C: There’s a SNARK for That

Alessandro Chiesa
Daniel Genkin
Eli Ben-Sasson
Eran Tromer
Madars Virza
& Co.
Problem: integrity on untrusted platform

- Faults
- Someone else’s cloud
- Platform trojans
- Blue pill
- OS bugs
- Untrusted data origins
- Crypto protocols
Solution: zk-SNARKs

- Zero knowledge
- Succint
- Noninteractive Argument of Knowledge

AKA

- Non-interactive CS proofs of knowledge
- Succint NIZK

For C programs with pointers and loops
#include <tinyram.h>
#define LEN 16

int main() {
    int state[256], stream[LEN];
    int i, j, t, k;

    for (i=0; i < 256; ++i)  state[i] = i;
    /* KSA: mix in key */
    k = 0; j = 0;
    for (i=0; i < 256; ++i) {
        t = state[i];
        keybyte = read_aux_input_tape();
        j = (j + t + keybyte) & 0xFF;
        state[i] = state[j]; state[j] = t;
    }
    /* PRGA: produce stream */
    i=0; j=0;
    for (k=0; k < LEN; k++)  {
        i = (i + 1) & 0xFF;
        t = state[i];
        j = (j + t) & 0xFF;
        state[i] = state[j];
        state[j] = t;
        stream[k] = state[(state[i] + state[j]) & 0xFF];
    }
    /* compare with the claim */
    for (i=0; i < LEN; i++)
        if (stream[i] != read_primary_input_tape()) {
            return 1;
        }
    return 0;
}
Compiler based on GCC

C program

TinyRAM program
## TinyRAM architecture for fast verification

<table>
<thead>
<tr>
<th>instruction mnemonic</th>
<th>operands</th>
<th>effects</th>
<th>flag</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>ri, rj, A</td>
<td>compute bitwise AND of [rj] and [A] and store result in ri</td>
<td>result is 0^W</td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>ri, rj, A</td>
<td>compute bitwise OR of [rj] and [A] and store result in ri</td>
<td>result is 0^W</td>
<td></td>
</tr>
<tr>
<td>xor</td>
<td>ri, rj, A</td>
<td>compute bitwise XOR of [rj] and [A] and store result in ri</td>
<td>result is 0^W</td>
<td></td>
</tr>
<tr>
<td>not</td>
<td>ri, A</td>
<td>compute bitwise NOT of [A] and store result in ri</td>
<td>result is 0^W</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>ri, rj, A</td>
<td>compute [rj]_u + [A]_u and store result in ri</td>
<td>overflow</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>ri, rj, A</td>
<td>compute [rj]_u - [A]_u and store result in ri</td>
<td>borrow</td>
<td></td>
</tr>
<tr>
<td>mull</td>
<td>ri, rj, A</td>
<td>compute [rj]_u × [A]_u and store least significant bits of result in ri</td>
<td>overflow</td>
<td></td>
</tr>
<tr>
<td>umull</td>
<td>ri, rj, A</td>
<td>compute [rj]_u × [A]_u and store most significant bits of result in ri</td>
<td>overflow</td>
<td></td>
</tr>
<tr>
<td>smulh</td>
<td>ri, rj, A</td>
<td>compute [rj]_u × [A]_u and store most significant bits of result in ri</td>
<td>over/underflow</td>
<td>[A]_u = 0</td>
</tr>
<tr>
<td>udiv</td>
<td>ri, rj, A</td>
<td>compute quotient of [rj]_u/[A]_u and store result in ri</td>
<td></td>
<td></td>
</tr>
<tr>
<td>umod</td>
<td>ri, rj, A</td>
<td>compute remainder of [rj]_u/[A]_u and store result in ri</td>
<td></td>
<td></td>
</tr>
<tr>
<td>shl</td>
<td>ri, rj, A</td>
<td>shift [rj] by ri</td>
<td></td>
<td></td>
</tr>
<tr>
<td>shr</td>
<td>ri, rj, A</td>
<td>shift [rj] by ri</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmpe</td>
<td>ri, A</td>
<td>none (“count equal”)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmpa</td>
<td>ri, A</td>
<td>none (“count not equal”)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmpeae</td>
<td>ri, A</td>
<td>none (“count equal or not equal”)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmpge</td>
<td>ri, A</td>
<td>none (“count greater than or equal”)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mov</td>
<td>ri, A</td>
<td>store [A] in ri</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmov</td>
<td>ri, A</td>
<td>if flag = 1, set pc to ri</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jmp</td>
<td>A</td>
<td>set pc to A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cjmp</td>
<td>A</td>
<td>if flag = 1, set pc to A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cnjmp</td>
<td>A</td>
<td>if flag = 0, set pc to A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>store</td>
<td>A, ri</td>
<td>store [ri] at memory address [A]_u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>load</td>
<td>ri, A</td>
<td>store the content of memory address [A]_u into ri</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read</td>
<td>ri, A</td>
<td>if the [A]_u-th tape has remaining words then consume the next word, store it in ri, and set flag = 0; otherwise store 0^W in ri and set flag = 1 (stall or halt and the return value is [A]_u)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>answer</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(1) All but the first two tapes are empty: if [A]_u \not\in \{0, 1\} then store 0^W in ri and set flag = 1.
(2) answer causes a stall (i.e., not increment pc) or a halt (i.e., the computation stops); the choice between the two is undefined.

Spec: [http://scipr-lab.org/tinyram](http://scipr-lab.org/tinyram)
C program

Compiler

based on GCC

TinyRAM program

Circuit Generator

based on theory of [BCGT13]

circuit
Converting TinyRAM verification to circuit satisfiability

_sumarray:
    store r0 r0
    mov  r1 0000
    mov  r2 1000
    mov  r3 2000
    mov  r4 0
    mov  r5 100

_loop:
    cmpe r4 r5
    cjmp _end
    load r6 r1
    pload r6 r1
    load r7 r2
    pload r7 r2
    add r8 r7 r5
    store r3 r8
    add r1 r1 1
    add r2 r2 1
    add r3 r3 1
    add r4 r4 1
    jmp _sum

_end:
C program

Compiler based on GCC

TinyRAM program

Circuit Generator based on theory of [BCGT13]

circuit

zkSNARK for CircuitSAT based on theory of [GGPR13], [BCIOP13]
“I know an RC4 key producing
26 41 5B C4 4C EC ED 6C 89 99 68 E1 82 04 DE”
“The execution of arbitrary C programs can be verified in a few milliseconds and 322 bytes”

http://scipr-lab.org
What Would you Like to Prove Today?