BOUNDED MODEL CHECKING FOR LLVM

Siddharth Priya*, Xiang Zhou, Yusen Su, Yakir Vizel, Yuyan Bao, and Arie Gurfinkel

MEET OUR TEAM

Siddharth Priya

University of Waterloo

Xiang Zhou

University of Waterloo

Yusen Su

University of Waterloo

Yakir Vizel

The Technion

Yuyan Bao

University of Waterloo

Arie Gurfinkel

University of Waterloo

OUTLINE

Context and Contributions

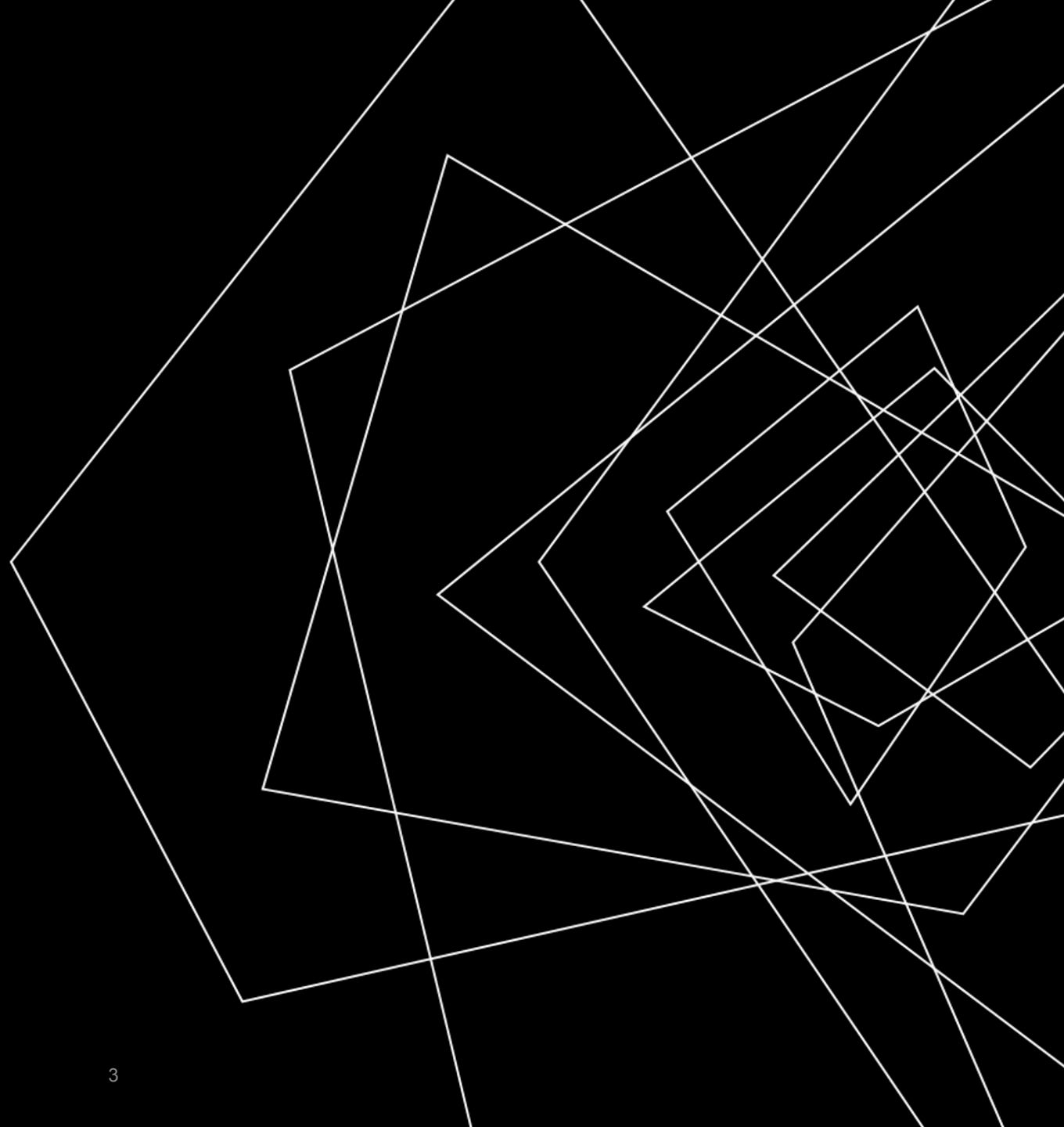
VC Generation

Tracking program state metadata

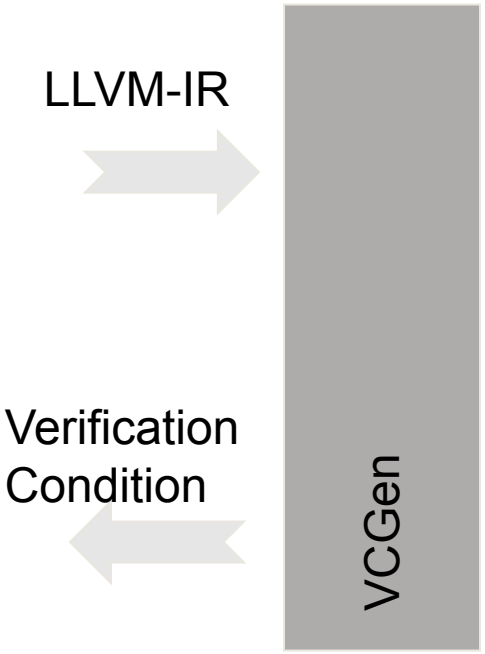
VCGen as a symbolic VM

Results

Future Work



CONTEXT



External View of SEABMC

Compile programs to LLVM IR, .e.g., from C.

Bound execution length (unroll and cut loops).

Generate verification condition (VC) in SMT form.

Verify VC using a solver.

CONTRIBUTIONS

MULTIPLE VCGEN STRATEGIES

Introduce an IR language on top of **LLVM** IR called **SEA-IR**.

Generate VCs from SEA-IR programs in control flow or data flow form and different memory representations – SMT theory of arrays vs Lambdas.

Configurability enables quick experimentation.

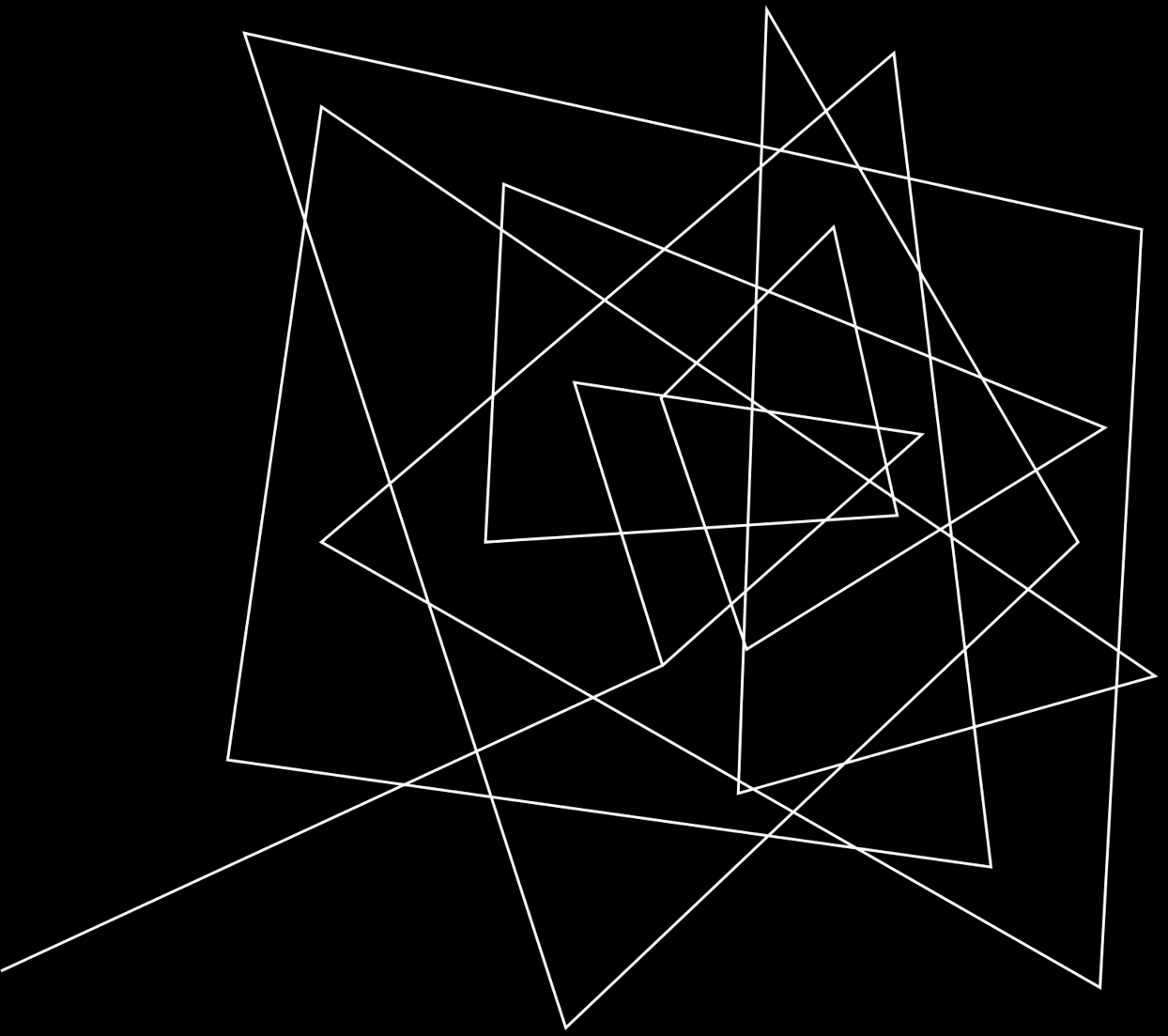
CONVENIENT PROGRAM STATE METADATA STORAGE

Provide mechanisms and interface to track program state metadata by allowing (**shadow**) **memory** and **(fat**) **pointers** to store metadata.

RESULTS ON PRODUCTION CODE

SEABMC - **Open-sourced** BMC engine for the SEAHORN program analysis framework.

Re-verify **`aws-c-common`** library using SEAHORN and compare with state-of-the-art verification tools.



VC GENERATION

SEA-IR – PURIFY MEMORY OPERATIONS

```
PR    ::=  fun main(){BB+}
BB    ::=  L : PHI* S+ (BR | halt)
BR    ::=  br E, L, L | br L
PHI   ::=  R = phi [R, L](,[R, L])* |
           M = phi [M, L](,[M, L])* |
           P = phi [P, L](,[P, L])*
S     ::=  RDEF | MDEF | VS
RDEF  ::=  R = E | P, M = alloca R, M |
           P, M = malloc R, M | R = load P, M |
           P = load P, M | M = free P, M
MDEF  ::=  M = store R, P, M | M = store P, P, M
VS    ::=  assert R | assume R
```

SEA-IR syntax

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

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;
}
```

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
}
```


Vcgen: SA Control Flow form can be used to generate verification conditions

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

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

VC generation can happen from control flow form with phi nodes

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 generation can happen from control flow (data flow) form with gamma nodes

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
}
```

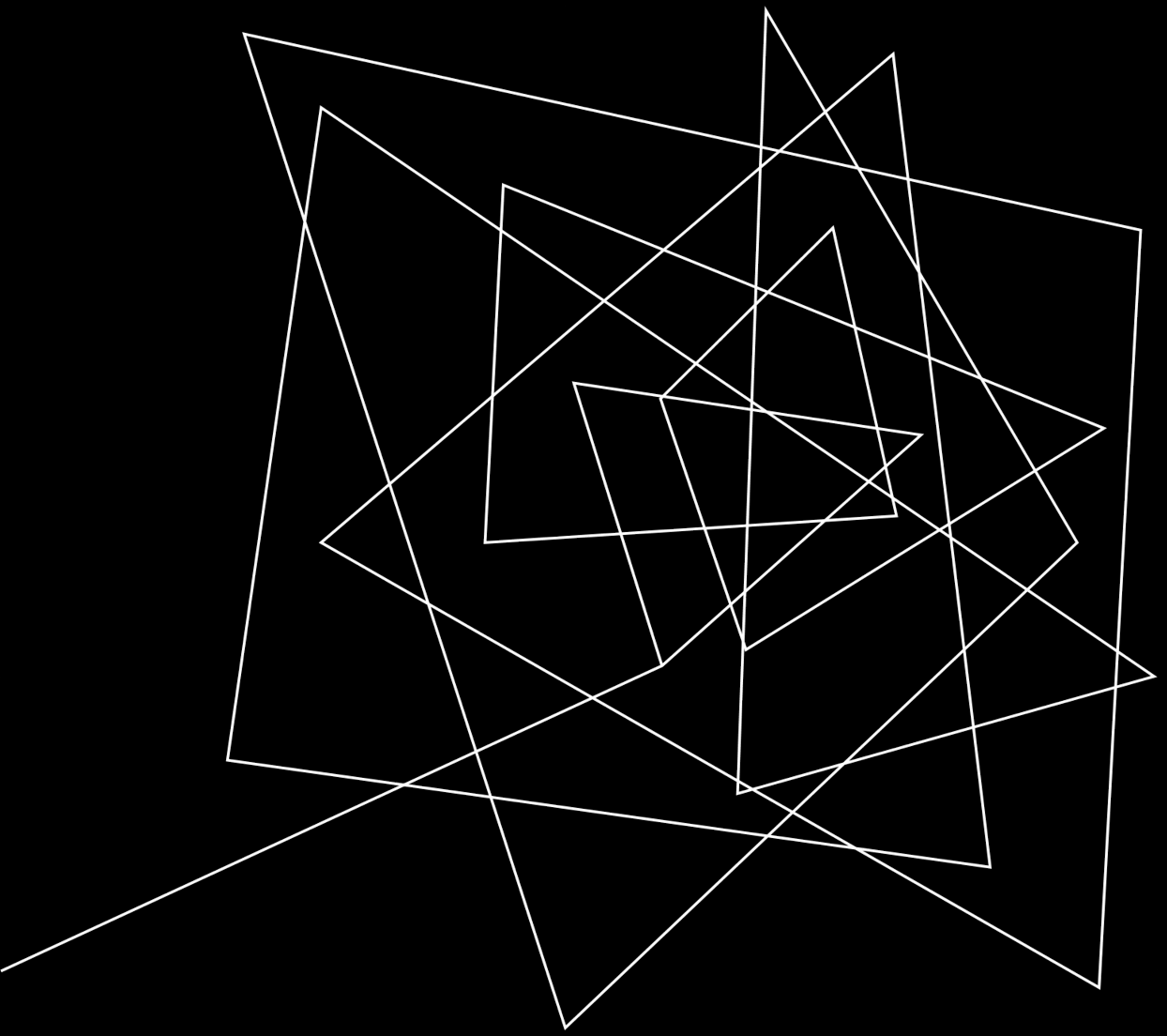
GSA program: SEA-IR program in gated SSA form (**GSA**). It has a single assume and a single assert (**SASA**).

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.



TRACKING PROGRAM STATE METADATA

Using Shadow memory and fat pointers

SHADOW MEMORY AND FAT POINTERS

Shadow every byte (or word) of program memory with program state metadata. E.g.,

- 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	Metadata 0	Metadata 1	Metadata 2
Addr0			
Addr1			
...			
AddrN			

Shadow mem representation

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

Address	Metadata0	Metadata1	Metadata2

Fat pointer representation

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

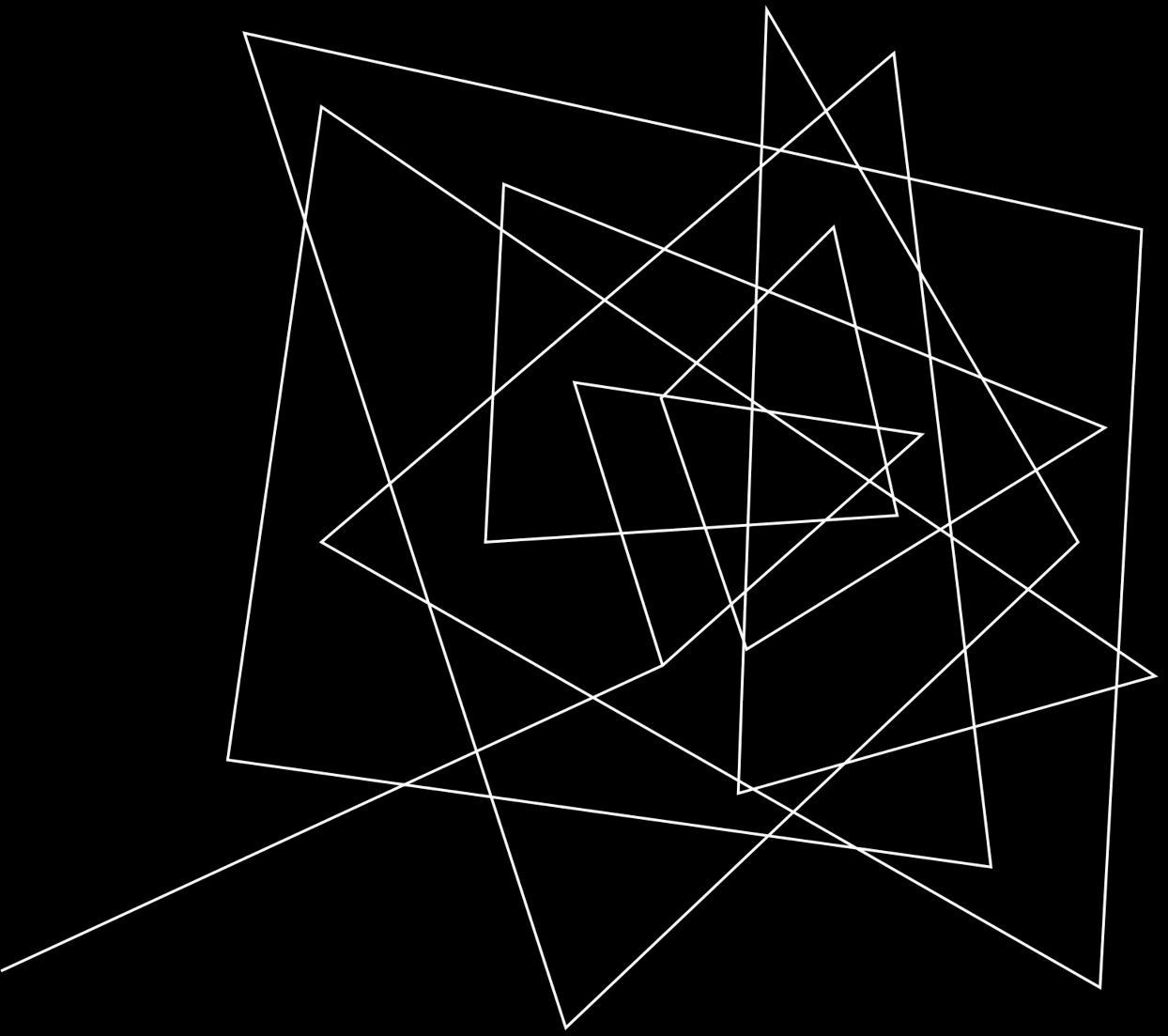
Intrinsic like `sea_is_alloc` operate on program metadata.

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)

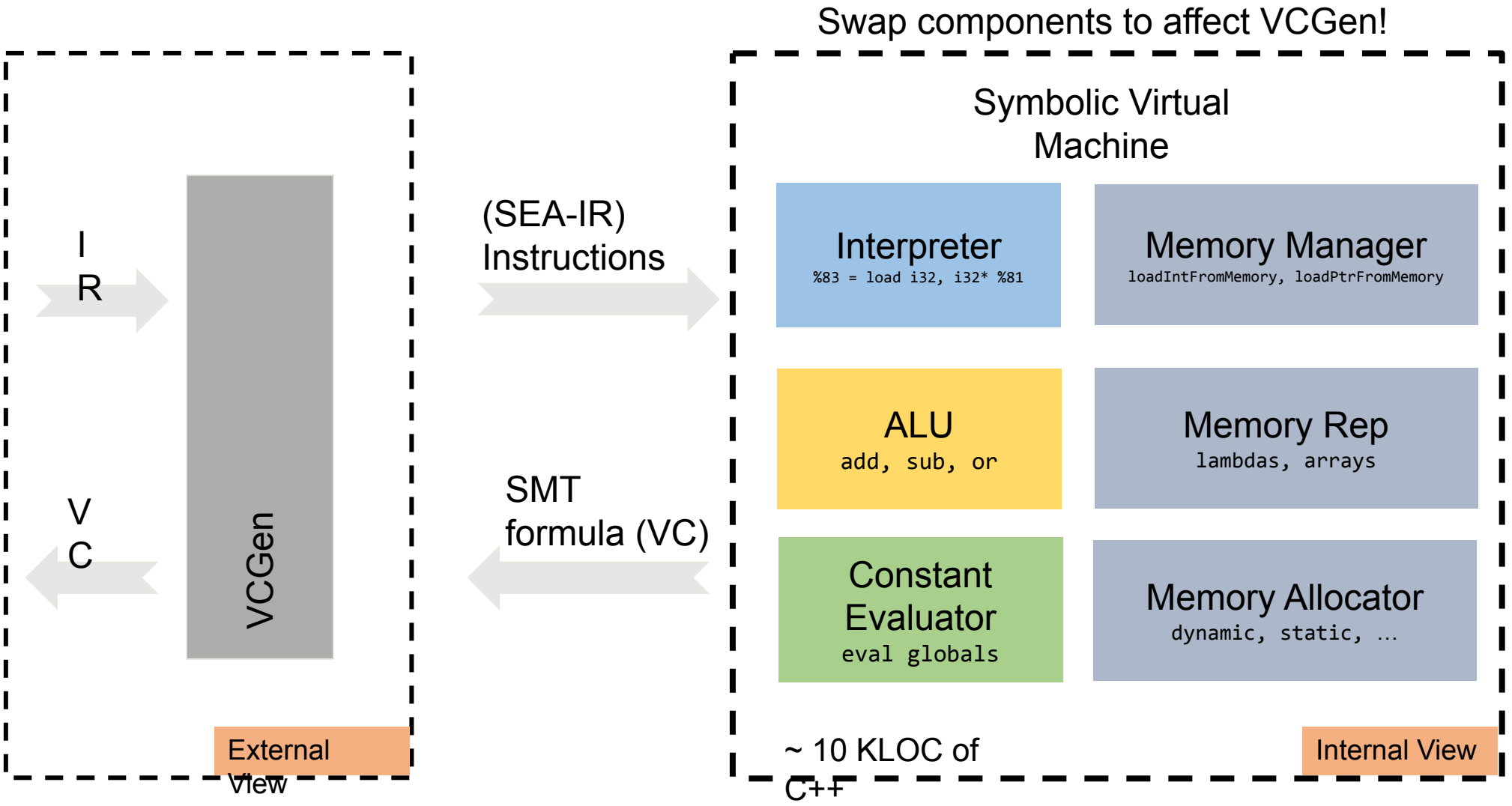
```
int main() {
  char *p = (char *) malloc(sizeof(char));
  sea_is_alloc(p);
  *p = 0;
  free(p);
  sea_is_alloc(p);
  *p = 255;
  return 0
}
```

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

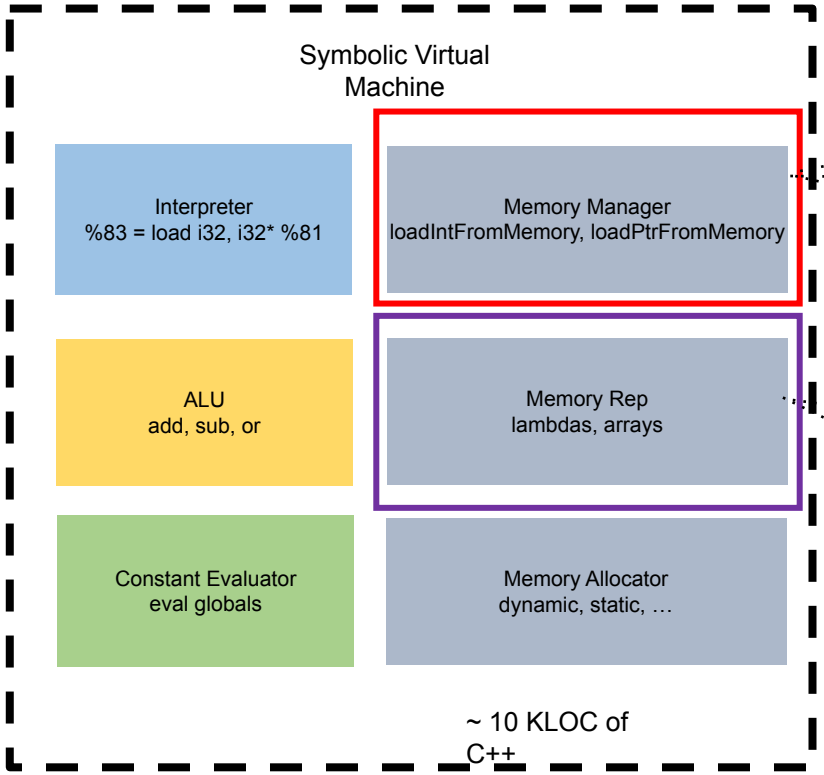


VCGEN AS A SYMBOLIC VM

BACKEND: VCGEN AS A (SYMBOLIC) VM



BACKEND: VCGEN AS A (SYMBOLIC) VM



Ordinary pointers

Fat pointers

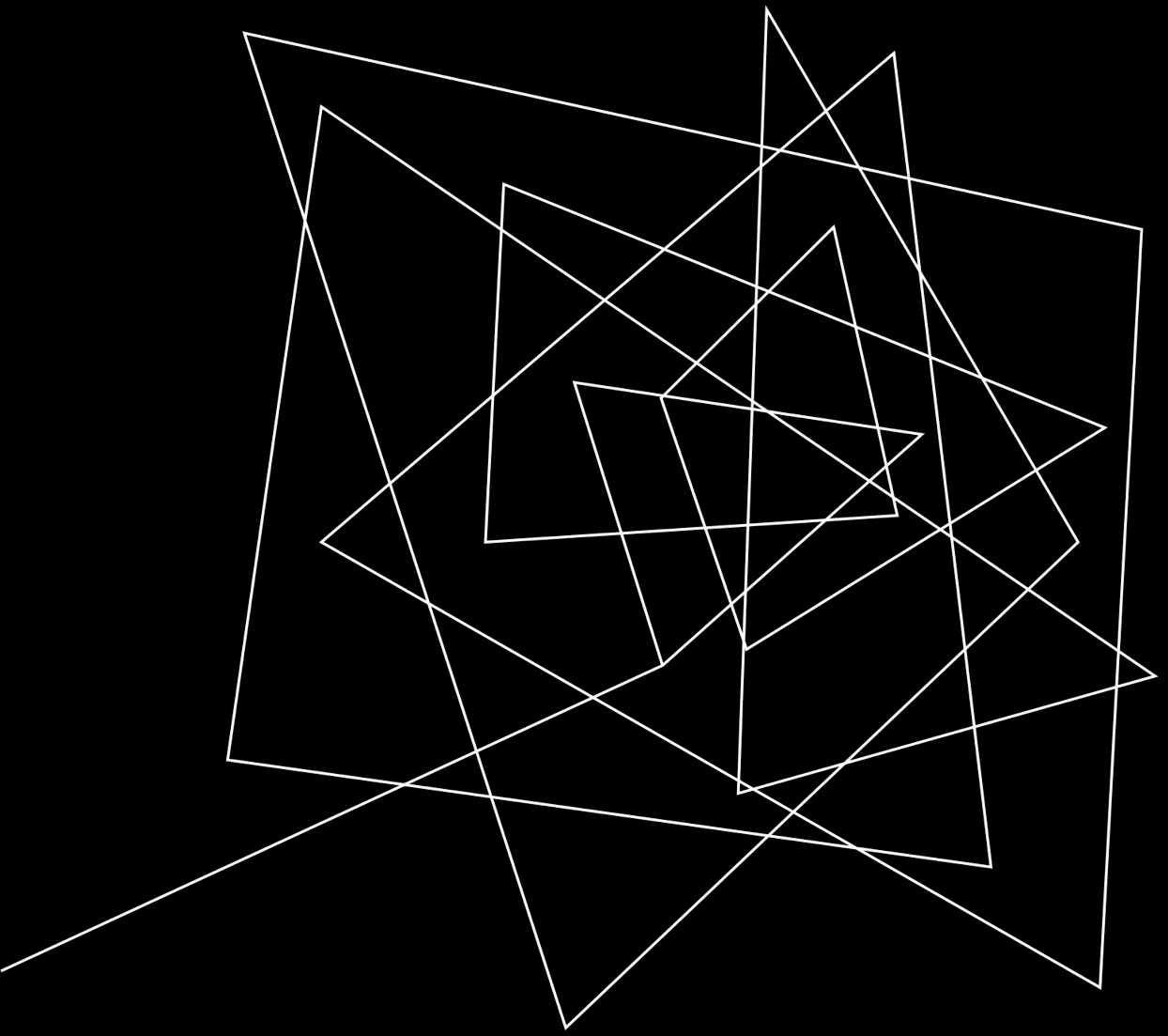
Fat pointer with
Shadow memory

`SMT theory of arrays`

Lambda memory: memory as a lambda abstraction

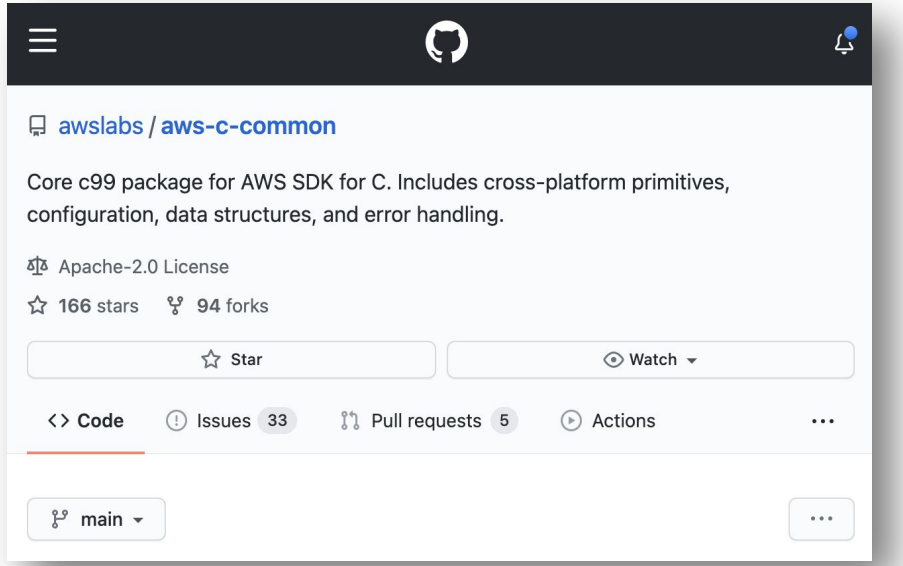
To **store**, add an ITE to top of tree
To **load**, beta-reduce the given abstraction with an address.

```
λa.ite(addr0=a, val0, ite(addr1=a, val1,…))
```



RESULTS

aws-c-common library



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

Core C99 package for AWS SDK

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

Self-contained

Low-level and platform specific C

Extensively verified using CBMC*

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

[*] Code-Level Model Checking in the Software Development Workflow, Chong et al., ICSE 2020

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

RESULTS: AWS-C-COMMON

OPT VCGEN STRATEGY

- Z3 with memory as lambdas
- Beta reduce lambdas early
- Use Pure dataflow program
- Reduce program using cone-of-influence.

VERIFICATION OUTCOME

Strengthen findings of original verification effort using CBMC

Found no bugs in production code but found bugs in proofs.

Shadow memory can make verification and specification simpler.

COMPARISION WITH STATE-OF-THE-ART

- Compared with CBMC, SMACK, SYMBIOTIC and KLEE
- 10x faster than these tools
- Open-source tool and reproducible results
- Continuous verification

FUTURE WORK – GENERATE SIMPLER VERIFICATION CONDITIONS

Utilize fat pointers and shadow memory to express safety properties in a user-friendly way and generate simpler VC.

Apply BMC to Rust. Use ownership semantics to simplify VC.

Use more sophisticated static analysis to solve assertions statically.

THANK YOU

Siddharth Priya

Siddharth.priya@uwaterloo.ca

FAT POINTERS -- SPATIAL MEMORY SAFETY

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

OOB access; Undefined behaviour

Figure: Fat pointers contain address and metadata

Add more metadata at pointers!

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

Figure: isderef semantics

Problem: Ensure all memory accesses are within allocated bounds.

Solution: In the symbolic VM, expand pointers to pointers. Provide API to compute on fat.

Pointer definition and manipulation
Allocation sets up base, offset and size. Offset is updated on pointer arithmetic.

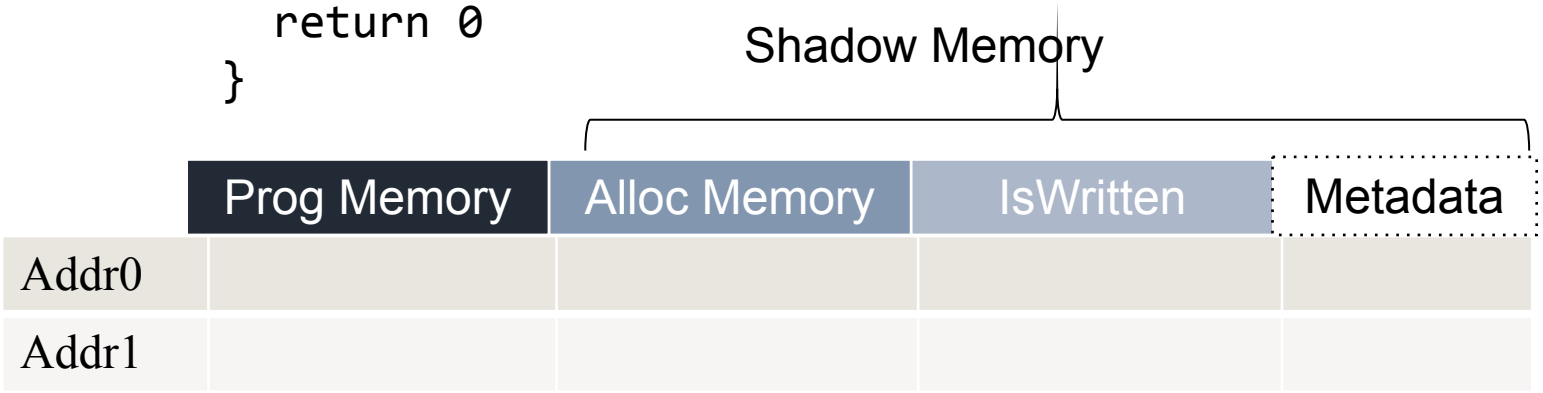
Pointer dereference
Add **isderef** checks on all accesses. Attempt to solve them using static analysis.

Isderef checks are automatically added for every access. Many checks are solved resolved before SMT solving.

SHADOW MEM -- TEMPORAL MEMORY SAFETY

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

UAF; Undefined behaviour



Add more metadata at addresses!

```
sym(R1 = isalloc P0 M) == r1 = read(m.alloc, p0.base)
```

Figure: isderef semantics

Problem: Ensure memory type state is OK; E.g., memory is allocated, read only memory is not mutated.

Solution: In the symbolic VM, add shadow memory. Store metadata keyed by address.

Memory Def/free
Set alloc memory to true/false.

Memory use
Add **isalloc** checks.

We record metadata at base of pointer. Thus, need fat pointers.

SEA-IR: PROGRAM TRANSFORMATION

Source form

```
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 form

```
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).

...

GSSA form

```
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
}
```

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

VCGEN

```
(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 **GSSA** program using pure dataflow analysis.

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