Review: Verifying Constant-Time Implementations

Mahyar Emami, Rishabh Iyer, Sahand Kashani
firstname.lastname@epfl.ch

November 13, 2021

1 Introduction

- Side-channel attacks are predominant today.
- Constant-time programming attempts to alleviate such attacks, but CT programming is hard and programs that one thinks are CT often are not.
- This paper presents the techniques used in a tool, ct-verif, that can formally assert if an input program is CT or not.
- Constant-time programs in their purest form often suffer from a performance penalty. Relaxing the definition of constant-time is possible to improve program performance as long as it does not compromise security.
- The novelty of this paper is its ability to determine if this category of relaxed constant-time programs are indeed constant-time.
- Takes as input LLVM IR and returns a simple yes/no result saying whether it is constant-time or not.

2 Formalization of Constant Time

Here we summarize the authors' formalization of constant-timedness for a simple, high-level, structured programming language. Fig. **??** lists the language's syntax. The operational semantics for the language's primitives are standard and are listed in Fig. **??** for reference.

A state s maps variables x and indices $i \in N$ to values $s(x, i)$, we use $s(e)$ to denote the value of expression e in state s. The distinguished error state $\perp$ represents a state from which no transition is enabled. A *configuration* $c = \langle s, p \rangle$ is a state s along with a program p to be executed, and an *execution* is a sequence $c_1, c_2, ...c_n$ of configurations such that $c_i \rightarrow c_{i+1}$ for $0 < i < n$. The execution

```c
void copy_subarray(uint8 *out, const uint8 *in,
    uint32 len, uint32 l_idx, uint32 sub_len){
  uint32 i,j;
  for(i=0,j=0; i<len; i++){
    if((i >= l_idx) && (i<l_idx + sub_len)){
      out[j] = in[i];
      j++;
    }
  }
}
```

Figure 1: Running example - sub-array copy

- Simple example that can easily be checked + example of benign branch that other tools cannot determine to be constant-time.
- Formalism used to model a program (while-language framework and what it supports).
- The paper proposes modeling contant-time verification of a program by encoding the input program as a safety condition and executing the program to check if it is safe.
- Define what leakage $L(c)$ is formally. A leakage model can either depend on
- 1. Path-based characterizaion of constant-time: leakage of branch conditions.
  2. Cache-based characterization of constant-time: leakage of memory access indices.
  3. Instruction-based characterization of constant-time: leakage of instruction operand sizes.
- Define what it means for a program to be safe (i.e., it does not violate an assertion inserted when the safety program).

3 Body

- How is the reduction to a safety check performed?

4 Conclusion

- Pros: Can correctly categorize a much larger set of security programs as being constant-time.
- Cons: Have to annotate benign branches.

References