Deductive Verification of Unmodified Linux Kernel Library Functions

Denis Efremov[1,2], Mikhail Mandrykin[2], and Alexey Khoroshilov[1,2,3,4]

[1] National Research University Higher School of Economics, Moscow, Russia
[2] Ivannikov Institute for System Programming of the RAS, Moscow, Russia
[3] Moscow Institute of Physics and Technology, Moscow, Russia
[4] Lomonosov Moscow State University, Moscow, Russia
defremov@hse.ru, {mandrykin,khoroshilov}@ispras.ru

Abstract. This paper presents results from the development and evaluation of a deductive verification benchmark consisting of 26 unmodified Linux kernel library functions implementing conventional memory and string operations. The formal contract of the functions was extracted from their source code and was represented in the form of preconditions and postconditions. The correctness of 23 functions was completely proved using AstraVer toolset, although success for 11 functions was achieved using 2 new specification language constructs. Another 2 functions were proved after a minor modification of their source code, while the final one cannot be completely proved using the existing memory model. The benchmark can be used for the testing and evaluation of deductive verification tools and as a starting point for verifying other parts of the Linux kernel.

Keywords: formal verification, deductive verification, Linux kernel

1 Introduction

Deductive verification is one of the most rigorous techniques to ensure software satisfies its requirements. In spite of significant advances in tool support, it still requires deep user involvement in the verification process to provide manual guidance (e. g., to specify the contract of each function and to identify loop invariants). As a result deductive verification is used mainly to analyze the most critical pieces of software.

Under such conditions, it is more cost-effective to rewrite code to make it easier to verify than to implement support for all the complex corner cases in the semantics of the target programming platform, most notably in low-level platforms based on C which lack well-defined semantics for many cases widely used in practice.

Nevertheless, there are situations where changing the code under verification is undesirable or even impossible. For example, components to be integrated into a predefined framework have to follow the coding style, interfaces and data structures of that framework. We have met such limitations with Linux kernel

modules where a lot of implementation details are imposed by Linux kernel core infrastructure.

One of the well-established approaches to specifying the behavioral contract of functions written in C is ANSI/ISO C Specification Language (ACSL) [1]. Frama-C [2] provides a framework for an analysis of C programs with optional ACSL specification annotations. Frama-C integrates specifications and code into a single intermediate representation and allows plugins to work with it. There are two plugins for deductive verification built on top of Frama-C: WP [2] and Jessie [3].

Because existing plugins were not able to correctly handle many constructs widely used in the Linux kernel (e. g., `container_of`, pointer type reinterpretation between integer types of different size), we started developing a new deductive verification AstraVer plugin based on Jessie. We implemented and proposed many improvements for the toolchain, including a new memory model [4], but there is no representative benchmark to evaluate the progress. The primary purpose of this work is to fill this gap.

Following previous efforts [5, 6], we have chosen for the first step Linux kernel library functions implementing conventional memory and string operations. The benchmark built from such functions helped us to detect a number of local tool issues and several fundamental problems discussed below.

The main contributions of the paper include:

- a benchmark of unmodified Linux kernel library functions extended with annotations formalizing their contracts in the form of preconditions and postconditions [7];
- a new approach to annotate modulo arithmetic operations on values of integral C types;
- evaluation of AstraVer deductive verification toolchain on the benchmark.

The paper is organized as follows. Section 2 discusses similar efforts aimed at the specification and deductive verification of C library functions. Section 3 provides background on ACSL basic concepts. Section 5 presents improvements in the toolchain made during the development of the benchmark. Sections 6 describes specification techniques designed and applied for specification of library functions. Section 7 defines a set of open problems. Section 8 presents the evaluation of the solvers. Section 9 summarizes the results of the work.

2 Related Work

Since the deductive verification tools, WP and Jessie are mature enough there are many examples where these tools were applied for verification of real-life software. In [6] 12 string functions from OpenBSD were examined, using Jessie as a deductive verification plug-in. The correctness of 7 functions was fully proved (all verification conditions, or VCs, were successfully discharged). For the other 5 functions, some VCs were left unproved. The author did three iterations on the development of a specification contract for each function. First, one was developed based on the standard and the author's experience. The second one was developed based on informal documentation (man pages) exclusively. The final one was

written based on the implementation (source code) and the man pages. The final revision in most cases has significant differences from previous versions. Thus it shows that it is difficult to develop a formal specification in ACSL language for already developed source code without taking the implementation into account. However, such an iterative approach allowed the author to find inaccuracies in the documentation for several functions, and a lack of documentation completeness in many cases.

To prove some functions, the author changed the source code. Changes were performed in two specific situations. In the first case `char *` type in `strcmp` and `strncmp` functions were cast to `unsigned char *`. In the second case, the unsigned loop iterator in `strlcat` underflowed at the last iteration step due to the postfix decrement. The loop termination, in this case, occurs when the variable equals zero, but after comparison, the value of the variable is still decreasing by one. This results in the unsigned integer underflow (which is not an undefined behavior). However, the unsigned underflow does not lead to an error in the code: after the loop, the variable is not used anywhere. But in this situation, it is not possible to prove the VC demanding the absence of an integer overflow (more generally, over- or underflow). The VC is necessary due to the use of the `defensive` integer model (see section Section 5.2) when the bounded integers are modeled using mathematical unbounded integers assuming the absence of integer overflows. To prove the VCs, Alt-Ergo (0.7.3), Simplify (1.5.4) and Z3 (2.0) solvers were used.

In [5] authors used Frama-C with the WP deductive verification plugin to verify the functions of the klibc library. The authors were able to fully prove 14 string functions. For 12 functions some VCs were not discharged. Four more functions failed to analyze due to errors in the verification tools. In addition to the string functions (from `string.h`), functions from the `stdio.h` were also analyzed. As noted by the authors, almost all functions from this header file use system calls, which in most cases results in a weak specification. To overcome the limitations of the verification tools and to simplify the generated VCs the authors made changes to the source code.

The authors analyzed in advance the problems with type casts modeling (for example, `unsigned char *` to `char *`) and modified the code to exclude such operations. The authors also faced the code pattern with the postfix decrement in a while loop. To prove the VCs, Alt-Ergo (0.95.1), CVC3 (2.4.1) and Z3 (4.3.1) solvers were used.

The most comprehensive document on ACSL specifications development is ACSL by Example [8]. It contains ACSL specifications for functions from the C++ standard library (Standard Template Library). Initial implementation converted from C++ function templates to C functions that work on arrays of type `int`. The authors regularly update the document with new specifications and functions, bug-fixes, etc. This project started in 2009. The document contains a number of fully verified functions. They were proved with Alt-Ergo, CVC3, CVC4, Z3, and EProver solvers. Authors use the WP deductive verification plugin.

GrammaTech report [9] describes typical problems the authors encountered when developing specifications for the GTlibc library. Frama-C with WP was used. Among other points, the authors report memory model problems with pointers type casts and pointers comparison.

3 ACSL

ACSL is designed to be suitable for specifying safety properties of C programs, including contract specifications (pre- and postconditions) and assertions with arbitrary predicates on one or several memory states. The language also supports the specification of function frame conditions, axiomatic theories and additional annotations required by particular verification tools to check the specified properties (examples of additional specifications are loop invariants and pragmas). ACSL includes specification constructs for expressing C-specific attributes related to explicit low-level memory management such as start addresses and lengths of allocated memory blocks, pointers with support for arbitrary pointer type conversions and accessibility predicates for read-only and read-write access.

```
1 /*@ requires valid_strn(s, count);
2     assigns \nothing;
3     behavior exists:
4      assumes ∃ char *p; s≤p<s+strnlen(s,count) ∧ *p ≡ (char %)c;
5      ensures s ≤ \result ≤ s+strnlen(s, count);
6      ensures *\result ≡ (char %) c;
7      ensures ∀ char *p; s ≤ p < \result ⇒ *p ≢ (char %)c;
8     behavior not_exists:
9      assumes ∀ char *p; s ≤ p < s+strnlen(s,count) ⇒ *p ≢ (char %)c;
10     ensures \result ≡ \null;
11    complete behaviors;
12    disjoint behaviors;*/
13 char *strnchr(const char *s, size_t count, int c) {
14  //@ ghost char *os = s;
15  //@ ghost size_t ocount = count;
16  /*@ loop invariant 0 ≤ count ≤ ocount;
17      loop invariant os ≤ s ≤ os+strnlen(os, ocount);
18      loop invariant s-os ≡ ocount-count;
19      loop invariant valid_strn(s, count);
20      loop invariant strnlen(os,ocount) ≡ s-os+strnlen(s, count);
21      loop invariant ∀ char *p; os ≤ p < s ⇒ *p ≢ (char %) c;
22      loop variant count;
23   */
24  for (; count--/*@%*/ && *s != '\0'; ++s)
25   if (*s == (char)/*@%*/c)
26    return (char *)s;
27  return NULL;
28 }
```

Listing 1. from Linux 4.12, `lib/string.c`

Let's consider an example of a simple C function with an appropriate ACSL specification. Listing 1 presents one of the implementations for function **strnchr** from the Linux kernel. The function searches for the first occurrence

of character `c` in a string `s` of length bounded by the parameter `cnt`. The precondition in line `1` requires the string `s` to address a valid memory area of length `min(strlen(s), cnt) + 1`. `strnchr` is a pure function, the absence of effects on memory state is specified in line `2`. The further specification is split into two cases: The first one when the string includes the searched character and the second one when it does not. ACSL includes a special construct for such composite specifications, which is called *behaviors*. In ACSL behaviors are not treated as syntactic sugar (unlike, e. g., JML), but fully integrated into the language such that nearly all specification constructs both in contracts and in function bodies are attributed to one or several behaviors and thus different behaviors of a function are intended to be checked separately. To verify the function `strnchr` against its contract specification with a deductive verification tool, the loop invariant and a ranking function (loop variant) are specified in lines `16`–`22`.

The implementation of `strnchr` contains an intentional type cast `(char)c` in line `25` and a postfix decrement of a loop iterator `count` in line `24`. In both of those cases, the corresponding operation (type cast or decrement) discards some parts of the bitwise representation of the argument (higher bits of the `int` value and the sign bit correspondingly), which corresponds to the intention of the programmer. To distinguish those intentionally overflowing operations, whose semantics is described in terms of bitwise interpretation of bounded integers, we introduced a special annotation construct `/*@%*/`.

4 Region separation in Jessie

Since there are two deductive verification plugins for the Frama-C platform, we had to make a choice between Jessie and WP. While there may be many arguments for choosing a more up-to-date and actively maintained WP plugin, which, among others, has capabilities for bitwise modelling of in-memory data representation and support for interactive proofs, here we emphasize that our initial justification for choosing Jessie over WP was due to its more flexible architecture that enabled easier experimentation with custom ACSL extensions (including the composite integer model described in section Section 5.2) and also its support for region-based modelling of the heap.

In particular, the heap in Jessie is separated into disjoint regions according to the results of a preliminary conservative static analysis presented in [10]. While the separation analysis is coarse (so that its soundness is easy to establish), it is still useful in many cases arising during verification of imperative code. For example, consider the following loop invariant:

```
1 /*@ loop invariant \at(src,Pre) ≤ src ≤
2                      \at(src,Pre)+strlen(\at(src,Pre));
3     loop invariant \at(dest,Pre) ≤ dest ≤
4                      \at(dest,Pre)+strlen(\at(src, Pre));
5     //...
6 */
7 while ((*dest++ = *src++) != '\0')
8   ;
```

Listing 2. from Linux 4.12, `lib/string.c`

Here, in general, proof of the fact that `strlen{LoopCurrent}(\at(src, Pre)) == strlen{Here}(\at(src, Pre))` holds at the end of every iteration requires inductive reasoning since the definition of `strlen` is recursive and the side effect of the assignment `*dest++ = *src++` can generally interfere with the memory footprint of the function `strlen`. But the static separation analysis implemented in Jessie assigns disjoint memory regions to the pointers `dest` and `src` and so both applications of `strlen` to `src` before and after the loop iteration are encoded using a heap variable separate from that of `dst` and therefore literally coincide. So in Jessie the non-interference trivially holds and does not require any additional proof effort. In general, the separation analysis is imprecise and may require explicit weakening, e. g., if the surrounding function can be called in context with more aliasing (e. g., when `src` may intersect with `dest`), but it can still considerably simplify the verification by eliminating the need in inductive framing lemmas.

5 Limitations of the current implementation

5.1 Jessie byte-level block memory model

There are a number of ways to logically represent pointers and memory blocks in the generated VCs. Jessie implements the *byte-level block memory model* [3], where pointers are logically represented as pairs of the form (l, o) and memory blocks are represented as triples of the form (l, a, s). Here l is a label uniquely identifying a memory block, o is the offset of the pointer from the starting address a of the block l, and s is the size of the block. The introduction of unique block labels allows us to ensure that no memory access occurs beyond the bounds of the pointed memory block even if the corresponding memory area is also allocated. Although such access cannot break segmentation checks, it is forbidden by C standard [11] (subsection 6.5.6, paragraph 8 classifies out-of-bounds pointers, except for pointers to the one past the last element of an array, as undefined). As explained in [3] describing the design choices behind the Jessie tool, byte-level block memory model in principle allows us to express common but non-standard C code fragments, such an implementation of the function `memmove`, while retaining the ability to detect use-after-free memory safety errors and potential pointer overflows.

The actual implementation of the memory model in the tool, however, diverges from its simple theoretical description in several ways and imposes a number of additional restrictions on the supported subset of C.

First, pointers are implemented in the corresponding Jessie theory (in WhyML) as values of an abstract type `pointer` with four corresponding abstract operations:

$$\begin{aligned}
&\texttt{sub_pointer} \ : \ \texttt{pointer} \times \texttt{pointer} \to \texttt{int},\\
&\texttt{shift} \ : \ \texttt{pointer} \times \texttt{int} \to \texttt{pointer},\\
&\texttt{same_block} \ : \ \texttt{pointer} \times \texttt{pointer} \to \texttt{bool}, \text{ and}\\
&\texttt{address} \ : \ \texttt{pointer} \to \texttt{int}.
\end{aligned}$$

Block sizes are represented implicitly by so-called *allocation tables*, mutable values of an abstract type with two axiomatically defined functions

$$\texttt{offset\_min} \ : \ \texttt{alloc\_table} \times \texttt{pointer} \to \texttt{int} \text{ and}$$
$$\texttt{offset\_max} \ : \ \texttt{alloc\_table} \times \texttt{pointer} \to \texttt{int}.$$

The functions represent the minimal and maximal allowed offset of a pointer in its corresponding allocated memory block i. e., for a pointer $p = (l, o)$ and its corresponding memory block (l, a, s) which has size s in the state represented by allocation table t, $\texttt{offset\_min}(t, p) = -o$, $\texttt{offset\_max}(t, p) = s - o - 1$. Thus a pointer p can be safely dereferenced iff $0 \le o \le s - 1$ i. e., $\texttt{offset\_min}(t, p) \le 0 \wedge \texttt{offset\_max}(t, p) \ge 0$. We denote this condition as $\texttt{valid}(t, p)$. There is no direct representation for block labels (l) or starting addresses (a) of the memory blocks. The VCs generated for dynamic memory allocations and deallocations (function calls to `kmalloc` and `kfree` are treated specially in Jessie[1]) involve only allocation tables and functions `sub_pointer`, `shift` and `same_block`. This makes the corresponding axiomatization inherently incomplete. In particular, the function `address` is not only left entirely uninterpreted in the current implementation, but cannot be even theoretically given a complete axiomatization. Consider the following property of this function: *two valid pointers from different blocks cannot have the same address*. It cannot be expressed as a logical proposition using the current Jessie theory since this would involve bounded existential quantification over all possible *reachable* states of the corresponding allocation table:

$$\forall p_1, p_2.\ \big(\exists t.\ \mathit{Reachable}(t) \wedge \texttt{valid}(t, p_1) \wedge \texttt{valid}(t, p_2)\big) \wedge \neg\texttt{same\_block}(p_1, p_2) \implies \texttt{address}(p_1) \neq \texttt{address}(p_2).$$

Since the problem of inferring an explicit representation of the predicate $\mathit{Reachable}(t)$ is undecidable, the tool should implement an implicit encoding of the pointer address properties at every allocation point:

$$\forall p.\ \texttt{valid}(t, p) \implies \texttt{address}(p) < \texttt{address}(p^*) \vee \texttt{address}(p) \ge \texttt{address}(p^*) + \mathit{sizeof}(*p^*) \times s,$$

where p^* points to the start of a freshly allocated memory block of size $s \times \mathit{sizeof}(*p^*)$ and t is the state of the allocation table just before the allocation. The unavailability of a precise formalization for the function `address` prevents the generation of the appropriate VCs for potential pointer overflows and a more flexible formalization of pointer comparison and difference operations (allowing the verifation of functions such as `memmove`).

Moreover, the pointer offset and difference, as formalized by the functions `shift` and `sub_pointer`, are measured in units equal to the sizes of the addressed values, according to the pointer indexing semantics of C, rather than in bytes or words. In particular, an expression `p + 1`, where `p` has type `int *`, is translated roughly as $p+1$ rather than as $p+s$, where s is the size of the integer type (usually equal to 4). Such translation immediately prevents many common combinations of pointer casts and arithmetic, including the uses of the `container_of` macro. To

[1] The special treatment is necessary because Jessie does not support arbitrary pointer type casts, in particular, reinterpretation casts such as `char *→int*`, so the return type of memory allocating functions should be specialized at each call site, which can not be directly expressed in ACSL.

see this, it is enough to consider two pointers: `p + 1` and `((char *)p) + 1`, where p has type `int *` and points to the beginning of an allocated memory block. In the byte-level block memory model with size-proportional offsets, these pointers would have the same representation $(l, 1)$, while their actual addresses cannot be equal (they should differ at least by 1, usually by 3). This contradicts the functional consistency of the function `address`. To circumvent this contradiction (and for other reasons, see [10,12]) current Jessie implementation makes use of two separate techniques. First, it introduces *tag tables* tracking the precise dynamic types of the objects in the allocated memory. These tag tables allow us to introduce the necessary checks for pointer shift operations in the generated safety VCs (more on this in [12,13]). Second, it implements a number of *normalizing* code transformations that rewrite nested structures and addressed fields of simple types into pointers to separately allocated structures or values of the corresponding type (the transformations are described in [10]). This allows us to express the addresses of nested objects in the Jessie memory model. However, a combination of these two approaches results in a number of significant restrictions. In particular, unions containing nested structures as their members cannot be soundly represented by the model. This is because that it is impossible to approximate statically whether a pointer to a structure obtained, say, as a function parameter is actually a pointer to a structure nested in some union and so writing to a field of this structure should be translated into a *strong coercion* [12] of the corresponding outer union possibly invalidating other representations of the underlying memory and updating the tag table.

To address these and some other limitations of the current Jessie memory model, a new model was proposed in [4]. This model, though, suggests simple byte-level modeling of pointers. Since we usually assume an arbitrary memory allocation strategy, this should not lead to missed C standard violations due to the dereferencing of valid pointers in different memory blocks in practice. This is because usually in such cases at least one of the possible arbitrary allocation strategies leads to the dereference of an invalid pointer and thus it is impossible to spuriously prove that such a dereference is safe. However, the memory model suggested in [4] is not yet implemented in the tool. So in this study, we used the current implementation of the Jessie memory model as-is.

The only change we made to the tool concerns the translation of pointer inequalities. Since the current implementation does not provide enough support for arbitrary pointer comparisons, we restricted pointer inequalities to support only pointers in the same memory block by generating the corresponding VCs and changing the translation of the corresponding predicates of the form $p_1 \lozenge p_2$ into $\texttt{sub\_pointer}(p_1, p_2) \lozenge 0 \wedge \texttt{same\_block}(p_1, p_2)$ (here $\lozenge \in \{>, <, =, \leq, \geq, \neq\}$). This made many specifications slightly shorter as the pervasive condition $\texttt{same\_block}(p_1, p_2)$ was made implicit.

5.2 Jessie integer models, composite integer model and modulo arithmetic annotations

Jessie originally implements three logical models for machine integer types of different size and signedness. The simplest model called `math` (or *unbounded*) unconditionally encodes values of all integer types as mathematical integers. It does not support overflow checks and does not model the wrap-around behavior of machine integers. It in principle allows the modelling of some bitwise operations on unbounded integers with an appropriate axiomatization, but in practice such modeling is usually very inefficient. Another, most commonly used integer model is called `defensive` (or *bounded*) and differs from the `math` model in two ways:

- for integer operations in code it generates appropriate VCs preventing arithmetic overflows;
- bounded integral types in *logic* (i. e., in specifications) are modeled by abstract types with special injection/projection functions (e. g., `int32_of_integer` / `integer_of_int32`), thus only allowing the injection of values fitting the destination type.

The `defensive` model is simple and efficient and is suitable for most cases except when precise modeling of machine arithmetic or bitwise operations is needed. For these purposes Jessie implements `modulo` integer model, which precisely models values of integral types as bitvectors.

Unfortunately, the integer model in Jessie can only be chosen once for the entire program analyzed using the corresponding pragma. In practice, however, it is desirable to be able to choose the appropriate integer model on a very fine-grained basis, down to every arithmetic operation. Consider the following example:

```
1 int strncasecmp(const char *s1, const char *s2, size_t len) {
2  unsigned char c1, c2;
3  if (!len) return 0;
4  do {
5    c1 = *s1++;
6    c2 = *s2++;
7    if (!c1 || !c2) break;
8    if (c1 == c2) continue;
9    c1 = tolower(c1);
10   c2 = tolower(c2);
11   if (c1 != c2) break;
12 } while (--len);
13 return (int)c1 - (int)c2;
14 }
```

Listing 3. from Linux 4.12, `lib/string.c`

Here in lines `5` and `6` the `modulo` integer model would be suitable as the cast from `char` to `unsigned char` may overflow and this is in line with the intention of the programmer. However we would also like a potential overflow to be detected if we accidentally change the return type of the function to `char`. So the `defensive` integer model is suitable to model the subtraction in line `13`.

To support such fine-grained integer model selection, we implemented an extension to the ACSL specification language with *modulo arithmetic annotations*. The following new modulo arithmetic annotations were introduced:

- for arithmetic operations: `+/*@%*/`, `-/*@%*/`, `*/*@%*/`, ...
- for compound assignments: `+=/*@%*/`, `-=/*@%*/`, `/=/*@%*/`, ...
- for prefix and postfix operators: `++/*@%*/`, `--/*@%*/`
- for explicit casts: `(unsigned char)/*@%*/`, ...
- for modulo arithmetic in logic: `+%`, `-%`, `*%`, ...

The integer model used to model both `defensive` (the default) and `modulo` arithmetic operations is a combined one. In this model, bounded integers are modeled as bitvectors with two injection/projection functions to/from the mathematical unbounded integers.

Let's illustrate the encoding employed by the model on a sample arithmetic operation `+`, a bitwise operation `&`, a relation `<`, and a sample bounded integer type `bint` (with injection function `to_int`). The operation `+` *in logic* is encoded simply as integer addition and has type $\texttt{int} \times \texttt{int} \to \texttt{int}$. The operation `+%` *in logic* is encoded as bitvector addition and has type $\texttt{bint} \times \texttt{bint} \to \texttt{bint}$. It is also augmented with axioms relating the operation to `+`, e. g., $\forall a, b : \texttt{bint}.\ \texttt{in\_bounds}\big(\texttt{to\_int}(a) + \texttt{to\_int}(b)\big) \implies \texttt{to\_int}(a +\% b) = \texttt{to\_int}(a) + \texttt{to\_int}(b)$. The operation `&` *in logic* is encoded as bitwise conjunction with the same type as `+%`. The relation `<` *in logic* is encoded as either bitwise or integer relation depending on the type of arguments. The bitwise relations is augmented with an axiom relating it to the integer one. The operation $a + b$ *in code* is encoded as an abstract operation (`val` in Why3ML) on bitvectors with precondition requiring $\texttt{in\_bounds}(\texttt{to\_int}(a) + \texttt{to\_int}(b))$ end ensuring two postconditions: $\mathit{result} = a +\% b$ and $\texttt{to\_int}(\mathit{result}) = \texttt{to\_int}(a) + \texttt{to\_int}(b)$. The operation a `+/*@%*/` b *in code* is also encoded as an abstract operation with no precondition and two postconditions: $\mathit{result} = a +\% b$ and $\texttt{to\_int}(\mathit{result}) = \texttt{norm}(\texttt{to\_int}(a) + \texttt{to\_int}(b))$, where `norm` stands for an expression for range normalization involving axiomatization of modulo arithmetic. The operation `&` *in code* is an abstract operation with a straightforward postcondition $\mathit{result} = a \,\&\, b$. Finally, the predicate `<` *in code* is an abstract operation with two postconditions $\mathit{result} \iff a < b$ and $\mathit{result} \iff \texttt{to\_int}(a) < \texttt{to\_int}(b)$. Other operations are represented similarly.

This encoding enables construction of more expressive and predictable models while avoiding direct use of any interpretation for function `to_int`, which usually can't be efficiently handled by the solvers. On the other hand, the use of quantified axioms significantly reduces both predictability and performance of the solvers. This can be potentially addressed by either adding some preliminary instantiation step or implementing similar support for the necessary operations as an SMT theory directly in the solver (by converting axioms into inference rules of the theory).

Lastly, let's demonstrate some practical capabilities of this integer model, even in the naive implementation, with an example proof of a bit-twiddling trick for computing average of two unsigned integers:

```
1 //@ ensures \result ≡ (a + b) / 2;
2 unsigned average(unsigned a, unsigned b)
3 {
4   /*@ ghost unsigned long long result1 =
5       (a ^ b) + ((unsigned long long) (a & b) << 1ULL); */
6   /*@ ghost unsigned long long result2 =
7       (unsigned long long) a + b; */
8   //@ assert result1 ≡ result2;
9   return (a & b) + ((a ^ b) >> 1U);
10 }
```

Here the expressions in the ghost code trigger succinct instantiation of necessary lemmas relating bitwise and integer interpretations of bounded integers (through the double post-conditions of the corresponding Why3 operations).

The use of such a combined model and the introduction of new fine-grained modulo arithmetic annotations allowed us to significantly simplify the specification and verification of many functions included in this study.

6 Formal Specifications

We were guided by several techniques in the development of specifications: the use of excessive specifications (explicit specifications and specifications that establish the correspondence with a logical function), the development of specifications based on source code, and the context of function calls.

The results described in [6] show that the development of a function contract, based exclusively on documentation is difficult: almost always, at the proof stage we have to rewrite the specification based on the source code. This approach is also explained by the fact that in this work we develop specifications on the complete code. Linux code is not written in accordance with a certain set of formal specifications. Also, the kernel does not have documentation for a lot of functions. We intentionally did not follow the standard documentation (man pages) for such functions, since their implementation in the kernel can differ from the others (for example, from implementation in the standard library), and the documentation is incomplete and may contain inaccuracies [6].

```
1 /*@ predicate valid_strn(char *s, size_t cnt) =
2       (∃ size_t n; n<cnt ∧ s[n] ≡ '\0' ∧ \valid(s+(0..n))) ∨
3       \valid(s+(0..cnt));
4     requires valid_strn(s, cnt);
5     assigns \nothing;
6     ensures \result ≡ strnlen(s, cnt);
7     behavior null_byte:
8       assumes ∃ ℤ i; 0 ≤ i ≤ cnt ∧ s[i] ≡ '\0';
9       ensures s[\result] ≡ '\0';
10      ensures ∀ ℤ i; 0 ≤ i < \result ⇒ s[i] ≢ '\0';
11    behavior cnt_len:
12      assumes ∀ ℤ i; 0 ≤ i ≤ cnt ⇒ s[i] ≢ '\0';
13      ensures \result ≡ cnt;
14    complete behaviors; disjoint behaviors;*/
15 size_t strnlen(const char *s, size_t cnt);
```

Listing 4. strnlen contract

Following this approach, the specifications for some functions have a slightly more detailed view. For example, for `strn*` functions (see Listings 4 and 5) we do not require the presence of the string's end marker. In the `strnlen`'s precondition (see Listing 4), it is assumed that the string should be valid until the minimum of the string's length (if there is one) and the second argument of the function `strnlen`. The return value is explicitly specified in the postcondition. In the `strncmp` case (see Listing 5), there are also no restrictions on the fact that the input strings must contain a zero byte. This leads to the point where it is necessary to explicitly describe the behavior of the function when the input strings with end markers differ in length. We tried to maximally weaken the preconditions and strengthen the postcondition in order to test the instruments of deductive verification, the expressiveness of the ACSL language, and the capabilities of solvers.

```
1 /*@ requires valid_strn(cs, cnt) ∧ valid_strn(ct, cnt);
2     assigns \nothing;
3     ensures \result ≡ −1 ∨ \result ≡ 0 ∨ \result ≡ 1;
4     behavior equal:
5       assumes cnt ≡ 0 ∨ (cnt > 0  ∧
6         (∀ ℤ i; 0 ≤ i<strnlen(cs,cnt) ⇒ (cs[i] ≡ ct[i])) ∧
7         strnlen(cs, cnt) ≡ strnlen(ct, cnt));
8       ensures \result ≡ 0;
9     behavior len_diff:
10      assumes cnt > 0;
11      assumes ∀ ℤ i; 0 ≤ i<min(strnlen(cs,cnt),strnlen(ct,cnt))
12                    ⇒ cs[i] ≡ ct[i];
13      assumes strnlen(cs, cnt) ≢ strnlen(ct, cnt);
14      ensures strnlen(cs,cnt)<strnlen(ct,cnt) ⇒ \result ≡ −1;
15      ensures strnlen(cs,cnt)>strnlen(ct,cnt) ⇒ \result ≡ 1;
16    behavior not_equal:
17      assumes cnt > 0;
18      assumes ∃ ℤ i; 0 ≤ i<strnlen(cs, cnt) ∧ cs[i] != ct[i];
19      ensures ∃ ℤ i; 0 ≤ i<strnlen(cs, cnt) ∧
20        (∀ ℤ j; 0 ≤ j<i ⇒ cs[j] ≡ ct[j]) ∧
21        cs[i] ≢ ct[i] ∧
22        ((u8 *)cs[i]<(u8 *)ct[i]? \result ≡ −1: \result ≡ 1);
23    complete behaviors; disjoint behaviors;*/
24 int strncmp(const char *cs, const char *ct, size_t cnt);
```

Listing 5. `strncmp` contract

6.1 Logic Functions

The specifications are redundant for some functions. In fact, they describe a function's behavior in two different ways. For example, `strlen` specification consists of the usual functional requirements and the requirement for the correspondence between the returned value and the logical function. This approach is motivated by the fact that the logic function `strlen` is convenient to use in specifications of other functions, e. g., `strcmp` (and a logical function that describes the behavior of the function `strcmp` — when describing the functional requirements for `strcpy`). All the basic properties of logic functions are specified by means of axioms and

lemmas. The lemmas were not proved at the first stage presented in this paper only contradiction checks were performed[2]. However, such specifications do not suit all situations. For example, in the general case, they cannot be translated by E-ACSL [18] as executable specifications. Therefore, for functions with an associated logical function, the "usual" specifications were also developed.

A logical function can be associated with a C function (one-to-one) only if the last one is "pure". A logical function is useful for developing specifications of other C-functions. For example, in postconditions of `memcpy`, you can express the equality of `src` and `dest` by calling the `memcmp` logical function.

7 Open Issues

At the specification level, the authors faced many problems related to significant inaccuracies in the modeling of pointer operations, as well as the insufficient level of ACSL language support by the tools.

Thus, for the `memmove` function, there is the VC, which states that the `dest` and `src` pointers should lie in the same allocated memory block. This is necessary in order for the result of their comparison to be determined by the standard [11]. Recall how the `memmove` function works: it copies a memory area of `n` bytes from the `src` address to `dst`, provided that the two memory regions can either overlap or be disjoint. To implement the latter condition, the function performs an ordinal comparison of the `dest` and `src`. In that case, if `dest` is located before `src` the byte-by-byte copy from the beginning of `src` is performed (thus, if the regions overlap, already copied part will be overwritten); if `dest` is located after `src`, then copying is performed starting at the end of the `src` memory region.

The memory model implemented in AstraVer plugin allows arithmetic operations on pointers (in `memmove` this is a comparison implemented through the difference between pointers) only when the pointers belong to the same allocated memory block. For `memmove`, this is not necessarily the case. If we state in the specification contract that `src` and `dest` may belong to different allocated memory blocks, then it is impossible to prove the VC that states that they should belong to the same memory block. The unproved VC is reflected in the results (Table 1). Although comparison of pointers to different memory blocks is the undefined behavior in ACSI C, the comparison can be made defined by casting the pointers to the corresponding underlying integral type. Yet adding such casts is in odds with the goal of the presented work (verifying the functions without modifications) and also not currently supported by the AstraVer plugin.

The `strcat` function concatenates two strings by appending the `src` to the `dest`. To do this, the end of the `dest` string is determined first. Then the `src` string is copied in the same way as in `strcpy`. In order to prove the VCs that state the safety of memory operations in this function, it was enough to require the validity of the strings `src` and `dest` and sufficient memory behind the end of `dest` to accommodate the contents of `src`. However, proving the functional correctness of

[2] Since then we proved all the lemmas using techniques of auto-active verifivation [14,15], in particular, *lemma functions* [16]. This work is available at [17].

the implementation, it was revealed that it is necessary to formulate an additional requirement stating that the sum of the string's lengths fits the `size_t` type. The function is implemented through the pointers iteration. Therefore, the ability to prove the memory operation safety without the last requirement in the function means that the AstraVer memory model does not take into account the possibility of pointer overflow.

It was required to change the code of two functions to prove their correctness. Despite the fact that we want to minimize code changes, in two cases we cannot fully prove correctness without code modification. The functions `memset` and `strcmp` use the implicit type cast with overflow. `memset` casts `int` to `char`, and `strcmp` casts implicitly `char` to `unsigned char`. To mark these overflows as intentional it was required to make the casts explicit. Our ACSL extension with modulo arithmetic annotations still lacks the corresponding construction (e. g., `/*@(unsigned int %)*/`) for implicit casts.

At the specification level, tools do not support the use of predicates in definitions of logical functions or predicates as first arguments of the ternary operator in lemmas and axioms. Because of this, it is sometimes difficult to give an explicit definition of a logical function, and we have to use an axiomatic (implicit) definition. This drawback prevents the explicit definition of the logical functions for `skip_spaces`, `strcspn`, `strpbrk`, and `strspn`.

Functions from the file `ctype.h` (`isspace`, `isdigit`, `isalnum`, `isgraph`, `islower`, ...) are defined as macros that operate on the array `__ctype` of 256 bytes, which specifies the belonging of each character to a particular class. To simplify the verification task, these macros have been replaced by inline functions: verification tools do not allow the writing of specifications for macros, only for functions. The `__ctype` array was redefined as a string (string initialization is translated into model axioms) because the global array initialization is not translated into the WhyML model. However, it was not possible to prove the correspondence of functions from the `ctype.h` file to their specifications even after the described transformations: solvers cannot cope with the proof when the model has an axiomatic definition of the `__ctype` array 256 characters long.

8 Evaluation of Solvers

AstraVer translates Frama-C's internal representation into the program model in WhyML [19], based on the memory model and semantics of operations with integers. The Why3 tool generates VCs for a WhyML program and converts them into an input for solvers. Why3 supports a number of solvers, such as Alt-Ergo, CVC3, CVC4, Z3, Spass, EProver, Simplify and others. Why3 also supports transformations of VCs, for example, splitting conjunctions into separate conditions.

Alt-Ergo (1.30) and CVC4 (1.4) SMT solvers are able to discharge all VCs generated (except for the one for `memmove`). However, it is interesting to evaluate other solvers on the given benchmark. For that purpose we conduct an experiment using the following system configuration: CPU — AMD FX-8120 (Eight-Core Processor), RAM — 16GB, time limit — 60 seconds, memory limit —

6000MB, OS — GNU/Linux (kernel: 4.12.12 (smp preempt) x86_64), software (from AstraVer repository): Why3 (0.87.3+git), Frama-C (Silicon-20161101), Jessie2 (alpha3).

8.1 VC transformation strategy

To put all solvers in similar conditions all VCs were transformed by Why3 using the following strategy:

1. Split goal by conjuncts (`split_goal_wp`) repeatedly until fixed point.
2. Inline definition of all logical symbols (`inline_all`).
3. Split goal by conjuncts (`split_goal_wp`) repeatedly until fixed point.
4. Skolemize goal (`introduce_premises`).

If there are many predicates with long dependency chains, the `inline_all` transformation makes the work of the solvers more difficult. This is not the case for the given benchmark and experiments have shown the positive impact of this transformation. The addition of `introduce_premises` transformation also comes from preliminary experiments demonstrating that solvers work better with formulas of the form $f(x) \wedge \neg g(x)$ than with ones of the form $\neg \forall x.\ f(x) \implies g(x)$. Otherwise, the strategy tries to split the VC into the smallest possible conjuncts.

During the development of specifications, the strategy is not applied by default. Only some of the transformations are applied if solvers fail to discharge VCs by themselves.

Function	**VC**	**Alt-Ergo 1.30**		**CVC3 2.4.1**		**CVC4 1.4**		**CVC4 1.5**		**Eprover 1.9.1-001**		**Spass 3.9**		**Z3 4.5.0**	
	total	vc	atime	vc	atime	vc	atime	vc	atime	vc	atime	vc	atime	vc	atime
_parse_integ.	282	276	0.10	280	0.83	✓	0.18	✓	0.10	212	0.24	197	1.69	279	0.06
check_bytes8	50	49	0.55	49	0.09	49	0.09	✓	0.11	38	1.76	31	8.38	36	1.52
kstrtobool	1096	✓	0.05	✓	0.08	✓	0.10	✓	0.09	1006	0.13	937	0.38	1065	0.15
memchr	39	✓	6.05	11	0.22	✓	0.37	✓	0.15	31	2.58	11	5.73	29	0.12
memcmp	60	58	0.13	✓	0.15	58	0.10	✓	0.10	49	0.51	36	4.45	55	0.15
memcpy	43	✓	4.18	✓	0.35	✓	0.16	✓	0.14	30	1.05	16	6.85	30	0.06
memmove	**93***(92)	90	3.94	✓	0.88	87	0.16	✓	0.18	63	0.95	43	11.87	68	0.30
memscan	47	46	0.07	✓	0.10	✓	0.09	✓	0.09	41	0.59	34	4.55	42	0.06
memset	27	26	5.02	14	0.19	✓	0.19	✓	0.16	19	3.82	12	11.12	18	0.08
skip_spaces	34	30	0.76	32	1.96	✓	0.51	33	0.14	27	0.70	24	0.34	30	0.09
strcasecmp	58	50	0.43	52	1.65	57	0.79	✓	0.53	43	0.28	35	2.85	49	0.49
strcat	73	68	0.58	66	2.16	✓	1.13	71	0.17	54	2.56	39	0.67	60	0.94
strchr	43	35	4.57	23	0.17	✓	0.23	✓	0.22	31	1.03	24	3.65	32	0.11
strchrnul	46	42	2.07	37	0.26	✓	0.19	✓	0.16	40	1.91	31	2.27	39	0.31
strcmp	60	51	1.76	16	0.60	✓	1.75	59	1.08	47	1.05	36	1.65	47	0.10
strcpy	46	43	1.33	45	0.66	✓	0.48	✓	0.17	33	1.13	26	0.65	39	1.43
strcspn	78	68	0.38	69	0.37	74	2.95	75	1.82	58	1.85	46	1.68	61	0.11
strlcpy	84	82	0.15	82	0.14	✓	1.08	✓	0.24	67	1.20	52	1.74	78	0.42
strlen	26	✓	1.12	24	0.12	✓	0.16	✓	0.23	19	3.36	14	2.96	21	0.08
strnchr	49	38	4.44	19	0.23	46	3.34	✓	0.72	35	2.57	24	1.56	27	0.09
strncmp	102	81	2.57	25	0.25	94	2.39	99	2.32	76	1.06	55	2.56	76	0.57
strnlen	44	39	1.91	42	1.04	39	1.23	✓	1.31	31	2.40	26	5.52	32	0.08
strpbrk	70	57	0.64	58	1.54	62	3.18	67	1.57	48	1.89	39	0.75	53	0.09
strrchr	62	53	4.57	12	0.17	✓	1.09	60	0.85	46	2.33	31	4.67	46	0.11
strsep	62	60	0.25	60	0.09	✓	0.19	✓	0.15	55	0.12	51	1.48	58	0.06
strspn	107	99	0.84	100	0.69	104	1.32	103	0.61	89	1.37	75	1.59	91	0.13
TOTAL	2781	2645	0.90	2454	0.42	2740	0.61	2761	0.37	2288	0.76	1945	1.72	2461	0.22

Table 1. Solvers. Proofs Statistics (times are given in seconds)

Alt-Ergo 1.30		**CVC3 2.4.1**		**CVC4 1.4**		**CVC4 1.5**		**Eprover 1.9.1-001**		**Spass 3.9**		**Z3 4.5.0**	
mtime	uniq	mtime	uniq	mtime	uniq	mtime	uniq	mtime	uniq	mtime	uniq	mtime	uniq
58.75	1	56.68	0	57.97	7	52.27	20	47.80	0	59.74	0	26.74	0

Table 2. Solvers. Max Time and Number of Uniq Proofs

8.2 Statistics

Table 1 presents the results of the evaluation. The first column contains the target function name the second one includes the number of VCs generated (safety and behavioral) after application of the transformation strategy. The rest of the table presents solver statistics: the amount of discharged VCs and the average time for successful runs.

The symbol ✓ marks cases when a solver proved all VCs for the corresponding function. Maximal numbers of discharged VCs are highlighted in green. Minimal VC counts are highlighted by red. The minimal average time is highlighted in cyan, the maximal average time is highlighted in brown.

8.3 Discussion

All VCs except one for `memmove` are successfully discharged by at least one of the solvers. The best result was achieved by Alt-Ergo and CVC4. This is expected as those solvers were most extensively used during the development and testing of the toolset.

CVC4 1.5 discharged the greatest number of VCs, while Z3 required the smallest amount of time. This can be partially explained by the fact that we counted only successful proof attempts. Z3 was able to prove fewer VCs than Alt-Ergo or CVC4.

Table 2 presents maximal solving times for successful proof attempts and counts of unique proofs, i. e., VCs that were only discharged by one of the solvers.

9 Conclusion

This paper presents results from the development and evaluation of a deductive verification benchmark consisting of 26 unmodified Linux kernel library functions implementing conventional memory and string operations. Formal contracts of the functions were extracted from their source code and were represented in the form of preconditions and postconditions. The benchmark detected a number of problems with existing deductive verification toolchains. Some of the issues required only fixes in the tools, some of them led to the design of proposals to extend ACSL language, others were left open.

For example, two newly proposed ACSL constructs allowed us to successfully proof 11 more functions without modification of their source code. With these extensions, the authors have successfully and fully proved the correctness of 23 functions. Another 2 functions were proved after a minor modification of their source code, while the final one cannot be completely proved using existing memory model. Specifications of the benchmark contain ≈ 2.6 times as many lines as the source code of the library functions.

The source code of the benchmark and proof protocols are publicly available together with instructions describing how to reproduce the results [7]. The benchmark can be used for the testing and evaluation of deductive verification tools and the starting point for verifying other parts of Linux kernel. A possible next step is to extend the benchmark with other library functions (e. g., bitwise operations).

References

1. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: Acsl: Ansi/iso c specification language. Tech. Rep. 1.12, CEA LIST and INRIA (March 2017)
2. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: A software analysis perspective. Formal Aspects of Computing 27(3), 573–609 (May 2015), https://doi.org/10.1007/s00165-014-0326-7
3. Moy, Y.: Automatic Modular Static Safety Checking for C Programs. Ph.D. thesis, Université Paris-Sud (January 2009), http://www.lri.fr/~marche/moy09phd.pdf
4. Mandrykin, M.U., Khoroshilov, A.V.: Region analysis for deductive verification of c programs. Programming and Computer Software 42(5), 257–278 (2016), http://dx.doi.org/10.1134/S0361768816050042
5. Carvalho, N., Silva Sousa, C., Pinto, J.S., Tomb, A.: Formal verification of klibc with the wp frama-c plug-in. In: Proceedings of the 6th International Symposium on NASA Formal Methods - Volume 8430. pp. 343–358. Springer-Verlag New York, Inc., New York, NY, USA (2014), http://dx.doi.org/10.1007/978-3-319-06200-6_29
6. Torlakcik, M.: Contracts in OpenBSD. Msc. dissertation report, University College Dublin (2010)
7. Verker: Verification of linux kernel library functions (2017), https://forge.ispras.ru/projects/verker
8. Burghardt, J., Clausecker, R., Gerlach, J., Pohl, H.: ACSL by example. Tech. rep., Fraunhofer Institute for Open Communication Systems (2017)
9. Cok, D.R., Blissard, I., Robbins, J.: C library annotations in acsl for frama-c: experience report. Tech. rep., GrammaTech, Inc. (March 2017)
10. Hubert, T., Marché, C.: Separation analysis for deductive verification. In: Heap Analysis and Verification (HAV'07). pp. 81–93. Braga, Portugal (March 2007), http://www.lri.fr/~marche/hubert07hav.pdf
11. ISO/IEC 9899: 2011: C11 standard for C programming language. Standard, JTC and ISO (2011), http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf
12. Moy, Y.: Union and cast in deductive verification. In: Proceedings of the C/C++ Verification Workshop. vol. Technical Report ICIS-R07015, pp. 1–16. Radboud University Nijmegen (July 2007), http://www.lri.fr/~moy/Publis/moy07ccpp.pdf
13. Mandrykin, M.U., Khoroshilov, A.V.: High-level memory model with low-level pointer cast support for jessie intermediate language. Programming and Computer Software 41(4), 197–207 (2015), http://dx.doi.org/10.1134/S0361768815040040
14. Leino, K.R.M., Moskal, M.: Usable auto-active verification (2010)
15. Dross, C., Moy, Y.: Auto-active proof of red-black trees in spark. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NASA Formal Methods. pp. 68–83. Springer International Publishing, Cham (2017)
16. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the verifast program verifier. In: Proceedings of the 8th Asian Conference on Programming Languages and

Systems. pp. 304–311. APLAS'10, Springer-Verlag, Berlin, Heidelberg (2010), http://dl.acm.org/citation.cfm?id=1947873.1947902
17. Verker: Verification of linux kernel library functions, lemma functions branch (2017), https://forge.ispras.ru/projects/verker/repository?rev=lemma_functions
18. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of c programs. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing. pp. 1230–1235. SAC '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2480362.2480593
19. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Proceedings of the 22nd European Symposium on Programming. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (Mar 2013)