IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 1 November 2022, accepted 9 December 2022, date of publication 15 December 2022,
date of current version 29 December 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3229461

APPLIED RESEARCH

Safer Linux Kernel Modules Using the D Programming Language

CONSTANTIN EDUARD STANILOIU, ALEXANDRU MILITARU, RAZVAN NITU, AND RAZVAN DEACONESCU
Faculty of Automatic Control and Computers, University POLITEHNICA of Bucharest, RO-060042 Bucharest, Romania
Corresponding author: Razvan Nitu (razvan.nitu1305@upb.ro)

ABSTRACT Since its creation, the Linux kernel has gained international recognition and has been employed on a large range of devices: servers, supercomputers, smart devices and embedded systems. Given its popularity, the security of the kernel has become a critical research topic. As a consequence, a wide range of third party tools were created to detect bugs in its implementation. However, new vulnerabilities are discovered and exploited every year. The explanation for this phenomenon lies in the fact that the programming language that is used for the kernel implementation, C, is designed to allow unsafe memory operations. In this paper, we show that it is possible to incrementally transition the kernel code from C to a memory safe programming language, D, by porting and integrating a device driver. In addition, we propose a series of code transformations that allow the D compiler to reason about the safety of certain memory operations. Our implementation increases the security guarantees of the kernel without incurring any performance penalties.

INDEX TERMS Memory safety, Linux kernel, driver development, security, D programming language.

I. INTRODUCTION

One of the most popular operating system kernels, Linux, is used on a wide range of hardware, from supercomputers to IoT devices. While Microsoft Windows dominates the desktop market, Linux is the most popular operating system used by supercomputers [29], in the server market [31], handheld devices, as part of the Android operating system [27] and the embedded world [1].

Like all operating system kernels, Linux runs in a privileged processor mode (called *kernel mode* or *supervisor mode*) with complete access to system memory and devices. A successful attack on Linux will provide the attacker full control of the entire system, making it a sought after target. Such attacks represent a common occurrence. Figure 1 highlights the number of vulnerabilities discovered based on the Common Vulnerability and Exposure (CVE) reports [12]. The trend appears to be slightly decreasing, however, it still amounts to an average of roughly 250 reports per year. This number is extremely large, considering the years of manpower invested in securing the kernel. In addition, there is no way of knowing how many undiscovered vulnerabilities exist and are being actively exploited.

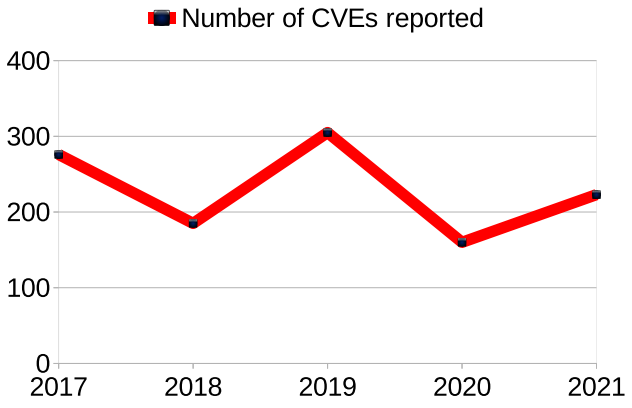

FIGURE 1. Number of Common Vulnerability and Exposure (CVE) reports.

To protect itself from potential security attacks, the Linux kernel employs a variety of self-protection mechanisms [10], [17] such as Kernel Address Space Layout Randomization (KASLR), Kernel Page Table Isolation (KPTI), stack protector etc.

The associate editor coordinating the review of this manuscript and approving it for publication was Alba Amato.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

Kernel self-protection mechanisms usually rely on enabling specific configuration parameters and adding runtime checks to prevent exploitation of code vulnerabilities. Vulnerabilities appear as a combination of programmer mistakes and lack of safety support from the programming language. The Linux kernel is mostly written in C, a fast programming language but with minimal safety features. C syntax allows easy access to the program memory such as liberal use of pointers, weak typing, no bounds checking for arrays etc. While these give flexibility to the programmer, they are also the main source of vulnerabilities: buffer overflows, pointers to expired data, pointers to uninitialized memory etc.

In this paper we propose a complementary approach to securing the Linux kernel: the use of a safe programming language, i.e. a language with features that assist the developer in writing secure code.

Our choice is the D programming language [5], that has a syntax similar to C/C++ and provides modern programming and safety features. D aims to provide as many of the performance benefits of the C programming language, with as few of the security downsides as possible.

With the goal of porting a Linux kernel module to the D programming language, we answer the overarching research question: *Can critical software components (operating system drivers) be rewritten in a safe programming language with reasonable effort while maintaining performance?*.

Rewriting a software component from an older language to a newer one offers the possibility to use more modern programming features. In our case there are safety benefits such as: array bounds checking, immutable variables, safe functions, guaranteed initialization. At the same time, the translation process poses multiple challenges. Firstly, each feature in the initial programming language has to be available in the new programming language; if not, it has to be adapted. Secondly, the newly rewritten software component has to be built and linked against the main program: symbol names, calling conventions, memory references have to be compatible. Thirdly, dependencies of the newly rewritten software component, such as its runtime library, have to be added to the new program or need to be disabled.

Additionally, the Linux kernel adds its own challenges. Certain features such as a standard C library or the use of floating point are missing. Memory allocations are typically resident in the Linux kernel. The stack size is limited.

While we also considered Rust and Go as programming languages for the Linux kernel port, we ultimately chose D. Our choice of D was based on three criteria: syntax similarity to the C programming language, interoperability with C programs and high performance generated code. D fitted these criteria, with its close syntax to C, its reasonably easy interoperability[1] with other languages and its proven track record of generating code that is on par with C's performance.

[1]The D primary compiler (DMD) has a feature called `-betterC` enabling it to build a C program with some D features.

We selected a Linux kernel driver (*virtio_net*) and ported it successfully in the D programming language. The ported driver benefited from the safety features of the D programming language, improving its security: bounds checking, safe functions, templates. The performance costs were negligible.

In summary, in this paper we make the following contributions:

- We demonstrate the feasibility of using a modern programming language in the Linux kernel by successfully porting a Linux kernel module to the D programming language. We ported *virtio_net*, the network driver of the *virtio* framework [11].
- We design and implement techniques that rely on specific D language features in order to improve the Linux kernel drivers. The performance costs are negligible with the security benefits being provided by the D programming language.
- We provide a methodology for porting Linux kernel modules to the D programming language. Demonstrated by our successful port, the methodology can be used to port other Linux kernel modules.

The rest of the paper proceeds as follows. Section II details the D programming language and Linux kernel specifics. Section III presents the methodology employed for porting Linux kernel modules to D. Section IV presents the concrete steps and challenges in porting the *virtio_net* Linux kernel module. Section V evaluates the security benefits and performance costs for the ported module. Section VI presents related work. Section VII concludes.

II. BACKGROUND

A. LINUX KERNEL MODULES

Linux source code consists of the kernel proper and a plethora of device drivers and configurable components. Loading a Linux kernel image with all the device drivers included will result in unnecessary memory consumption and an increase of the attack surface. For this, similarly to other modern operating systems, Linux uses *loadable kernel modules*, i.e. object files that can be added to the kernel at runtime to extend its functionality. Kernel modules can be loaded or unloaded upon request, without the need to reboot the system or to recompile the kernel.

Device drivers are typically implemented as kernel modules. On a given system, only drivers for its particular set of hardware devices will be loaded in the kernel. The loading of these specific device drivers usually takes place at startup.

Past studies have shown that device drivers host security vulnerabilities. Johnson et al. have found that 9 of 11 vulnerabilities in the Linux kernel located in device drivers [9]. An investigation using Parfait, a C/C++ static analyzer, has found that 81% of the bugs are located in device driver code [4]. A two year investigation has revealed that 85% of Android kernel bugs are found in vendor drivers. As such, this paper focuses on securing a device driver by porting it to the D programming language. We use the *virtio_net* Linux kernel module as a proof-of-concept of our approach.

B. THE D PROGRAMMING LANGUAGE

D is a general-purpose, statically typed, systems programming language. It has a similar syntax to the C programming language and it compiles to native code, i.e. it is not interpreted nor does it use a virtual machine. D supports both automatic and manual memory management: one can rely on the garbage collector (GC) for memory management or directly use the `malloc` and `free` functions for manual allocation and deallocation of memory, similarly to C.

D is designed as a more feature rich and safe alternative to the C programming language. It aims to create programs with comparable performance to those written in C but without the safety issues of it. D provides a set of features aimed at reducing the likelihood of memory issues and vulnerabilities typically found in C programs.

D implements bounds checking for both static and dynamic arrays. To address the C design flaw of conflating pointers with arrays and losing the length information, D implements two separate types for pointers and arrays. While the normal pointers have the same implementation as in C, the arrays are implemented as *fat pointers*: the pointer representation is extended to a structure that includes length information used in bounds checking.

In D, the type system is more stringent and void pointers are not implicitly converted to other pointer types. Moreover, local variables marked with the `scope` keyword are limited to the function scope, reducing the presence of dangling pointers.

Besides common pointers such as those found in C, D provides a memory-safe option called *slices*. A slice acts as a "view" of a precise segment of an array. It tracks both the pointer and the length of the segment. Instead of referring an array through a pointer that may cause an out of bounds memory access, one can use a bounded slice.

D offers the `@safe` annotation for functions. This enables the compiler to statically check the body of annotated functions for instructions that could lead to memory corruption such as pointer arithmetic and casts. By default, D relies on the GC to safely manage the lifetime of objects. Although the GC has proven to aid productivity and memory safety, its use is incompatible with performance critical or real-time applications such as the Linux kernel.

As a consequence, an advanced user has the possibility of opting out of using the GC and using a different approach for lifetime management. Among the possible alternatives are reference counting or the Resource Acquisition Is Initialization (RAII) technique. As an alternative to reference counting [16], the language maintainers have added support for an ownership/borrowing system [7] that can be mechanically checked, similar to Rust's borrow checker. At the time of this writing, October 2022, D's ownership system is not on par with Rust's, but it is under active development.

We note that the garbage collector is not involved in any of the safety checks that the compiler employs, apart from lifetime management. Array bounds checking, compile time safety checks and scope analysis are performed even when the GC is turned off.

Another important part of the D language are its metaprogramming features. Template metaprogramming is a technique that allows the user to make decisions based on the template type properties. This technique makes generic programming even more powerful, allowing generic types to be more flexible based on the types that they are instantiated with. We have used metaprogramming to employ compile-time polymorphism inside the Linux kernel in order to replace the use and casts to/from `void*` with concrete types.

C. INTERFACING C WITH D

Regarding interoperability with C, the D programming language was designed to match most of the C data types, data structure memory layout and calling convention. Moreover, the compatibility extends to the format of the object files. D and C use the same application binary interface (ABI) and the same linkers. D permits access to the C standard library through bindings in the D runtime library and the D standard library; similarly, C programs can access D functions. Due to name mangling, C functions called in D need to be declared with the appropriate linkage attribute (`extern "C"`); similarly, D functions called in C code are prepended with the same linkage attribute. This is identical to the integration of C++ functions in C code and viceversa.

Linking D code to a C program relies on restricting D objects only to the C standard library. D-generated object files can be linked to C-generated object files by restricting D code to a subset that is not reliant on the D runtime library. This is achieved through the `-betterC` compiler switch that limits the language to a specific subset that meets the foregoing requirement. This subset, called *BetterC*, results from removing or altering certain features of the language that rely on the runtime library. While some important functionalities, such as garbage collection, are removed, most relevant memory-safety features are preserved. Array bounds checking and slicing, metaprogramming facilities, automatic initialization of local variables, function safety are part of the *BetterC* subset.

D. INTEGRATING D CODE IN THE LINUX KERNEL

We ported *virtio_net*, the network driver of the *virtio* framework.

While C and D integration of user space applications is a well documented process, integrating D code in the Linux kernel poses its own set of challenges. To the best of our knowledge, we are the first to have successfully integrated a D software component in the Linux kernel.

In the next sections we highlight the challenges, methodology and outcomes of integrating D code in the Linux kernel.

III. METHODOLOGY FOR PORTING AND ENHANCING KERNEL MODULES USING D

A. INTRODUCING D CODE IN THE LINUX KERNEL

There are two ways of adding new functionalities to the Linux kernel: (1) statically linking the new object file directly with

the core kernel or (2) compiling the code as a loadable module and linking it into the kernel on demand.

A general rule of thumb is to add new functionalities as a loadable module. This practice has the advantage of keeping the kernel code as clean as possible and is easier to maintain. Also, it permits customization to a greater extent, as necessary functionalities can be loaded and unloaded on demand. Moreover, it keeps the Trusted Computing Base (TCB) small and reduces the overall susceptibility to compromise, thus increasing security.

Regardless of the type of module that has to be built, the kernel build system assumes the source files are written in C. As such, a source file written in another programming language won't successfully compile and the build will fail. This is also the case for the D language. At the same time, the module entry point and exit point functions must be in C, so that the kernel can reach them. Summarizing, porting a module to the D programming language requires:

- writing the corresponding source code in D
- providing module entry points as C interface functions
- updating the build system files to link the new module

For the 2nd requirement, a C interface must be implemented between the kernel and the D-written module. This C interface should contain only the entry point functions and bindings to macros and functions that can not be ported to D. This interface will imply that new features will require at least two source code files: one in C and the ones in D. Therefore the directives in the Linux kernel build file must be written accordingly, for the 3rd requirement.

The kernel build system assumes that it is dealing with C source files and it tries to build the object files accordingly. Fortunately, the build system also accepts pre-built object binaries, as dependencies, that it will link with the object files it built in order to create the kernel module. This is done by changing the name of the dependency from *module-file.o* to *module-file.o_shipped*. To link D object files into a kernel module, the D source files must be compiled beforehand and have their name with the suffix *.o_shipped*. The source files will be compiled by a D compiler with the `-betterC` switch. One can choose between using the LLVM-based D Compiler (LDC) and the GCC-based D Compiler (GDC). After they are compiled independently, they will be shipped to the kernel build system to be linked together with the other C objects.

B. PORTING A KERNEL MODULE

Porting the kernel module, we followed 5 steps, including testing and benchmarking:

- Port the data structures used inside the module. Ensure the size and layout of each new ported structure is identical to the size and layout of the original one.
- Port the module implementation one function at a time. Check module functionality after each new ported function.
- Conduct the first set of benchmarks: assess the module behaviour. Compare the D and the C versions of the module.
- Introduce D idiomatic constructs and features into the code. Add bounds checking, replace macros and casts with metaprogramming, add `@safe`, `@trusted` and other useful features.
- Perform the second set of benchmarks: assess the effect of the idiomatic code added. Compare the idiomatic D and the rough D versions of the module. Compare the D and the C versions of the module.

The first step, the porting of data structures, is the most complex one. In a kernel module, some structures are defined inside the code of the module, while others come from different header files. To be able to generate an object that can pass and receive structures from a C program, a D compiler (like any other compiler) must know the layout in memory of those C structures. This means porting them to D.

This porting can be done using *dpp* [28], "a compiler wrapper that will parse a D source file with the *.dpp* extension and expand in place any *#include* directives it encounters, translating all of the C or C++ symbols to D, and then pass the result to a D compiler". However, a high level of branching in header files or recursive inclusions may lead to the impossibility of using *dpp*. In this case, one has two alternatives: (1) port the data structures by hand or (2) make *dpp* work with the Linux kernel headers. We chose the former.

Regardless of the porting method, the size and layout of each new structure ported to D should be compared with the size and layout of the original one from C. In the case of a size or layout mismatch, the bug can be easily detected by comparing the offsets of the fields from D with the offsets of their C counterparts. In D, the offset of a field can be obtained using the *.offsetof* field property. In the Linux kernel, it can be obtained using the *offsetof(TYPE, MEMBER)* macro.

A difference to consider is the size of an empty structure: the C kernel size of an empty structure is 0, while in D this kind of structure has the size of 1 byte. We used D's powerful compile-time introspection to solve this issue. Also, one should consider the fact that the D language does not implicitly support bitfields. However, the same functionality can be achieved using the `std.bitmanip.bitfields` library type.

While porting the implementation, the D functions called from C must be annotated with the *extern(C)* linkage attribute. The attribute instructs the linker to use the C naming and calling convention instead of the D one. The same must be done when declaring, in the D header, a function that is implemented in C.

In D, the non-immutable global variables are placed in the thread-local storage (TLS), while in C they are placed in the global storage. To achieve functional parity, one must annotate D global variables with the *__gshared* attribute. Also, the *const* qualifier is transitive in D, meaning that it applies recursively to every subcomponent of the type that it is applied to.

Primitive data type equivalence can be problematic too. The equivalence between basic C and D types is described in [6].

Not all the functionalities that are used or implemented in a kernel module are worth to be ported. This is the case of certain macros, which in their turn call other macros and so on and are very deeply rooted in the kernel code. It is also the case of certain kernel functions that use GCC features that extend the standard C language and which may not be implemented in the D compiler. A way to avoid the porting these macros or functions is to create C bindings (functions that only call other functions), that can be exposed to a D object and called from there. These bindings should be created in the C interface of the module.

After each new ported function, a functionality test suite should be run. If bugs were introduced, there is only one function to debug. The process of porting should be more syntax-oriented in the first two steps of the methodology. One straightforward way of solving syntax related issues is to follow and solve the errors that are issued by the compiler. On the other hand, step 4 should be more oriented towards functionality and one should use all the features that the *BetterC* subset retains, in order to improve the safety and the performance of the module. Several techniques for enhancing the safety of a module are presented in the next section.

The benchmarks (steps 3 and 5) should be done according to the module functionality. As a rule of thumb, a benchmark should be done after the module is ported (step 3) to assess if the D version of the module can "keep up" with the C version. Then, one should take into account that memory safety features can lead to further performance penalties. Safety checks are likely to introduce additional overhead. The second benchmark (step 5) should be done to assess if the addition of idiomatic code and safety features is worthwhile from a performance perspective.

C. SAFETY ENHANCEMENTS

These are some of the security enhancements provided by the D programming language. They are used to implement and build the newly implemented kernel module in D.

1) VARIABLES

are initialized to a default value of their type, removing initialization bugs.

2) IMPLICIT CONVERSIONS

of void pointers to any other pointer types are not permitted. D requires an explicit cast for converting pointers of different types.

The C **implicit switch fall-through** behaviour is not permitted in D. D also uses the `final switch` statement where the default case is not required nor permitted, useful when the `default` statement is useless. The `final switch` statement is especially useful when it is applied on an `enum` type, as it will enforce the use of all the enum members in the `case` statements.

3) STATIC ARRAYS

are by default bounds-checked.

4) SLICES

specify a part of an array, via a reference and length information. They are used to bounds-check dynamically-allocated arrays. Note that this requires knowledge of the initial size of the dynamically-allocated arrays.

5) TEMPLATES

can be used as replacement for C void pointers and macro definitions for generic programming, thus enabling type system checks.

6) SAFE FUNCTIONS

(annotated with `@safe`) are statically verified against cases of undefined behavior. Within safe functions, there are several language features that cannot be used, such as casts that break the type system or pointer arithmetic.

Scope, **return ref** and **return scope** function parameters are used to ensure that parameters do not escape their scope, do not outlive their matching parameter lifetime and are correctly tracked even through pointer indirections.

7) TRUSTED FUNCTIONS

(annotated with `@trusted`) provide the same guarantees as a safe function, but checks must be done by the programmer.

8) SAFE FUNCTIONS

can only call other safe functions and trusted functions.

IV. METHODOLOGY IN ACTION. THE *virtio_net* DRIVER

Given the steps described above, the goal was to select and port a Linux kernel driver from C to D. This was an iterative process with the methodology being updated with feedback from the porting process.

To select a target driver we considered the following criteria:

- The driver is in the Linux kernel mainline and it is maintained, so it is relevant for the kernel community.
- The driver is easy to test and benchmark: being a network driver, one can easily send and receive packets and measure what bandwidth is achieved.
- The driver should be medium-sized (thousands of lines of code). This is a nice trade-off between feature complexity and porting effort.

Based on these criteria, we selected the `virtio_net` driver, part of the *virtio* framework [11]. As it's name suggests, it is a virtual network device driver, used as a communication channel between the guest and the hypervisor in a paravirtualized environment. It satisfies the three criteria: (1) it is actively maintained and used for virtualization use cases, (2) it can be easily tested with network tools: network functionality and network metrics such as bandwidth and latency can be part of a comparison evaluation process and

(3) it has roughly 3.3k lines of code, fitting into the medium-size range we wanted.

The Linux kernel version used, and the compatible driver, was 4.19.0. For development, testing and evaluation we used a virtual machine (VM) based on QEMU.

V. EVALUATION

To validate our approach we show that:

1) The D code has the exact same behavior as the C code that it replaces.
2) The safety mechanisms inserted successfully prevent the occurrences of memory corruption bugs.
3) The performance of the replacement software does not degrade with regards to its predecessor.

We created a setup where we provide both implementations of the *virtio_net* driver (C and D) and ran similar scenarios to compare functionality, safety and performance.

A. EXPERIMENTAL SETUP

We created a virtual machine image with the 4.19.0 version of the Linux kernel. The virtual machine is run as two instances: one running the C version of the *virtio_net* driver, and the other one running the D version. We refer to the virtual machines using *guest* and the physical system using *host*.

We compiled the D source files of the module using the GDC compiler, version 10.3.0, with the following flags: `-fno-druntime -mcmodel=kernel -O2 -c`.

For evaluation we focused on functional correctness / parity, safety and performance.

B. FUNCTIONAL CORRECTNESS

We then run network tools in each virtual machine to check for parity of functionality. For example, using `ping` to validate functionality, using `wget` to download information from the Internet. Additionally, we check whether the transferred file is the correct one by comparing its MD5 hash with the expected one.

C. SAFETY

To enhance the safety of the ported driver code we modified the code as to use several D language features: array bounds checking, `@safe` functions and templates.

1) ARRAY BOUNDS CHECKING

The *virtio* driver uses both statically and dynamically allocated arrays. In the case of static arrays defined inside the driver, the D language compiler has sufficient information at compile time to insert bounds checking code. Dynamic arrays, on the other hand, are represented in C as a pointer to a chunk of data, therefore there isn't sufficient information at compile time to offer the possibility of implementing runtime checks. However, using slices, we are able to enable bounds checking for dynamic arrays that are defined inside the ported driver. Accesses to arrays that are dynamically allocated outside the driver remain without bound checks.

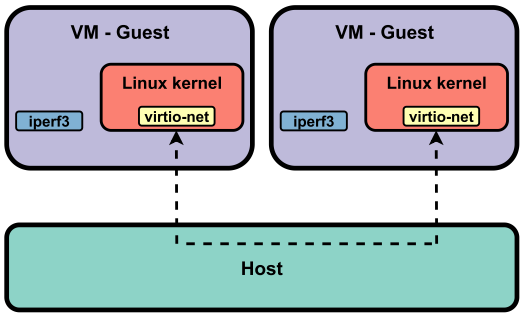

FIGURE 2. VM to VM setup. One VM runs the iperf3 server, the other is running the client.

From the total number of array accesses inside the *virtio_net* driver, we were able to enable array bounds checking in 88.4% of the cases. The rest of 11.6% represent accesses to dynamic arrays that have been allocated outside of the ported driver. To test the effect of adding array bounds checking on the driver, we have added artificial out of bounds accesses to the code. In 60% of the cases, the C version of the driver has finished execution gracefully, whereas the D version has stopped with a kernel panic in 100% of the cases.

2) @SAFE FUNCTIONS

To enable the D compiler to check the safety of the code, we aimed to annotate all the functions present in the driver with the *@safe* keyword. 19% of the functions have successfully compiled without any modifications, whereas 81.2% have failed compilation due to performing unsafe operations. Most of these functions rely on pointer operations and casts that are forbidden in *@safe* code. Additional modifications are required to bring the code in a *@safe* state, however, this can be done incrementally after the initial port of the driver.

3) TEMPLATES

D code may use templated functions that are instantiated at compile time with the right type. In case of a type mismatch, that will result in a compilation error, thus making it impossible to have runtime memory corruption bugs. By using templated functions, we replaced 56% of the total number of *void* pointer usages. The remaining 44% could not be replaced because there was no conversion pattern that we could detect and leverage for our transformation.

D. PERFORMANCE

For performance, we used the *iperf3* tool that sends packets between a client and a server. We used a virtual machine instance running the original C version of the *virtio_net* driver and a virtual machine running the D version. Each VM was allocated 1GB of RAM and 1 CPU. *iperf3* was deployed on both VMs.

We devised 3 setups:

- **vm-to-vm** (in Figure 2): One VM is running the server, one VM is running the client. Both machines are of the same type: either C and either D.

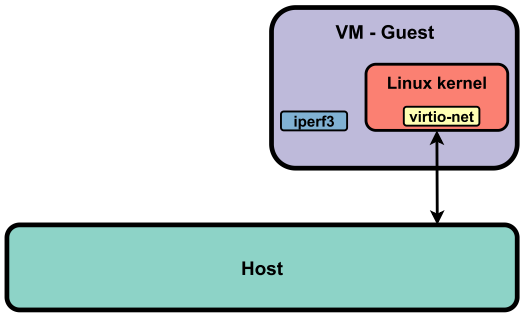

FIGURE 3. VM to host setup. The host is running the iperf3 server, the VM is running the client.

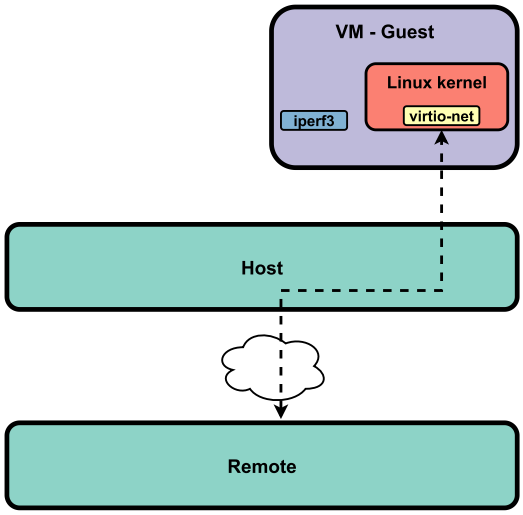

FIGURE 4. VM to remote setup. Another system in the host network is running the iperf3 server, the VM is running the client.

TABLE 1. Comparative performance.

		C	D	Slowdown
TCP (Gbps)	vm-to-vm	2.662	2.642	0.88%
	vm-to-host	2.447	2.502	-2.24%
	vm-to-remote	0.934	0.935	-0.1%
UDP (pkts)	vm-to-vm	87677	85285	2.72%
	vm-to-host	151873	152253	-0.25 %
	vm-to-remote	135606	135698	-0.06%

- **vm-to-host** (in Figure 3): The host is running the server, the VM is running the client.
- **vm-to-remote** (in Figure 4): Another system in the host network is running the server, the VM is running the client.

Each of those setups was used for 2 × 2 types of measurements: (1) the VM is running D **or** or the VM is running C and (2) *iperf3* is using TCP or it is using UDP.

Results are summarized in Table 1 and in Figure 5 and Figure 6.

Results show negligible overhead for the D module implementation compared to the C implementation. Given that parts of the measurements show a negative slowdown,

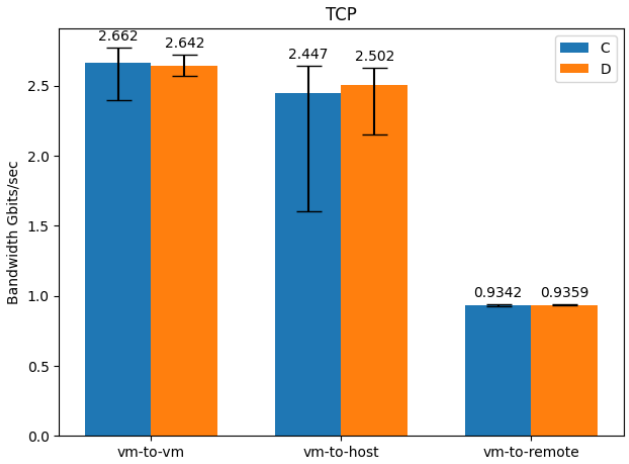

FIGURE 5. Comparative TCP Performance (C vs D).

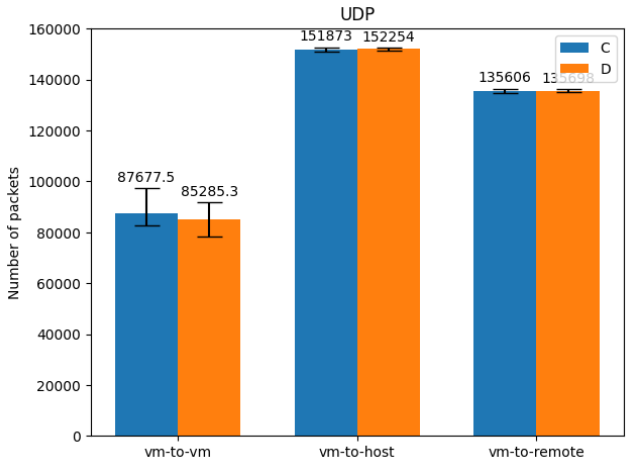

FIGURE 6. Comparative UDP Performance (C vs D).

we consider performance similar and subject to network and measurement variation.

One thing to note is the relatively reduced impact of the changes: the basic network driver functionalities are unmodified, most of the code responsible for that being shared between the two implementations. Porting other drivers may affect a larger part of the implementation and could feature a higher slowdown. This is subject for analysis in the future.

E. REPLICABILITY

In the interest of the validating our work, we provide it to the community on GitHub as a fork of the Linux kernel, an implementation of the D `virtio_net` driver and experiment scripts: https://github.com/edi33416/d-virtio.

The implementation of the D `virtio_net` is on the `test_dvirtio_gdc` branch, in the `drivers/net/dfiles` folder. Alongside the `.d` source files present in the `drivers/net/dfiles` path, there are also a `Makefile` and two `test.sh` files. The `Makefile` is used to compile the *.d* source files into the `.o_shipped` objects that, in turn, will be linked by the kernel build system to build the

virtio_net.ko module. The test.sh and test2.sh helper scripts are used to validate the experimental setup. They load the compiled kernel module, configure the IP address and routing table, and validate that the network is working properly; this is done by downloading a file and comparing its *md5sum* with the reference value.

In order to be able to compile the D driver, one needs to install gdc-10, the GCC based D compiler. As we are using QEMU to run the VMs, one also needs to ensure that it has installed qemu-system-x86_64 with KVM support. We have been connecting to our VMs using a serial port with a serial communication program, such as *Minicom*.[2]

Once all the prerequisites are met, one can build the kernel module, boot-up the VM and start using the compiled driver. This process is automated in the tools/labs/ directory. The tools/labs/Makefile, through the run target, will (1) compile the .o_shipped object, (2) trigger the kernel build system that will result in the virtio_net.ko module, (3) download the YOCTO_IMAGE specified in the tools/labs/qemu/Makefile and boot-up the VM, and (4) copy the module inside the VM. It will also setup IP forwarding and NAT Masquerading for the eno1 network interface on the host machine, so one must update the Makefile if one's system is using a different network interface name.

Once the VM has booted, one can connect to it through the serial1.pts serial pipe with the help of the minicom utility tool, as such minicom -D serial1.pts. The default login username is root and requires no password. All the files can be found in the skels/ directory inside the VM, the kernel object being named virtio_net_tmp.ko. Precompiled .ko objects can be found in the *Releases*[3] on Github:

- Precompiled D .ko: https://github.com/edi33416/d-virtio/releases/download/dvirtio-ko/virtio_net_tmp.ko
- Precompiled C .ko: https://github.com/edi33416/d-virtio/releases/download/cvirtio-ko/virtio_net_tmp.ko

It is our hope that the availability of our work will make it easier to evaluate, to replicate and to provide a critical eye on.

VI. RELATED WORK

Improving the safety of the Linux kernel and its drivers is the constant focus of the professional and research security community. There are different approaches ranging from static analysis of the Linux kernel code [4], [9], [14] to fuzzing [3], [8], [22], [25] to the use of runtime checks and/or instrumentation [13], [24].

The idea of using programming languages that implement different memory safety features in order to make the Linux kernel code safer has also been tackled.

The recent availability of Rust as a programming language in the Linux kernel [19], [20] paves the way for adding code written in a secure programming language. This is compatible with our own approach of using D to write code in the Linux kernel. Although the memory safety guarantees that Rust offers are superior when compared to D, integrating it in the Linux kernel is a very complicated task. As evidence, the work required to add support for Rust in the Linux kernel was done by 173 people (present in the commit changelog [21]) over the course of 18 months. This included solely the implementation of the infrastructure required to integrate Rust code in the kernel. It does not implement any device driver or any parts of the Linux kernel in Rust. By comparison, our work was done by 3 people over the course of 4 months, including the initial exploratory phase of the Linux infrastructure as well as the porting of the kernel header files. The actual porting time of the device driver required only 2 to 3 weeks. The reader should consider that, in the meantime, work has been advanced to automate the porting of kernel header files to D [26], thus reducing the required time to integrate D device drivers to a minimum. In addition, the effort to integrate Rust in the kernel has required compiler changes to accommodate the esoteric code encountered, whereas our work does not necessitate any compiler changes.

A previous attempt to create a memory-safe version of the C language and to use it into the Linux kernel is CCured [2]. CCured is a program transformation system that extends the existing type system of the C language by classifying new pointer types according to their usage. There are three pointer categories: (1) SAFE qualified pointers may be dereferenced, but cannot be cast to other types or be used as part of pointer arithmetic operations, (2) SEQ qualified pointers may be used as part of pointer arithmetic, but not in type casts and (3) WILD qualified pointers that can be cast to other pointer types. Each category is treated separately at runtime. SAFE pointers simply require a null check. SEQ pointers are subjected to bounds checking, since they are typically used for array operations. WILD pointers are the most expensive in terms of runtime cost, because they require runtime type information to track the various conversion types that the pointer may be subjected to. It has been previously discovered that, in practice, a large percentage of the casts in C codebases between different types are either upcasts or downcasts [23]. This is also true for the Linux kernel where void* is used as a generic base type in order to enable polymorphism. These types of casts will be treated as WILD pointers by CCured which will be subjected to the costs of runtime checks. By using D, we were able to leverage it's metaprogramming support in order to achieve compile-time polymorphism and type safety without adding any runtime costs.

The pointers defined by CCured are fat pointers: a structure that packs together the raw pointer and metadata related to the boundaries and type information. The authors acknowledge [15] that, because of this, multithreaded programs that rely on shared memory will not work with CCured. The isssue with shared memory programs stems from the fact that the programs not written using CCured will assume that the pointers are one word long and can be written to atomically, when they are, in fact, a fat pointer that occupies multiple words in memory and requires multiple instructions in order

[2]https://wiki.emacinc.com/wiki/Getting_Started_With_Minicom
[3]https://github.com/edi33416/d-virtio/releases

to perform the write, and thus the pointer could get in an inconsistent state. As D's arrays are also fat pointers, they suffer from the same problem. We, as do the authors of CCured, believe that this problem can be resolved by acquiring locks on the shared memory before accessing it. Although, in theory, this solution will impact performance we have not encountered it in practice while interfacing D with programs written in other languages.

CCured was used on two Linux kernel device drivers, on Linux kernel version 2.4.5, with no significant performance penalties. However, it has incurred performance penalties ranging from 11% to 87% on other programs, as it is detailed in its paper.

Another approach to use a modern programming language for the Linux kernel drivers, in order to increase the reliability of the system, was done using the Decaf drivers architecture [18]. The Decaf architecture partitions the code of a driver in two separate parts: one that must run in the kernel-space for high performance and must satisfy the OS requirements and one that can be moved to the user-space and be rewritten in another language. The communication between these two parts was done through extension procedure call (XPC). Using this architecture and the Linux kernel version 2.6.18.1, five drivers were converted to Java, gaining exception handling and automatic memory management through garbage collection. The performance achieved was close to the one achieved by the native kernel drivers. The drawbacks of using Decaf result are traced to the Java programming language, that has no pointers support. As such, critical paths in the code that use pointers are left in the unsafe part, still running in kernel space.

Conversely, our methodology covers the use of the D language for memory safety enhancements in any type of kernel modules, including those that use multithreading and shared memory, as is the case with CCured [2]. The implementation of new components and the interfacing with other kernel components can be easily done thanks to the language's high compatibility with C, compared to the more complicated syntax of Rust. The entire code of a kernel module can be rewritten in D to improve memory safety, with no need of leaving any part of the code unchanged, as is the case with Decaf [18].

VII. CONCLUSION

In this paper we presented an approach to improve the security of Linux kernel modules using the D programming language. We selected *virtio_net* as our target driver, a medium-sized and actively maintained component in the Linux kernel. We ported the driver in the D programming language and highlighted the functional and performance parity to the original C driver and discussed the security benefits. We elaborated a methodology that can be used on other types of drivers for the same purpose.

The safety features added to the driver show that the D language is able to leverage safety improvements in a kernel module, array bounds checking and compile-time polymorphism being the most important ones.

It is important to note that unsafety inside the kernel is a fact of life. Although one can use a programming language that uses different mechanics that increase the safety of the code that a developer writes, at one point the developer will be forced to perform unsafe actions. Those can come from the need to interact with specific pins on the underlying hardware or the need to interact with the kernel API. Most of the kernel API core works with raw pointers; as such, even though the safe code might implement a sound object lifetime algorithm, being forced to pass the raw pointer to the kernel will void all the safety bets and assumptions. In spite of this, we believe that there are two strong arguments that enable the use of safe languages in practice: 1) the kernel core is extremely stable and robust as it benefits from 30 years of development and bug fixes, and 2) the kernel API clearly defines whose responsibility, the kernel's or the driver's, is to free allocated resources.

Another important observation is that a programming language must be able to adhere to the constraints and design patterns implemented inside the Linux kernel. As Linus Torvalds has stated [30], kernel needs to trump any programming language's needs. For this reason, we believe that the D programming language is a good fit given its proven ease of interoperability with C and the kernel infrastructure.

The extent to which the kernel safety can be improved depends on the degree to which the module implementation is self-sufficient. The more external functionalities the module uses, the fewer safety enhancements can be done.

The performance evaluation we conducted on the *virtio_net* driver shows that the D version of the driver adds little to no overhead to the original C variant. The safety features added are sustainable and do not introduce overhead, therefore, we consider the performance results encouraging.

Given the methodology we created, we are confident other drivers could be ported to D with reasonable effort. Given the similarity to the C programming language, getting accustomed to the D programming language will have minimal impact on the driver developer. This is in contrast to the Rust programming language whose syntax and features are very different from the C programming language. We believe that the increasing interest of adding safe languages into the Linux kernel is a great step forward, as it provides kernel developers with alternatives and flexibility such that they can strike the right balance for their needs and goals.

With these solutions, further drivers could be ported using the methodology described in this paper. Later on, this could be extended to entire built-in components and subsystems in the Linux kernel. Those would bring a much needed improvement in the overall security of the kernel with close-to-no overhead, with a welcoming C-similar programming language.

REFERENCES

[1] AspenCore. (Nov. 2019). *Mobile Operating System Market Share Worldwide*. Accessed: Apr. 17, 2022. [Online]. Available: https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf

[2] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer, "Cured in the real world," *ACM SIGPLAN Notices*, vol. 38, no. 5, pp. 232–244, 2003.
[3] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "DIFUZE: Interface aware fuzzing for kernel drivers," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 2123–2138.
[4] D. Dawson, N. Hawes, C. Hoermann, N. Keynes, and C. Cifuentes, "Finding bugs in open source kernels using parfait," Sun Microsyst. Lab., Brisbane, QLD, Australia, Tech. Rep., 2009. [Online]. Available: https://www.researchgate.net/publication/242083507_Finding_Bugs_in_Open_Source_Kernels_using_Parfait
[5] *D Programming Language*. Accessed: Apr. 17, 2022. [Online]. Available: https://dlang.org/
[6] *Programming in D for C Programmers*. Accessed: Apr. 17, 2022. [Online]. Available: https://dlang.org/articles/ctod.html
[7] *Live Functions: Ownership and Borrowing in D*. Accessed: Oct. 29, 2022. [Online]. Available: https://dlang.org/spec/ob.html
[8] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 754–768.
[9] R. Johnson and D. Wagner, "Finding user/kernel pointer bugs with type inference," in *Proc. 13th USENIX Secur. Symp. (USENIX Security)*. San Diego, CA, USA: USENIX Association, Aug. 2004, pp. 119–134.
[10] *Kernel Self-Protection*. Accessed: Apr. 17, 2022. [Online]. Available: https://www.kernel.org/doc/html/latest/security/self-protection.html
[11] (2022). *Virtio*. Accessed: Apr. 17, 2022. [Online]. Available: https://wiki.libvirt.org/page/Virtio
[12] *Linux Kernel CVEs*. Accessed: Apr. 17, 2022. [Online]. Available: https://www.linuxkernelcves.com/
[13] K. Lu, A. Pakki, and Q. Wu, "Automatically identifying security checks for detecting kernel semantic bugs," in *Proc. Eur. Symp. Res. Comput. Secur.* Luxembourg: Springer, 2019, pp. 3–25.
[14] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "DR.CHECKER: A soundy analysis for Linux kernel drivers," in *Proc. 26th USENIX Secur. Symp. (USENIX Security)*, 2017, pp. 1007–1024.
[15] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 3, pp. 477–526, May 2005.
[16] R. Nitu, E. Staniloiu, R. Deaconescu, and R. Rughinis, "Adding support for reference counting in the d programming language," in *Proc. 17th Int. Conf. Softw. Technol.*, H.-G. Fill, M. van Sinderen, and L. A. Maciaszek, Eds. Lisbon, Portugal: SCITEPRESS, 2022, pp. 299–306.
[17] A. Popov. *Linux Kernel Defence Map*. Accessed: Apr. 17, 2022. [Online]. Available: https://github.com/a13xp0p0v/linux-kernel-defence-map
[18] M. J. Renzelmann and M. M. Swift, "Decaf: Moving device drivers to a modern language," in *Proc. USENIX Annu. Tech. Conf.*, 2009, p. 14. [Online]. Available: https://www.researchgate.net/publication/234787227_Decaf_moving_device_drivers_to_a_modern_language/citation/download
[19] *Rust in the Linux Kernel: Good Enough*. Accessed: Apr. 17, 2022. [Online]. Available: https://thenewstack.io/rust-in-the-linux-kernel-good-enough/
[20] *Rust for Linux*. Accessed: Apr. 17, 2022. [Online]. Available: https://github.com/Rust-for-Linux
[21] *Linux Kernel Commit to Add Rust Support*. Accessed: Oct. 22, 2022. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8aebac82933ff1a7c8eede18cab11e1115e2062b
[22] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-assisted feedback fuzzing for OS kernels," in *Proc. 26th USENIX Secur. Symp. (USENIX Security)*, 2017, pp. 167–182.
[23] M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps, "Coping with type casts in C," *ACM SIGSOFT Softw. Eng. Notes*, vol. 24, no. 6, pp. 180–198, Nov. 1999.
[24] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity," in *Proc. NDSS*, 2016, pp. 1–15.
[25] D. Song, F. Hetzelt, J. Kim, B. B. Kang, J.-P. Seifert, and M. Franz, "Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints," in *Proc. 29th USENIX Secur. Symp. (USENIX Security)*, 2020, pp. 2541–2557.
[26] E. Staniloiu, R. Nitu, C. Becerescu, and R. Rughinis, "Automatic integration of d code with the Linux kernel," in *Proc. 20th RoEduNet Conference: Netw. Educ. Res. (RoEduNet)*, Nov. 2021, pp. 1–6.
[27] *Mobile Operating System Market Share Worldwide*. Accessed: Apr. 17, 2022. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide
[28] *Project Highlight: DPP*. Accessed: Apr. 17, 2022. [Online]. Available: https://dlang.org/blog/2019/04/08/project-highlight-dpp/
[29] *Operating System Family/Linux | TOP50*. Accessed: Apr. 17, 2022. [Online]. Available: https://www.top500.org/statistics/details/osfam/1/
[30] *LKML: Linus Torvalds: Re: [Patch v9 12/27] Rust: Add Kernel Crate*. Accessed: Oct. 29, 2022. [Online]. Available: https://lkml.org/lkml/2022/9/19/1105
[31] W3Techs. *Linux vs. Windows Usage Statistics for Websites*. Accessed: Apr. 17, 2022. [Online]. Available: https://w3techs.com/technologies/comparison/os-linux,os-windows

CONSTANTIN EDUARD STANILOIU received the B.Sc. and M.Sc. degrees in computer science and engineering from the University POLITEHNICA of Bucharest (UPB), Bucharest, Romania, in 2016 and 2018, respectively, where he is currently pursuing the Ph.D. degree in computer science and information security.

Since 2018, he has been a Teaching Assistant with the Department of Computer, Faculty of Automatic Control and Computers, UPB. He is also a member of the Secure Systems Group, Department of Computer. His research interests include programming languages, security and vulnerability detection, code and binary analysis, distributed systems, the IoT, and computer vision.

ALEXANDRU MILITARU received the B.Sc. and M.Sc. degrees in computer science and engineering from the University POLITEHNICA of Bucharest (UPB). He is currently pursuing the M.A. degree in philosophy with the University of Bucharest, Bucharest, Romania.

His research interests include programming languages and compilers.

RAZVAN NITU received the B.Sc. and M.Sc. degrees in computer science and engineering from the University POLITEHNICA of Bucharest (UPB), Bucharest, Romania, where he is currently pursuing the Ph.D. degree in programming languages and security. His research interests include programming languages, security, computer architecture, and education techniques.

RAZVAN DEACONESCU is currently an Associate Professor at the Computer Science and Engineering Department, University POLITEHNICA of Bucharest, Romania. His research interests include operating systems and security, with a penchant for teaching and mentoring. If a class uses "operating systems" as part of its name, it's likely he is part of the team. Research-wise, he is working on software security, particularly Apple iOS security and the Unikraft unikernel in recent years. He is a part of the open source and security community in the university and in Romania.