Saturn - Software Deobfuscation Framework based on LLVM

```
     /\
    (  )
    (  )
    (  )
   /|/\|\
  /_||||_\
    /  \
  (*#*)
    (*#)
     (#*)
     (*)
```

Peter Garba, Thales, DIS - Cybersecurity
Matteo Favaro, Zimperium, Mobile Security

Who are we

Peter Garba - Software Security Expert

- Working on Thales internal obfuscation tool *Zcrambler*
- Tries to automate attacks with strong tooling
- Reverse Engineering for more than 20 years

Matteo Favaro - Malware Analyst

- Working at Zimperium, analysing malware and protectors
- Attacks commercial obfuscators to have fun and learn
- Reverse engineering since 2015

Background story - Classical (de)obfuscation

Classical obfuscation: obfuscation patterns, constant unfolding and junk code insertion.

Classical deobfuscation: pattern matching, but it's time consuming and every obfuscator update brings new patterns. Constant folding and junk code removal, at the binary level they are annoying.

1. Obfuscated code pattern

```
MOV EBX,DWORD PTR DS:[146CFD0]
PUSH EBX
PUSHFD
MOV EBX,DWORD PTR DS:[1467D94]
NOT EBX
XOR EBX,DWORD PTR DS:[47AE30]
XCHG DWORD PTR SS:[ESP+4],EBX
POPFD
RETN
```

2. Deobfuscated code

```
JMP 0xXXXXXXXX
```

Background story - Modern (de)obfuscation

Modern obfuscation: more often than not it is applied at the source code or intermediate representation level, but has to deal with unwanted optimizations.

Modern deobfuscation: uses intermediate languages at different abstraction layers to analyse and automate the attacks. Every modern reverse engineering tool comes with one or more embedded intermediate representations.

Adoption: with the introduction of open source projects like *Obfuscator-LLVM* or *ADVObfuscator* the door for compiler-based obfuscation tools has been widely opened to the public.

Background story - IDEA!

- Compilers expose:
  - feature rich intermediate languages
  - solid analysis techniques
  - strong optimizations
  - accessible API
- Using a compiler we can handle the obfuscation problem at the same conceptual level.
- Tired of reinventing the wheel with custom or error prone optimization implementations.

The idea for a generic deobfuscation tool based on *LLVM* was born!

Background story - Proof of Concept

- **2012** - First implementation of an x86 binary code lifter based on *LLVM* that proved that the idea could work out
- Unfortunately the lifter was not supporting all the x86 opcodes as it was made for a different use case

```
MOV EAX, 12340000h
MOV EDX, 5678h
XOR EAX, EDX
```

- It took some more years until ***Trail of Bits*** released *Remill* which made the work on ***SATURN*** possible

Motivating example

1. Obfuscated code

```c
int func(char chr, char ch1, char ch2) {
  char garb = 0; char ch = 0;
  // FOR trick
  for (int i = 0; i < chr; i++)
    ch++;
  // SPLIT trick
  if (ch1 > 60)
    garb++;
  else
    garb--;
  if (ch2 > 20)
    garb++;
  else
    garb--;
  // MBA based opaque predicate
  if ((chr + ch2) == ((chr ^ ch2) + 2 * (chr & ch2)))
    ch ^= 97;
  else
    ch ^= 23;
  return (ch == 31);
}
```

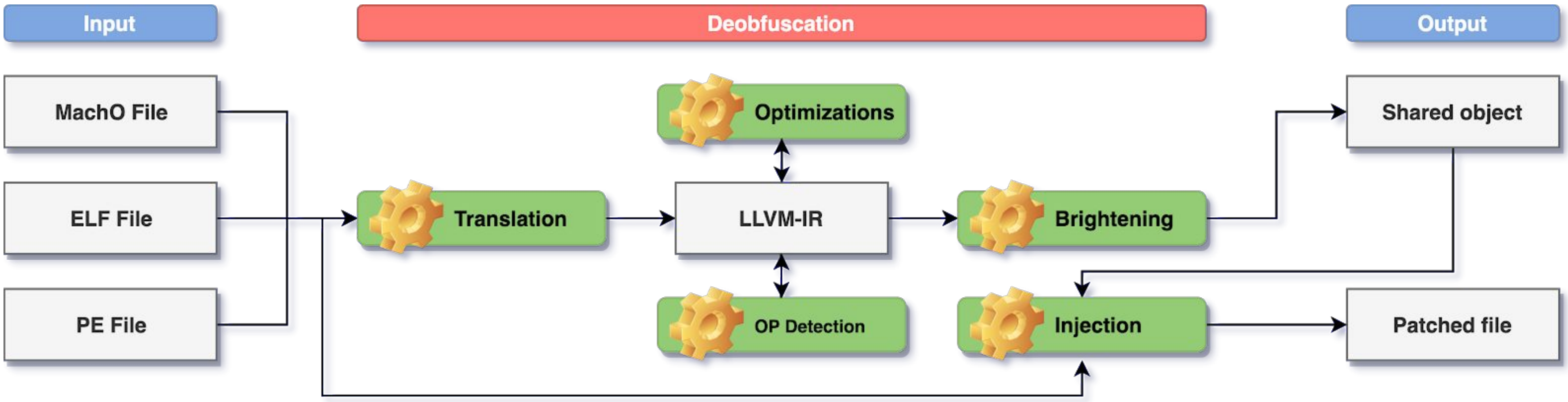
2. Deobfuscated code

```
define dso_local i32 @func(i8 signext) local_unnamed_addr #0 {
  %2 = icmp eq i8 %0, 126
  %3 = zext i1 %2 to i32
  ret i32 %3
}
```

3. Decompiled code

```c
int func(char in) {
  return (in == 126);
}
```

SATURN'S WORKFLOW



***Brightening* [COMP.]** *verb - Reshaping code to make it more readable and understandable for humans*

Translation - REMILL

- Highly focused on semantic correctness compared to similar tools
- Based on a high level ***State*** structure to guarantee generality in the implementation of the lifting
- Extended support for the *x86*, *amd64* and *aarch64* ISA and good documentation on how to add the missing instructions
- Exposes a set of API to inspect and manipulate the lifted LLVM-IR in the context of the ***State*** structure
- The generated LLVM-IR may seem verbose at first, but the unnecessary code easily folds when optimizations are applied to it

Translation - Opcode Lifting "(0x50) PUSH RAX"

REMILL

1. Opcode pseudocode

```
IF StackAddrSize = 64
    THEN
        IF OperandSize = 64
            THEN
                RSP ← RSP - 8;
                Memory[SS:RSP] ← SRC;
        ELSE IF OperandSize = 32
            THEN
                RSP ← RSP - 4;
                Memory[SS:RSP] ← SRC;
            ELSE (* OperandSize = 16 *)
                RSP ← RSP - 2;
                Memory[SS:RSP] ← SRC;
        FI;
```

2. Remill's implementation

```
template <typename T>
DEF_HELPER(PushToStack, T val) -> void {
  addr_t op_size = ZExtTo<addr_t>(ByteSizeOf(val));
  addr_t old_xsp = Read(REG_XSP);
  addr_t new_xsp = USub(old_xsp, op_size);
  Write(WritePtr<T>(new_xsp _IF_32BIT(REG_SS_BASE)), val);
  Write(REG_XSP, new_xsp);
}
```

3. Lifted result

```
; <label>:77:                                       ; preds = %entry
  %78 = load i64, i64* %RAX
  %79 = load i64, i64* %PC
  %80 = add i64 %79, 1
  store i64 %80, i64* %PC
  %81 = load %struct.Memory*, %struct.Memory** %MEMORY
  %82 = call %struct.Memory* @"??$PUSH@U?$In@_K@@@?A0x8DC3AA66@@YAPEAUMemory@@PEA
U0@AEAUState@@U?$In@_K@@@Z"(%struct.Memory* %81, %struct.State* %0, i64 %78)
  store %struct.Memory* %82, %struct.Memory** %MEMORY
  br label %83
```

Translation - REMILL - Building blocks

REMILL

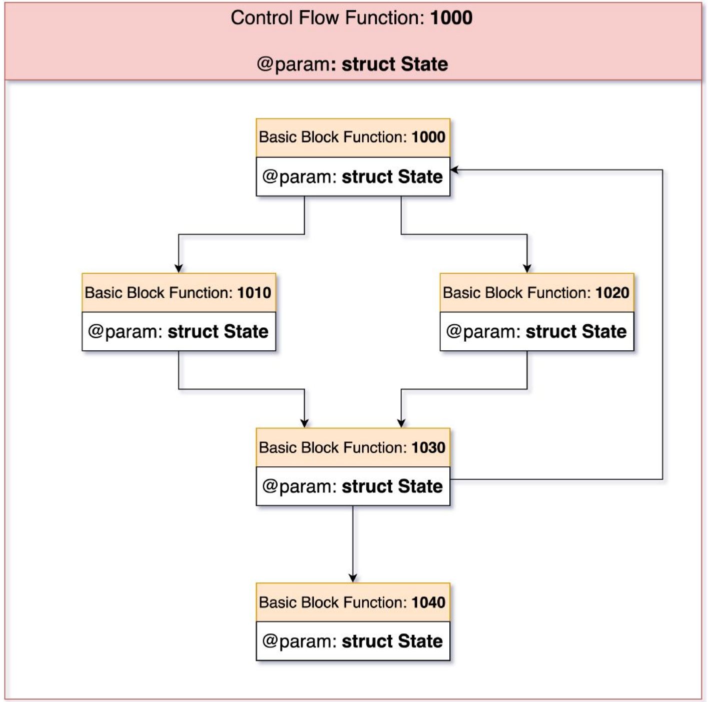
1. Remill *State* structure definition

```
struct State {
  VectorReg vec[kNumVecRegisters];
  ArithFlags aflag;
  Flags rflag;
  Segments seg;
  AddressSpace addr;
  GPR gpr;
  X87Stack st;
  MMX mmx;
  FPUStatusFlags sw;
  XCR0 xcr0;
  FPU x87;
  SegmentCaches seg_caches;
}
```

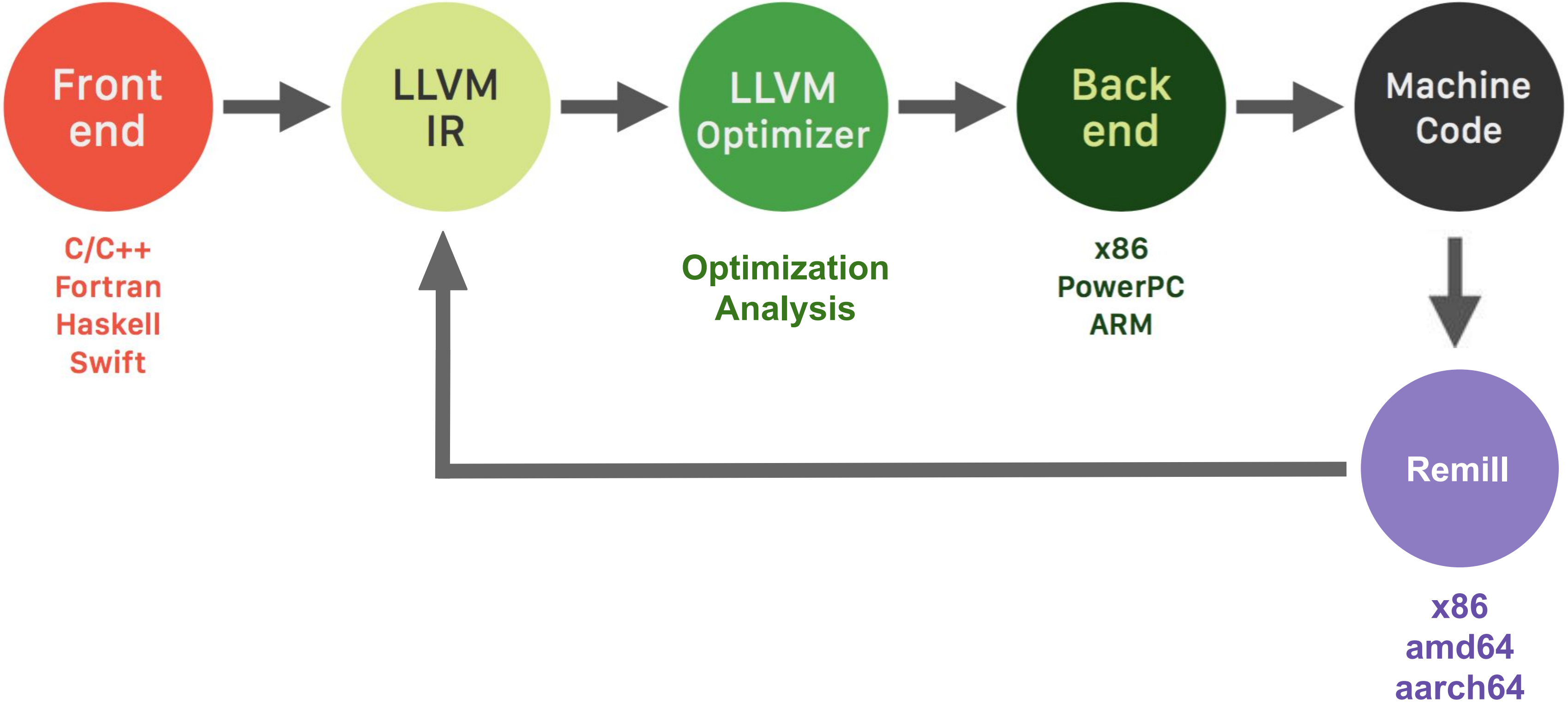
2. Remill basic block definition

```
Memory *__remill_basic_block(State &state, addr_t curr_pc, Memory *
    ↪ memory);
```

3. Remill control flow graph



Optimization - LLVM



Optimization - LLVM - Benefits

- Easy inspection and modification of functions, control flow graphs, basic blocks and instructions
- World class optimization and analysis passes:
  - Dead code elimination
  - Peephole optimization
  - Constant folding
  - Loop analysis and optimization
  - Memory and pointer aliasing
  - Many more...
- High code quality and correctness standards
- Hundreds of projects are relying on it (e.g. *Souper* and *KLEE*)

Considering what stated above, the LLVM-IR nicely fits as an intermediate representation for deobfuscation tasks

Optimization - Souper

1. Unoptimized code

```
unsigned
g(unsigned a) {
  switch (a % 4) {
  case 0:
    a += 3;
    break;
  case 1:
    a += 2;
    break;
  case 2:
    a += 1;
    break;
  }
  return a & 3;
}
```

2. Inferred optimization

```
%0 = block 4
%1:i32 = var
%2:i32 = urem %1, 4:i32
%3:i1 = ne 0:i32, %2
%4:i1 = ne 1:i32, %2
%5:i1 = ne 2:i32, %2
blockpc %0 0 %3 1:i1
blockpc %0 0 %4 1:i1
blockpc %0 0 %5 1:i1
blockpc %0 1 %2 2:i32
blockpc %0 2 %2 1:i32
blockpc %0 3 %2 0:i32
%6:i32 = add 1:i32, %1
%7:i32 = add 2:i32, %1
%8:i32 = add 3:i32, %1
%9:i32 = phi %0, %1, %6, %7, %8
%10:i32 = and 3:i32, %9
infer %10

⇒

result 3:i32
```

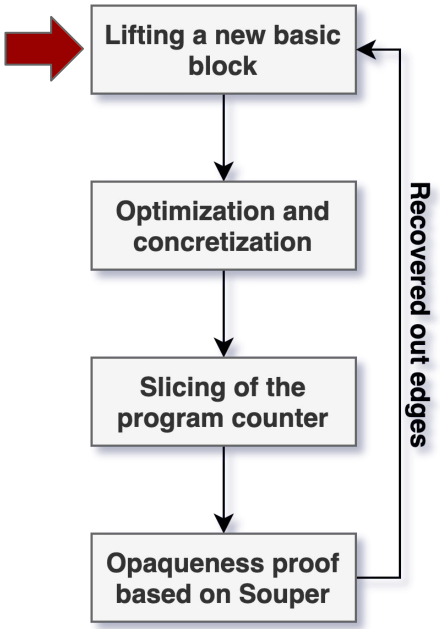
Optimization - Souper - Benefits

- Optimizes away obfuscation patterns missed by *LLVM*
- Keeps a cache of inferred optimizations in a *Redis* database
- Highly configurable:
    - More or less aggressive synthesis steps
    - Several constant synthesis strategies

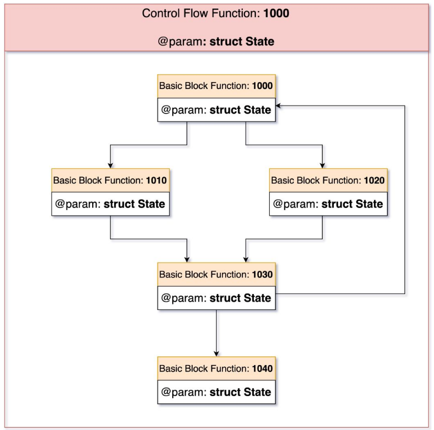
Internally used to:

- Translate LLVM-IR into SMT representation thanks to *KLEE*
- Support several SMT solvers like *Z3*, *STP*, *Boolector*, *CVC4*
- Proving the opaqueness of a control flow instruction
- Calculate the range of destinations in a switch-case

Saturn - Exploration

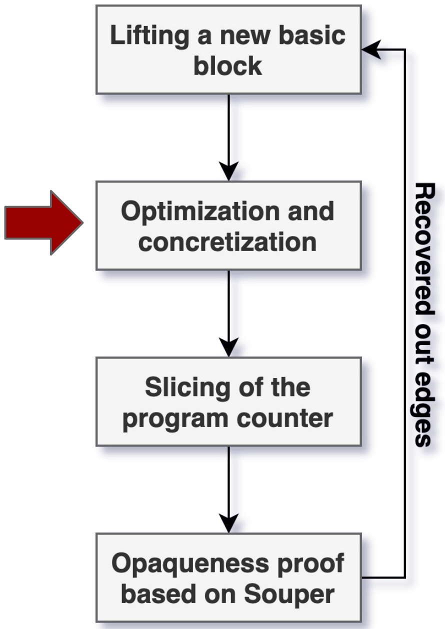


1. We start at the entry point of a function and we lift a basic block at a time
2. Based on the final instruction in a basic block we decide if continuing or stopping the exploration
3. Some control flow instructions require an opaqueness proof according to the opaqueness table
4. We connect the discovered basic blocks to form a control flow graph



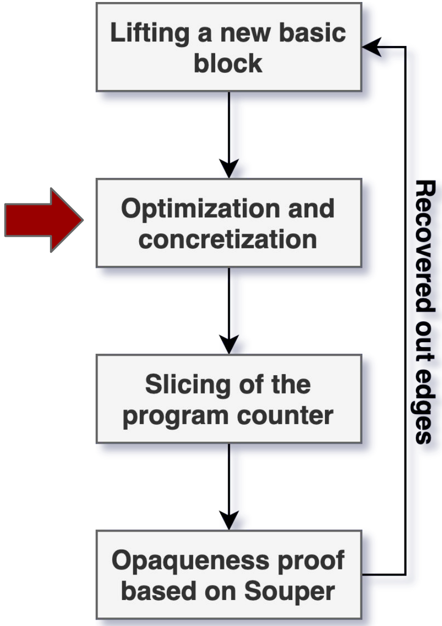
kCategory	Exploration	Opaqueness Proof
NoOp	Continue	No
Normal	Continue	No
FunctionReturn	Stop	Yes
IndirectJump	Stop	Yes
DirectJump	Stop	No
ConditionalBranch	Stop	Yes
IndirectFunctionCall	Stop	Yes
DirectFunctionCall	Continue	No

Saturn - Stack Slot Analysis



- We need the alias analysis to be able to track the values that are written and read on the stack
- *LLVM* is failing to do the alias analysis on the **IntToPtr** instructions obtained from the *Remill* memory access intrinsics
- We need to convert the **IntToPtr** values into **GetElementPtr** to enable the *LLVM* alias analysis pass
- We identify the stack slots and create a local variable for each of them
- The **IntToPtr** instructions are then replaced with **GetElementPtr** instructions
- The *LLVM* alias analysis is now able to properly work

Saturn - Concretization



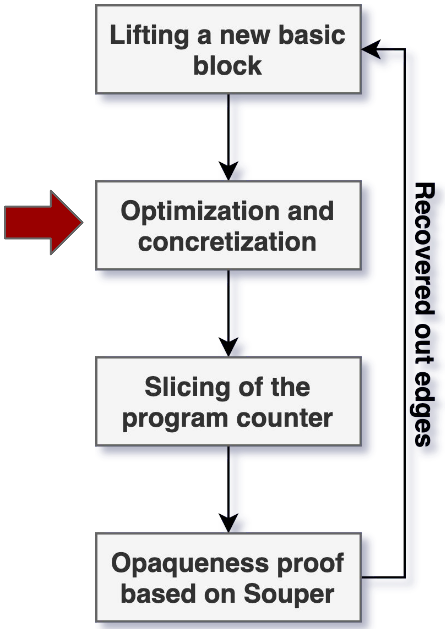
1. Simple no-op pattern

```
push   rax
pop    rax
```

2. Code lifted by Remill

```
define dso_local %struct.Memory* @stub_0(%struct.State*, i64, %struct.Memory*) {
  %RAX_PTR = getelementptr inbounds %struct.State, %struct.State* %0, i64 0, i32 6, i32 1, i32 0, i32 0
  %RAX = load i64, i64* %RAX_PTR, align 8
  %RSP_PTR = getelementptr inbounds %struct.State, %struct.State* %0, i64 0, i32 6, i32 13, i32 0, i32 0
  %RSP = load i64, i64* %RSP_PTR, align 8
  %Tmp = add i64 %RSP, -8
  %Mem = tail call %struct.Memory* @__remill_write_memory_64(%struct.Memory* %2, i64 %Tmp, i64 %RAX) #3
  store i64 %RSP, i64* %RSP_PTR, align 8
  %RAX_1 = tail call i64 @__remill_read_memory_64(%struct.Memory* %Mem, i64 %Tmp) #3
  store i64 %RAX_1, i64* %RAX_PTR, align 8
  ret %struct.Memory* %Mem
}
```

Saturn - Concretization



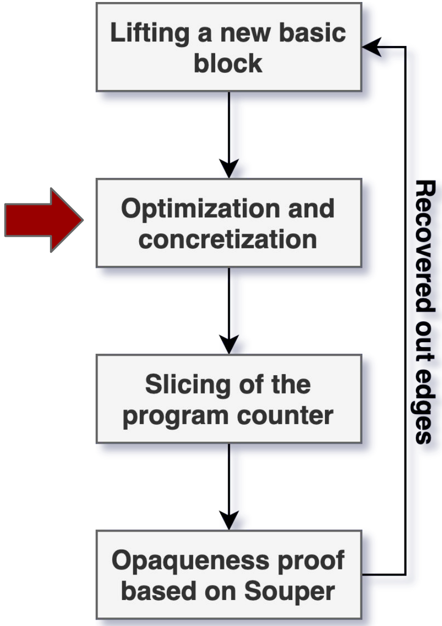
1. Simple no-op pattern

```
push    rax
pop     rax
```

2. Concretize RSP and optimize

```
define dso_local %struct.Memory* @stub_0(%struct.State*, i64, %struct.Memory*) {
  %RAX_PTR = getelementptr inbounds %struct.State, %struct.State* %0, i64 0, i32 6, i32 1, i32 0, i32 0
  %RAX = load i64, i64* %RAX_PTR, align 8
  %RSP_PTR = getelementptr inbounds %struct.State, %struct.State* %0, i64 0, i32 6, i32 13, i32 0, i32 0
  store i64 100000, i64* %RSP_PTR, align 8
  %RSP = load i64, i64* %RSP_PTR, align 8
  %Tmp = add i64 %RSP, -8
  %Mem = tail call %struct.Memory* @__remill_write_memory_64(%struct.Memory* %2, i64 %Tmp, i64 %RAX) #3
  store i64 %RSP, i64* %RSP_PTR, align 8
  %RAX_1 = tail call i64 @__remill_read_memory_64(%struct.Memory* %Mem, i64 %Tmp) #3
  store i64 %RAX_1, i64* %RAX_PTR, align 8
  ret %struct.Memory* %Mem
}
```

Saturn - Concretization



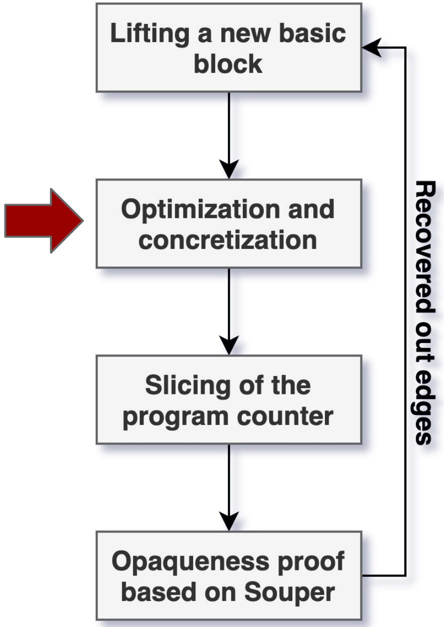
1. Simple no-op pattern

```
push   rax
pop    rax
```

2. RSP gets propagated and we can detect the stack slots

```
define dso_local %struct.Memory* @stub_0(%struct.State*, i64, %struct.Memory*) {
  %RAX_PTR = getelementptr inbounds %struct.State, %struct.State* %0, i64 0, i32 6, i32 1, i32 0, i32 0
  %RAX = load i64, i64* %RAX_PTR, align 8
  %RSP_PTR = getelementptr inbounds %struct.State, %struct.State* %0, i64 0, i32 6, i32 13, i32 0, i32 0
  %Mem = tail call %struct.Memory* @__remill_write_memory_64(%struct.Memory* %2, i64 99992, i64 %RAX) #2
  store i64 100000, i64* %RSP_PTR, align 8
  %RAX_1 = tail call i64 @__remill_read_memory_64(%struct.Memory* %Mem, i64 99992) #2
  store i64 %RAX_1, i64* %RAX_PTR, align 8
  ret %struct.Memory* %Mem
}
```

Saturn - Concretization



1. Simple no-op pattern

```
push    rax
pop     rax
```

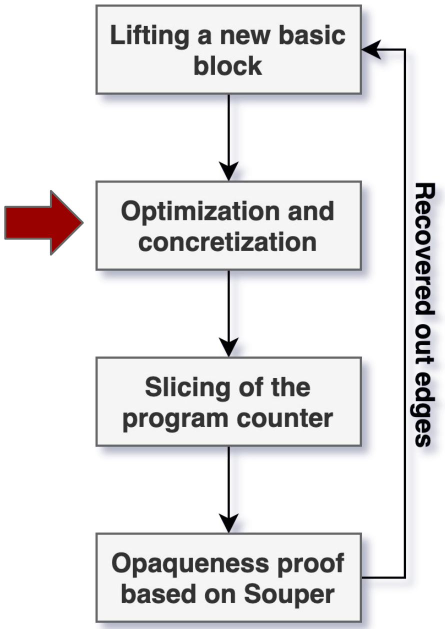
3. Detected stack slot

```
99992 SS0
```

2. Create the stack slots, replace the intrinsics and optimize

```
define dso_local %struct.Memory* @stub_0(%struct.State*, i64, %struct.Memory*) {
  %SS0 = alloca i64, align 8
  %RAX_PTR = getelementptr inbounds %struct.State, %struct.State* %0, i64 0, i32 6, i32 1, i32 0, i32 0
  %RAX = load i64, i64* %RAX_PTR, align 8
  %RSP_PTR = getelementptr inbounds %struct.State, %struct.State* %0, i64 0, i32 6, i32 13, i32 0, i32 0
  store i64 %RAX, i64* %SS0
  store i64 100000, i64* %RSP_PTR, align 8
  %RAX_1 = load i64, i64* %SS0, align 8
  store i64 %RAX_1, i64* %RAX_PTR, align 8
  ret %struct.Memory* %2
}
```

Saturn - Concretization



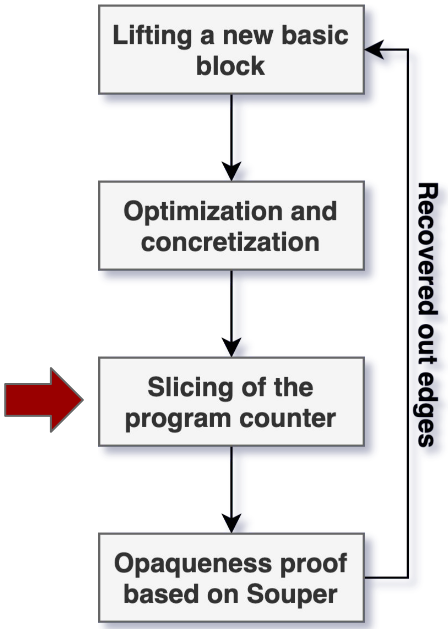
1. Simple no-op pattern

```
push   rax
pop    rax
```

2. Alias analysis now works and the no-op pattern gets optimized away

```
define dso_local %struct.Memory* @stub_0(%struct.State*, i64, %struct.Memory*) {
  ret %struct.Memory* %2
}
```

Saturn - Slicing



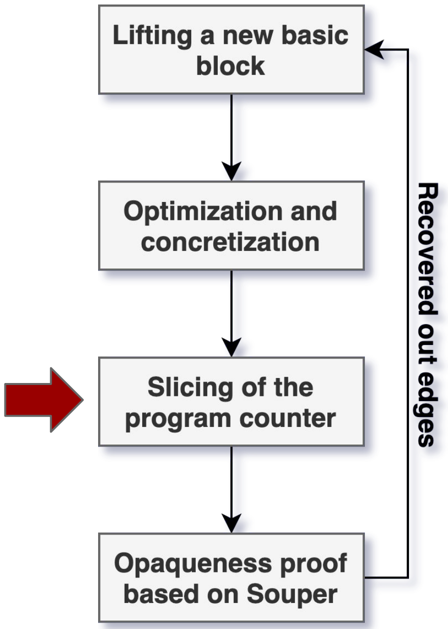
1. Sample program

```
mov rax, rbx
add rdx, rcx
sub rdx, rax
xor r8, rdx
shld rax, rax, 0x12
or r9, r15
add rax, 0x6720
mov rcx, 0x1872
imul r8
add r8, r11
sub rax, rcx
mov rbx, rax
```

2. Sliced register rbx

```
mov rax, rbx
shld rax, rax, 0x12
add rax, 0x6720
mov rcx, 0x1872
sub rax, rcx
mov rbx, rax
```

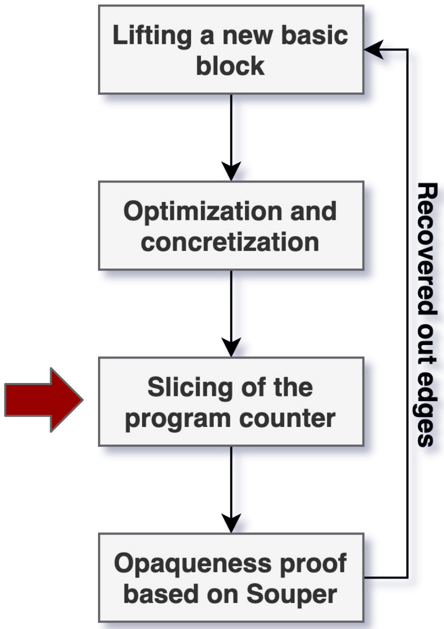
Saturn - Slicing - Problems



- Slicing looks simple but we had wrong results while trying open source slicers that rely on the LLVM-IR
- Slicing is crucial to retrieve only the parts of the code that we are interested in
- We had to come up with a simple and reliable way to slice the LLVM-IR

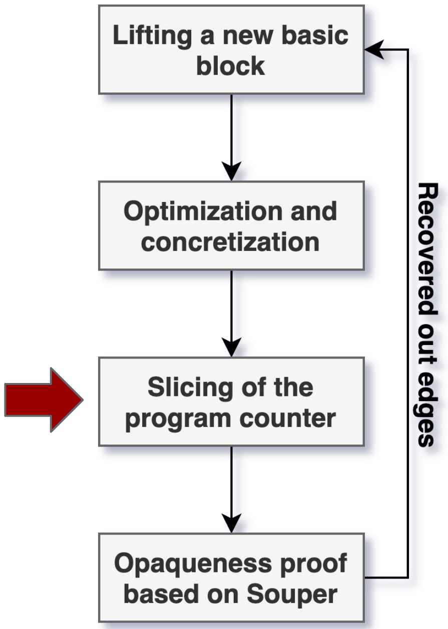
=> **Poor Man's Slicing**

Saturn - Poor Man's Slicing




```
extern "C" uint64_t __saturn_slice_rip(State state, addr_t curr_pc,
      ↪ Memory *memory, uint64_t *Stack) {
 // 1 Allocate a local Remill State structure and initialize it
 State S;
 S.gpr.rax.qword = state.gpr.rax.qword;
 ...
 S.gpr.rsp.qword = (uint64_t) Stack;
 S.gpr.r15.qword = state.gpr.r15.qword;
 S.aflag.af = state.aflag.af;
 ...
 S.aflag.zf = state.aflag.zf;
 // 2 Concretize RIP
 S.gpr.rip.qword = curr_pc;
 // 3/4 Call opaque basic block with initialized State struct
 // This function call will be replaced with the lifted one
 __remill_basic_block(S, curr_pc, memory);
 // 5 Inspect the value of RIP
 return S.gpr.rip.qword;
}
```

Saturn - Poor Man's Slicing



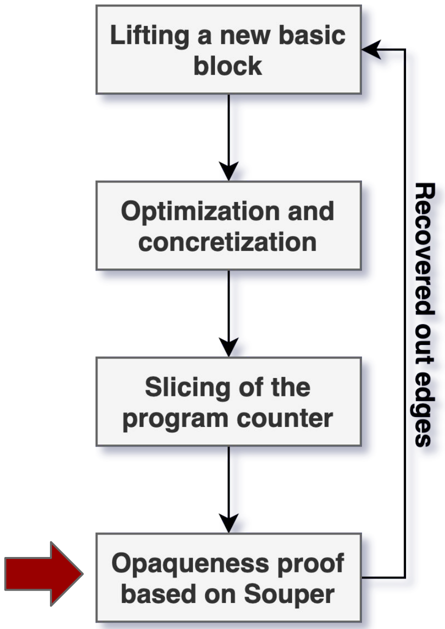
1. Opaque predicate

```
// MBA based opaque predicate
if ((chr + ch2) == ((chr ^ ch2) + 2 * (chr & ch2)))
  ch ^= 97;
else
  ch ^= 23;
```

2. Sliced opaque predicate

```
define i64 @__saturn_slice_rip(%struct.State.32* %state, i64 %
    ↪ curr_pc, %struct.Memory* %memory, i64* %Stack) #2 {
entry:
  %0 = getelementptr inbounds %struct.State.32, %struct.State.32* %
      ↪ state, i64 0, i32 6, i32 17, i32 0, i32 0
  %1 = load i64, i64* %0, align 8, !tbaa !9
  %2 = getelementptr inbounds %struct.State.32, %struct.State.32* %
      ↪ state, i64 0, i32 6, i32 19, i32 0, i32 0
  %3 = load i64, i64* %2, align 8, !tbaa !9
  %4 = shl i64 %1, 56
  %5 = ashr exact i64 %4, 56
  %6 = add i64 %5, %3
  %7 = xor i64 %3, %1
  %8 = shl i64 %7, 56
  %9 = ashr exact i64 %8, 56
  %10 = and i64 %3, %1
  %11 = shl i64 %10, 56
  %12 = ashr exact i64 %11, 55
  %13 = add nsw i64 %12, %9
  %14 = trunc i64 %6 to i32
  %15 = trunc i64 %13 to i32
  %16 = icmp eq i32 %14, %15
  %17 = select i1 %16, i64 5368713261, i64 5368713259
  ret i64 %17
}
```

Saturn - Opaqueness Proof

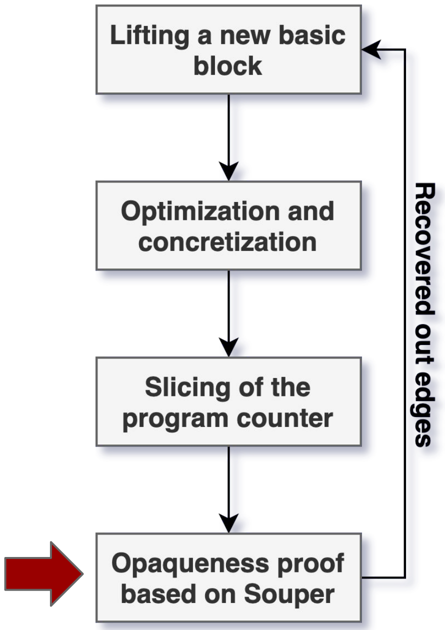


- Opaque predicates that are not resistant to compiler optimizations will directly fold into a constant

```
define dso_local i64 @__saturn_slice_rip(%struct.State*, i64, %
    ↪ struct.Memory*, i64*) {
entry:
 ret i64 5475437417 ; 0x1465C8B69
}
```

- Opaque predicates that are resistant, are going to be proven by an SMT solver
- Opaque predicates that are not provable will result in both paths being explored

Saturn - SMT Solving



1. Sliced opaque predicate

```
define i64 @__saturn_slice_rip(%struct.State.32* %state, i64 %
      ↪ curr_pc, %struct.Memory* %memory, i64* %Stack) #2 {
entry:
  %0 = getelementptr inbounds %struct.State.32, %struct.State.32* %
        ↪ state, i64 0, i32 6, i32 17, i32 0, i32 0
  %1 = load i64, i64* %0, align 8, !tbaa !9
  %2 = getelementptr inbounds %struct.State.32, %struct.State.32* %
        ↪ state, i64 0, i32 6, i32 19, i32 0, i32 0
  %3 = load i64, i64* %2, align 8, !tbaa !9
  %4 = shl i64 %1, 56
  %5 = ashr exact i64 %4, 56
  %6 = add i64 %5, %3
  %7 = xor i64 %3, %1
  %8 = shl i64 %7, 56
  %9 = ashr exact i64 %8, 56
  %10 = and i64 %3, %1
  %11 = shl i64 %10, 56
  %12 = ashr exact i64 %11, 55
  %13 = add nsw i64 %12, %9
  %14 = trunc i64 %6 to i32
  %15 = trunc i64 %13 to i32
  %16 = icmp eq i32 %14, %15
  %17 = select i1 %16, i64 5368713261, i64 5368713259
  ret i64 %17
}
```

2. Generated SMT formula

```
(set-logic QF_BV )
(declare-fun arr () (_ BitVec 8) )
(declare-fun arr0 () (_ BitVec 8) )
(assert (let ( (?B1 arr0 ) (?B2 arr ) ) (let ( (?B3 ((_ sign_extend
      ↪ 24) ?B1 ) ) (?B4 ((_ sign_extend 24) ?B2 ) ) (?B5 (bvand ?
      ↪ B2 ?B1 ) ) (?B6 (bvxor ?B2 ?B1 ) ) ) (let ( (?B11 ((_
      ↪ sign_extend 24) ?B6 ) ) (?B10 ((_ sign_extend 24) ?B5 ) )
      ↪ (?B8 (bvadd ?B4 ?B3 ) ) (?B9 (bvashr ?B4 (_ bv31 32) ) ) (?
      ↪ B7 (bvashr ?B3 (_ bv31 32) ) ) ) (let ( (?B14 ((_ extract 0
      ↪  0) ?B9 ) ) (?B12 ((_ extract 0 0) ?B7 ) ) (?B15 (bvshl ?
      ↪ B10 (_ bv1 32) ) ) (?B13 (bvashr ?B8 (_ bv31 32) ) ) (?B16
      ↪ (bvashr ?B11 (_ bv31 32) ) ) ) (let ( (?B17 ((_ extract 0
      ↪ 0) ?B13 ) ) (?B22 ((_ extract 0 0) ?B16 ) ) (?B19 (bvadd ?
      ↪ B15 ?B11 ) ) (?B20 (bvashr ?B15 (_ bv31 32) ) ) (?B21 (
      ↪ bvashr ?B15 (_ bv1 32) ) ) (?B18 (= ?B14 ?B12 ) ) ) (let (
      ↪ (?B27 ((_ extract 0 0) ?B20 ) ) (?B25 (bvashr ?B19 (_ bv31
      ↪ 32) ) ) (?B23 (= ?B17 ?B14 ) ) (?B28 (= ?B21 ?B10 ) ) (?B24
      ↪  (= false ?B18 ) ) (?B26 (= ?B19 ?B8 ) ) ) (let ( (?B31 (
      ↪ ite ?B26 (_ bv5368713423 64) (_ bv5368713442 64) ) ) (?B30
      ↪ ((_ extract 0 0) ?B25 ) ) (?B29 (or ?B24 ?B23 ) ) (?B32 (=
      ↪ ?B27 ?B22 ) ) ) (let ( (?B35 (= false ?B32 ) ) (?B33 (= ?
      ↪ B30 ?B27 ) ) (?B34 (= (_ bv5368713423 64) ?B31 ) ) ) (let (
      ↪  (?B36 (or ?B35 ?B33 ) ) ) (let ( (?B37 (and ?B36 ?B28 ) )
      ↪ ) (let ( (?B38 (and ?B37 ?B29 ) ) ) (let ( (?B39 (= false ?
      ↪ B38 ) ) ) (let ( (?B40 (or ?B39 ?B34 ) ) ) (and ?B38 (=
      ↪ false ?B34 ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) )
(check-sat)
(exit)
```

Saturn - Brightening

1. Recovered LLVM-IR

```
define dllexport i64 @F_140001000(%struct.State.32* %S, i64 %curr_pc,
    ↪  %struct.Memory.0* %memory) {
entry:
  %0 = getelementptr inbounds %struct.State.32, %struct.State.32* %S,
      ↪  i64 0, i32 6, i32 33, i32 0, i32 0
  %1 = getelementptr inbounds %struct.State.32, %struct.State.32* %S,
      ↪  i64 0, i32 13
  store i8 0, i8* %1, align 1
  %2 = getelementptr inbounds %struct.State.32, %struct.State.32* %S,
      ↪  i64 0, i32 6, i32 5, i32 0
  %3 = bitcast %union.anon.2* %2 to i8*
  %4 = getelementptr inbounds %struct.State.32, %struct.State.32* %S,
      ↪  i64 0, i32 6, i32 17, i32 0
  %5 = bitcast %union.anon.2* %4 to i8*
  %6 = load i8, i8* %5, align 1
  %7 = load i8, i8* %3, align 1
  store i64 5368713251, i64* %0, align 8
  %8 = sext i8 %7 to i64
  %phitmp = icmp eq i8 %7, 126
  %9 = getelementptr inbounds %struct.State.32, %struct.State.32* %S,
      ↪  i64 0, i32 6, i32 1, i32 0, i32 0
  %10 = getelementptr inbounds %struct.State.32, %struct.State.32* %
      ↪ S, i64 0, i32 6, i32 5, i32 0, i32 0
  %11 = sext i8 %6 to i64
  %12 = and i64 %11, 4294967295
  store i64 5368713372, i64* %0, align 8
  %13 = getelementptr inbounds %struct.State.32, %struct.State.32* %
      ↪ S, i64 0, i32 6, i32 7, i32 0, i32 0
  %14 = xor i8 %7, %6
  %15 = sext i8 %14 to i64
  %16 = getelementptr inbounds %struct.State.32, %struct.State.32* %
      ↪ S, i64 0, i32 6, i32 17, i32 0, i32 0
  store i64 %12, i64* %16, align 8
  %17 = and i64 %12, %8
  %18 = shl nuw nsw i64 %17, 1
  %19 = and i64 %18, 4294967294
  store i64 %19, i64* %13, align 8
  %20 = add nsw i64 %18, %15
  %21 = and i64 %20, 4294967295
  store i64 %21, i64* %10, align 8
  store i8 0, i8* %1, align 1
  %22 = zext i1 %phitmp to i8
  store i8 %22, i8* %3, align 1
  %23 = zext i1 %phitmp to i64
  store i64 %23, i64* %9, align 8
  ret i64 %23
}
```

We start to detect the amount of function arguments based on the binary ABI and we use them to model a helper function

```
extern "C" Memory * F_Lifted(State &state, addr_t curr_pc, Memory *
    ↪ memory);
extern "C" uint64_t x64_MS_2_ARG(uint64_t *RCX, uint64_t *RDX) {
 struct State S;
 // Set 1. arg
 S.gpr.rcx.qword = (uint64_t) RCX;
 // Set 2. arg
 S.gpr.rdx.qword = (uint64_t) RDX;
 // Call lifted function which will be replaced and inlined
 F_Lifted(S, 0, nullptr);
 // Return result
 return S.gpr.rax.qword;
}
```

Saturn - Brightening

1. Recovered LLVM-IR

```
define dllexport i64 @F_140001000(%struct.State.32* %S, i64 %curr_pc,
    ↪  %struct.Memory.0* %memory) {
entry:
  %0 = getelementptr inbounds %struct.State.32, %struct.State.32* %S,
      ↪  i64 0, i32 6, i32 33, i32 0, i32 0
  %1 = getelementptr inbounds %struct.State.32, %struct.State.32* %S,
      ↪  i64 0, i32 13
  store i8 0, i8* %1, align 1
  %2 = getelementptr inbounds %struct.State.32, %struct.State.32* %S,
      ↪  i64 0, i32 6, i32 5, i32 0
  %3 = bitcast %union.anon.2* %2 to i8*
  %4 = getelementptr inbounds %struct.State.32, %struct.State.32* %S,
      ↪  i64 0, i32 6, i32 17, i32 0
  %5 = bitcast %union.anon.2* %4 to i8*
  %6 = load i8, i8* %5, align 1
  %7 = load i8, i8* %3, align 1
  store i64 5368713251, i64* %0, align 8
  %8 = sext i8 %7 to i64
  %phitmp = icmp eq i8 %7, 126
  %9 = getelementptr inbounds %struct.State.32, %struct.State.32* %S,
      ↪  i64 0, i32 6, i32 1, i32 0, i32 0
  %10 = getelementptr inbounds %struct.State.32, %struct.State.32* %
      ↪ S, i64 0, i32 6, i32 5, i32 0, i32 0
  %11 = sext i8 %6 to i64
  %12 = and i64 %11, 4294967295
  store i64 5368713372, i64* %0, align 8
  %13 = getelementptr inbounds %struct.State.32, %struct.State.32* %
      ↪ S, i64 0, i32 6, i32 7, i32 0, i32 0
  %14 = xor i8 %7, %6
  %15 = sext i8 %14 to i64
  %16 = getelementptr inbounds %struct.State.32, %struct.State.32* %
      ↪ S, i64 0, i32 6, i32 17, i32 0, i32 0
  store i64 %12, i64* %16, align 8
  %17 = and i64 %12, %8
  %18 = shl nuw nsw i64 %17, 1
  %19 = and i64 %18, 4294967294
  store i64 %19, i64* %13, align 8
  %20 = add nsw i64 %18, %15
  %21 = and i64 %20, 4294967295
  store i64 %21, i64* %10, align 8
  store i8 0, i8* %1, align 1
  %22 = zext i1 %phitmp to i8
  store i8 %22, i8* %3, align 1
  %23 = zext i1 %phitmp to i64
  store i64 %23, i64* %9, align 8
  ret i64 %23
}
```

2. Expected deobfuscated result

```
define dso_local i32 @func(i8 signext) local_unnamed_addr #0 {
  %2 = icmp eq i8 %0, 126
  %3 = zext i1 %2 to i32
  ret i32 %3
}
```

3. LLVM-IR after the brightening step

```
define dllexport i64 @F_140001000_args(i64* %RCX, i64* %RDX, i64* %
    ↪ R8) {
entry:
  %0 = ptrtoint i64* %RCX to i64
  %1 = trunc i64 %0 to i8
  %2 = icmp eq i8 %1, 126
  %3 = zext i1 %2 to i64
  ret i64 %3
}
```

Saturn - Recompilation

We can easily recompile the lifted code by using one of the available LLVM backends

1. Deobfuscated code

```
define dllexport i64 @F_140001000_args(i64* %RCX, i64* %RDX, i64* %
    ↪ R8) {
entry:
  %0 = ptrtoint i64* %RCX to i64
  %1 = trunc i64 %0 to i8
  %2 = icmp eq i8 %1, 126
  %3 = zext i1 %2 to i64
  ret i64 %3
}
```

2. x86-64

```
push   rbp
mov    rbp, rsp
xor    eax, eax
cmp    dil, 126
sete    al
pop    rbp
ret
```

3. AArch64

```
and    w8, w0, #0xff
cmp    w8, #126
cset   w0, eq
ret
```

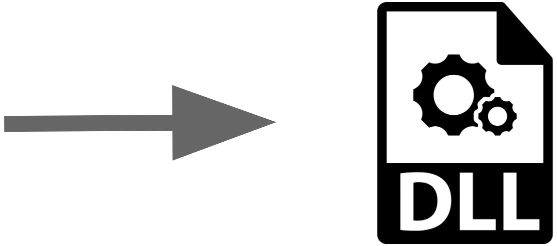
4. RiscV-64

```
addi    sp, sp, -16
sd      ra, 8(sp)
sd      s0, 0(sp)
addi    s0, sp, 16
andi    a0, a0, 255
xori    a0, a0, 126
seqz    a0, a0
ld      s0, 0(sp)
ld      ra, 8(sp)
addi    sp, sp, 16
ret
```

Saturn - Execution

1. We compile the recovered LLVM-IR into a shared library
2. We inject the shared library into the original binary
3. We redirect the original function into our shared library

```
push   rbp
mov    rbp, rsp
xor    eax, eax
cmp    dil, 126
sete    al
pop    rbp
ret
```



If the original function signature detection fails, the recovered LLVM-IR is still going to rely on the ***State*** structure and ***SATURN*** provides a context switch injection step to guarantee a proper patching of the original binary.

Saturn - Testing

- During the development we tested ***SATURN*** against several real world protectors:
  - *Tigress* (implements many state of the art obfuscation tricks)
  - *O-LLVM* (constant unfolding, junk code, opaque predicates)
  - *Themida* (patterns, constant unfolding, opaque predicates)
  - *Denuvo* (patterns, constant unfolding, opaque predicates)
  - *VMProtect* (junk code)
  - *SecuROM* (patterns)
- We ran into edge cases and errors, so we relied on symbolic deobfuscation as a mean to double-check the results obtained by ***SATURN***
- Some of the mentioned protectors are relying on high-level obfuscation tricks that need a custom handling or a manual guidance from the reverser (e.g. virtual machine obfuscation)

Saturn - Limitations

- **Unimplemented or bugged opcodes**: *Remill* has a great opcodes coverage, but some are still missing and are used by real protectors (e.g. FXSAVE, FXRSTOR). During the development we also spotted wrong behaviours (e.g. POP RSP) that have been promptly fixed
- **Switch-cases**: SMT-based range analysis is helpful, but ad-hoc switch-table parsing may be needed to overcome some sparse switch-case implementations
- **Anti-DSE tricks**: hardened versions of the FOR and SPLIT tricks are currently non optimizable away by *LLVM*
- **MBA opaque predicates**: opaque predicates based on strong MBA expressions or unprovable formulas may lose the precision of the exploration phase

Saturn - Improvements

- **Plugin system**: to be able to implement custom analysis and optimization passes (e.g. using Drill&Join against the MBA expressions)
- **Stack propagation**: the stack propagation is currently based on the concretization of the stack pointer, but a fully symbolic version is in development
- **Aarch64 support**: given that *Remill* supports *aarch64* opcodes it would be ideal to integrate it to extend the support to Android and iOS native libraries
- **Exploration strategies**: to be able to adopt different exploration strategies based on the obfuscation used by the target (e.g. a custom version of Microsoft's SAGE)

Demo - Vector instructions

1. Obfuscated code

```
1: 1: 0x140001000: sub rsp, 0x38
1: 2: 0x140001004: mov dword ptr [rsp + 0x34], edx
1: 3: 0x140001008: mov dword ptr [rsp + 0x30], ecx
1: 4: 0x14000100c: movsxd rax, dword ptr [rsp + 0x30]
1: 5: 0x140001011: movsxd r8, dword ptr [rsp + 0x34]
1: 6: 0x140001016: movq xmm0, r8
1: 7: 0x14000101b: movq xmm1, rax
1: 8: 0x140001020: punpcklqdq xmm1, xmm0
1: 9: 0x140001024: movaps xmmword ptr [rsp + 0x20], xmm1
1: 10: 0x140001029: movsxd rax, dword ptr [rsp + 0x34]
1: 11: 0x14000102e: movsxd r8, dword ptr [rsp + 0x30]
1: 12: 0x140001033: movq xmm0, r8
1: 13: 0x140001038: movq xmm1, rax
1: 14: 0x14000103d: punpcklqdq xmm1, xmm0
1: 15: 0x140001041: movaps xmmword ptr [rsp + 0x10], xmm1
1: 16: 0x140001046: movaps xmm0, xmmword ptr [rsp + 0x20]
1: 17: 0x14000104b: movaps xmm1, xmmword ptr [rsp + 0x10]
1: 18: 0x140001050: movaps xmm2, xmm0
1: 19: 0x140001053: paddq xmm2, xmm1
1: 20: 0x140001057: movaps xmm3, xmm0
1: 21: 0x14000105a: psrlq xmm3, 0x20
1: 22: 0x14000105f: pmuludq xmm3, xmm1
1: 23: 0x140001063: movaps xmm4, xmm1
1: 24: 0x140001066: psrlq xmm4, 0x20
1: 25: 0x14000106b: movaps xmm5, xmm0
1: 26: 0x14000106e: pmuludq xmm5, xmm4
1: 27: 0x140001072: paddq xmm5, xmm3
1: 28: 0x140001076: psllq xmm5, 0x20
1: 29: 0x14000107b: pmuludq xmm0, xmm1
1: 30: 0x14000107f: paddq xmm0, xmm5
1: 31: 0x140001083: movaps xmm1, xmm2
1: 32: 0x140001086: psrlq xmm1, 0x20
1: 33: 0x14000108b: pmuludq xmm1, xmm0
1: 34: 0x14000108f: movaps xmm3, xmm0
1: 35: 0x140001092: psrlq xmm3, 0x20
1: 36: 0x140001097: movaps xmm4, xmm2
1: 37: 0x14000109a: pmuludq xmm4, xmm3
1: 38: 0x14000109e: paddq xmm4, xmm1
1: 39: 0x1400010a2: psllq xmm4, 0x20
1: 40: 0x1400010a7: pmuludq xmm2, xmm0
1: 41: 0x1400010ab: paddq xmm2, xmm4
1: 42: 0x1400010af: movaps xmmword ptr [rsp], xmm2
1: 43: 0x1400010b3: mov rax, qword ptr [rsp]
1: 44: 0x1400010b7: mov ecx, eax
1: 45: 0x1400010b9: mov eax, ecx
1: 46: 0x1400010bb: add rsp, 0x38
1: 47: 0x1400010bf: ret
```

2. Deobfuscated LLVM-IR

```
define dso_local i64 @F_140001000_args(i8* %RCX, i8* %RDX) {
entry:
  %0 = ptrtoint i8* %RCX to i64
  %1 = ptrtoint i8* %RDX to i64
  %2 = and i64 %0, 4294967295
  %3 = and i64 %1, 4294967295
  %4 = add i64 %1, %0
  %5 = mul nuw i64 %3, %2
  %6 = and i64 %5, 4294967295
  %7 = and i64 %4, 4294967295
  %8 = mul nuw i64 %6, %7
  %9 = and i64 %8, 4294967295
  ret i64 %9
}
```

3. Recompiled LLVM-IR

```
lea     eax, [rdx + rcx]
imul     edx, ecx
imul     eax, edx
ret
```

Conclusion

- We would like to thank everyone who worked on *LLVM*, *Remill* and *Souper*, without these tools ***SATURN*** wouldn't exist!
- Last but not least, remember that ***generic approaches can only produce generic results***, so ***SATURN*** is a step towards a generic approach that relies on strong optimizations implemented in *LLVM* and *Souper*, but also gives the user a high flexibility in the implementation of unavoidable custom deobfuscation passes.

Questions?!

Thanks!

```
   /\
  (  )
  (  )
  (  )
 /|/\|\
/_||||_\
  /__\
 (*#*)
  (*#)
   (#*)
   (*)
```