University POLITEHNICA of Bucharest

Faculty of Automatic Control and Computers,
Computer Science and Engineering Department

PHD THESIS

SUMMARY

Security Hardening for IoT Architectures

Scientific Adviser:

Prof. Dr. Ing. Răzvan-Victor Rughiniș

Author:

Ing. Constantin Eduard Stăniloiu

Bucharest, 2023

Contents

Chapter 1

Introduction

Software security is essential for modern systems due to the increasing number of cyber attacks in recent years. The world has witnessed the rise of cybersecurity attacks in recent years, as depicted in Figure 1.1. Exploits of in the wild vulnerabilities have cost companies billions[1], having researchers at the IBM System Science Institute estimate that fixing a vulnerability costs 100 times more than the development costs[2].

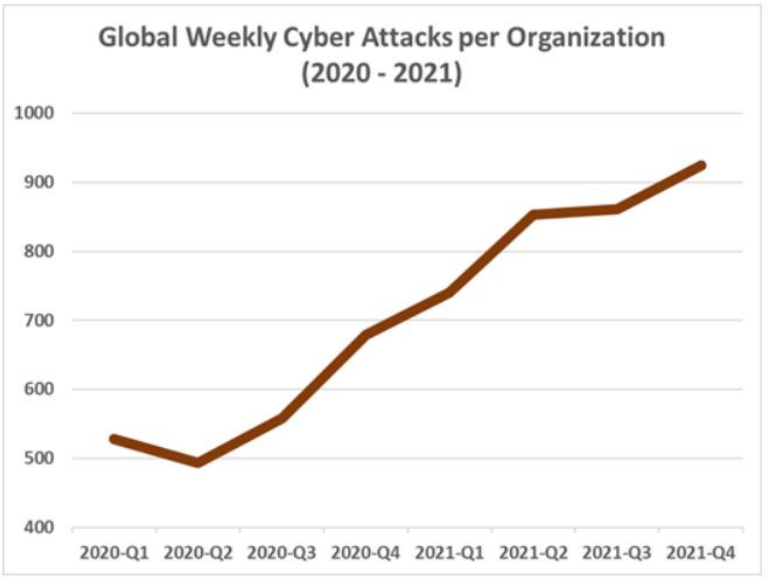
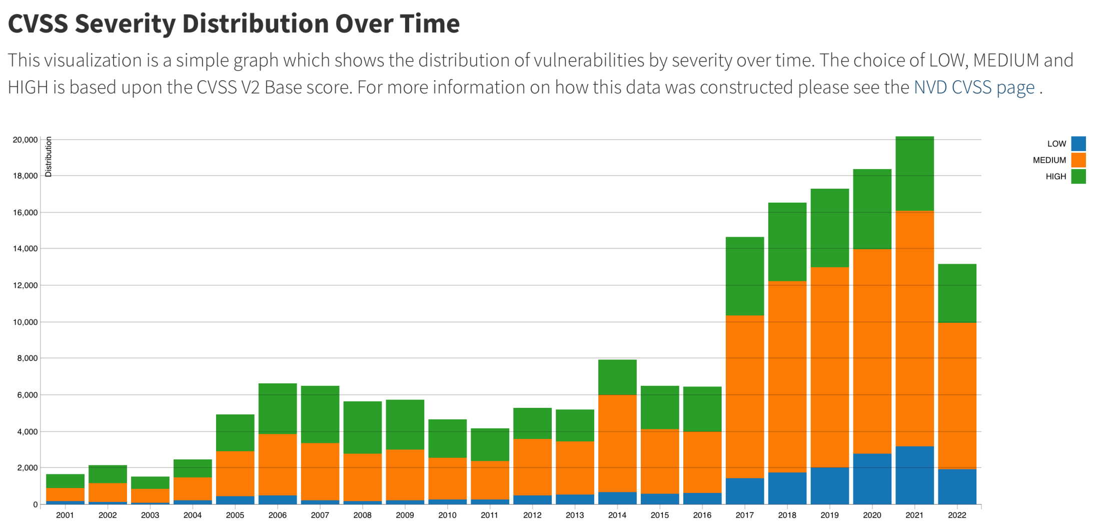


Figure 1.1: Rise of Global Weekly Cyber Attacks per Organization[3]

The growing number of IoT devices (and their vendors) [4] rises privacy and security concerns. The number of vulnerabilities in computer systems and devices, particularly IoT devices, is increasing, and this trend is expected to continue in the future due to the growth in the number of IoT devices[5]. This increase in vulnerabilities presents a significant risk, as it expands the potential attack surface and makes it more likely that these vulnerabilities will be exploited by malicious actors[6, 7]. One way to understand this idea is by looking at the two pieces of evidence provided: Figure 1.2 shows that the number of vulnerabilities is on the rise, with a lot of them being medium and high severity. Figure 1.3 shows that the number of IoT devices is expected to double by 2030. These two pieces of information are related, as the increase in the number of IoT devices will likely lead to an increase in the number of vulnerabilities, as there are more devices that can potentially be exploited.

To tackle this issue, programming language designers have developed systems program-



CVSS Severity Distribution Over Time

This visualization is a simple graph which shows the distribution of vulnerabilities by severity over time. The choice of LOW, MEDIUM and HIGH is based upon the CVSS V2 Base score. For more information on how this data was constructed please see the NVD CVSS page .

Figure 1.2: Common Vulnerability Scoring System (CVSS) Severity Distribution Over Time[8]

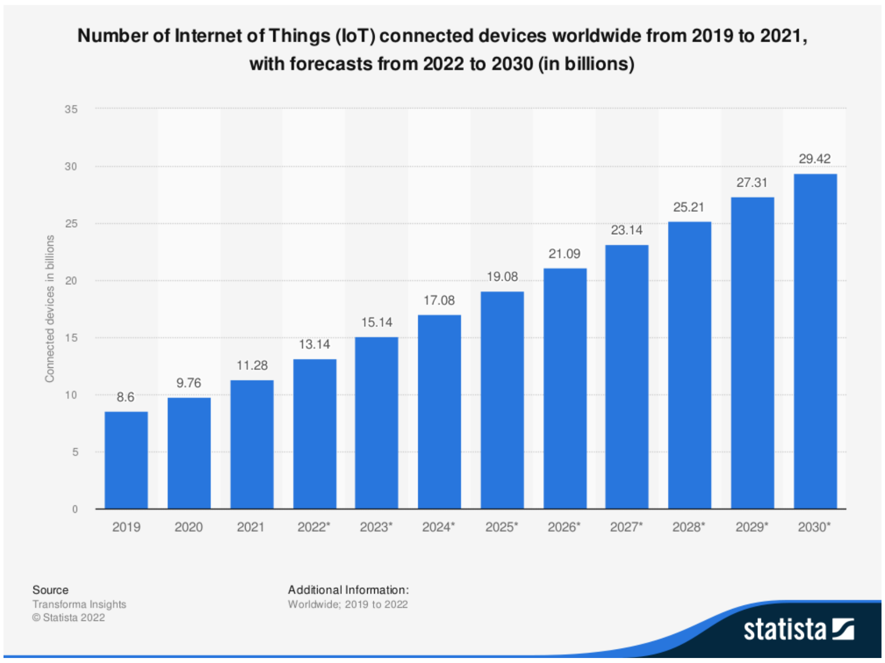


Figure 1.3: Number of Internet of Things (loT) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030 (in billions)[9]

ming languages, such as Rust and D, that are focused on providing providing memory safety and increasing developer productivity without sacrificing performance, all while keeping the resulting binary size small. This means that memory safety and fast prototyping are no longer a feature of high level, resource hungry, programming languages. Those key properties allow for the development of IoT devices in safer alternatives (to C) that can prevent vulnerabilities as early as compilation time.

Operating systems are ubiquitous. Any smartphone, laptop, cloud computing machine, wearable or Internet of Things (IoT) device has an operating system behind the scenes [10]. Due to being such a fundamental component of every device, the operating system must be safe and secure, as any vulnerability, however small, has the potential to compromise the entire system [11].

The kernels of most popular operating systems, including Linux, Android, MacOS, iOS, and Windows [12], are implemented using the *C* programming language. Virtually any mobile, desktop, or laptop computer uses the *C* language at its core. In Linux, the majority of the source code is represented by device driver implementations [13]. The device driver is a critical component of the operating system, and, as such, the importance of its security cannot be overstated [14]. It is very well-known that C is currently the go-to language for systems programming. But, just as well-known are the potential security issues it exposes, including, but not limited to, memory-related bugs such as buffer overruns.

One such example is the case of the *RTLWIFI* driver, built for Realtek Wi-Fi chips, which, in 2019, permitted a buffer overflow to be triggered inside the Linux kernel, with no user interaction whatsoever [15]. The highlight here is on **no user interaction**, suggesting the flaw was intrinsic to the driver itself.

As device drivers' code represent the biggest part of the Linux kernel, it also represents a huge attack surface. We believe that the D programming language can be a great candidate for writing kernel device drivers in a safe programming language. D is fully compatible with C and mechanically checks, during compilation, the user code for unsafe behavior. The language also boasts powerful compile-time features that make it possible to write highly specific, high performance code. It also has a smooth learning curve and does not force language features onto the users, allowing them to gradually improve their code base, as they become more comfortable with the language.

Increasing the security of the Linux kernel automatically increases the security of all the applications that run on top of it. Lightweight devices, such as sensors, have a firmware flashed onto them and do not run on top of a kernel. Although that is the case, they are usually connected to a gateway device that runs on Linux so that they can send the collected data, over the Internet, for the end user to access. Because of this, we make the case that a vulnerability in the kernel of the gateway device would affect the entire system. Thus, increasing the security of the Linux kernel will benefit all the devices.

As previously stated, lots of IoT applications communicate with cloud services. The standard way of accessing services is through a RESTful API: the available endpoints of a service are published using the OpenAPI Specification[16] - a JSON file that describes the endpoint URI, the methods supported by the URI, the accepted arguments and the expected return values. To harden the security of the developed applications, secure libraries and API implementations are needed. We aim to provide a way of automatically generating secure client side libraries based on a given API JSON description.

Testing and validating the security of IoT systems is difficult, due to the fact that most of the software is proprietary (COTS - custom off-the-shelf - binaries) and the embedded nature of the system makes it hard to collect data and audit device security. The user is forced to trust a 3rd party and is not able to validate the software, as they do not have access to the source code. Fuzz testing is an established automated testing procedure that helps teams discover untested paths and vulnerabilities in their products. The goal is to implement a fuzzer that can test COTS binaries, providing IoT users with an audit tool for their devices.

1.1 Thesis Structure

The rest of this summary is structured as follows:

Chapter 2 presents our contributions regarding improving the security of the Linux kernel. It details how the D programming language can be used to write secure drivers in the kernel ecosystem. We show that C code can be ported with ease to D without incurring any performance penalties. We also present our work that further eases the process, by automatically translating the kernel data structure definitions to D compatible ones.

Chapter 3 presents our contributions with respect to improving the security of applications. First, it presents our work in developing a new collections framework for the D standard library. Our collections are able to infer the safety of the operations from the contained, user provided, type. We show that our implementation may provide performance benefits of up to 2x compared to the existing standard library implementations. Next, we present our contribution in building a libraries generator based on OpenAPI Specifications.

Chapter 4 presents our work on improving the River fuzzing framework. First, it details the additions that enable the framework to fuzz IoT networks. Next, it presents the challenges of performing symbolic execution on COTS binaries and the limitations of current, state of the art, symbolic execution engines, and how our solution overcomes those.

Chapter 5 concludes, discusses future work and presents the list of publications.

Chapter 2

Improving the Security of the Linux Kernel

2.1 Safer Linux Kernel Modules Using the D Programming Language

The Linux kernel is being used 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 chapter, 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.

2.1.1 Introduction

The Linux operating system kernel is widely used on a wide range of hardware including supercomputers [17], servers [18], handheld devices [19], and embedded systems [20], making it the most popular operating system.

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 2.1 highlights the number of vulnerabilities discovered in Linux based on the Common Vulnerability and Exposure (CVE) reports [21], which is an average of roughly 250 reports per year. There is no way of knowing how many undiscovered vulnerabilities exist and are being actively exploited.

To protect itself from potential security attacks, the Linux kernel employs a variety of self-protection mechanisms [22, 23] such as Kernel Address Space Layout Randomization (KASLR), Kernel Page Table Isolation (KPTI), stack protector etc. However, these vulner-

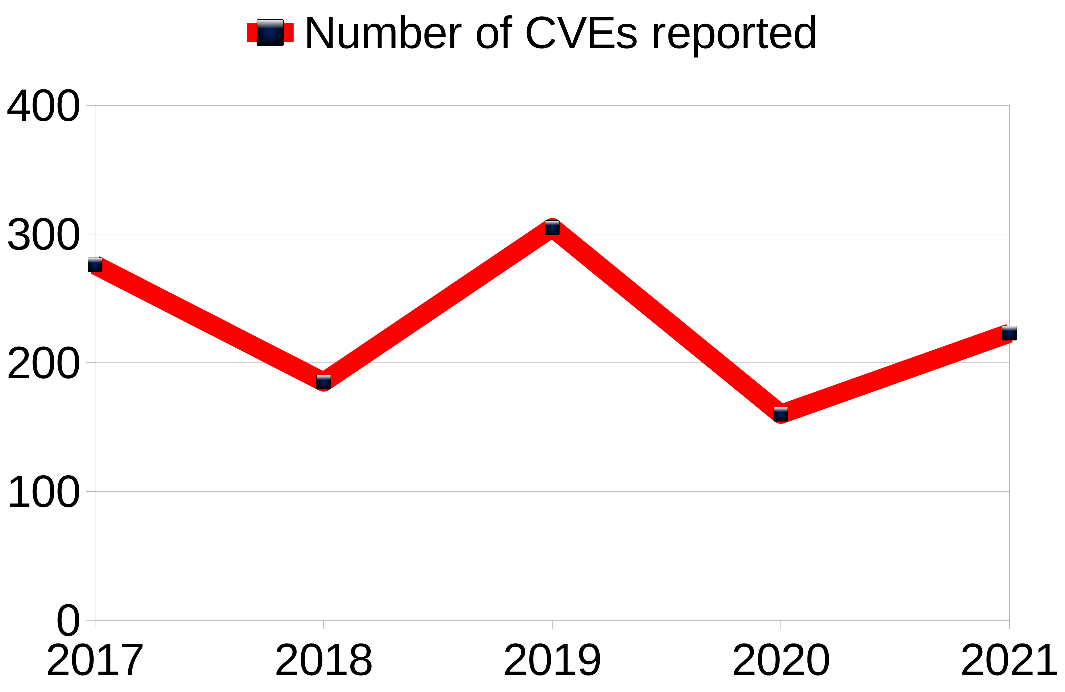


Figure 2.1: Number of Common Vulnerability and Exposure (CVE) reports.

abilities appear due to a combination of programmer mistakes and lack of safety support from the programming language. The Linux kernel is mostly written in C, a fast language but with minimal safety features. C syntax allows easy access to the program memory, but it is also the main source of vulnerabilities like buffer overflows, pointers to expired data, pointers to uninitialized memory.

In this chapter 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 [24], 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. Thus, 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, templates, and more.

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

- We have shown that it is possible to use a safe language in the kernel by successfully porting a Linux network driver to D. Specifically, we ported `virtio_net` of the virtio framework [25].

- 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 in the D programming language. Demonstrated by our successful port, the methodology can be used to port other Linux kernel modules.

2.1.2 Methodology of Porting Modules to D

Introducing D Code into the Linux Kernel

There are two options for extending the functionality of the Linux kernel: (1) statically linking a new object file directly to the core kernel, or (2) building a loadable module and dynamically linking it as needed.

The recommended best practice is to enhance the functionality of the Linux kernel through dynamic loading. This helps preserve the kernel codebase uncluttered and easily maintainable. Additionally, using loadable modules allows runtime customization and helps to keep the attack surface small, reduces vulnerabilities risks and improves security.

Porting a module in 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 building the linking the new module

For the 2nd requirement, to allow the kernel to communicate with a module written in D, a C interface must be created. For it, the goal is to include the bare minimum: registration macros and the corresponding stubs that call into the D implementation. Finally, the Linux kernel build configuration must accommodate 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 with the `-betterC` switch beforehand, and have their name appended with the suffix *.o_shipped*. Once they are built, they can be integrated into the build system and linked, alongside regular objects, to form a module (`.ko` kernel object).

Porting Modules to D

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

1. Port the data types, such as `structs` and `typedefs`, required by the module implementation so that the layout is consistent between D and C objects. This ensures that the module will function properly after the porting process.

2. Port the module implementation incrementally: port one routine then run simple tests that check the module's functionality. Repeat the process until all the functions have been ported.

3. After completing the porting process, the first set of benchmarks should be run to evaluate the behavior of the module. This involves comparing the D version of the module to the original C version to assess any differences in correctness.

4. Improve the implementation by adding language features: `@safe`, replace macros and casts with metaprogramming, and other memory safety features.

5. Benchmark for a second time in order to evaluate runtime performance. Compare the original module implementation with the raw and improved D variants.

Safety Enhancements

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

Variables are initialized to a default value of their type, removing initialization bugs. Furthermore, local variables marked with the **scope** keyword are limited to the function scope, reducing the presence of dangling pointers.

In D, unlike C, the type system does not allow **implicit casts** from the `void*` type to any other type of pointer, as this could lead to silent data corruption bugs. D requires an explicit cast for converting pointers of different types.

D disallows **implicit switch fall-through**. 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.

Static arrays are by default bounds-checked.

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.

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

Safe functions (annotated with `@safe`) are statically verified against cases of undefined behavior. Uninitialized variables, taking the address of a local variable, explicit casts, pointer arithmetic, calls to unsafe functions and more, are disallowed and will issue a compile time error.

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.

2.1.3 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, and 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.

Functional Correctness. We ran 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.

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.

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.

`@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.

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.

Table 2.1: 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%

Performance

To assess performance, we utilized the *iperf3* tool, which sends packets back and forth in a client-server communication. 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: 1) **vm-to-vm** One VM is running the server, one VM is running the client. Both machines are of the same type: either C and either D; 2) **vm-to-host** The host is running the server, the VM is running the client; and 3) **vm-to-remote** Another system in the host network is running the server, the VM is running the client.

Results are summarized in Table 2.1 and show negligible overhead for the D module implementation compared to the C implementation. Given that parts of the measurements show a negative slowdown, we consider performance similar and subject to network and measurement variation.

2.1.4 Key Takeaways

We investigated the possibility of using the D, a memory safe, systems programming language, to develop Linux kernel modules in order to increase the overall robustness and security of the system.

We developed a methodology and applied it to port the `virtio_net` networking driver to D, as our Proof-of-Concept. We demonstrated the functional correctness and performance parity of our ported driver to the original C implementation, and highlighted the added security benefits.

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.

2.2 Automatic Integration of D Code With the Linux Kernel

The D programming language provides built-in features for compiler checked memory safety and high-level programming concepts as object-oriented programming, metaprogramming, and functional programming concepts. It also has native support for interoperability with C. However, as seen in Section 2.1, in order to use D in the kernel, one must manually translate the necessary C header files into D header files, which can be a tedious and time-consuming process. While DPP is a tool that can automate this task for user-space C header files, it fails when working with the complex and sometimes convoluted code of the kernel.

In this chapter, we extend our previous work and enhanced DPP to allow for the translation of kernel headers to D. This eases the use of D code within the Linux kernel, providing a means for improving the kernel's memory safety.

2.2.1 Introduction

As discussed previously, we believe that drivers can greatly improve their security by using a modern programming language that was designed with memory safety in mind, like the *D* programming language [26]. *D* can automatically perform safety checks (e.g. automatic array bounds checking, avoiding use after free, double free, avoid pointer arithmetic etc.), compile to fast and efficient native code and interface with relative ease with *C* code. To this end, one is required to declare the function header and the layout (eg. `struct` declaration) of the types that are going to used, much like one does inside of *C* header files. Given the repetitive, and frankly boring, nature of the task, *DPP* [27] was developed in the *D* community to help developers reuse *C* code from their *D* modules. However, DPP does not work well when used with Linux kernel header files due to the magic constructs inside the kernel. In this work, we improve DPP to successfuly translate such header files and facilitate the integration of D, a memory safe programming language into the linux kernel.

2.2.2 DPP

DPP wraps the D compiler in order to support the *#include* syntax inside a D module. In other words, DPP is a preprocessor for the D language, generating bindings inside the D implementation module itself. DPP has many success stories, including the translations of *curl.h*, *Python.h*, *stdio.h*, *stdlib.h*, and *pthread.h*. However, DPP struggles with Linux headers, specifically:

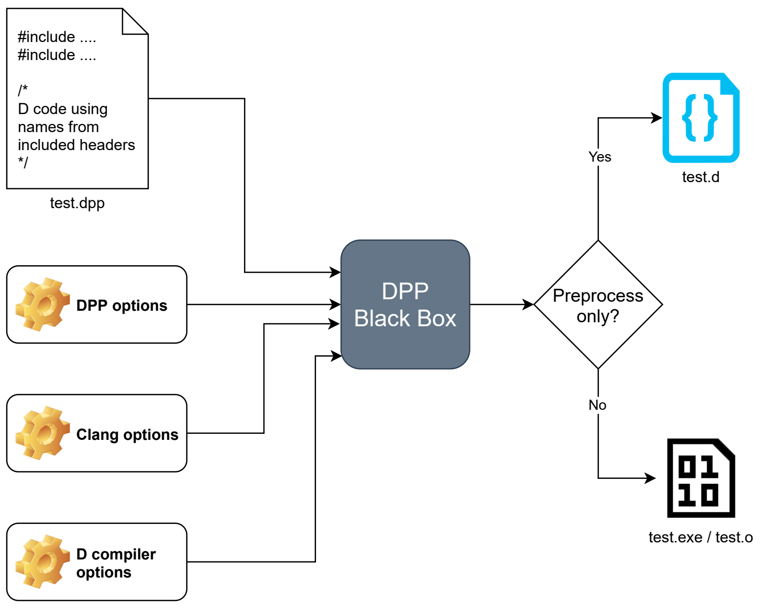


Figure 2.2: DPP workflow: (1) Write a normal D module, but also *#include* the desired C headers, (2) Configure (i.e. clang options are used for the C headers; D compiler options are discarded if DPP is used only for preprocessing, thus for generating a valid D module), (3) Run the tool, (4) The output depends on the options DPP was run with: ready-to-compile D module, object file, or executable file.

- Incorrect translations of multiple macros, functions and structures.
- Name collisions and correlations not dealt with correctly and consistently.
- Crashes when testing with specific headers (i.e. out of bounds indexing).
- Abnormal memory consumption (more than 8GB of RAM required).
- Long execution times (multiple headers could even last for more than 30 minutes).

A typical workflow when using the *DPP* tool is depicted in Figure 2.2. The DPP Black Box is comprised of four different components, as shown in Figure 2.3. Each component has its own definite role, some decomposing into further granular modules (e.g. the *Runtime* and *Translation* modules). Note that, in this project, there are three different concepts revolving the Clang compiler: (1) the *libclang* library, which offers a *C* interface for parsing source code, (2) the *libclang* D bindings library, which also wraps the *libclang* library and (3) the *Clang* component, which wraps a small part of the *libclang* D bindings library. The DPP architecture is highlighted in Figure 2.3.

Figure 2.4 shows how DPP's components interconnect, the exchange of information between the components, and the overall internal workflow of the application. Below, we describe each component.

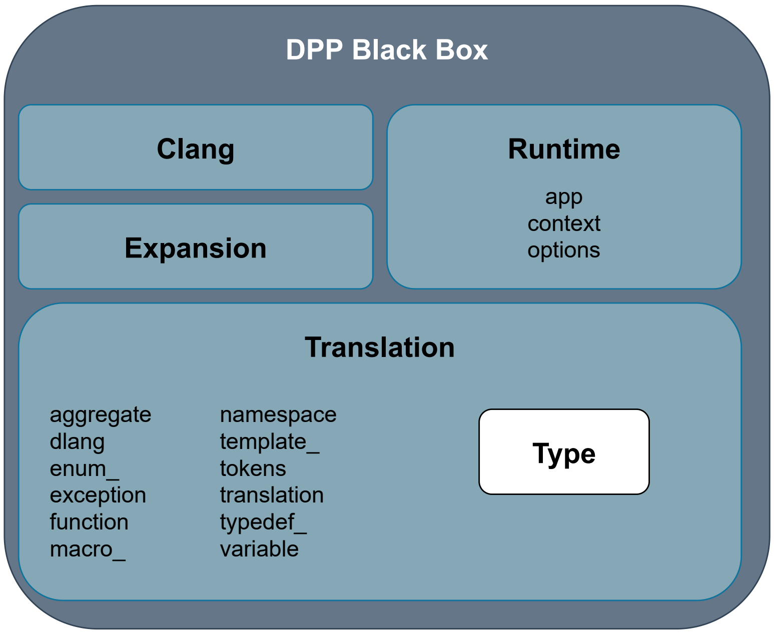


Figure 2.3: DPP Organization

2.2.3 Implementation

App Component. The *App* module is the entry point of *DPP*. It needs to be provided with an object of type *Options*, which carries the command line options used for *DPP*. Based on the provided options, a *Context* object is instantiated that is used to track the already processed *AST cursors*. Next, the *.dpp* file is preprocessed by the *Expansion* component. In turn, the latter expands all the *#include* directives inline, translates all the definitions, and redefines any macros defined therein. Finally, the *App* component manages the compilation and file *I/O* operations: creates the output *.d* file, writes the translated lines of code to it, runs the *C* preprocessor, runs the *D* compiler.

Context Component. The *Context* module encapsulates the necessary information for the current translation, avoiding declaring global variables. It solves name clashes and collisions, declares unknown structures (i.e. when a header only uses a pointer to a type, while not providing a definition for that type). Further, it keeps track of: visited *AST cursors*, lines of the output translated so far, information regarding name clashing, collisions and renaming mappings, all defined macros, aggregates and their fields and other information specific to *C++* code. A *Context* is instantiated only once by the *App* module of *Runtime*, before starting the translation process.

Options Component. The *Options* module holds all the command line options passed by the user: *DPP* options, *Clang* options (internally passed to *libclang*), and *D* compiler options. It can stop the execution of *DPP* after generating the bindings, but before compiling the resulted source file (i.e. using the preprocess-only option of *DPP*). This component is used by the *App* module directly, and by the *Expansion* component, indirectly through *Context*.

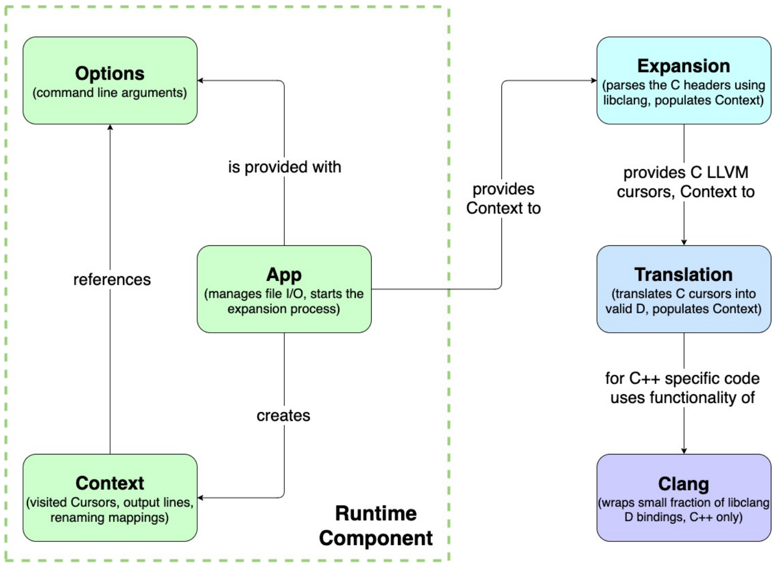

Options
(command line arguments)
is provided with
references
App
(manages file I/O, starts the expansion process)
creates
Context
(visited Cursors, output lines, renaming mappings)
Runtime Component
provides Context to
Expansion
(parses the C headers using libclang, populates Context)
provides C LLVM cursors, Context to
Translation
(translates C cursors into valid D, populates Context)
for C++ specific code uses functionality of
Clang
(wraps small fraction of libclang D bindings, C++ only)

Figure 2.4: DPP Internal Components' Workflow

Expansion Component. The *Expansion* component is responsible for expanding (e.g. parsing) the *#include* directives inline, populating the *Context* instance with the resulted *D* definitions. It uses the custom *libclang D* bindings to start the parsing of the *C* headers, providing the *clang* options from the *Options* instance within *Context*. The result of the parsing is an *LLVM AST*. The *Expansion* component solves the multiple declarations and tentative definition issues of *C*, by modifying the cursors of the *AST*. *D* does not support multiple declarations and one definition of a type (i.e. it supports either a declaration, or a definition, but not both at the same time).

Another function of this component is to iterate over all the cursors in the *AST* and use the *Translation* component to translate them into valid *D* code.

Translation Component. This component's goal is to translate every *C* construct encountered while ex- panding the headers in the original *.dpp* input file. This component has twelve constituent modules:

- *Aggregate* is used to translate *struct*s, *union*s, *class*es, anonymous records, and *enum*s.
- *Dlang* deals with *D*-specific translations (e.g. avoiding using *D* keywords as aggregates' names).
- *Enum* translates only *enum*s which store single-value constants.
- *Exception* is an internal module which defines an exception thrown when using *C++* concepts with no *D* equivalent.
- *Function* is used to translate function declarations and definitions, as well as class methods, constructors, destructors, move constructors etc.
- *Macro* is used to handle macros, and their respective definition (i.e. code fragments,

string representations in hex, octal).

- *Namespace* and *Template* handle the *C++* specific concepts with the same name.
- *Tokens* is an internal module which provides utility functions to help manage tokens inside cursors.
- *Translation* is used for translating C AST cursors, provided by the Expansion component.
- *Typedef* and *Variable* translate typedef constructs, and variable declarations respectively.

The *Translation* component also contains a sub-component called *Type*, which is used to translate/ map the C/C++ primitive and builtin types into corresponding D types.

2.2.4 Evaluation

Performance

In its previous state, DPP was not able to translate any of the Linux kernel headers, in part due to multiple bugs and untreated syntax constructs. In addition, DPP's relatively high memory consumption was hindering the translation of multiple headers at the same time.

Currently, DPP can successfully translate a module containing all the Linux C headers necessary for the porting of *virtio_net.c*, listed in Listing 2.1. The resulted translation output file has 150,000 lines of code and no syntax errors whatsoever, needs no manual interventions and is ready for compilation.

```c
#include <linux/device.h>
#include <linux/mod_devicetable.h>
#include <linux/workqueue.h>
#include <linux/netdevice.h>
#include <linux/filter.h>
#include <linux/socket.h>
#include <linux/bpf.h>
#include <linux/virtio.h>
#include <linux/virtio_config.h>
#include <linux/virtio_net.h>
#include <linux/cpuhotplug.h>
#include <linux/netdev_features.h>
#include <linux/cpu.h>
```

Listing 2.1: C Linux headers necessary for porting virtionet.c

As shown in Listing 2.2, the translation of the headers in Listing 2.1 took 17 minutes and used 4.5GB of memory. This is very encouraging for DPP's future, considering that previously it struggled to offer reasonable runtime figures: for a couple of headers, DPP used to last more than 30-40 minutes and, eventually, run out of memory. Before our changes, DPP used 6 to 7GB of memory for the *netdevice.h* header alone.

```
User time (seconds):: 992.60
System time (seconds): 30.63
Percent of CPU this job got: 100%
Elapsed (wall clock) time (h:mm:ss or m:ss): 17:03.07
[...]
Maximum resident set size (kbytes): 4780252
[...]
```

Listing 2.2: /usr/bin/time output

Productivity

The most important metric DPP aims to improve is the productivity of D programmers who need to interface with C libraries. We identify two aspects of a software engineer's productivity: development time, and maintenance time.

We know from our previous work that it took us several months to manually translate **only** the needed kernel structures and functions declarations. In the end, our codebase had 8284 lines of code (LoC) and span across 27 files. By contrast, using DPP, the translation time dropped to 17 minutes and the codebase was reduced to 3821 LoC and just 3 files.

Using DPP, the translation time is orders of magnitude lower, measurable in minutes, rather than months. This is highly valuable as it enables developers to invest their time on actual development logic, rather than on the repetitive (and error-prone) task of manually defining interfaces. Manual translations are by default more error-prone and difficult to solve than the translations provided by an automated generator. A bug in a manual translation could mean making changes in multiple parts of the code, while solving a bug in a generator automatically solves the issues in all parts of the code.

2.2.5 Key Takeaways

The goal of the project was to make DPP robust and capable of translating even the most uncanny C constructs and syntax, eventually leading to the successful translation of the Linux kernel headers required to build the *virtio_net* driver. The strategy was to incrementally test the kernel headers, identify and investigate issues, identify the issues' patterns and provide a generic solution that solves them.

We have solved issues ranging from C11 anonymous aggregates not being handled, name clashes, renaming inconsistencies, and added new functionality to the tool itself. One of the final blockers that made DPP unusable in the Linux kernel was the excessive memory consumption. This is no longer an issue and we have proven that DPP works with decent resources available.

As it stands, DPP can generate bindings for all the functions, structs, enums etc. used by *virtio_net*. This result is utterly important and relevant in the field of systems programming, especially for building operating systems drivers or modules in an inherently safe way.

Chapter 3

Improving the Security of Microservices and Applications

3.1 A New Collections Framework for the D Programming Language Standard Library

D is a general purpose, high level and high performance, programming language capable of interfacing with the *operating system API* and *system's hardware*. One of the core features of D is represented by ranges, a powerful and new approach of iterating through a set of elements. However, being a state of the art feature, ranges are not used in the D ecosystem at their full potential.

In this work we provide a *new collections framework for the D Standard Library* that is compatible with the existing algorithms and that provides an API similar to the one provided by ranges. Our implementation enables the collections to infer the safety of the operations from the contained, user provided, type.

We show that our implementation may provide performance benefits of up to 2x compared to the existing standard library implementations, when used in conjunction with a custom allocator.

3.1.1 Introduction

The D programming language [28] aims to provide a fast and effective way of writing correct, fast and maintainable programs. D implements modern and powerful features, such as memory safety, function purity and improved code readability, to satisfy the needs of the continuously growing software engineering industry.

Ranges, as we will elaborate in the next chapter, are the D way of iterating through a set of elements. A range is capable of the following operations: accessing elements, testing for emptiness and modifying elements. Ranges provide a great access adaptor for the algorithms in the standard library. This is due to the fact that having a common interface is simplifying the use of the data structures and allows the user to have reasonable expectations about collections.

Currently, the D standard library provides users with a set of collections to use, but the implementation of those predates some of the existing language features. The existing collections use the builtin Garbage Collector [29], assume that the underlying user type is unsafe and don't work well with the language's *const* and *immutable* type qualifiers.

In this work, we propose a set of general purpose collections (vector, list, map, hashtables, heaps etc.) that are fast, reliable and compatible with the existing algorithms, while leveraging the powerful features of the language. The collections allow the user to choose the desired allocation scheme, by providing a custom allocator to be used by the data structure. They do not make assumptions about the underlying user type, instead they infer the safety and purity of it, all while being able to work in an immutable environment. The end result of our work is a collections library that has been integrated in the D standard library that is more efficient, more flexible and safer in terms of memory usage than the existing alternative solutions.

3.1.2 Iterators

An iterator represents a way of providing access to the elements of a container. Because pointers represent the fundamental abstraction model used in STL [30], there aren't many use cases for a single iterator; you need both the **begin**ning and the **end**ing iterators of the container in order to safely traverse it. This approach suffers from two shortcomings:

- When working with iterators, the user needs to be careful with the pairing of the **begin** and **end** iterators. Wrongfully pairing two iterators is both a frequent mistake and a hard bug to discover.
- When implementing algorithms, a user, essentially, needs to provide both iterators in order to define the **range** of his container. This leads to clunkier, error-prone and less maintainable code.

Ranges represent an alternative to iterators that offer all of the benefits, but do no suffer of any of the shortcomings. The next section will present in detail how Ranges are defined and how they improve the existing iterators.

3.1.3 Ranges in D

Ranges are an abstraction of element access and a core part of D. They provide a new approach to the problem of accessing the elements of a container. Ranges were introduced by Andrei Alexandrescu in the **On Iteration** [31] article, pointing out the weaknesses in the *iterator design*, used by the C++ Standard Template Library (STL), and demonstrating the advantages of the *range design*.

The range design that was presented by Alexandrescu has been implemented in the D programming language. For simplicity, we will present the concept of ranges following the D implementation. Note that other implementations may be slightly different, however, the core concept remains the same.

In D, any structure that provides access to the elements of a container, in order to be considered a range, needs to implement three methods:

- **empty()** - acknowledges if the range is empty or not.
- **front()** - provides access to the first element of the range.
- **popFront()** - removes the first element of the range, shrinking it's size by one.

Those three operations define the basic range type, the **InputRange**. The input range abstracts the sequential iteration of a container and it adapts well to streams of data, such as reading from the standard input.

Some containers need to be iterated more than once. Such a scenario makes the InputRange not suitable. The ForwardRange adds to the InputRange interface the **save()** operation, which returns a copy of the range.

For situations where a reversed traversal is required, the **BidirectionalRange** adds two more operations to the ForwardRange interface:

- **back()** - provides access to the last element of the range
- **popBack()** - removes the last element of the range

Lastly, the most powerful range there is, the **RandomAccessRange** is providing the **opIndex(size_t i)** operator, which provides access to the **i**'th element of the container.

Ranges vs Iterators

Ranges in D programming language are generally considered to be a more convenient and expressive way to iterate over collections compared to C++ iterators. Some of the reasons for this include:

- Ranges are more flexible: Ranges can be easily composed and transformed using a variety of range primitives, such as map, filter, and fold, which makes it easier to perform complex operations on collections.
- Ranges are more expressive: Ranges provide a more intuitive and readable syntax for iterating over collections, which can make it easier to understand the intent of the code.
- Ranges are more efficient: Ranges are implemented in a way that allows them to be used lazily and avoid unnecessary memory allocations, which can make them more efficient than C++ iterators in some cases.
- Ranges are safer: Ranges are designed to be safer to use than C++ iterators because they provide bounds checking and automatically handle iterating over subranges.

Overall, ranges provide a more convenient, expressive, efficient, and safe way to iterate over collections in D programming language, which is why they are generally preferred over C++ iterators. Ranges have also been added to the C++ standard library starting with C++20.

3.1.4 Implementation

Speed. We want our collections framework to be as fast as possible, given the user constraints. To achieve this goal we track the lifetime of a collection using reference counting and enable the user to use custom allocators [32].

Memory Safety. The type systems guarantees that code annotated with **@safe** does not cause memory corruption. However, a collection will interact with user defined types that may call unsafe functions. As a consequence, no matter how safe the collections code is, the safety of the whole will be determined by the safety of the contained type.

In order to achieve safety, the following two observations need to be taken into account: (1) Memory allocation is a safe operation. There should not be anything unsafe here. We just ask the allocator for a chunk of memory, which he will provide if he has any more left. (2) Deallocation is an unsafe operation from the allocator's point of view because it can not know it there are any more references left to the memory buffer he is about to free. Deallocations made by the collections, on the other hand, are safe because the reference counting provides the guarantee that there are no more references to the buffer at the point of freeing it.

Functional Purity. The concept of *pure* functions comes from functional programming [33]. A pure function has two main characteristics that are detailed below. Firstly, a pure function will produce the same result for the same set of parameters. As a result, a pure function's result may be cached and used to elide subsequent calls of the same function with the same parameters. Secondly, a pure function is not allowed to access global mutable data. As a consequence, pure functions don't have side effects. Such functions may be formally proven to be correct or not. Purity both helps the user to reason about code logic and represents a powerful optimization enabler for the compiler.

Functional style. D provides the *const* and *immutable* type qualifiers, immutability being an essential part of the functional programming paradigm. Prior to our work, the container module did not support the use of such qualifiers.

The new collections, on the other hand, support such qualifiers. This enables the implementation of multithreading algorithms and the use of functional style idioms.

Collection properties

The collections defined in our library have the following properties:

- Provide at least the following methods: *empty()*, *front()*, *popFront()*, *save()*.
- Are usable in *safe*, *pure*, *nogc*, *nothrow* contexts.
- Are usable in conjunction with D's transitive type qualifiers: *const*, *immutable* and *shared*
- Are usable with existing algorithms that work on ranges

- Are optimal in terms of performance, provided that the user constraints are met.

The new collections framework enables constraint driven choice: the user can request a collection that satisfies certain constraints [34]. For example: if the user desires a collection that has a key - value mapping with constant insertion and retrieval time, he will be provided a hashtable. In order to achieve this, the internal implementation of the collections uses D's powerful compile-time introspection.

3.1.5 Evaluation

To evaluate our work we have created a synthetic benchmark that is designed to assess the performance implications of our implementation.

```
auto getSList(size_t steps, RCIAllocator alloc)
{
    SList!size_t a = SList!size_t(alloc);
    for (size_t i = 0; i < steps; ++i) {
        a.insert(i);
    }
    return a;
}

void benchmark(size_t steps, RCIAllocator alloc)
{
    for (size_t i = 0; i < 10; ++i) {
        auto a = getSList(steps, alloc);
        auto b = getSList(steps, alloc);
        auto c = a ~ b;
    }
}
```

Listing 3.1: Benchmark sample

Listing 3.1 highlights a high-level overview of the benchmarking code. We create two singly-linked lists using a custom allocator. We insert a number of elements (ranging from 10K to 40M) to analyze the implications of using different allocators.

Figure 3.1 exhibits the results of running our benchmark in 3 separate scenarios: (1) using the standard library implementation of a singly-linked list that uses the garbage collector, our singly-linked list implementation (tagged as exp) using both (2) the garbage collector and (3) the malloc-based allocator. Our observations show that as the number of allocated elements increase, our implementation that uses malloc is the fastest (with a maximum speed-up factor of 2, given our experimental setup). The difference is explained by the fact that malloc is a lighter alternative to the garbage collector. What is surprising, however, is that the standard library implementation is faster than the experimental singly-linked list implementation that uses the garbage collector. The explanation lies in the fact that in the first case, the garbage collector has the freedom to decide when memory is allocated and freed and therefore can optimally apply its memory allocation strategy, whereas, in the second scenario, calls to the garbage collector are inserted by the collection framework.

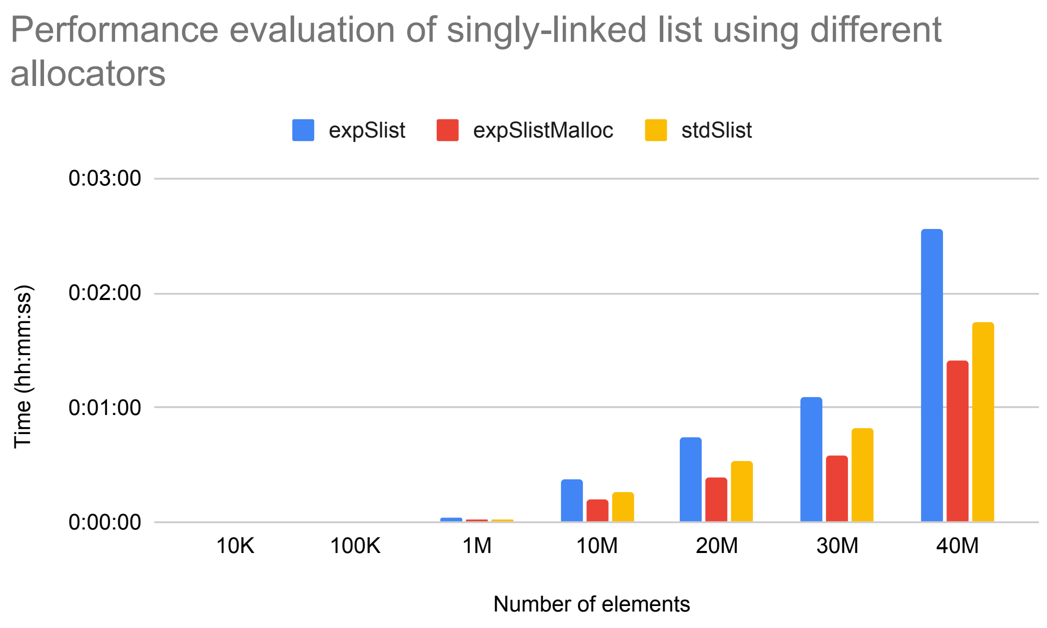


Figure 3.1: **Performance evaluation of singly-linked list using different allocators**: *stdSlist* is the Standard Library implementation that uses the GC; *expSlist* is the experimental framework implementation that uses the GC; *expSlistMalloc* is the experimental framework implementation that uses the MAllocator (malloc based allocator). *expSlistMalloc* shows the benefits of using a custom allocator

This forces the garbage collector to act upon command, without taking into account its formal metrics.

Our results demonstrate that our singly-linked list implementation can be used in conjunction with a custom allocator to obtain better performance than the standard garbage collected implementation.

3.1.6 Key Takeaways

D is a growing language that needs to provide a strong suite of collections. Using the benefits provided by adhering to the ranges interface, the collections will fit right in with the algorithms in the standard library without any extra effort.

We have developed a new collections framework that brings together all the important features of the D Language, while being easy to use and intuitive for the user.

This novel approach of designing collections as ranges with optional primitives has proven to deliver better performance than the standard garbage collected alternative. Moreover, it offers flexibility to the user to choose the most suitable allocation strategy.

Additionally, we have successfully updated the allocators module to a safer API without impacting performance and improved the function attributes of the allocators API. This has improved the compile time type inference of the system. We have released dub packages for the collections framework[1] and allocators module[2]; at the time of this writing the libraries has been downloaded 6916 times, and 1473901 times, respectively.

3.2 Extending Client-Server API Support for Memory Safe Programming Languages

Google web applications have become an integral component in the day to day life of both organizations and individuals alike. These may be accessed through the graphical user interface (GUI) or through the application programming interface (API). The latter is primarily used by programmers to integrate such services into their applications.

Most of the languages used to implement such applications are designed with performance in mind, often neglecting security. However, security has become a major concern for such systems thus increasing demand for memory safe languages. Unfortunately, languages such as D and Rust, known to be memory safe, are lacking support for Google services.

To that end, we develop a methodology of integrating Google services with safe programming languages. We show that the D programming language can easily and successfully integrate such services bringing a boost in security and productivity.

3.2.1 Introduction

Google APIs is a set of interfaces that offer developers programmatic access to Google services, including Google Drive, Google Calendar, etc., thus providing the possibility to integrate them into other applications. Accessing Google APIs by making *HTTP* requests to the server may appear like a straightforward method, but it is recommended to use client libraries that make it easier to access the services from code. Thus, one is not forced to handle all the requests and errors, those being already part of the library implementation. By using libraries, the amount of boilerplate code one has to write is reduced and it surely works as intended because the libraries are periodically tested and updated. Also, client libraries handle all the details and can be easily installed using a package manager such as *pip*, *npm*, or *dub* (Dlang package manager).

The D programming language [35] is a modern, high-level and safe programming language able to interface directly with the operating system's API and hardware. As it's name suggests, D was created as a form of C++ re-engineering, preserving its efficiency and low-level access while also gaining memory safety and simpler syntax [28]. Also, Dlang is compatible with C++, meaning that C++ code can be used in D source code. As with all the modern programming languages, D has a package manager named *dub*, which facilitates the addition of the packages one needs to develop an application.

[1]https://code.dlang.org/packages/collections
[2]https://code.dlang.org/packages/stdx-allocator

The D programming language does not have client libraries for Google APIs, thus making it difficult for the Dlang developers to interact with services from Google. Also, the lack of libraries may be leading to a decrease in popularity, even though D has cutting edge features. To interact with Google APIs, one is required to use more than one library, some of which are fairly heavy libraries, or write all the *HTTP* requests [36] necessary to authenticate and access the resources, since the APIs expose a simple traditional *REST* interface [37]. While offering more control over the code, this approach can lead to many errors or undesired results. For example, one could use wrong parameters for a certain API call or try to modify a read-only resource. In an open source environment, it is better to implement libraries that can be later updated by the community, thus ensuring that every user has a fully functional and up-to-date.

In this work we implement a client-side library written in the D programming language, designed to be used to write memory safe, high performance programs that interact with Google services. We have open sourced our library and made it accessible via Github. By using our library, the developers can easily and safely write programs without reimplementing the same functionality and gaining the advantages of using a robust library.

3.2.2 Google APIs

An Application Programming Interface (API) [38] is an interface between computer systems or programs. Everything, from the most popular operating system to the banking apps we use and our favorite streaming service are based on APIs. In the past fifteen years, the number of public APIs has grown exponentially. According to ProgrammableWeb [39], as of 2021, there are over 24.000 APIs. The evolution of mobile phones to smartphones and the web expansion are some of the major factors that impacted the growth of APIs.

Web APIs need a defined standard format for the data they exchange. Javascript Object Notation[40] is a human-readable format for representing data, based on Javascript object syntax. It can pack primitive data types, such as strings, booleans, and numbers as well as arrays and objects, allowing the API to construct a data hierarchy [41].

A RESTful API[42], most commonly known as REST API, is an Application Programming Interface (API), characterized by:

- client-server architecture using *HTTP* requests
- focus on resources[43]
- unconnected requests

Google APIs represent a set of public APIs, mainly RESTful, that allows developers to interact with Google products and services and integrate them into other products and services [44]. Common use cases for these APIs include:

- user registration or authentication that allows users to log into third-party services or applications using their Google accounts

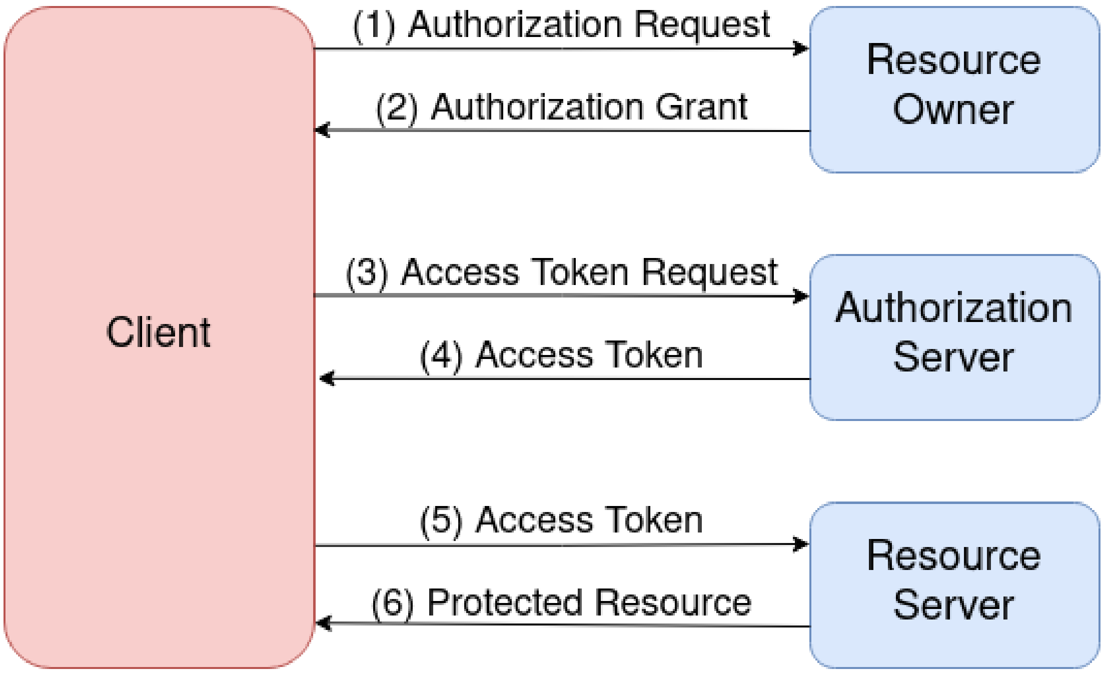


Figure 3.2: OAuth 2.0 flow: (1) client requests authorization, (2) client receives authorization, (3) client requests access token, (4) client receives access token, (5) client requests resource, (6) client receives resource

- documents storage from various tools, such as draw.io in Google Drive
- custom search that allows websites to embed search boxes thus providing a method of searching for that website with the assist of Custom Search API.

Most of the Google APIs require authorization using the OAuth 2.0 protocol [45][46]. OAuth is an open standard for access delegation, used by users to grant other applications access to their information securely and without sharing the password. This standard is not about authentication but authorization. While authentication is the function of specifying access rights/privileges to resources, authorization represents the function of proving that someone is who it claims to be.

Figure 3.2 highlights the steps that are necessary for a client to access a resource: authorization, authentication and resource acquire.

For a developer to integrate Google APIs in other services or products, details about the API must be known to interact with it, such as how to authorize the requests, how to structure the requests and responses, what methods may be used for a resource. Google provided a solution for this issue in the form of documents that describe each version of each API, named discovery documents. Discovery documents are files in JSON format that provide details on how to access the API. The documents include a description of the resources and methods associated, parameters for the methods, requests, and response structure, as well as OAuth 2.0 scopes.

Discovery documents

Google API Discovery Service[47] is an API that provides discovery documents for developers who intend to implement client libraries and other tools used to interact with Google APIs. The discovery documents provided by this services have a specific format including information that split into six main sections:

- Information about the API
- Authentication information - protocol and scopes
- Resources and models(or schemas for requests and responses)
- Methods for each resource
- Other features supported by the API
- Documentation

Listing 3.2 displays a sample of a potential discovery document. It contains the description for a specific version, including the name of the API, the root URL, the service path and the protocol that is fixed to ”rest”.

```
{
  "rootUrl": "https://www.googleapis.com/",
  "servicePath": "drive/v3/",
  "version": "v3",
  "protocol": "rest",
  "discoveryVersion": "v1",
  "name": "drive",
  "ownerDomain": "google.com",
  "ownerName": "Google"
}
```

Listing 3.2: Google Drive API description

3.2.3 Evaluation

In order to validate the generated source code each language had an unit test[48] file, therefore we decided to write one for D to ensure that the result is structured as we expect. As an example, Listing 3.3 represents a simple test case used to validate the module path generated by this tool.

```
def testPackagePath(self):
    discovery = {
        'name': 'foo',
        'version': 'v1',
        'packagePath': 'data',
        'schemas': {},
        'resources': {},
        }
    gen = d_generator.DGenerator(discovery)
    gen.AnnotateApiForLanguage(gen.api)
    api = gen.api
    self.assertEquals('Google.Apis.Data.Foo.v1', api.module.name)
    self.assertEquals('Google.Apis.Data.Foo.v1.Data', api.model_module.
        name)
```

Listing 3.3: D support unit test

Furthermore, apart from the testing made during the development process, we wrote a set of functional tests that makes this process easier and faster. For each method of an API, we implemented a test that executes it and ensures that the results are as we expected. If the method that is tested alters the setup, it reverts the modifications at the end of the test, thus the next time the tests will run, failure will be avoided.

The functional testing process is split into two parts, namely setup and the testing itself. The setup consists in building the service for the API tested and getting the ids of the resources that will be used in the next part. For example, the Listing 3.4 is part of the `initUtils()` used in the tests for Google Drive API and retrieve the id of the file used to test the replies on comments.

```
/* Get Replies Test File Id */
auto response = Utils._drive.files()
                .list_()
                .setQ("name = 'repliesTestFile.txt'")
                .execute();

assert(!response.getFiles().empty);

Utils._replies = response
                 .getFiles()[0]
                 .getId();
```

Listing 3.4: D init tests

The tests execute the requests and then compare the response with the expected response using the `assert` expression. Currently, the only full tested client libraries are those used to interact with Google Drive and Gmail, with more to be tested in future work. Listing 3.5 highlights how a file from Google Drive can be copied using our implementation.

```
import Google.Apis.Drive.v3.Drive: Drive;
import Google.Apis.Drive.v3.DriveScopes: Scopes, DriveScopes;

void main()
{
    Drive _drive = new Drive("credentials_file", Scopes.Drive);
    auto result = _drive.files().copy(DUMMY_FILE_ID).execute();
}
```

Listing 3.5: Retrieving a copy of a file from Google Drive

3.2.4 Key Takeaways

In this work we have added support for Google API services in the D programming language by implementing a library. We have achieved this goal by using a generator for such libraries maintained by Google as a voluntary effort. A substantial part of the generated libraries share a common structure with the Java client libraries, making it easier for developers that used them in other languages to use them in Dlang.

The D programming language community expressed its interest and support in the implementation of these client libraries, therefore proving the relevance of this project. Google Drive[49] and Gmail[50] API client libraries, as well as the Google Client Libraries Generator's fork with the D support added are available on Github. Thus, the Dlang developers can easily clone or fork the repositories in order to use the library. Furthermore, to facilitate the installation process, we intend to provide dub packages for each client library.

Google services expose complex APIs, resulting in a difficult and time consuming testing process. This is the reason for making only two client libraries available for now. We managed to fully test Google Drive and Gmail client libraries. As soon as other libraries will be tested, they will be available on Github and soon after on dub as well.

Chapter 4

Auditing the Security of IoT Architectures and Applications

4.1 IoT Fuzzing using AGAPIA and the River Framework

As the use of Internet of Things (IoT) systems expands, the security risks associated with connecting devices from various vendors also increases. It can be challenging to effectively test and validate the security of IoT systems, as a result of the proprietary (closed-source) nature of much of the software and the difficulties in collecting data, such as memory corruptions, due to the embedded nature of the systems. We propose the extension of the AGAPIA language such that it will enable IoT teams to create more secure applications that can be audited by fuzzing tools like RiverIoT. This would also allow users to test their systems, improve overall trustworthiness. We evaluate how small extensions to existing languages can aid security investigations of IoT systems.

4.1.1 Introduction

The Internet of Things (IoT) is a system of interconnected devices that can collect data from their surrounding environment and act upon it. IoT has grown significantly in recent years, helping people to work smarter and improving their quality of life. We can now find smart devices everywhere, from our homes (smart light bulbs, IP cameras), to our cars (smart sensors, cameras) and cities (weather sensors, pollution sensors). With the continuous development of network technologies, such as 5G, the number of IoT devices will continue to grow [5], bringing even more technology into our lives in a bid to make everything faster, smarter and more comfortable.

The growing number of IoT devices (and their vendors) rises privacy and security concerns. In a fast paced world, where the vendors compete with each-other to achieve the best time to market for device development [4], there is a lot of room for programming errors that can lead to vulnerabilities and exploits [6] [7]. The typical architecture for an IoT system is composed of multiple heterogeneous devices, not necessarily from the same vendor, connected through different protocols over the Internet. Most of the devices run proprietary firmware, with in-house implementations of protocol standards, developed in a memory unsafe language (such as the C programming language), making them a lucrative target for attackers.

Given the closed-source nature of the IoT devices code, it is difficult for a 3rd party (user, another vendor, etc) to audit and validate the correctness of the implementation. Because it can not access the code, the 3rd party must rely on black-box testing, combining fuzzing techniques with functional testing. RiverIoT [51] is an open-source framework that enables the fuzzing of end-to-end IoT applications. In order to provide efficient fuzzing, RiverIoT relies on a JSON specification of the Input/Output (I/O) of the devices fuzzed. We'll discuss this in greater detail in Section 4.1.3.

As previously stated, the programming language used to develop the devices' firmware plays a significant role in the security and robustness of the system. IoT systems and their applications are, in fact, a highly dynamic and modular distributed system. Distributed programming and synchronization is generally regarded as a non-trivial task. There are multiple frameworks and high-level languages that address the issue of distributed programming. The AGAPIA language [52] attempts to increase the developer productivity and the language expressiveness by using a transparent communication model and simple high level statements. AGAPIA expresses distributed applications as *modules* that expose a clear, simple and structured I/O interface.

AGAPIA is a Domain Specific Language (DSL) built on top of the C programming language. Because of this, (we believe that) it can easily be used with existing firmware code to model the IoT system's interaction and expose the device's I/O interface.

We explore the option of expressing IoT devices as AGAPIA modules, and representing the IoT system's interactions as relationships between AGAPIA modules. By doing so, we have a clear, structured, specification of the I/O of a device. We can easily extend AGAPIA to convert the structured I/O spec to (and from) JSON, so we can easily use RiverIoT to test both the device and the IoT system.

4.1.2 AGAPIA

There is a real need for developing a unifying programming language that allows developers to quickly prototype and test their software before deploying it to an IoT device. The current trend is to write the code in a low-level programming language, like C, that will be compiled with specialized compilers (like the Arduino IDE) for each IoT device since the code can be run on different architectures (like PIC, AVR, ARM).

The problem with writing in a low-level programming language is that it is a tedious and error prone process, since one must carefully implement the basic data structures and communication channels. While this offers a lot of flexibility and control in terms of instructions granularity, it hinders development. Even when one has access to a powerful IoT device (like the Raspberry PI) and can write code in a higher-level programming language, like Python, a problem still persists: it is difficult and time consuming to write correct, distributed programs.

There are papers that investigate a way to program using visual objects like [53] and [54], but they fall short on developing for the multiprocessor devices.

AGAPIA [55] is a Domain Specific Language (DSL) that was designed to simplify the development of parallel software using a simpler syntax for spawning new processes via forks and integrating with Open-MPI [56] to deliver a friendlier distributed and parallel programming experience.

The main component of an AGAPIA program is the module that can be thought of a 2 dimensional block (a square) that can receive information from both the north and west sides and can pass information through both east and south parts (Figure 4.1). The inputs and the outputs are optional, so one can define modules that generates data (no inputs, only outputs) or that produce side effects when given certain inputs (only inputs, no outputs).

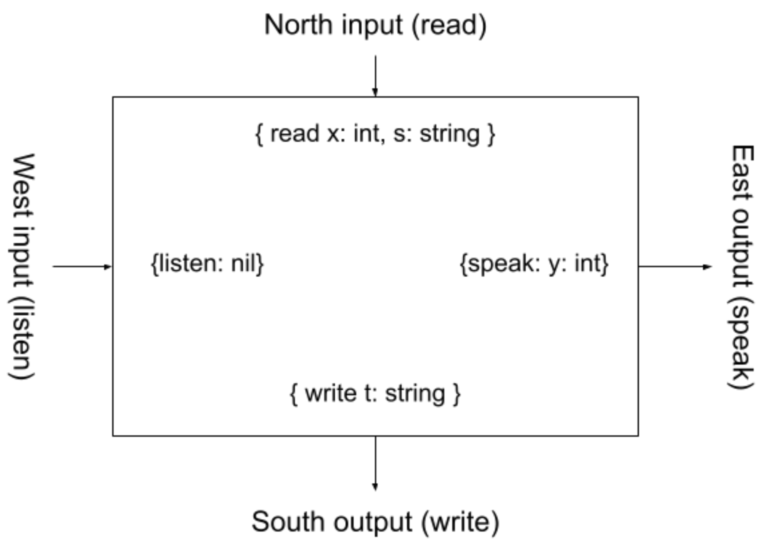


Figure 4.1: An AGAPIA module

The main job of a module is to transform the given input(s) into the output(s). The basic way of defining a module is to define its interface (the inputs and the outputs) and based on them one can compose modules together.

By defining a module in this way, one can compose modules in three ways: vertically, horizontally, and diagonally (Figures 4.2, 4.3, and 4.4). One can think of an AGAPIA program as a two dimensional structure [57], composed of modules that talk to one another in order to solve a problem more efficiently. The composition model used by AGAPIA is multiplexed both in space (Figure 4.2) and time (Figure 4.3) [58].

After defining the interface for each module, one can write the code for that module in either C or C++ language. The user code for each module can contain either pure C/C++ code (called here atomic) or can contain AGAPIA instructions or compositions [59]. The difference between the two modes is that the atomic code must have access to all the variables before it is scheduled, whereas when using AGAPIA instructions one can run code in parallel.

The AGAPIA team is currently working on producing a graphical user interface where one can easily define the topology of an AGAPIA program and define the interface between the modules and after that can write the module's code.

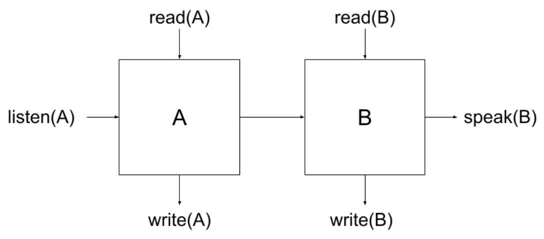


Figure 4.2: Horizontal module composition

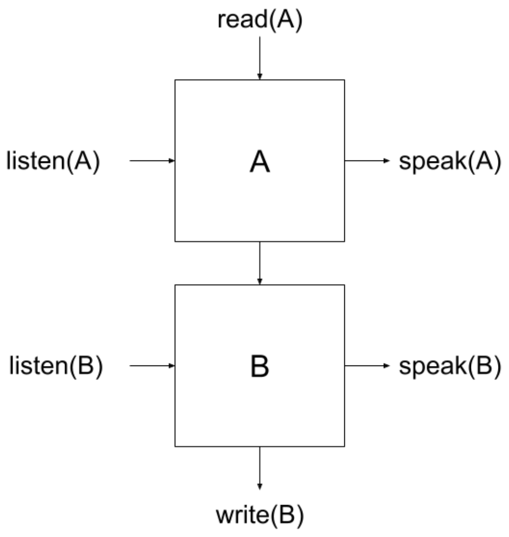


Figure 4.3: Vertical module composition

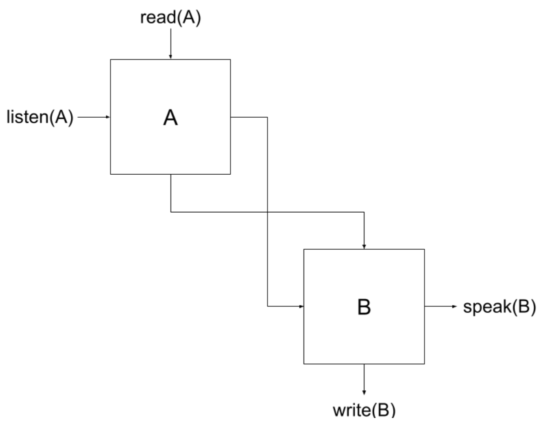


Figure 4.4: Diagonal module composition

4.1.3 RiverIoT

RiverIoT is an integrated testing framework that enables end-to-end fuzzing for IoT systems. The framework employs guided fuzzing through state of the art methods, like concolic execution [60] and AI techniques [61], [62], [62], [63].

There are two major challenges when fuzzing embedded devices: 1) the hardware dependence of the firmware and 2) the closed-source nature of the firmware. Inspired by [64] and [65], RiverIoT addresses both of those issues with emulation. Using an emulator allows RiverIoT to run the firmware on a general purpose system, without requiring 1) the physical hardware of the IoT device. When a binary is emulated, the emulator translates each binary instruction into an intermediate representation before translating it again into instructions for the host machine architecture. Doing so, RiverIoT can dynamically instrument the binary instructions without needing the 2) source code.

Since an IoT system can be logically represented as a graph of connected components, RiverIoT can perform fuzz testing at three different levels:

- graph level - changing the nodes and edges inside the deployed IoT system
- node level - fuzzing each device individually
- interconnection of input/output nodes between different nodes and their communication (i.e., how the input of a node can be influenced by other nodes that are connected to it, or how the communication channel between nodes can affect their input).

In order to use RiverIoT, one must provide a description of the graph (nodes and edges) and the I/O structure that the applications expect, in JSON format. With this JSON specification, the framework can understand the communication between the different components and it can start mutating the graph and generate inputs for the applications.

RiverIoT expects the graph to be able to handle dynamic configurations at runtime, like changing nodes and edges. The reason for this is that the framework attempts to simulate issues like device failures, restarts, repositioning thus testing the resilience of the system under test.

As it can be seen in the graph specification presented in Listing 4.1, the model is quite simple and safe-explanatory. One must define the IoT applications (”iot-nodes”) and their communication channels (”io-edges”). Each application is assigned an ID and a description of it’s input and output buffers, as exemplified in Listing 4.2. Each edge, also identified by an ID, describes the input and output node (based on the node’s ID) and the buffer specification used for that node (based on the node’s buffer ID).

```
{
  "configuration-name": "Generic Network",
  "iot-nodes": {
    "1": { // Buffer description },
    "2": { ... }
  },
  "io-edges": {
    "1": {
      "vout": 3, // Output node
      "vin": 1, // Input node
      "vout-buffer": 1, // Buffer id sending the data
      "vin-buffer": 2, // Buffer id receiving the data
    },
    "2": { ... }
  }
}
```

Listing 4.1: Compatibility graph specification example

The example in Listing 4.2 probably contains some more fields than one might be expecting, but most of them are actually there to provide more context to the reader. RiverIoT is mostly interested in the ”token-type” and ”byte-size” fields. With those two, it can start generating inputs for the application under test.

```
{
  "device-name": "An IP Camera",
  "optional": "false",
  "class": "camera",
  "buffers": {
    "1": {
      "token-delimitators": " ",
      "protocol": "HTTP",
      "protocol-setting": "http://192.168.0.112:8080/",
      "buffer-tokens": [{
        "name": "Camera command",
        "description": "Input that selects command",
        "token-type": "string",
        "byte-size": 256,
        "regex-rule": "[a-zA-Z]+=[a-zA-Z0-9]+", // Optional parameter to
            guide fuzzer generator
        "optional": false
      },
      {
        "name": "Camera ISO Value",
        "description": "Sensitivity to light",
        "token-type": "int",
        "byte-size": 4,
        "optional": true
      }]
    },
  }
}
```

Listing 4.2: Single device buffers’ specification

With the JSON specification ready, the framework can start the fuzzing process. For each node, the framework will perform guided fuzzing in isolation, testing each binary program that can be deployed on a physical device. RiverIoT employs a combination of symbolic execution [60] and genetic algorithms [66] to fuzz the targets in a bid to achieve good code coverage while keeping the runtime performance and overhead acceptable.

4.1.4 Key Takeaways

We propose an extension of the AGAPIA programming language with IoT-aware deployment macros that enables a fast and simple integration with a testing framework, such as RiverIoT. The added extensions generate a JSON specification of the entire system under test (SUT), describing both the graph of connected IoT devices, as well as their I/O communication interfaces. By doing so, it provides the testing framework with a high level view of the entire system, as well as the expected input and output of each node. The generated specification is read by the orchestrating component of RiverIoT which decides if it's going to perform input generation or graph mutations for the SUT. In summary, this research investigates how minor modifications to existing programming languages can enhance the testing and validation processes for IoT systems.

4.2 Scalable Concolic Execution of COTS Binaries with the River Framework

Software systems increase in complexity as the time passes and new business requirements arise. Coupled with the fast pace and increasing demand of software development, this makes room for bugs and leads to a bigger attack surface. Fuzz testing and symbolic execution have proven to be successful methods to uncover untested paths and vulnerabilities, and have been adopted by teams across the globe. Software products regularly use 3rd party pre-compiled libraries and components. Those custom of-the-shelf (COTS) binaries make symbolic execution difficult as the fuzzer cannot insert the instrumentation code during compilation, thus having to resort to dynamic binary instrumentation. We present River, a concolic execution fuzzer for COTS binaries. We emulate the system-under-test (SUT) with Triton, a dynamic binary analysis library, and we delegate system and library calls to GDB. By doing so, we avoid path explosion and the need to manually implement system and library calls in the fuzzing framework, two problems that KLEE and angr have. We compare our solution with KLEE by testing on the libraries from Google FuzzBench. After running libarchive under the two for 30 minutes, River has a branch coverage of 4.56%, while KLEE has a 1.78% coverage; after 1h30 minutes, River has reached 6.06% compared to the 2.07% reached by KLEE.

4.2.1 Introduction

Software security is essential to nowadays systems. The world has witnessed the rise of cybersecurity attacks in recent years, as depicted in Figure 1.1. Exploits of in the wild vulnerabilities have cost companies billions [1], having researchers at the IBM System Science Institute estimate that fixing a vulnerability costs 100 times more than the development costs[2].

In fields such as IoT and automotive, auditing software security is challenging because the user receives a piece of hardware with a custom off-the-shelf (COTS) binary that runs on it. The user is forced to place his trust in the 3rd party, as he is not able to validate the software, since he does not have access to the source code.

Fuzz testing can help teams automate the testing procedure, and discover untested paths and vulnerabilities in their products and 3rd party components. If the source code is available, the fuzzing framework will add instrumentation code to the generated binary, which allows the framework to monitor the system under test (SUT) during runtime and improve the input generation phase of the fuzzer; this is known as *white-box fuzzing*. When there is no source code available, the fuzzer generates input (either from scratch or from an initial corpus) and checks the output (end state) of the SUT to determine if a crash has occured; this is known as *black-box fuzzing*. In-between the two methods is *gray-box fuzzing*: even though the source code is not available, the fuzzer will instrument the binary instructions in order to monitor system during runtime and guide the fuzzing process.

Static binary rewriting means being able to modify a compiled binary such that it remains executable after the changes. Static binary instrumentation would provide the best performance for a fuzzer, but binary rewriting is a hard, active research problem[67][68][69][70][71] that has yet to have a sound, cross-architecture solution. Dynamic binary instrumentation (DBI) modifies the binary execution during runtime. This is a lot easier to achieve, as it does not depend on complex static analysis and it enables the user to leverage runtime information[72][73][74][75][76]. The drawback of using DBI is the runtime overhead that it incurs, which can range between 1.2x to 5x. When fuzzing, one desires the smallest instrumentation overhead, as that will allow the fuzzer to try more inputs in the same amount of time. We plan to compensate the overhead incurred by DBI by using symbolic execution.

We want to be able to perform efficient gray-box fuzzing on COTS binaries. The problem with black-box fuzzing is that you don't know anything about your target: as such, just testing against random input will not get one very far. Thus, we want to enhance the River fuzzer in order to perform symbolic execution on binaries through DBI: 1) trace the binary execution, 2) each time we encounter a branching condition, we save the encountered condition in the form of relations between symbolic variables, 3) the symbolic variables represent our path constraints and will build an algebraic equation of the current execution path, 4) so we can provide the equation to a satisfiability modulo theories (SMT) [77][78] solver, such as Z3[79], to discover if there are any input values that satisfy the constraints of a given program path. By using symbolic execution, one can quickly discover the "interesting" input values that improve the code coverage of a SUT and trigger vulnerabilities; this is because the constraints equation mathematically represents the different states that a regular fuzzer might discover by running multiple inputs.

Library and system function calls represent the biggest challenges of performing symbolic execution on binaries. Figure 4.5 presents a regular application flow, with all the transitions starting from user space and ending in kernel space. Starting with the program entry point instruction, we will iterate through the binary instructions until we reach a function call, eventually translated into a system or library function call.

Library function calls lead to a huge complicated constraints equation, known in the literature as *path explosion*[80]. The equation could undoubtedly become exponential as a consequence of the number of branches, which greatly increases the time required by the SMT solver to find equation solutions, to the point that it takes too long to be usable or it can not find a solution anymore. Even if we would have an ideal SMT solver that is not affected by the path explosion problem, system calls pose a new challenge: the implementation of the call resides in kernel space so the fuzzer cannot instrument those instructions as it does not have access to them. The solution is to either reimplement or emulate such calls.

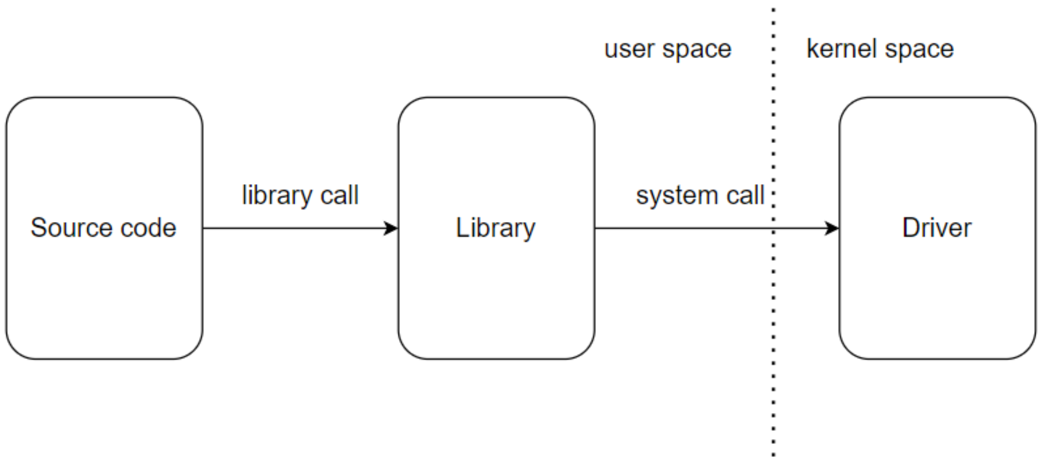


Figure 4.5: Application Flow: from User Space to Kernel Space

Other symbolic execution fuzzers and frameworks, such as angr[81] and KLEE[82], reimplement common library functions (eg. `strlen, printf` etc.) and system calls (eg. `open, read, write` etc.) in order to circumvent the issues briefed above. There are two problems with this approach: 1) this does not scale as one cannot know all the functions that will be used by a user in his codebase, thus it requires extensive analysis to implement the missing stubs so one could fuzz the binary target and 2) the custom implementation might not behave exactly the same as the original library or system call, thus unwilling modifying the behaviour of the system under test.

Our proposed solution will use River to dynamically instrument only the instructions that are part of the analysed binary (no libraries or system calls). We integrate River with GDB so we can delegate the execution of library and system calls to it. Thus, River will only trace and symbolically execute the binary instructions of interest.

There are a series of aspects that we have to consider after the GDB executes the system or library calls:

- must maintain the consistency between memories of every process. The process from River has to have the same information even after the execution of GDB.
- must restore the function context after the execution of GDB.

- must keep the consistent symbolic state before and after a system or library function call.

The significant advantage that we have through our approach is that we emulate system and library function calls. Thus, the developer does not have the overhead associated with the implementation of every system call, nor does he have to do anything else in order to perform concolic execution. We claim that our implementation is scalable as it does not require the developer to debug the fuzzing tool and implement any missing utilities.

4.2.2 The Architecture of The River Fuzzer

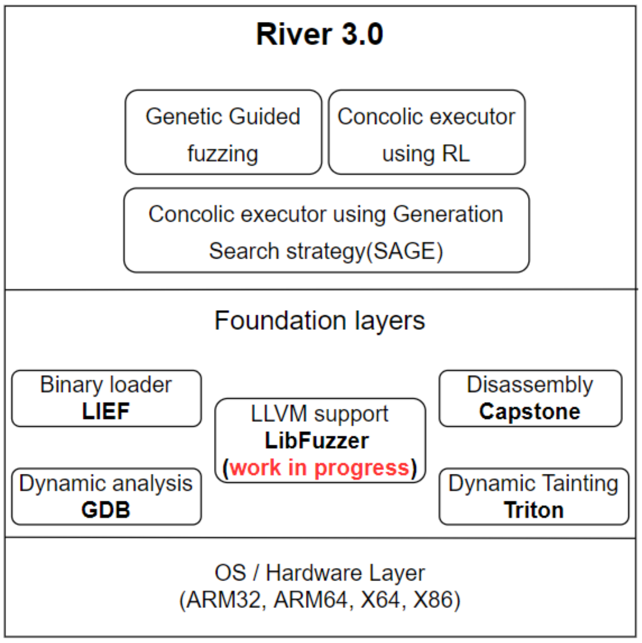

River 3.0
Genetic Guided fuzzing
Concolic executor using RL
Concolic executor using Generation Search strategy(SAGE)
Foundation layers
Binary loader LIEF
LLVM support LibFuzzer (work in progress)
Disassembly Capstone
Dynamic analysis GDB
Dynamic Tainting Triton
OS / Hardware Layer (ARM32, ARM64, X64, X86)

Figure 4.6: River Architecture

River is a cross-platform fuzzer that can run on all common operating systems and supports the following architectures: **X64, X86, ARM32** and **ARM64**. The fundamental components in its architecture are built using the following libraries:

- **LIEF**[83] serves at loading binary files. Using this tool, we can modify the executable file, access offsets, sections and the entry point, functionalities used in restoring the memory, and help to determine the position for necessary instructions.
- **Triton**[84] used in dynamic symbolic execution. Moreover, its usage in the River is to create an AST and make a dynamic taint analysis that verifies the security implications based on the flow generated by the user input. We can access functions from memory with this in River. This component has two substantial functionalities: the taint analysis function and the symbolic execution engine. Moreover, we can modify regions of memory and set symbolic expressions at different memory addresses.
- **Capstone**[85] is a component in the LIEF library because we can disassemble exe-

cutable files through it. This is a high-performance framework, designed to provide the semantics of the disassembled instructions. Moreover, it is a thread-safe tool.

- **z3-solver**[79] is a theorem solver. Its usage is in the River and the Triton library to solve different constraints gathered from the branch instructions. The result after evaluating such expressions is a valid input that feeds the fuzzer in order to detect flaws in the system.
- **GDB**[86] is a tool through which we emulate the library and system calls and restore the context. We can inspect the memory at different moments from execution, using its capabilities of dynamic analysis. We can analyze the content of registers and acquire information about memory sections.

This framework supports three fuzzing methods, as you can see in Figure 4.6: concolic execution based on Generation Search strategy[87, 88], Genetic Guided fuzzing, and concolic execution using Reinforcement Learning[89, 88]. The similarity is that all of these methods generate paths based on the instructions from the executable files. Differences come from the way fuzzers choose paths. Genetic-guided fuzzing calculates traces based on a genetic algorithm, whereas the concolic execution using RL relies on an algorithm of Reinforcement Learning.

4.2.3 Evaluation

In order to validate our solution, we evaluated a series of applications that try to meet real-world requirements. We used a HTTP parser[90] (Figure 4.8) and an application that tries to parse a JSON object, named Fuzzgoat[91] (Figure 4.7). We choose the former because it contain regular system and library function calls met in every program, and the latter because it is intended to be used as a testbench for fuzzers. Another reason is that the fuzzer covers a significant part of the code in a reasonable amount of time despite the overhead caused by the emulation.

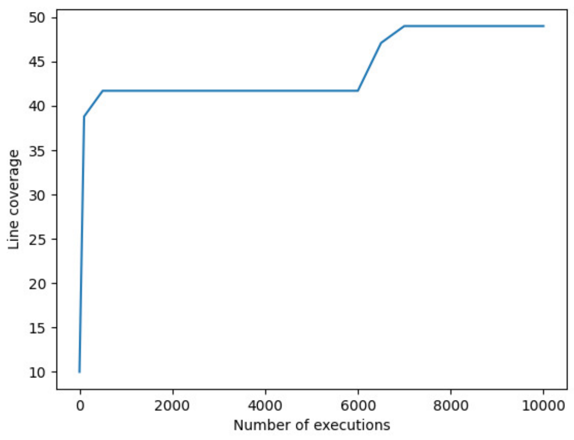


Figure 4.7: Line coverage for Fuzzgoat test

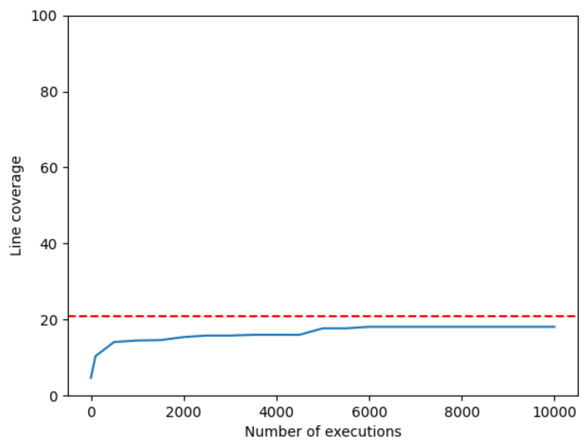


Figure 4.8: Line coverage for http parser test

The HTTP parser has two components: the parser for the request and one for the URL. The line coverage for this library is that the URL parser has 21%, whereas the other component has 79%. We choose to test the URL parser since the input is not highly structured as it is for HTTP request parse. To support highly structured inputs in a reliable amount of time, River has to generate the data tests based on grammar. The current approach creates inputs relying on symbolic execution, which will take much more time to build suitable inputs.

Metrics and performances

So far we have evaluated that our functionality requirements are successfully met by our solution. Next, we compare our approach with similar ones from Klee. We used the Libre [92] and Libarchive [93] libraries from Google's Fuzzbench [94, 95]. We also used Fuzzgoat, previously mentioned, in our data set. The results are shown in the following tables:

TIME	RIVER	KLEE
30min	49%	65%
1h	49%	73%
1h30min	49%	76%

Table 4.1: Branch coverage for Fuzzgoat example.

TIME	RIVER	KLEE
30min	4.56%	1.78%
1h	6.02%	1.92%
1h30min	6.06%	2.07%

Table 4.2: Branch coverage for Libarchive example.

TIME	RIVER	KLEE
30min	28.9%	fail
1h	28.9%	fail
1h30min	28.9%	fail

Table 4.3: Line coverage for Libre example.

Our metric to compare both tools is branch coverage because we can get only the branch and instruction coverage, whereas we have the possibility to analyze the line and branch coverage in River since the latter tests the coverage using the gcov tool and the former has an internal mechanism that makes the measurements. So, the shared metric is the branch coverage.

We can notice from the above results that Klee has a large coverage for the Fuzzgoat example, which contains regular functions and library calls. The other two examples are more complex in terms of system and library calls. Moreover, these applications imply a substantial number of such calls. Klee fuzzer fails with a SEGFAULT signal for Libre example because of the lack of implementation for an inline assembly call from the `valgrind.cc` file. The error occurs even though the Klee was compiled with support for libc++. A possible reason is the lack of some implementation for some system or library function calls. River tries to avoid this issue by emulation such calls through GDB. We can observe from Table 4.2 that Klee is slower when the application involves a notable number of system and library function calls. Moreover, the Libre example implies a significant number of system calls from POSIX. Even though Klee has support for such calls, their approach implies a substantial overhead for them.

4.2.4 Key Takeaways

We have successfully added support for symbolic execution for COTS binaries in the River fuzzer. By delegating the execution of syscalls and library functions to GDB, we significantly reduce the risc of triggering a path explosion. Moreover, the associated overhead with the reimplementation of such calls drastically diminishes. Moreover, through this approach, we want to underline the possibility of emulating the system and library function calls by keeping the concrete and symbolic states consistent.

Restoring the symbolic state is a core aspect of the current approach, since the fuzzer can otherwise miss specific symbolized values after system or library function calls. To prevent any errors, we have an ”aggressive” approach, restoring the memory and symbolic contexts after each GDB delegated call. The restore policy could be relaxed in order to obtain better runtime performance.

Future performance improvements

Our context restoration solution is fully implemented, which is crucial for our approach. However, the current implementation is very cautious and copies large sections of memory from GDB into the River process each time a system or library call is made in order to update the context in the fuzzer. This causes the memory to become a bottleneck because the performance of the tool is directly tied to the size and number of memory sections being copied. In addition to optimizing the use of memory, the testing tool could also benefit from using the functionalities of GDB and leveraging GDB’s signal handler to improve the input generation process by filtering inputs based on previous data tests that have caused errors in the system.

One important factor to consider is how the fuzzer generates its inputs because if it uses the same path multiple times, it can lead to a decrease in performance. To improve performance, the fuzzer could avoid generating redundant data tests by altering its approach for input generation. Optimizing this component can lead to a reduction in the amount of time needed for testing. Thus, the fuzzer could achieve a wider code coverage in the same amount of time by adjusting its input generation process.

Future emulation improvements

During emulation, the fuzzer aims to guide the concrete execution to achieve the correct execution. However, as discussed throughout this chapter, the fuzzer also has an internal representation of the symbolic state, which GDB is not aware of because its purpose is only to execute the application. In most cases, the fuzzer does not need to consider the symbolic state when handling system and library function calls because it does not affect other addresses. This changes when the library calls in case should operate on symbolic memory, as it must be taken into account by taint analysis. For example, after a `memcpy` call, the destination address should have the same symbolic information as the source address, but the symbolic state of the destination address will not be updated by GDB because it cannot interact with symbolized information. In this regard, the current solution relies on hooks of well known, memory altering, functions, but this limits support. To improve this, we must design a more generic approach to handle these calls that require working with the symbolic state.

Chapter 5

Conclusions

5.1 Thesis Summary

Chapter 2 presented our contributions regarding improving the security of the Linux kernel. We focused on the use of the D programming language as a way to improve the security of the Linux kernel. Specifically, we highlighted how D can be used to write drivers for the kernel that are more secure than those written in other languages, such as C. We demonstrated that it is possible to easily convert C code to D without any loss of performance. This is important because it means that developers can use D to write secure drivers without having to sacrifice performance.

In addition to this, we also presented our work on DPP, a tool that automates the process of translating kernel data structures to ones that are compatible with D. This makes it easier for developers to use D to write drivers within the kernel ecosystem, and can help to further improve the security of the Linux kernel. Overall, our contributions aim to make it easier to create secure drivers within the Linux kernel ecosystem.

Chapter 3 presented our contributions to improving the security of applications. One of the main areas we focused on was the development of a new collections framework for the D standard library. This framework is designed to be able to infer the safety of the operations it performs based on the type of data it contains. For example, if the data type is known to be safe, the operations performed on it will also be considered safe. Our collections also support custom memory allocators, the use of which can offer significant performance benefits compared to the existing standard library collections.

Furthermore, we also discussed our work on building a library generator based on OpenAPI Specifications. This tool is designed to help developers quickly and easily create libraries for use in their applications, and it is intended to improve the security and reliability of those applications.

Chapter 4 focuses on validating the security of Internet of Things (IoT) devices. First, it explores the possibility of automatically generating a structured description of an IoT network that could be used by the River fuzzing framework in order to evaluate the network devices.

Next, it presents the work we have done to improve the fuzzing of proprietary binaries with the River framework. We addressed the challenges of performing symbolic execution on COTS binaries. Symbolic execution is a technique used to analyze the behavior of a program by treating certain inputs as symbolic variables, rather than concrete values. This can be useful for detecting and identifying vulnerabilities in software, but it can be difficult to perform on COTS binaries because they are compiled and optimized for specific hardware architectures. We discussed the limitations of current symbolic execution engines and how our solution was able to overcome these issues.

5.2 Contributions

This thesis has presented several contributions that improve the security of IoT applications and the process of auditing such applications.

- We provided a comprehensive background regarding memory safe programming languages, past and current proposals for languages, other than C, to be used in the Linux kernel, and state-of-the-art works related to binary analysis and automated vulnerability detection.
- We presented an approach that employs D in order to improve the security of the Linux kernel.
- 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 developed a methodology for porting kernel drivers to D in order to improve the overall safety of a system.

  owing to the
- We demonstrated that the kernel can leverage D to benefit from safety improvements in a kernel module, array bounds checking and compile-time polymorphism being the most important ones.
- Using our work, further drivers may be ported to D and thus increase the safety mechanisms that are present in the kernel.
- We improved DPP, a C/C++ headers translation tool, in order to enable it's use on Linux kernel headers. Thus, we automate the translation of kernel data structures: an otherwise manual task that is repetitive, time consuming and error prone.
- We validated our work by applying it on our `virtio_net` D port Proof-of-Concept (PoC). This allowed us to reduce the driver implementation size by 53%; a reduced codebase represents a smaller attack surface, thus increasing the overall security.
- We improved automatic code generation for the D ecosystem, which leads to increased productivity and reduced maintenance time. All D projects, not only kernel-related ones, benefit from our work.

- We have developed a new collections framework that brings together all the important features of the D Language, while being easy to use and intuitive for the user. Our collections enable the use of custom, user provided, memory allocators, thus allowing for improved performance.
- We have improved the safety of the standard library allocators with respect to multithreading applications. Our work is actively being used by the D community.
- We implemented an automatic D library generator for services that describe their API throught the OpenAPI Standard. The generated libraries adhere to D's memory safety rules, enabling applications to safely interact with Internet services.
- We proposed an extension to the AGAPIA programming language with IoT-aware deployment macros that enable a fast and simple integration with a testing framework, such as RiverIoT. The extensions describe the relationships and interactions between the IoT devices of a network. The description can be leveraged by the fuzzer in order to discover relationships that may lead to vulnerabilities in the network.
- We have added support for symbolic execution of COTS (commercial off-the-shelf) binaries in the River fuzzer. This was achieved by using GDB to execute syscalls and library functions, which reduces the risk of path explosion and reduces the overhead associated with re-implementing these calls. This approach also demonstrates the feasibility of emulating system and library function calls while maintaining consistency between concrete and symbolic states.

5.3 List of Publications

Journals

Eduard Stăniloiu, Răzvan Nițu, Alexandru Militaru, Răzvan Deaconescu, *"Safer Linux Kernel Modules Using the D Programming Language"*, IEEE Access, doi: 10.1109/ACCESS.2022.3229461, 2022.

Eduard Stăniloiu, Răzvan Nițu, Răzvan Deaconescu, Răzvan Rughiniș, *"A New Collection Framework For the D Programming Language Standard Library"*, U.P.B. Scientific Bulletin Series C, Vol. 84, Iss. 3, Bucharest, Romania, 2022.

Eduard Stăniloiu, Rareș Ștefan Epure, Răzvan Deaconescu, Răzvan Nițu, Răzvan Rughiniș, *"Scalable Concolic Execution of COTS Binaries with the River Framework"*, accepted for publication at U.P.B. Scientific Bulletin Series C, Bucharest, Romania, 2022.

Răzvan Nițu, **Eduard Stăniloiu**, Răzvan Deaconescu, Răzvan Rughiniș, *"Designing Copy Construction for the D Programming Language"*, accepted for publication at U.P.B. Scientific Bulletin Series C, Bucharest, Romania, 2022.

Tarbă, N., Schmidt, D., Popovici, A.E., **Stăniloiu, E.**, Avatavului, C. and Prodan, M., *"On*

Performing Skew Detection and Correction Using Multiple Experts' Decision", Journal of Information Systems & Operations Management (ISSN 1843-4711), 2020.

Dicher, I.M., Ţurcus, A.G., Cojocea, E.M., Penariu, P.S., Bucur, I., Prodan, M. and **Stăniloiu, E.**, *"Unsupervised Merge of Optical Character Recognition Results"*, Journal of Information Systems & Operations Management, pp.60-67., 2020.

Conferences

Răzvan Nițu, **Eduard Stăniloiu**, Răzvan Deaconescu, Răzvan Rughiniş, *"Adding Support for Reference Counting in the D Programming Language"*, Proceedings of the 17th International Conference on Software Technologies (ICSOFT), Lisbon, 2022.

Stăniloiu, E., Nitu, R., Becerescu, C. and Rughinis, R., *"Automatic Integration of D Code With the Linux Kernel"*. In 2021 20th RoEduNet Conference: Networking in Education and Research (RoEduNet) (pp. 1-6). IEEE. 2021.

Staniloiu, E., Cristea, R. and Ghimis, B., *"IoT Fuzzing using AGAPIA and the River Framework"*. In ICSOFT (pp. 324-332). 2021.

Stăniloiu, E., Nitu, R., Aron, R. and Rughiniş, R., *"Extending Client-Server API Support for Memory Safe Programming Languages"*. In 2021 20th RoEduNet Conference: Networking in Education and Research (RoEduNet) (pp. 1-5). IEEE. 2021.

Păduraru, Ciprian, Rareş Cristea, and **Eduard Stăniloiu**. *"RiverIoT-a Framework Proposal for Fuzzing IoT Applications"*. 2021 IEEE/ACM 3rd International Workshop on Software Engineering Research and Practices for the IoT (SERP4IoT). IEEE, 2021.

Nitu, R., **Stăniloiu, E.**, Done, C. and Rughinis, R., *"Security Audit for the D Programming Language"*. In 2021 20th RoEduNet Conference: Networking in Education and Research (RoEduNet) (pp. 1-6). IEEE, 2021.

Nitu, R., **Stăniloiu, E.**, Creteanu, C. and Rughiniş, R., *"Building an Interface for the D Compiler Library"*. In 2021 20th RoEduNet Conference: Networking in Education and Research (RoEduNet) (pp. 1-6). IEEE. 2021.

Lăpuşteanu, A., Boiangiu, C.A., Tarbă, N., Voncilă, M.L., **Stăniloiu, C.E.** and Vlăsceanu, G.V., *"Improving Upon Photographic Steganography"*. In 2021 20th RoEduNet Conference: Networking in Education and Research (RoEduNet) (pp. 1-6). IEEE. 2021.

Patents

Costin-Anton Boiangiu, Giorgiana Violeta Vlăsceanu and **Constantin Eduard Stăniloiu**, *"Method for identifying connected components in binary images"*. Pending approval. Registered at OSIM A/00599, 30.09.2021.

Bibliography

[1] A. Bannister, “Substandard software costs us economy $2tn through security flaws, legacy systems, abandoned projects,” *URL: https://portswigger.net/daily-swig/substandard-software-costs-us-economy-2tn-through-security-flaws-legacy-systems-abandoned-projects*, 2021.

[2] M. Dawson, D. N. Burrell, E. Rahim, and S. Brewster, “Integrating software assurance into the software development life cycle (sdlc),” *Journal of Information Systems Technology and Planning*, vol. 3, no. 6, pp. 49–53, 2010.

[3] “Check point research: Cyber attacks increased 50% year over year - check point software,” https://blog.checkpoint.com/2022/01/10/check-point-research-cyber-attacks-increased-50-year-over-year/, accessed: 2021-10-20.

[4] J. Wurm, K. Hoang, O. Arias, A.-R. Sadeghi, and Y. Jin, “Security analysis on consumer and industrial iot devices,” in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2016, pp. 519–524.

[5] S. Li, L. Da Xu, and S. Zhao, “5g internet of things: A survey,” *Journal of Industrial Information Integration*, vol. 10, pp. 1–9, 2018.

[6] K. V. English, I. Obaidat, and M. Sridhar, “Exploiting memory corruption vulnerabilities in connman for iot devices,” in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 247–255.

[7] G. Hernandez, O. Arias, D. Buentello, and Y. Jin, “Smart nest thermostat: A smart spy in your home,” *Black Hat USA*, no. 2015, 2014.

[8] “Nvd - cvss severity distribution over time,” https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time, accessed: 2021-10-20.

[9] “Iot connected devices worldwide 2019-2030 | statista,” https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide, accessed: 2021-10-20.

[10] A. Radovici, I. Culic, D. Rosner, and F. Oprea, “A model for the remote deployment, update, and safe recovery for commercial sensor-based iot systems,” *Sensors*, vol. 20, no. 16, p. 4393, 2020.

[11] I.-M. Stan, D. Rosner, and Ş.-D. Ciocîrlan, “Enforce a global security policy for user access to clustered container systems via user namespace sharing,” in *2020 19th RoEduNet Conference: Networking in Education and Research (RoEduNet)*. IEEE, 2020, pp. 1–6.

[12] G. Keizer, "Windows comes up third in os clash two years early https://www.computerworld.com/article/3050931/windows-comes-up-third-in-os-clash-two-years-early.html," 2016.

[13] Z. Schuermann and K. Guha, "Linux device drivers in rust https://zachschuermann.com/static/6118.pdf," 2020.

[14] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux device drivers.* " O'Reilly Media, Inc.", 2005.

[15] D. Goodin, "Unpatched linux bug may open devices to serious attacks over wi-fi https://arstechnica.com/information-technology/2019/10/unpatched-linux-flaw-may-let-attackers-crash-or-compromise-nearby-devices," 2019.

[16] H. Ed-Douibi, J. L. Cánovas Izquierdo, and J. Cabot, "Example-driven web api specification discovery," in *European Conference on Modelling Foundations and Applications.* Springer, 2017, pp. 267–284.

[17] "Operating system family / Linux | TOP50," https://www.top500.org/statistics/details/osfam/1/, accessed: 2022-04-17.

[18] W3Techs, "Linux vs. Windows usage statistics for websites," https://w3techs.com/technologies/comparison/os-linux,os-windows, accessed: 2022-04-17.

[19] "Mobile operating system market share worldwide," https://gs.statcounter.com/os-market-share/mobile/worldwide, accessed: 2022-04-17.

[20] AspenCore, "Mobile operating system market share worldwide," https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf, nov 2019, accessed: 2022-04-17.

[21] "Linux kernel CVEs," https://www.linuxkernelcves.com/, accessed: 2022-04-17.

[22] "Kernel self-protection," https://www.kernel.org/doc/html/latest/security/self-protection.html, accessed: 2022-04-17.

[23] A. Popov, "Linux kernel defence map," https://github.com/a13xp0p0v/linux-kernel-defence-map, accessed: 2022-04-17.

[24] "D programming language," https://dlang.org/, accessed: 2022-04-17.

[25] "virtio," https://wiki.libvirt.org/page/Virtio, 2022, accessed: 2022-04-17.

[26] A. Alexandrescu, *The D programming language.* Addison-Wesley Boston, MA, 2010.

[27] "Project highlight: Dpp," https://dlang.org/blog/2019/04/08/project-highlight-dpp/, accessed: 2022-04-17.

[28] W. Bright, A. Alexandrescu, and M. Parker, “Origins of the d programming language,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. HOPL, pp. 1–38, 2020.

[29] P. R. Wilson, “Uniprocessor garbage collection techniques,” in *International Workshop on Memory Management*. Springer, 1992, pp. 1–42.

[30] D. Gregor and S. Schupp, “Making the usage of stl safe,” in *Generic Programming*. Springer, 2003, pp. 127–140.

[31] A. Alexandrescu, “On Iteration,” http://www.informit.com/articles/printerfriendly/1407357, 2009.

[32] E. D. Berger, B. G. Zorn, and K. S. McKinley, “Composing high-performance memory allocators,” *ACM SIGPLAN Notices*, vol. 36, no. 5, pp. 114–124, 2001.

[33] M. Barnett, D. A. Naumann, W. Schulte, and Q. Sun, “99.44% pure: Useful abstractions in specifications,” in *ECOOP workshop on formal techniques for Java-like programs (FTfJP)*. Citeseer, 2004.

[34] U. W. Eisenecker, “Generative programming (gp) with c++,” in *Joint Modular Languages Conference*. Springer, 1997, pp. 351–365.

[35] D. L. Foundation, “What is d?” https://dlang.org/overview.html#what_is_d.

[36] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, “Hypertext transfer protocol–http/1.1,” 1999.

[37] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[38] D. Ashby and C. T. Jensen, *APIs for dummies*. Hoboken, New Jersey: John Wiley & Sons Inc., 2018.

[39] ProgrammableWeb, “Search the largest api directory on the web,” https://www.programmableweb.com/category/all/apis.

[40] Mozilla, “Working with json,” https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON.

[41] I.-M. Culic and A. Radovici, “Development platform for building advanced internet of things systems,” in *2017 16th RoEduNet Conference: Networking in Education and Research (RoEduNet)*. IEEE, 2017, pp. 1–5.

[42] RedHat, “What is a rest api?” https://www.redhat.com/en/topics/api/what-is-a-rest-api.

[43] G. Jansen, “Resources,” https://restful-api-design.readthedocs.io/en/latest/resources.html.

[44] L. Richardson and S. Ruby, *RESTful web services*. ” O’Reilly Media, Inc.”, 2008.

[45] D. Hardt *et al.*, "The oauth 2.0 authorization framework," 2012.

[46] Google, "Using oauth 2.0 to access google apis," https://developers.google.com/identity/protocols/oauth2.

[47] Google, "Google api discovery service," https://developers.google.com/discovery.

[48] P. S. Foundation, "unittest — unit testing framework," https://docs.python.org/3/library/unittest.html.

[49] R. Aron, "D google drive client library," https://github.com/Robert-Aron293/d-google-drive-client.

[50] R. Aron, "D google mail client library," https://github.com/Robert-Aron293/d-google-gmail-client.

[51] C. I. Paduraru, R. Cristea, and E. Staniloiu, "Riveriot - a framework proposal for fuzzing iot applications," *International Conference on Software Engineering ICSE 2021, Workshop on Software Engineering Research and Practices for the IoT (SERP4IoT)*, p. to appear, 2021.

[52] C. I. Paduraru, "Dataflow programming using agapia," in *2014 IEEE 13th International Symposium on Parallel and Distributed Computing*. IEEE, 2014, pp. 87–94.

[53] P. Leonardo, "Child programming: an adequate domain specific language for programming specific robots," 2013.

[54] M. Boshernitsan and M. S. Downes, *Visual programming languages: A survey*. Citeseer, 2004.

[55] C. Paduraru, "Research on agapia language, compiler and applications," *Ph. D. dissertation*, 2015.

[56] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[57] I. T. Banu-Demergian, C. Paduraru, and G. Stefanescu, "A new representation of two-dimensional patterns and applications to interactive programming," in *International Conference on Fundamentals of Software Engineering*. Springer, 2013, pp. 183–198.

[58] C. I. Paduraru and G. Stefanescu, "Adaptive virtual organisms: A compositional model for complex hardware-software binding," *Fundamenta Informaticae*, vol. 173, no. 2-3, pp. 139–176, 2020.

[59] G. Stefanescu and C. I. Paduraru, "Self-assembling heterogeneous interactive systems," in *Proceedings of the International Colloquium on Software-intensive Systems-of-Systems at 10th European Conference on Software Architecture*, 2016, pp. 1–7.

[60] B. Ghimis, M. Paduraru, and A. Stefanescu, "River 2.0: an open-source testing framework using ai techniques," in *Proceedings of the 1st ACM SIGSOFT International Workshop on Languages and Tools for Next-Generation Testing*, 2020, pp. 13–18.

[61] C. Paduraru, M.-C. Melemciuc, and M. Paduraru, "Automatic test data generation for a given set of applications using recurrent neural networks," in *Software Technologies*, M. van Sinderen and L. A. Maciaszek, Eds. Cham: Springer International Publishing, 2019, pp. 307–326.

[62] C. Paduraru. and M. Melemciuc., "An automatic test data generation tool using machine learning," in *Proceedings of the 13th International Conference on Software Technologies - Volume 1: ICSOFT,*, INSTICC. SciTePress, 2018, pp. 472–481.

[63] C. Paduraru, M. Paduraru, and A. Stefanescu, "Optimizing decision making in concolic execution using reinforcement learning," in *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2020, pp. 52–61.

[64] B. Feng, A. Mera, and L. Lu, "P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1237–1254.

[65] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "Halucinator: Firmware re-hosting through abstraction layer emulation," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1201–1218.

[66] C. Paduraru, M.-C. Melemciuc, and A. Stefanescu, "A distributed implementation using apache spark of a genetic algorithm applied to test data generation," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2017, pp. 1857–1863.

[67] M. Wenzl, G. Merzdovnik, J. Ullrich, and E. Weippl, "From hack to elaborate technique—a survey on binary rewriting," *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, pp. 1–37, 2019.

[68] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1497–1511.

[69] P. O'sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. D. Keromytis, "Retrofitting security in cots software with binary rewriting," in *Ifip International Information Security Conference.* Springer, 2011, pp. 154–172.

[70] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making reassembly great again." in *NDSS*, 2017.

[71] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 627–642.

[72] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[73] D. Bruening and Q. Zhao, "Building dynamic tools with dynamorio on x86 and arm," 2017.

[74] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE, 2003, pp. 265–275.

[75] N. Voss, "afl-unicorn: Fuzzing arbitrary binary code," *Hacker Noon*, 2017.

[76] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "{AFL++}: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[77] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[78] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of model checking.* Springer, 2018, pp. 305–343.

[79] L. d. Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 2008, pp. 337–340.

[80] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[81] F. Wang and Y. Shoshitaishvili, "Angr-the next generation of binary analysis," in *2017 IEEE Cybersecurity Development (SecDev).* IEEE, 2017, pp. 8–9.

[82] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[83] "Lief : Library to instrument executable formats," https://lief-project.github.io/doc/latest/intro.html, accessed: 2021-10-20.

[84] F. Saudel and J. Salwan, "Triton: Concolic execution framework," in *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, 2015.

[85] N. A. Quynh, "Capstone: Next-gen disassembly framework," *Black Hat USA*, vol. 5, no. 2, pp. 3–8, 2014.

[86] "Gdb: The gnu project debugger," https://www.sourceware.org/gdb/documentation/, accessed: 2021-10-20.

[87] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[88] C. Paduraru, B. Ghimis, and A. Stefanescu, "Riverconc: An open-source concolic execution engine for x86 binaries." in *ICSOFT*, 2020, pp. 529–536.

[89] C. Paduraru, M. Paduraru, and A. Stefanescu, "Riverfuzzrl-an open-source tool to experiment with reinforcement learning for fuzzing," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 430–435.

[90] "nodejs/http-parser: Http request/response parser for messages written in c," https://github.com/nodejs/http-parser, accessed: 2021-10-20.

[91] "fuzzstati0n/fuzzgoat: A vulnerable c program for testing fuzzers," https://github.com/fuzzstati0n/fuzzgoat, accessed: 2021-10-20.

[92] "google/re: Re2 is a fast, safe, thread-friendly alternative to backtracking regular expression engines like those used in pcre, perl, and python. it is a c++ library." https://github.com/google/re2, accessed: 2021-10-20.

[93] "libarchive/libarchive: Multi-format archive and compression library," https://github.com/libarchive/libarchive, accessed: 2021-10-20.

[94] "Fuzzbench: Fuzzer benchmarking as a service," https://google.github.io/fuzzbench/, accessed: 2021-10-20.

[95] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "Fuzzbench: an open fuzzer benchmarking platform and service," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1393–1403.