Klever
Enabling Model Checking for the Linux Kernel

Evgeny Novikov

Ivannikov Institute for System Programming of the Russian Academy of Sciences,
Linux Verification Center

ELISA Webinar series, May 27, 2019

Goal of This Presentation

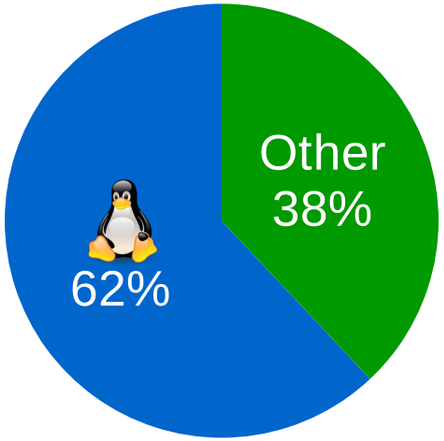
Consider benefits, restrictions and the state-of-the-art
of using formal methods for big critical industrial programs
like the Linux kernel

Outline

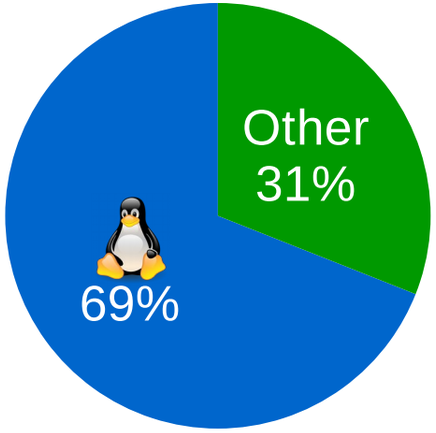
- Why is it worth to apply formal methods for the Linux kernel?
- Klever: Enabling model checking for the Linux kernel
- Current achievements and directions for further development of Klever
- Demonstration of Klever
- Conclusion

Linux Conquers the World

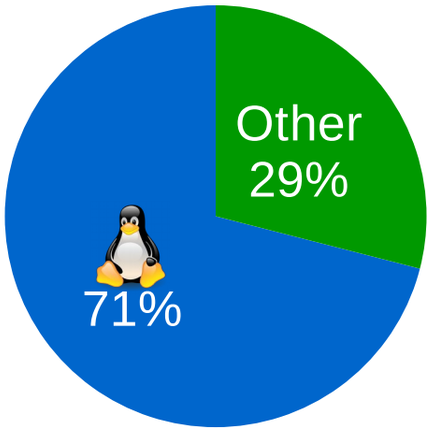
Embedded[1]



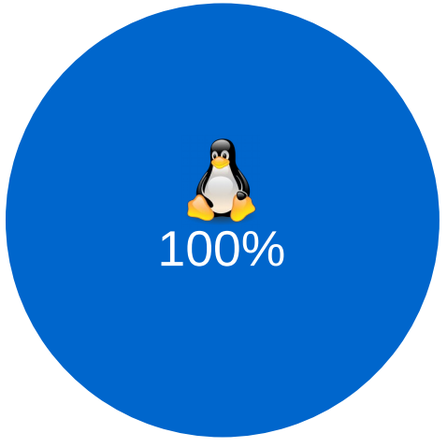
Servers[2]



Mobile & Tablet[3]



Supercomputers (TOP500)[3]



[1] 2017 Embedded Markets Study
[2] Usage of operating systems for websites, May 2019
[3] Mobile & Tablet Operating System Market Share Worldwide, April 2019
[4] TOP500 List Statistics (Operating system family), November 2018

Linux Kernel Development*

Size	18 MLOC
Changes on average per a day	184 commits 150 KLOC
Contributors per a last year	4 thousands
Estimated efforts	6 thousands of person-years

*The Linux Kernel Open Source Project on Open Hub, May 2019.
We have to divide some numbers by 2 because of a bug in the Open Hub service.

Software Quality Assurance Techniques

Code review	Finding issues according to reviewers experience
Testing	Ensuring absence of violations of checked requirements on considered execution paths
Static analysis	Detecting rather many defects and security vulnerabilities quite easy and fast
Formal methods	Detecting hard-to-find faults and proving formal correctness of programs checked against particular requirements under certain assumptions

Comprehensive Formal Verification of the Linux kernel

	seL4*	Linux kernel
Size	10 KLOC	18 MLOC
Estimated efforts of development	2.2 person-years	6 thousands of person-years
Estimated efforts of comprehensive formal verification	25 person-years	**68 thousands of person-years****

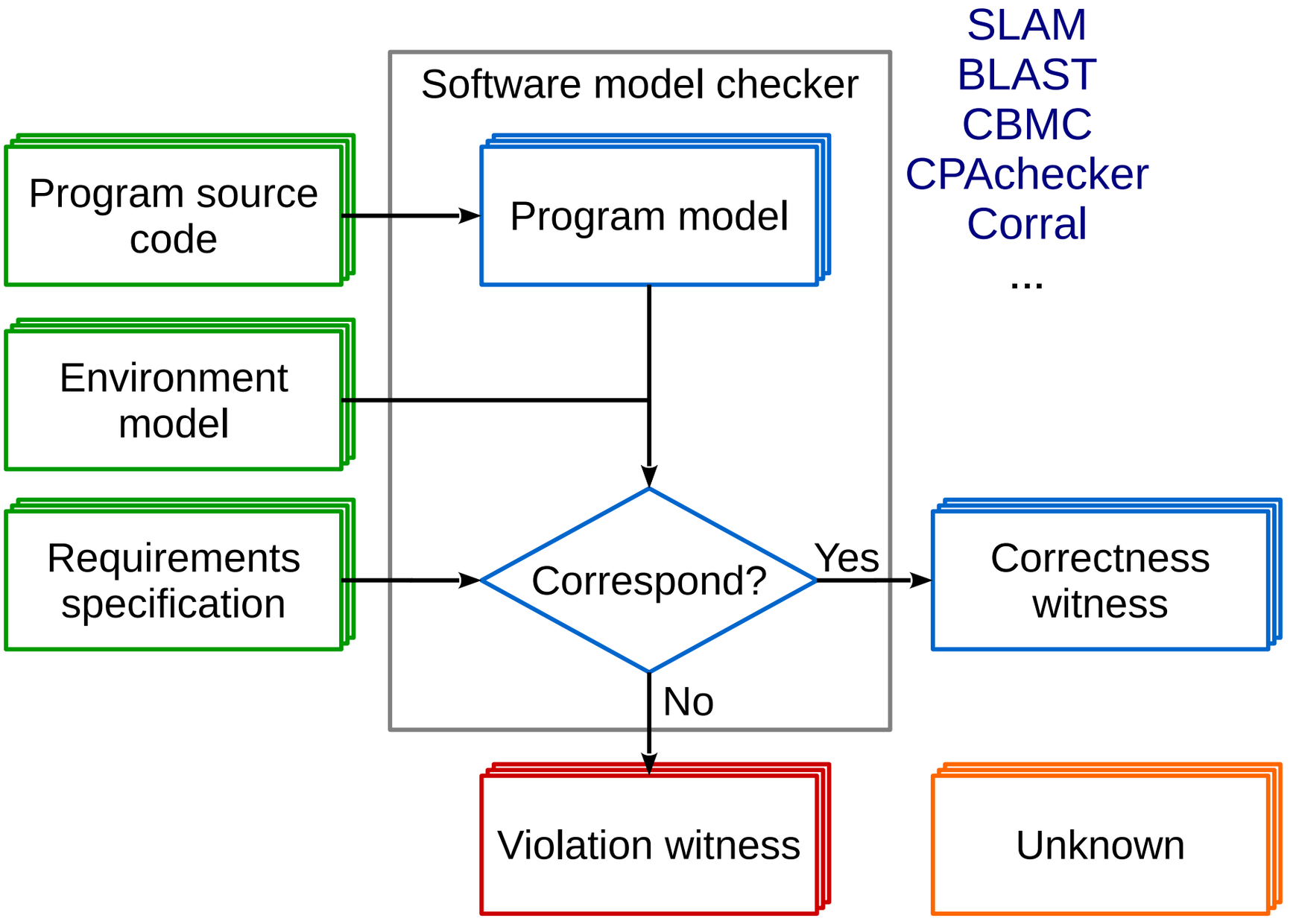
*According to Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive formal verification of an OS microkernel. ACM Trans. Comput. Syst. 32, 1, Article 2 (February 2014), 70 pages. DOI=http://dx.doi.org/10.1145/2560537

**By an order of magnitude

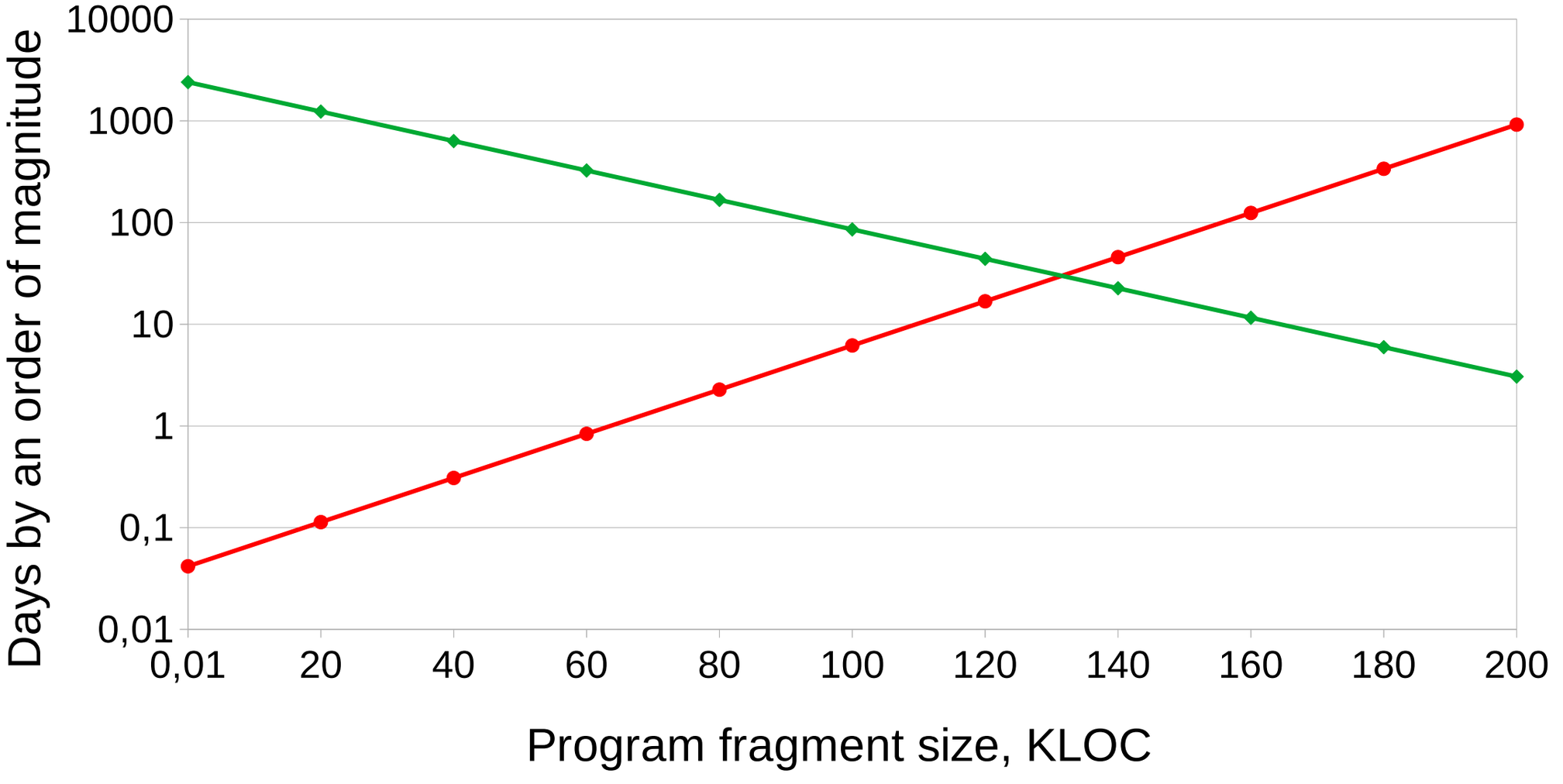
Ways to Apply Formal Methods for the Linux Kernel

- Improve existing and develop new methods and tools, teach much more proof engineers
  - it is worth doing but likely it will not change the situation dramatically in near future
- Reduce code bases under verification
  - 0.1% (36 KLOC) needs 68 person-years
  - 10% (3.6 MLOC) needs 6.8 thousands of person-years
  - ...
- **Check for some vital requirements, e.g. memory safety**
  - seems to be the only suitable way to apply formal methods for big critical industrial programs as a whole with moderate efforts

Software Model Checking (Automatic Software Verification)



Estimated Complexity of Software Model Checking for the Linux kernel



Pros and Cons of Software Model Checking

Pros

- No missed bugs*
- No false alarms*
- Automatic extraction and checking of models
- Formal and accurate evidences as verification results

Cons

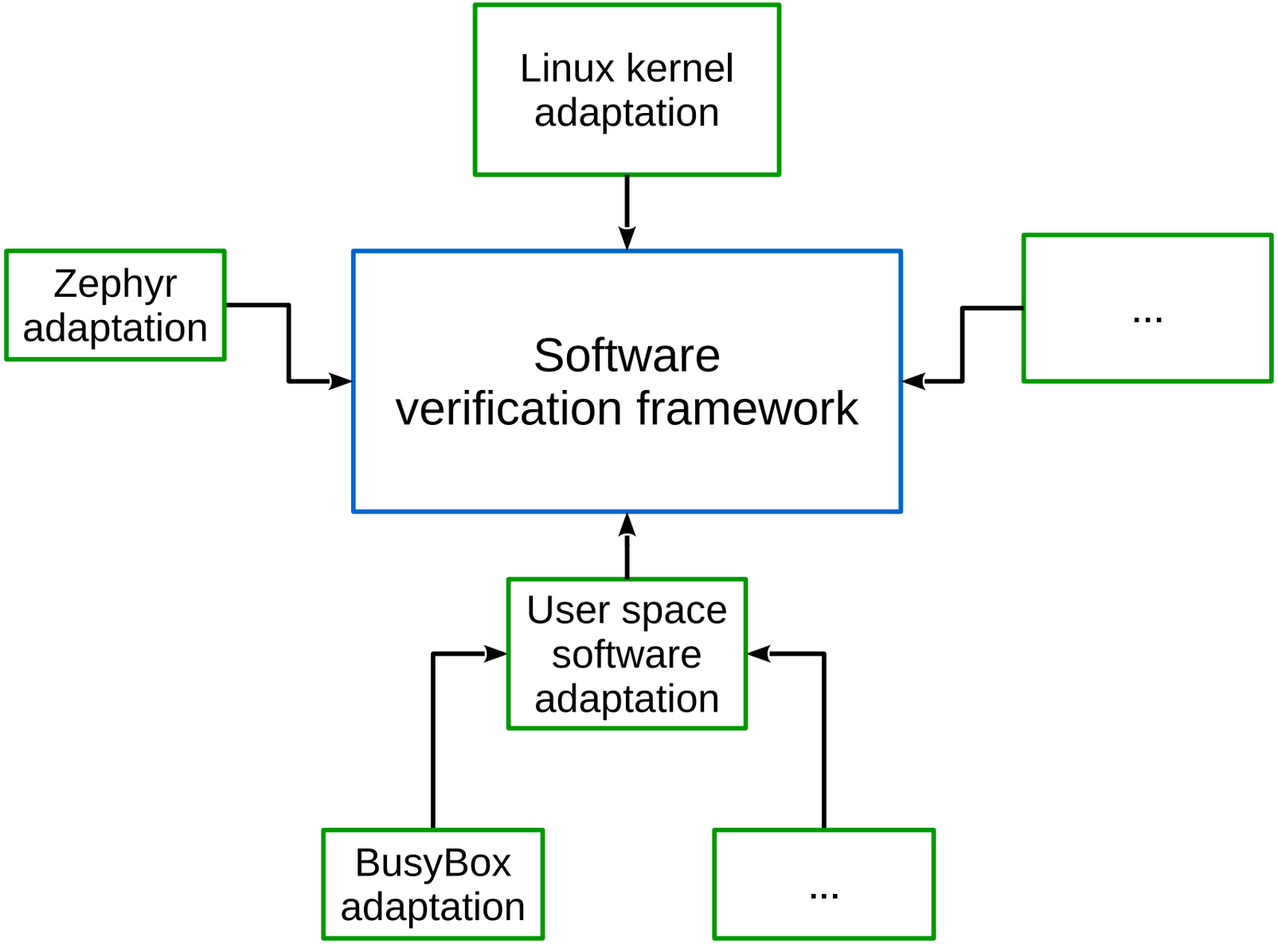
- Poor scalability for complex programs
- Need for rather accurate environment models
- Requirement specifications are not expressive
- Witnesses are not intended for manual analysis

*In case of ideal models and specifications

Outline

- Why is it worth to apply formal methods for the Linux kernel?
- Klever: Enabling model checking for the Linux kernel
- Current achievements and directions for further development of Klever
- Demonstration of Klever
- Conclusion

Klever Architecture

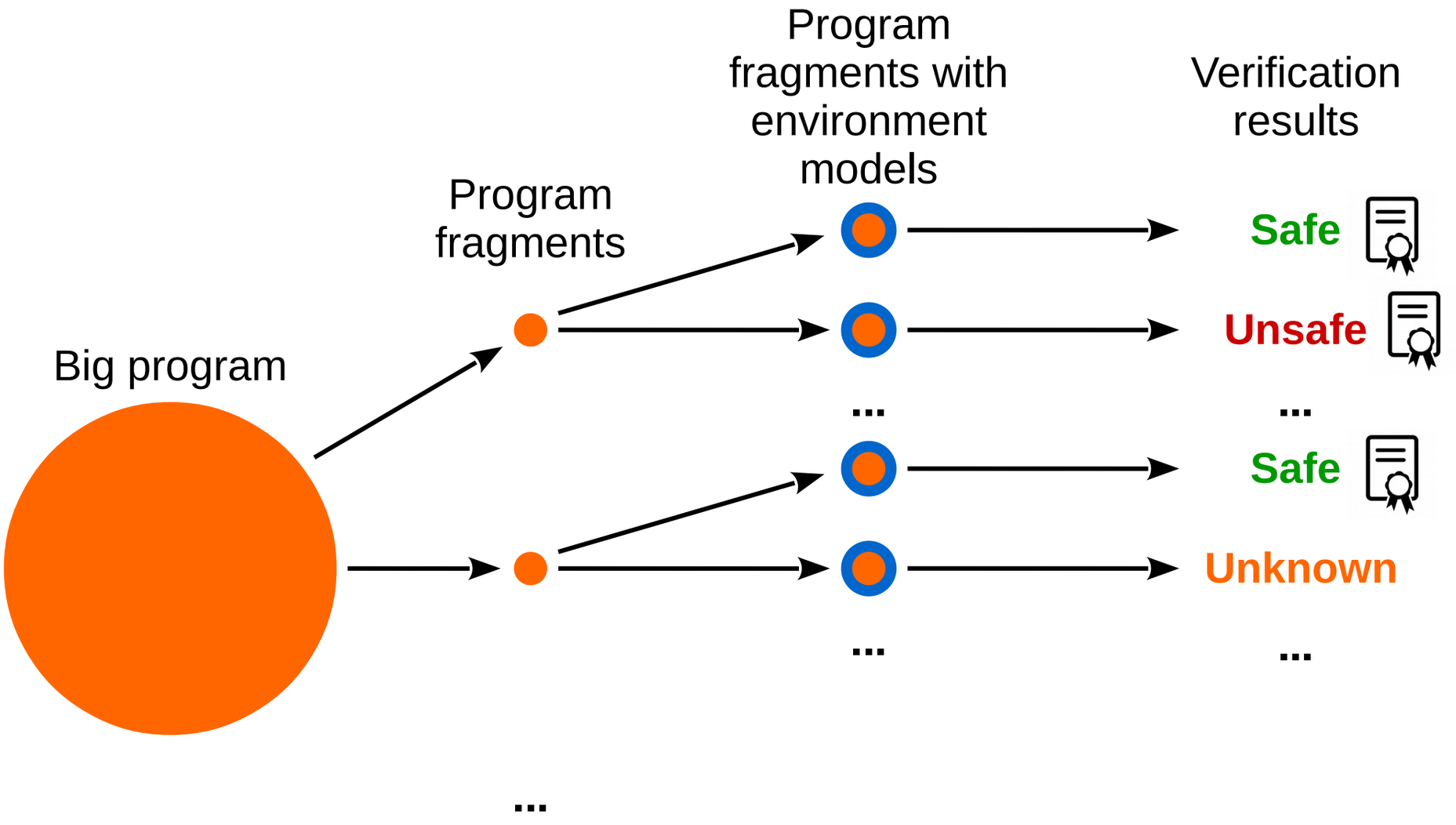


Klever Software Verification Framework

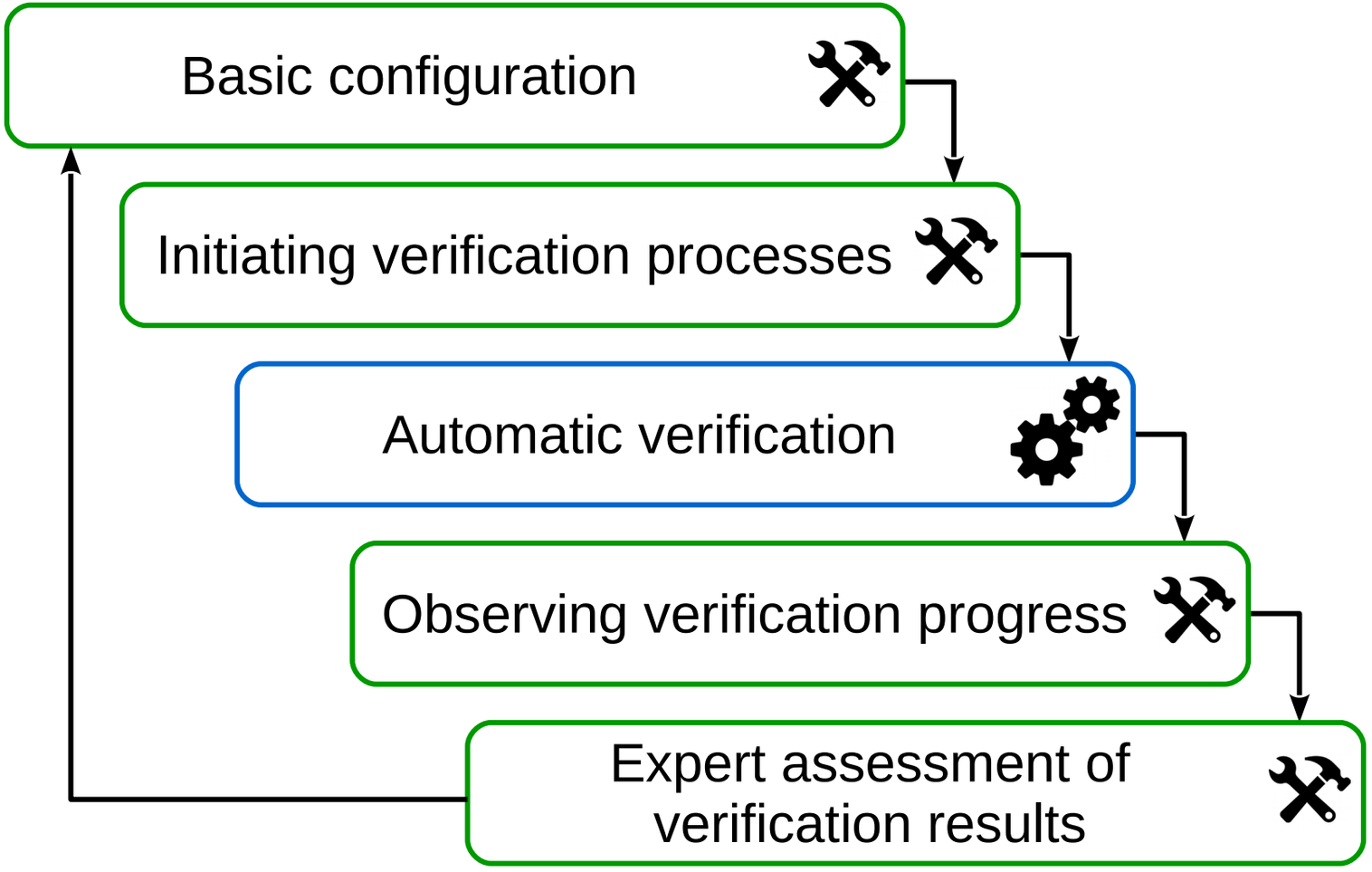
- Automating verification of software written in the GNU C programming language by means of software model checkers
  - decomposition of programs into fragments of moderate size⚒
  - generation of environment models⚒
  - configuration of software model checkers (CPAchecker & Ultimate Automizer)⚒
  - launch of software model checkers (locally or in the cloud)
  - preliminary processing of verification results
- Providing a convenient interface for
  - managing verification processes
  - expert assessment of verification results
- Ready for different versions of target programs

⚒These steps require extra efforts for developing configurations and specifications as well as for implementing more appropriate algorithms (project specific adaptation)

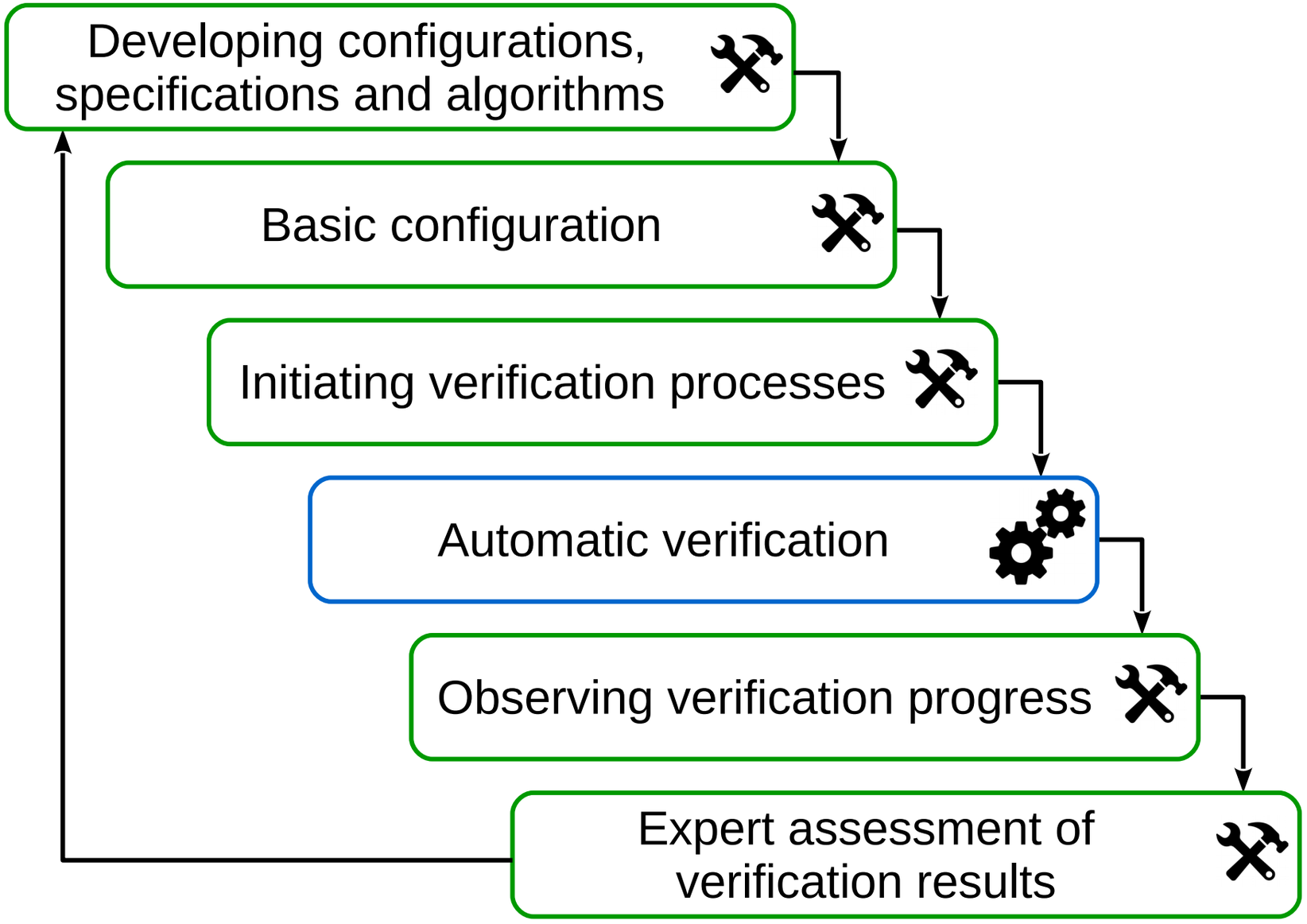
Klever Automatic Verification Workflow



Klever Complete Verification Workflow with Ready Project Adaptation



Klever Complete Verification Workflow without Ready Project Adaptation



Klever Adaptation for the Linux Kernel

- Automated decomposition of the Linux kernel into
  - loadable kernel modules
  - subsystems (proof-of-concept)
- Environment model specifications and generators for invoking most popular Linux kernel API
  - interrupts, timers, workqueues
  - callbacks of different device types (USB, PCI, SCSI, NET, etc.)
  - file system operations
- Requirement specifications for detecting
  - incorrect usage of the most popular Linux kernel API
  - memory safety issues
  - data races

Klever Development*

Size	110 KLOC
Contributors	9
Estimated efforts	27 person-years
Open source	https://forge.ispras.ru/projects/klever

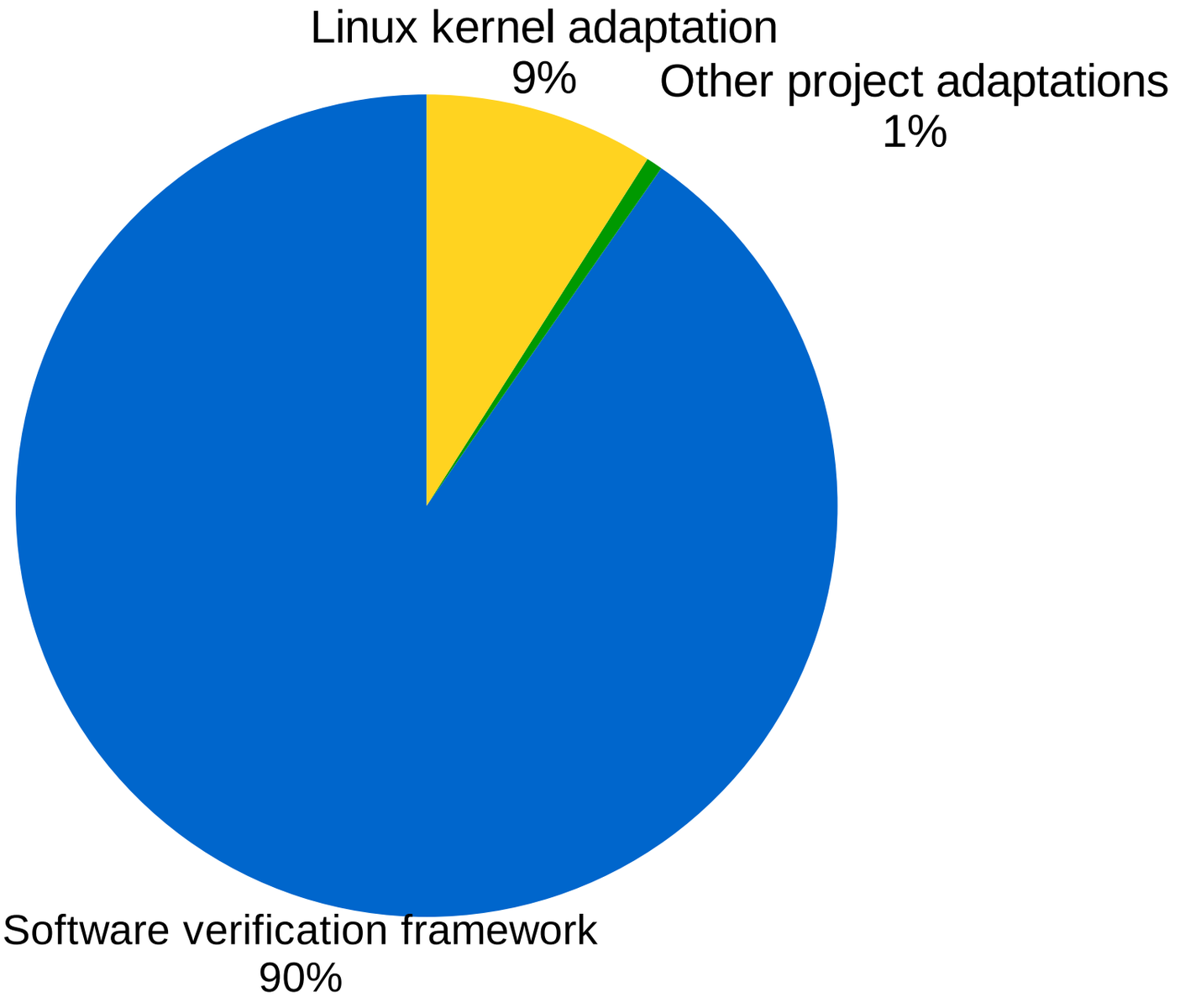
*The Klever Open Source Project on Open Hub, May 2019

CPAchecker Development*

Size	300 KLOC
Contributors	91
Estimated efforts	81 person-years
Open source	https://cpachecker.sosy-lab.org/

*The CPAchecker Open Source Project on Open Hub, May 2019

Estimated Efforts of Klever Development



Outline

- Why is it worth to apply formal methods for the Linux kernel?
- Klever: Enabling model checking for the Linux kernel
- Current achievements and directions for further development of Klever
- Demonstration of Klever
- Conclusion

Linux Kernel Support

Versions	3.14 – good 2.6.33, 3.x, 4.6, 4.11, 4.16 – partial
Architectures	x86 and x86_64 – full ARM – experimental
Configurations	*allmodconfig**

*As well as all generic enough configurations of the vanilla Linux kernel for building target parts as loadable kernel modules

Code Coverage*

Files	Line coverage	Function coverage
source files	**51%** (1616932/3154667)	**39%** (54862/138611)
drivers	**54%** (1318331/2441335)	**43%** (46188/106109)
infiniband	**36%** (39143/106934)	**26%** (1185/4449)
media	**48%** (114161/235853)	**28%** (2609/9143)
net	**67%** (382841/564959)	**62%** (14627/23512)
scsi	**67%** (140040/206180)	**55%** (4642/8421)
staging	**62%** (129055/204928)	**52%** (3766/7123)
usb	**53%** (85123/158925)	**48%** (3202/6608)
fs	**31%** (74624/235808)	**21%** (2287/10488)
net	**31%** (57164/180487)	**29%** (2906/9766)
sound	**57%** (151810/262443)	**29%** (3029/10419)

*Covered and total numbers of lines of code and functions are given just for successfully verified loadable kernel modules of Linux 3.14, x86_64, *allmodconfig*. Empty lines and preprocessor directives are not treated. Directories with less than 100 KLOC are omitted.

False Alarm Rate*

Type of requirements	Faults	False alarms
Incorrect usage of the most popular Linux kernel API	110 (26%)	309 (74%)
Memory safety	28 (8%)	337 (92%)
Data races	53 (60%)	36 (40%)

*Numbers are for successfully verified loadable kernel modules of Linux 3.14, x86_64, *allmodconfig*

False Alarm Reasons*

Type of requirements	Inaccurate specifications	Inaccurate software model checkers
Incorrect usage of the most popular Linux kernel API	69%	31%
Memory safety	85%	15%
Data races	58%	42%

*Numbers are for successfully verified loadable kernel modules of Linux 3.14, x86_64, *allmodconfig*

Verification Time*

Type of requirements	CPU time, days	Wall time, days**
Incorrect usage of the most popular Linux kernel API	68.7	1
Memory safety	14.4	0.2
Data races	3.2	0.2

*Numbers are for all verified loadable kernel modules of Linux 3.14, x86_64, *allmodconfig*
**In case of using powerful enough cluster/cloud

Detecting Hard-to-Find Faults

- As proof of concept and thanks to several GSoC projects 350+ faults found by Klever in the Linux kernel were reported and acknowledged by developers
  - almost all faults are in device drivers
  - most faults are on error handling paths
  - many faults require complex interactions between device drivers and the kernel
- Much more faults can be detected
  - perform systematic analysis of verification results and report found faults
  - increase code coverage
  - reduce false alarm rate
  - check for additional requirements

  } Primarily through fixing existing specifications and development of new ones

  - fix and optimize automatic verification workflow

Directions for further development of Klever*

- Adaptability and usability
  - automatic deployment
  - user interface
  - documentation
- Verification workflow improvements
- Tasks specific for verification of the Linux kernel
  - basic support of various Linux kernel architectures, configurations and versions
  - fixing existing specifications and development of new specifications
  - verification of all Linux kernel device drivers and subsystems against all specified requirements
- Tasks specific for verification of other software

*Complete list of high-level tasks

Outline

- Why is it worth to apply formal methods for the Linux kernel?
- Klever: Enabling model checking for the Linux kernel
- Current achievements and possible directions for further development of Klever
- Demonstration of Klever
- Conclusion

Access on demand

- If somebody wants to play with Klever without deploying it, please, push a demand to me (mailto:novikov@ispras.ru) with the following data
  - Name
  - Organization
  - Time slot
- We will provide an individual access to Klever on dedicated virtual machines to test verification of all loadable kernel modules of Linux
  - version: 3.14
  - architecture: x86_64
  - configuration: *allmodconfig*

Outline

- Why is it worth to apply formal methods for the Linux kernel?
- Klever: Enabling model checking for the Linux kernel
- Current achievements and directions for further development of Klever
- Demonstration of Klever
- Conclusion

Conclusion

- Formal methods can be used for
  - detecting hard-to-find faults
  - proving formal correctness of programs checked against particular requirements under certain assumptions
- Klever software verification framework
  - enables model checking for big critical industrial programs like the Linux kernel
  - is ready at a good level
  - is gradually improved because of needs in thorough verification for various projects
- Klever adaptations for the Linux kernel, BusyBox and Zephyr
  - demonstrate modest verification results
  - have a huge potential in case of additional support