Zero-knowledge Proofs of Binary Exploitability

Ben Perez, Eric Hennenfent, Gabriel Kaptchuk, Gijs Van Laer, Mathias Hall-Andersen, Matthew Green

Acknowledgement

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Approved for Public Release, Distribution Unlimited

Zero-knowledge Proofs

- **Allow a prover to convince a verifier that they posses some piece of knowledge without revealing the information itself**
    - Prove knowledge of a SHA256 preimage for some value **x**
    - Demonstrate your private transaction in Zcash is valid
- **The statement being proved is represented as either:**
    - Boolean circuit: XOR, NOT, AND
    - Arithmetic circuit: ADD, NEG, MUL
- **Circuit must be a DAG and the entire circuit must be executed**
- **Performance depends on number of AND/MUL gates**

ZK Proofs of Exploitability

- **Allow vulnerability researchers to prove they have a valid exploit without revealing techniques**
- **Remove trust from bug disclosure process, protect users**
- **Must operate at the binary and processor level**
  - Source-level vulnerabilities are very common and rarely lead to exploits
  - Memory protections, heap layout, and syscalls are processor/runtime specific
  - Rarely have access to source
- **Challenges:**
  - How do we model exploits as ZK circuits
  - Provide users with simple-to-use ZK statement compiler
  - Find ZK proof systems that are efficient for very large circuits

Results: Microcorruption

- **CTF where players break into a lock controlled by an MSP430**
- **Covers common exploitation techniques:**
    - Stack/heap overflow, command injection, and ROP gadgets
    - Bypass protections such as ASLR, DEP, and stack canaries
- **Our toolchain can prove the Microcorruption challenges with a ZK MSP430 processor running at 30.5 Hz**
- **Proofs require 128 kb per instruction**
- **Can complete an exploit that takes 12k steps to finish in ~7 min**

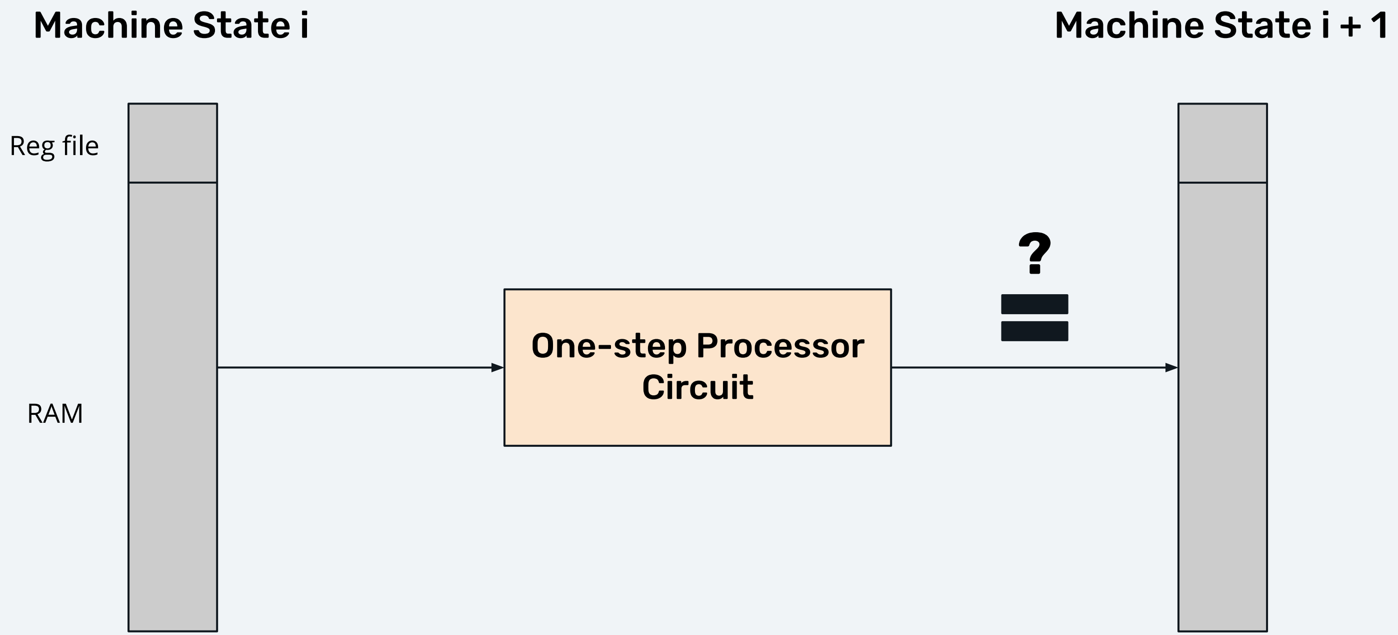
Results: Toolchain

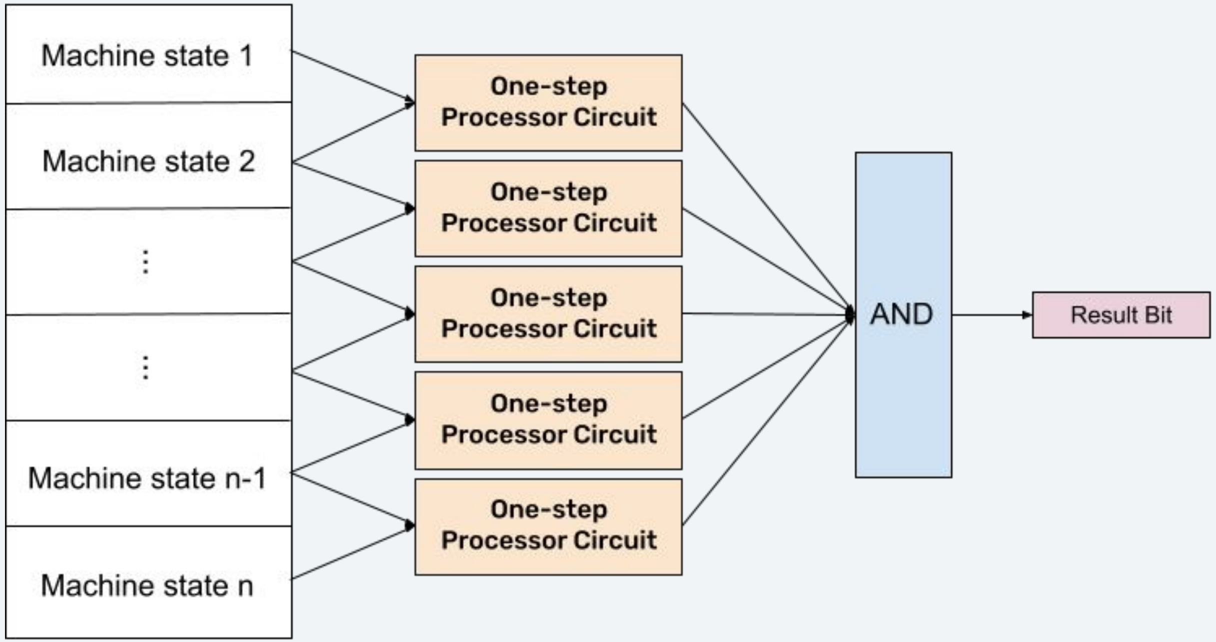
- **Circuit Compiler:**
  - Model processor behavior in Verilog
  - Use a combination of Yosys and custom backend to generate ZK statements
  - Improve synthesis time and memory usage by 99% and 88%, respectively
  - User only needs to input a valid MSP430 binary
- **ZK Proof System: Reverie**
  - First highly-optimized Rust implementation of an MPC-based ZK proof system
  - No trusted setup or non-standard cryptographic assumptions
  - Optimizes for prover time: 2 orders of magnitude faster than other ZK provers

Modeling RAM Programs in ZK

Naive Approach

- Prover runs program on an emulator with their exploit
- Emulator produces a program trace
- Trace is the secret input to the ZK proof system
- ZK statement is a sequence of circuits that check whether each trace entry logically follows from the previous one
- Each step of the circuit contains a register file, ALU, RAM, etc


Machine State i
Machine State i + 1
Reg file
RAM
One-step Processor Circuit
?
=

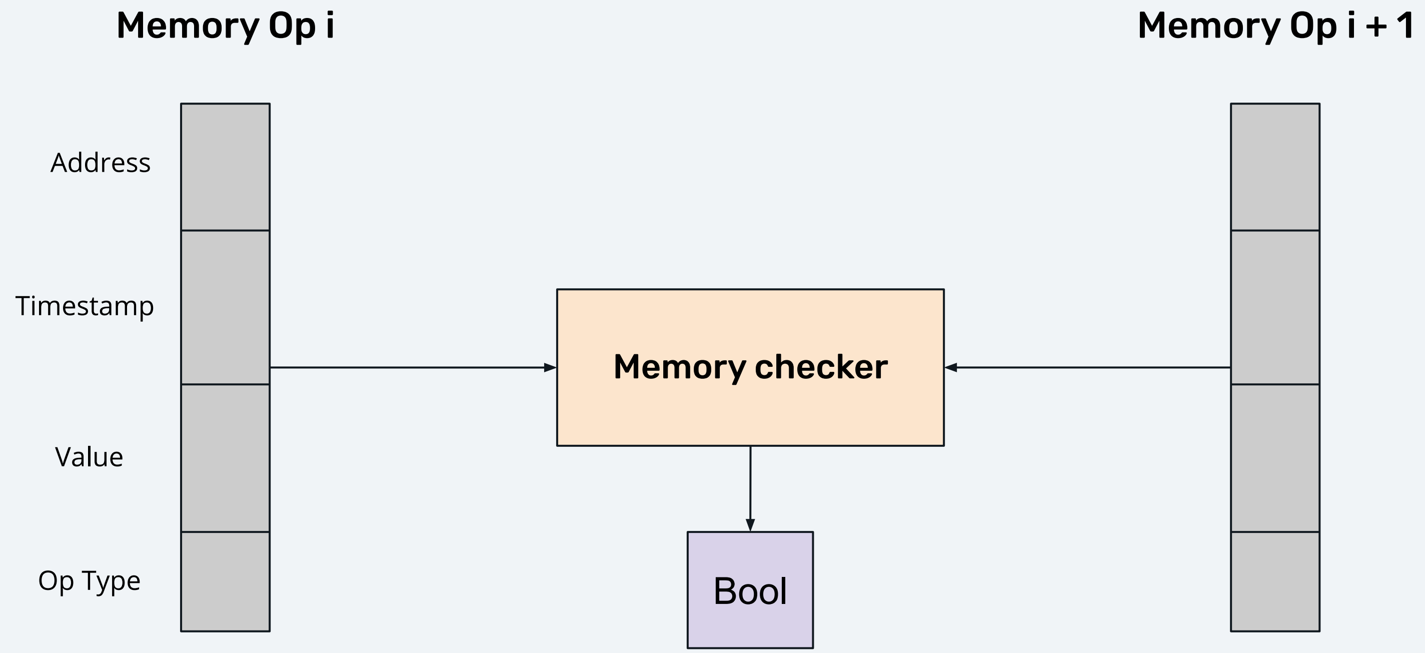

Machine state 1
Machine state 2
⋮
⋮
Machine state n-1
Machine state n
One-step Processor Circuit
One-step Processor Circuit
One-step Processor Circuit
One-step Processor Circuit
One-step Processor Circuit
AND
Result Bit

Problems with Naive Approach

- **Very large constant factors**
  - Processor circuit (decoder, ALU, register file)
  - Must mux the entirety of RAM at every step
- **Totally infeasible for even small amounts of RAM**
- **Want to develop solution that scales linearly with number of memory accesses, not total size of RAM**

Better Approach (Ben-Sasson et al.)

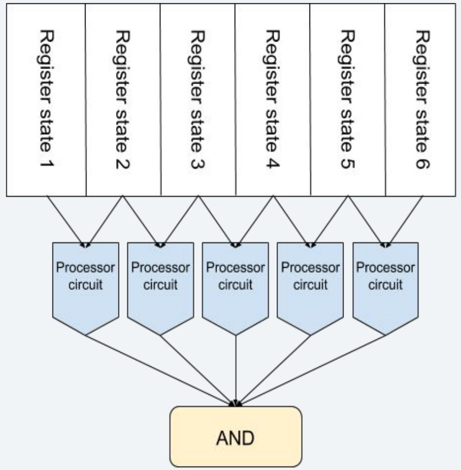
- **Check memory in separate proof**
- **Provide *memory sorted* trace as auxiliary input**
  - Memory accesses are sorted by address
  - Ties are broken by timestamps
- **Memory checker verifies that adjacent reads/writes are consistent**
  - If a value **x** is written to address **y**, check that subsequent reads from that address contain **x**.
  - No muxing, easy to verify constraints, linear in number of memory ops
- **Augment program trace with *memory hints*, the alleged values being read from memory**


Memory Op i
Memory Op i + 1
Address
Timestamp
Value
Op Type
Memory checker
Bool

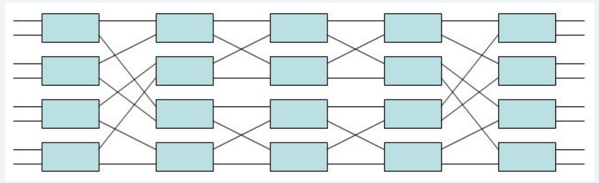
Better Approach (Ben-Sasson et al.)

- Problem: prover could input a program trace and memory trace that have nothing to do with each other
- Must prove program trace is a permutation of the memory trace
- Accomplished via routing networks
- Proof requires a circuit n*log(n) in the trace size
- Asymptotically worse than naive approach, but in reality much more efficient

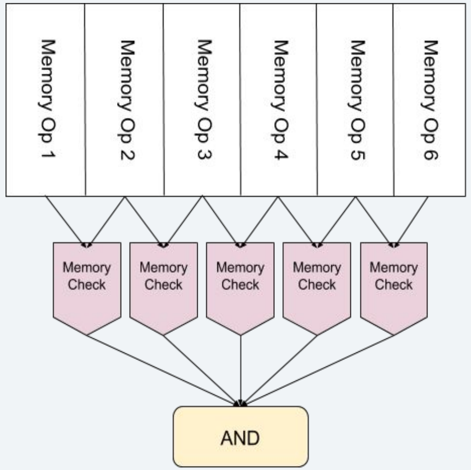
Check trace validity



Permutation Proof



Validate memory



Our Work

- **Eliminate log(n) factor in permutation proof**
- **Use polynomial argument: Given two lists A and B and a challenge x, perform the following check:**

$$\Pi_{i=1}^{n}(A_i - x) = \Pi_{i=1}^{n}(B_i - x)$$

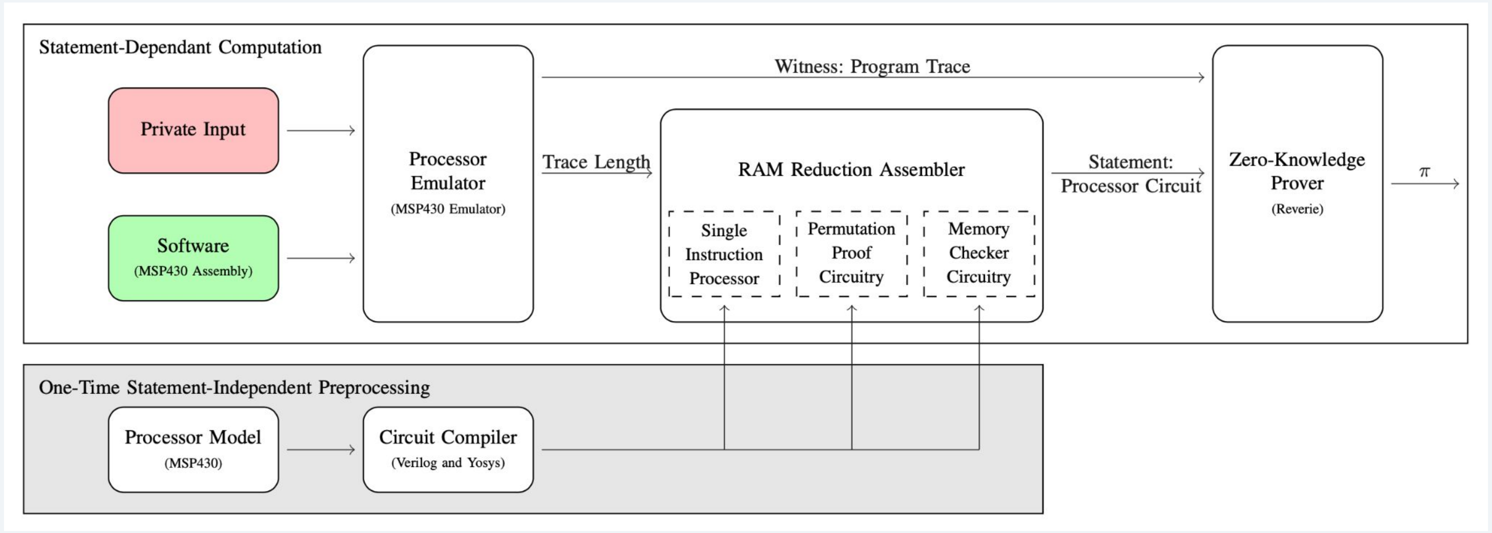
- **Challenge is generated using Fiat-Shamir**
- **Prover cannot cheat because of Schwartz-Zippel**

Our Work

- **<u>Problem:</u> this permutation proof is efficient for arithmetic circuits, but we use Boolean (multiplication is expensive)**
- **<u>Solution:</u> develop techniques for switching between Boolean and arithmetic circuits**
  - Uses 128 AND gates to convert 64 Boolean values into 64-bit arithmetic values
  - Final cost: 256 AND gates and 2 MUL gates per memory operation
- **Useful independent of the proof of vulnerability application**

Proof of Exploitability Toolchain

Toolchain


Statement-Dependant Computation
Private Input
Software
(MSP430 Assembly)
Processor Emulator
(MSP430 Emulator)
Witness: Program Trace
Trace Length
RAM Reduction Assembler
Single Instruction Processor
Permutation Proof Circuitry
Memory Checker Circuitry
Statement: Processor Circuit
Zero-Knowledge Prover
(Reverie)
π
One-Time Statement-Independent Preprocessing
Processor Model
(MSP430)
Circuit Compiler
(Verilog and Yosys)

Circuit Compiler

- **Write core circuits in Verilog and synthesize with Yosys**
- **<u>Problem:</u> when unrolling the whole RAM reduction Yosys spends time looking for optimizations we know don't exist**
    - For a trace with 7k instructions, Yosys uses 160GB of RAM and takes 24 hrs to finish
    - Will not scale to real exploits
- **<u>Solution:</u> develop circuit flattener that takes advantage of the repetitive nature of the RAM reduction circuit**
    - Use Yosys only for the one-step processor circuit and memory checker
    - Aggressively cache flattened versions these components and avoid repeating work
    - Can do 7k step trace in 6 minutes using 20GB RAM - a 99% and 88% improvement over Yosys, respectively

Modeling Exploits

- **Many IoT exploits can be identified by the fact that the an attacker has the ability to execute a certain system call**
  - Unlock a door
  - Turn on a camera/microphone
- **Privilege escalation can be detected via the results of a system call, e.g. checking if** `geteuid() == 0`
  - ZK proof concludes with such a system call and returns the output
  - Can be extended to other syscalls like `mprotect`
- **Prover can demonstrate RCE/ACE by having verifier challenge them to set PC to random values**

Reverie

- **Based on recent work of Katz *et al.***
- **Prover run MPC protocol “in their head” and verifier opens all but one player**
- **Large proofs, but facilitates streaming (only needs 3.9mbps bandwidth)**
- **Benchmarks for computing 511 iterations of SHA256**

	Setup (sec)	Prove (sec)	Verify (sec)	Size (KB)
libSNARK	1,027	360	0.002	.013
Bulletproofs	-	2,555	0.044	395
Ligero	-	400	4	1,500
Reverie	-	9.6	7.67	112,000

Future Work

- **Model larger processors such as ARM and x86**
  - Already working on x86 circuit based on 80386
  - Want to automate circuit generation for new architectures
- **Support more realistic runtime environments**
  - The DARPA Cyber Grand Challenge has a large corpus of vulnerable x86 binaries that run on DECREE, a simple operating system
- **Take advantage of recent ZK breakthroughs to minimize proof size**
  - Interactive systems (sVOLE, GC)
  - Free disjunctions