Verifying Verified Code

Prof. Arie Gurfinkel
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

virtual presentation at VSTTE 2023

joint work with S. Priya, Y. Su, Y. Bao, X. Zhou, and Y. Vizel

The Team

Siddharth Priya

University of Waterloo

Xiang Zhou

University of Waterloo, now Intel

Yusen Su

University of Waterloo

Prof. Yuyan Bao

University of Waterloo, now Augusta University

Prof. Yakir Vizel

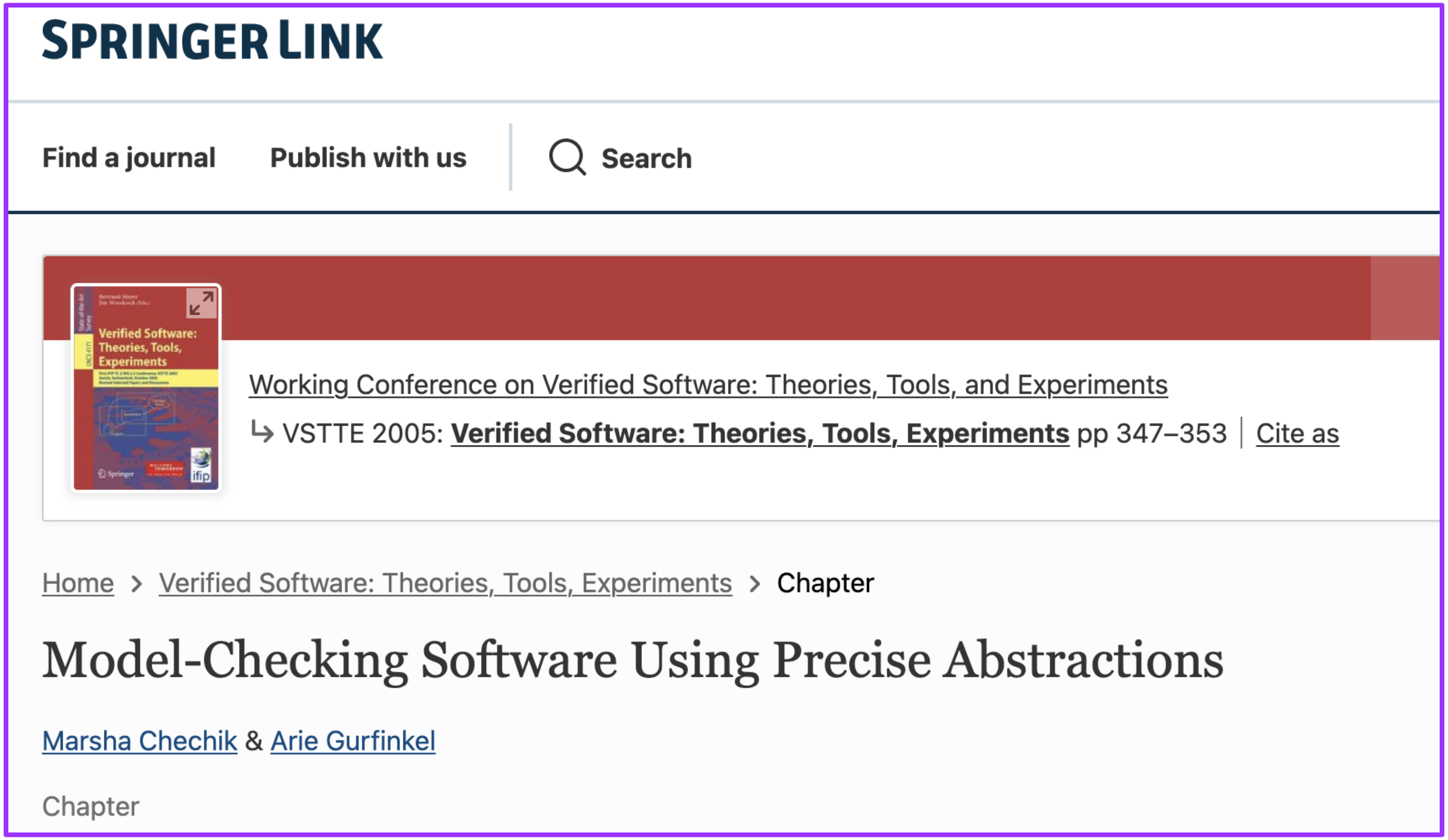
The Technion

Prof. Arie Gurfinkel

University of Waterloo

VSTTE 2005

VSTTE 2005



Working Conference on Verified Software: Theories, Tools, and Experiments

↳ VSTTE 2005: **Verified Software: Theories, Tools, Experiments** pp 347–353 | Cite as

Home › Verified Software: Theories, Tools, Experiments › Chapter

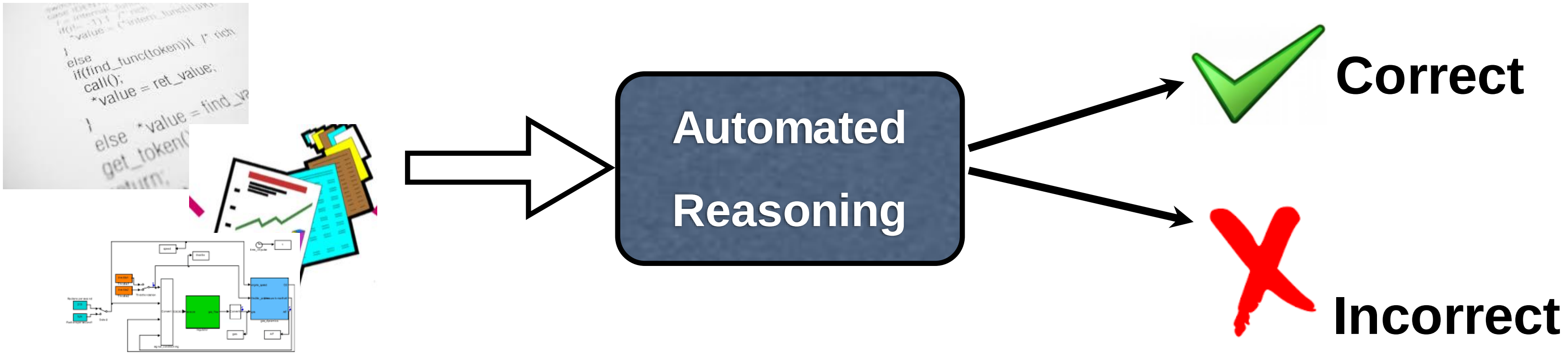
Model-Checking Software Using Precise Abstractions

Marsha Chechik & Arie Gurfinkel

Chapter

Automated (Software) Verification

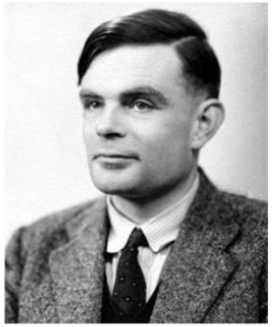
Program and/or model



Automated Reasoning

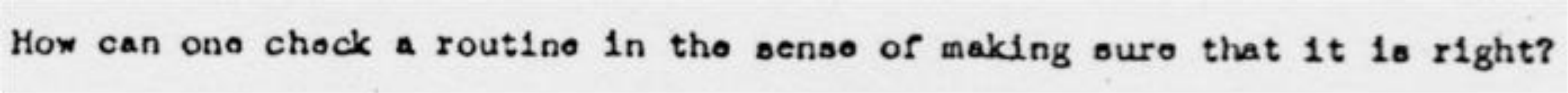
Correct

Incorrect

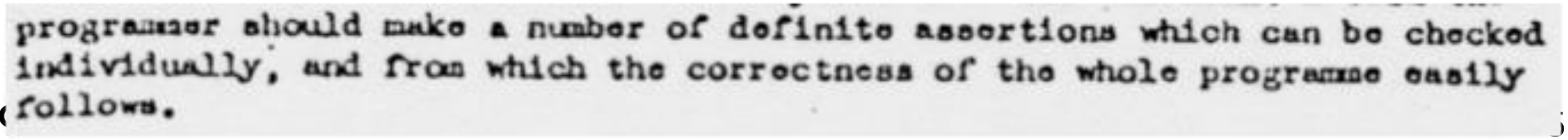


Alan M. Turing. 1936: “Undecidable”

Alan M. Turing. ”Checking a large routine” 1949



How can one check a routine in the sense of making sure that it is right?



programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

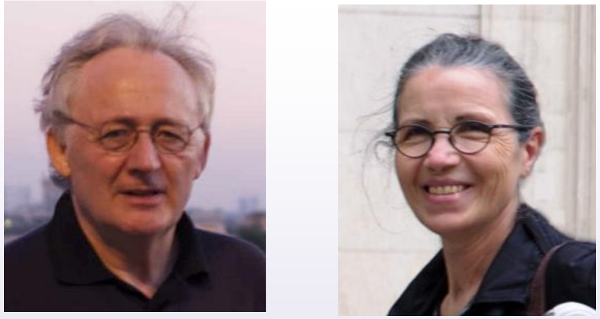
Automated Software Analysis

Model Checking

[Clarke and Emerson, 1981]

[Queille and Sifakis, 1982]

Abstract Interpretation



[Cousot and Cousot, 1977]

Symbolic Execution

[King, 1976]

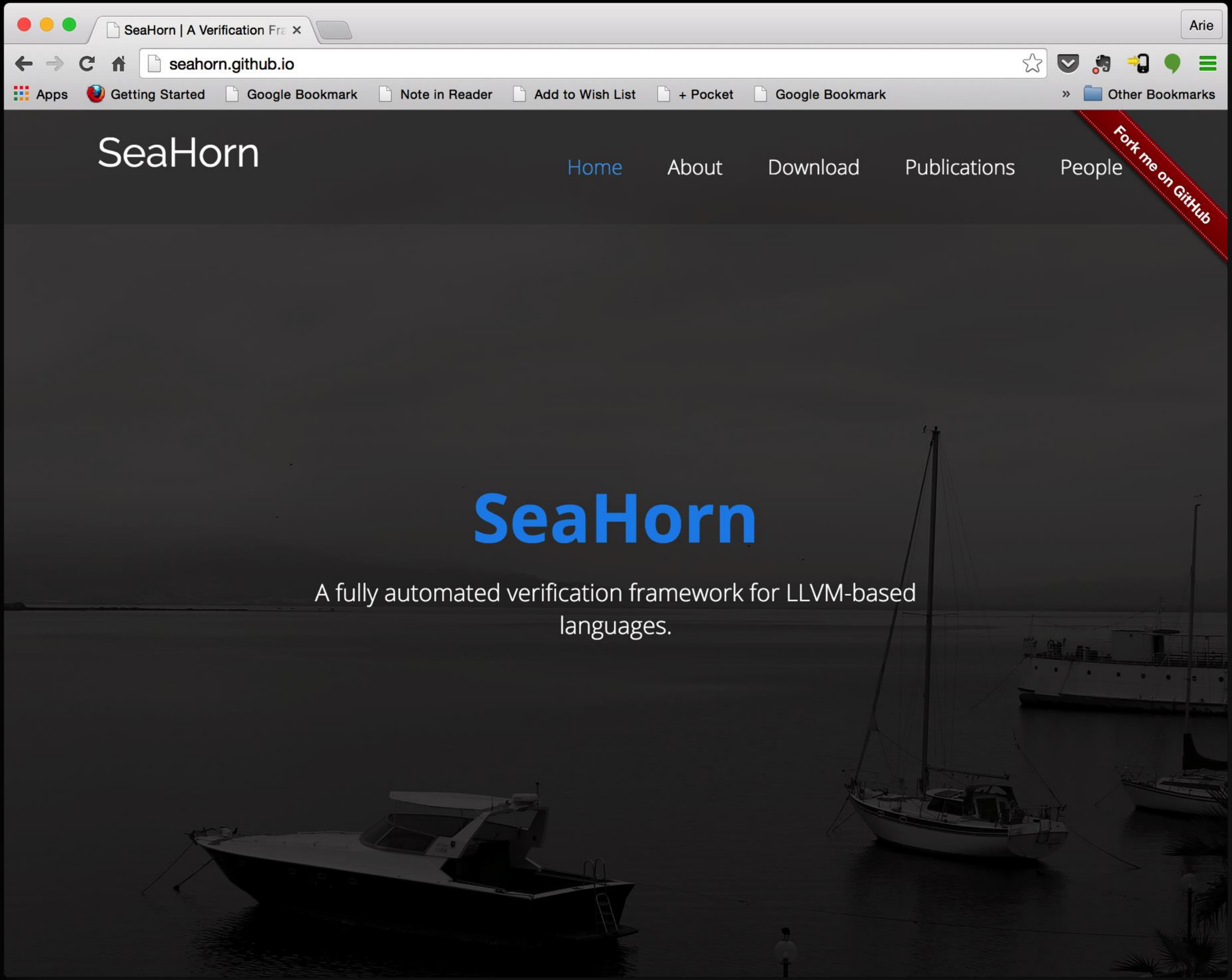
Automated Verification

Deductive Verification

- A user provides a program and a verification certificate
  - e.g., inductive invariant, pre- and post-conditions, function summaries, etc.
- A tool automatically checks validity of the certificate
  - this is not easy! (might even be undecidable)
- Verification is manual but machine certified

Algorithmic Verification

- A user provides a program and a desired specification
  - e.g., program never writes outside of allocated memory
- A tool automatically checks validity of the specification
  - and generates a verification certificate if the program is correct
  - and generates a counterexample if the program is not correct
- Verification is completely automatic – “push-button”

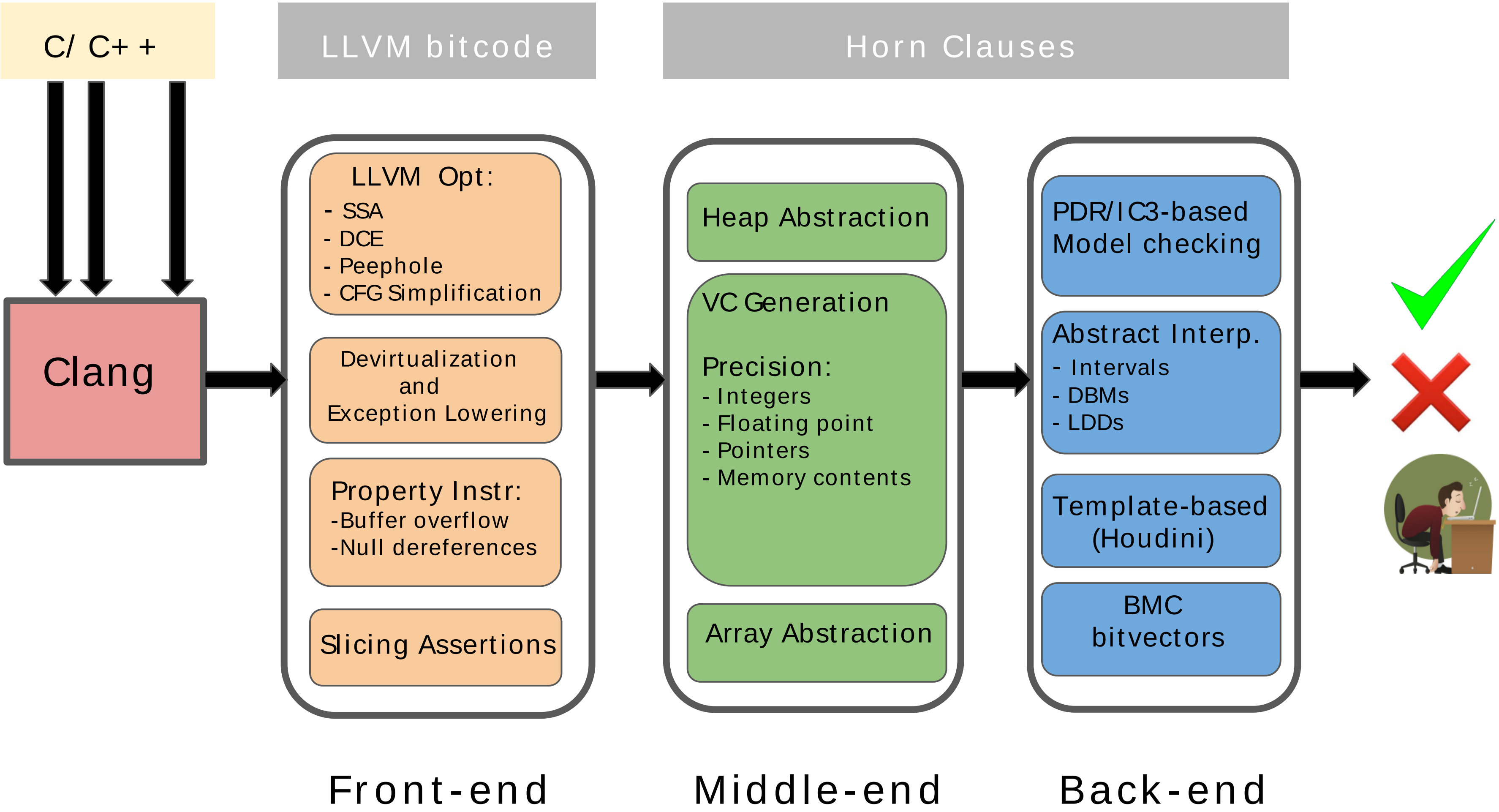


SeaHorn

A fully automated verification framework for LLVM-based languages.

http://seahorn.github.io

Architecture of Seahorn



C/ C++

LLVM bitcode

Horn Clauses

Clang

LLVM Opt:
- SSA
- DCE
- Peephole
- CFG Simplification

Devirtualization and Exception Lowering

Property Instr:
- Buffer overflow
- Null dereferences

Slicing Assertions

Heap Abstraction

VC Generation

Precision:
- Integers
- Floating point
- Pointers
- Memory contents

Array Abstraction

PDR/IC3-based Model checking

Abstract Interp.
- Intervals
- DBMs
- LDDs

Template-based (Houdini)

BMC bitvectors

Front-end

Middle-end

Back-end

Bounded Model Checking (BMC)

BMC: is a precise static analysis (i.e., verification) technique

- reduce verification to constraint solving with SAT- and SMT-solvers

Pros

- precision, including path sensitivity, machine arithmetic, bit-vector operations, etc.
- ease of use – everything can be modeled in code

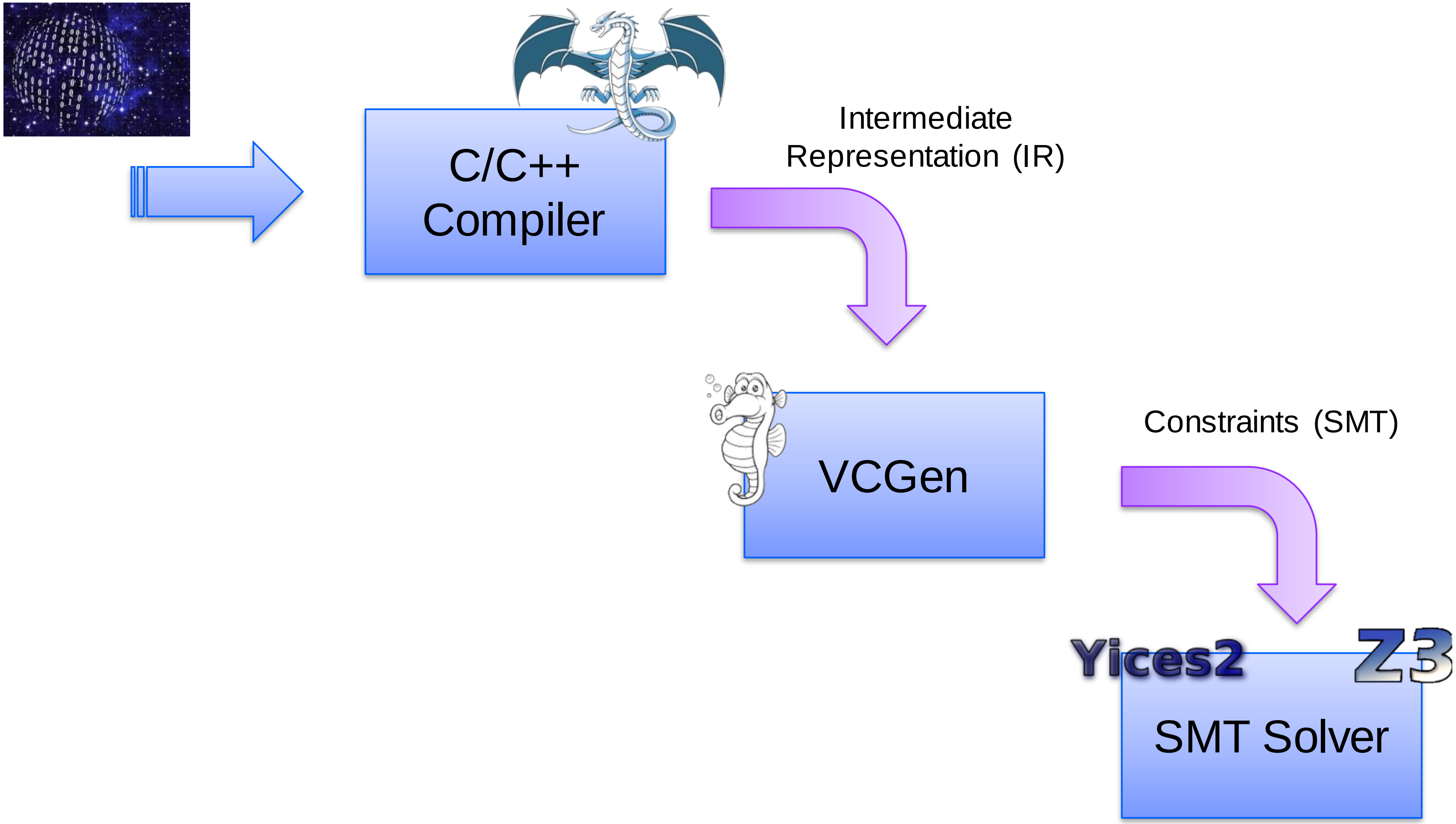
Cons

- scalability (scales to thousands LOC, but not millions)
- requires “unit proofs” and ”mocks” to be effective

Well suited for security properties

- spatial memory safety, information flow, side-channels

Backend: Verification Condition Generation



C/C++ Compiler

Intermediate Representation (IR)

VCGen

Constraints (SMT)

SMT Solver

SOTA Competition

CBMC – Bounded Model Checker for C

- started at CMU and Oxford, now supported by diffblue
- oldest, mature, actively used in industry (Amazon)
- custom C parser, some semantic particularities

KLEE – Symbolic Execution for LLVM

- mature, actively used in academic community
- de-facto symbolic execution engine in LLVM
- unlike BMC, targets bug finding rather than verification

SMACK

- open-sourced BMC engine for LLVM
- uses some components from SeaHorn

SYMBIOTIC

- combines KLEE with slicing for scalability
- winner of multiple SVCOMP competitions

SeaBMC: BMC for LLVM

SeaHorn-based Open-sourced BMC engine for LLVM

- bit-precise, byte-precise, path-sensitive

Supports many different encodings of verification conditions

- different encodings are better for different SMT solvers
- different encodings are better for different properties

Supports verification-specific extension to computer architecture

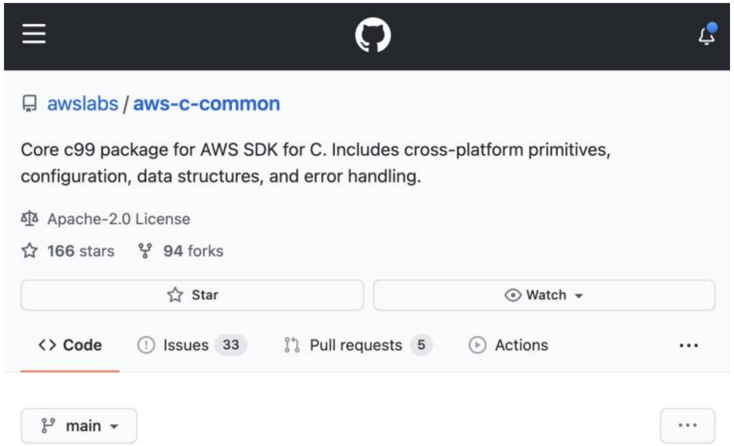
- store pointer-specific information directly with a pointer (i.e., fat-pointer)
- store memory object specific information directly with the memory object (i.e., shadow memory)
- extensions are done at the semantic level and exposed to developer via simple API

Case Study: aws-c-common library

Core C99 package for AWS SDK

- cross-platform primitives
- configuration
- data structures
- error handling

Self-contained



awslabs / aws-c-common

Core c99 package for AWS SDK for C. Includes cross-platform primitives, configuration, data structures, and error handling.

Apache-2.0 License

166 stars 94 forks

Low-level and platform specific C

Extensively verified using CBMC

- >160 unit proofs
- verify memory safety, representation invariants, basic operations

UNIVERSITY OF WATERLOO

https://github.com/awslabs/aws-c-common

Code as Spec (CaS): A Unit Proof

```
int main() {
/* data structure */
struct aws_array_list list;
initialize_bounded_array_list(&list);
/* assumptions */
assume(aws_array_list_is_valid(&list));
assume(list.item_size > 0);
...
/* perform operation under verification */
size_t capacity = aws_array_list_capacity(&list);
/* assertions */
assert(aws_array_list_is_valid(&list));
assert(capacity == list.current_size /
  list.item_size);
...
return 0;
}
```

→ initialization

→ pre-condition

→ function to be verified

→ post-condition

Code-as-Spec (CaS) features

Use code to write pre-and-post conditions

- empower developers to write and maintain specifications
- share specifications between multiple tools and techniques
- structure verification effort around unit proofs

A unit proof (like unit test)

- sets the environment for verification (pre-condition)
- calls function under verification
- validates the result (post-condition)

Extend programming language with specification primitives

- non-deterministic (i.e., symbolic) input
- `verifier.assume()` built-in to specify desired pre-condition
- `verifier.assert()` built-in to specify statically checked assertions

https://github.com/seahorn/verify-c-common

Research Questions

RQ1: Does CaS empower multiple (analysis) tools?

- **Yes** – we use BMC, Symbolic Execution, and Fuzzing all at once
- **But** – semantics, semantics, semantics
- **And** – need to design specs with multiple tools in mind

RQ2: Are there bugs in verified code?

- **Yes (and No)**
  - found serious bugs in specifications (but they did not hide bugs in code)
  - found (potential) bugs that might manifest in the future

RQ3: Can CaS specs be improved?

- **Yes**
  - writing effective specifications is challenging, no matter what the language
  - need built-ins specific for verification to directly express required concepts
    - size of referenced memory, modifiability of memory region, etc.

UNIVERSITY OF
WATERLOO

RQ1: Semantics is important!

CBMC

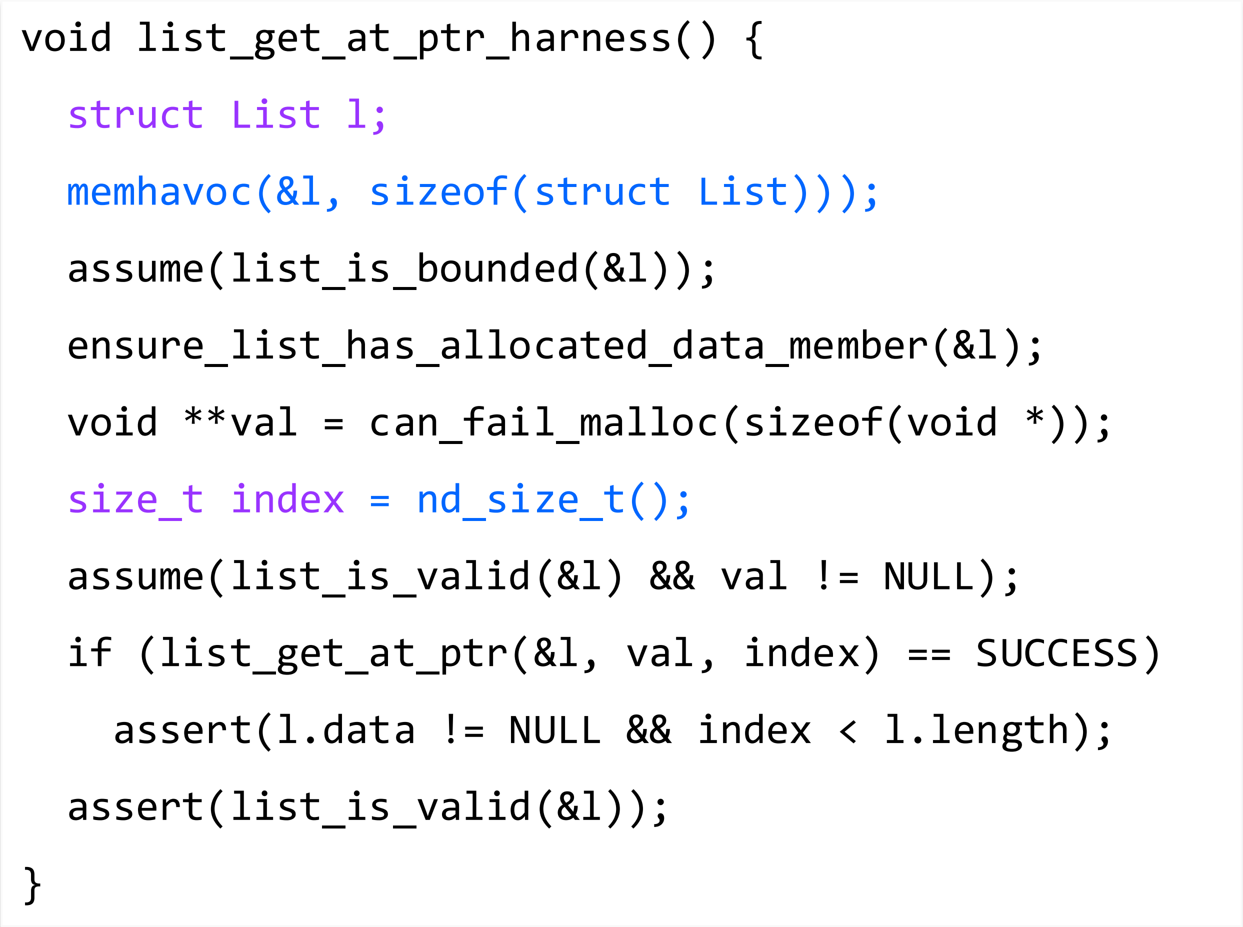
```
void list_get_at_ptr_harness() {
  struct List l;

  assume(list_is_bounded(&l));
  ensure_list_has_allocated_data_member(&l);
  void **val = can_fail_malloc(sizeof(void *));
  size_t index;
  assume(list_is_valid(&l) && val != NULL);
  if (list_get_at_ptr(&l, val, index) == SUCCESS)
    assert(l.data != NULL && index < l.length);
  assert(list_is_valid(&l));
}
```

Undefined behavior (uninitialized variables) are resolved based on the internal semantics of CBMC

RQ1: Semantics is important!

SEAHORN



```
void list_get_at_ptr_harness() {
  struct List l;
  memhavoc(&l, sizeof(struct List)));
  assume(list_is_bounded(&l));
  ensure_list_has_allocated_data_member(&l);
  void **val = can_fail_malloc(sizeof(void *));
  size_t index = nd_size_t();
  assume(list_is_valid(&l) && val != NULL);
  if (list_get_at_ptr(&l, val, index) == SUCCESS)
    assert(l.data != NULL && index < l.length);
  assert(list_is_valid(&l));
}
```

No undefined behaviour. Semantics depend on implementation of the explicit initialization function

RQ1: Does CaS empower multiple (analysis) tools?

Different tools require somewhat different specification of assumptions

- refactor to have different implementation of specs for each style of tools

SeaHorn

```
size_t len = nd_size_t();
size_t cap = nd_size_t();
assume(len <= cap);
assume(cap <= MAX_BUFFER);

buf->len = len;
buf->capacity = cap;
buf->buffer = can_fail_malloc(
    cap * sizeof(*(buf->buffer)));
buf->allocator = sea_allocator();
```

libFuzzer

```
size_t len = nd_size_t();
size_t cap = nd_size_t();
cap %= MAX_BUFFER;
len = (cap == 0) ? 0 : len % cap;

buf->len = len;
buf->capacity = cap;
buf->buffer = can_fail_malloc(
    cap * sizeof(*(buf->buffer)));
buf->allocator = sea_allocator();
```

RQ1: Does CaS empower multiple (analysis) tools?

Different tools require somewhat different specification of assumptions

- refactor to have different implementation of specs for each style of tools

SEAHORN

```
size_t len = nd_size_t();
size_t cap = nd_size_t();
assume(len <= cap);
assume(cap <= MAX_BUFFER);

buf->len = len;
buf->capacity = cap;
buf->buffer = can_fail_malloc(
    cap * sizeof(*(buf->buffer)));
buf->allocator = sea_allocator();
```

libFuzzer

```
size_t len = nd_size_t();
size_t cap = nd_size_t();
cap %= MAX_BUFFER;
len = (cap == 0) ? 0 : len % cap;

buf->len = len;
buf->capacity = cap;
buf->buffer = can_fail_malloc(
    cap * sizeof(*(buf->buffer)));

buf->allocator = sea_allocator();
```

RQ2: Are there bugs in verified code?

Found bugs in representation invariant

- representation invariant defines basic properties of a data structure
- assumed to be true before any function under verification
- checked that it is maintained at every call

Bug was subtle enough to be preserved by each function

- could hide real bugs in real code (but did not in this case)

```
bool aws_byte_buf_is_valid(const struct aws_byte_buf *const buf) {
  return buf != NULL &&
  ((buf->capacity == 0 && buf->len == 0 && buf->buffer == NULL) ||
  (buf->capacity > 0 && buf->len <= buf->capacity &&
  AWS_MEM_IS_WRITABLE(buf->buffer, buf->len)));
}
```

RQ2: Are there bugs in verified code?

Found bugs in representation invariant

- representation invariant defines basic properties of a data structure
- assumed to be true before any function under verification
- checked that it is maintained at every call

Bug was subtle enough to be preserved by each function

- could hide real bugs in real code (but did not in this case)

```
bool aws_byte_buf_is_valid(const struct aws_byte_buf *const buf) {
  return buf != NULL &&
  ((buf->capacity == 0 && buf->len == 0 && buf->buffer == NULL) ||
  (buf->capacity > 0 && buf->len <= buf->capacity &&
  AWS_MEM_IS_WRITABLE(buf->buffer, buf->capacity)));
}
```

Vacuity in CaS

Vacuity is a known sanity check in temporal model checking

- also known as *antecedent failure*
- a property is satisfied *vacuously* if
  - it is true in the model
  - a much stronger property is true
- e.g., **always if p then q** is true, but also **always not p** is true

In CaS, properties are not specified in a specialized language

- properties are embedded in code, they are part of code
- What is vacuity in this case?

Our definition: **sassert** is satisfied vacuously iff it is never reachable

- e.g., `if (c) { sassert(0); }`

RQ2: (potential) bug due to UB: Bug

```
AWS_STATIC_IMPL
bool aws_is_mem_zeroed(const void *buf, size_t bufsize) {
  const uint64_t *buf_u64 = (const uint64_t *)buf;
  const size_t num_u64_checks = bufsize / 8;
  size_t i;

  for (i = 0; i < num_u64_checks; ++i) {

    if (buf_u64[i]) {
      return false;
    }
  }
  ...
}
```

Found (potential) bug due to UB: Fix

```
AWS_STATIC_IMPL
bool aws_is_mem_zeroed(const void *buf, size_t bufsize) {
  const uint64_t *buf_u64 = (const uint64_t *)buf;
  const size_t num_u64_checks = bufsize / 8;
  size_t i;
  uint64_t val;
  for (i = 0; i < num_u64_checks; ++i) {
    memcpy(&val, &buf_u64[i], sizeof(val));
    if (val) {
      return false;
    }
  }
  ...
}
```

RQ3: Can CaS specs be improved?

Writing specifications is no different than writing code

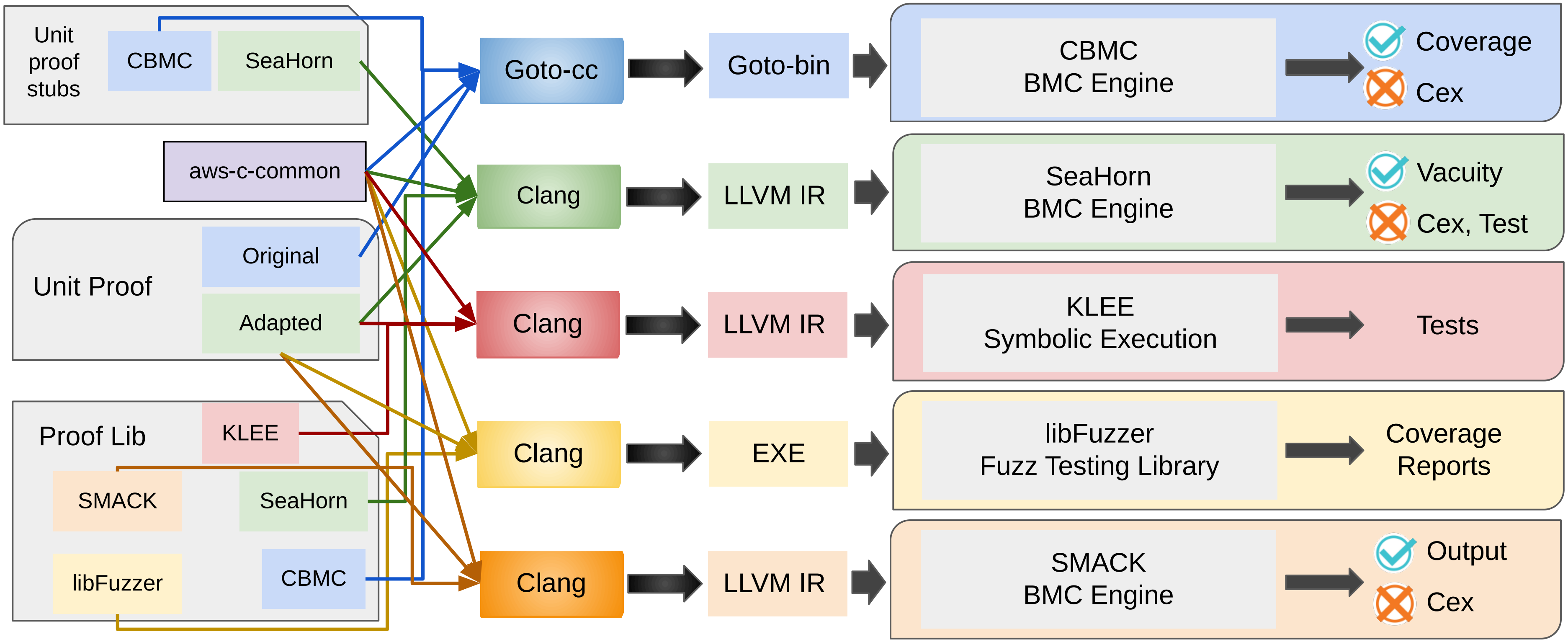
- there is good code, there is ok code, there is bad code
- sometimes, there is better code
- e.g., we improve specification of linked list data structure to verify faster while checking stronger properties

However, need additional built-in functions to communicate intention in the specifications

- `is_deref(p, sz)` – true if pointer p points to at least sz accessible bytes
- `is_mod(p)` – true if object pointed to by pointer p has been modified recently
- `is_alloc(p)` – true if object pointed to be p is (still) allocated
- …

Reusing CaS between tools requires standard for built-ins!

Case Study Architecture



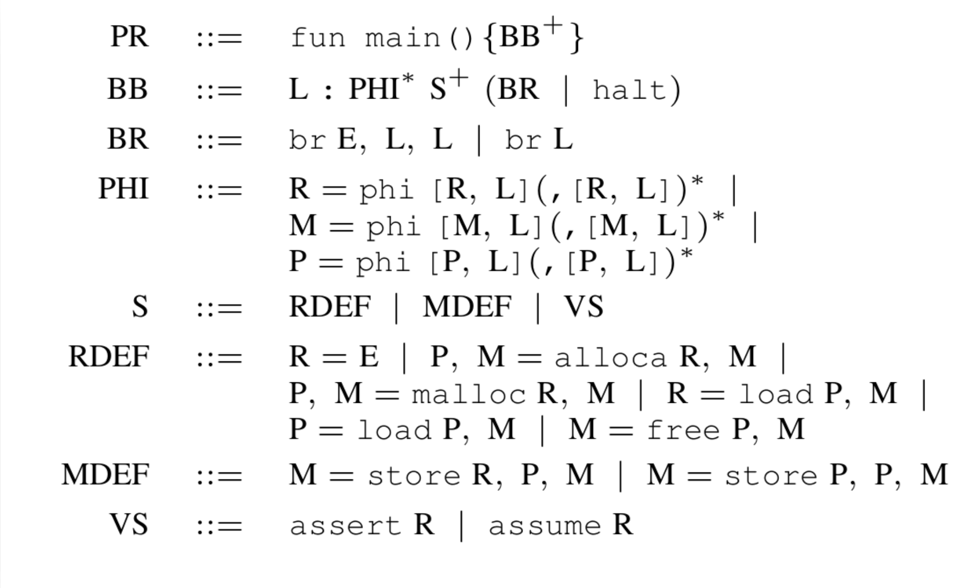
https://github.com/seahorn/verify-c-common

UNIVERSITY OF WATERLOO

Bounded Model Checker for LLVM (C, C++, Rust …)

SEABMC

SEA-IR – purify memory operations

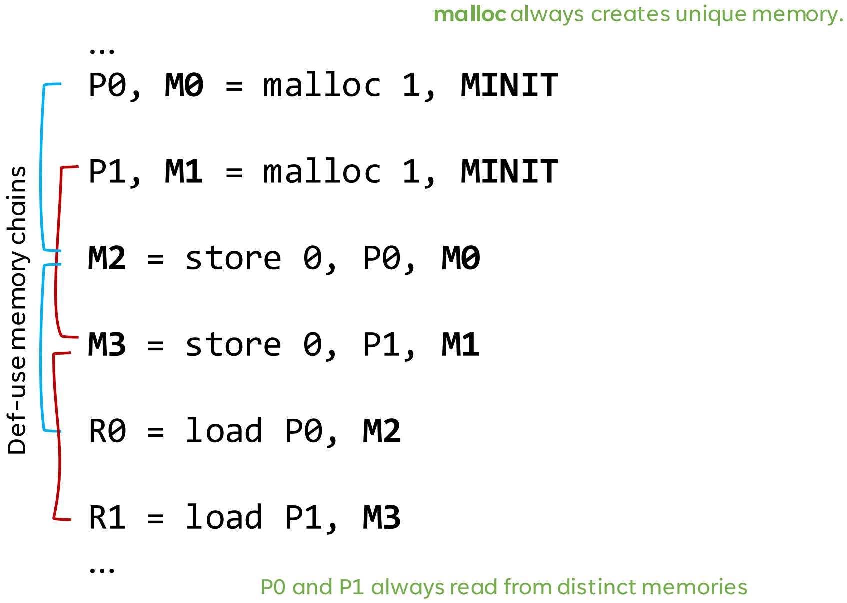


$$
\begin{array}{rcl}
\mathrm{PR} & ::= & \texttt{fun main()}\{\mathrm{BB}^{+}\} \\
\mathrm{BB} & ::= & \mathrm{L} : \mathrm{PHI}^{*}\ \mathrm{S}^{+}\ (\mathrm{BR} \mid \texttt{halt}) \\
\mathrm{BR} & ::= & \texttt{br}\ \mathrm{E},\ \mathrm{L},\ \mathrm{L} \mid \texttt{br}\ \mathrm{L} \\
\mathrm{PHI} & ::= & \mathrm{R} = \texttt{phi}\ \texttt{[}\mathrm{R},\ \mathrm{L}\texttt{]}(\texttt{,}\ \texttt{[}\mathrm{R},\ \mathrm{L}\texttt{]})^{*} \mid \\
 & & \mathrm{M} = \texttt{phi}\ \texttt{[}\mathrm{M},\ \mathrm{L}\texttt{]}(\texttt{,}\ \texttt{[}\mathrm{M},\ \mathrm{L}\texttt{]})^{*} \mid \\
 & & \mathrm{P} = \texttt{phi}\ \texttt{[}\mathrm{P},\ \mathrm{L}\texttt{]}(\texttt{,}\ \texttt{[}\mathrm{P},\ \mathrm{L}\texttt{]})^{*} \\
\mathrm{S} & ::= & \mathrm{RDEF} \mid \mathrm{MDEF} \mid \mathrm{VS} \\
\mathrm{RDEF} & ::= & \mathrm{R} = \mathrm{E} \mid \mathrm{P},\ \mathrm{M} = \texttt{alloca}\ \mathrm{R},\ \mathrm{M} \mid \\
 & & \mathrm{P},\ \mathrm{M} = \texttt{malloc}\ \mathrm{R},\ \mathrm{M} \mid \mathrm{R} = \texttt{load}\ \mathrm{P},\ \mathrm{M} \mid \\
 & & \mathrm{P} = \texttt{load}\ \mathrm{P},\ \mathrm{M} \mid \mathrm{M} = \texttt{free}\ \mathrm{P},\ \mathrm{M} \\
\mathrm{MDEF} & ::= & \mathrm{M} = \texttt{store}\ \mathrm{R},\ \mathrm{P},\ \mathrm{M} \mid \mathrm{M} = \texttt{store}\ \mathrm{P},\ \mathrm{P},\ \mathrm{M} \\
\mathrm{VS} & ::= & \texttt{assert}\ \mathrm{R} \mid \texttt{assume}\ \mathrm{R}
\end{array}
$$

Unlimited registers: Each register has a type – scalar, pointer, or memory

All operations are pure: **SEA-IR** extends LLVM IR by making dependency information between memory operations explicit

SEA-IR – purify memory operations



malloc always creates unique memory.

```
…
P0, M0 = malloc 1, MINIT

P1, M1 = malloc 1, MINIT

M2 = store 0, P0, M0

M3 = store 0, P1, M1

R0 = load P0, M2

R1 = load P1, M3
…
```

P0 and P1 always read from distinct memories

Example: SEA-IR program with pure memory operations

- **Blue** and **Red** are distinct def-use memory chains
- This distinction helps generate simpler VC

SEA-IR: Program transformation

Source prog.

```
int main() {
  int s = nd_int();
  assume(s > -5);
  if (s > 0) {
    s = s - nd_int();
  }
  assert(s > -5);
  return 0;
}
```

C program: **nd_int** returns a non-deterministic int; **assume** and **assert** have usual meanings

SA prog.

```
define main() {
BB0:
  R0 = nd_int()
  R1 = R0 > -5
  assume R1
  R2 = R0 > 0
  br R2, BB1, BB2
BB1:
  R3 = nd_int()
  R4 = R0 - R3
  br BB2
BB2:
  PHINODE = phi [R4, BB1], [R0, BB0]
  R5 = PHINODE > -5
  assume(!R5)
  assert false
  halt
}
```

SA program: SEA-IR program in control flow form with **phi** nodes. It has a single assert (SA).

GSA prog.

```
define main() {
BB0:
  R0 = nd_int()
  R1 = R0 > -5
  R2 = R0 > 0
  br R2, BB1, BB2
BB1:
  R3 = nd_int()
  R4 = R0 - R3
  br BB2
BB2:
  GAMMA = select R2, R4, R0
  R5 = GAMMA > -5
  R6 = !R5
  R7 = R1 && R6
  assume R7
  assert false
  halt
}
```

GSA program: SEA-IR program in gated SSA form (**GSA**). It has a single assume and a single assert (**SASA**).

VC

```
(r4 = r0 - r3) &&
(r2 = r0 > 0)
(gamma = ite(r2, r4, r0)) &&
(gamma > -5)
(r6 = !r5) &&
(r1 = r0 > -5) &&
(r7 = r1 && r6) &&
r7 &&
!false
```

VCGen from **GSA** program using pure dataflow analysis.

VC generation can happen from different SEA-IR forms – control flow or dataflow.

UNIVERSITY OF WATERLOO

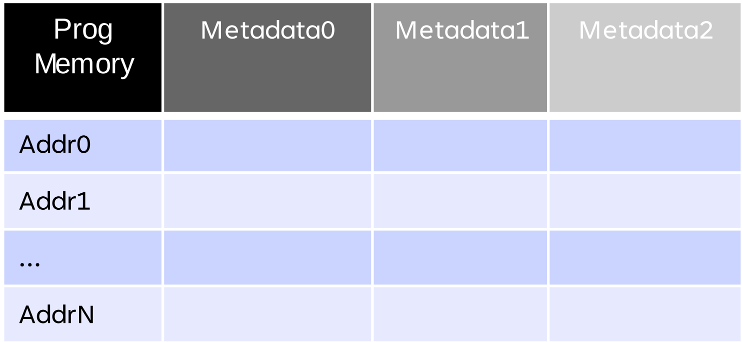
Shadow memory and fat pointers

Shadow every byte (or word)
of program memory with program state metadata

- Memcheck – addressable, initialized memory?
- Eraser – concurrent access follows locking discipline

  Recent CBMC-SSM extension has shadow memory for CBMC

  - CBMC-SSM: Bounded Model Checking of C Programs with Symbolic Shadow Memory, ASE 2022, Bernd Fischer, Salvatore La Torre, Gennaro Parlato, Peter Schrammel



Prog Memory	Metadata0	Metadata1	Metadata2
Addr0			
Addr1			
…			
AddrN			

Shadow mem representation

Address	Metadata0	Metadata1	Metadata2

Fat pointer representation

Some metadata can be "cached" at pointers instead of memory, saving memory accesses. This scheme is called Fat pointers.

Fat pointer application – detect OOB access

```
int main() {
  char *p = (char *) malloc(sizeof(char));
  *p = 255;
  *(p+8) = 255; ←
  return 0
}
```

OOB access;
Undefined behaviour

```
int main() {
  char *p = (char *) malloc(sizeof(char));
✓ sea_is_deref(p, 0);
  *p = 255;
✗ sea_is_deref(p, 8);
  *(p+8) = 255;
  return 0
}
```

```
sym(R1 = isderef P0 B) ==
    r1 = 0 <= p0.offset + B < p0.size
```

isderef semantics

Base Address	Offset	Size
p	0	1

Base Address	Offset	Size
p	8	1

Contrast with CBMC: CBMC overloads pointer bits to store metadata adding constraints on the addresses that can be modelled. Fat pointers have no such limitation!

Shadow memory application – detect UAF

```
int main() {
  char *p = (char *)malloc(sizeof(char));
  *p = 0;
  free(p);
  *p = 255;
  return 0
}
```

← UAF; Undefined behaviour

```
  int main() {
    char *p = (char *) malloc(sizeof(char));
✓   sea_is_alloc(p);
    *p = 0;
    free(p);
✗   sea_is_alloc(p);
    *p = 255;
    return 0
  }
```

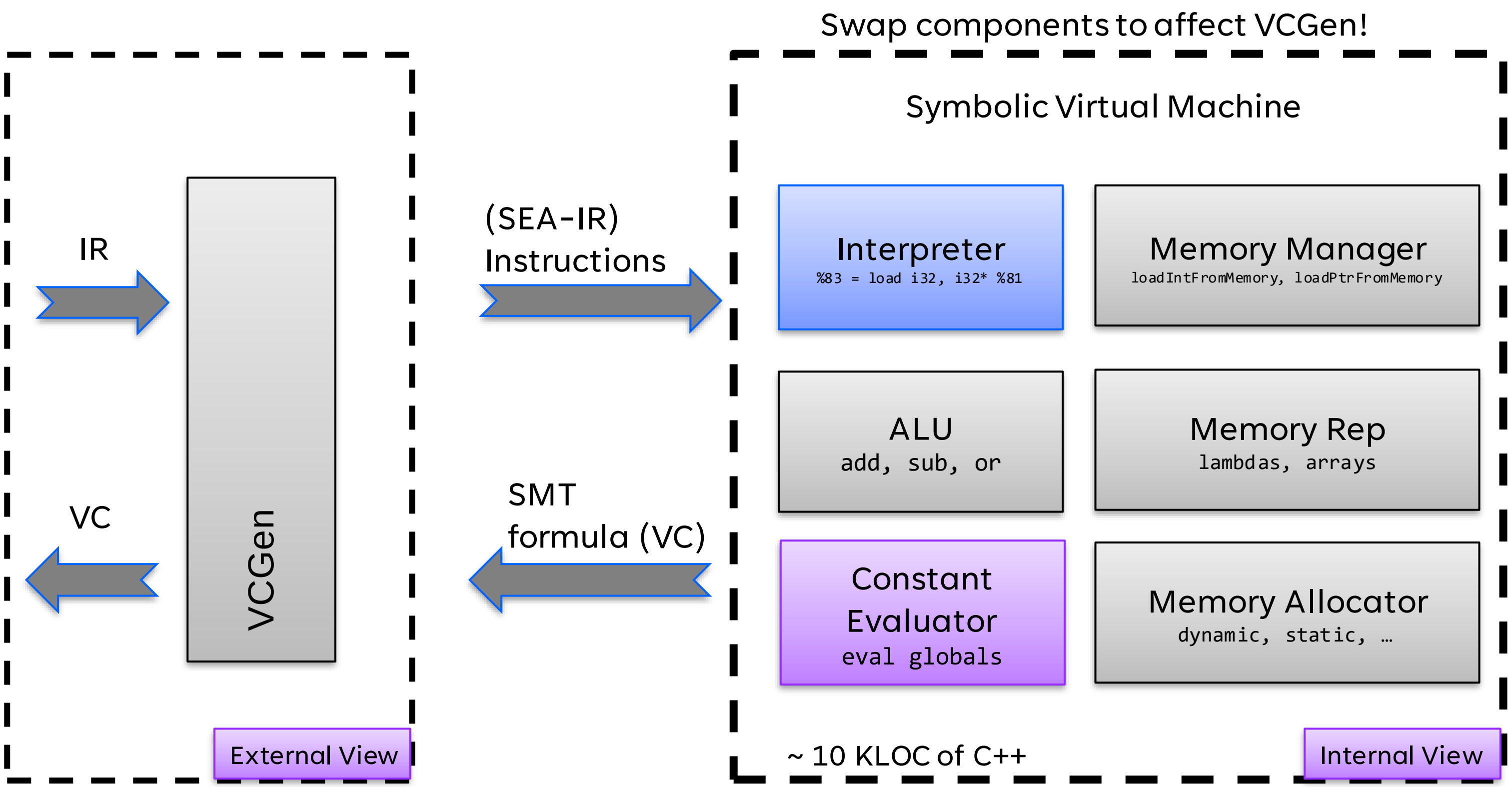
Intrinsic like `sea_is_alloc` operate on program metadata.

Note: This scheme relies on fat pointers that store base address.

Intrinsics to track other program properties – e.g., `sea_is_mod` (RO memory integrity)

Prog Memory	Base	Offset	Size	isAlloc
p	--	--	--	0 or 1

BACKEND: VCGEN as a (symbolic) VM



Swap components to affect VCGen!

IR

VC

VCGen

External View

(SEA-IR) Instructions

SMT formula (VC)

Symbolic Virtual Machine

Interpreter
`%83 = load i32, i32* %81`

Memory Manager
`loadIntFromMemory, loadPtrFromMemory`

ALU
`add, sub, or`

Memory Rep
`lambdas, arrays`

Constant Evaluator
`eval globals`

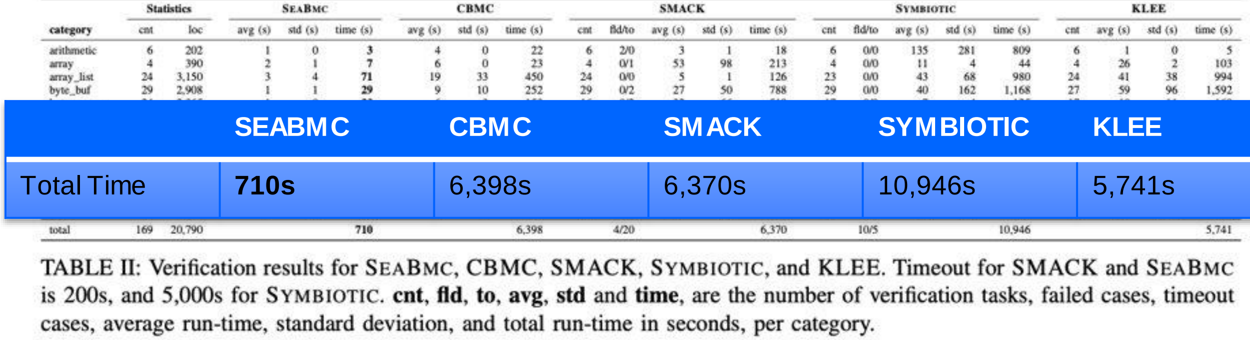
Memory Allocator
`dynamic, static, …`

~ 10 KLOC of C++

Internal View

aws-c-common benchmark verification time

Comparision with SeaBMC, CBMC, SMACK, SYMBIOTIC, KLEE



	Statistics		SeaBmc			CBMC			SMACK					Symbiotic					KLEE			
category	cnt	loc	avg (s)	std (s)	time (s)	avg (s)	std (s)	time (s)	cnt	fld/to	avg (s)	std (s)	time (s)	cnt	fld/to	avg (s)	std (s)	time (s)	cnt	avg (s)	std (s)	time (s)
arithmetic	6	202	1	0	**3**	4	0	22	6	2/0	3	1	18	6	0/0	135	281	809	6	1	0	5
array	4	390	2	1	**7**	6	0	23	4	0/1	53	98	213	4	0/0	11	4	44	4	26	2	103
array_list	24	3,150	3	4	**71**	19	33	450	24	0/0	5	1	126	23	0/0	43	68	980	24	41	38	994
byte_buf	29	2,908	1	1	**29**	9	10	252	29	0/2	27	50	788	29	0/0	40	162	1,168	27	59	96	1,592
total	169	20,790			**710**			6,398		4/20			6,370		10/5			10,946				5,741

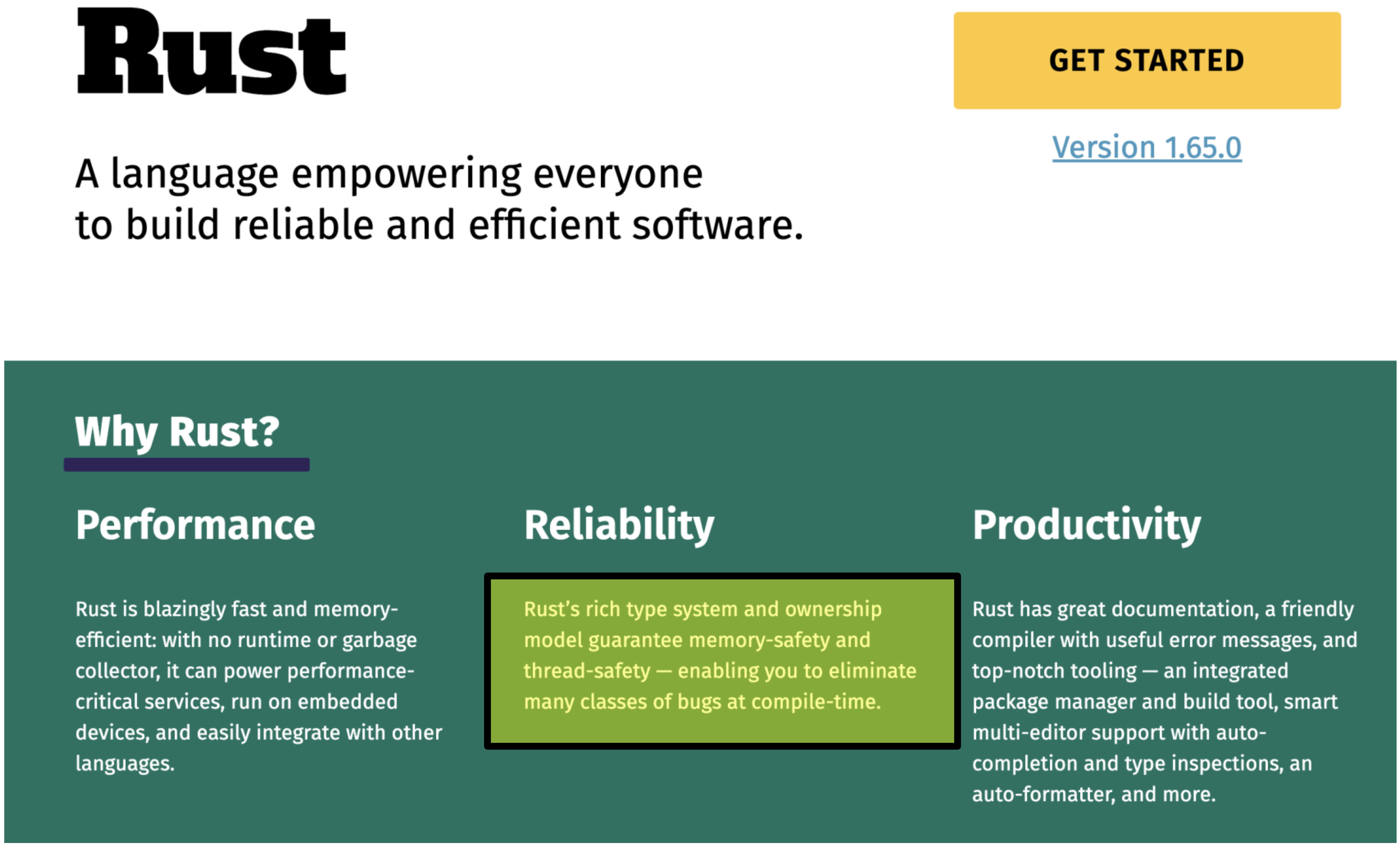
TABLE II: Verification results for SEABMC, CBMC, SMACK, SYMBIOTIC, and KLEE. Timeout for SMACK and SEABMC is 200s, and 5,000s for SYMBIOTIC. **cnt**, **fld**, **to**, **avg**, **std** and **time**, are the number of verification tasks, failed cases, timeout cases, average run-time, standard deviation, and total run-time in seconds, per category.

	SEABMC	CBMC	SMACK	SYMBIOTIC	KLEE
Total Time	**710s**	6,398s	6,370s	10,946s	5,741s

Read only memory proof using shadow memory (rewrite 70 proofs)

SEABMC config	Total time
Shadow	**90s**
No shadow	143s

https://github.com/seahorn/verify-c-common



Rust

GET STARTED

Version 1.65.0

A language empowering everyone
to build reliable and efficient software.

Why Rust?

Performance

Rust is blazingly fast and memory-efficient: with no runtime or garbage collector, it can power performance-critical services, run on embedded devices, and easily integrate with other languages.

Reliability

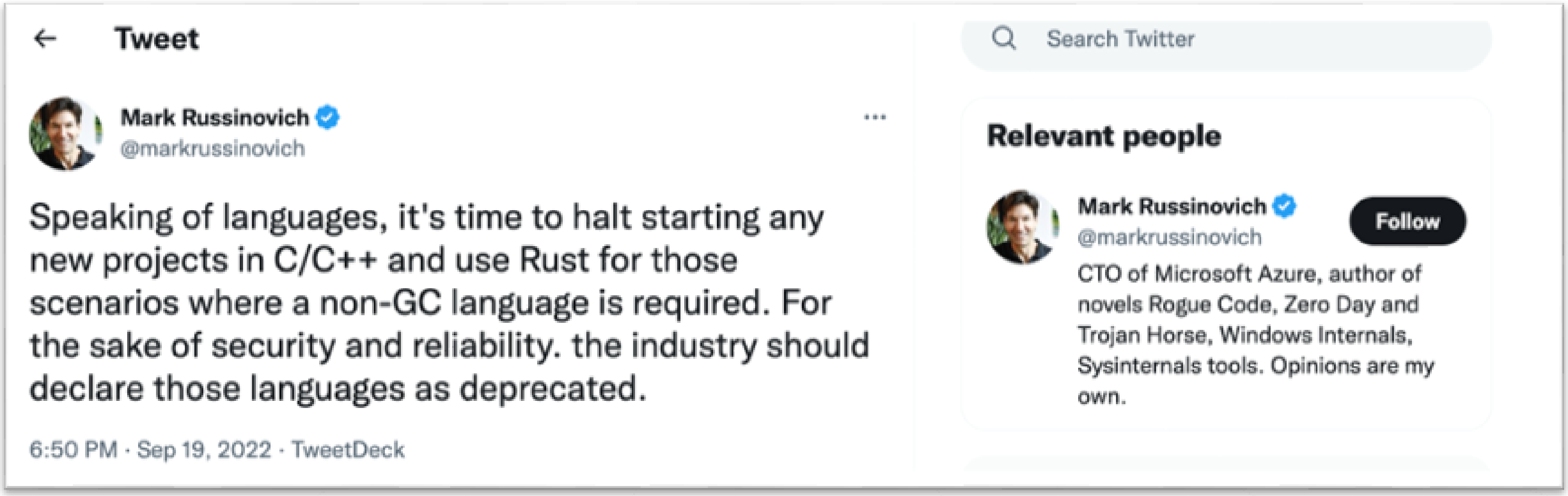
Rust's rich type system and ownership model guarantee memory-safety and thread-safety — enabling you to eliminate many classes of bugs at compile-time.

Productivity

Rust has great documentation, a friendly compiler with useful error messages, and top-notch tooling — an integrated package manager and build tool, smart multi-editor support with auto-completion and type inspections, an auto-formatter, and more.

https://www.rust-lang.org/

Rust makes memory problems obsolete…



Mark Russinovich
@markrussinovich

Speaking of languages, it's time to halt starting any new projects in C/C++ and use Rust for those scenarios where a non-GC language is required. For the sake of security and reliability. the industry should declare those languages as deprecated.

6:50 PM · Sep 19, 2022 · TweetDeck

Relevant people

Mark Russinovich
@markrussinovich

CTO of Microsoft Azure, author of novels Rogue Code, Zero Day and Trojan Horse, Windows Internals, Sysinternals tools. Opinions are my own.

Rust is mature enough to be used in Linux

Home / Innovation / Services & Software / Operating Systems / Linux

Linus Torvalds: Rust will go into Linux 6.1

At the Kernel Maintainers Summit, the question wasn't, "Would Rust make it into Linux?" Instead, it was, "What to do about its compilers?"

Written by **Steven Vaughan-Nichols,** Senior Contributing Editor

on Sept. 19, 2022

https://www.zdnet.com/article/linus-torvalds-rust-will-go-into-linux-6-1/

Do only safe Rust programs compile?

```rust
fn main() {
    let a = [1, 2, 3, 4, 5];
    let index = 10;
    let element = a[index];
    println!("{}", element);
}
```

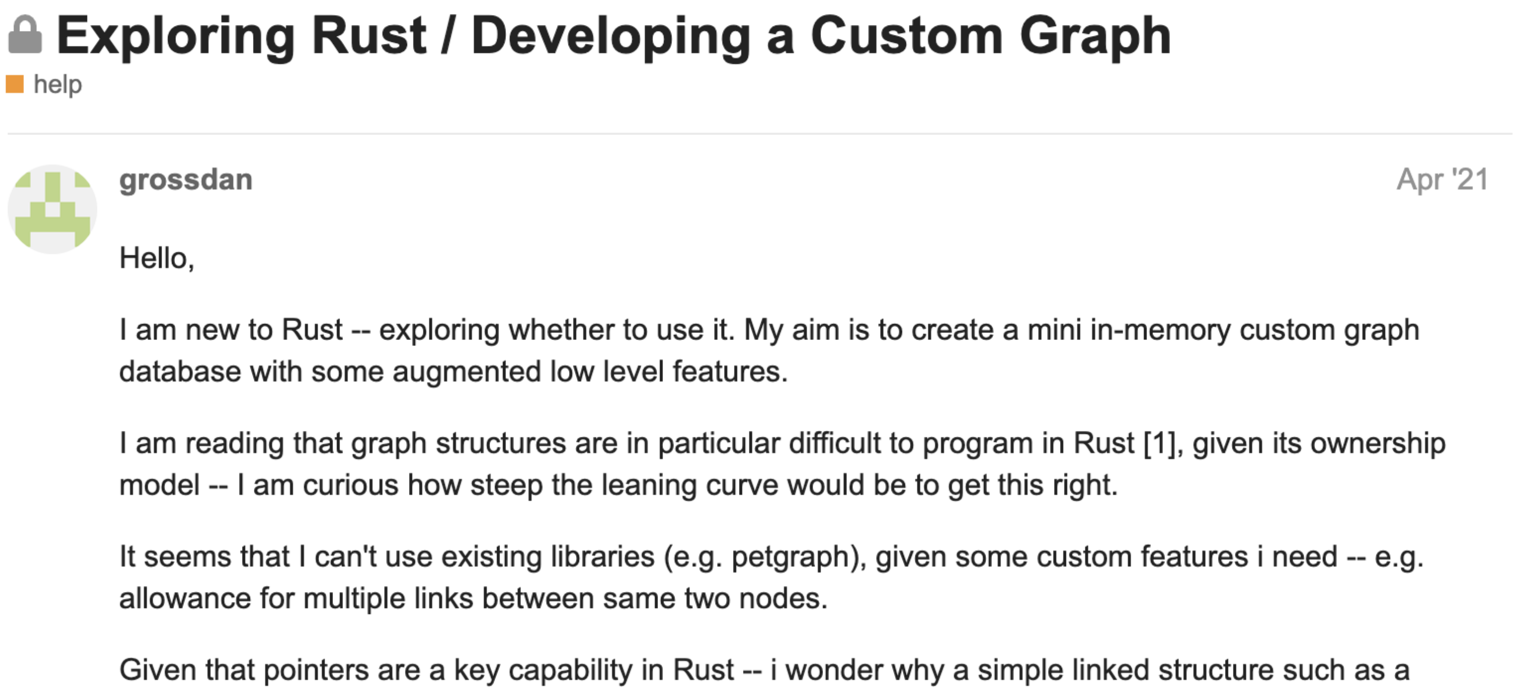
```
let element = a[index];
 |                    ^^^^^^^^ index out of
bounds: the len is 5 but the index is 10
```

Compile fails: Out-of-bounds access detected at compile time.

```rust
fn main() {
    let a = [1, 2, 3, 4, 5];
    let mut p = a.as_ptr();
    let slice;
    unsafe {
      p = p.offset(10);
      slice = slice::from_raw_parts(p, 1);
    }
    println!("{}", slice[0]);
}
```

Unsafe access in safe code!

Compile ok: Out-of-bounds access not detected at compile time or run time.



Exploring Rust / Developing a Custom Graph

grossdan Apr '21

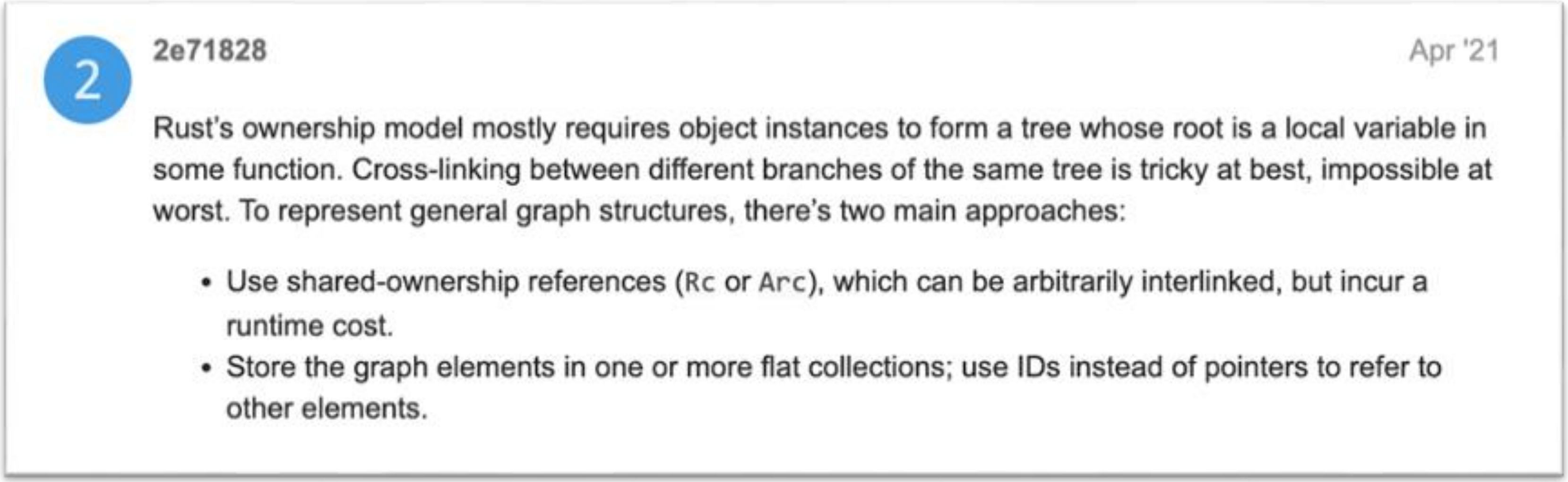
Hello,

I am new to Rust -- exploring whether to use it. My aim is to create a mini in-memory custom graph database with some augmented low level features.

I am reading that graph structures are in particular difficult to program in Rust [1], given its ownership model -- I am curious how steep the leaning curve would be to get this right.

It seems that I can't use existing libraries (e.g. petgraph), given some custom features i need -- e.g. allowance for multiple links between same two nodes.

Given that pointers are a key capability in Rust -- i wonder why a simple linked structure such as a



2e71828 Apr '21

Rust's ownership model mostly requires object instances to form a tree whose root is a local variable in some function. Cross-linking between different branches of the same tree is tricky at best, impossible at worst. To represent general graph structures, there's two main approaches:

- Use shared-ownership references (`Rc` or `Arc`), which can be arbitrarily interlinked, but incur a runtime cost.
- Store the graph elements in one or more flat collections; use IDs instead of pointers to refer to other elements.

https://users.rust-lang.org/t/exploring-rust-developing-a-custom-graph/57785

The Case for BMC for Rust

Rust is a great advancement for low-level languages

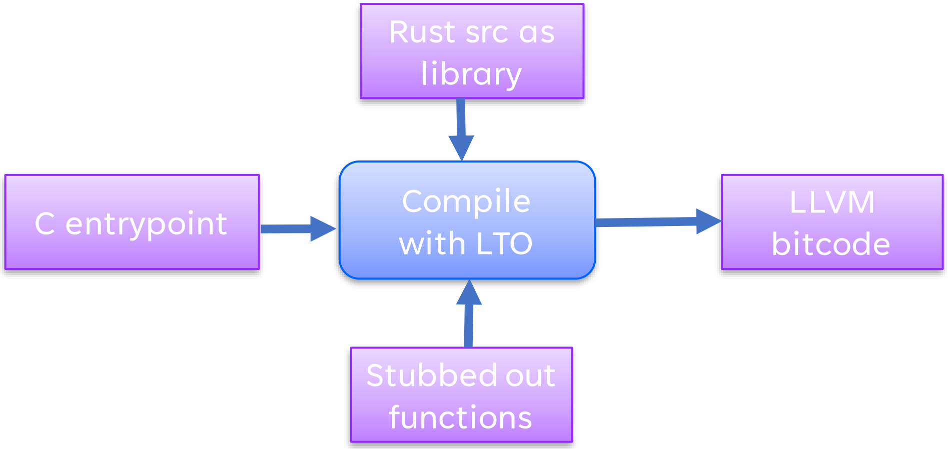
The type system is great at finding many subtle issues

However, any non-trivial program requires **unsafe** handling of raw pointers

Problem: Unsafe bits in Rust, make the whole program unsafe!

Solution: Use LLVM Bounded Model Checking for whole program analysis of *unsafe* Rust programs

c-rust: The SeaHorn Rust Pipeline

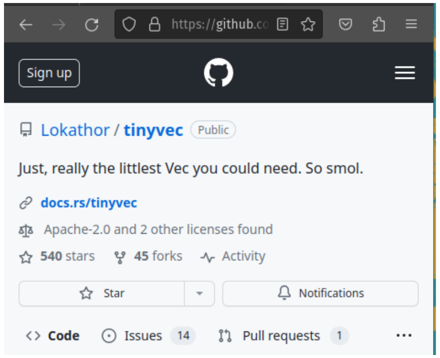

Rust src as library

C entrypoint

Compile with LTO

LLVM bitcode

Stubbed out functions



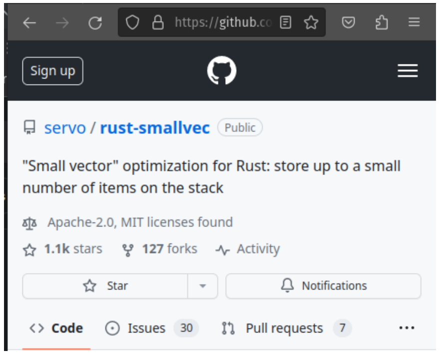
Lokathor / tinyvec Public

Just, really the littlest Vec you could need. So smol.

docs.rs/tinyvec

Apache-2.0 and 2 other licenses found

540 stars 45 forks Activity



servo / rust-smallvec Public

"Small vector" optimization for Rust: store up to a small number of items on the stack

Apache-2.0, MIT licenses found

1.1k stars 127 forks Activity

https://github.com/agurfinkel/c-rust

with Siddharth Priya, Boris Jancic, Thomas Hart

c-rust: Research Challenges

We can now verify Rust programs by reducing them to LLVM IR (i.e., C-like programs)

This is similar to analyzing x86 executables (or analyzing assembly)

- Pro: verification does not depend on the compiler (and finds compiler bugs)
- Cons: scalability is a challenge
  - verification does not benefit from higher-level ownership concepts
  - verification is complicated by complex complier lowering of concepts

Challenge 1

- develop low-level intermediate representation (IR) that captures ownership semantics

Challenge 2

- efficient verification of IR with ownership semantics

Extend SEA-IR with explicit Ownership Operations

SEA-IR program with aliasing..

```
// Px are pointer registers
// Rx are data registers
// Mx are memory registers
// (contain a memory region)
P0 = load PP0, M0
R0 = load P0, M0
R1 = R0 + 1
M1 = store R1, P0, M0
R2 = load P1, M1
R3 = R2 + 1
M2 = store R3, P1, M1
// P0, P1 may alias.
// Reload data at P0.
R4 = load P0, M2
assert R4 == 5
die P0
```

SEA-IR with **explicit ownership**

```
// P0 is moved.
// Hence, a unique ptr
P0 = mvmem2reg PP0, M0
R0 = load P0, M0
R1 = R0 + 1
// Cache write through P0
// as P0 unique
P2 = wrcache R1, P0
M1 = store R1, P2, M0
R2 = load P1, M1
R3 = R2 + 1
M2 = store R3, P1, M1
// Cache read as P2 unique
R5 = rdcache P2
R4 = load P0, M2
// Assert on cached value
assert R5 == 5
die P2
```

Extraneous instructions removed, including mem access through P0

```
P0 = mvmem2reg PP0, M0
R0 = load P0, M0
R1 = R0 + 1
P2 = wrcache R1, P0
R5 = rdcache P2
assert R5 == 5
```

Introduce new SEA-IR operations

- `mvmem2reg`, `die`, `brmem2reg`, to represent, moving, returning, and borrowing ownership of pointers between memory and registers
- `wrcache`, `rdcache` to cache metadata directly at a pointer

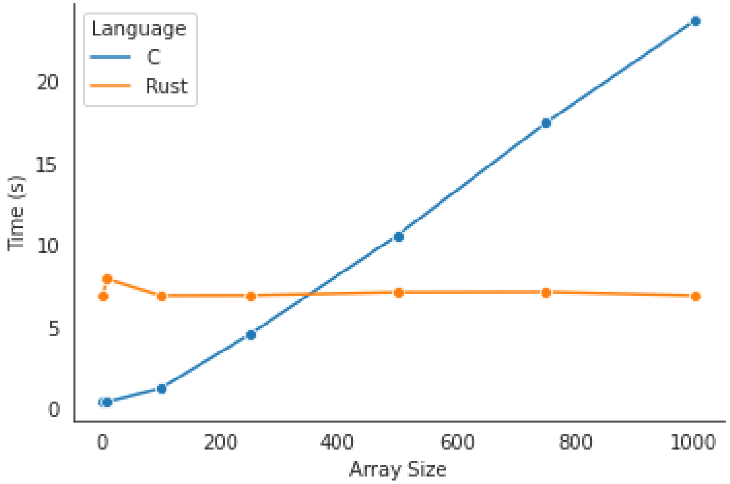
Verification time comparison

C

- memory operations generated in verification conditions
- verification time grows with array size

Rust

- verification time does not depend on array size
- very small vectors on stack generate memory load/store operations



Conclusion

Code-as-Spec empowers developers to write specifications

- familiar from unit testing
- executable counterexamples provide familiar feedback
- effectively share specifications between very different QA tools

Beware of bugs in specifications

- simple vacuity is very helpful
- much more research work is required!

SeaBMC – a new Bounded Model Checker for LLVM

- supports many features of LLVM IR
  - including memcpy, memmove, overflow instrinsics, etc.
- easy-to-use by using standard "de-facto" language semantics and integration with mainstream constructor
- performance comparable / better than state-of-the-art
- support for fat pointers and shadow memory simplifies property specification
- support for Rust is in active development – looking for interesting case studies

References

SeaHorn

- http://seahorn.github.io/
- includes CHC (spacer), AbsInt (clam/crab), BMC (SeaBmc), Alias (SeaDsa)

Case-Studies

- https://github.com/seahorn/verify-c-common
- https://github.com/seahorn/verifyTrusty
- https://github.com/agurfinkel/c-rust

Papers

- Siddharth Priya, Xiang Zhou, Yusen Su, Yakir Vizel, Yuyan Bao, Arie Gurfinkel: Verifying Verified Code. The 19th International Symposium on Automated Technology for Verification and Analysis (ATVA 2021)
- Siddharth Priya, Xiang Zhou, Yusen Su, Yakir Vizel, Yuyan Bao, Arie Gurfinkel: Bounded Model Checking for LLVM. The 22nd International Conference on Formal Methods in Computer-Aided Design (FMCAD 2022)

Blogs (on c-rust)

- http://seahorn.github.io/blog/

END