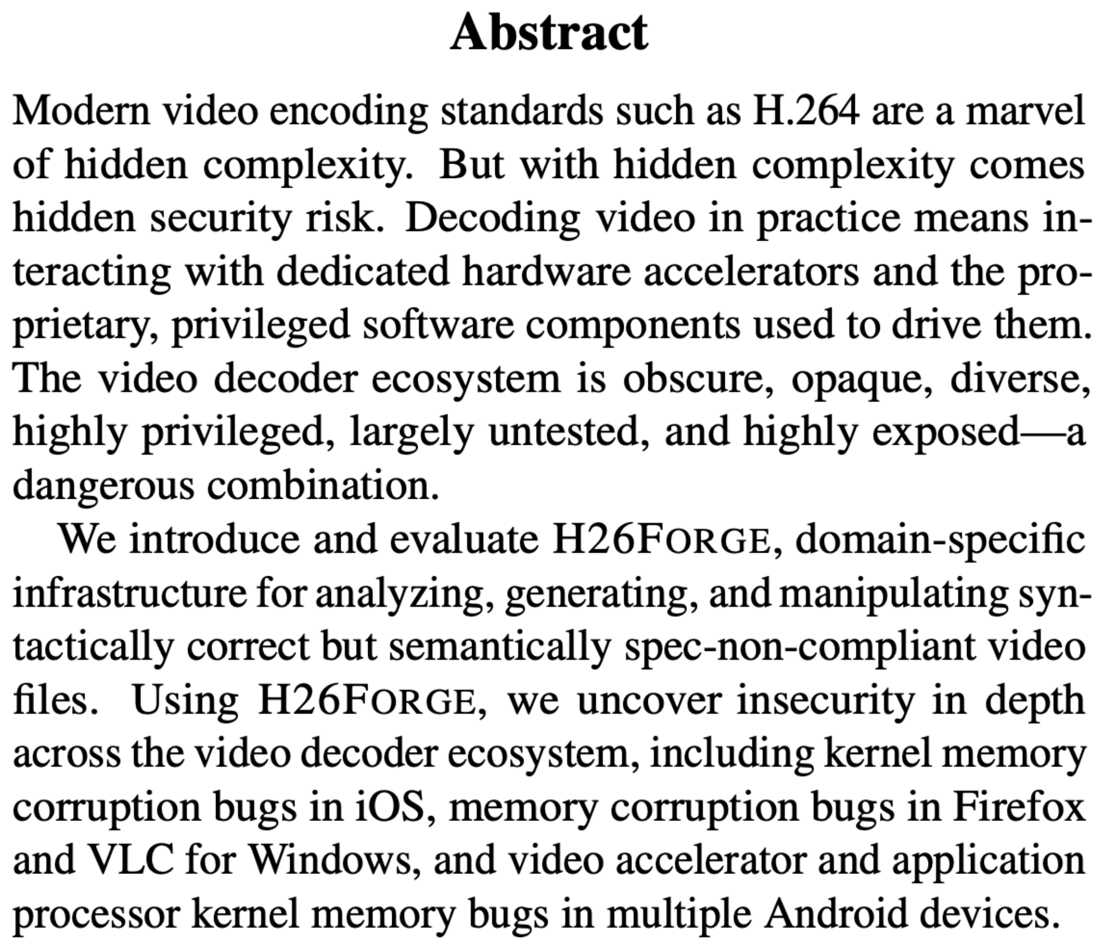
- Codec drivers ingest a lot of data from userspace
- Previous intentions were to write a driver
- A driver would need a layer of bindings

## The Most Dangerous Codec in the World: Finding and Exploiting Vulnerabilities in H.264 Decoders

Willy R. Vasquez
*The University of Texas at Austin*

Stephen Checkoway
*Oberlin College*

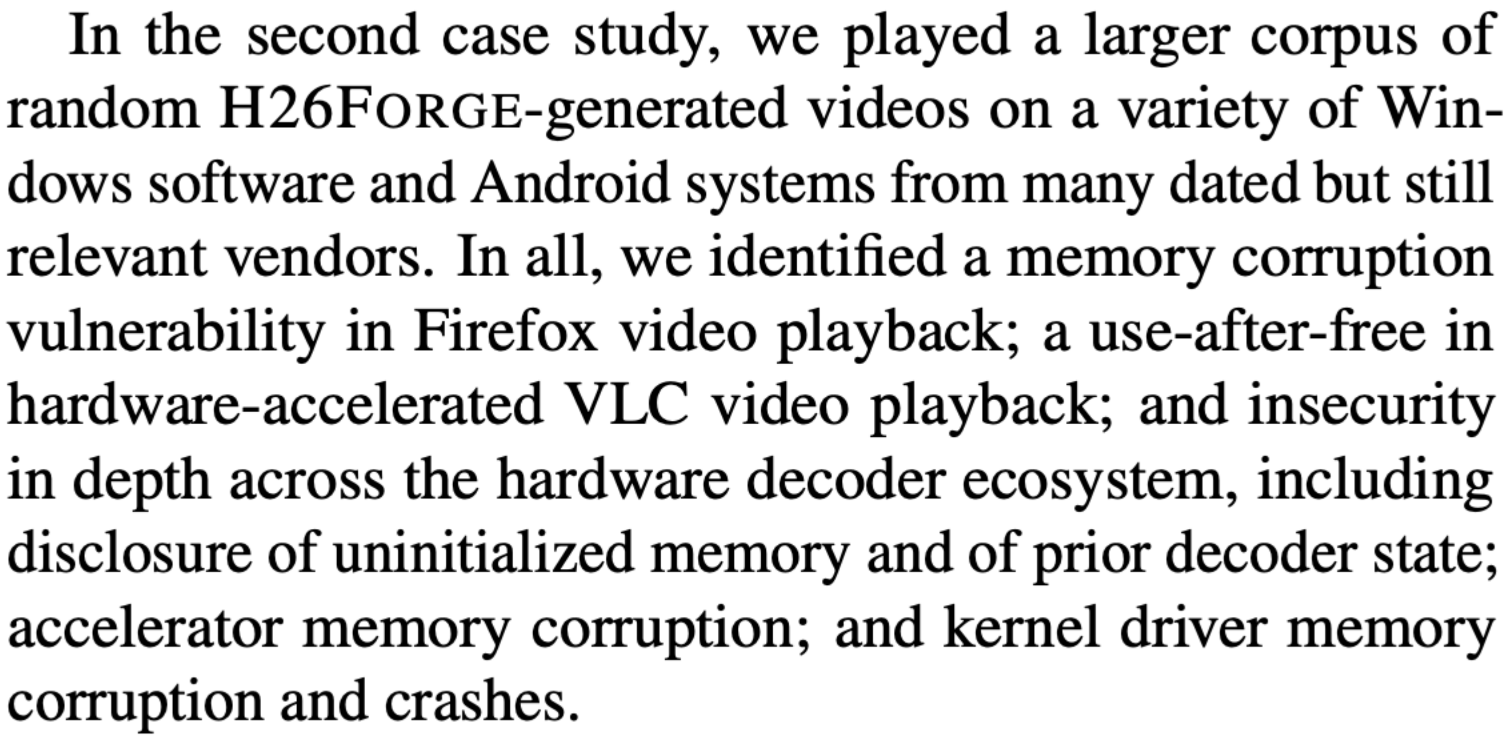
Hovav Shacham
*The University of Texas at Austin*



## Abstract

Modern video encoding standards such as H.264 are a marvel of hidden complexity. But with hidden complexity comes hidden security risk. Decoding video in practice means interacting with dedicated hardware accelerators and the proprietary, privileged software components used to drive them. The video decoder ecosystem is obscure, opaque, diverse, highly privileged, largely untested, and highly exposed—a dangerous combination.

We introduce and evaluate H26Forge, domain-specific infrastructure for analyzing, generating, and manipulating syntactically correct but semantically spec-non-compliant video files. Using H26Forge, we uncover insecurity in depth across the video decoder ecosystem, including kernel memory corruption bugs in iOS, memory corruption bugs in Firefox and VLC for Windows, and video accelerator and application processor kernel memory bugs in multiple Android devices.


In the second case study, we played a larger corpus of random H26Forge-generated videos on a variety of Windows software and Android systems from many dated but still relevant vendors. In all, we identified a memory corruption vulnerability in Firefox video playback; a use-after-free in hardware-accelerated VLC video playback; and insecurity in depth across the hardware decoder ecosystem, including disclosure of uninitialized memory and of prior decoder state; accelerator memory corruption; and kernel driver memory corruption and crashes.

## Feedback from last year

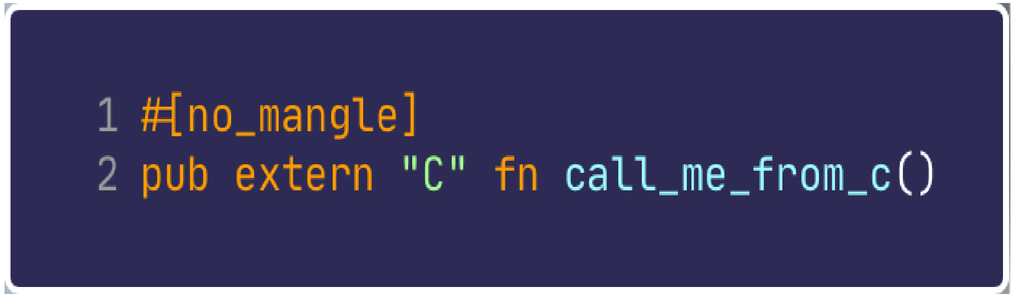
- Who maintains what?
- This will slow down development in C
- This may break C code
- The community is overwhelmed

COLLABORA

## What if we could write Rust code without bindings?

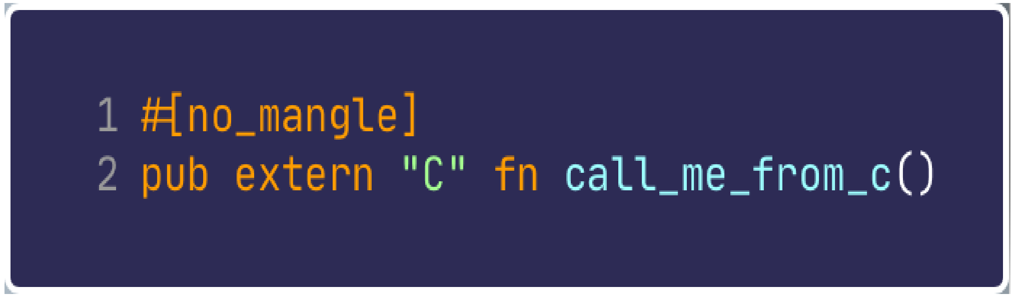
## We can do so by converting a few *functions* at a time

## This would sidestep most of the issues raised last year!



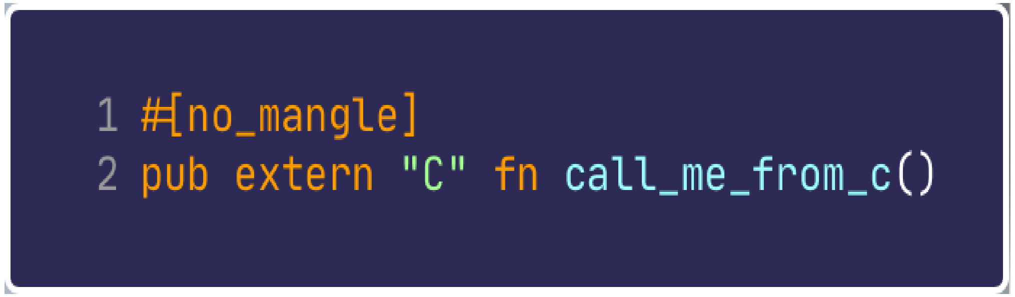
```rust
#[no_mangle]
pub extern "C" fn call_me_from_c()
```

- Generate machine code that can be called from C
- Make it so the linker can find it
- Can be used as an entry point to call other Rust code



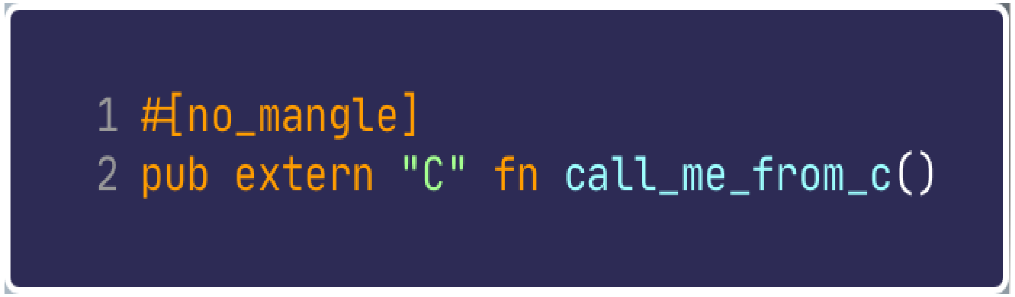
```rust
#[no_mangle]
pub extern "C" fn call_me_from_c()
```

- We don’t want this: _RNvNtCs1234_7mycrate3foo3bar
  - So no generics,
  - No closures
  - No namespacing
  - No methods, etc.



```rust
#[no_mangle]
pub extern "C" fn call_me_from_c()
```

- We need this to be callable from C, hence “extern C”
- Rustc will give us the machine code for the symbol.
- That’s it really, the linker will happily comply.



```rust
#[no_mangle]
pub extern "C" fn call_me_from_c()
```

- The **public API** is then rewritten as per above
- But we need a way to expose the new API to C somehow.
- Because...

```c
/* src/foo.h */
void call_me_from_c();
```

- This works.
- But it is not a good idea.
- It can quickly get out of sync.
- Nasty bugs can creep if we are not careful.

## No worries, there’s a tool

# Cbindgen

- Cbindgen can automatically generate a C header
  - Keeps things in sync
  - Ensure proper type layout and ABI
- Avoids link errors and/or subtle bugs
- Maintained by Mozilla

## Cbindgen

- If a function takes arguments, *cbindgen* will generate equivalent C structs
- This works because of **#[repr(C)]**

COLLABORA

## #include the header, that’s it.

## Potential targets

- This type of conversion works best when:
  - There is a self-contained component
  - That exposes a small public API
- For *video4linux*, this means:
  - Codec libraries
  - Codec parsers

## Codec libraries

- Codec algorithms that run on the CPU
- Results are fed back to the hardware
- Abstracted so drivers can rely on a single implementation
- Very self-contained

## Rewriting the VP9 library

- Two drivers were converted
- There is a testing tool
- **We got the exact same score when running the tool**
- Relatively pain-free process

COLLABORA

## New proposals:

# Proposals

- Merge the code
- Gate it behind a KCONFIG
- Users get the C implementation by default
- Run the Rust implementation on a CI
- Eventually deprecate the C implementation

COLLABORA

## Thoughts?

Thank you!