LinuxTesting.org

Specifying and proving correctness of Linux kernel components with ACSL

Alexey Khoroshilov Mikhail Mandrykin

ISP RAS

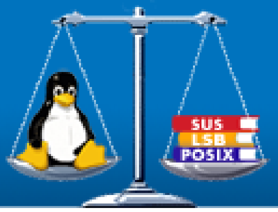
Institute for System Programming of the Russian Academy of Sciences

Linux Verification Center

LinuxTesting.org

Institute for System Programming of Russian Academy of Sciences

VERIFICATION CENTER OF THE OPERATING SYSTEM Linux



founded in 2005

- OLVER Program
- Linux Standard Base Infrastructure Program
- Linux Driver Verification Program
- Linux File System Verification Program
- Linux Deductive Verification

ISP RAS

Historical Perspective

- Teaching Floyd-Hoare methods (2007-...)
  - used Caduceus for course practice
  - then Frama-C + Jessie
- 100 hours student practice (2008-2013)

  Typical verification target: sort or arrays
  - Part 1: Model-based Testing
    - 100% line coverage required
  - Part 2: Deductive Verification
    - Caduceus/Frama-C+Jessie
    - PVS interactive theorem prover if required

// returns a number of nonempty elements of range array

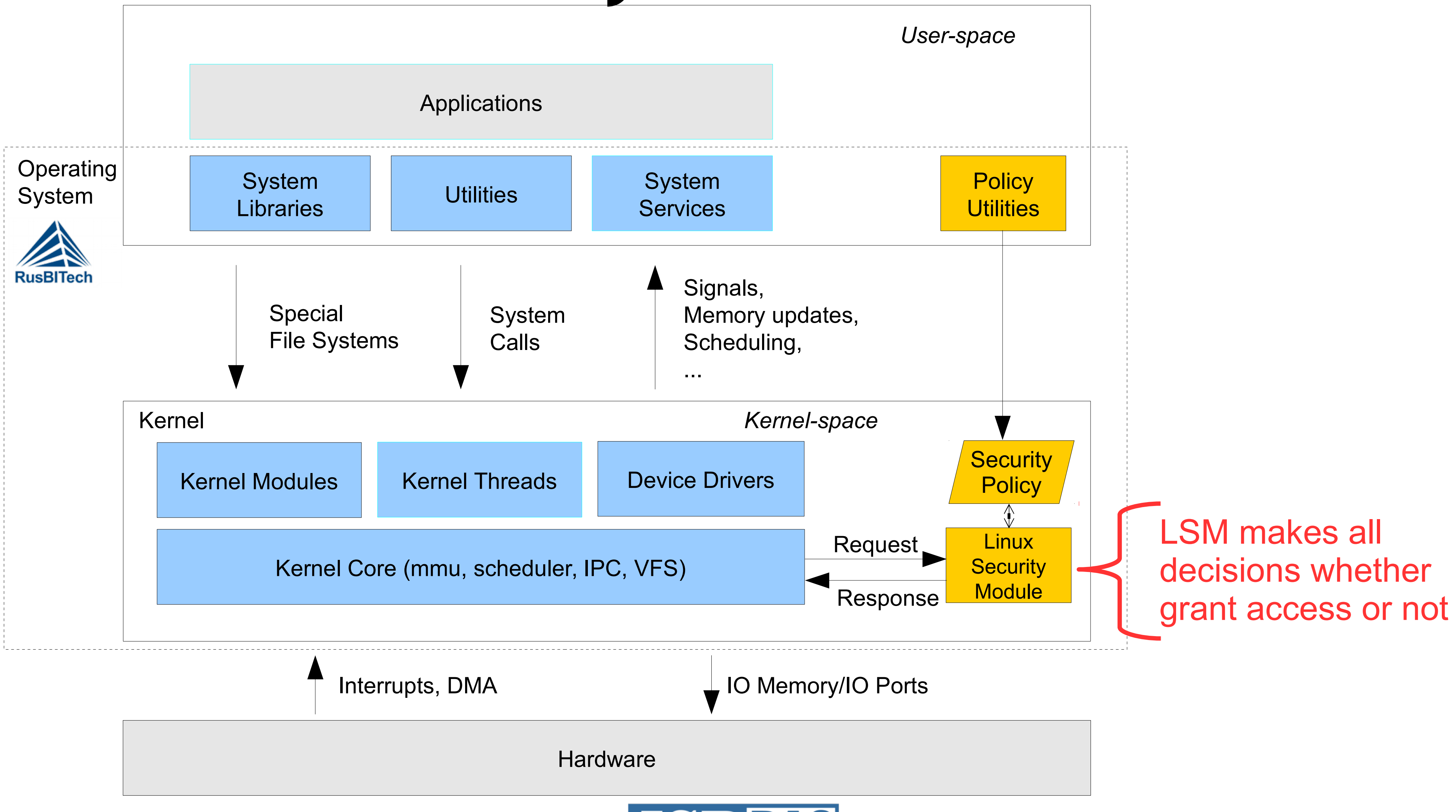
```
int clean_sort_range(struct range *range, int az)
{
        int i, j, k = az - 1, nr_range = 0;

        for (i = 0; i < k; i++) {
                if (range[i].end)
                        continue;
                for (j = k; j > i; j--) {
                        if (range[j].end) {
                                k = j;
                                break;
                        }
                }
                if (j == i)
                        break;
                range[i].start = range[k].start;
                range[i].end   = range[k].end;
                range[k].start = 0;
                range[k].end   = 0;
                k--;
        }
        /* count it */
        for (i = 0; i < az; i++) {
                if (!range[i].end) {
                        nr_range = i;
                        break;
                }
        }

        /* sort them */
        sort(range, nr_range, sizeof(struct range), cmp_range, NULL);

        return nr_range;
}
```

// number of nonempty elements is evaluated by
// finding the first empty element of the array

Historical Perspective (2)

```
commit 834b40380e93e36f1c9b48ec1d280cebe3d7bd8c
Author: Alexey Khoroshilov <khoroshilov@ispras.ru>
Date:   Thu Nov 11 14:05:14 2010 -0800

    kernel/range.c: fix clean_sort_range() for the case of full array

    clean_sort_range() should return a number of nonempty elements of range
    array, but if the array is full clean_sort_range() returns 0.

    The problem is that the number of nonempty elements is evaluated by
    finding the first empty element of the array.  If there is no such element
    it returns an initial value of local variable nr_range that is zero.

    The fix is trivial: it changes initial value of nr_range to size of the
    array.

    The bug can lead to loss of information regarding all ranges, since
    typically returned value of clean_sort_range() is considered as an actual
    number of ranges in the array after a series of add/subtract operations.

    Signed-off-by: Alexey Khoroshilov <khoroshilov@ispras.ru>
    Signed-off-by: Andrew Morton <akpm@linux-foundation.org>
    Signed-off-by: Linus Torvalds <torvalds@linux-foundation.org>
```

Thorough Verification of Linux Security Module

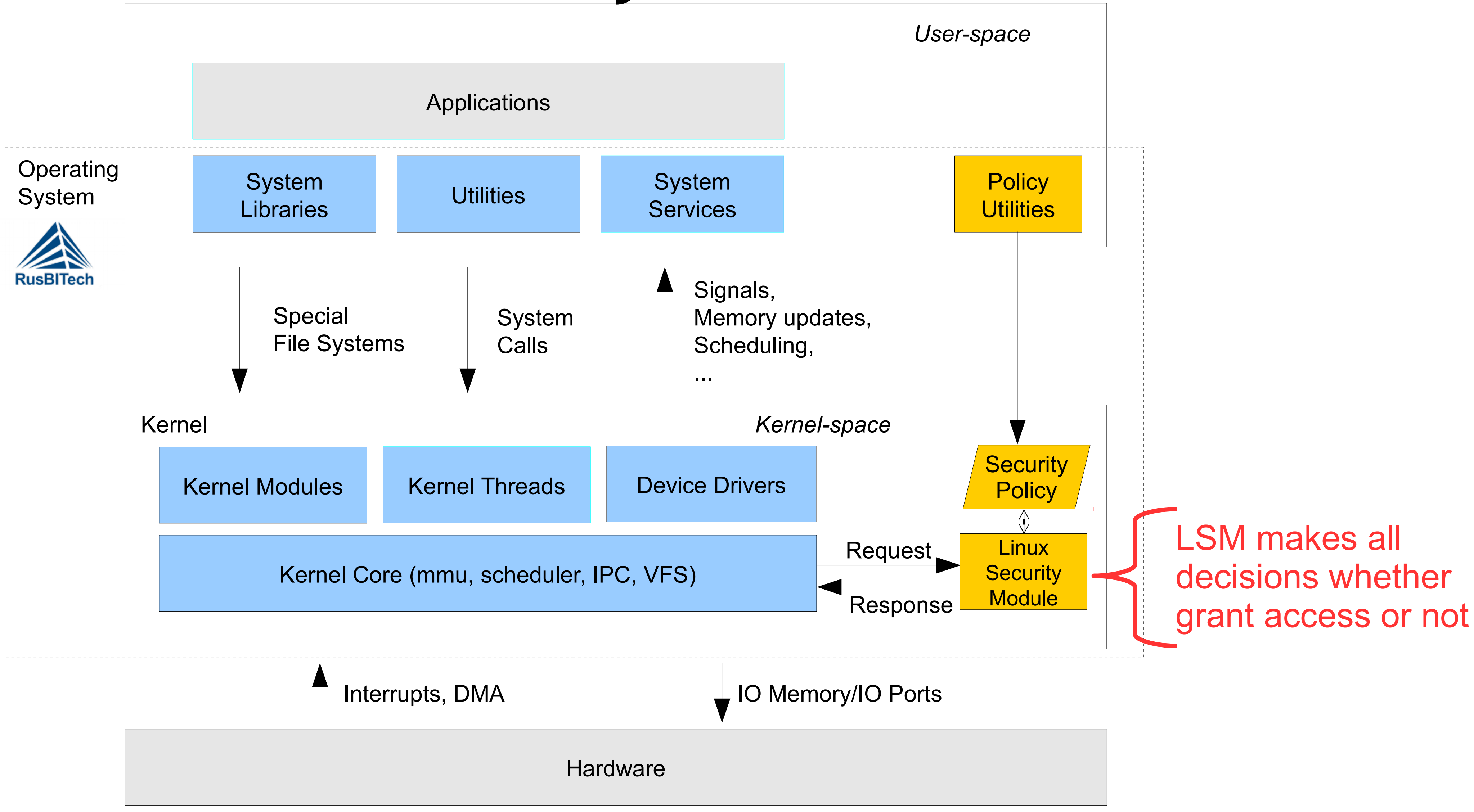


Thorough Verification of Linux Security Module

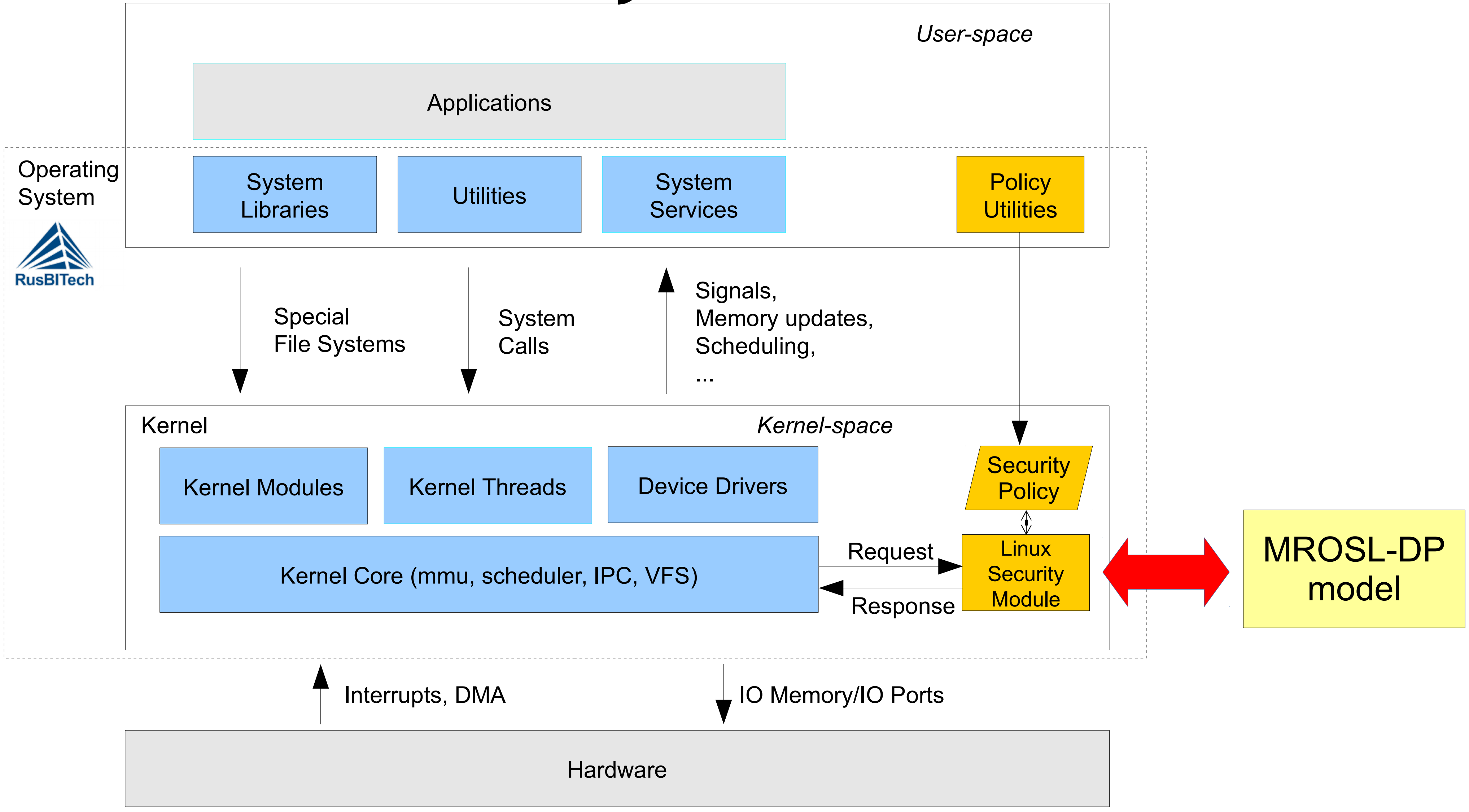
- Custom security policy model (MROSL-DP)
  - Lattice-based mandatory access control (MAC)
  - Mandatory integrity control (MIC)
  - Role-based access control (RBAC)
- Custom Linux Security Module (LSM) implementation
  - Astra Linux Special Edition

RusBITech

Thorough Verification of Linux Security Module



Thorough Verification of Linux Security Module



Project Settings

- Target code: Custom Linux Security Module
  - Hardware independent
  - Sometimes verification unfriendly
  - Out of our control
- Properties to prove:
  - Absence of run-time errors
  - Compliance to MROSL-DP functional specifications
- Assumptions
  - Linux kernel core conforms with its specifications
    - It is not target to prove
  - No concurrent access to data

Tools?

- Frama-C with Jessie plugin
    - open source
    - WP was not mature at that time

But …

- Low level memory operations
    - Arithmetics with pointers to fields of structures (container_of)
    - Prefix structure casts
    - Reinterpret casts
- Integer overflows and bit operations
- Complex functionality requires manual proof
- …

Specifying and Proving Correctness of Linux Kernel Components with ACSL

Operations on bounded integers

Mikhail Mandrykin

ISP RAS

May 30th, 2017

Real world examples: integer models

Linux kernel (`lib/string.c`, `memcmp`)

```
int memcmp(const void *cs, const void *ct, size_t count)
{
        const unsigned char *su1, *su2;
        int res = 0;
        for (su1 = cs, su2 = ct; 0 < count; ++su1, ++su2, count--)
                if ((res = *su1 - *su2) != 0)
                        break;
        return res;
}
```

Overflow checks for arithmetic operations

- 🟩 — integer arithmetic operations
- **No overflow**, both checks pass
- **No implicit truncation**
- Use general "*defensive*" integer model

Real world examples: integer models

Linux kernel (lib/string.c, memset)

```
void *memset(void *s, int c, size_t count)
{
        char *xs = s;
        while (count--)
                *xs++ = c;
        return s;
}
```

Overflow checks for arithmetic operations

- **Intentional overflow** in decrement (--), wrap-around semantics
- **Intentional implicit truncation** of int to char in assignment (=)
- Use special "*modulo*" integer model

Real world examples: integer models

Linux kernel (`lib/string.c`, `strncasecmp`)

```
int strncasecmp(const char *s1, const char *s2, size_t len)
{
        unsigned char c1, c2;
        if (!len) return 0;
        do {
                c1 = *s1++;
                c2 = *s2++;
                if (!c1  !c2) break; if (c1 == c2) continue;
                c1 = tolower(c1); c2 = tolower(c2);
                if (c1 != c2) break;
        } while (--len);
        return (int)c1 - (int)c2;
}
```

Overflow checks for arithmetic operations

- **Intentional implicit conversion** of `char` to `unsigned char` in assignment (`=`)
- If we accidentally put `char` as the return type (`int`), we can get **erroneous implicit truncation** of the result
- We want the overflow to be reported on the *implicit truncation* (`char`)
- *Which* integer model should we use?

Our solution: Composite integer model

Defensive by default, new annotations

- Arithmetic operations

  `+/*@%*/`

- Compound assignment operators

  `+=/*@%*/`

- Postfix operators

  `++/*@%*/`

- Explicit casts

  `(unsigned char)/*@%*/`

- Implicit casts (also checks the specified target type)

  `/*@(unsigned char %)*/`

Comparison operations in code have double semantics

`x < y` $\rightarrow$ `(integer)x < (integer)y && x < y`

Our solution: Composite integer model

Linux kernel (lib/string.c, memset)

```
void *memset(void *s, int c, size_t count)
{
        char *xs = s;
        while (count-- /*@%*/)
                *xs++ = /*@(char %)*/c;
        return s;
}
```

Explicitly mark *intentional overflow* and *intentional implicit truncation*

Our solution: Composite integer model

Linux kernel (lib/string.c, strncasecmp)

```
int strncasecmp(const char *s1, const char *s2, size_t len)
{
        unsigned char c1, c2;
        if (!len) return 0;
        do {
                c1 = /*@(unsigned char %)*/*s1++;
                c2 = /*@(unsigned char %)*/*s2++;
                if (!c1  !c2) break; if (c1 == c2) continue;
                c1 = tolower(c1); c2 = tolower(c2);
                if (c1 != c2) break;
        } while (--len);
        return (int)c1 - (int)c2;
}
```

Explicitly mark *intentional implicit truncation*, but *not* the subtraction (-)

Composite integer model

Combined reasoning for linear operations

Artificial example 1

```
unsigned x, y;
/*@ requires x + y <= 4 && x - y >= 2;
  @ ensures x - 1 <= 0;
  @*/
void f(void)
{
  x += y;
  y = x - y;
  x -= y;
}
```

Composite integer model

Combined reasoning for linear operations

Artificial example 1

```
unsigned x, y;
/*@ requires x + y <= 4 && x - y >= 2;
  @ ensures x - 1 <= 0;
  @*/
void f(void)
{
  x +=/*@%*/ y;
  y = x -/*@%*/ y;
  x -=/*@%*/ y;
}
```

- Using *linear reasoning* we can infer `\at(y, Pre) - 1 <= 0`
- Using *bitwise reasoning* we can infer `\at(x, Post) == \at(y, Pre)`
- Combined model allows to prove the post-condition **automatically**

Composite integer model

What about logic?

Artificial example 2

```
unsigned x, y;
/*@ requires x + y <= 4 && x - y >= 2;
  @ ensures (x & 2) == 0;
  @*/
void f(void)
{
  x +=/*@%*/ y;
  y = x -/*@%*/ y;
  x -=/*@%*/ y;
}
```

Can't prove the post-condition automatically, need auxiliary *assertions*, so need bitwise semantics operations *in logic*

Composite integer model

Typing annotations

- Bitvector operations in *logic*

```
&, |, ^, +%, -%, *%, (int %), ...
```

- New typing rules to support bit-vector operations in *logic*
  - Backward-compatible **implicit casts** to `integer` ($\mathbb{Z}$) for *arithmetic operations*

    ```
    +, -, *, /, %
    ```

  - Precisely matching integer types for *bit-vector operations*

    ```
    +%, -%, *%, /%, %%, (int %)
    &, |, ~, ^
    ```

  - Integer constants are implicitly casted to bitvectors, if in range

    ```
    x +% 1
    ```

  - Semantics of comparisons is ad-hoc, i.e. depends on the types of operands

Composite integer model

Auto-instantiated axiomatic definitions

Artificial example 2

```
unsigned x, y;
/*@ requires x + y <= 4 && x - y >= 2;
  @ ensures (x & 2) == 0;
  @*/
void f(void)
{
  x += /*@%*/ y;
  y = x -/*@%*/ y;
  x -= /*@%*/ y;
  /*@ assert x <= (integer)1; */
  /*@ assert x <= 1; */
}
```

What about auto-instantiating axiom

`\forall unsigned x; x <= 1 <==> x <= (integer) 1;`

for every occurrence of `<=`?

Composite integer model

Artificial example 2

Solution using ghost code

```
unsigned x, y;
/*@ requires x + y <= 4 && x - y >= 2;
  @ ensures (x & 2) == 0;
  @*/
void f(void)
{
  x += /*@%*/ y;
  y = x -/*@%*/ y;
  x -= /*@%*/ y;
  //@ ghost int b = x <= 1;
  /*@ assert b; */
}
```

Composite integer model

A bit-twiddling hack

```
unsigned average(unsigned a, unsigned b)
{
  return (a & b) + ((a ^ b) >> 1U);
}
```

Composite integer model

A bit-twiddling hack: proof

```
/*@ ghost
  @ //@ ensures a + b == (a ^ b) + (a & b) * 2;
  @ void Sum_as_xor_plus_and(unsigned a, unsigned b) {
  @   int aux = (a ^ b) + ((unsigned long long)(a & b) << 1ULL) ==
  @             (unsigned long long)a + b;
  @ }
  @*/

/*@ ensures \result == (a + b) / 2;
  @*/
unsigned average(unsigned a, unsigned b)
{
  //@ ghost Sum_as_xor_plus_and(a, b);
  return (a & b) + ((a ^ b) >> 1U);
}
```

Proof management

Lemma functions

```
/*@ ghost
  @ //@ ensures a + b == (a ^ b) + (a & b) * 2;
  @ void Sum_as_xor_plus_and(unsigned a, unsigned b) { ... }
  @*/
```

$\Updownarrow$

```
//@ lemma Sum_as_xor_plus_and: \forall unsigned a, b; a + b == (a ^ b) + (a & b) * 2;
```

Parameterized axiomatic inclusion

```
/*@ axiomatic Matrix {
  @   type t; type index; type element;
  @   logic t const(element e);
  @   logice get(t map, index i1, index i2); ...
  @ } */
```

```
/*@ axiomatic List {
  @   type t; type element;
  @   logic t empty;
  @   logic t add(t list, element e); ...
  @  } */
```

$\Uparrow$

```
/*@ axiomatic Nqueens {
  @   type solution;
  @   include Map {
  @     type t = solution;
  @     type index = integer;
  @     type element = boolean;
  @   }
  @   include List {
  @     type t = solutions;
  @     type element = solution;
  @   }
  @   ...
  @  }
  @ */
```

- Useful for *complex* specifications

Proofs as C code

Advantages

- Shallower learning curve

    no need to study complex reasoning tools e.g. Coq)

- Maintainability

    code and proofs in the same codebase

- Simpler proofs

    reuse of the translation engine (integer and memory models), more capable decision procedures

Disadvantages

- Bloating specification language (ACSL) with many features would make it more complicated
- Duplicated functionality

  Coq may be still needed for some complex proofs, e.g. existing
- Only suitable for deductive verification plugins
- Poorer reproducibility of proofs (on different ATP versions, different machines)

Proofs as C code

What's needed

- ☑ Annotations in ghost code (`/@ ... @/`)
- ☐ Allow logic types in ghost code
- ☐ Lemma functions with multiple labels
- ☐ Parameterized axiomatic inclusion
- ☐ Auto-instantiation of statements (axioms, lemmas) triggered by logic symbol applications

Proofs as C code

Results so far

- ☑ Tricky average computation

  ```
  (a & b) + ((a ^ b) >> 1U)
  ```

- ☑ Kernighan's bitcount
- ☑ Gray codes (SWAR decoding and properties)
- ☐ n queens
  (from the paper *Verifying Two Lines of C with* WHY3)

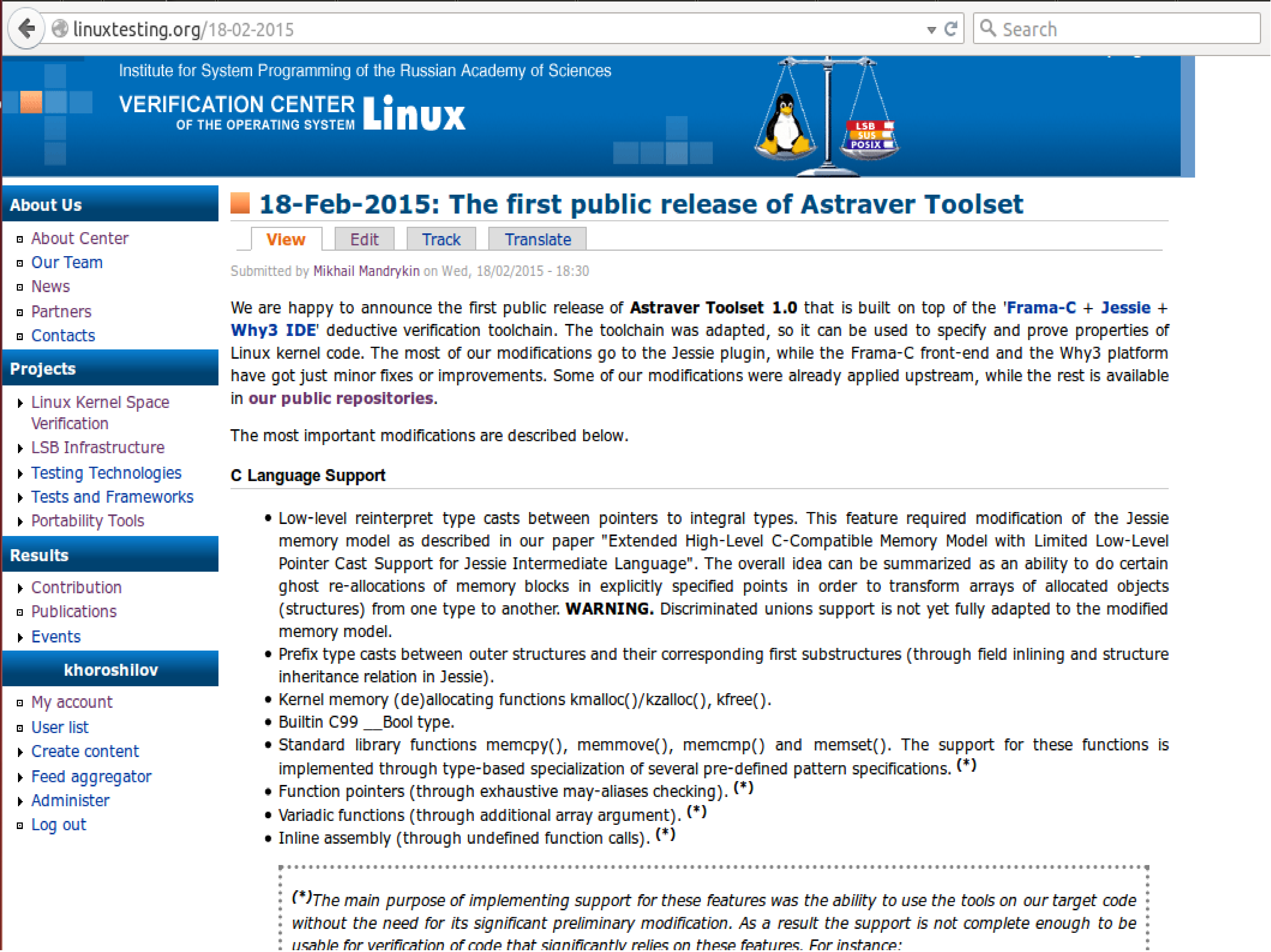
Summary

- Linux kernel code makes extensive use of **bitwise** and **wrap-around** operations
- Support for **composite integer model** combined with **ghost functions** allows to efficiently reason about non-trivial code fragments involving bitwise operations without the use of external interactive tools (e.g. Coq or Isabelle)
- Further ACSL extensions are needed for more complex specifications (e.g. n queens)

AstraVer Toolset

- Frama-C
- AstraVer plugin (fork of Jessie)
- Why3

- Open source:
  - http://linuxtesting.org/astraver
  - http://forge.ispras.ru/projects/astraver/wiki


linuxtesting.org/18-02-2015
Search
Institute for System Programming of the Russian Academy of Sciences
VERIFICATION CENTER OF THE OPERATING SYSTEM Linux
LSB
SUS
POSIX

18-Feb-2015: The first public release of Astraver Toolset

Submitted by Mikhail Mandrykin on Wed, 18/02/2015 - 18:30

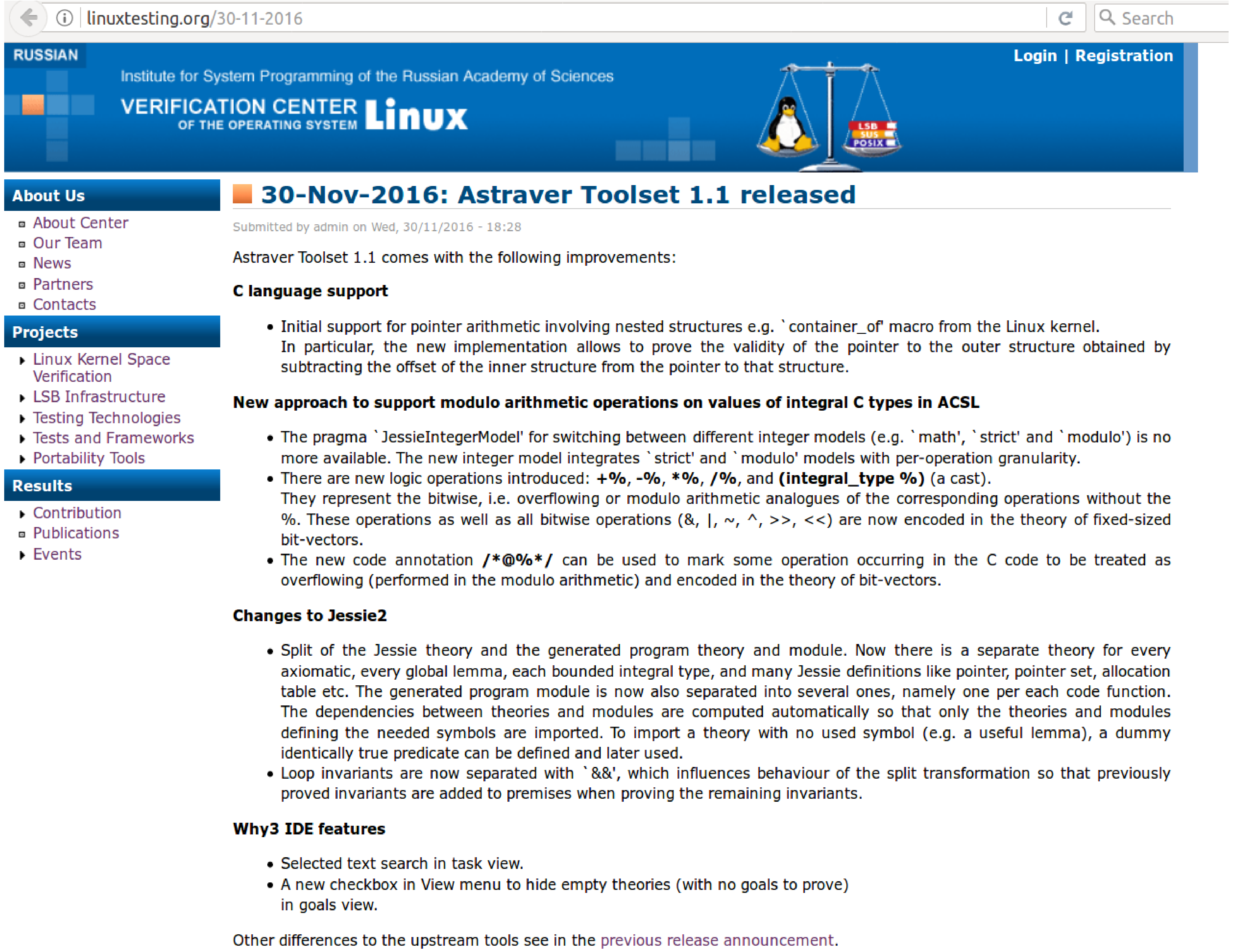
We are happy to announce the first public release of **Astraver Toolset 1.0** that is built on top of the '**Frama-C** + **Jessie** + **Why3 IDE**' deductive verification toolchain. The toolchain was adapted, so it can be used to specify and prove properties of Linux kernel code. The most of our modifications go to the Jessie plugin, while the Frama-C front-end and the Why3 platform have got just minor fixes or improvements. Some of our modifications were already applied upstream, while the rest is available in **our public repositories**.

The most important modifications are described below.

C Language Support

- Low-level reinterpret type casts between pointers to integral types. This feature required modification of the Jessie memory model as described in our paper "Extended High-Level C-Compatible Memory Model with Limited Low-Level Pointer Cast Support for Jessie Intermediate Language". The overall idea can be summarized as an ability to do certain ghost re-allocations of memory blocks in explicitly specified points in order to transform arrays of allocated objects (structures) from one type to another. **WARNING.** Discriminated unions support is not yet fully adapted to the modified memory model.
- Prefix type casts between outer structures and their corresponding first substructures (through field inlining and structure inheritance relation in Jessie).
- Kernel memory (de)allocating functions kmalloc()/kzalloc(), kfree().
- Builtin C99 __Bool type.
- Standard library functions memcpy(), memmove(), memcmp() and memset(). The support for these functions is implemented through type-based specialization of several pre-defined pattern specifications. **(*)**
- Function pointers (through exhaustive may-aliases checking). **(*)**
- Variadic functions (through additional array argument). **(*)**
- Inline assembly (through undefined function calls). **(*)**

(*)The main purpose of implementing support for these features was the ability to use the tools on our target code without the need for its significant preliminary modification. As a result the support is not complete enough to be usable for verification of code that significantly relies on these features. For instance:



linuxtesting.org/30-11-2016

RUSSIAN

Login | Registration

Institute for System Programming of the Russian Academy of Sciences

VERIFICATION CENTER OF THE OPERATING SYSTEM Linux

About Us

- About Center
- Our Team
- News
- Partners
- Contacts

Projects

- Linux Kernel Space Verification
- LSB Infrastructure
- Testing Technologies
- Tests and Frameworks
- Portability Tools

Results

- Contribution
- Publications
- Events

30-Nov-2016: Astraver Toolset 1.1 released

Submitted by admin on Wed, 30/11/2016 - 18:28

Astraver Toolset 1.1 comes with the following improvements:

C language support

- Initial support for pointer arithmetic involving nested structures e.g. `container_of' macro from the Linux kernel.
  In particular, the new implementation allows to prove the validity of the pointer to the outer structure obtained by subtracting the offset of the inner structure from the pointer to that structure.

New approach to support modulo arithmetic operations on values of integral C types in ACSL

- The pragma `JessieIntegerModel' for switching between different integer models (e.g. `math', `strict' and `modulo') is no more available. The new integer model integrates `strict' and `modulo' models with per-operation granularity.
- There are new logic operations introduced: **+%**, **-%**, ***%**, **/%**, and **(integral_type %)** (a cast).
  They represent the bitwise, i.e. overflowing or modulo arithmetic analogues of the corresponding operations without the %. These operations as well as all bitwise operations (&, |, ~, ^, >>, <<) are now encoded in the theory of fixed-sized bit-vectors.
- The new code annotation **/*@%*/** can be used to mark some operation occurring in the C code to be treated as overflowing (performed in the modulo arithmetic) and encoded in the theory of bit-vectors.

Changes to Jessie2

- Split of the Jessie theory and the generated program theory and module. Now there is a separate theory for every axiomatic, every global lemma, each bounded integral type, and many Jessie definitions like pointer, pointer set, allocation table etc. The generated program module is now also separated into several ones, namely one per each code function. The dependencies between theories and modules are computed automatically so that only the theories and modules defining the needed symbols are imported. To import a theory with no used symbol (e.g. a useful lemma), a dummy identically true predicate can be defined and later used.
- Loop invariants are now separated with `&&', which influences behaviour of the split transformation so that previously proved invariants are added to premises when proving the remaining invariants.

Why3 IDE features

- Selected text search in task view.
- A new checkbox in View menu to hide empty theories (with no goals to prove)
  in goals view.

Other differences to the upstream tools see in the previous release announcement.

Conclusions

- Linux kernel code can be deductively verified with Frama-C/AstraVer
  - under some assumptions
  - tool improvements required

LinuxTesting.org

Thank you!

- http://linuxtesting.org/astraver

ISP RAS

Institute for System Programming of the Russian Academy of Sciences