galois

FROMAGER: A scalable toolchain for complex ZK Proofs about software

Alex Malozemoff, *James Parker* (james@galois.com), David Archer

Galois, Inc.

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001120C0085. 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 Defense Advanced Research Projects Agency (DARPA). Approved for Public Release, Distribution Unlimited.

Research focus

Proving properties about software in ZK, efficiently

By properties, we mean:

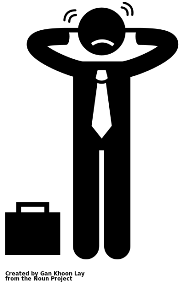
- Prove **vulnerability** exists without revealing **trigger**
- Prove **safety** of code without revealing its source

By efficiently, we mean:

- Prover **and** verifier efficiency, **minimal** communication
- Handle very large circuits (billions of gates)

Should you disclose software vulnerabilities?

Challenges of vulnerability disclosure



Vendors

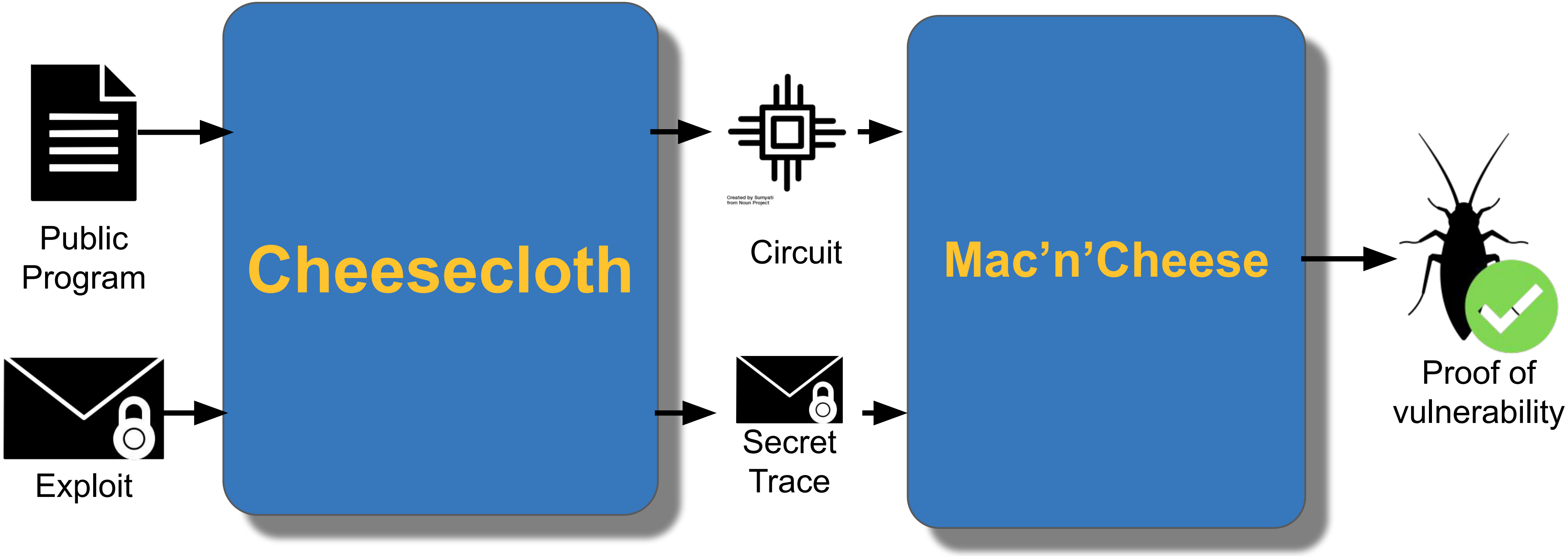
Bad actors

Users

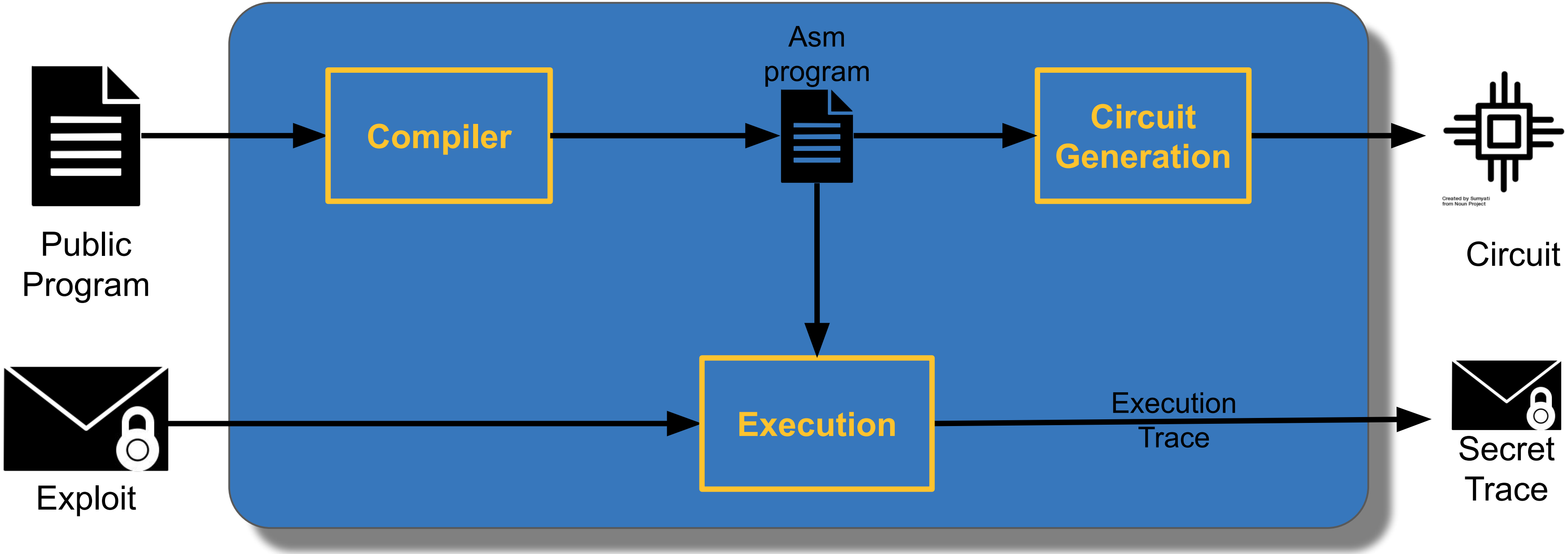

Public
Program
Exploit
Fromager
(Cheesecloth + Mac’n’Cheese)
Proof of
vulnerability

Fromager Demonstrations

Vulnerability class	**Example Program**	**Trace length**	**Gates**
Buffer overflow	GRIT 	6K	27M
Out-of-bounds array access	FFmpeg (CVE-2013-0864)	80K	673M
Information Leakage	OpenSSL (Heartbleed)	25,000K	17B
Cryptographic protocol bug	Scuttlebutt	WIP	WIP


Public
Program
Exploit
Cheesecloth
Circuit
Secret
Trace
Mac’n’Cheese
Proof of
vulnerability

Cheesecloth - The “Front end”



Components of the Circuit

- **For each instruction in the program**
  - A circuit that checks whether the CPU state transition was “legal”
- **For each memory reference**
  - A circuit that verifies reads correctly reflect most recent writes
- **For all memory references**
  - A circuit that verifies the ordering used in the above circuit is a correct permutation of the actual execution order

Components of the Circuit

$$\text{Trace} = [\text{state}_0, \text{state}_1, \ldots, \text{state}_T]$$

state_i = { registers, program counter, bug_flag}

- State transition function

$ST(m_i, \text{state}_i) = \text{state}_{i+1}$

- Memory consistency

$MC(m_1, m_2, \ldots m_T) = \text{True}$

State transition function

$ST(m_i, state_i) = state_{i+1}$

```
fn transition(current_st) -> State {
    let instr = fetch_instr(current_st.pc);
    let arg1 = index(current_st.regs, instr.op1);
    let arg2 = index(current_st.regs, instr.op2);

    let result = mux(instr.opcode == XOR, xor(arg1, arg2),
                     instr.opcode == ADD, add(arg1, arg2),
                     ...);
    ...
}
```

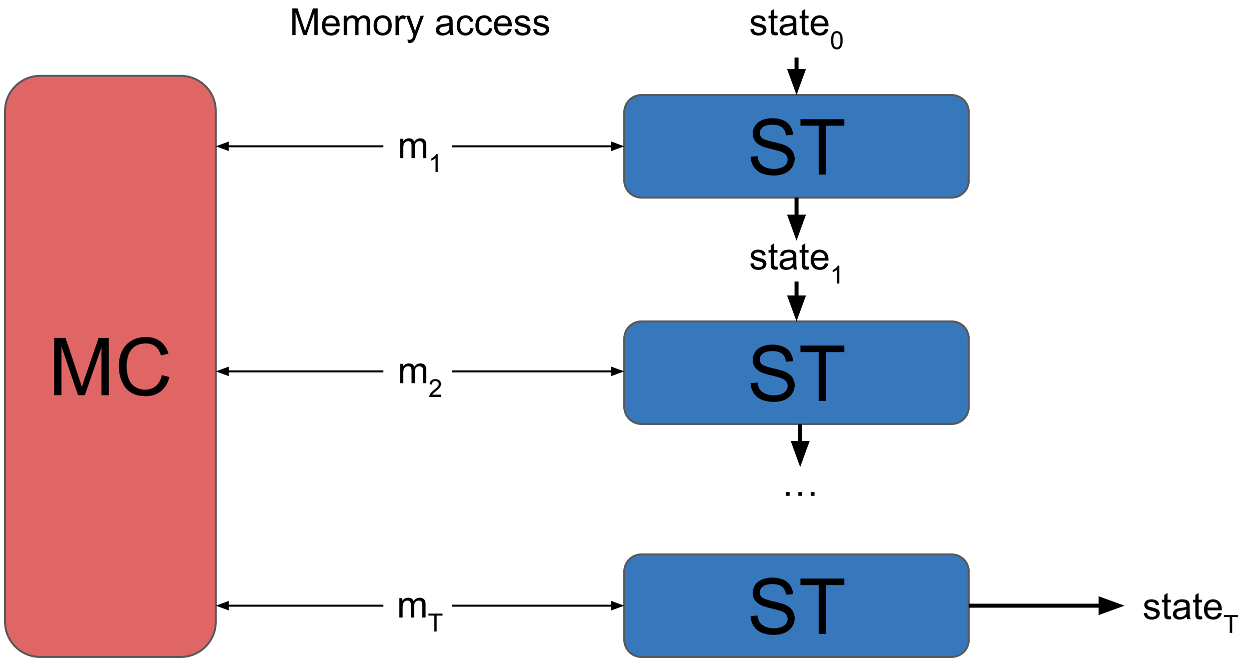
Memory Consistency Check

$MC(m_1, m_2, \ldots m_T) = \text{True}$

1. Sort memory operations by address and cycle
   a. Check that reads match previous write
2. Check that memory operations are a permutation of the execution trace

WRITE	0xdeadbeef	@0	=	42
READ	0xdeadbeef	@6	=	42
READ	0xdeadbeef	@9	=	42
WRITE	0xdeadbefe	@2	=	63
READ	0xdeadbefe	@8	=	63

Circuit generation



O(T · log T)

O(T · ST)

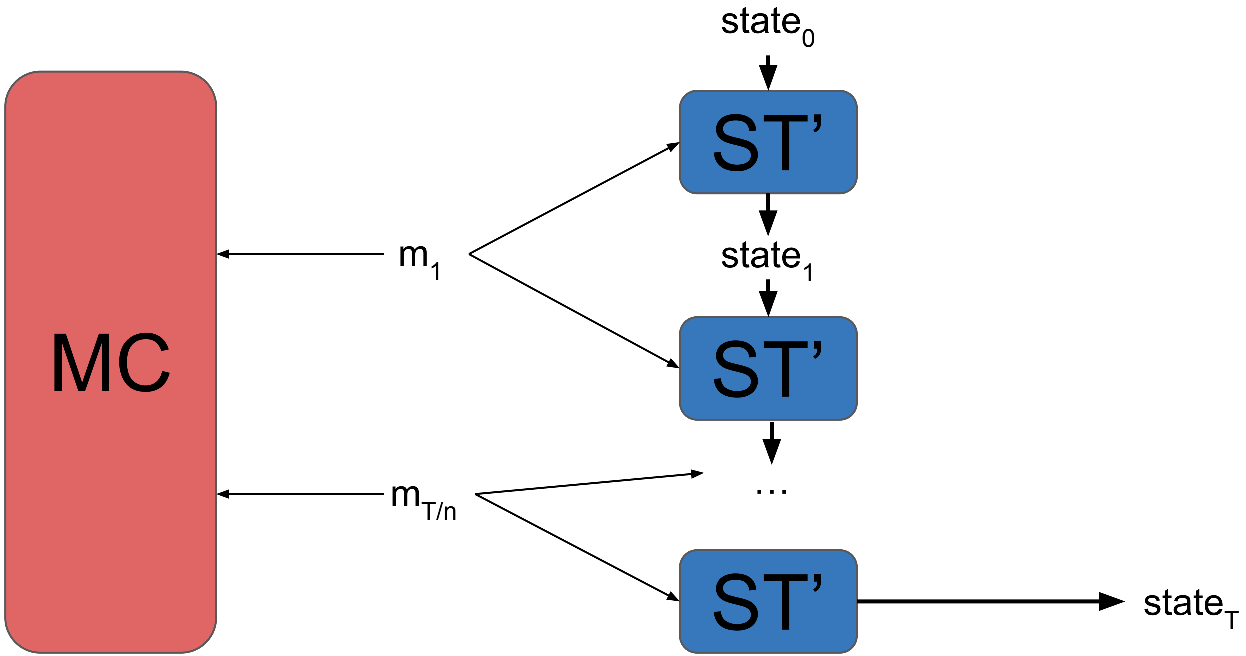
Novel Contributions in Our Framework

- **Optimizations** that minimize circuits that verify program executions
  a. Sparsity
  b. Public PC
- Efficient **encodings** to detect vulnerabilities
  a. Memory errors
  b. Information leakages

Optimization: Sparsity

- Certain instructions are expensive and used infrequently
  - Example: memory operations
- What if we don't run these instructions every cycle?
- Savings: no need to generate memory check circuits for excluded cycles
  - Instructions that are sparsely used are only inserted once every **n** cycles.
  - The circuit cost becomes **1 / n**

Optimization: Sparsity



Optimization: Public PC

- Encoding the state transition function for each instruction is expensive
- What if we generate circuits for frequently used basic blocks in the program?
- Savings: The program counter (PC) is public, so we can generate optimized circuit representations for the entire basic block

Optimization: Public PC

```
fn transition(current_st) -> State {
    let instr = fetch_instr(current_st.pc);
    let arg1 = index(current_st.regs, instr.op1);
    let arg2 = index(current_st.regs, instr.op2);

    let result = mux(instr.opcode == XOR, xor(arg1, arg2),
                     instr.opcode == ADD, add(arg1, arg2),
                     ...);
    ...
}
```

Optimization: Public PC

fn transition(current_st) -> State {

let instr = ~~fetch_instr(~~current_st.pc~~)~~;

let arg1 = ~~index~~(current_st.regs, instr.op1);

let arg2 = ~~index~~(current_st.regs, instr.op2);

let result = ~~mux(instr.opcode == XOR,~~ xor(arg1, arg2),

~~instr.opcode == ADD, add(arg1, arg2)~~,

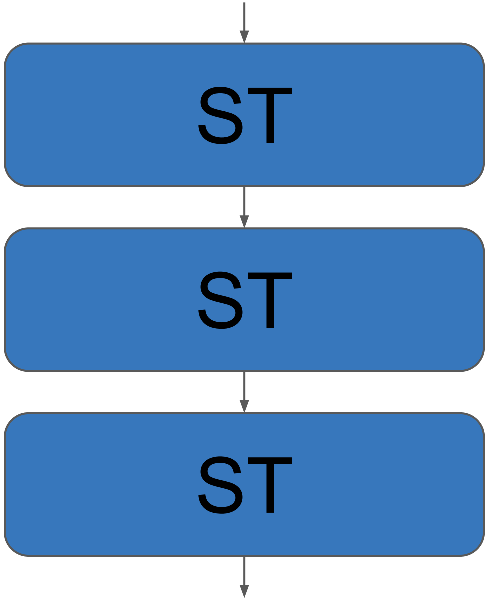
~~...)~~;

...

}

Optimization: Public PC

Private execution



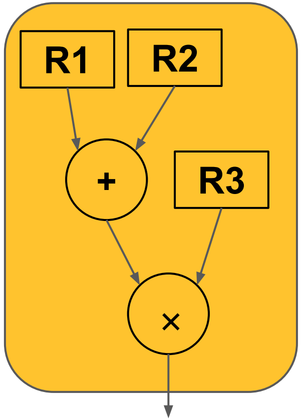
Code block

```
MOV R0 R1

ADD R0 R0 R2

MUL R0 R0 R3
```

Public execution



Encoding: Memory vulnerabilities

Catch memory vulnerabilities:

- Reads/writes before allocation
- Out-of-bounds access
- Reads/writes after free
- Use after free

Example vulnerabilities proven so far:

- **GRIT:** Gameboy advanced Image Transmogrifier
- **FFmpeg**: Library for handling multimedia

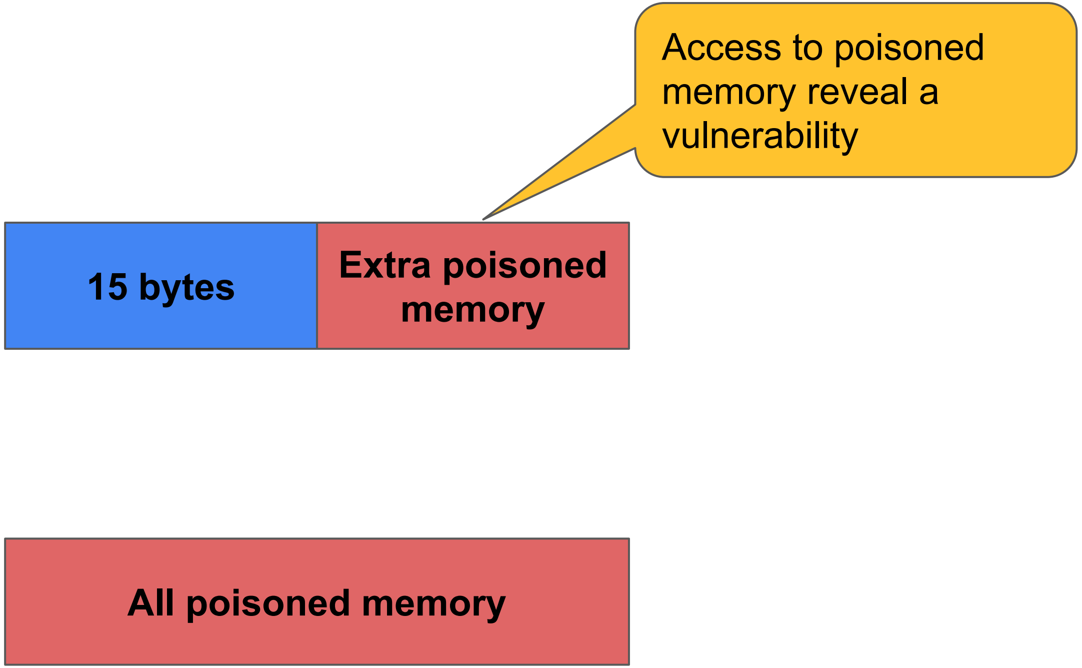
Encoding: Memory vulnerabilities

Memory allocation

```
ptr = malloc(15);
```

Memory free

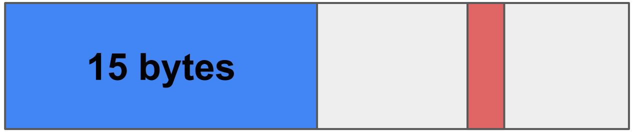
```
free(ptr);
```



Encoding: Memory vulnerabilities

Memory allocation

```
ptr = malloc(15);
```



Memory free

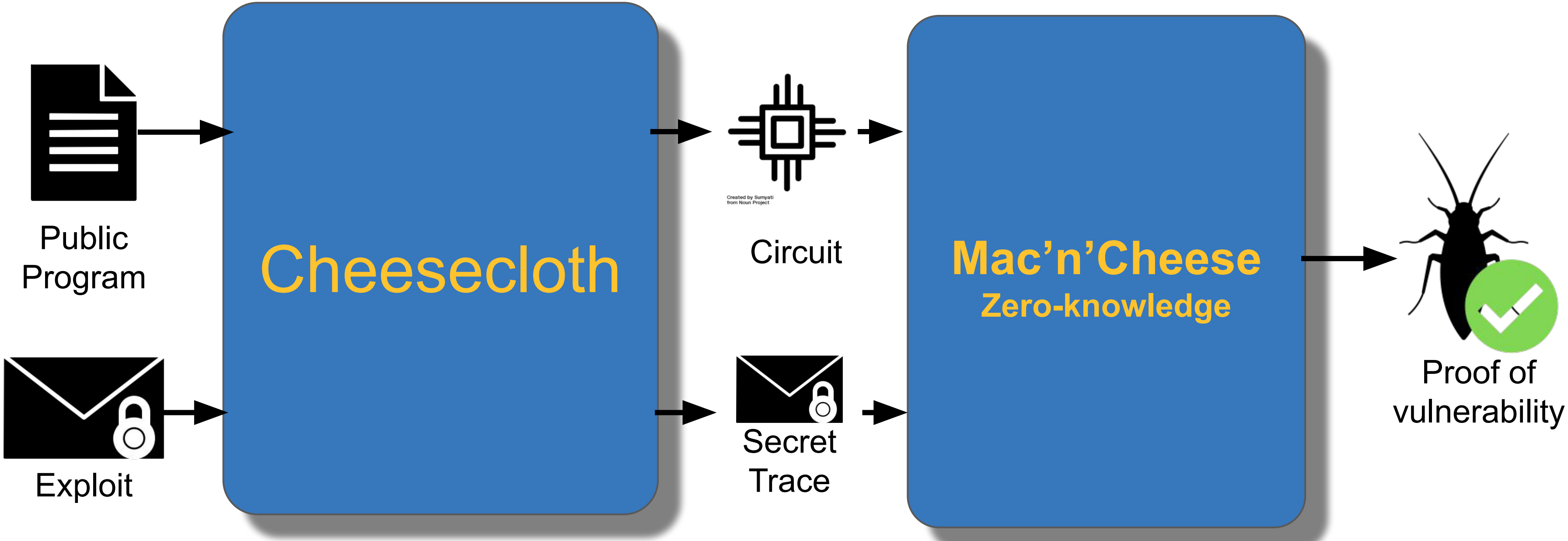
```
free(ptr);
```

Encoding: Information leakage

Two options:

- **Two executions:** executions should look identical to attacker even when sensitive data changes.
  - Circuit twice as large!
- **Taint tracking:** Mark sensitive data and and show that tainted data is revealed to the attacker.
  - Under approximation is non-trivial!

Mac’n’Cheese Backend

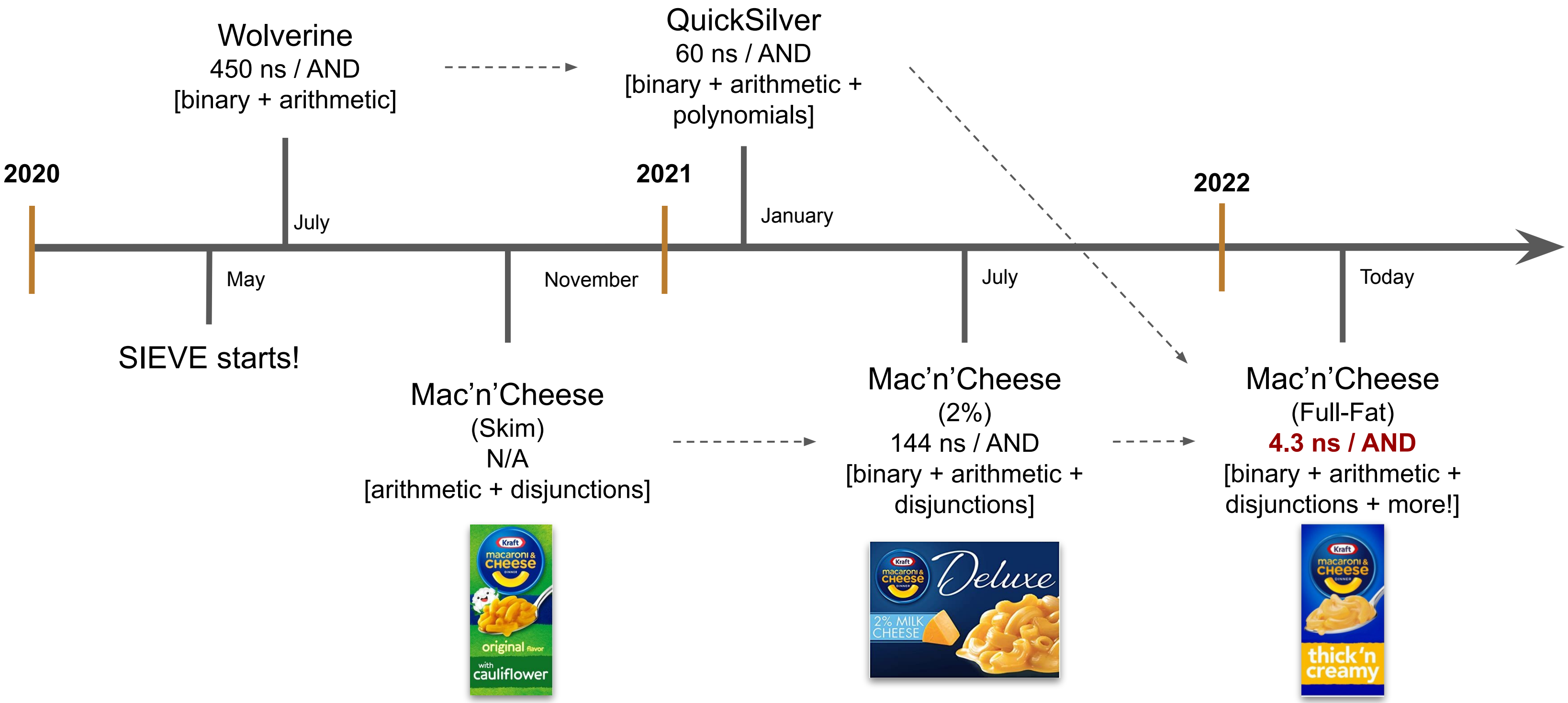


What is Mac’n’Cheese

ZK backend for VOLE-based protocols

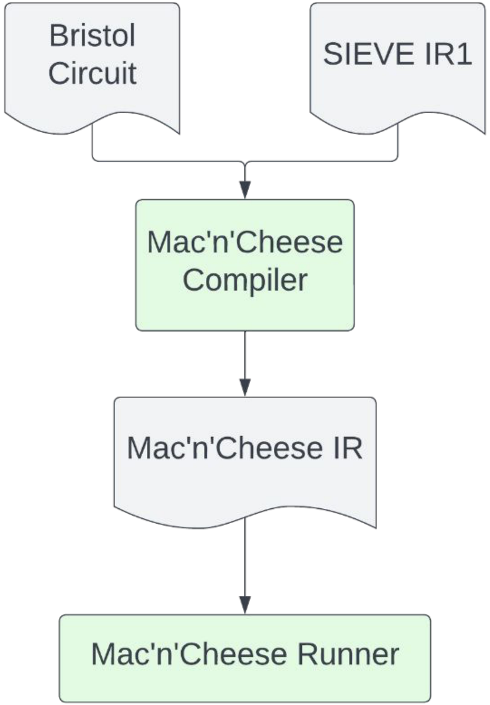
- Combines multiple VOLE-ZK protocols in a highly optimized architecture
- Currently: QuickSilver [YSWW21] (for non-disjunctive computations), Mac’n’Cheese (for disjunctive computations) [BMRS21]
- In the pipeline: Appenzeller2Brie [BBMRS21] (for field switching), Mozzarella [BBMS22] (for ring support), AntMan [WYYXW22] (for optimizing for-loops), …

VOLE-ZK implementation timeline



Sources: https://target.scene7.com/is/image/Target/GUEST_5824921a-638b-4936-b57a-bab60b774594, https://target.scene7.com/is/image/Target/GUEST_0ad8e33a-65b1-49ec-9fb0-a9d7fefed37b, https://target.scene7.com/is/image/Target/GUEST_1dc489ef-a819-4de3-b299-4f258ea28a68

Mac’n’Cheese Architecture



Compiles input program into Mac'n'Cheese IR

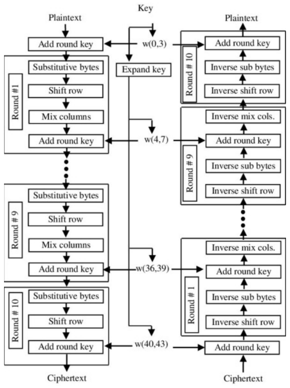
Represented as "Task Graph"
Task: Minimal unit of work for Mac'n'Cheese runner

Schedules + executes tasks

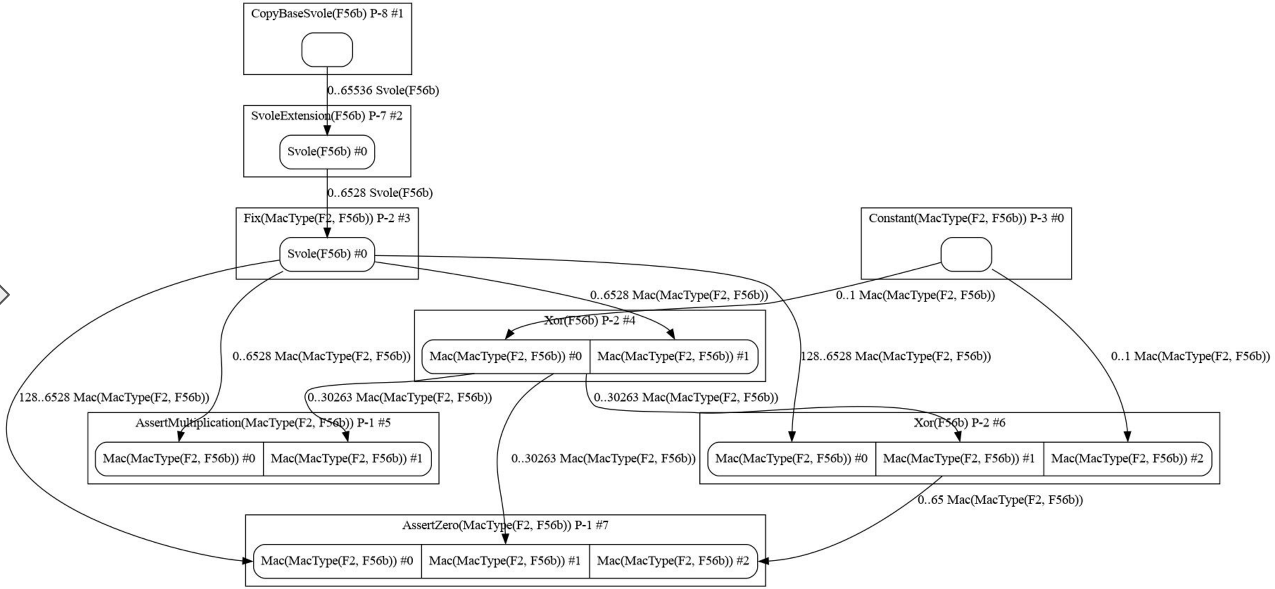
Existing Tasks

`VoleBase`: Generate base VOLEs
`VoleExtend`: Extend base VOLEs
`Constant`: Set constant values
`Copy`: Copies values
`Fix`: Sets prover-secret values
`Xor`: Computes XORs over binary values
`AssertZero`: Asserts wire is zero
`AssertMult`: Asserts mult is correct

Compiling AES using Mac’n’Cheese



Compilation



Input circuit

Mac’n’Cheese IR

Benefits: Modularity, Parallelizability, Performance

Modular

- Can add tasks easily + independently
    - Ligero integration, RAM, polynomial checks, etc.
- Can build complex components from tasks
    - Field switching is “just” composition of tasks

Parallelizable

- Versus “sequential” nature of circuit representation

Performant

- Tasks do “one thing”, which can be done much faster

Mac’n’Cheese performance

Experiment:

- Ran on **m5.8xlarge** instances between Virginia and Oregon
- Circuit contained **2.3 B** AND gates + **10 B** XOR gates
- Overall Time: **9.9 seconds**

⇒ **4.3 ns / AND gate**

⇒ **232 million AND gates / second**

Mac'n'Cheese vs software AES + SHA-2

Software AES: **483 ns** per block

- Using https://github.com/kokke/tiny-AES-c

Mac'n'Cheese: **51200 ns** per block

⇒ **~106x** slower than software AES!

SHA-2 is similar:

- **~72x** slower versus `hashlib`'s sha256!

Mac'n'Cheese run across the US is
~100x slower than native computation*

* For heavily optimized AES / SHA-2 circuits

Availability

- Planning to open source implementation “soon”
    - Reach out if you’re interested in early access!

- Also: if you have use cases that require interactive ZK
    - We’re happy to share access to our implementation / Docker-ized test bed

FROMAGER: A scalable toolchain for complex ZK Proofs about software

galois

AARHUS UNIVERSITET

COLUMBIA UNIVERSITY

qedit

Santiago Cuellar

Stuart Pernsteiner

Ben Razet

Bill Harris

Carsten Baum

Peter Scholl

Eran Tromer

Daniel Benarroch

Marc Rosen

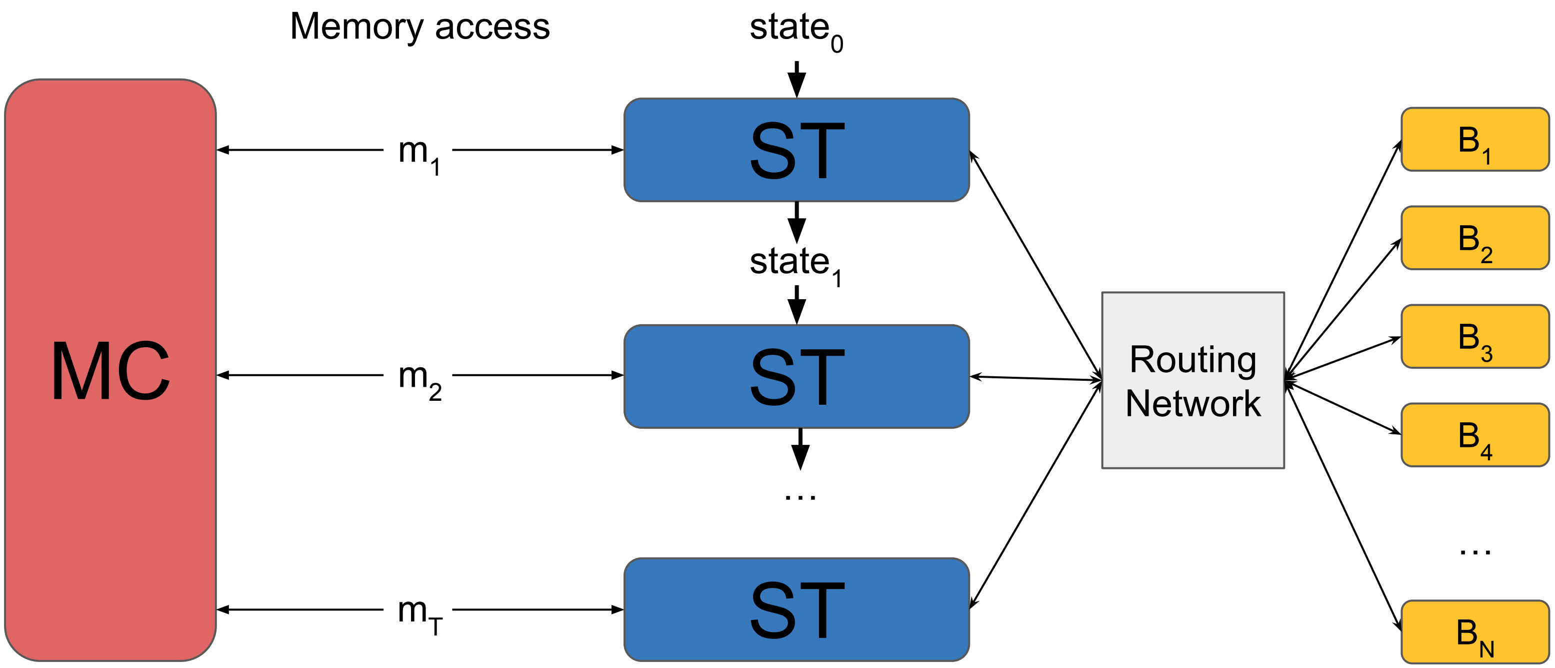
Chris Phifer

Dave Archer

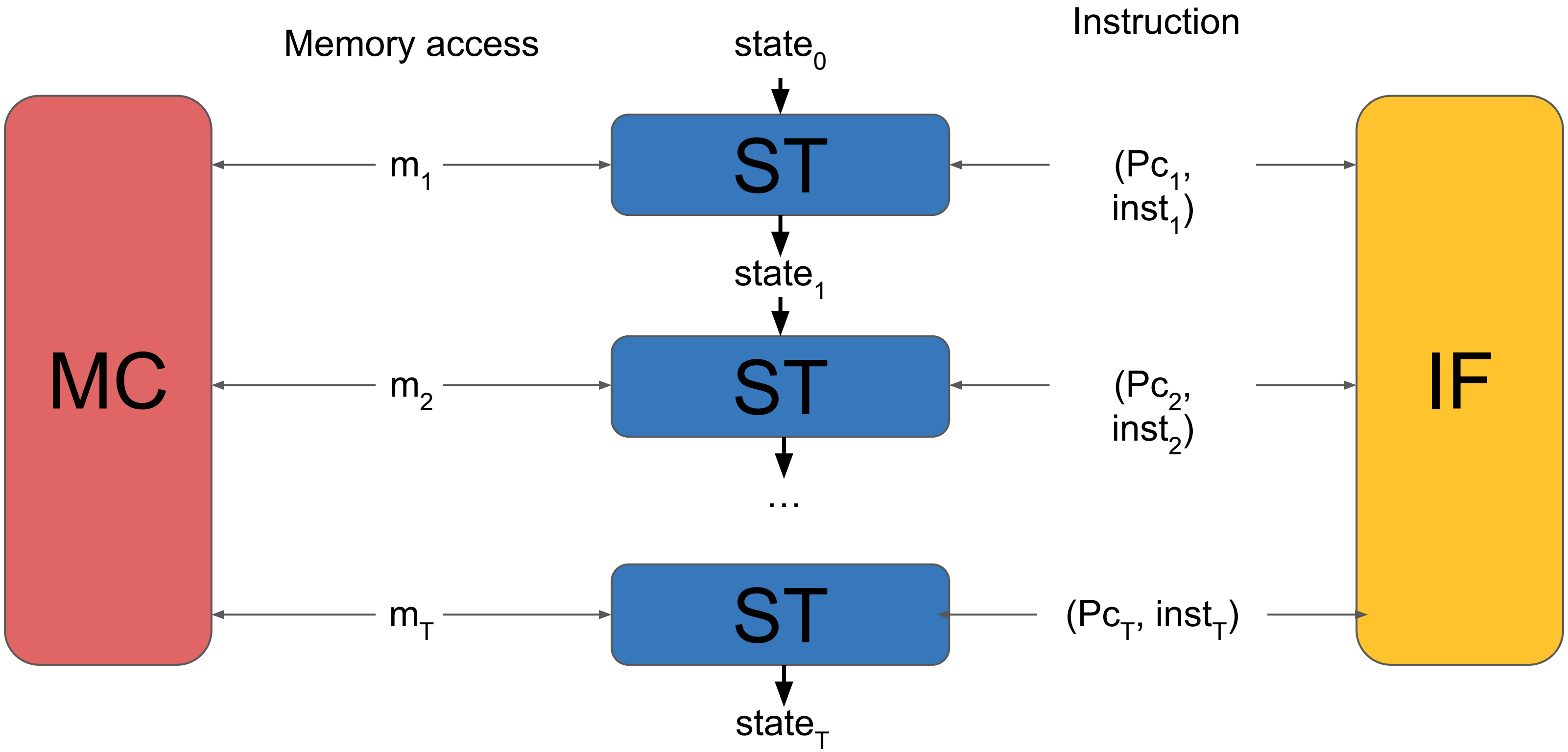
Alex Malozemoff

Constance Beguier

Optimization: Public PC



Circuit generation



Execution: Information leakage

Taint tracking

```
{secret x}
y = x + 1
```

Execution: Information leakage

Taint tracking

```
{secret x}
if x then
   y = r1;
else:
   y = r2;
```

Execution: Information leakage

We use conservative taint tracking (underapproximation):

- Punt on branches and arrays
- Guarantees a vulnerability.

Enough to prove:

- heartbleed
- other protocols that don’t branch on secret data.

Encoding: Cryptographic protocol (WIP)

Simulate adversarial program

- Multiple programs communicating through network
- Secret program (adversary)
- Support random-seed
  - Fiat-Shamir doesn't work!

Outline

1. **Background**
2. **TA1**
    a. Intro/Motivation
    b. Optimizations (sparsity, public pc)
    c. Vulnerability encodings (memory, taint analysis)
3. **TA2**
    a. High level architecture
    b. Performance numbers
    c. Open sourcing
4. **Call for users/use cases**