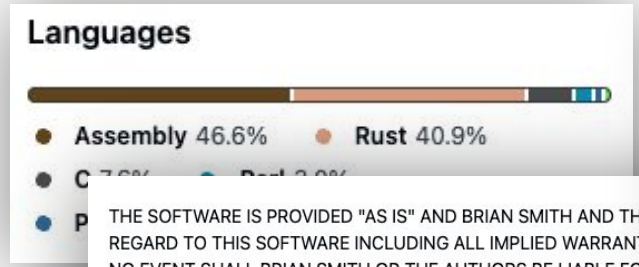
From Rust Till Run:
Extending Memory Safety From Rust To Cryptographic Assembly

Shai Caspin, Nikhil Pimpalkhare, and Amit Levy

PRINCETON
UNIVERSITY

Practical cryptography uses assembly

Ring



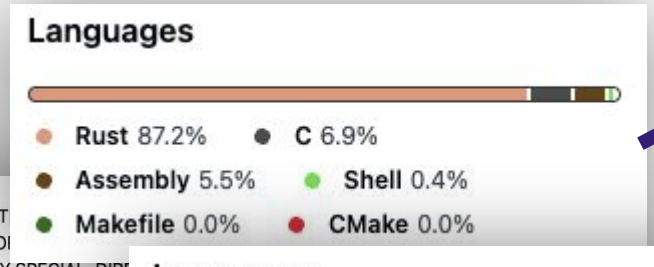
THE SOFTWARE IS PROVIDED "AS IS" AND BRIAN SMITH AND THE AUT
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES O
NO EVENT SHALL BRIAN SMITH OR THE AUTHORS BE LIABLE FOR ANY SPECIAL, DIRE
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOS
PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOU
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

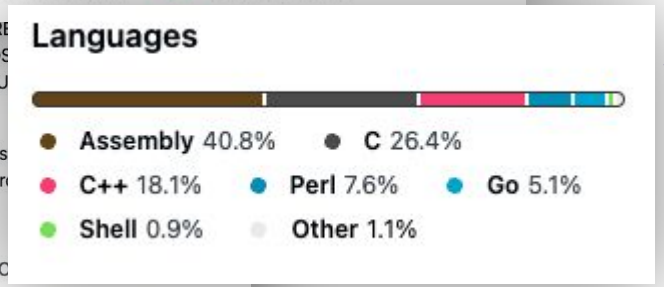
Most of the C and assembly language code in *ring* comes from BoringSSL. BoringSSL is
quote from the BoringSSL README.md discouraging you from using it applies to this pr

> BoringSSL is a fork of OpenSSL that is designed to meet Google's needs.
>
> Although BoringSSL is an open source project, it is not intended for general use, as C
> recommend that third parties depend upon it.

This project was originally shared on GitHub in 2015 as an experiment. It was put on crates.io shortly to help other people with their experiments. It is an experiment.

AWS LibCrypto (-rs)



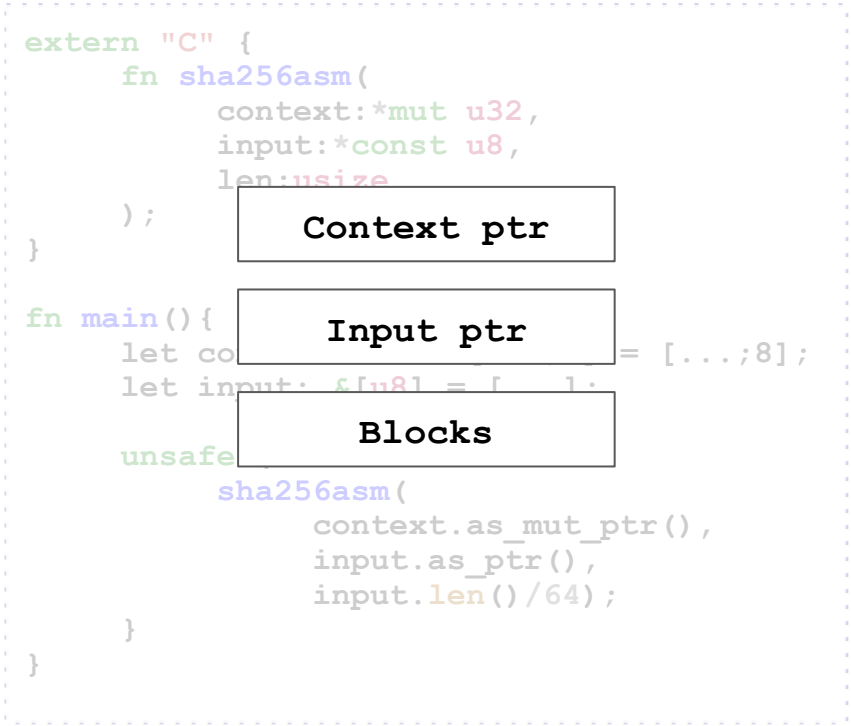


Unsafe FFI allows users to introduce bugs

```
extern "C" {
      fn sha256asm(
            context:*mut u32,
            input:*const u8,
            len:usize
      );
}

fn main(){
      let context: &mut [u32;8] = [...;8];
      let input: &[u8] = [...];

      unsafe {
            sha256asm(
                  context.as_mut_ptr(),
                  input.as_ptr(),
                  input.len()/64);
      }
}
```

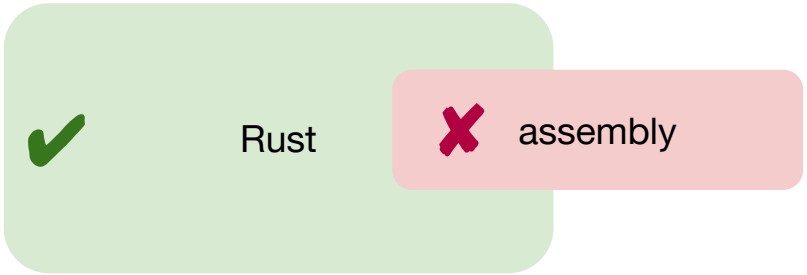
```
Sha256:
      Read context
      // calculate end of input
      add     x2,x1,x2,lsl#6

Loop:
      Read a block and hash it
      Increment pointer x1 by block size
      cmp     x1,x2
      Store results in context
      b.ne    Loop
ret
```

Unsafe FFI allows users to introduce bugs




```
Sha256:
    Read context
    // calculate end of input
    add     x2,x1,x2,lsl#6

Loop:                          ✘ oob – seg fault
    Read a block and hash it
    Increment pointer x1 by block size
    cmp     x1,x2              ✘ bad perm
    Store results in context
    b.ne    Loop
ret ✘ correctness errors
```

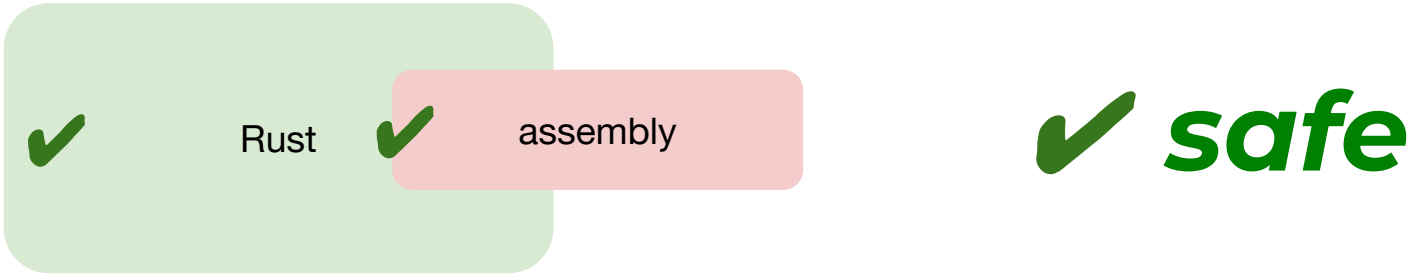
Memory Safety of Rust + Assembly



✘ unsafe

All pointer dereferences point to valid, allocated, well-aligned, and accessible memory

Memory Safety of Rust + Assembly



➔ ***Assembly cannot violate Rust memory safety***

➔ ***Rust types match expected assembly inputs***

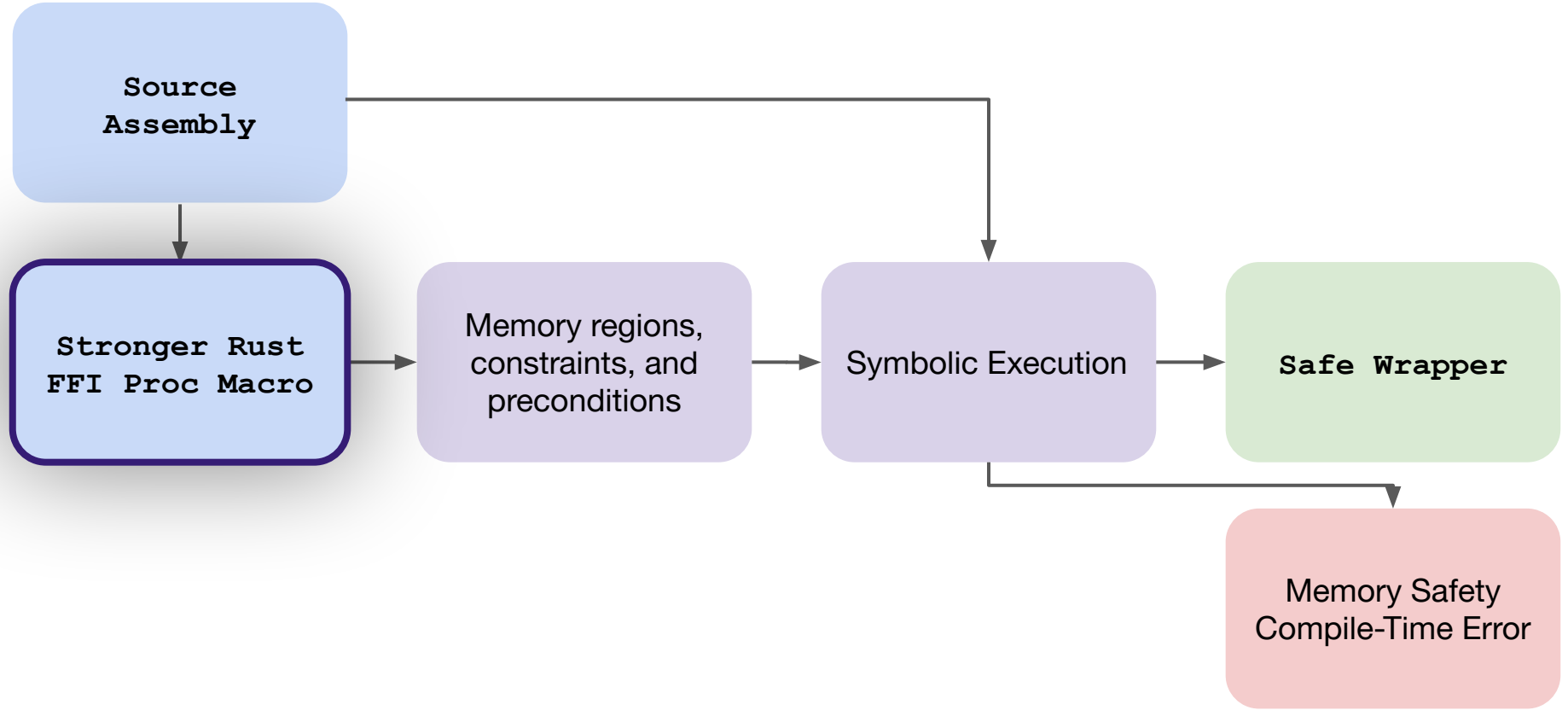
CLAMS: ***C****hecking* ***L****inked* ***A****ssembly for* ***M****emory* ***S****afety*

Strengthening the FFI
with ***preconditions***
enables ***automatic***
memory safety verification

Strengthening the FFI with ***preconditions*** enables ***automatic*** memory safety verification*

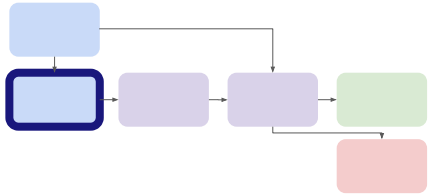
Specializing for common cryptography control flow and memory patterns

Inputs: Automatic: Output:



Stronger FFI

A procedural macro transforms Rust code to Rust code before compilation



Usage:

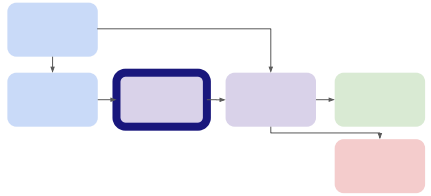
```
#[clams(filename, parameter remappings, [preconditions])]
fn function_name(parameters: . . . ) -> return_value;
```

✔ ***connect types***

(ex: a buffer is passed as a pointer and length)

✔ ***refine types***

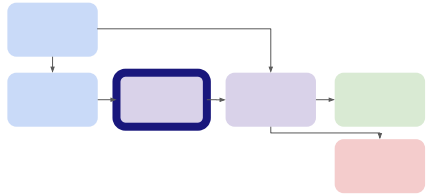
(ex: a length will be in block size multiples)



Memory Safe Regions

```
#[clams("sha256.S",
    context.as_mut_ptr(), input.as_ptr(), input.len()/64,
          [input.len()>=64, input.len()%64==0])]
fn sha256asm(context: &mut [u32;8], input: &[u8]);
```

context	input
Writable	Readable
256 bits	input_len



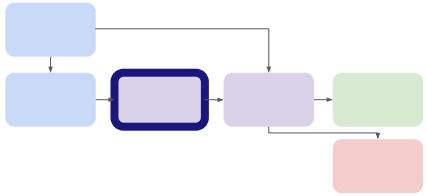
Parameter Remapping

```
#[clams("sha256.S",
    context.as_mut_ptr(), input.as_ptr(), input.len()/64,
          [input.len()>=64, input.len()%64==0])]
fn sha256asm(context: &mut [u32;8], input: &[u8]);
```

x0	context_mut_ptr
x1	input_ptr
x2	input_len/64

context
Writable
256 bits

input
Readable
input_len



Preconditions

```
#[clams("sha256.S",
    context.as_mut_ptr(), input.as_ptr(), input.len()/64,
          [input.len()>=64, input.len()%64==0])]
fn sha256asm(context: &mut [u32;8], input: &[u8]);
```

x0	context_mut_ptr
x1	input_ptr
x2	input_len/64

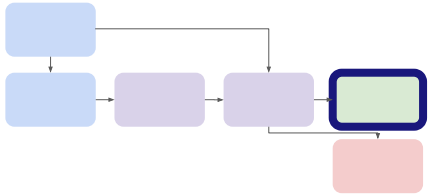
context
Writable
256 bits

input
Readable
input_len

Length is unknown @compile

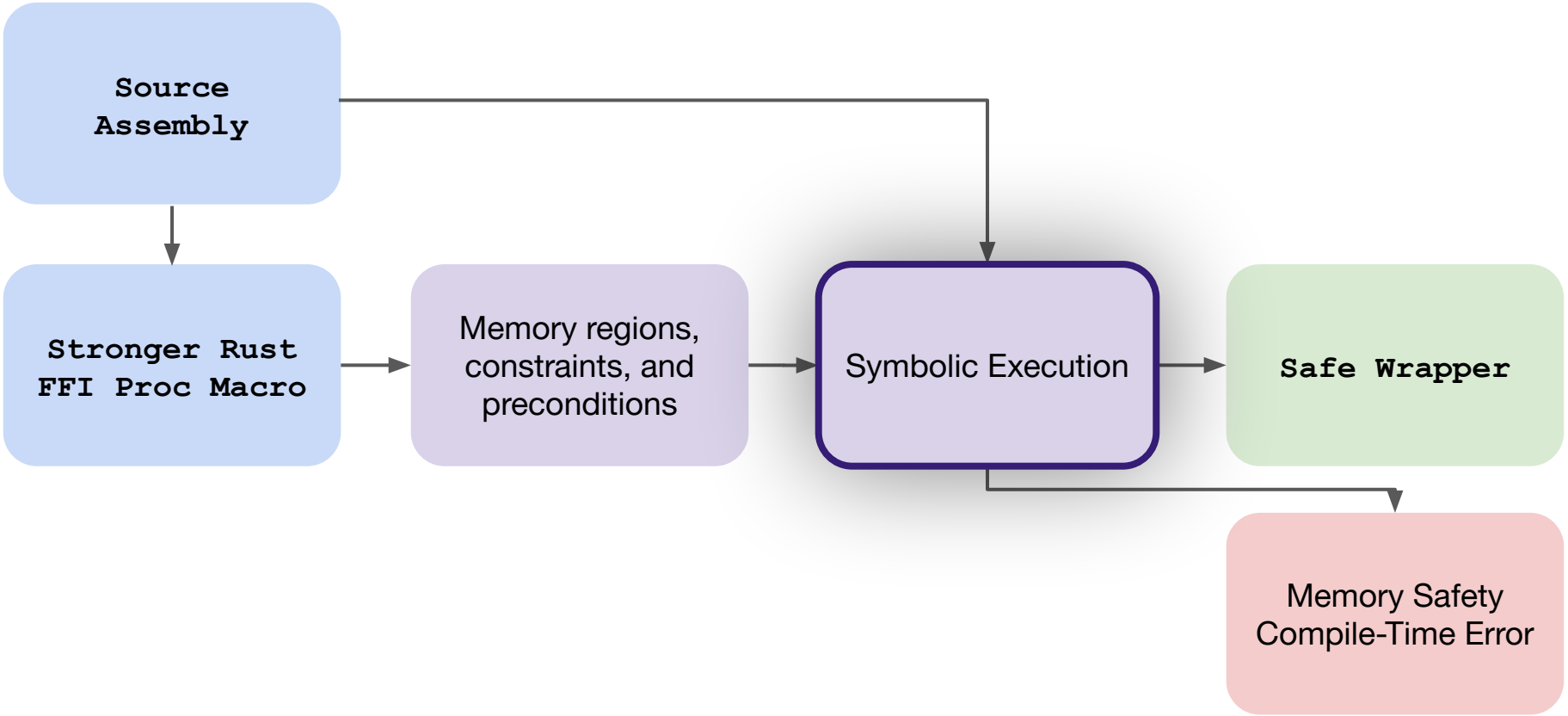
```
input_len >= 64
input_len%64==0
```

Assembly is always called correctly

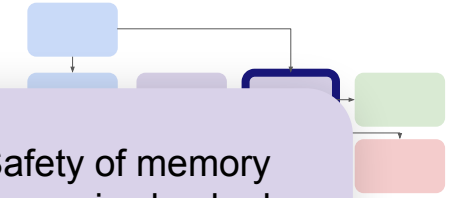


```
extern "C" {
    #[link_name=sha256asm]
    fn sha256asm_extern(context:*mut u32,input:*const u8,
    input_len:usize);
}

fn sha256asm(context: &mut [u32;8], input: &[u8]){
    assert!(input.len()>=64);
    assert!(input.len()%64==0);

    unsafe {
        return sha256asm_extern(
    context.as_mut_ptr(),input.as_ptr(),input.len()/64); }
}
```

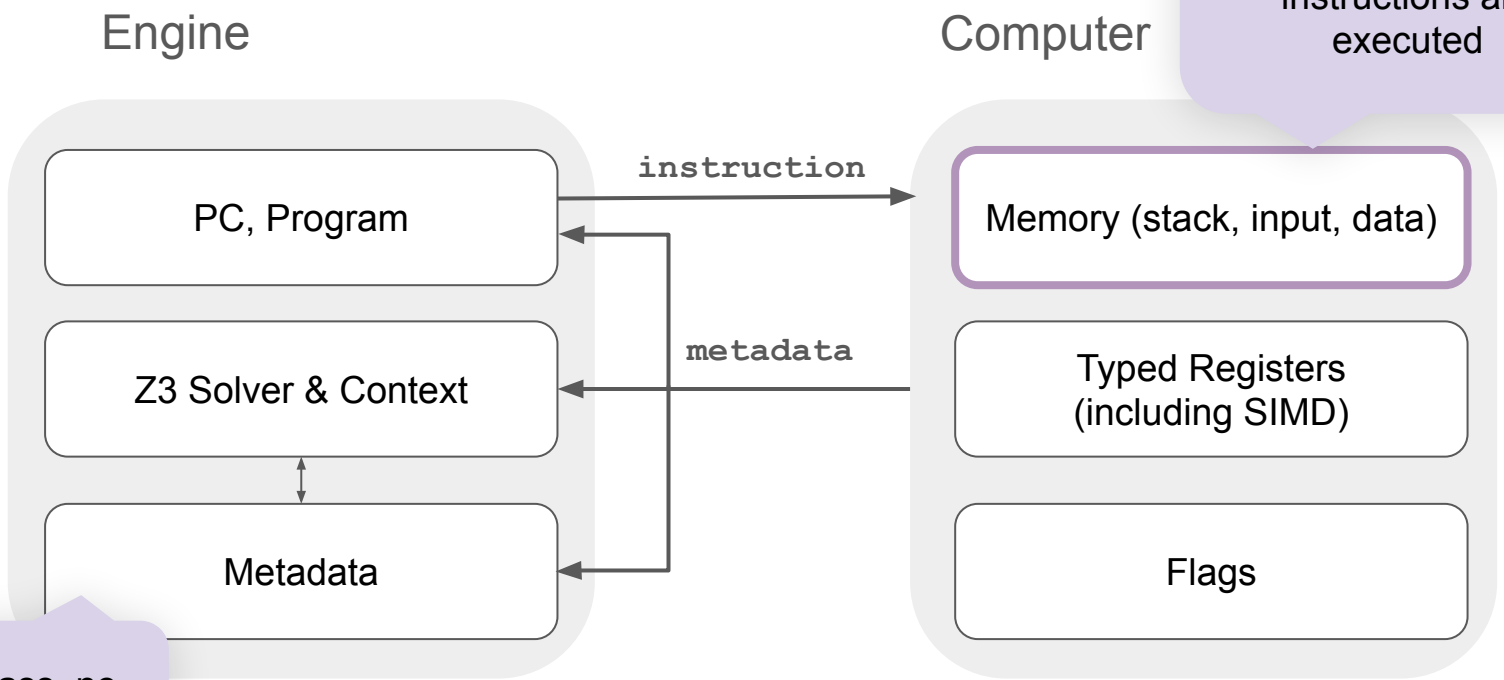
Inputs: Automatic: Output:



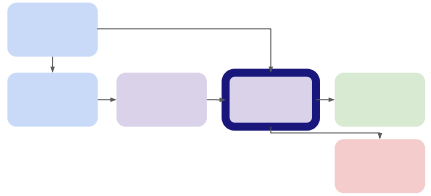
Symbolic Execution



Safety of memory accesses is checked as instructions are executed



One-pass, no branch merging



Memory Accesses w/Symbolic Length

input
Readable
input_len

read	input_ptr	8	32	→ ?

Possibly out of bounds?

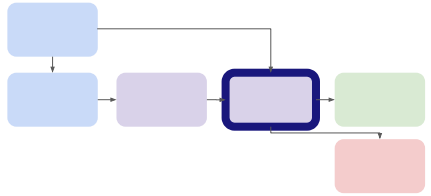
$8 \geq 0$

8 + 32 bits $\leq$ `input_len`?

Is 40 > `input_len` satisfiable? → **No** / **Yes**

Need condition to bound `input_len` > 40

Is access well-aligned?

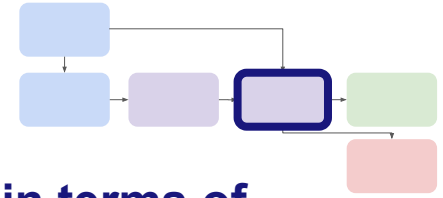


Loop over input buffers

		x1	
read	input_ptr	0	word
read	input_ptr	16	word
read	input_ptr	32	word
read	input_ptr	496	word

Cannot explore loops infinitely

Use first loop iterations to derive loop step invariant



Induction over safety

Base case

read	input_ptr	0	word
read	input_ptr	16	word
read	input_ptr	32	word

read	input_ptr	496	word

`input.len()>=64`

⬇️

First iteration is safe

Rewrite state in terms of variable `k` to model iter

512*k
512*k + 16
512*k + 32

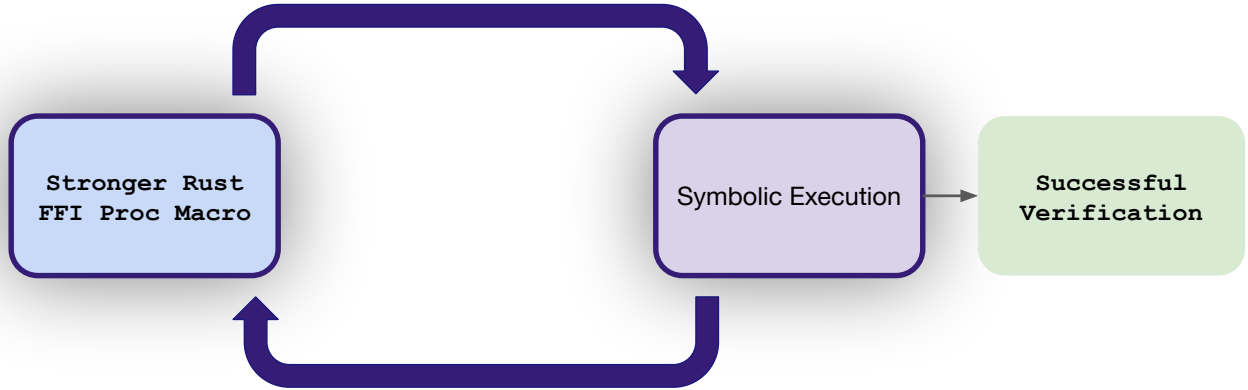
512*k + 496

`input.len()%64==0`

⬇️

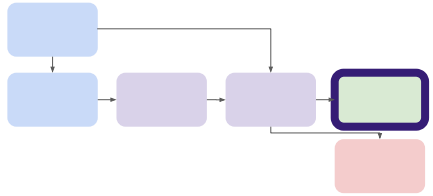
input_len ∝ k

Preconditions and Verification Interplay



➔ ***Preconditions are easy to define***

➔ ***Preconditions simplify verification***

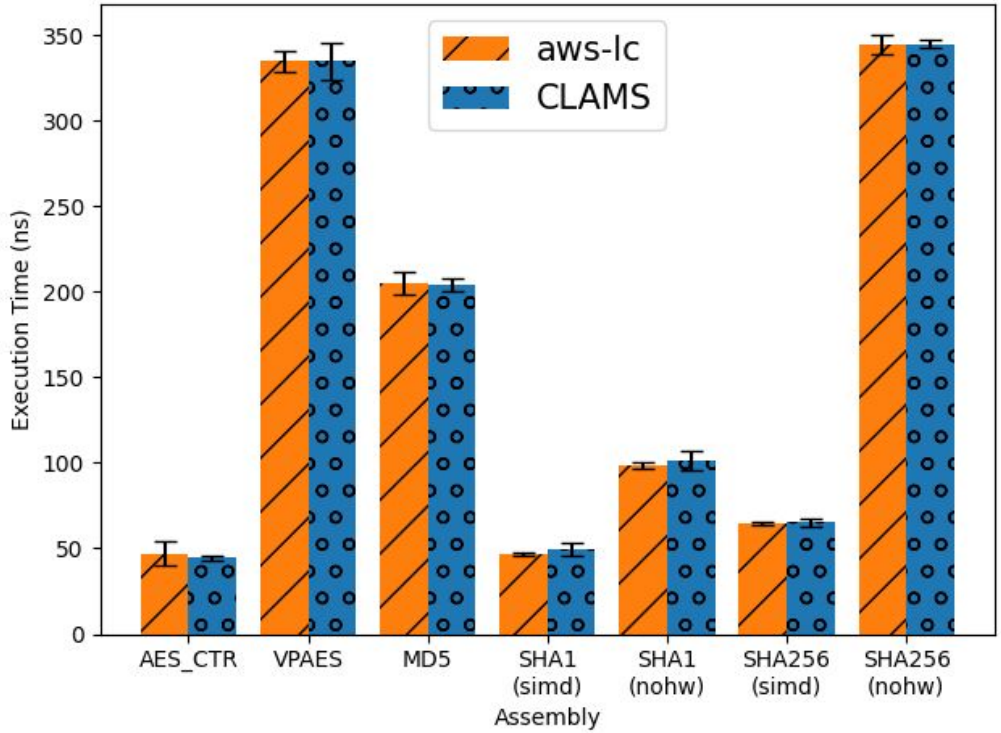


CLAMS is able to verify many algorithms

Function family	# functions	Param remap	Struct to tuple	Input length preconditions	Other preconditions
Hashes	7	✔	✘	✔	✘
AES	5	✔	✔	✔	✔
GHASH	3	✔	✘	✔	✘
Keccak/SHA-3	3	✔	✘	✔	✔
Bignum	2	✔	✘	✔	✔
ChaCha	3	✔	✘	✘	✘

Verification succeeds in under 100ms in all cases

Checking preconditions is not prohibitive



On average 0.2% runtime overhead, not statistically significant

CLAMS

Strengthening the FFI with preconditions

✔ Strong procedural macro interface

enables automatic memory safety verification

✔ Quick compile-time symbolic execution

This work was funded in part by the National Science Foundation under Grants No. 2303639, 2443589, and a gift from the Internet Security Research Group. Shai Caspin is supported by an NSF GRFP No. 2039656.
Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.