VRIJE UNIVERSITEIT AMSTERDAM,
SYSTEMS AND NETWORK SECURITY GROUP (VUSec)

BACHELOR THESIS

AdaFS: An Exploration of the Use of Formal Methods For More Reliable Filesystems

Alexander Balgavý (2619644)

1st supervisor: person
daily supervisor: person

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science.

August 1, 2020

Abstract

Unreliable filesystems have many dangerous implications, worst of all the loss of data. A well-designed filesystem is able to deal with errors by isolating them, repairing broken metadata, and mitigating damage. But could we instead prevent errors from occurring in the first place? In this paper, we take a look at current research done in the area of formal verification of filesystems. We develop a small filesystem in Ada, a language used for high-reliability software in a range of applications. We then evaluate it, finding that the language and its SPARK subset offer many powerful features that do not negatively impact the filesystem's performance.

Contents

List of Figures

List of Tables

Listings

1 Introduction

As K. J. Parker said, "the fastest, cheapest and easiest way to build something is properly the first time" [1]. Software bugs can cost companies customers, reputation, and up to millions of dollars. If these bugs are in the filesystem, they can destroy potentially irreplaceable and priceless data. Unfortunately, given that large filesystem projects contain millions of lines of code, bugs are inevitable. In 2017, a bug in the NT File System used by Windows was found, which allowed anyone to crash Windows 7 or 8.1 [2]. Just this year, a bug in the Apple File System could prevent users from making a bootable clone of their disk [3]. There are currently 112 bugs reported in the Bugzilla database for the Ext4 filesystem, and 611 bugs reported for the Btrfs filesystem [4].

It is safe to say that we need a way to build more reliable software, without spending time and money fixing issues that could have been prevented from the start. C is in widespread use in the development of operating systems and their components, including the Ext4 filesystem. However, C is an inherently unsafe language, and its permissiveness means that errors are relatively easy to make. Attacks exploiting these errors can be devastating, such as the 2001 CodeRed worm that infiltrated enterprise networks [5].

This paper explores alternative approaches to writing reliable software, specifically in the context of filesystems. A small filesystem based on that of MINIX 2 is developed, in Ada and its SPARK subset. Formal verification of some of its parts is conducted, and its effectiveness is analyzed and evaluated.

2 Background information

2.1 C as the implementation language

The choice of an implementation language may affect the bugs or vulnerabilities that are present in the system. The de-facto standard implementation language for operating systems and their components has long been C. Many popular file systems are implemented in C, such as Ext4 [6]. C is based on typeless languages, BCPL and B, which were developed specifically for operating system programming in early Unix. A design principle of C was to be grounded in the operations and data types provided by the computer, while offering abstractions and portability to the programmer [7]. Due to its history, and because C compilers exist for essentially any processor, C is the natural choice for systems programming.

However, many of the characteristics that make C flexible and versatile also place more responsibility on the programmer to ensure that the resulting code is bug-free. The permissiveness of C means that, especially in a large codebase, bugs are hard to avoid and often even harder to troubleshoot. In the worst case, errors in the code allowing null pointer dereferences or buffer overflows can lead to serious security vulnerabilities affecting thousands of devices [8]. In 2017, Ray et al. analyzed 850 projects across 17 different programming languages, and found that those with unmanaged memory, such as C, introduce more memory errors [9]. Furthermore, C introduced 19.15% of concurrency errors. Another study conducted by WhiteSource found that C has the highest vulnerabilities out of all seven analyzed languages, with 50% of all reported vulnerabilities in ten years [10]. Of course, it is important to note that this is in part because more code has been written in C than in any other language, and because C has been in use for much longer than most other languages. WhiteSource note that buffer errors, with the Common Weakness Enumeration (CWE) number CWE-119, are by far the most common security vulnerability in C code.

Specifically in filesystems, a study of 157 cases reported to the Common Vulnerabilities and Exposures (CVE) database between the years 1999 and 2019 was conducted, covering a variety of filesystems including Ext4 and XFS [11]. Cai et al. found that errors that cause denial of service account for 75% of all vulnerabilities. They concluded that the four largest causes of denial of

service are kernel crashes (35% of vulnerabilities), memory corruption (16% of vulnerabilities), memory consumption (13%), and system hang (9%). Kernel crashes and memory corruption can be caused by exploiting invalid pointer dereferences or out-of-bounds memory access, and memory consumption is caused by not properly freeing allocated objects.

In summary, although C is the 'traditional' systems programming language, code written in C can be error prone and lead to issues, unless the programmer is aware of the issues and takes explicit steps to prevent them. Even a simple "Hello World" program exposes some of the most dangerous features of the language [12]. Therefore, it would be beneficial to find a better method, a way to write more reliable systems with less burden resting on the programmer.

2.2   Possible alternatives to C

Given the issues with C code, what alternatives are there? It is possible to subset C and allow only statements or expressions considered suitable for reliable software, or perhaps instead of retroactively imposing restrictions on a language, it would be more beneficial to use a language designed with safety properties in mind.

Restricting C   One option is to use a restricted version of C. MISRA C is a coding standard which was initially aimed at the automotive sector (with MISRA standing for the Motor Industry Software Reliability Association), but which later spread to other sectors that require safety and security in C code [13]. Originally supported by the UK government, the goal of the MISRA project was to develop best practice guidelines for reliable software, and the guidelines also deeply influenced NASA's coding standards. These guidelines prescribe restricting a "standardized structured language" to a subset of its operations, banning non-definite behavior and constraining implementation-defined behavior and compiler extensions. They come in the form of directives (specifications for information that is not fully contained in the code, such as requirements or design) and rules (specifications for code). Every guideline is in one of three categories: mandatory (the code must comply with the guideline), required (the code shall comply with the guideline, and a formal deviation description is required if this is not the case), and advisory (the code should comply with the guideline; formal description of a deviation is not required, but the deviation should be documented). If a piece of C code follows these guidelines, it is easier to verify that safety and security properties hold.

Another approach is Frama-C, a collection of plugins that perform a series of checks (static analysis, deductive verification, testing) on C programs [14]. Frama-C uses the ACSL formal specification language, and extends CIL (a C front-end that normalises ISO C99 programs such that loop constructs have a single form, expressions are free from side-effects, etc.) to support ACSL annotations in the source code. The annotations describe the functional properties of C programs, stating the pre-conditions a function requires from its caller and the post-conditions it ensures when returning. There is also a clause to specify which memory locations are assigned by the function. Furthermore, it is possible to insert annotations directly into the code, as assertions.

The D language   D is a multi-paradigm language for systems programming that is compatible with the C application binary interface, and has a direct interface to the operating system APIs and hardware [15]. It is statically typed, and allows programming with contracts. Specifically, programmers can define assertions, and pre- and post-conditions, which take the form of boolean expressions or blocks of arbitrary code [16]. D also allows the definition of invariants – properties of classes or structs that must always be true [16]. It is possible to use the *pure* keyword to ensure that a function does not access (read or write) any global mutable state, and to preserve referential transparency, function parameters can be marked as immutable [17]. Moreover, D has a memory-safe subset, called SafeD, which also removes undefined behavior. The design team expects the majority of programmers to operate within SafeD [12].

Rust Rust is a relatively new project that attempts to compete with languages such as C as C++ in the systems development space, while putting the largest emphasis on safety [18]. It ensures static safety by forbidding wild and null pointers, and functions are pure by default. All errors cause failure, and task failures are non-recoverable. The static rules can be broken in code, but only if this is explicitly authorised in the code. There is no shared mutable state, and the language has strict ownership rules where each value has exactly one owner at any time [19]. Rust has already been used for several large projects, including the Stratis storage management system[1] and the Redox microkernel operating system[2].

Coq Coq is an interactive theorem prover for development of mathematical proofs, and for verification of programs with respect to their formal specifications [20]. Specifications are written in the Gallina language, whose terms can represent programs, properties of the programs, and proofs of the properties. All three of these are formalised in the Calculus of Inductive Constructions, which is a lambda-calculus with a rich type system. Coq can be used to build certified programs that are relatively efficient, and programs can be extracted in the functional languages OCaml, Haskell, and Scheme. Program extraction is conducted via a constructive proof of its specification. For example, the CompCert compiler[3] for almost the entirety of the C language is largely programmed and proved using Coq.

Ada In 1975, the US Department of Defence (DoD) established the "Common High Order Language" program [21]. Their goal was to select one high order programming language appropriate for their embedded computer systems. A group was created to formulate the requirements for the language, to evaluate existing languages in relation to those requirements, and to select (or implement) the languages. The final, refined version of these requirements received the name Steelman. Ada (Ada83) was specifically designed to meet these requirements, as no other language satisfied them at the time. The Steelman requirements include requirements regarding the maintainability of the language, efficiency, strong typing, and machine independence, among others. For example, requirement 1B (reliability) states that, "the language shall be designed to avoid error prone features and to maximize automatic detection of programming errors" [21]. Wheeler found that Ada95 satisfies 93% of the requirements, while C satisfies only 53% (though it is natural that Ada should satisfy more requirements, given its original purpose). Ada has since changed, with subsequent versions imposing fewer restrictions, becoming supersets of the original version (Ada83). For example, requirement 5D (not permitting access to subprograms) is no longer satisfied in Ada95, because the requirements were found to be too restrictive [21]. However, Ada still meets the majority of these requirements.

Ada has been used successfully in a number of real-world applications that require high reliability and security. For example, in aviation, it has been used in air traffic management systems for the Netherlands and Germany, and Praxis chose AdaCore's GNAT Pro and SPARK as the implementation language for the UK's Interim Future Area Control Tools Support air traffic control system [22]. It has been used in the Airbus 380 aircraft [23], and almost all of the systems in the Boeing 777 jet plane are written in Ada [24]; when Honeywell conducted a study into the benefits of Ada versus C, they concluded that due to Ada's built-in safety features and its accuracy during compilation, less time, expense, and concern would be spent on debugging [24]. Ada has been used in space exploration: it was used in the Atlas V rocket (known for launching NASA's Mars Reconnaissance Orbiter[4] and New Horizons[5] missions), and aboard scientific space vehicles such as the International Space Station, NASA's CloudSat, and the European Space Agency's Huygens

---

[1] https://stratis-storage.github.io/
[2] https://www.redox-os.org/
[3] http://compcert.inria.fr/compcert-C.html
[4] https://www.nasa.gov/home/hqnews/2005/aug/HQ_05218_MRO_success.html
[5] http://pluto.jhuapl.edu/News-Center/Resources/Press-Kits/011607_JupiterPressKit.pdf

probe [23]. It has also been used in railway transport, e.g. for the European Train Control System and the TGV (whose software was converted from C to Ada).

2.3   Formal verification

Chen et al. say that provably correct software has been a long-term goal of computer science [25]. Formal verification is the only way to guarantee this, but though its applications in the area of filesystems have recently increased, it is not yet at the same level of detail and assurance as in other domains [26]. Amani et al. argue that removing human intervention in the verification process is a key step [26].

Parts of a program (namely subprograms) begin running in a state of the host machine. After some computation, they might terminate, and they do so in another state (that is usually different from the original). This state transition can be represented by the Hoare triple, which forms the basis for most verification systems:

$$(\!|\phi|\!)P(\!|\psi|\!)$$

In essence, the above means that if the (sub)program P runs in a state satisfying ϕ, the state after P's execution will satisfy ψ [27]. ϕ is the pre-condition of P, and ψ is the post-condition. The triple is satisfied under *partial correctness* if for all states satisfying ϕ, the state after P satisfies ψ, and P terminates (thus, if P does not terminate, it satisfies its specification). On the other hand, *total correctness* requires that the program terminates, so an infinite loop does not satisfy any specification. Partial correctness is introduced because in some cases, proving total correctness is made easier by first proving partial correctness and then proving termination. Proof rules and calculi exist to prove partial and total correctness, and the aim is for these proofs to be carried out automatically by prover tools.

2.4   Filesystem in Userspace (FUSE)

FUSE, an acronym for Filesystem in Userspace, is a library that allows mounting filesystems in userspace. It consists of two parts: a kernel part, and a user-level daemon [28]. The kernel part is a Linux kernel module that registers a *fuse* filesystem and a block device at */dev/fuse*. In general, the daemon reads requests from, and writes replies to, this block device. FUSE essentially acts as an interface between the mounted filesystem and the kernel. When an application makes a request to the FUSE-mounted filesystem, FUSE calls the appropriate handler function of the mounted filesystem, and sends the filesystem's reply back to the application.

The advantage of this approach is that filesystem development can be done in userspace; i.e., no integration with the kernel is necessary. The filesystem only needs to implement the handlers required by FUSE. This makes the development of a new filesystem much easier, as it is not necessary to worry about kernel-specific elements, or other such issues. However, the disadvantage is that the filesystem may incur performance degradation; if the workload is among those unfriendly to FUSE, its performance may be degraded by up to 83% [28]. In summary, for a system that requires maximum efficiency and reliability, a different solution may be better, but for development, FUSE remains a powerful tool.

3   Design & implementation

To explore ways of writing reliable code, a small, prototype filesystem (AdaFS) was implemented using the Ada 2012 programming language. Some parts of this filesystem are written in the SPARK 2014 language, which is a subset of Ada that removes features not amenable to formal verification, and defines new aspects to support modular, constructive, formal verification

[29]. The AdaCore GNAT Community 2020 package[6] is used, which provides, among others, the compiler and prover tools. For testing purposes, a FUSE driver is written in C, and the built executables are linked with libfuse 3.9.2[7]. The GNAT Project Manager is used to facilitate compilation of source files in different languages and linking of other required libraries.

3.1 Filesystem design

AdaFS is based on the MINIX 2 filesystem [30], with some simplifications due to time constraints. The MINIX operating system was written by Andrew Tanenbaum as an educational tool, and is compatible with UNIX, but has a more modular structure. The MINIX filesystem was chosen as a model because it is not part of the operating system, but rather runs entirely as a user program. As such, it is self-contained. Furthermore, due to its educational purpose, it is thoroughly commented and easy to understand. The second version of the file system was chosen because MINIX 3 is more complex, as it adds numerous improvements (e.g. for reliability) to place emphasis on its use in research and production [31].

A disk is formatted as an AdaFS filesystem using the *mkfs* executable. The resulting disk layout is shown in Figure 1. The disk is divided into blocks of 1024 bytes, similarly to MINIX. Blocks are collected in zones, which can be of size 2^n blocks. This abstraction of blocks into zones can make it possible to allocate multiple blocks at once.

1	2	3..n	n+1..m	m+1..i	i+1..s
boot block	superblock	inode bitmap	zone bitmap	inodes	data.....

Figure 1: AdaFS disk layout. (n = number of inode bitmap blocks, m = number of zone bitmap blocks, i = number of inode blocks, s = number of blocks on disk)

The disk begins with a boot block that would optionally contain executable code. Then, it contains a superblock, and two bitmaps. The bitmaps are used for inode and zone allocation, and can potentially span multiple blocks. Next, there are a few blocks containing space for inodes, potentially with more than one inode per block. Finally, the rest of the blocks contain user data.

The superblock is the second block on the disk, and contains information about the layout of the filesystem. In particular, it contains the number of inodes and zones on disk, the number of inode and zone bitmap blocks, the number of the first data zone, the base-2 logarithm of the number of blocks per zone, the maximum file size, and the magic number. The magic number used to identify a correctly formatted disk is $CACA_{16}$; MINIX uses the magic number 2468_{16}, but AdaFS avoids using the same number because a disk formatted by the AdaFS *mkfs* utility is not necessarily equivalent to a disk formatted by the MINIX *mkfs* utility.

The two bitmaps keep track of available inodes and zones, where the nth bit of the inode or zone bitmap corresponds to the nth inode or zone on disk, respectively. If a bit in the inode bitmap is set to 1, that means the corresponding inode is allocated, and if it is set to 0, the inode is free. This is the same for the zone bitmap, pertaining to zones. One difference with MINIX is that, in MINIX, the first bit (bit zero) in the bitmaps must always be allocated, as the procedure that searches for a free inode or zone returns zero if no free inode/zone is found. In Ada, indexing generally starts at 1, so all bits can be used – as the first bit is bit one, this does not interfere with the allocation procedure.

---

file size
zones
number of links

(a) On disk

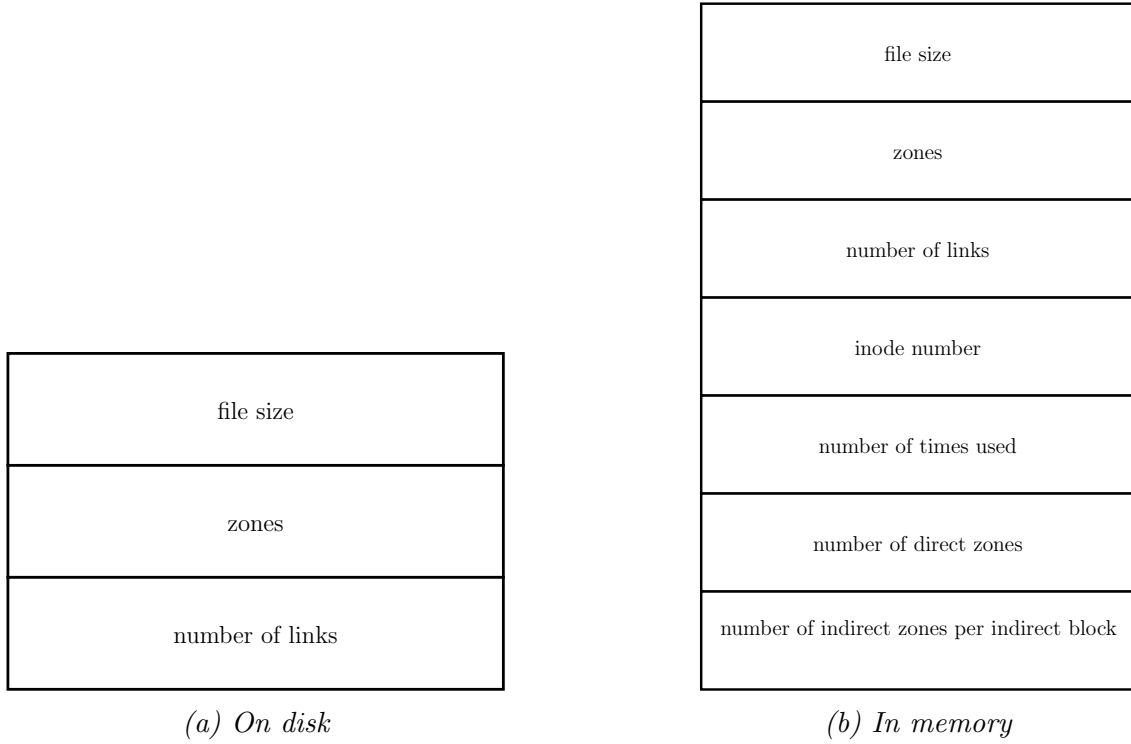| file size |
| --- |
| zones |
| number of links |
| inode number |
| number of times used |
| number of direct zones |
| number of indirect zones per indirect block |

(b) In memory

Figure 2: Inode representations

The penultimate section of the disk contains inodes; the number of inodes is determined by the size of the disk. The inodes have two representations: on-disk (Figure 2a) and in-memory (Figure 2b). This allows the filesystem to make efficient use of disk space, while providing faster access to important values when the inode is loaded in memory. There are 10 zones per inode: 7 direct zones (those that contain data), 1 single indirect zone (indicates a block that contains more direct zones), 1 double indirect zone (indicates a block that contains more single indirect zones), and 1 unused zone. The unused zone is present for future use, potentially as a triple-indirect zone.

The final section spans the rest of the blocks on the disk, broken into zones, and is available to store user data. It can contain file data, or directory entries.

There are also some in-memory structures for working with open files. The filesystem keeps a process table in memory, which is indexed by process ID (PID), and keeps track of inode information for all processes using the filesystem. One entry corresponds to one process, and contains the inode numbers for the root and working directories of the process, as well as the list of open file descriptors. Each open file descriptor corresponds to an entry in the filesystem's *filp* table, which is shared among all processes and contains all the file position. The rationale for a shared file position table comes from MINIX, and is based on problems with the semantics of the *fork* system call [30]. An entry in the filp table contains the number of file descriptors using that entry, the inode number, and the file position for the inode.

Due to time constraints, a number of simplifications were made compared to MINIX. In particular, only the features necessary for basic functionality of the system were implemented. The filesystem does not cache any information, and there is no in-memory inode table; all data are written directly to disk. Furthermore, the current implementation does not keep track of file mode, owner, group, or timestamps. This results in a limited filesystem, which nonetheless completes its function as a proof-of-concept.

3.2 Language features & implementation specifics

Strong typing Ada is a strongly typed language, which helps the programmer distinguish between types that are logically different, even if their underlying representation is the same.

Furthermore, the compiler will automatically catch any bugs that would be caused by assigning a value of an incorrect type to a variable. In order for a value to be assigned to a variable, two constraints must be satisfied: the value and variable must have the same type, and the value must satisfy all constraints on the variable (such as the range for an integral data type) [32]. Conversion between types is allowed, but only if it is explicit, and if the target type is an ancestor of the current type (for example, a positive integer may be converted to an integer, but not vice-versa). An example of the use of types is the definition of a character buffer of arbitrary size, followed by the definitions of different block types including a constrained version of the buffer type, shown in Listing 1.

```ada
type data_buf_t is array (Positive range <>) of Character;

type inode_block_t is array (1..block_size/on_disk'Size) of on_disk;
type zone_block_t is array (1..n_indirects_in_block) of Natural;
type dir_entry_block_t is array (1..block_size/direct'Size) of direct;
subtype data_block_t is data_buf_t (1..data_block_range'Last);
```

Listing 1: Block type definitions

Controlled types Controlled types allow the programmer to specify precisely what happens when a variable of a given type is declared, or when it goes out of scope. They are not permitted in SPARK, because the compiler inserts implicit calls to allow this functionality [29]. The initialization and destruction of variables of a controlled type is handled with custom initialize and finalize procedures for the type, thus the compiler needs to ensure these procedures are called in execution. However, in AdaFS, controlled types are not used for any filesystem logic, and are only used with disk I/O to make sure that e.g. the variable containing the superblock is initialized properly with the type of I/O the filesystem uses. Therefore, this was deemed acceptable for this prototype.

Lack of pointers The C codebase of MINIX makes extensive use of pointers to refer to inodes and other structures in memory, addresses on disk, etc. Ada has access types, which are similar to pointers in some ways, but due to the complexity of verifying a program's behavior when it contains pointers, SPARK severely restricts the possibilities of using access types [29]. Thus, alternative methods must be used to provide similar functionality.

One way to solve this is with parameter modes. Ada allows specifying the mode of each parameter in a procedure, which designates how the parameter will be used in execution. If a parameter is of 'in' mode, it is only read in the procedure, and is not modified – this is the default mode. If a parameter is of 'out' mode, the value of the parameter before the call is irrelevant, as it will receive a value in the procedure. Finally, if a parameter is of 'in out' mode, it is both read and updated in the procedure. This third mode provides functionality similar to a pointer.

However, SPARK requires that functions be purely functional; that is, they cannot have side effects, such as parameters with a mode of 'out' or 'in out'. Here, two solutions are possible. In some cases, it may be preferred to add the return value as an 'out' parameter, and rewrite the function as a procedure. In other cases, it is better for the function to return multiple values, which is possible with a record type. An example is the function parse_next in Listing 2, which returns two values, wrapped in the record type parsed_res_t.

```ada
subtype cursor_t is Natural range path'Range;

type parsed_res_t is record
```

```
   next : adafs.name_t;
   new_cursor : cursor_t;
end record;

function parse_next
   (path : adafs.path_t; cursor : cursor_t) return parsed_res_t
is
   ...
end parse_next;
```

Listing 2: Parse function returning the parsed component and the new cursor position (ellipses denote code omitted for brevity)

Modularisation Ada supports modularisation in the form of packages, and forms a single translation unit, which can contain member entities such as subprograms, variables, and types. Information hiding is done by defining members as private. A package is separated into two parts, which are placed in separate files: the *specification* (the public interface for the package), and the *body* (the implementation). The compiler always checks whether the package body matches the specification, and refuses to compile the code if this is not the case. Listing 3 shows an excerpt of the specification and body of the adafs.inode package.

A package can also have child packages: for example, the adafs.inode package is a child of the adafs package. A child package has access to all member entities defined in the specification of its parent(s), including private members.

```
-- adafs-inode.ads
package adafs.inode
   with SPARK_Mode
is
   ...
   function calc_num_inodes_for_blocks (nblocks : Natural) return Natural
      with ...;
end package adafs.inode

-- adafs-inode.adb
package body adafs.inode
   with SPARK_Mode
is
   function calc_num_inodes_for_blocks
      (nblocks : Natural) return Natural
   is
      inode_max : constant := 65535;
      i : Natural := nblocks/3;
   begin
      ...
      return i;
   end calc_num_inodes_for_blocks;
end adafs.inode;
```

Listing 3: Excerpt from the adafs.inode package specification and body (ellipses denote code omitted for brevity)

It is also possible to create *generics*. Generics are somewhat similar to objects in the OOP

paradigm, in that they can be instantiated with parameters. However, an important difference is that instantiation can only occur in a declarative region. Both subprograms and packages can be generic. For example, Listing 4 shows a generic function for reading data from a disk, and its instantiation. As Ada's Stream_IO requires specifying the type to be read, using a generic function helps with code reuse, with elem_t designating the type to be read, which is passed in at instantiation. The downside of generics is that they cannot be analyzed directly by SPARK, but must instead be verified from the context of instantiation (i.e., SPARK mode must be enabled in the package or subprogram that instantiates the generic) [29].

```ada
-- disk.ads
generic
   type elem_t is private;
function read_block (num : block_num) return elem_t;

-- disk.adb
function read_block (num : block_num) return elem_t is
   result : elem_t;
begin
   if not is_reading then
      sio.set_mode(stream_io_disk_ft, sio.in_file);
   end if;
   go_block (num);
   elem_t'read (stream_io_disk_acc, result);
   return result;
end read_block;
```

Listing 4: Generic function for reading a block of type elem_t

Input & output   In AdaFS, we use a disk image file to represent the filesystem's disk. Therefore, an appropriate way of reading and writing the file was needed. Ada offers several types of input and output (IO), in the form of packages. The simplest is *Text_IO*, which provides sequential file IO for human-readable text only. The package *Sequential_IO* provides sequential access for heterogeneous data (data of varying types). There are also two packages for random-access IO: *Direct_IO* and *Stream_IO*. *Direct_IO* is used for files with homogeneous data (data of a single, uniform type), and *Stream_IO* is for heterogeneous data.

For this project, we required random-access IO, to allow the filesystem to read and write data at any position on the disk. We also preferred heterogeneous IO, to be able to easily read and write data of varying types, such as different types of blocks (a data block, directory block, inode block, etc.). Therefore, we opted for *Stream_IO*, as it best fulfilled our requirements.

Interfacing with other languages   Ada has a mechanism to allow interfacing with other programming languages, such as Fortran, COBOL, or C. This is done by replicating the types and subprogram signatures in Ada. The Interfaces library package[8] provides types and subprograms for this purpose. For example, for C, there are the Interfaces.C[9] and Interfaces.C.Strings[10] packages, which provide the types chars_ptr (mirroring **char∗** in C), int (mirroring **int** in C), etc. This allows *exporting* subprograms from Ada, and calling them from a C program. Listing 5 shows a specification of a subprogram that is exported to C by specifying the Export and Convention

---

[8]https://www.adaic.org/resources/add_content/standards/05aarm/html/AA-B-2.html
[9]http://www.ada-auth.org/standards/12rm/html/RM-B-3.html
[10]http://www.ada-auth.org/standards/12rm/html/RM-B-3-1.html

aspects (as well as an external name to use when calling the subprogram from C). The subprogram is then declared as **extern** in C, and called from the C program's main function. Since the main program is written in a language different from Ada, the initialization and finalization procedures (adainit(**void**) and adafinal(**void**)) must also be declared and called before and after any other Ada subprograms, respectively. The GNAT Project Manager handles compiling and linking of files written in different languages. Unfortunately, interfacing code is not amenable to formal verification.

```ada
-- declarations.ads
with Interfaces.C;
package Declarations is
   function Factorial (n : Interfaces.C.int) return Interfaces.C.int
      with Export => True,
           Convention => C,
           External_Name => "ada_factorial";
end Declarations;
```

```c
-- main.c
#include <stdio.h>
extern void adainit (void);
extern void adafinal (void);
extern int ada_factorial(int n);

int main(int argc, const char *argv[]) {
   adainit();
   int n = 5;
   printf("%d\n", ada_factorial(n)); // 120
   adafinal();
}
```

Listing 5: *Interfacing code written in C and Ada.* declarations.adb *is omitted for brevity, but is assumed to contain an implementation of the factorial function conforming to the specification.*

FUSE & the FUSE driver   FUSE is a software interface that allows running filesystem code in userspace, with FUSE bridging the gap between the filesystem and the kernel. This simplifies the development of filesystems, because access to the kernel and modification of kernel code is not necessary. FUSE allows a filesystem to be developed iteratively; i.e., first it can be implemented and tested with FUSE, and later connected to a kernel if needed.

To implement a filesystem with FUSE, the code needs to be linked with the FUSE library (libfuse), and a driver must be written to specify the filesystem's handler functions for various operations. As FUSE is written in C, the driver for AdaFS is currently also written in C, with a wrapper in Ada to convert values between C types and Ada types. FUSE specifies a **struct** with pointers to functions that should be written by the programmer for the specific filesystem they are implemented. The library also provides, among others, a function to fill file entries into a buffer, a **struct** to store open file information, and a function to get the context of the current operation (such as the PID requesting the operation). Listing 6 shows an example from the driver, with an implementation of the open file operation.

```c
#define FUSE_USE_VERSION 31
#include <fuse.h>
```

```
// Declare the external filesystem open function written in Ada
extern int ada_open(const char *path, pid_t pid);

// The driver's open function
int adafs_open(const char *path, struct fuse_file_info *finfo) {
  pid_t pid = fuse_get_context()->pid;
  int fd = ada_open(path, pid);
  finfo->fh = fd;
  return 0;
}

// Register the function with FUSE
static struct fuse_operations adafs_ops = {
  .open = adafs_open
};

int main(int argc, char **argv) {
  ...
  return fuse_main(argc, argv, &adafs_ops, NULL);
}
```

Listing 6: FUSE driver implementation of open

Formal verification   Unfortunately, much of the code in its current form is not amenable to formal verification. These are namely the parts involving file input and output, and functions that work with C types (i.e. the FUSE driver). To mitigate this, we attempted to make use of an existing project containing FUSE bindings for Ada[11], but as the project has not been maintained since 2016, we were unable to compile it.

Nevertheless, large parts of filesystem logic are formally verifiable. Therefore, the code base was split into two distinct packages: the *adafs* package, which contains filesystem logic that is strictly in the SPARK language, and the *disk* package, which contains unverifiable elements such as disk I/O. The *adafs* package does not include any specifics about the type of disk being used, as the *disk* interface hides implementation details. Thus, when needed, and when an alternative is found, the *disk* package can simply be replaced with a verifiable implementation that exposes an identical interface.

For the parts that are formally verifiable, two types of contracts are available: functional and data contracts. Functional contracts describe how a subprogram should function; that is, the pre- and post-conditions for a given subprogram. They are written as boolean predicate logic expressions. Pre-conditions are evaluated before entry into the subprogram, and post-conditions are evaluated after exit from the subprogram (and can therefore mention the subprogram's result). SPARK can check these conditions at each call site to ensure that no subprogram call violates the conditions, and that the output(s) are shown to be conformant with the specification (the returned result for a function, and the **out** or **in out** parameters for a procedure).

The second type of contract available are data contracts. SPARK conducts flow analysis, which models the flow of information during a subprogram's execution. It checks for uninitialized variables, ineffective statements, and incorrect parameter modes. It is possible to specify which global variables are read, written, or both read and written in the subprogram, using the Global aspect. If no global variables are used, the value of the aspect is set to **null**. It is also possible to specify data dependencies between a subprogram's inputs and outputs.

---

[11]https://github.com/medsec/ada-fuse

For example, Listing 7 shows a function to get an entry from the process table, which specifies the functional and data contracts to be fulfilled for the function. The Global aspect specifies that the function only depends on the value of the package variable tab for input. The Depends aspect states that the result of the function only depends on the tab and pid variables (that is, the variable stated in the Global aspect, and the parameter of the function). The post-condition states that the is_null component of the returned variant record will be set to True if there is no entry in tab for the provided PID; otherwise, the inode number of the PIDs working directory will be non-zero. With the SPARK toolchain, we can verify that these constraints are all satisfied.

```
function get_entry (pid : tab_range) return entry_t with
   Global => (input => tab),
   Depends => (get_entry'Result => (tab, pid)),
   Post => (if tab(pid).is_null
              then get_entry'Result.is_null
              else get_entry'Result.workdir > 0);
```

Listing 7: Functional and data contracts

4 Results & evaluation

4.1 Correctness

As stated earlier, the filesystem was split into two parts: a filesystem logic part (the *adafs* package), and an I/O part (the *disk* package). Because the logic part is easier to verify, it was the primary focus of formal verification for this project. Currently, approximately 25% of the filesystem is verified, and most of the unverified code is in the *disk* package. Since the FUSE driver is written in C, and in Ada with C types, it is also not verified.

Subprograms were given functional and data contracts, and type definitions according to their expected functionality. Although this does not formally verify the functionality of the entire filesystem, it is a good starting point. Listing 8 shows a summary of the conducted checks, and Listing 9 shows some of the locations of various checks.

```
Summary of SPARK analysis
```

SPARK Analysis results	Total	Flow	CodePeer	Provers	Justified	Unproved
Data Dependencies	5	5	.	.	.	.
Flow Dependencies	4	4	.	.	.	.
Initialization	1	1	.	.	.	.
Non-Aliasing	.	.	.	.	.	.
Run-time Checks	17	.	.	17 (CVC4 89%, Trivial 11%)	.	.
Assertions	2	.	.	2 (CVC4)	.	.
Functional Contracts	6	.	.	6 (CVC4 92%, Trivial 8%)	.	.
LSP Verification	.	.	.	.	.	.
Termination	.	.	.	.	.	.
Concurrency	.	.	.	.	.	.
Total	35	10 (29%)	.	25 (71%)	.	.

```
max steps used for successful proof: 1

Analyzed 12 units
```

Listing 8: Prover summary (gnatprove)

```
adafs-inode.adb:6:27: info: division check proved (CVC4: 1 VC in max 0.0 seconds and 1 step)
adafs-inode.adb:6:27: info: range check proved (CVC4: 1 VC in max 0.0 seconds and 1 step)
adafs-inode.adb:8:44: info: division check proved (CVC4: 1 VC in max 0.0 seconds and 1 step)
adafs-inode.adb:13:12: info: overflow check proved (CVC4: 1 VC in max 0.0 seconds and 1 step)
adafs-inode.adb:14:12: info: division check proved (CVC4: 1 VC in max 0.0 seconds and 1 step)
adafs-inode.adb:14:12: info: range check proved (CVC4: 1 VC in max 0.0 seconds and 1 step)
adafs-inode.adb:14:28: info: overflow check proved (CVC4: 1 VC in max 0.0 seconds and 1 step)
adafs-inode.ads:40:60: info: range check proved (CVC4: 1 VC in max 0.0 seconds and 1 step)
```

```
adafs−inode.ads:66:10:  info:  data dependencies proved
adafs−inode.ads:67:10:  info:  flow dependencies proved
adafs−inode.ads:69:18:  info:  postcondition proved (CVC4: 2 VC in max 0.0 seconds and 1 step)
adafs−proc.adb:6:14:  info:  discriminant check proved (CVC4: 2 VC in max 0.0 seconds and 1 step)
adafs−proc.adb:12:16:  info:  discriminant check proved (CVC4: 2 VC in max 0.0 seconds and 1 step)
adafs−proc.ads:26:05:  info:  data dependencies proved
adafs−filp.adb:18:30:  info:  loop invariant initialization proved (CVC4: 1 VC in max 0.0 seconds and 1 step)
adafs−filp.adb:18:30:  info:  loop invariant preservation proved (CVC4: 1 VC in max 0.0 seconds and 1 step)
adafs−filp.ads:28:13:  info:  postcondition proved (CVC4: 3 VC in max 0.0 seconds and 1 step)
adafs−filp.ads:32:28:  info:  initialization of "free_fd" proved
adafs−filp.ads:33:05:  info:  data dependencies proved
adafs−filp.ads:34:05:  info:  flow dependencies proved
adafs−filp.ads:35:13:  info:  postcondition proved (CVC4: 2 VC in max 0.0 seconds and 1 step)
```

*Listing 9: Excerpt from the prover output (*gnatprove*)*

The checks done by the prover can each be related directly to a Common Weakness Enumeration (CWE) number; these are shown in Table 1 [33].

Message	CWE number
divide by zero	369
index check	120
overflow check	190
fp_overflow check	739
range check	682
predicate check	682
predicate check on default value	682
null pointer dereference	476
memory leak	401
memory leak at end of scope	401
discriminant check	136
tag check	136
use of an uninitialized variable	457
precondition	628
postcondition	682

Table 1: SPARK checks with their respective CWE numbers

Due to Ada's type system, many of the checks do not take much effort to verify; in fact, all of the conducted checks were verified in one step. The assurances provided by the prover are powerful, as they mean that e.g. overflow errors and range errors can *never* happen at those locations. In standard Ada, instructions are emitted to check that e.g. array accesses are within the bounds of the array. However, with SPARK, such verifications can be conducted ahead of time. This means that the runtime checks done by Ada can be safely disabled, eliminating a number of machine instructions and thus improving the performance of the system.

4.2 Performance

We compare the performance of AdaFS with a C implementation of a FUSE-based MINIX 3 filesystem, fuse-mfs[12]. As AdaFS is not a complete filesystem, but only supports a small set of operations, it was not possible to use a standardised utility to test it. Therefore, we created a microbenchmark in C that executes the relevant system calls and counts the number of CPU clock cycles necessary for each operation. We compared some common filesystem operations, such as creating, writing, reading, and deleting files. We used a virtual machine in VirtualBox 6.1.10 on a macOS host, running Debian 4.19.118, with FUSE version 3.4.1. The machine has 4 CPUs, 2 GB of RAM, and a Intel Core i7-3615QM processor at 2.3 GHz. The disk images for the filesystem were stored on an SSD.

To calculate the number of CPU clock cycles an operation takes, we use a deterministic version of the ReaD Time Stamp Counter (RDTSC) instruction, RDTSCP, which is added as inline assembly in the C microbenchmark code. The code to calculate elapsed clock cycles is shown in Listing 10.

```
static __inline__ int64_t rdtsc_s(void) {
  unsigned a, d;
  asm volatile("cpuid" ::: "%rax", "%rbx", "%rcx", "%rdx");
  asm volatile("rdtsc" : "=a" (a), "=d" (d));
  return ((unsigned long)a) | (((unsigned long)d) << 32);
}

static __inline__ int64_t rdtsc_e(void) {
  unsigned a, d;
  asm volatile("rdtscp" : "=a" (a), "=d" (d));
  asm volatile("cpuid" ::: "%rax", "%rbx", "%rcx", "%rdx");
  return ((unsigned long)a) | (((unsigned long)d) << 32);
}
typedef int64_t cycle_t;

int main() {
  cycle_t cycles_before, cycles_after, cycles_per;
  cycles_before = rdtsc_s();
  // Call the test function here...
  cycles_after = rdtsc_e();
  cycles_per = cycles_after-cycles_before;
  printf("CPU_cycles:_%ld\n", cycles_per);
}
```

Listing 10: Calculating clock cycles

We choose RDTSCP instead of RDTSC because the CPU cannot reorder it relative to other instructions, as this reordering would cause noise in our results [34]. We set the isolcpus kernel option at boot to isolate CPU 1, and then use the sched_affinity function call to explicitly schedule each test on this CPU. Listing 11 shows the code for this.

```
#define _GNU_SOURCE
#include <sched.h>
void run_test(void) {
  // Schedule
  cpu_set_t the_cpu;
```

---
[12]https://github.com/redcap97/fuse-mfs

```
    int the_cpu_num = 1;
    CPU_ZERO(&the_cpu);
    CPU_SET(the_cpu_num, &the_cpu);
    if (sched_setaffinity(0, sizeof(the_cpu), &the_cpu) == -1) {
      perror("setaffinity_failed");
    }

    // Run any tests below
    ...
  }
```

Listing 11: Schedule a test on CPU 1

We also assign interrupts to CPU 0 to avoid interference. This ensures that no other processes are scheduled on the same CPU as our tests, i.e. that our results are as noiseless as possible. We run FUSE in single-thread mode for both implementations of the filesystem.

We run the benchmark 30 times for each filesystem implementation; Table 2 presents the median values for each implementation and operation. It is apparent that AdaFS is significantly slower than fuse-mfs. The largest difference is apparent in the create operation, which is approximately 412.5 times slower. The remove operation is around 2.5 times slower than that of fuse-mfs. For other operations, the difference is not as drastic, though the Ada implementation is still around 10-20% slower on average.

Implementation	Create		Remove	
	Cycles	Std. dev.	Cycles	Std. dev.
AdaFS	548615020	16668215	516977	146179
fuse-mfs	1329948	1709262	208077	267264

(a) CPU cycles to create and remove files

Implementation	Read 1 B		Read 1 KB		Read 10 KB		Read 100 KB	
	Cycles	Std. dev.	Cycles	Std. dev.	Cycles	Std. dev.	Cycles	Std. dev.
AdaFS	5852	733	2165	154	2172	3813	2129	166
fuse-mfs	5144	3262	1713	525	1698	537	1703	516

(b) CPU cycles to read files of varying sizes

Implementation	Write 1 B		Write 1 KB		Write 10 KB		Write 100 KB	
	Cycles	Std. dev.	Cycles	Std. dev.	Cycles	Std. dev.	Cycles	Std. dev.
AdaFS	2095	226	1953	190	1917	372	2014	326
fuse-mfs	1898	492	1621	403	1744	358	1779	348

(c) CPU cycles to write files of varying sizes

Table 2: Comparing filesystem operation performance in C (fuse-mfs) and Ada (AdaFS), showing the median and standard deviation across 30 observations.

These differences in performance are mostly small enough that they are not generally noticeable in everyday life. However, a create operation that takes 0.24 seconds (548615020 cycles at 2.3 GHz) may have a visible impact, and the other operations may also cause issues in applications where speed is critical. Thus, it is important to find out why this slowdown happens, or eliminate factors that do not cause decreased performance. It may not necessarily be due to the extra code needed for formal verification; AdaFS does not employ any optimisations or caching that MINIX 2 uses, and MINIX 3 is even more optimised for speed than MINIX 2. For example, part of the *create* operation is the creation of a new inode. Among other things, MINIX keeps track of

free inodes in a table in memory, which is flushed to disk at various points. This makes looking up a free inode much faster, improving the performance of the *create* operation. Due to time constraints, AdaFS does not implement this table, and thus has to read it from the disk. This is also the case with other similar mechanisms.

To determine if the verification code or implementation language may play a role in the slowdown, we implemented two versions of a factorial function for numbers up to 5, in C and Ada. In Ada, we use the same settings and pragmas as AdaFS. The Ada version of the function includes a full formal specification, with data and functional contracts, and with type definitions. The Ada function is exported and called from C. The C version follows the Ada version as closely as the language allows.

We run both implementations 500 times, counting the clock cycles. The results (median and standard deviation) are shown in Table 3. From this, we can see that the Ada implementation only takes 8 more clock cycles than the C implementation. Therefore, the implementation language and formal verification code are likely not major contributors to the decreased performance we see in AdaFS. The performance degradation must thus be caused by other differences in the implementations.

Implementation	Cycles	Std. dev.
Ada	132	55.21
C	124	38.33

Table 3: Factorial function comparison in Ada and C.

Another potential source of performance degradation could be the choice of IO package. Ada offers two packages for random-access data IO: *Stream_IO* (heterogeneous data) and *Direct_IO* (homogeneous data). In this project, we opted for the former, but the latter may offer better performance. For example, because the implementation of *Stream_IO* might include some code to manipulate incoming or outgoing data into a suitable format. To test whether this is the case, we implemented a function that opens a file, reads a record from the file, modifies the record, writes the modified record to the file, and closes the file. We created two implementations, one with *Direct_IO*, and one with *Stream_IO*. Both implementations exported functions to C, and both were called from C; this helped eliminate any potential clock cycle differences at the boundary between C code and Ada code. We used the same setup as for the other tests (i.e. schedule the tests on an isolated CPU, and count clock cycles with RDTSC), and ran each implementation's function 500 times. The results (median and standard deviation) are shown in table Table 4.

Implementation	Cycles	Std. dev.
Stream IO	1463269	664984
Direct IO	1953586	190209

Table 4: Random-access IO implementation comparison in Ada.

We can see that *Stream IO* is actually faster than Direct IO, taking around 25% fewer clock cycles for the same operations. One reason might be because Direct IO could have some checks in place to verify that the data is indeed homogeneous. The difference in clock cycles could also be the result of switching between reading and writing modes. *Stream IO* has a dedicated function to set the mode of the file, which may be optimised in some way, while with Direct IO, the file has to be closed and reopened with the new mode. Therefore, for the purposes of AdaFS where switches between file modes are common, *Stream IO* seems more performant. It could also be worthwhile to compare *Stream IO* with one of the sequential IO packages, but as using only sequential reads and writes would require a different filesystem design or some levels of abstraction, it is not covered in this paper.

5   Related work

In this section we summarise the existing body of work relating to formal verification of filesystems.

5.1   Flashix

In 2014, Schellhorn et al. presented their work on a verified Flash filesystem [35]. Since Flash memory can only be written sequentially, and data in Flash memory cannot be overwritten in place (i.e. space can only be reused by erasing entire blocks), a special Flash file system must be used that is designed to work with Flash memory. Flashix, the verified Flash filesystem they developed, is based on UBIFS. They refined the high-level POSIX system interface using a Virtual Filesystem Switch (VFS) that maps POSIX operations to one or more Abstract File System (AFS) operations; the AFS is a model that captures the functional behavior of a specific filesystem [36]. The reason for this separation is that specific filesystems do not implement generic functionality, but instead satisfy an internal interface. To provide safety for crashes, power cuts, and other such events, they used a transactional journal providing atomic writes [37]. They verified the code using the KIV interactive theorem prover, whose specification language is based on Abstract State Machines and Abstract State Machine refinement. They then used tools to generate a Scala implementation, which could be mounted using the FUSE library and executed on the Java Virtual Machine.

5.2   COGENT

COGENT is a new, restricted language developed by Amani et al. [26], as an approach to writing and formally verifying high-assurance filesystem code. It was designed to bridge the gap between the formal specification of a program and its low-level implementation, and to allow programmers that do not have a background in formal verification to provably avoid common errors. Many of the characteristics required to guarantee the absence of common filesystem implementation errors were encoded in the language, which is strongly typed, type safe, and uses a linear type system (meaning that every variable must be used exactly once). The compiler for the language generates C code and translation correctness proofs, enforces memory safety, and forbids undefined behavior in C code (such as null pointer dereferences or buffer overflows). In evaluation, they found that code generated from COGENT has throughput that is almost identical to a C implementation, albeit with slightly increased CPU usage. The disadvantage is that programmers were required to learn a new language (which has a functional style and whose type system is less common) solely for the development of a filesystem, though the authors state that these were not major barriers.

5.3   Recon

Fryer et al. found that filesystem bugs that severely corrupt metadata are common, and that solutions to the necessary recovery procedures were unsatisfactory. Therefore, they developed Recon, a system to protect filesystem metadata from arbitrary implementation bugs [38]. Recon sits between the filesystem and the block layer, and checks consistency invariants at commit points before allowing writes to disk. Thus, failures that would be silent become detectable violations of these invariants. Consistency invariants are declarative statements that must be satisfied before data is committed to disk, otherwise the filesystem may become corrupted. The authors use the consistency rules used by a file system checker (*e2fsck* in their implementation) to derive the invariants. Consequently, the system can detect random corruption at runtime as effectively as a filesystem checker. In essence, Recon can conduct online checks similar to those conducted by offline filesystem checkers (e.g. *fsck*).

5.4   FSCQ & DFSCQ

FSCQ was the first file system that has a machine-checkable proof (as opposed to an interactive proof) that its specification and implementation match, and whose specification included crashes [25]. Chen et al. found that to achieve their design goals of atomic system calls, preventing real bugs, enabling proof automation, and allowing modularity, an extended variant of Hoare logic worked best. Accordingly, they developed Crash Hoare logic, which allows programmers to write a specification of the behavior of a storage system in the face of a crash, and to prove them correct (i.e. if a computer crashes, the storage system will reboot into a state consistent with its specification). This extension of Hoare logic takes into account the fact that during a crash, a procedure may stop at any point, and that recovery procedures may run. The filesystem uses FscqLog, a write-ahead log also certified with Crash Hoare logic, which provides atomic transactions on top of asynchronous disk writes. FSCQ was developed with the same features as the educational xv6 filesystem, which implements the Unix v6 filesystem with write-ahead logging. The implementation used the Coq proof assistant, and generated Haskell code, which could be mounted with the FUSE library and a Haskell driver. Based on FSCQ, DFSCQ (Deferred-write FSCQ) was later written to provide a precise specification for *fsync* and *fdatasync* in the case of log-bypass writes [39]. Deferring writing data to persistent storage allow filesystem to achieve high I/O performance, and DFSCQ's implementation would provide crash safety for these operations. In building DFSCQ, the authors presented a tree-based approach to specifying filesystem behavior, and a metadata-prefix specification to specify behavior for crashes. Compared to FSCQ, DFSCQ has several optimisations and provides a number of missing features.

5.5   Yggdrasil & Yxv6

In 2016, Sigurbjarnarson et al. presented Yggdrasil, a toolkit which uses a push-button approach to formal verification [40]. Yggdrasil does not require manual annotations or proofs, and aids programmers by producing counterexamples for failed verification constraints. To achieve this, the authors used *crash refinement*, i.e. that the set of all disk states reachable in the implementation must be a subset of those allowed in the specification. Push-button verification means that Yggdrasil asks the programmer to enter the specification of the expected behavior, the implementation, and any consistency invariants for the state of the filesystem image, and it then checks if the implementation meets the specification while satisfying the invariants. All three inputs are written in the same language (a subset of Python), and Yggdrasil generates C code, which is compiled to an executable filesystem and can be mounted using the FUSE library. The authors used Yggdrasil to implement a journaling filesystem, Yxv6, which is similar to xv6 and FSCQ, as well as a file copy utility (Ycp) and a persistent log (Ylog).

5.6   Argosy

Argosy is a framework to allow machine-checked verification of storage systems, and introduces recovery refinement, which is a set of conditions that guarantee that an implementation of an interface with a recovery procedure is correct [41]. Recovery refinement ensures correctness for anything using the specification, and can compose with other refinements to prove that an entire system is correct. Its semantics are formulated in Kleene algebra. The system implements Crash Hoare logic, which was introduced by and used in FSCQ, to prove recovery refinement. Therefore, the authors produce a verified transactional disk API for unreliable disks. Similarly to FSCQ, the code is verified using the Coq proof assistant, and produces Haskell code.

6 Conclusion

This paper has aimed to explore ways of writing reliable software, in particular reliable filesystems. "A computer program does what you tell it to do, not what you want it to do" [42], thus we have investigated methods to allow the programmer to specify what they want a filesystem to do, so that it performs accordingly with minimal or no maintenance effort. We highlighted Ada and its SPARK subset, a programming language that has seen great success in industries requiring high-reliability software such as aviation and space exploration, but which has not yet, to the best of our knowledge, been used for the development of a filesystem. As a proof-of-concept, we developed AdaFS, a small filesystem with components that are formally verified to perform according to their specification. In our evaluation, we found that formal verification, as well as the switch from the traditional choice of C to Ada, does not significantly impact performance.

The aim of this paper is not to criticise or dismiss C, nor to suggest Ada as the sole alternative. It is rather to explore tools that have not seen much use yet in the domain of filesystems. Writing good, reliable C code is possible; one only needs to look at the number of large, widely-used C projects to see this. Nonetheless, we hope this paper has demonstrated that Ada is equally suitable as a low-level systems development language, while providing features that allow programmers to ensure that their code runs exactly as intended, such as functional and data contracts. These features not only provide developers with compile-time guidance, but also allow tools to prove the correctness of a program, which is more difficult to do for code written in C. Given that Ada was designed specifically for cases where reliability is the primary concern, it only seems natural that it would excel in those cases.

Future work still needs to be done in this area to produce a filesystem that is suitable for regular use. Many crucial optimisations and caching mechanisms are missing, and would need to be implemented. The I/O part of the filesystem would need to be formally verified, perhaps using a variant of the Crash Hoare logic developed for other projects discussed in section 5 to include system crashes in the specification. Furthermore, the filesystem could be integrated with e.g. a formally verified operating system kernel [43], instead of running in userspace. This would provide a solid foundation for reliable system development.

References

[1] K. J. Parker, *The escapement*. New York: Orbit, 2007, ISBN: 0316003409.

[2] P. Bright, *In a throwback to the '90s, NTFS bug lets anyone hang or crash Windows 7, 8.1*, https://arstechnica.com/information-technology/2017/05/in-a-throwback-to-the-90s-ntfs-bug-lets-anyone-hang-or-crash-windows-7-8-1/, May 2017. (visited on 07/05/2020).

[3] M. Bombich, *A bug in macOS 10.15.5 impacts bootable backups, but we've got you covered*, https://bombich.com/blog/2020/05/27/bug-in-macos-10.15.5-impacts-bootable-backups-weve-got-you-covered, May 2020. (visited on 07/05/2020).

[4] *Kernel.org Bugzilla – bug list*, https://bugzilla.kernel.org/. (visited on 07/05/2020).

[5] Trend Micro, Inc., "Enterprise prevention and management of mixed-threat attacks: Why current antivirus and content security approaches are limited," Dec. 2002.

[6] *Ext4 file system tree*, https://git.kernel.org/pub/scm/linux/kernel/git/tytso/ext4.git/. (visited on 06/23/2020).

[7] D. M. Ritchie, "The development of the C language," in *The Second ACM SIGPLAN Conference on History of Programming Languages*, ser. HOPL-II, Cambridge, Massachusetts, USA: Association for Computing Machinery, 1993, pp. 201–208, ISBN: 0897915704. DOI: 10.1145/154766.155580. [Online]. Available: https://doi.org/10.1145/154766.155580.

[8] CERT Division, "2001 CERT advisories," Carnegie Mellon University, Tech. Rep., 2017.

[9] B. Ray, D. Posnett, P. Devanbu, and V. Filkov, "A large-scale study of programming languages and code quality in GitHub," *Commun. ACM*, vol. 60, no. 10, pp. 91–100, Sep. 2017, ISSN: 0001-0782. DOI: 10.1145/3126905. [Online]. Available: https://doi.org/10.1145/3126905.

[10] WhiteSource, "What are the most secure programming languages?" WhiteSource Software, Tech. Rep., 2019. (visited on 07/02/2020).

[11] M. Cai, H. Q. Huang, and J. Huang, "Understanding security vulnerabilities in file systems," in *APSys '19*, 2019.

[12] B. Milewski, *SafeD*, https://dlang.org/articles/safed.html. (visited on 07/03/2020).

[13] R. Bagnara, A. Bagnara, and P. M. Hill, "The MISRA C coding standard and its role in the development and analysis of safety- and security-critical embedded software," *ArXiv*, vol. abs/1809.00821, 2018.

[14] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C - a software analysis perspective," in *SEFM*, 2012.

[15] D Language Foundation, *Welcome to D*, https://tour.dlang.org/. (visited on 07/03/2020).

[16] ——, *Contract programming*, https://dlang.org/spec/contracts.html. (visited on 07/03/2020).

[17] D. Nadlinger, "Purity in D," May 2012. (visited on 07/03/2020).

[18] G. Hoare, *Technology from the past come to save the future from itself*, http://venge.net/graydon/talks/intro-talk-2.pdf, Jul. 2010. (visited on 07/03/2020).

[19] S. Klabnik and C. Nichols, *The Rust Programming Language*. No Starch Press, Aug. 2019.

[20] *Coq reference manual*, https://coq.inria.fr/distrib/current/refman/. (visited on 07/03/2020).

[21] D. A. Wheeler, "Ada, C, C++, and Java vs. the Steelman," *ACM Sigada Ada Letters*, pp. 88–112, 1997.

[22] AdaCore, *GNAT Pro chosen for UK's next generation ATC system*, https://www.adacore.com/press/adacore-gnat-pro-chosen-for-uk-next-generation, Jun. 2007. (visited on 07/03/2020).

[23] M. B. Feldman, *Who's using Ada? real-world projects powered by the Ada programming language*, https://www2.seas.gwu.edu/~mfeldman/ada-project-summary.html, Nov. 2014. (visited on 07/03/2020).

[24] Ada Information Clearinghouse, *Boeing flies on 99% Ada*, http://archive.adaic.com/projects/atwork/boeing.html. (visited on 07/03/2020).

[25] H. Chen, D. Ziegler, A. Chlipala, M. F. Kaashoek, E. Kohler, and N. Zeldovich, "Specifying crash safety for storage systems," in *HotOS*, 2015.

[26] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, J. Tuong, G. Keller, T. C. Murray, G. Klein, and G. Heiser, "COGENT: Verifying high-assurance file system implementations," in *ASPLOS '16*, 2016.

[27] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*, 2nd ed. Cambridge: Cambridge University Press, 2004. DOI: 10.1017/CBO9780511810275.

[28] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To FUSE or not to FUSE: Performance of user-space file systems," in *FAST*, 2017.

[29] AdaCore and Altran UK Ltd, *SPARK 2014 reference manual*, http://docs.adacore.com/live/wave/spark2014/html/spark2014_rm/index.html. (visited on 06/26/2020).

[30] A. Tanenbaum, *Operating systems: design and implementation*, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 1997, ISBN: 978-0136386773.

[31] *History of MINIX 3*, https://wiki.minix3.org/doku.php?id=www:documentation:read-more, 2014. (visited on 06/26/2020).

[32] J. Barnes, *Programming in Ada 2012*. Cambridge: Cambridge University Press, 2014, ISBN: 978-1-107-42481-4.

[33] AdaCore and Altran UK Ltd, *SPARK 2014 user's guide*, https://docs.adacore.com/spark2014-docs/html/ug/index.html. (visited on 07/05/2020).

[34] D. J. Delorie, *Practical micro-benchmarking with 'ltrace' and 'sched'*, https://developers.redhat.com/blog/2016/03/11/practical-micro-benchmarking-with-ltrace-and-sched/, Accesed on: 2020-07-20, Mar. 2016.

[35] G. Schellhorn, G. Ernst, J. Pfähler, D. Haneberg, and W. Reif, "Development of a verified Flash file system," in *ABZ*, 2014.

[36] G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif, "A formal model of a virtual filesystem switch," in *SSV*, 2012.

[37] G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif, "Inside a verified Flash file system: Transactions and garbage collection," in *VSTTE*, 2015.

[38] D. Fryer, K. Sun, R. Mahmood, T. Cheng, S. Benjamin, A. Goel, and A. D. Brown, "Recon: Verifying file system consistency at runtime," *TOS*, vol. 8, 15:1–15:29, 2012.

[39] H. Chen, T. Chajed, A. Konradi, S. Wang, A. M. Ileri, A. Chlipala, M. F. Kaashoek, and N. Zeldovich, "Verifying a high-performance crash-safe file system using a tree specification," *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[40] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang, "Push-button verification of file systems via crash refinement," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA: USENIX Association, Nov. 2016, pp. 1–16, ISBN: 978-1-931971-33-1. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/sigurbjarnarson.

[41] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich, "Argosy: Verifying layered storage systems with recovery refinement," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019, Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 1054–1068, ISBN: 9781450367127. DOI: 10.1145/3314221.3314585. [Online]. Available: https://doi.org/10.1145/3314221.3314585.

[42] A. Bloch, *Murphy's law and other reasons why things go wrong*. Los Angeles: Price/Stern/Sloan, 1977, ISBN: 0843104287.

[43] H. King, "Development tools & techniques for a more robust operating system," 2019.