From CVEs to proof: Make your USB device stack great again

Ryad Benadjila[1], Cyril Debergé[2], Patricia Mouy[1], Philippe Thierry[1]
[1]firstname.lastname@ssi.gouv.fr
[2]cyril.deberge@irsn.fr

[1] ANSSI
[2] IRSN

Abstract. Nowadays, many devices embed a full USB stack, whose main components are made of software elements dealing with hardware IPs. USB sticks, hard-disk drives, smartphones, vehicles, industrial automatons, IoT devices: they all usually offer a USB physical connection, and a USB software driver dealing with it. In critical environments where attackers are able to tamper with this interface, any exploitable software Run Time Error (RTE) such as a buffer overflow might lead to a remote code execution on the vulnerable device, usually in privileged mode. This is even worse when the USB stack runs from a BootROM [12,45], yielding unpatchable software. This matter of fact exhibits the need for a portable RTE-free USB stack with concrete proofs: the current article proposes an open-source implementation of such a stack using the Frama-C framework [35], with proofs and various use cases (DFU, HID, mass storage, and more to come). Beyond providing the mere implementation, we bring a generic methodology to adapt complex protocols software stacks to Frama-C with strong embedded contexts constraints.

1 Introduction

Software is becoming the core component of many systems, from small embedded devices to bigger desktop Personal Computers. Even for what seems to be simple and low-level tasks, dedicated hardware with hard-wired logic circuits are almost always driven by pieces of software that tend to become more and more complex. From network cards to hard-drives controllers and motherboards chipsets, nowadays every piece of hardware contains firmware that is often made of thousands of lines of (usually) C or assembly code, let alone the drivers that interact with them on the Operating System side.

Vulnerabilities in such software stacks have proven to be critical from a security standpoint [3, 24, 27, 29]: due to the *bare metal* nature of such code, any Run Time Error (RTE) allowing remote code execution (RCE)

permits an attacker to gain control over a system usually at its highest privilege level, or at least compromises the confidentiality and integrity of sensitive user data. Examples of RTEs are (among others) buffer and integer overflows and invalid pointer access.

USB, a CVE minefield: An example of complex protocol that is becoming ubiquitous is USB. As a flexible and versatile bus, many devices offer a USB interface and must embed a stack handling it. A rather erroneous naive idea would be to consider that a USB stack is mainly made of hardware accelerated hard-wired blocks: this is far from reality as the public specifications [9, 19, 22, 33] expose abstract and portable automatons. The physical transport part of USB is usually implemented in a hardware Intellectual Property (IP), while the core, control and class handling parts are developed in software on top of this IP. This usually results in thousands lines of code for the core and control functions, and a few hundreds to few thousands more for each class depending on its complexity. As an example on the versatile desktop and embedded side, the Linux kernel `xhci.c` [1] Extended Host Control Interface (XHCI) stack is made of 3,747 sloc [3] of C in host mode in kernel release `5.11-rc7`. On the more constrained specific embedded side, STMicroelectronics USB device stack [4] targeting MCUs is made of 1,000 lines of C for the core, 600 lines for the CDC (Communication Device Class), 1,100 lines for the MSC (Mass Storage Class), 1,100 lines for the DFU (Device Firmware Upgrade) class and 250 lines for the HID (Human Interface Devices) class.

Although a few thousands lines of C code might seem rather "small", the very nature of the USB protocols makes them error prone and hard to implement: variable length fields to parse, generic types requests and asynchronous event-driven automatons leave plenty of room for vulnerabilities [29]. Among these vulnerabilities, Run Time Errors that could allow remote code execution should be avoided, but recent examples [12,45] show that even USB stacks developed with security in mind are not immune to such disastrous attacks. This is even more true when these stacks are implemented in a BootROM: no update is possible and the devices are doomed unpatchable. Even in less critical contexts, a vulnerability such as [46] (an exploitable buffer overflow on STMicroelectronics' USB stack on embedded MCUs) might prove hard to deploy in some situations, e.g. when implemented in on-field devices with complex physical or remote access.

3. Single lines of code.

RTE and formal guarantees: This state of affairs pushes the need for a USB stack with formal guarantees regarding security (no exploitable RTE), and ideally runtime behavior (concurrency and timing constraints) as well as functionality (i.e. USB specifications are correctly implemented). When it comes to catching RTEs, there are various paths to solve the issue. Although using safe languages is an interesting one, we have mainly focused on the C language because of its ubiquity even across exotic platforms, as well as its optimal volatile and non-volatile memory footprints (e.g. making it the *de facto* choice for BootROMs). Hence, we are mainly interested in formal guarantees on C code in the scope of this article.

Our contributions: In this work, we provide an open source C implementation of a USB 2.0 device stack with proofs against RTEs in sequential contexts and some functional guarantees that we will detail in section 5.2. The outcome is a RTE-free proven USB stack with limitations regarding concurrency and multithreading: although all the possible RTEs are formally not covered, we stress out that this is a big step forward when compared to the state of the art on such a code base.

For proving the stack, we have used Frama-C [35], an open source framework for C code analyzes which targets both academic and industrial use cases. Frama-C is well-tested on various projects targeting many platforms (from embedded to desktop contexts) and with various purposes (security or safety checks, verification of coding rules or browsing of unfamiliar code among others). This framework can be seen as an extensible collection of various tools (called plug-ins) working in a collaborative way on top of a shared kernel with compliance to a common specification language, ACSL [40]. The results from various Frama-C plug-ins are integrated by the kernel and given as input to the next analyzes with the remaining unproved properties.

The current article builds upon the results of a previous work using the same framework and proving the RTE-freeness of a X.509 parser [25]. Although similar techniques are used, attempting to prove a full USB 2.0 device stack brings its share of non trivial challenges and limitations that will be discussed in section 3.

Section 4 describes in more details the USB stack software architecture, chosen to fit with a modular proof strategy described in 5.1 where each module is independently proven with Frama-C. The proofs target the low-level USB HS (High-Speed) driver that has an adherence to the STM32F4 MCU family, the portable USB core and control module, as well as the MSC, DFU and HID classes. Despite some parts of the USB

HS driver are dedicated to a given IP, most of the USB stack is versatile and USB specifications centric: porting it to another MCU, SoC or CPU either in a bare-metal fashion or using an Operating System integration is simplified.[4] We stress out that we have chosen MSC and DFU classes as use cases examples due to their wide usage, sensitivity and the various CVEs jeopardizing these USB classes.

In order to validate our results we have used the proven USB stack in a concrete physical device: the WooKey platform [13]. The USB MSC, DFU and HID classes have been integrated to the existing SDK on top of the EwoK microkernel, without noticeable performance degradation when compared to the old USB stack of the project (while offering much more flexibility and versatility). Security gains and performance results are presented in section 6 along with the limitations and future outcome of our work in progress on the proof as well as on the development sides.

2 About security, RTEs and formal methods

2.1 Security and Run Time Errors (RTEs)

It is not an easy task to find a common definition of a RTE which is clear and precise but for some characteristics, a consensus exists. RTEs are errors encountered by a program when it is executed, which roughly corresponds to the *undefined behaviors* as defined in the C standards. For this work, we are compliant with the ISO C99 standard [26] and a description of the targeted RTEs is given in [10, 17]. Our security target is all the exploitable RTEs. Typical examples are buffer overflows when accessing non allowed memory, invalid pointer access, division by zero, signed integer overflow, and so on. Such "dangerous" behaviors sometimes provoke immediate crashes of the program, but they could also silently occur while corrupting the program's nominal intended functionality. In some cases, and beyond the mere crash and denial of service, such RTEs can lead to remote code execution (RCE) allowing attackers to take the full control of the faulty program. Of course, RTE-free does not mean bug-free: functional proofs ensuring that a program follows its formal specifications are beyond the scope of the RTE-freeness and must ideally also be covered. However, when focusing on the security of a program (written in C), RTE-freeness is a minimal requirement in order to ensure the absence of (or at least heavily limit) exploitable RCEs.

4. The choice of the STM32F4 family, a MCU built around a Cortex-M4 core, is mainly due to previous work on such microcontrollers.

We also want to emphasize the fact that by default the undefined behaviors as defined for the ISO C99 are caught by static analyzers, but other code defects (or dangerous patterns) which can also be considered as RTEs such as *unsigned integers overflows* can be unhandled. Such runtime errors are of course dangerous from a security perspective as they can lead to RCE or to an unexpected behavior. Consequently, one should activate the detection of such critical bugs using extra toggles (e.g. `-warn-unsigned-overflow, -warn-unsigned-downcast, -warn-signed-downcast, -warn-right-shift-negative` in the case of Frama-C).

2.2 Formal proofs and RTEs

The purpose of formal verification (for hardware or software systems) is, *in a few words*, to prove or disprove the correctness of these systems with a formal specification or any given property using formal logic. Formal verification is always associated to a given property (absence of RTE, respect of a functional property, etc.). One of the basic security expectations is the absence of RTEs as previously defined, which is a critical asset to prove when considering sensitive, complex and error prone pieces of software such as a USB stack. To reach this proof of RTE-freeness, a sound analyzer appears to be the appropriate approach. In [21], the characteristics of static analyzers to detect RTEs and this notion of soundness are described in more details. To make a long story short, a sound analyzer overapproximates all the possible executions and then, it will not miss any erroneous execution i.e. an execution with the violation of the proved property.

We have chosen to focus on Frama-C, an open-source platform and in particular the two well-known plug-ins EVA and WP dedicated to the formal verification. The combination of these two plug-ins has previously proved successful to ensure the RTE-freeness of a X.509 parser [25]. Other sound static analysis tools exist for C code such as Astrée [8], CodeProver [42], IKOS [16] among others. Such tools can be very powerful, but they are either commercial or less mature than Frama-C whose main advantage is (beyond its open source aspect) the possibility of combining the results of different formal methods through the usage of various plug-ins. Since our proofs are to be published along with the code, Frama-C fits our needs. As we aim at a generic proving strategy transferable to other embedded software stacks, our work must be reproducible on any piece of C code with the open source version `22/Titanium` of Frama-C.

2.3 Using Frama-C: EVA and WP

In [25], the authors expose how they have used the Frama-C framework, namely the EVA and WP plug-ins, to formally and automatically prove the absence of RTEs on a fully functional X.509 parser using dedicated annotations to ease the proofs. Since we build upon this work in this article, we briefly recall EVA and WP main characteristics and how they interact. The curious reader might refer to the original article [25] for more insights and details on these.

EVA: The purpose of EVA is to pinpoint the RTEs and to help the investigation of their cause. EVA [36] uses abstract interpretation [20], it automatically proceeds to a complete value analysis of the analyzed program and a set of RTEs are proven absent while some cannot be proven. The best advantage of using EVA is the low level of user interactions needed, the downside being it cannot discharge all emitted alarms because of the over-approximation of all the possible executions. In practice, such false positives have to be discharged manually one by one but this work can rapidly become cumbersome and very time-consuming, leading to WP usage as introduced hereafter. As far as we know, the largest system-level code proved with EVA is a scada system of more than 100 ksloc of C code used in nuclear power plants [44]: the coverage was about 80% with less than 100 remaining alarms.

WP: To avoid the tedious manual task of false positives investigation, WP [39] is used after EVA in our approach. In short, WP uses the proved properties with EVA and targets the remaining unproved properties. WP implements deductive verification calculus [23], a modular sound technique to prove that a property holds after the execution of a function if some other properties hold before it (i.e. function contracts expressed with pre/post conditions explained in section 2.3). WP is able to verify more complex logical annotations and assertions using external automated or interactive provers but requires extra user efforts with the code annotations including function contracts and especially loop contracts (a concrete example is provided hereafter when introducing the ACSL language). Indeed, without loop contracts, at each loop, WP uses an implicit specification which is equivalent to “anything can happen”.

WP is generally used to prove that the source code matches functional properties (its specification). It has been adopted by Airbus to verify safety properties of critical control-command code of avionic systems [15].

For security properties as the absence of RTE, the use of WP is more complicated for automatic separations because it requires a low-level memory model. WP proposes three main memory models: *Hoare* which provides automatic proofs but does not allow pointers, *Typed* which is a good compromise between expressiveness and automation but excludes casts, and *Typed+ref* that is more automatic than the later one but excludes aliasing. This last model is the one used in our case.

```
1 int i;
2 //@ loop invariant 0 <= i <= 10;
3 //@ loop assigns i;
4 //@ loop variant (10 - i);
5 for (i = 0; i < 10; i++) {
6   ++i;
7 }
```

Fig. 1. A loop contract example

ACSL annotations: As previously explained, additional annotations are necessary when using WP. These annotations are expressed in ACSL [28, 40], a formal specification language for C. It is a contract-based language designed for program proving and based on first order logic using pre/post conditions. A precondition is a property or predicate that must always be true just prior to the execution and the postcondition the property that must always be verified just after the execution. The ACSL language is close to C language with pure C expressions with specific but explicit keywords (`result`, `old`, etc.) and additional expressions dedicated to contracts: `requires`, `ensures`, `assigns` respectively for preconditions, postconditions or assignations (side-effects). These contracts can be instantiated for each function or loop (adding the `loop` prefix) with ACSL annotations added as C comments starting with `/*@` or `//@`.

A simple example of loop contract is provided on Figure 1. `loop invariant` is a condition that remains true during each iteration of the loop but *also just after* the loop exit, e.g. a loop counter remains between its bounds with the exit condition `0 <= i <= 10`. `loop assigns` specifies the modification of elements allocated outside the loop but modified inside. `loop variant` optionally provides a strictly decreasing non-negative integer value at each loop iteration (e.g. the difference between a maximum counter boundary and its current value) and is necessary to prove the loop termination.

3 From proving a X.509 parser to proving a USB stack

In this section, we discuss the stakes and challenges of proving the RTE-freeness of a USB stack when compared to more classical pieces of software such as a parser.

3.1 Feedbacks on proving a parser

The X.509 parser proven RTE-free in [25] is made of 5,000 sloc of C code with additional 1,200 lines of ACSL annotations (yielding a rate of 0.24 annotation per code line). One of the major idiosyncrasies of a code such as the X.509 parser are its inherent *sequential* aspect as well as its non-adherence to *hardware*: its execution is only dictated by software. All the possible states of the code are sequentially reached and only depend on the external API input (a buffer containing a certificate to parse), with no possible interruption of a function when it is executed. Hence, *concurrency* and *reentrancy* are not problems to address when proving that the code is innocuous regarding RTEs.

On the opposite side, a USB stack exhibits at least four issues:

1. It is mostly based on an asynchronous execution of events (except for some subparts such as the control plane handling that is executed synchronously). The Frama-C framework does not handle parallel units executions. Consequently, stacks requiring asynchronous events (backend responses, host messages reception or acknowledgement) are not easy to analyze without any modification.
2. Uncontrolled hardware registers and memory areas (C `volatile`) are the source of the previously described asynchronous events, meaning that some states of the program will be reached depending on variables whose values can be anything in wide ranges without more precision, yielding a combinatorial explosion of the static analysis and even more with sound static analyses which overapproximate all the possible executions.
3. The code base is substantially wider for the USB stack as it contains more than 12,000 sloc with different layers (the software architecture is detailed in 4). Soundness comes at the cost of analyzing all the execution paths with possible inputs, producing an amount of false alarms proportional to the code size. If the same one-piece proving strategy is used as for the parser, the analysis time will quickly diverge to unsustainable values. This is specifically striking

when a patch must be applied and the whole code base must be proven again. Moreover, since the USB stack is expected to be extensible through independent additional modules, we want to avoid to perform a very costly Frama-C proof analysis over all the modules each time a new one is added (while the others are untouched).

4. Contrary to regular code where only one `main` entrypoint is usually exposed (e.g. a `parse_certificate` function), the USB stack possesses multiple entrypoints that can be accessed. This is related to the concurrency aspect as multiple functions are designed to be called in parallel.

In addition to the previous items, the X.509 parser made use of a restricted subset of the C99 language, with no allocation (dealing with certificate size limit through statically allocated buffers of known fixed sizes) and a limited usage of function pointers. This last point was the major difficulty to deal with for the Frama-C platform and we use this feedback for the USB stack where function pointers usage is almost mandatory to have a flexible stack. In the case of the USB stack, we had to face additional difficulties. Dynamic allocation, although rarely used, can be necessary for some classes such as DFU which is a divergence from the parser that does not use it. This calls for some adaptations on the proof side. Beyond this, minimalism has been applied regarding other features of the language (no Variable Length Arrays, use of qualifiers such as `static` and `const`, strict compilation options, etc.).

These elements bring new challenges to tackle when compared to the work performed on the parser. These will be addressed in detail in section 5.2. The scope of the proofs and the implications are discussed hereafter.

Finally, it is worth mentioning that on the Frama-C learning curve side, we have strongly taken advantage of the experience on the parser: this led us to adapt our coding patterns and annotations as described in [25] to optimize the manual annotation work and the proof time.

3.2 The scope of proofs in this article

The previously described issues on the USB stack can be dealt with using *ad hoc* solutions. Dealing with the large code base is performed using a dedicated *modular proof strategy* between independent modules that we describe in 4. Dealing with the multiple entrypoints can be resolved using a dedicated code that emulates all the functions calls that are expected

(main code and interrupts) so that code coverage is maximized. Finally, dealing with the limitations of Frama-C regarding hardware interactions and concurrency calls for minor code modifications to enforce sequentiality where necessary (e.g. polling loops to avoid waiting states when a hardware interrupt is expected to trigger events) and deal with `volatile` variables.

Proofs on an equivalent sequential code All in all, the version of the code on which RTE-freeness proofs are ensured is no more the same than the running code: a sequential code ersatz is crafted by transforming some small parts to ensure attainable states during the analysis.

As we have already stated, a formal proof is only relevant on precise properties in a given context. In the scope of this article, when we talk about RTEs, we only consider those possible in a sequential execution of the code. Hence, all the RTEs that could be related to concurrency and race conditions are not covered. In the sequel of the article, RTE absence and RTE-freeness regarding the USB stack are shortcuts for *"absence of RTE in a sequential context"*, ruling out all the possible runtime errors related to other contexts.

Limitations for parallel code We try to limit the classes of bugs abusing race conditions, concurrency and reentrancy through a pragmatic methodology described in 5.2, but we are well aware that this does not stand up to formal guarantees as strong as the ones expected when talking about a (fully) formally proven code without RTE. Nonetheless, we claim that this is a first (big) step for proving such a complex code base yielding a strong warranty against a large class of exploitable bugs supported by the discovery (and patching) of classical RTEs during the development process as detailed in 6.1.

4 Architectural constraints and design overview

4.1 Overview of the software architecture

One of the big challenges when designing a portable USB stack is to limit the hardware specific part, and to allow enough flexibility for an integration in any context. It should be easy to integrate this stack in privileged mode with a rich OS such as Linux, in userland when there is an access API to low-level hardware, and in bare-metal embedded contexts. Memory constraints (both in volatile and non-volatile memory) must also be taken into consideration for a reasonable trade-off between portability

and versatility. We present on Figure 2 the software architecture of the USB stack we provide with the rationale behind our choices.

The standardized abstraction presented in the USB specifications [19] greatly spurs to isolate a hardware specific part. Usually, the existing hardware IPs for USB ❶ will expose some configuration registers, interrupt handling flags in event registers (usable in interrupt or polling mode), as well as endpoints related configuration and input/output streams. A classical way of representing USB endpoints in hardware are FIFOs: a memory mapped area is dedicated to various endpoints allocation through dedicated configuration registers for elements such as maximum packet length and other USB physical layer related items.

The software responsible for configuring and monitoring all these bare-metal elements is the USB driver ❷. In an interest of portability, even if the driver itself is not portable and adherent to a dedicated hardware, it exposes a USB-centric API ❸ to upper layers for stack initialization, endpoints allocation and manipulation (registering callbacks for sending and receiving packets on specific endpoints, Zero Length Packets, etc.).

The USB core ❹ is responsible for the enumeration and configuration phases automaton as well as the control requests on EP0 afterwards. It is the main part implementing the USB core automaton (from the powered state to all other possible states). After the enumeration, specific USB classes (mass storage, DFU, HID, modem, etc.) are instantiated (and other EP allocated) after the host has configured the device. The USB class module ❺ handles this part with a specific more or less complex automaton (depending on the class) managing the allocated endpoints through the driver API. This module exposes a USB-class abstract API ❻ to the upper layer ❼ that manages possible upper stack automatons, such as the CTAPHID one for FIDO/U2F and FIDO2 standards, and exposes its upper-class dedicated API ❽. Finally, backend functions ❾ have an abstract view of the USB stack and can use it to retrieve or send data to the host through either the class or upper-class APIs. Such functions can be a flash read/write backend for DFU (to fetch and update the firmware), (flash, SSD or solid state) sectors read/write backend for MSC SCSI, keyboard or mouse event handling for HID, two factor FIDO cryptography for CTAPHID, etc.

In the scope of this article, we mainly provide a *device USB stack*. All the layers provided in our implementation are portable across platforms except for the driver ❷ that is specific to the STM32F4 MCU USB IP ❶ [5], specifically on the WooKey platform [13] using its SDK. There are actually two drivers: one for the USB HS (High-Speed) IP, and one

for USB FS (Full-Speed) IP since the way hardware is handled in the two cases presents some variations on the STM32F4. The USB core ❹ is quite complete although integrating some advanced USB features is still work in progress. Finally, we have validated our stack with various classes ❺ (MSC, DFU, HID, CDC) and some upper-classes ❼ (CTAPHID), as well as different backends ❾ supporting these classes in the context of the WooKey platform (full DFU support for firmware update, fully functional mass storage device with transparent encryption, FIDO/U2F token, keyboard emulation, etc.).

On the Frama-C side, the USB HS driver ❷, the USB core ❹ as well as three classes ❺ (MSC, DFU and HID) are proven to be RTE free using the modular methodology we describe in 5.1. The main rationale behind choosing DFU and MSC as proof of concepts is the fact that they implement quite complex and error prone automatons as emphasized by many public CVEs [12, 31, 32, 45].

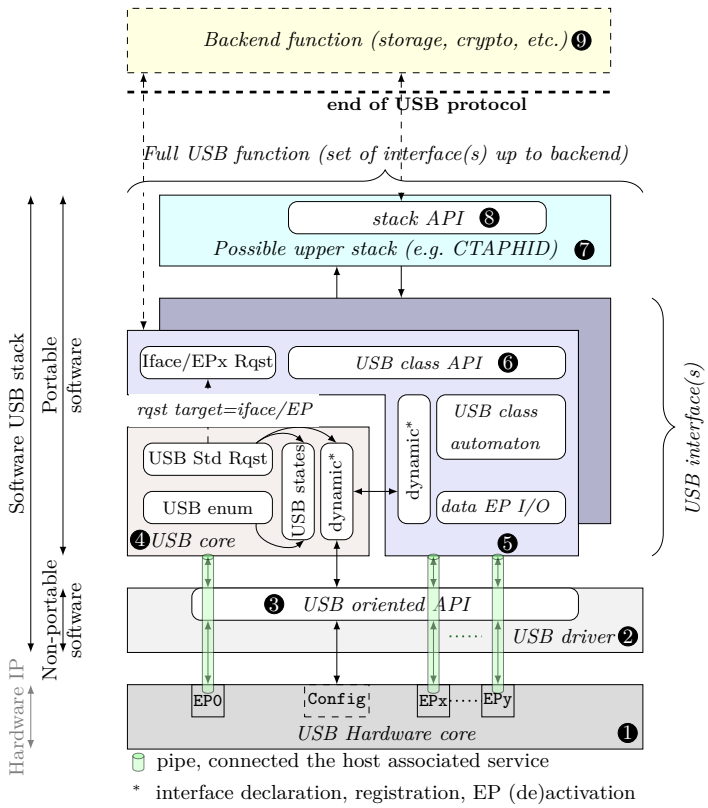

Fig. 2. Logical overview of the proven USB stack

4.2 A new USB stack in C: why?

Two rightful questions are to ask when considering formal verification and security insurance on a USB stack: why develop a new stack from scratch, and why use the C language?

Notwithstanding its imperfections and inherent security flaws [10], the C language combines performance (on both memory and CPU footprints) allowing embedding code in BootROMs, as well as portability across platforms: even though other languages support major architectures (e.g. thanks to the LLVM backends), more outlandish IPs might not have a compatible compiler or lack optimizations while a C compiler is (almost) always guaranteed. Moreover, C allows adding low-level countermeasures such as side-channel attacks and fault attack resistance: beyond RTEs, such attack contexts are relevant for many embedded use cases [30, 34, 43] and a C code base allows fine grain checks handling.

Our volatile and non-volatile memory footprint expectations rule out many existing large USB stacks. Various MCU manufacturers provide open source stacks [2, 4] to support their hardware and boards. Efforts are usually put on the portability side as they usually support various MCU families with different architectures. Some embedded RTOS (Real Time Operating Systems) also provide generic implementations of host and device USB stacks, such as Zephyr Project [6]. A major drawback of all these stacks though is the lack of defensive programming and security oriented development: some public CVEs exhibit exploitable buffer overflows [31, 32, 46], which implies too much work to converge to security proofs.

Very few USB stacks combine both movability across platforms and code simplicity, `TinyUSB` [7] is one of them. Although this could seem to be a good candidate for provability, such projects suffer from two issues. First, on the functionality side these stacks implement static interfaces and classes instances: the control layer is instantiated using fixed elements at compilation time (MSC, HID, DFU, etc.). The stack we bring provides more flexibility with dynamic descriptors handling while still ensuring memory-safety. Secondly, on the proof side many prevalent C coding patterns are found in such projects with large use - or abuse - of macros and undue volatile variables presence.[5] Such patterns do not comply very well with static analysis and formal methods assisted frameworks such as Frama-C. All in all, starting from an existing code base usually implies too much work when it comes to RTE guarantees. As we will develop it

5. The `volatile` related issues are thoroughly discussed in section 5.2.

in the next sections, intertwining functions implementations with their formal properties and contracts incrementally during the development process is the best strategy, with the benefit of progressively providing feedback and patches.

5 Let's prove the USB stack

Proving the USB stack using Frama-C requires many adaptations to overcome both the complexity of the code as well as the limitations of the framework regarding concurrency.

In the current section, we will expose our proving strategy (named *modular*) allowing us to deal with the large size of the USB stack by efficiently proving independent modules. We will also provide insights on how we handle sequential versus parallel code adaptations, and what is the divergence with the runtime code. Finally, we also discuss how entrypoints and external dependencies of the stack have been dealt with.

5.1 The proving strategy: a modular bottom-up approach

In the light of the constraints previously exposed, and using a divide and conquer logic, we have chosen a bottom-up approach for the proving strategy (on the sequential code transformed from the original runtime code) as it suits well with the vertical layers of the USB stack architecture. Each layer is proven RTE-free under the hypothesis that the underlying layer is also proven without RTE: the full stack is considered RTE-free since hypothesis are exhibited true layer by layer, and WP functions contracts are ensured on the APIs between layers. This is what we call "modular": it has the advantages of portability (modules hold their own proofs and are transferable) as well as keeping computational resources for running Frama-C reasonable (when compared to proving the whole complex stack in a row).

Taking the numbering from Figure 2, the driver ❷ is first proven to be RTE-free. Then, using its APIs with contracts on the (already) proven functions we perform proofs on the USB control module core ❹ that sits on top of the driver, and apply the same modular strategy on the MSC mass storage, DFU and HID classes modules ❺.

The same strategy moving from one unit to another would self-evidently hold for other classes, and for upper layer modules of the stack API ❼. Although this part is a work in progress and is beyond the sheer scope of this article, we stress out that the strategy we present alleviates the proving efforts through an almost mechanical methodology.

For each module, the steps to achieve a proof are described hereafter:

1. First of all, prepare the code to be analyzed by transforming it to sequential code as described in 5.2.
2. Secondly, prepare all the entrypoints to maximize coverage of the functions in the module, and deal with the external dependencies as described in 5.3. Maximizing coverage means that most of the code parts must be covered by the analysis (as reported by EVA).
3. Then, loop through the analysis of the code as described in detail on Figure 3:
   (a) Pre-process the source files with the Frama-C kernel to ensure there is no parsing error and that all the files are present;
   (b) Use EVA with default options, refine them to maximize code coverage while keeping enough precision (i.e. minimize false alarms for less manual analysis). Fix the found RTEs;
   (c) Following EVA's analysis, use WP to automatically check the remaining properties using automatic annotations left by EVA;
   (d) Add manual annotations (mainly function and loop contracts and assertions in the code) and adapt WP options so that the provers provide a result without any timeout.

5.2 Coding constraints and adaptations for the analysis

Making the code sequential: A first issue to tackle are polling loops on asynchronous events that would never terminate without a parallel execution. An easy way to handle this would be to remove the polling loop in the Frama-C execution context. This naive approach brings major drawbacks as the asynchronous event is still not executed and hence not analyzed, and its side effects do not propagate to the global state used after this waiting point. This is obviously diverging from the runtime behavior.

Another way to handle this quirk is to synchronously execute the asynchronous event instead of waiting for it. The perk of this solution is that the side effects of the asynchronous events are properly propagated to the current program execution context. On the downside though, we have modified the effective program execution by locally calling a potentially large routine, making the current function contract being modified in consequence. Figure 4 shows a typical code substitution due to the Frama-C constraints made in the USB MSC mass storage stack implementation.

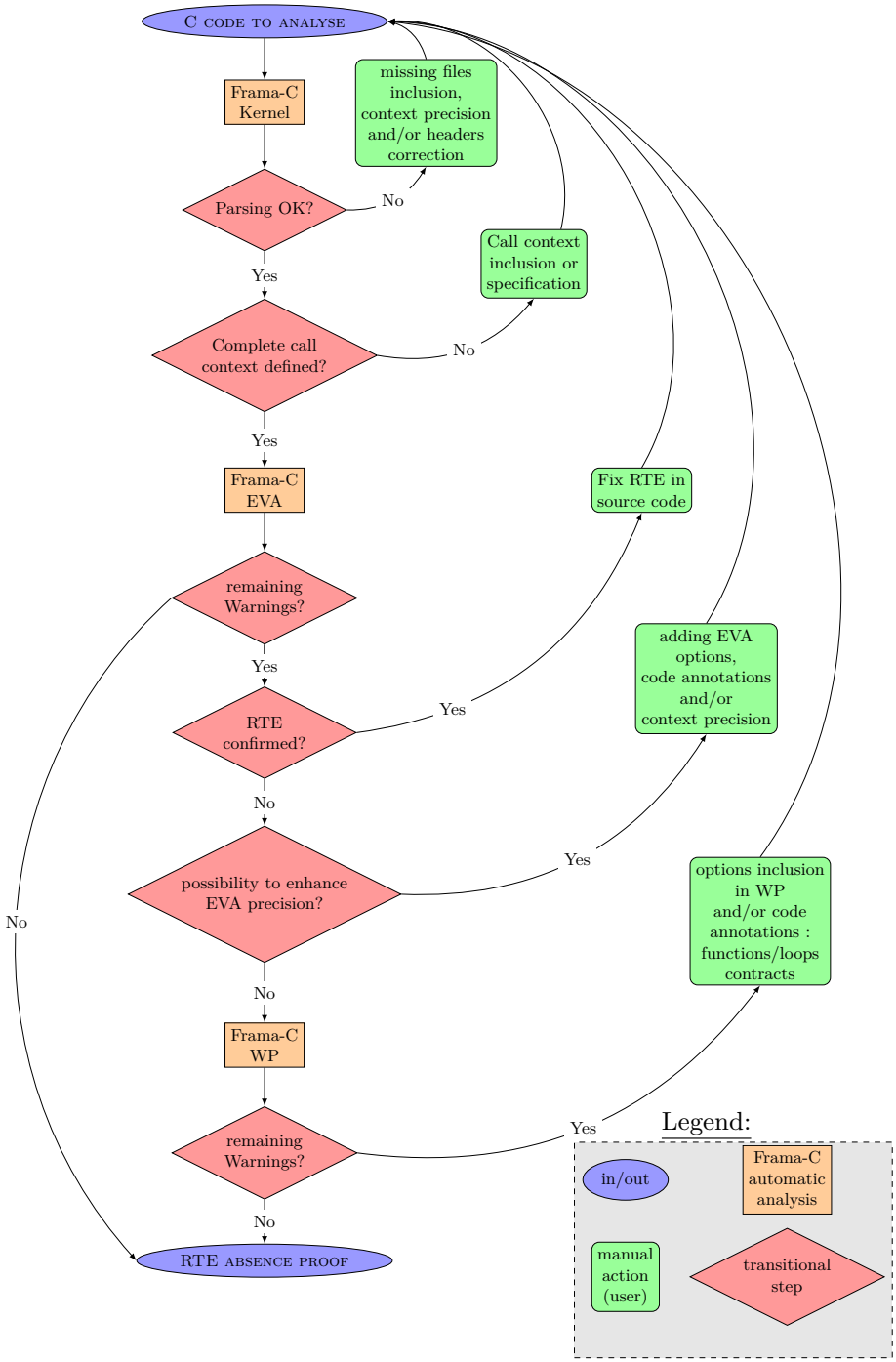

Fig. 3. Frama-C analysis strategy

```
1  // Active wait for data to be sent. Here, we wait for an asynchronous execution of a
       trigger setting the IN EP as ready. This trigger is scsi_data_sent(), which is
       executed when all the previously data configured to be sent has been transmitted to
       the host. Using FramaC, we cannot emulate multithreaded execution, so we
       synchronously execute this trigger, instead of waiting for its asynchronous execution
       .
2  #ifdef __FRAMAC__
3          if (!scsi_is_ready_for_data_send()) {
4              scsi_data_sent(); /* async event exec by the core */
5          }
6  #else
7          while (!scsi_is_ready_for_data_send()) {
8              request_data_membarrier();
9              continue;
10         }
11 #endif
```

Fig. 4. Handling asynchronous events in USB MSC class

Handling `volatile` globals: As Frama-C does not handle multithreading, impacts of functions side effects (typically global variables updates) during the execution of an asynchronous event cannot be taken into account in a direct way.

A basic approach would be to declare all global variables as `volatile` as most USB stacks do (like [7]). In this case, Frama-C automatically considers that their values are unstable[6] taking into account Time of Check Time of Use (TOCTOU) and race conditions risks.

Nonetheless, using `volatile` for all global variables (contexts, buffers and so on) is problematic:

— Volatile access is not optimized by the compiler (caching, access removed while optimizing, etc.), highly impacting the resulting performance and footprint.

— For most of these globals, values are relatively stable (fixed in time intervals) and their setting controlled, highly impacting the RTE validation by over-approximating the dynamic of the value (yielding uncontrolled computational time).

A better approach is to properly handle global variables by removing the `volatile` keyword, using *memory barriers* instead. The counterpart is that TOCTOU and race conditions are not highlighted by the framework. Moreover, all asynchronous-based RTEs might not be detected in a fully sequential analysis. A typical example is the usage of global buffers and callbacks that can be updated by another thread. A race between two threads may lead to an invalid behavior if a global variable is updated between its value checking and its usage, leading to a local TOCTOU. A

6. Frama-C does not assume that the value read from a `volatile` variable is identical to the last value written and all the possible values of these variables are considered.

way to avoid such RTEs is to observe specific programming constraints for global variables so that they can be updated concurrently:

— Any variable that is not polled for an event must be locally copied using atomic read instruction set, memory barrier and using a locking mechanism shared with other threads. These mechanisms are also required to avoid compiler optimization that may lead to threads desynchronization.
— Any check on a global variable must be performed solely on the local safe copy.
— Any usage of the global variable content (dereference, calculation) must be performed exclusively on the local copy.
— Any write back must be done with the hardware architecture atomic write instruction set, memory barrier, and potential locking mechanism shared with other threads.

Some specific cases exist though. Among all the globals, some of them should stay `volatile`. This is the case of hardware registers, FIFOs and so on, for which handling a local copy may lead to invalid functional behavior in comparison to the initial algorithm. For these specific cases, such variables are usually handled in a single function, reducing the risk for concurrency errors, although keeping reentrancy risks.

Aware that these solutions are disputable and may not be satisfactory from a formal point of view, we are seeking to reach higher guarantees on the provided USB stack: beyond Frama-C, dedicated static analysis tools could be of use. This is discussed in more details in 6.5.

5.3 Entrypoints and external dependencies

Entrypoints: The EVA plug-in needs an entrypoint to explore the code, which can be tedious for libraries such as the USB stack where there is no `main` function. Hence, an artificial `.c` file must be created for each proven module, with a `main` routine containing calls to all the functions that will activate the stack functionalities. It is important using these calls to ensure the exploration of the nominal code paths (with correct inputs) as well as the errors handling ones (with incorrect inputs). The code coverage provided by Frama-C is a good indication of the entrypoint completeness. The WP plug-in does not need entrypoints as the analysis can be performed on each function using the dedicated function and loop contracts to ensure proper behavior (pre/postconditions) and termination. Asynchronous calls to functions such as interrupts, handlers and callbacks can be executed at any point in time during the run time of another function.

External dependencies (assumptions): Although most of the USB stack code is self-contained, we use some functions that are external dependencies and that we consider as *valid under hypothesis*: we suppose that a RTE-free implementation of such functions is provided and used at link time in concrete projects. Such functions are classical and mainly taken from the standard library or compiler built-ins APIs: `memcpy`, `memset`, basic string manipulations with `strlen`, `malloc` and `free`. While the copy and string related functions are straightforward and quite easy to implement, we are well aware that proving the memory-safety of some complex algorithms (e.g. allocators) is not an easy task. Nonetheless, we underline the fact that such projects are side work, and some even already exist [28, 41] and can be used almost as is with our USB stack while inheriting from the proofs using modularity. Tuning the memory footprint and performance of such proven algorithms is out-of-scope of this article, and considered as future work.

WP cannot handle heterogeneous casts due to its memory model: functions with `void*` parameters have to be locally specialized with the type used by the callee with the associated function annotations to be efficiently used by WP. This means that functions with `void*` parameters and more precisely their contracts were locally specialized with the used type to be verified at each call site. We have to say that Frama-C has a dedicated plug-in, Instantiate [18] to do this work on the standard library (but only on it for now).

6 Results and discussion

6.1 Security gains

RTE and RCE related security gains: A first feedback is that developing the USB stack in parallel of using Frama-C [7] to prove it allowed us to find and patch vulnerabilities (and not only RTEs) in an incremental manner. This shows that our methodology thwarts many classical human mistakes when it comes to develop an error prone complex software stack. As examples of interesting findings, 10 RTEs have been discovered in the USB control core library, and 8 RTEs in the USB HS driver. Here is an overview of such bugs:

— Invalid memory access: Frama-C allowed to catch overflows when accessing interfaces descriptors and endpoints static tables in memory, possibly leading to memory leak (read sensitive areas) or RCE.

7. The term "Frama-C" is a shortcut for the described combination of EVA and WP.

— Unsigned integer overflows and unsigned integer downcasts: such RTEs can lead to incorrect behavior, or memory leaks and RCEs when the integers are used as access indexes in tables.
— Uninitialized variables: read and use an uninitialized memory area, leading to memory leak or unknown and incorrect behavior.
— Division by zero: divide by a variable that can be zero, leading to a denial of service.

```
1  mbed_error_t usbctrl_handle_class_requests(usbctrl_setup_pkt_t *pkt,
2                                             usbctrl_context_t   *ctx)
3  {
4    ...
5    /* Get interface from the packet index */
6    iface_idx = (((pkt->wIndex) & 0xff) - 1);
7    ...
8    /* Call our interface handler */
9    usbctrl_ctx[ctxh].ifaces[iface_idx]();
10   ...
11 }
```

Fig. 5. Detected RTE leading to RCE

In order to emphasize the benefits of programming with Frama-C, we provide a practical example of an *unsigned integer downcast* that has been detected by EVA and patched during the USB control library development cycle. This example is of particular interest as it could have led to a concrete RCE. In the USB stack, the `wIndex` field of control requests contains a target interface identifier strictly greater than 0 per USB specifications. The interface descriptors are stored in a C table with index starting from 0, and hence the addressing in the table was previously made using the formula computing `iface_idx` shown on Figure 5 and without boundary check at access time: a simple check on `wIndex` compared to the number of interfaces is not enough.

As one can see, a downcast is possible due to the minus one operation, producing `iface_idx=0xFF` when `pkt->wIndex=0`. The bad indexing will then call a function pointer in a corrupted interface structure `usbctrl_ctx[ctxh].ifaces[0xff]()` possibly controlled by the attacker, achieving the RCE. Although this particular case would have been caught by our defense-in-depth handlers sanitizers, catching such a RTE with potentially disastrous effects using Frama-C is gratifying. The patched code using the appropriate check and ACSL assertions on `pkt->wIndex` and `iface_idx` is shown on Figure 6.

Using Frama-C iteratively allows to naturally introduce defensive programming assertions when each alarm is treated, checked as truly positive RTE and fixed (as shown on the `wIndex` field example). When compared to known existing CVEs, immediate and straightforward RTEs

```
1  mbed_error_t usbctrl_handle_class_requests(usbctrl_setup_pkt_t *pkt,
2                                             usbctrl_context_t   *ctx)
3  {
4    ...
5    /* Get interface from the packet index with check */
6    if(((uint8_t)((pkt->wIndex) & 0xff)) == 0)
7      /*@ assert ((pkt->wIndex) & 0xff) == 0 ; */
8      errcode = MBED_ERROR_INVPARAM ;
9      goto err ;
10   }
11   /*@ assert ((pkt->wIndex) & 0xff) > 0 ; */
12   iface_idx = (((pkt->wIndex) & 0xff) - 1);
13   if (iface_idx > usbctrl_ctx[ctxh].num_ifaces) { ... }
14   /*@ assert (iface_idx < usbctrl_ctx[ctxh].num_ifaces ; */
15   ...
16   /* Call our interface handler */
17   usbctrl_ctx[ctxh].ifaces[iface_idx]();
18   ...
19 }
```

Fig. 6. Fixing the RTE

on the 16-bit field `wLength` of control requests [31, 32, 46, 46] (exploiting a buffer overflow for RCE) are trivially caught, and consequently prevented, by using our approach hence confirming its security gains.

As we have previously stated, elaborate RTEs that are consequences of TOCTOU, race conditions and concurrency are not captured at all using our current methodology with Frama-C, and our proofs of RTE absence on sequential code do not capture all runtime errors. Nonetheless, the observed results-oriented feedback reassures us on the practical usefulness of our approach. Many of the classical and dangerous bugs leading to RCE should be covered (with formal guarantees) which represents a notable positive leap toward a fully immune and bug-free USB stack.

Additional security gains: In addition to RTE findings and fixing, some common programming mistakes have been caught using Frama-C. Collateral RTE detection (bad `goto` labels usage) allowed to exhibit incorrect labels or missing `break` in functions. EVA code coverage feature allowed to detect dead code [8] or redundant tests, and hence perform optimization and code cleaning passes. As a side note, EVA coverage does not reach 100% of the library modules source code. There are multiple reasons for this. Some dead code is kept due to compiler's constraints: `default` fallback for `switch/case` structures must be present even though a previous piece of code inherently discards it (or else *warnings* are usually emitted by the compiler). Additionally to this, defensive programming against hardware fault injection contexts [43] inclined us to use code patterns that are legitimately considered as unreachable in nominal and safe execution paths of the program.

8. The term *dead code* is used here to refer to code which can never be executed.

6.2 Beyond RTE: functional verifications

Although RTE on sequential code is our major focus in the current article, we also seek some functional guarantees through elaborate function contracts for high level USB specifications conformity. In order to achieve this, we use the ACSL annotations to write function contracts that cover the functional elements we want to prove. We have first started doing it opportunistically on a few functions with simple specifications and then we have extended this process more systematically.

```
1  /*@ ...
2    // NOTE: USB 2.0 conformity: chap. 9.4.6
3    @ behavior invalid_pkt_windex:
4    @   assumes pkt->wIndex != 0;
5    @   ensures ctx->address == \old(ctx->address);
6    @   ensures \result == ERROR_INVPARAM;
7    @ behavior invalid_pkt_wlength:                    [...]
8    @ behavior std_requests_not_allowed:               [...]
9    @ behavior invalid_addr:                           [...]
10   [...]
11   @ complete behaviors ;
12   @ disjoint behaviors ; */
13 mbed_error_t
14 usbctrl_std_req_handle_set_address(usbctrl_setup_pkt_t
15                                   const * const pkt,
16                                   usbctrl_context_t *ctx)
```

Fig. 7. `usbctrl_std_handle_set_address` function contract

Since many of our functions, e.g. in the USB control module, implement the core USB specifications [19], we want assurance that the implementation does not diverge from our high level understanding of it. ACSL introduces the `behaviors` keyword that expresses possible executions of a function. Precise function contracts with various `behaviors` can be of great use and provide confidence that a logical ACSL description meets the C code. It is possible to form precise descriptions of each functional behavior of a function based on conditions for input and output values (pre/postconditions). An example of such a contract is shown on Figure 7 for the function handling the `set_address` phase of the enumeration between host and device, described in [19] chapter 9.4.6. Many `behaviors` are defined, each one providing a possible state depending on preconditions on the inputs, and leading to a deterministic postcondition `result` dictated by the specification. The first one reads (lines 3 to 6): when `pkt->wIndex` is not zero, this is considered as a request with invalid parameters and `ctx->address` must not be changed (equal to the previous `old` value before entering the function) while the returned `result` must be `ERROR_INVPARAM`. The notions of complete and disjoint behaviors are crucial: the statement `complete behaviors` means that our defined behaviors intend to cover all the possible states of the function so there is

no missing behavior, while the `disjoint behaviors` means that they do not share potential common states in function entry.

We are working on the systematic use of advanced contracts on critical parts of the modules that are strongly related to the USB specifications (other internal or simple functions usually do not need them).

Moreover, we have also began to add proofs on the finite state automatons by proving that the USB 2.0 control state automaton implementation is conforming to the specifications [19]: transition functions contracts map the automaton transitions model, proving the implementation adequacy. This work implies to locally verify all the transition functions by adding ACSL annotations where needed. For this purpose, we have used an additional Frama-C plug-in, MetACSL [37], that automatically generates all ACSL annotations corresponding to a high-level property (here the correct control state automaton implementation). This work is inspired by the results presented by the CEA-List team on proving the WooKey platform Bootloader correctness [38].

6.3 Overview of the proofs

We provide on Table 1 some statistics about our work on the USB stack with Frama-C (we use the same numbering for the modules as in the architectural view of Figure 2). The Table shows the modules that are already proven RTE-free.

For each proven module, we provide the C code sloc count as well as the manual ACSL annotations count: the ratio between them is an interesting metric to exhibit the necessary work on the Frama-C side to achieve the RTE-freeness. Another metric that we provide is a rough estimate of the amount of function contracts (WP columns in Table 1): simple contracts are those helping the RTE-freeness proofs on sequential code, while more elaborated ones provide functional guarantees on some of the modules sub parts. These amounts are obviously correlated to the ACSL annotation ratio.

First, the USB OTG HS driver in device mode shows a ratio of 0.24 annotation per C code line, the ACSL code mainly consists of contracts helping the RTE-freeness and ensuring that the hardware FIFOs and registers are correctly handled without touching other regions. This 0.24 ratio is on par with the annotations of the X.509 parser (as such annotations were also prioritizing the absence of RTE). The specifications related contracts are marked as Not Available (N.A.) since we do not have a specification per se for the driver: even if the datasheet [5] contains state automatons that could be translated to contracts, the hardware adherence

makes it impossible to formally check the state of memory and registers that are modified by the underlying IP.

Module	SLOC	ACSL	EVA coverage	EVA (RTE)	Functional simple (WP)	Functional spec (WP)	Proof time
❷ USB OTG HS driver	2,900	700	97%	100%	70%	N.A.	20m02
❹ USB control (DCI)	3,000	1,088	98.92%	100%	70%	50%	24m52
❺ USB MSC	2,900	614	98.95%	100%	50%	0%	18m17
❺ USB DFU	1,700	532	96%	100%	50%	0%	2h15
❺ USB HID	1,595	500	95.31%	100%	40%	10%	7m17
	Total						
	12,095	3,434					

Table 1. Overview of the current state of proofs on the USB stack

The USB control module presents a rather high ratio of 0.36 annotation per C line: this shows a particular focus on functional contracts on par with the USB core specification [19] similar to the example provided in 6.2. An important element to note here is that this module being *purely synchronous* by design (it fully runs in interrupt mode), the proof of the absence of RTE is immediately transferable to the *runtime code*. USB mass storage MSC, DFU and HID classes currently have lower ratios since the ACSL conformance to specifications [9, 22, 33] is to be refined. Our modules show a code coverage of nearly 100% (collected using EVA's coverage feature providing reached code lines percentage over all the analyzed modules), showing that almost all the functions and their corner cases are analyzed (with the limitations of legitimately unreachable code described in 6.1).

Finally, we provide for each module the total proof time using Frama-C version `22/Titanium`. These proofs are performed on github's cloud computing resources as we have integrated our stack development cycle to the github actions CI. The rather large computation time for DFU comes from its asynchronous automaton (when compared to the synchronous MSC one) and an external dependency to a memory allocator.

6.4 Performance and runtime tests of the USB stack

A statically proven USB stack with Frama-C would obviously be useless if it does not have a run time usage. In order to validate our device stack against various host stacks (Linux, FreeBSD, Windows, Mac OS), we have materialized it in concrete devices through the WooKey project [13]. This project aims at providing a SDK for applications development on top of a microkernel with defense-in-depth mechanisms. Since it natively provides mass storage and DFU features, it has been a natural playground

to replace the project's original and limited USB stack with ours. We have also integrated the newly developed USB HID and CTAP HID classes to the U2F2 FIDO token project [14].

Class	OS support	Throughput (Mbits/s)	ROM footprint [a] (Kbytes)	RAM footprint [b] (Kbytes)	Dynamic (reset, suspend, VM)
Original USB stack from the WooKey SDK					
MSC	Windows7+ Linux Mac OS	read: 52 write: 36	23	stack: 6 data: 25	No
DFU	Windows7+ Linux Mac OS	∼1.6 [c]	26	stack: 4 data: 12	
This work's USB stack compiled with the WooKey SDK					
MSC	Windows7+ Linux Mac OS	read: 50 write: 34.4	28	stack: 6 data: 30	Yes
DFU	Windows7+ Linux Mac OS	∼1.6 [c]	32	stack: 4 data: 16	

Table 2. Comparison: RTE-free stack versus original WooKey's stack

An interesting element to notice is that performance are preserved using our new USB stack integration, showing that defensive programming and Frama-C proofs do not noticeably impact CPU cycles and memory footprints. Table 2 summarizes some figures for both stacks with regard to the MSC and DFU classes. Throughput for MSC is measured in read and write directions using the `dd` tool under Linux, and using the `dfu-util` utility with 4096 bytes chunks size for DFU. Nominal functionality is stressed through harness file systems access (MSC), and multiple firmware updates (DFU). We want to emphasize the fact that figures provided here are not "absolute" but relative to the platform: they must be used to compare the two stacks, no more (some WooKey SDK elements and libraries beyond the mere USB stack are included in the footprints, etc.).

All the tests have been performed on the same WooKey hardware board, with the same SDK version and compiler. As we can see, the two stacks are both compatible with various OS hosts. The throughput is almost the same between them: RTE-freeness proof with Frama-C does not really have big impacts here. On the memory footprint side, we can observe that there are some differences: the one with proofs uses slightly more non-volatile flash memory and volatile SRAM memory. This is actually mainly due to increased features of the RTE-free stack (and not to extra

a. Mainly size in read only memory, could be flash or BootROM.

b. Size in RAM or SRAM, composed of the stack usage (local variables, functions frames) as well as writable data (initialized and uninitialized global and static variables).

c. Download mode only: throughput limited by the cryptographic tasks of WooKey.

code induced by proofs): more requests types and more automaton states are supported, producing a larger code and slightly decreased throughput. This work's stack also supports new features: properly handling USB reset and suspend events as well as Virtual Machines hot plugging and unplugging, dynamically configuring new classes, etc.

6.5 Limitations and future work

The path towards the RTE-freeness on runtime code: As we have seen, the proof done with Frama-C is somehow limited by two factors: the fact that the running code slightly diverges from the code parsed by Frama-C, and the fact that EVA and WP do not handle multithreading, concurrency and race conditions. This means that we cannot claim that our stack is absolutely proved, although our methodology paves the way for this goal and brings strong guarantees against Remote Code Execution (and against unexpected behaviors). As a matter of fact, our RTE-freeness proofs are on an equivalent sequential code that diverges from the runtime code, except for the USB control module that is *purely sequential* by nature. For the other modules, we have tried to heavily limit the code divergence between the part analyzed by Frama-C and the runtime code.

All the globals handling uses strict coding constraints detailed in 5.2 to limit too much usage of the `volatile` keyword. This coding pattern *immediately transfers to runtime* and does not incur a divergence on the absence of RTE proofs: concurrency issues are limited by protected accesses to variables, but reentrancy issues must be considered.

Regarding the divergence from runtime code, the only problematic patterns are those replacing polling loops on external events. Using synchronous calls as a replacement in the Frama-C version (see 5.2) are handled with care in each specific situation in the USB stack layers. We always try to ensure that no side effect (on a variable) could occur in the runtime version when compared to the analyzed one. This is usually easy since our polling loops are very simple. Table 3 presents such transformations per module: as we can see, this number remains controlled.

	HS driver	USB control	MSC	DFU	HID
#Transformed loops	3	0	9	6	4

Table 3. Number of analyzed versus runtime code divergence per module

These elements with their rationale provide a sound basis for transferring the proofs, although we are well aware of the (error prone) human

factor when dealing with them. Bringing proofs on a one to one code equivalent USB stack with the Frama-C framework, dealing with concurrency and reentrancy issues for all the modules (at least from a RTE-freeness perspective), is a non trivial yet very interesting subsequent work. Beyond the Frama-C framework, and in order to pragmatically deal with these limitations, we explore at least three complementary paths:

— Use fuzzing (e.g. using [47, 48]). Although this empirical approach has no formal foundations, it allows to dynamically stress software stacks and quickly detect bugs.
— Try to use other sound tools on the proven stack that could detect other kinds of bugs than RTEs, and/or deal with concurrency issues.
— Try to use other unsound tools to detect other bug classes.

Beyond RTE – functional proofs with Frama-C: Our (long term) ambition is to increase the ratio of functional proofs on our USB stack by using complete and detailed function contracts with WP. This will help to increase insurance in the implementation of the specifications. Beyond the work already performed on the USB control automaton, we seek to improve modeling and proofs on the class automatons such as MSC, HID or DFU. We are however well aware that the amount of necessary work is very variable depending on the considered USB class: while MSC automatons remain quite simple, DFU ones can be very tricky to apprehend.

Future development: On the development side, we seek to extend the USB control stack up to USB 3.2 specifications [11]. We are also in the process of increasing the supported classes (to CDC, CCID) with (sequential) RTE-freeness and functional proofs using the same modular methodology that has been exposed in this article. Beyond classes, we want to ideally expand this strategy to upper class modules, e.g. with CTAP-HID on top of HID for FIDO two factors authentication tokens. Also, our work has mainly focused on a *device* stack: an incipient *host* mode is already present in the driver and will hopefully be expanded to other layers. On the portability side, we seek to transfer the USB-centric layers (control plane and classes) on top of existing third party drivers (e.g. Linux kernel OTG and so on, despite their lack of proofs).

Finally, we also want to put some efforts to render the stack more robust against fault injections and glitch attacks, as hybrid (hardware and software) adversaries have recently proved efficient and relatively easy to achieve using cheap material [30, 34, 43].

7 Conclusion

This article presents how we provide an open source C implementation of a versatile USB 2.0 device stack with RTE-freeness proofs in a sequential context and some functional guarantees. The proof methodology uses a novel (as far as we know) composition tactic that became a necessity due to the complexity of the code, with a bottom-up (from hardware drivers to high level software) strategy.

We stress out that RTE-freeness is an important goal to achieve on crucial software stacks such as the USB one: this prevents dangerous CVEs [31, 32, 46] potentially leading to Remote Code Execution at the highest privilege level on vulnerable platforms. Our RTE-freeness proofs have been achieved on C code using the Frama-C framework EVA and WP plug-ins [36, 39] combination: this approach paves the way to a generic usage of the methodology on other projects, specifically those implementing advanced protocols stacks on embedded platforms. We also expose how we deal with Frama-C limitations regarding asynchronous events and strong hardware interactions such as interrupts.

To validate our results, we have ported the proven USB stack on the WooKey platform [13] that uses a STM32F4 MCU. Our tests expose similar performance and memory footprint when compared to the original USB stack of the project, showing an almost zero-cost effect of the RTE-freeness proofs and defensive programming. We have focused our efforts on the mass storage MSC, HID and DFU classes, but our future work in short term will consist in expanding this endeavor to proofs on the CDC, CCID and other USB classes, top layers, as well as integrating more functional properties with regard to the USB specifications. Other work in progress concerns the host mode development, portability to other hardware platforms, and compatibility with the USB 3 standard.

The limitations of using Frama-C in an asynchronous context with hardware interactions make the analyzed code diverge from the compiled one running on the target. This arises a legitimate question regarding the foundations of our absence of RTE guarantees when it comes to concurrency, race conditions and reentrancy in the multithreading runtime model. Section 5.2 provides key insights on how we try to empirically address these issues using simple and controlled code rewriting. Anyhow, the feedback on the concrete RTEs fixed during our development cycle indicates that many classical exploitable bugs have been caught. Fully bridging the gap between the proved and the executed code with Frama-C is a challenging task that we reflect on for future improvements.

References

1. Linux XHCI source code. https://github.com/torvalds/linux/blob/d8c849037d9398abe6a5f5d065eafc777eb3bdaf/drivers/usb/host/xhci.c.
2. MQX USB Stack. https://www.nxp.com/design/software/embedded-software/mqx-software-solutions/mqx-usb-host-device-stack:MQXUSB.
3. SoK: "Plug & Pray" Today – Understanding USB Insecurity in Versions 1 Through C. In *2018 IEEE*, San Francisco, CA.
4. STM32Cube USB library. https://www.st.com/resource/en/user_manual/dm00108129-stm32cube-usb-device-library-stmicroelectronics.pdf.
5. STM32F429/439. https://www.st.com/resource/en/reference_manual/.
6. The Zephyr Project. https://zephyrproject.org/.
7. TinyUSB. https://github.com/hathach/tinyusb.
8. AbsInt. Astrée. https://www.absint.com/astree/index.htm.
9. DEC Alps, Cybernet et al. Universal serial bus Device Class Definition for HID 1.11. In *USB Implementers' Forum*. sn, 2011.
10. ANSSI. Guide C. https://www.ssi.gouv.fr/guide/regles-de-programmation-pour-le-developpement-securise-de-logiciels-en-langage-c/.
11. Hewlett-Packard Apple Inq., Intel Corporation, et al. Universal serial bus 3.2 specification. In *USB Implementers' Forum*. sn, 2017.
12. Axi0mx. Apple iBoot BootROM DFU exploit, 2019. https://habr.com/en/company/dsec/blog/472762/.
13. Ryad Benadjila et al. WooKey: designing a trusted and efficient USB device. In *Proceedings of the 35th Annual Computer Security Applications Conference*.
14. Ryad Benadjila and Philippe Thierry. U2F2 : Prévenir la menace fantôme sur FIDO/U2F. In *SSTIC*, 2021.
15. Abderrahmane Brahmi et al. Formalise to automate: deployment of a safe and cost-efficient process for avionics software. In *EERTS 2018*.
16. G. Brat et al. IKOS: A Framework for Static Analysis Based on Abstract Interpretation. Springer, 2014.
17. CEA. RTE — Runtime Error Annotation Generation. https://frama-c.com/fc-plugins/rte.html.
18. CEA-List. Instantiate description page. https://frama-c.com/fc-plugins/instantiate.html.
19. Hewlett-Packard Compaq, Lucent Intel, et al. Universal serial bus specification revision 2.0. In *USB Implementers' Forum*. sn, 2000.
20. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.
21. Patrick Cousot et al. Varieties of static analyzers: A comparison with astree. In *TASE 2007*.
22. Ellisys Cypress, Intel Hagiwara, et al. Universal serial bus Mass storage class specification overview revision 1.4. In *USB Implementers' Forum*. sn, 2010.
23. E.W. Dijkstra. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". *ACM*, 1975.

24. Loïc Duflot et al. What if you can't trust your network card? In *RAID 2011*.
25. Arnaud Ebalard et al. Journey to a RTE-free X.509 parser. https://www.sstic.org/2019/presentation/journey-to-a-rte-free-x509-parser/.
26. International Organization for Standardization (ISO). The ANSI C standard (C99). http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf.
27. Matheus E. Garbelini et al. Sweyntooth: Unleashing mayhem over bluetooth low energy. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.
28. Jens Gerlach. ACSL by Example. https://github.com/fraunhoferfokus/acsl-by-example/blob/master/ACSL-by-Example.pdf.
29. NCC Group. Lessons learned from 50 bugs: Common USB driver vulnerabilities. https://research.nccgroup.com/wp-content/uploads/2020/07/usb_driver_vulnerabilities_whitepaper_v2.pdf.
30. NCC Group. There's A Hole In Your SoC: Glitching The MediaTek BootROM, 2020. https://research.nccgroup.com/2020/10/15/theres-a-hole-in-your-soc-glitching-the-mediatek-bootrom/.
31. NCC Group. Zephyr Project USB DFU buffer overflow, 2020. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-10019.
32. NCC Group. Zephyr Project USB MSC buffer overflow, 2020. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-10021.
33. Trenton Henry et al. Universal Serial Bus Device Class Specification for Device Firmware Upgrade. *Aug*, 5:47, 2004.
34. ITSEFs and ANSSI. Inter-CESTI: Methodological and Technical Feedbacks on Hardware Devices Evaluations. In *SSTIC*, 2020.
35. Florent Kirchner et al. Frama-C: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.
36. CEA LIST. EVA. https://frama-c.com/download/frama-c-eva-manual.pdf.
37. CEA LIST. MetACSL. https://frama-c.com/fc-plugins/metacsl.html.
38. CEA LIST. MetACSL Gitlab. https://git.frama-c.com/pub/meta/-/tree/master/case_studies/wookey.
39. CEA LIST. WP. https://frama-c.com/download/frama-c-wp-manual.pdf.
40. CEAL LIST. ACSL. https://frama-c.com/html/acsl.html.
41. Frédéric Mangano et al. Formal Verification of a Memory Allocation Module of Contiki with Frama-C: a Case Study. In *CRiSIS 2016* , 2016.
42. MathWorks. Polyspace Code Prover. https://fr.mathworks.com/products/polyspace-code-prover.html.
43. Colin O'Flynn. MIN()imum Failure: EMFI Attacks against USB Stacks. In *13th USENIX (WOOT 19)*, Santa Clara, CA, 2019.
44. Alain Ourghanlian. Evaluation of Static Analysis Tools used to Assess Software Important to Nuclear Power Plant Safety. *Nuclear Engineering and Technology*.
45. Kate Temkin. CVE-2018-6242. https://github.com/Qyriad/fusee-launcher.
46. Grzegorz Wypych. CVE-2020-15808: STM32FCubeMX exploit in CDC implementation. https://twitter.com/horac341/status/1311911734572208129.
47. Grzegorz Wypych. usb-tester. https://github.com/h0rac/usb-tester.
48. Michał Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl.