Linux Drivers Verification – Achievements and Prospects

Vadim Mutilin, IARCS 2026

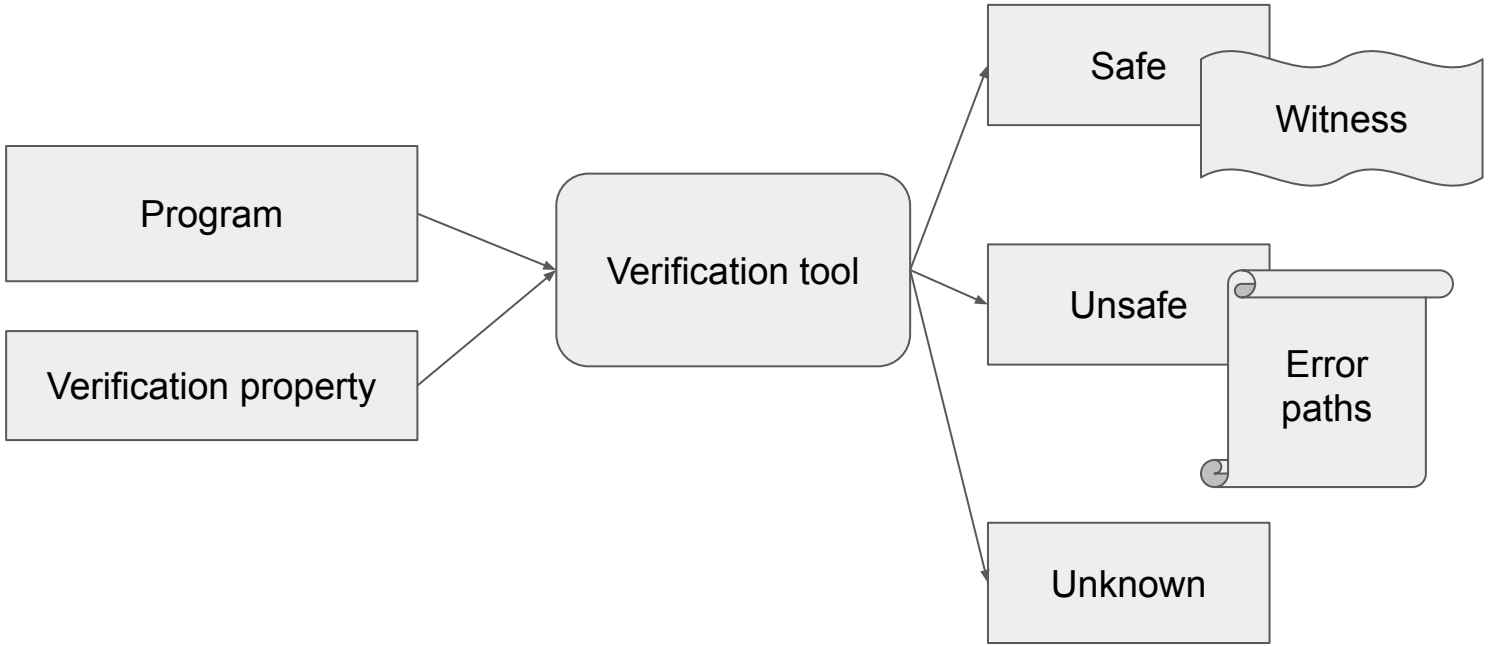
The Team

Pavel Andrianov, Vladimir Gratinskiy, Alexey Khoroshilov, Mikhail Mandrykin, Vitaliy Mordan, Vadim Mutilin, Anton Vasilyev, Oleg Petrov, Veronika Rudenchik, Vladislav Glazkov

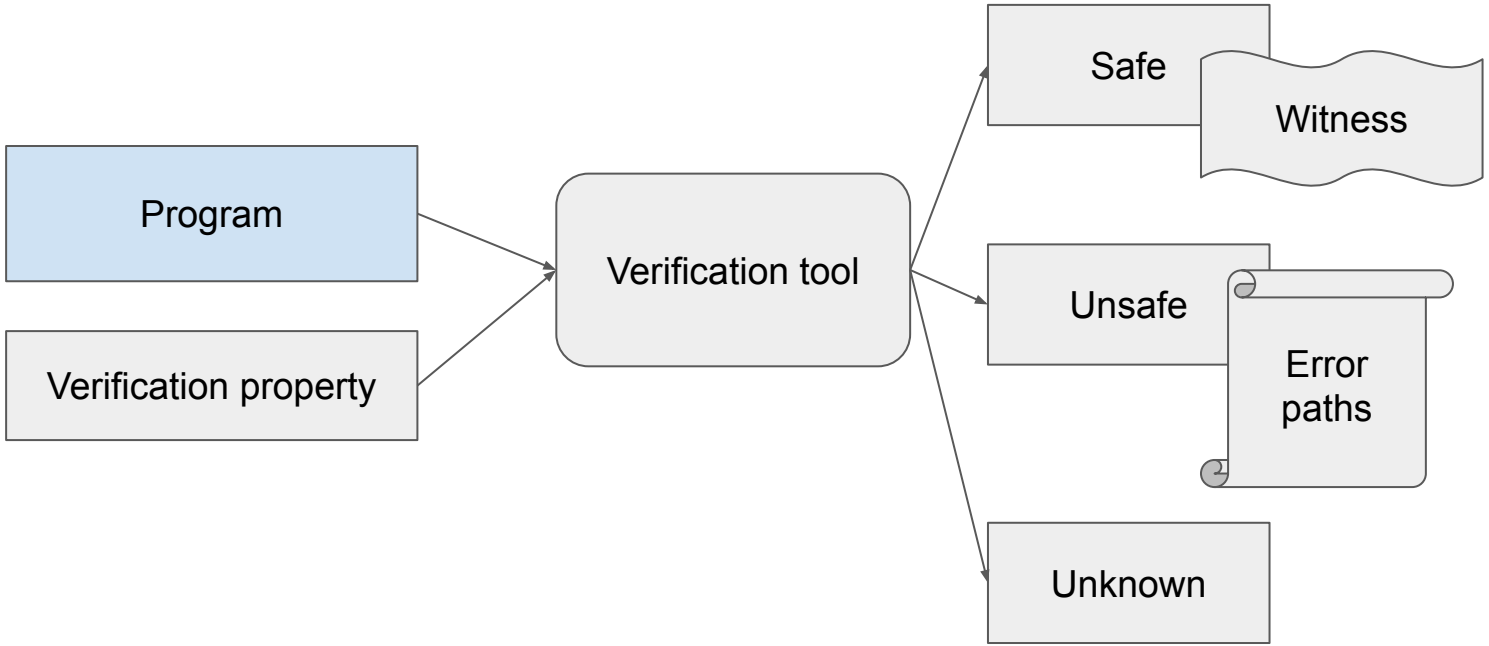
Former colleagues: Eugene Novikov, Ilya Shchepetkov, Ilya Zakharov, Oleg Strikov, Alexander Strakh, Pavel Shved, Marina Makienko, Anton Volkov

Colleagues from Passau, Munich, Paderborn…

Automatic Program Verification



Automatic Program Verification



Kernel source size

The Linux kernel keeps growing in size over time as more hardware is supported and new features are added.

For the following numbers, we have counted everything in the released Linux source package as "source code" even though a small percentage of the total is the scripts used to configure and build the kernel, as well as a fair amount of documentation. Those files, too, are part of the larger work, and thus merit being counted.

Version	Files	Lines
4.8	55,472	22,070,760
4.9	56,201	22,348,062
4.10	57,167	22,839,361
4.11	57,959	23,137,101
4.12	59,801	24,170,555
4.13	60,538	24,766,703

The information in the table above shows the number of files and lines in each kernel version.

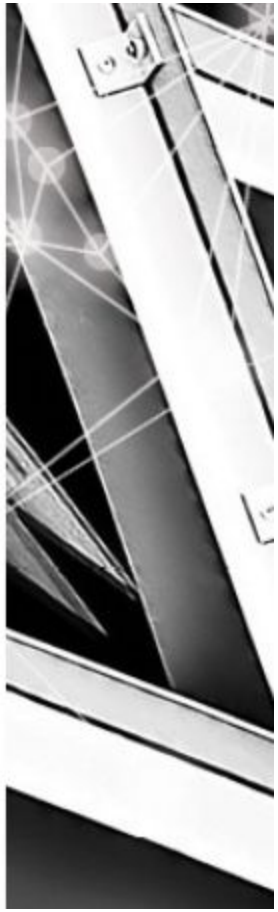
Linux Kernel Development Report, The Linux Foundation, 2017

New drivers

Naturally, the kernel developers also added hundreds of new drivers and thousands of fixes over the past year.

Hardening

Work on hardening the kernel against attack continues with the addition of several new technologies, many of which have their origin in the grsecurity and PaX patch sets. New hardening features include virtually mapped kernel stacks, the use of the GCC plugin mechanism for structure layout randomization, the hardened usercopy mechanism, and



Linux Kernel Development Report, The Linux Foundation, 2017

Kernel Version	Changes per hour
4.8	7.97
4.9	9.65
4.10	7.63
4.11	7.57
4.12	9.64
4.13	8.52

Linux Kernel Development Report, The Linux Foundation, 2017

stable_api_nonsense.txt

This is being written to try to explain why Linux does not have a binary kernel interface, nor does it have a stable kernel interface. Please realize that this article describes the **in kernel** interfaces, not the kernel to userspace interfaces. The kernel to userspace interface is the one that application programs use, the syscall interface. That interface is **very** stable over time, and will not break.

Executive Summary

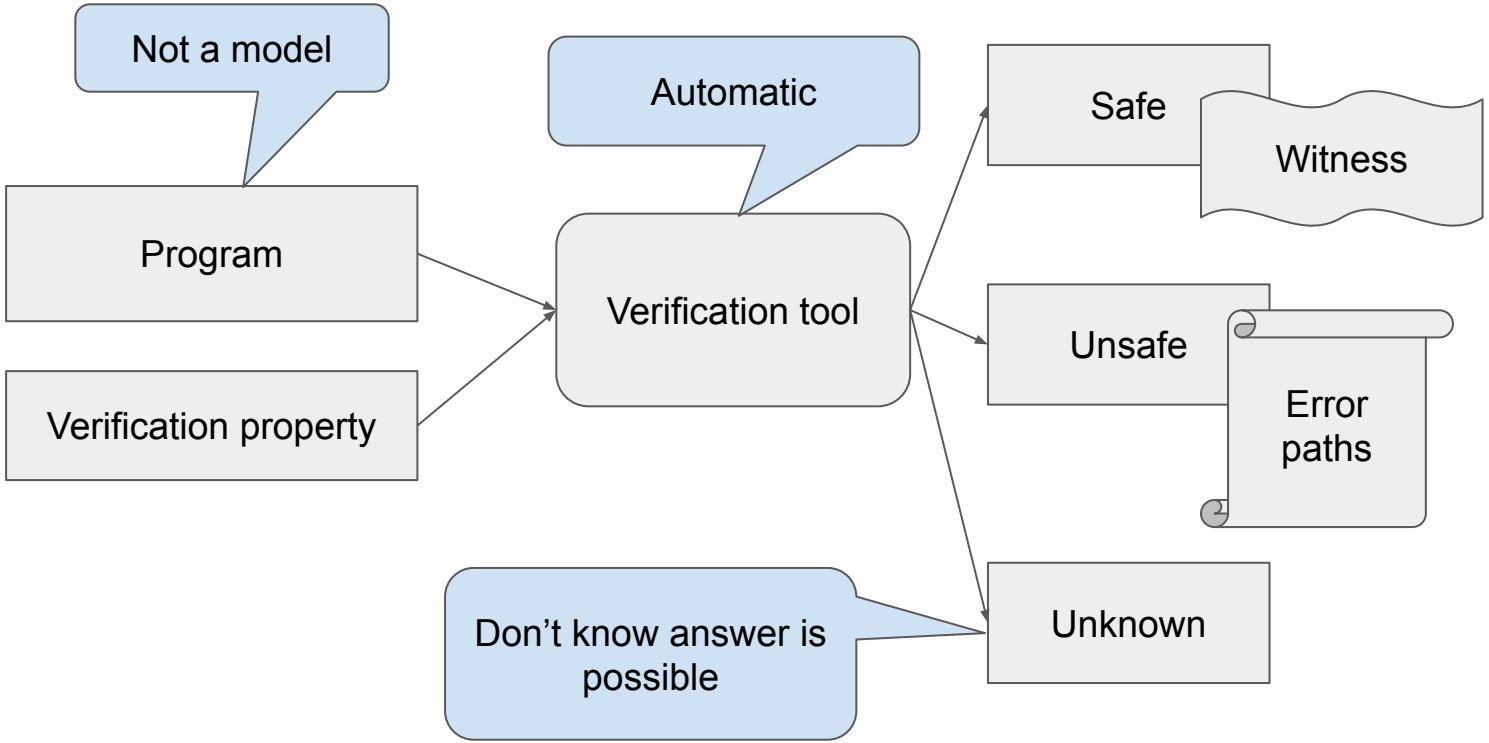
You think you want a stable kernel interface, but you really do not, and you don't even know it. What you want is a stable running driver, and you get that only if your driver is in the main kernel tree. You also get lots of other good benefits if your driver is in the main kernel tree, all of which has made Linux into such a strong, stable, and mature operating system which is the reason you are using it in the first place.

Greg Kroah-Hartman

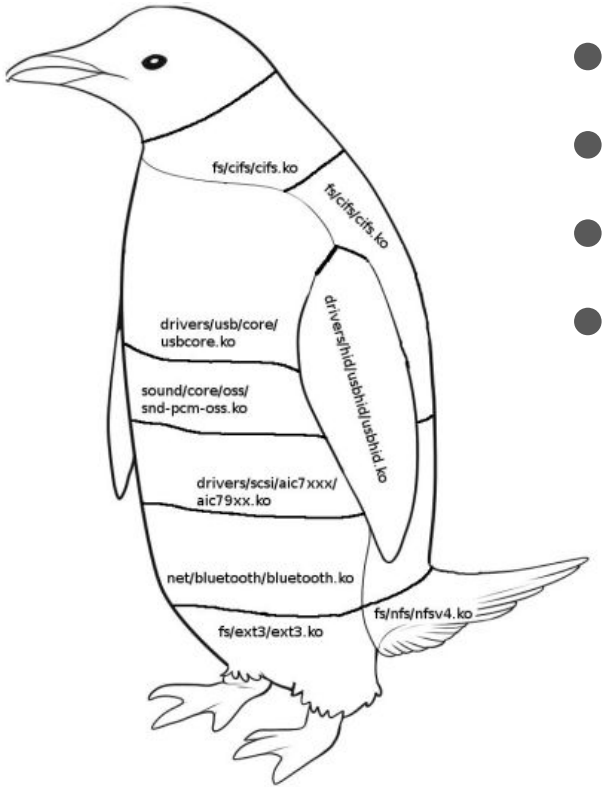
Essential Verification Features

- Scalable
- Continuous
- Automatic

Automatic Program Verification



Linux Device Drivers



- uniform
- important
- not too big
- not too complex

Many drivers share common structural patterns

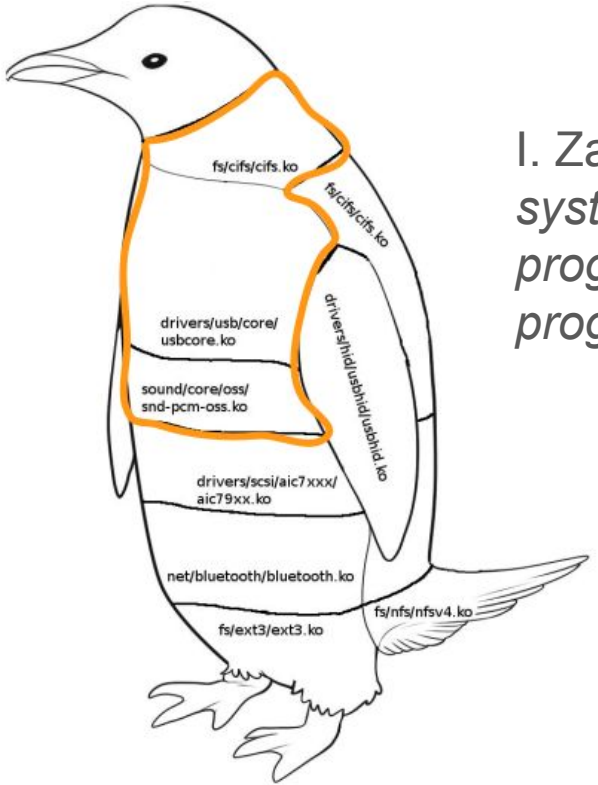
Share memory and privilege with the rest of the kernel

95% < 15 KLOC
90% < 10 KLOC
80% < 5 KLOC

(*) size of C files in .ko

No floating point, rarely use recursion

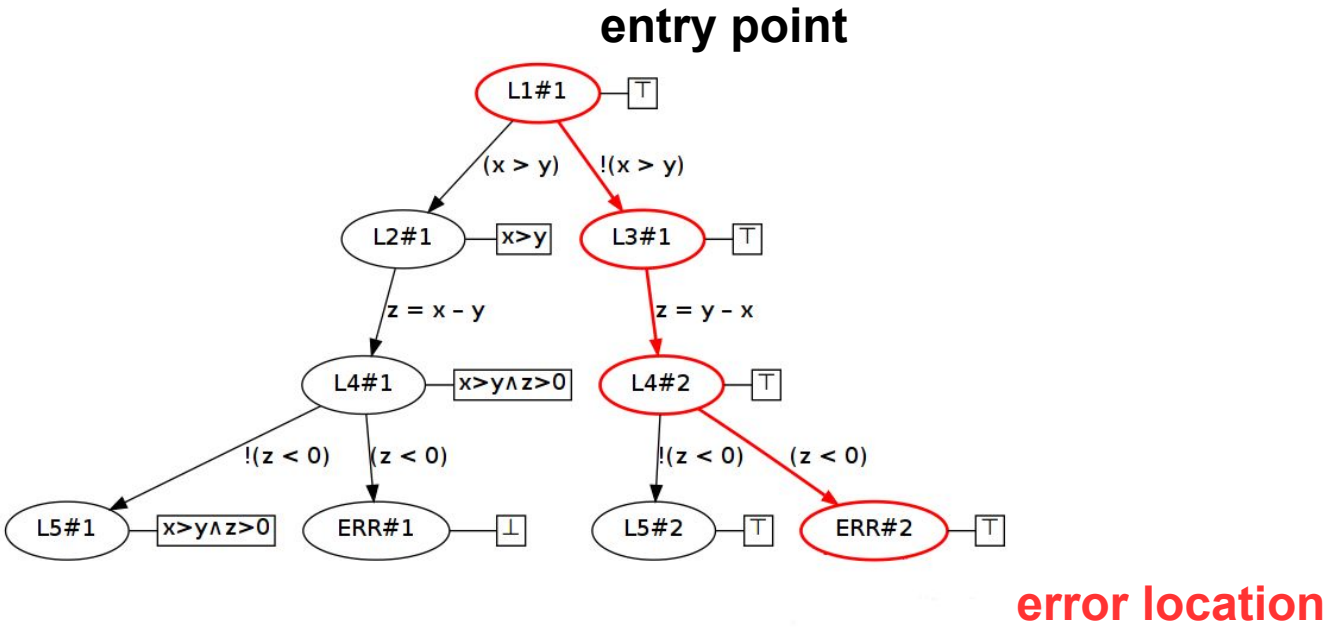
Decomposition



I. Zakharov. *Methods of decomposing systems and modeling environment of program modules for verification of C programs*, PhD Thesis, 2019

Software Model Checking

Reachability problem



Verification Tools World

```
int main(int argc, char* argv[])
{
 ...
 other_func(var);
 ...
}
```

```
void other_func(int v)
{
  ...
  assert( x != NULL);
}
```

Device Driver World

```
int usbpn_open(struct net_device *dev) { ... };
int usbpn_close(struct net_device *dev) { ... };
struct net_device_ops usbpn_ops = {
        .ndo_open = usbpn_open, .ndo_stop = usbpn_close
};
int usbpn_probe(struct usb_interface *intf, const struct usb_device_id *id){
        dev->netdev_ops = &usbpn_ops;
        err = register_netdev(dev);
}
void usbpn_disconnect(struct usb_interface *intf){...}

struct usb_driver usbpn_struct = {
        .probe = usbpn_probe, .disconnect = usbpn_disconnect,
};
int __init usbpn_init(void){ return usb_register(&usbpn_struct);}
void __exit usbpn_exit(void){usb_deregister(&usbpn_struct );}

module_init(usbpn_init);
module_exit(usbpn_exit);
```

Device Driver World

```
int usbpn_open(struct net_device *dev) { ... };
int usbpn_close(struct net_device *dev) { ... };
struct net_device_ops usbpn_ops = {
        .ndo_open = usbpn_open, .ndo_stop = usbpn_close
};
int usbpn_probe(struct usb_interface *intf, const struct usb_device_id *id){
        dev->netdev_ops = &usbpn_ops;
        err = register_netdev(dev);
}
void usbpn_disconnect(struct usb_interface *intf){...}

struct usb_driver usbpn_struct = {
        .probe = usbpn_probe, .disconnect = usbpn_disconnect,
};
int __init usbpn_init(void){ return usb_register(&usbpn_struct);}
void __exit usbpn_exit(void){usb_deregister(&usbpn_struct );}

module_init(usbpn_init);
module_exit(usbpn_exit);
```

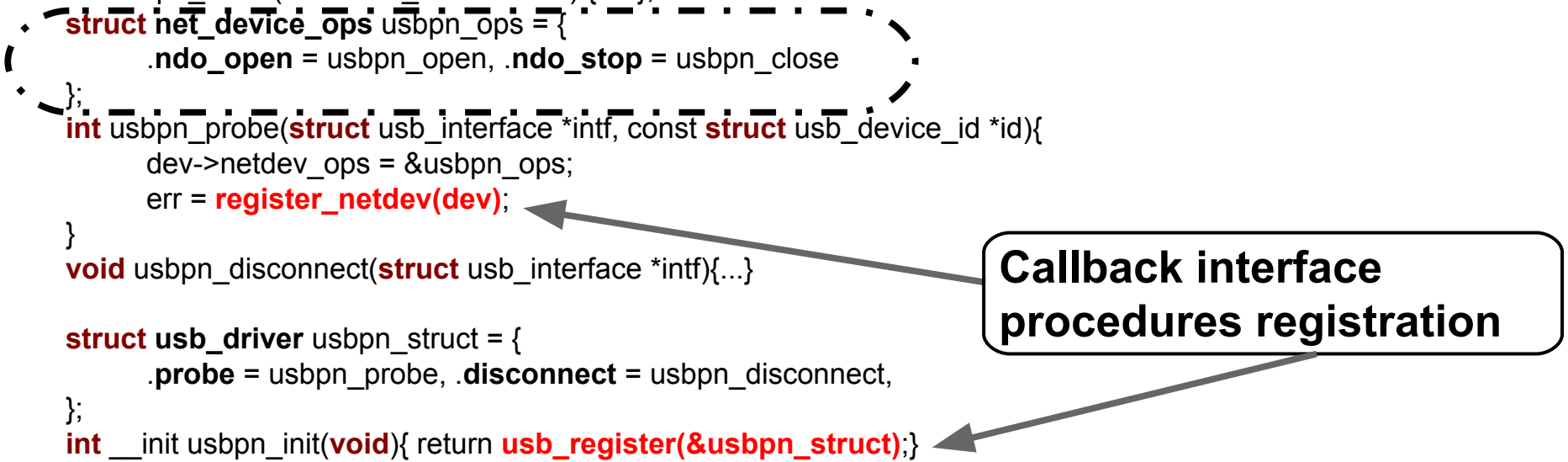
Callback interface procedures registration

Device Driver World

```
int usbpn_open(struct net_device *dev) { ... };
int usbpn_close(struct net_device *dev) { ... };
struct net_device_ops usbpn_ops = {
        .ndo_open = usbpn_open, .ndo_stop = usbpn_close
};
int usbpn_probe(struct usb_interface *intf, const struct usb_device_id *id){
        dev->netdev_ops = &usbpn_ops;
        err = register_netdev(dev);
}
void usbpn_disconnect(struct usb_interface *intf){...}

struct usb_driver usbpn_struct = {
        .probe = usbpn_probe, .disconnect = usbpn_disconnect,
};
int __init usbpn_init(void){ return usb_register(&usbpn_struct);}
void __exit usbpn_exit(void){usb_deregister(&usbpn_struct );}

module_init(usbpn_init);
module_exit(usbpn_exit);
```



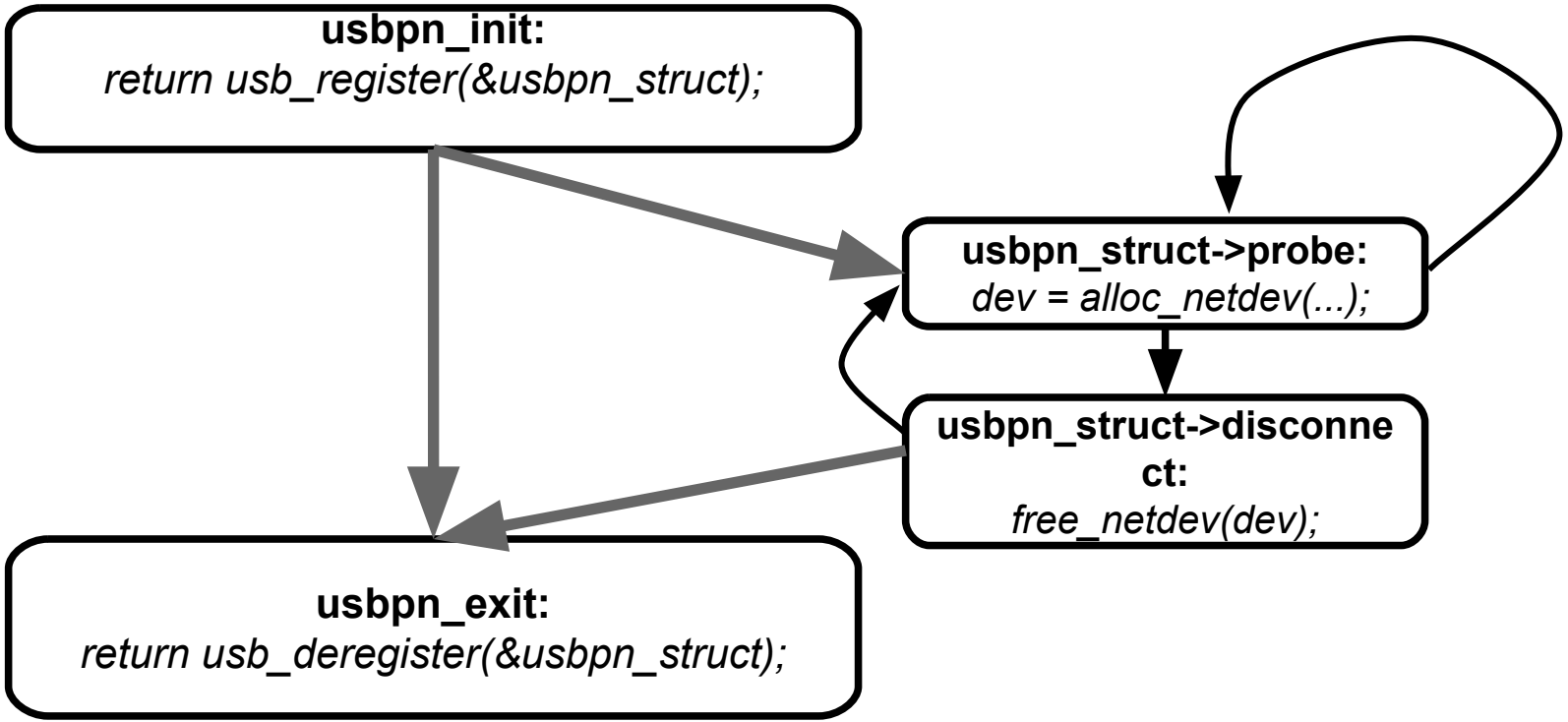
Driver Environment Model

```
int main(int argc, char* argv[])
{
  usbpn_init();
  for(;;) {
    switch(*) {
     case 0: usbpn_probe(*,*,*); break;
     case 1: usbpn_open(*,*); break;
     ...
    }
  }
  usbpn_exit();
}
```

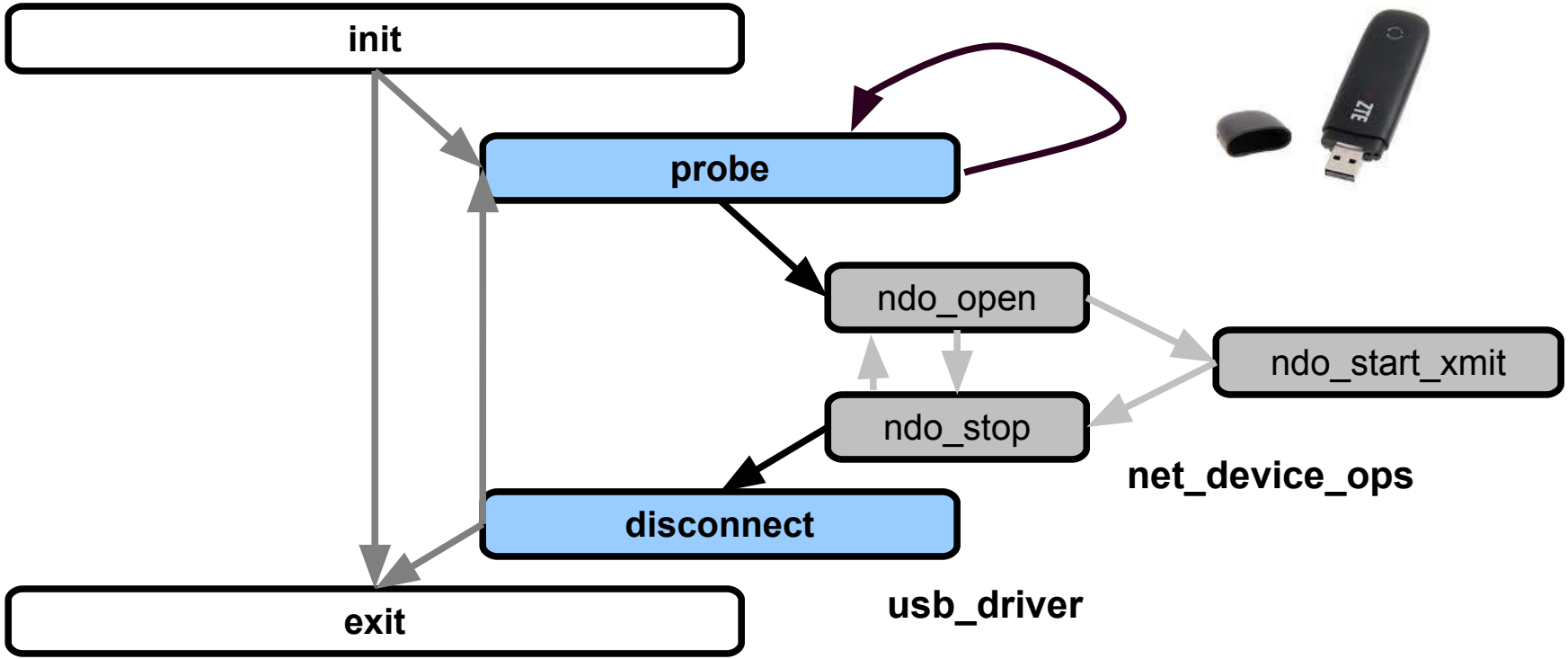
Driver Environment Model (2)

- Order limitation
  - open() after probe(), but before disconnect()
- Implicit limitations
  - read() only if open() succeed
  - and it is specific for each class of drivers

Scenarios of Handler Invocations



Scenarios of Several Groups



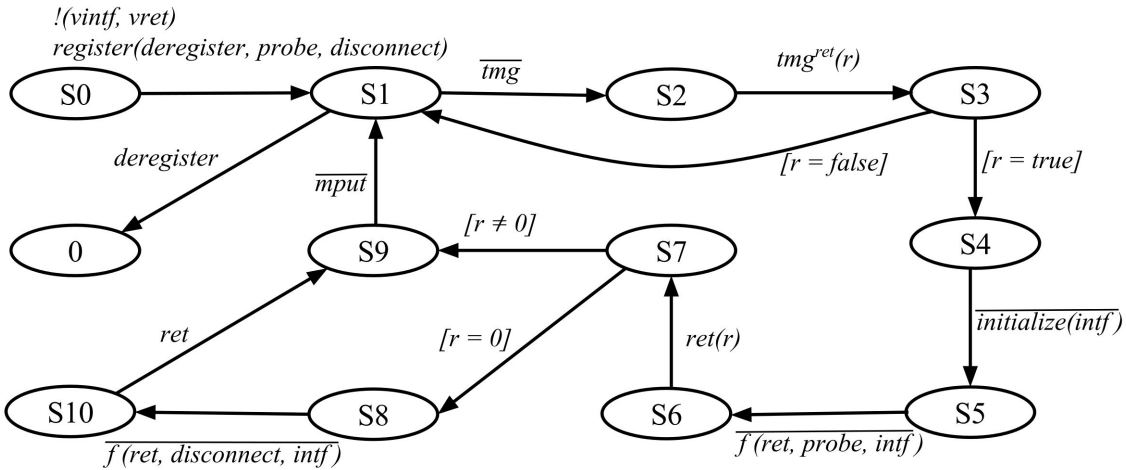
Driver Environment Model (3)

- Correct & Complete
  - should reproduce the same scenarios of interaction with a driver as in the real kernel environment

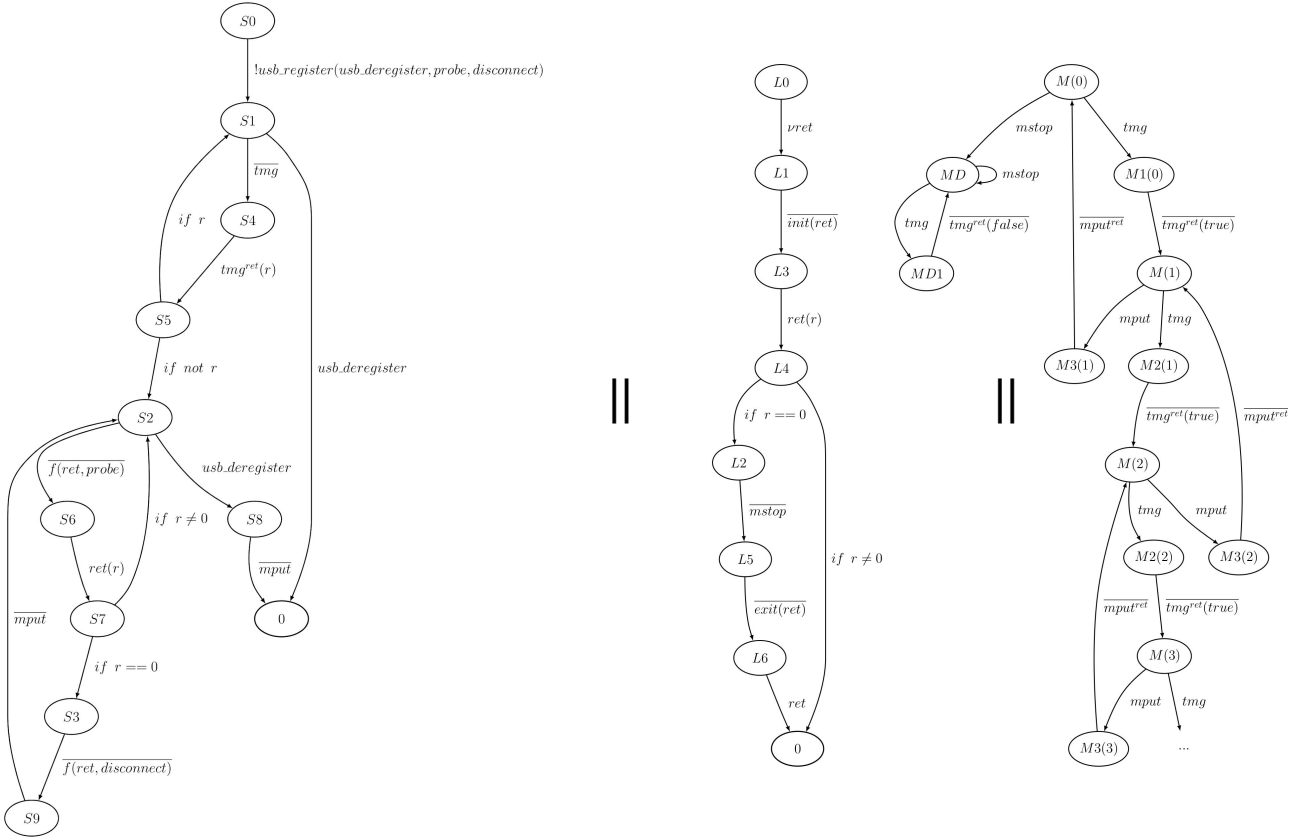
Driver Environment Model (3)

- Correct & Complete
  - should reproduce the same scenarios of interaction with a driver as in the real kernel environment
- Simple Enough
  - should satisfy the requirements of static verification tools

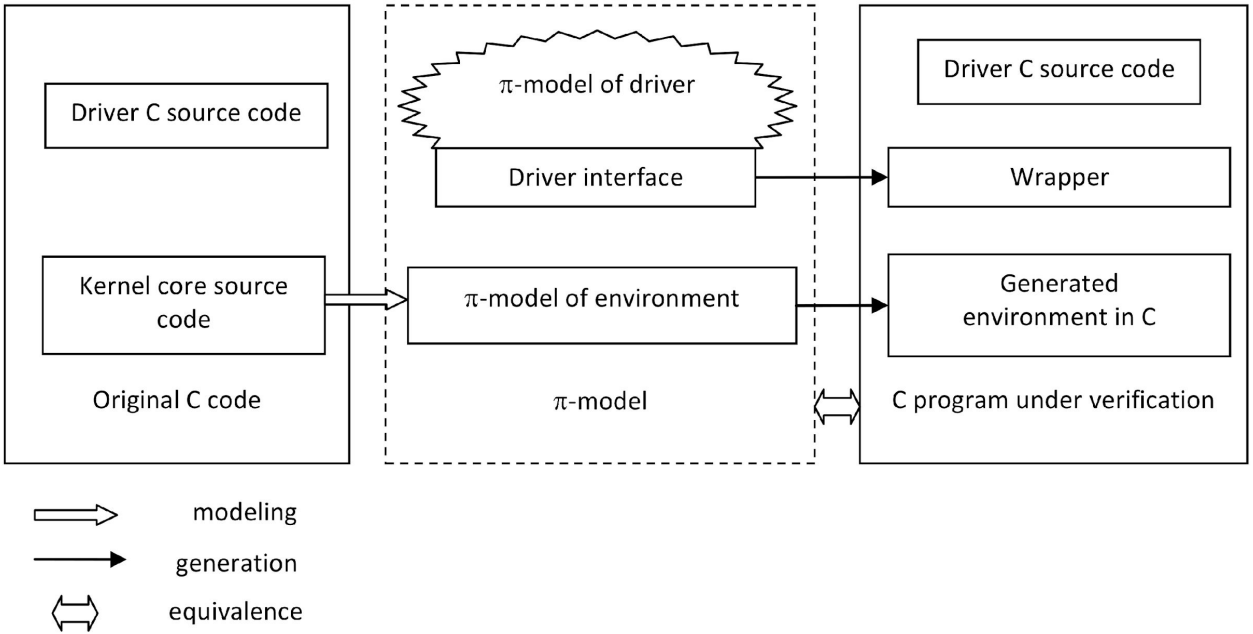
π-model of Environment



π-model of Environment



Modeling Driver Environment



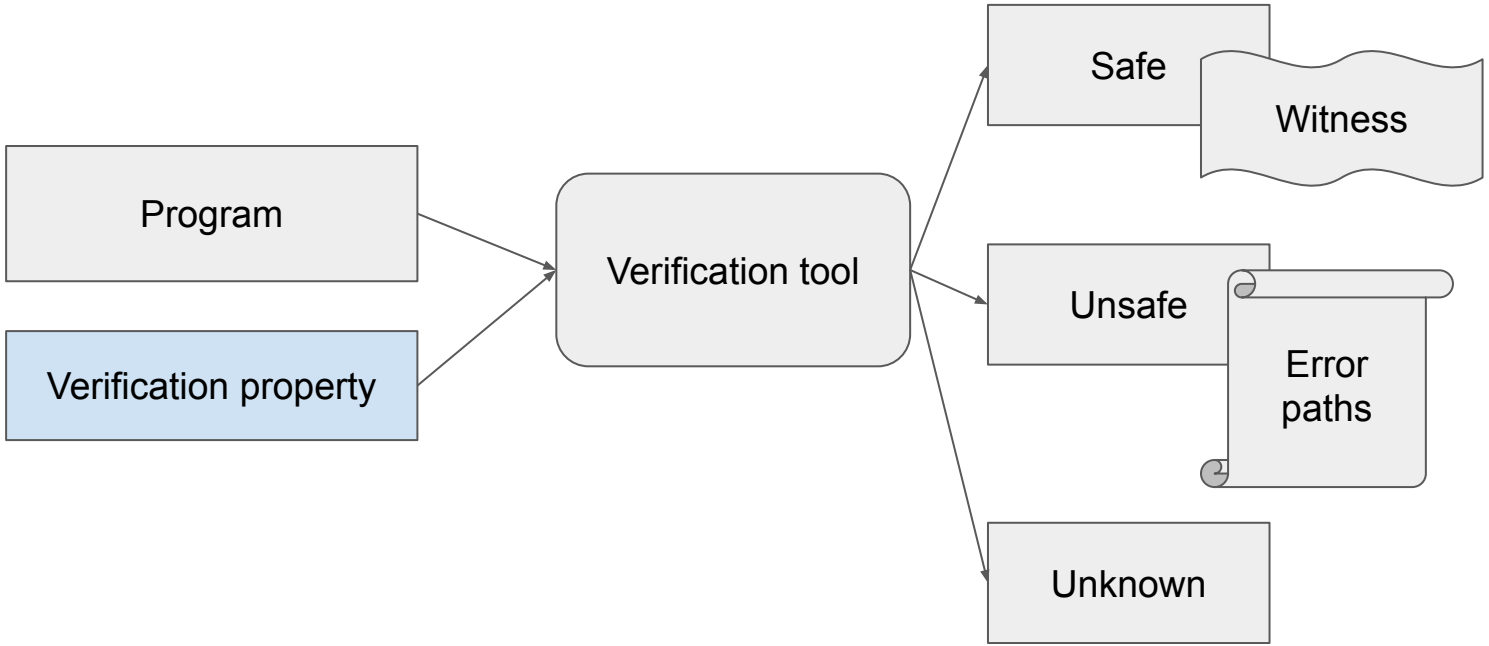
Generated Environment in C

```
void main(int argc, char* argv[]) {
  while(true) {
    switch(__VERIFIER_nondet_int()) {
      case 0: //process P1, action α1:
        if(s->state == K ) {
          ...
        }
        break;
      case 1: //process P1, action α2:
      ...
    }
  }
  break_loop:
}
```

Coverage of Drivers in Linux 3.14.78

Directory Path	Line Coverage	Function Coverage
drivers/gpu	13% (13476/96446)	4% (190/4626)
drivers/infiniband	34% (36314/104686)	24% (1056/4357)
drivers/isdn	58% (30777/52354)	52% (993/1877)
drivers/media	47% (118324/249319)	27% (2687/9677)
drivers/net	66% (363300/549366)	62% (15033/24147)
drivers/staging	66% (161825/244187)	55% (4554/8250)
drivers/usb	57% (96389/168336)	53% (3796/7101)
drivers/video	58% (32225/54924)	48% (1127/2302)

Automatic Program Verification



Commit Analysis

- All patches in Linux stable kernels 2.6.35 – 3.0:
  - Period: 26 Oct 2010 – 26 Oct 2011
  - 1503 device driver patches
- Main goal: detect and classify typical bugs

Commit Analysis (2)

<table>
<tr><td colspan="3">Changes in drivers</td></tr>
<tr><td rowspan="2">Improvements in functionality - 21%</td><td colspan="2">Bug fixes - 79%</td></tr>
<tr><td>Non-typical bugs - 52%</td><td>Typical bugs - 27%</td></tr>
</table>

1503 patches for device drivers in stable trees

Commit Analysis (3)

	Class	#	%
1	sync:race	60	17.2%
2	specific:resource	32	9.2%
3	generic:null_ptr_deref	31	8.9%
4	specific:check_params	25	7.2%
5	generic:resource	24	6.9%
6	specific:context	19	5.4%
7	specific:uninit	17	4.9%
8	generic:syntax	14	4.0%
9	specific:lock	12	3.4%
10	sync:deadlock	11	3.2%
11	specific:style	10	2.9%

	Class	#	%
12	specific:net	10	2.9%
13	specific:usb	9	2.6%
14	generic:int_overflow	8	2.3%
15	generic:buffer_overflow	8	2.3%
16	specific:check_ret_val	7	2.0%
17	generic:uninit	6	1.7%
18	specific:dma	4	1.1%
19	specific:device	4	1.1%
20	specific:misc	27	7.7%
21	generic:misc	11	3.2%

Properties

- specific (domain-specific rules)
  - Reachability
- sync
  - Concurrency safety (data races, deadlocks)
- generic
  - Memory safety (invalid dereference, invalid free, memory leaks, …)
  - Overflow
  - Termination

Properties

- specific (domain-specific rules)
  - Reachability
- sync
  - Concurrency safety (data races, deadlocks)
- generic
  - Memory safety (invalid dereference, invalid free, memory leaks)
  - Overflow
  - Termination

CEGAR, BAM, Predicate and Value Analysis

CEGAR, BAM, Thread Modular, Lock, Predicate Analysis

SMG

CPAchecker Tool

- Input language C, Java
- Implemented in Java
- Open Source: Apache 2.0 License
- ~130 contributors so far from 15 universities/institutions
- 1,420,000 lines of code
- Started 2007

https://cpachecker.sosy-lab.org

CPA Algorithm

a0 - initial abstract state, R - reached set, W - waitlist

R = {a0}, W = {a0}

```
while(W ≠ ∅) {
  take a from W
  foreach( a' | a ⇝ a') {

    foreach(a'' in R) {
      m = merge(a', a'')
      if(m ≠ a'') { replace a'' by m in R and W }
    }

    if(!stop(a', R)) {  add a' to R and W  }
    if(a' is error state) { return UNSAFE }
}

return SAFE
```

Configurable Program Analysis

CPA = (D, ⇝, merge, stop)

D - abstract domain (E, C, [[.]])

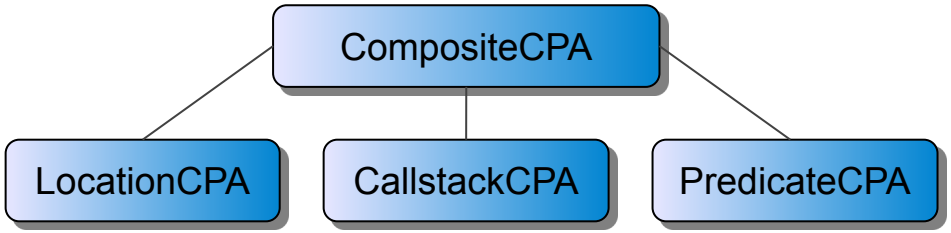
⇝ - transfer operator

merge operator

stop operator

Configurable Program Analysis

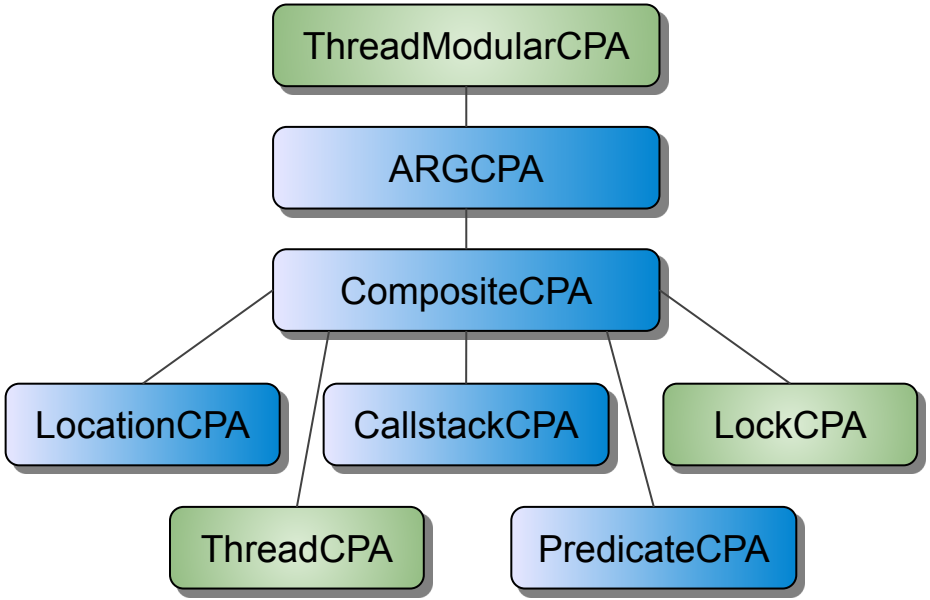
- Composite - C1 × C2 × … × CN
- Location
- Callstack
- Predicate
- Value
- Block Abstraction Memoization (BAM)
- Symbolic Memory Graph (SMG)
- Thread-Modular (TM)
- Lock
- …



Algorithms

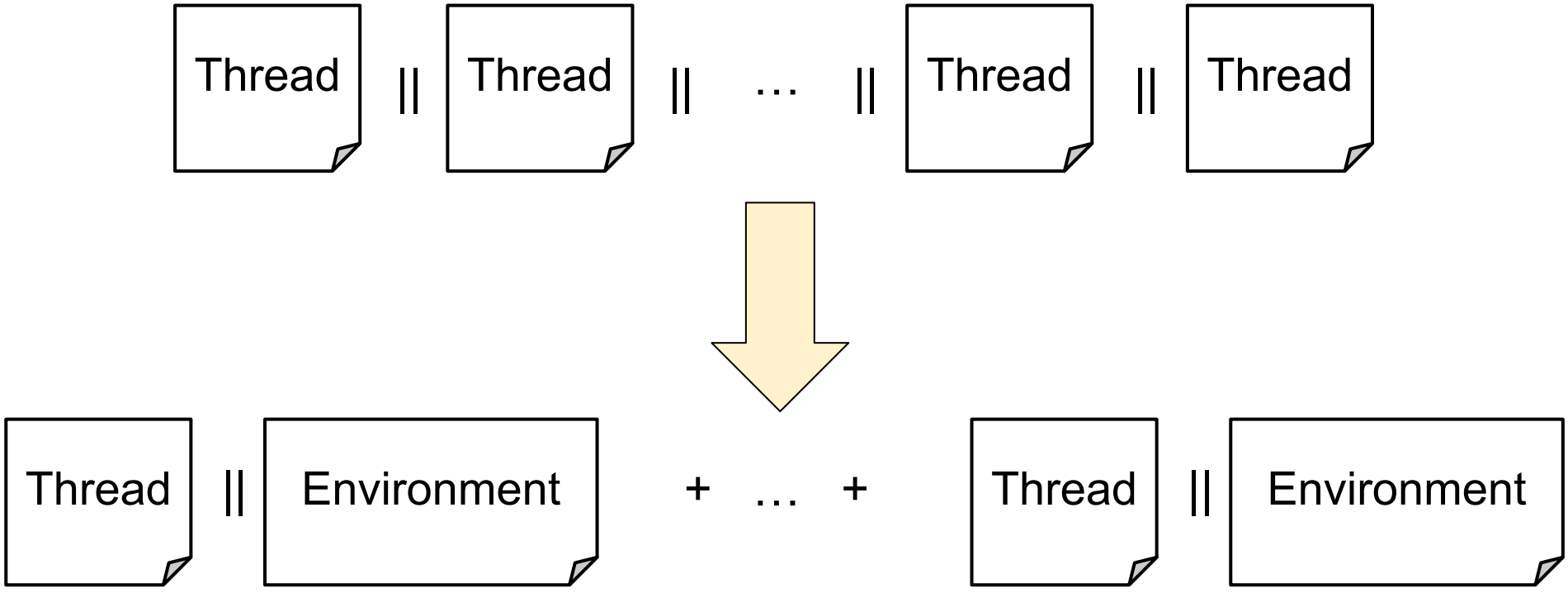
- Configurable Program Analysis (CPA)
- Counter-example Guided Abstraction Refinement (CEGAR)
- Bounded Model Checking (BMC)
- k-Induction
- IMPACT
- Property-directed reachability (PDR)
- …

Concurrency Properties
CPALockator - Thread-Modular Approach with Projections

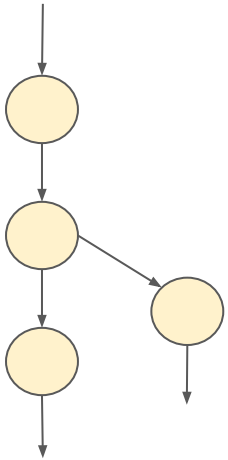


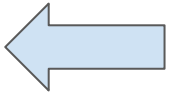
P. Andrianov, *Analysis of synchronization correctness in operating systems components*, PhD Thesis, 2021

Thread-Modular Approach



Thread Interaction via Projections

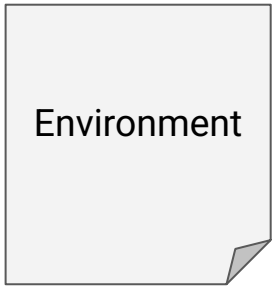


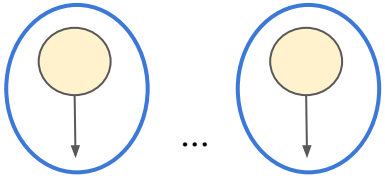


g = 5

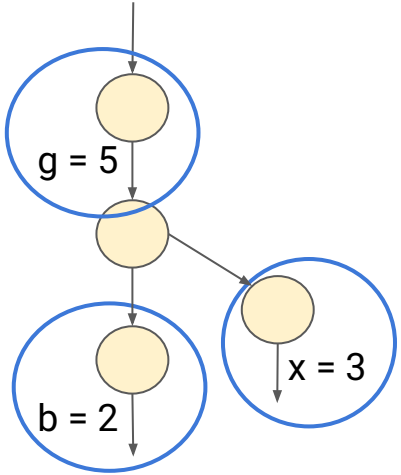
x = 3

b = 2

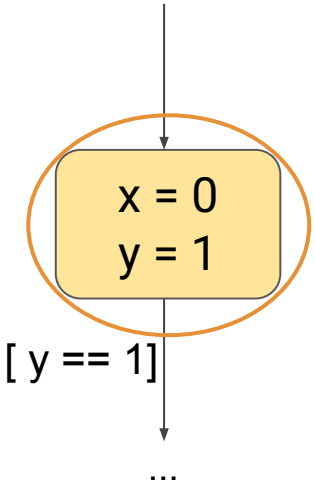




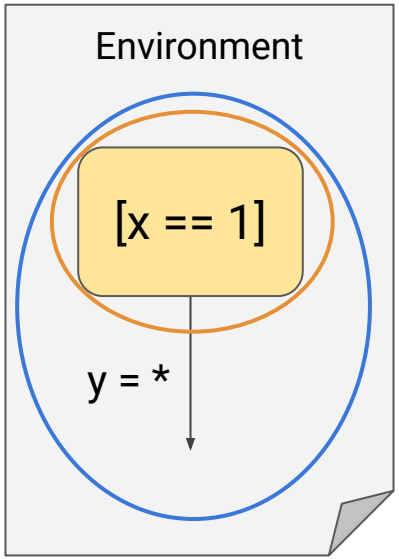
Projections



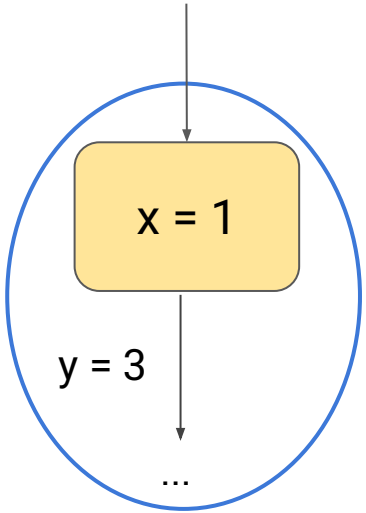
Application of Projections



Transition in the Thread1

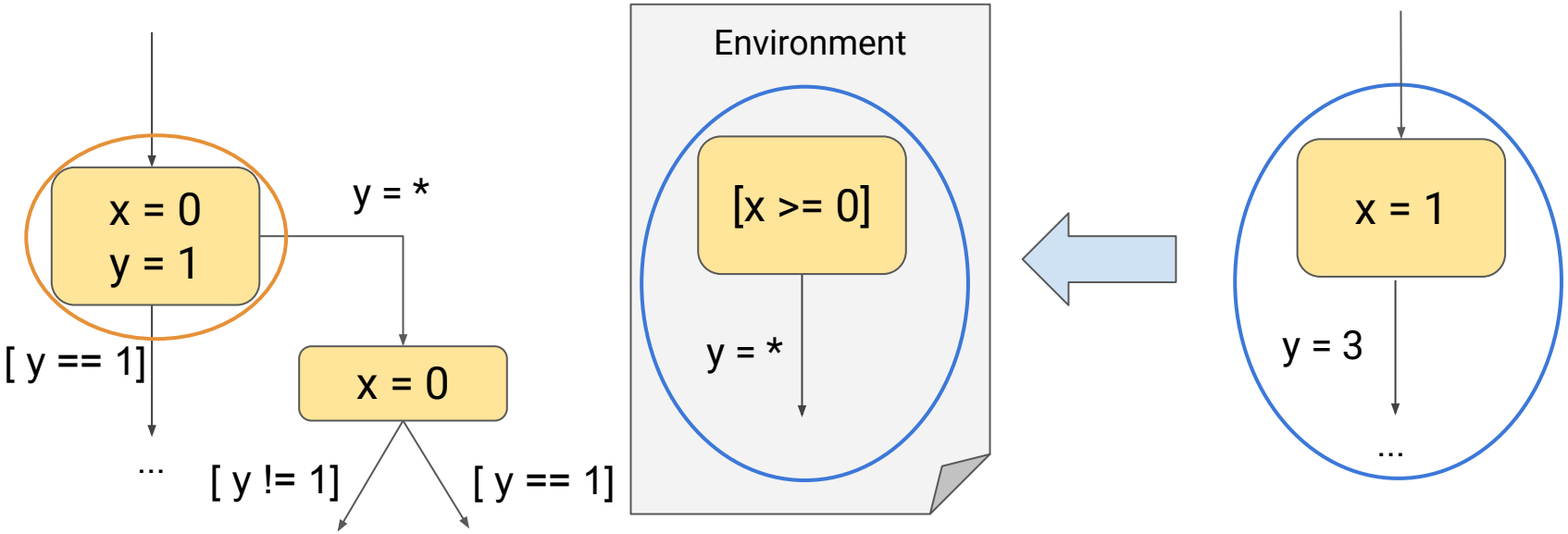


Projection



Thread 2

Application of Projections



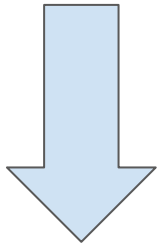
Transition in the Thread1

Projection

Thread 2

New Requirements for CPA Operators

- $\exists k > 0.\ \forall R \subset E\ .\ [[\text{Reach}^k(R)]] \supseteq \cup_{\tau \in [[R]]} \{\tau' \mid \tau \rightarrow \tau'\}$
- $\forall e, e'\ .\ e' \sqsubseteq \text{merge}(e, e')$
- $\forall e\ .\ \text{stop}(e, R) \Rightarrow [[e]] \subseteq \cup_{e' \in R} [[e']]$



$$[[\text{CPA}(\mathbb{D}, s_0)]] \supseteq \text{Reach}_{\rightarrow}([[s_0]])$$

Comparison on SV-COMP Benchmarks

Benchmark	kLOC	Threads	CPALockator	Other participants
cafe_ccic.ko_false	13	4	X	X
nsc-ircc.ko_false	13	5	X	X
w83977af_ir.ko_false	10	3	V	X
spi-tegra20-slink.ko_false	7	4	V	X
adutux.ko_false	9	3	V	X
iowarrior.ko_false	8	3	V	X
cafe_ccic.ko_true	13	4	X	X

Verification Results - Data races

Linux kernel v4.2.6, drivers/net subsystem,
425 tasks

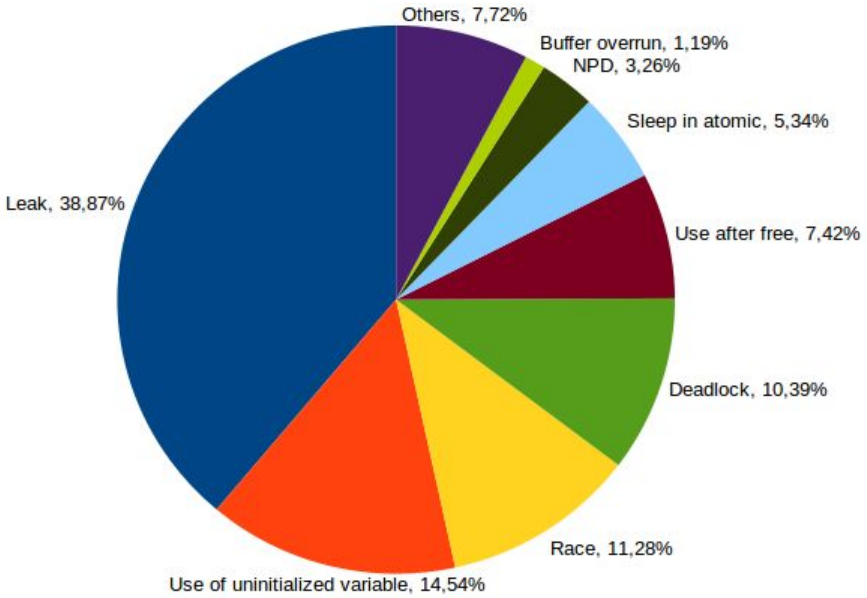
Number of warnings	85
Confirmed bugs	48%
CPU time	36 hours
Wall time	3 hours

Verification Results - Data races

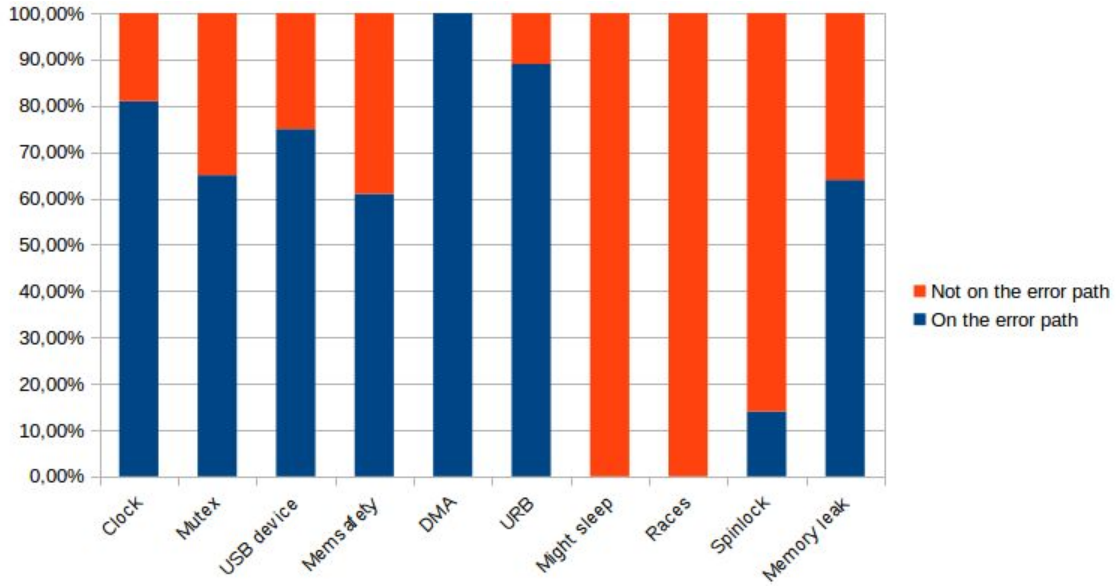
Real-Time OS

	RTOS 1	RTOS 2
Number of warnings	461	981
Confirmed bugs	42%	21%
CPU time	9 min	14 min
Number of analyzed threads	497	37
Lines of code, thousands	400	300

Consequences



On the Error Path? (for top 10 rules)



Links

- Results of Linux kernel verification
  - More than 400 problems were found (commits and commits since 2024).
  - 8+ GSOC projects on Linux kernel verification
- Frameworks

  KLEVER Verification Framework

  ISPRAS fork of CPAchecker