A case study on the use of a tool to verify constant time code in a cryptography library

Ernest DeFoy, Danfeng Zhang
College of Engineering
The Pennsylvania State University

Background

Side channels are sources of leakage of information in a system. Timing channels, a type of side channel, are sources of leakage in a computer system that are the result of incorrect implementation rather than an insecure program. They are realized from variations in program runtime, which can leak information about the behavior of an algorithm. A timing attack is a computer security attack applicable to general software systems and based on the measurement of time various computations [1]. In the simple example below, *k* is confidential data, and any variable computed from *k* is tainted and treated as confidential data as well The second example is taken from the mbedTLS source wherein the tainted variable *N* is directly compared to public input *A*. Depending on the the comparison, the code either does a copy operation or conducts a modulus operation.

$k = \text{sensitive information}$
$d.x = k + 1$
$d.y = 0$
$a = -1$
if $d.y == k/2$ **then**
  $a = 4$
end if

```
1  if( mbedtls_mpi_cmp_mpi( A, N ) >= 0 )
2      MBEDTLS_MPI_CHK( mbedtls_mpi_mod_mpi(
           &W[1], A, N ) );
3  else
4      MBEDTLS_MPI_CHK( mbedtls_mpi_copy( &W
           [1], A ) );
```

Figure 1. Left, leakage of variable *k* and flow to variable *a*. **Right,** This branch is a high-risk positive that results in a notcieable variation in runtime .

Introduction

Ct-verif is a software tool that verifies constant time code [2]. It uses a static analysis to detect violations of constant time policies. A program *P* is reduced to assertion safety by encoding two executions into a new program *Q*. Each copy of *P* has its own set of inputs, and their equality is asserted before each branch when executing *Q*. If an assertion fails, constant time policy has been violated. In other words, ct-verif simulates two instances of the same program and ensures that their branch decisions do not rely on secret information. **However, ct-verif is limited in its capabilities in practice, and an evaluation needs to take them into consideration. Source needs to be refactored to accommodate these limitations.**

```
1  void
2  function_wrapper (uint32_t *x, const
       uint32_t *m, ...) {
3
4      public_in(__SMACK_value(x));
5      public_in(__SMACK_value(m));
6      public_in(__SMACK_values(x,LENGTH));
7      ...
8      br_i32_modpow (x,m,...);
9  }
```

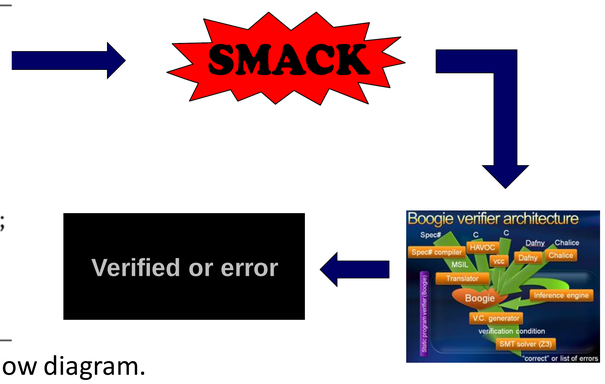

Figure 2. Ct-verif analysis flow diagram.

Experiment

We analyze crypto library mbedTLS by annotating public inputs of its modular exponentiation function and passing in the source to ct-verif. Ct-verif compiles annotated C code and generates optimized LLVM intermediate representation (IR) before leveraging the software verification tools SMACK and Boogie. It uses SMACK to translate the IR into Boogie code, performs the reduction on the Boogie code, and passes it into the Boogie verifier. The verifier terminates with a binary result—the code is either verified or it is not.

Experiemental Results

Positives reported can be categorized by the cause of the error, and broadly grouped into low-risk and high-risk results. A low risk result is any positive that leaks a benign amount of information, such as the length of a secret key. A high-risk result leaks discrete information about a key, such as a loop bound dependent on the properties of a secret key. High-risk positives in the mbedTLS function mbedtls_mpi_exp_mod function were identified manually. Ct-verif reported false positives but identified all high-risk and low-risk positives.

Library	LOC	Annotations	Revisions	Average	Positives	High-Risk	Low-Risk
mbedTLS 2.9.0	2447	18	29	01:31	52	4	3

Figure 3. Results from mbedTLS 2.9.0 using Ct-verif. The total lines of code (LOC) analyzed from the library required numerous annotations for public inputs and revision to accommodate the limitations of ct-verif. Its performance is measured in minutes. The number of positives reported is compared to the number of real high-risk and low-rsik errors in the source.

Conclusions

The primary objective of the research was to identify a practical tool to verify constant time code. Ideally, the practical interpretation of a sound and complete methodology would identify errors that leak secret information. Many positives is troubling for developers who wish to use the tool to repair errornous code. Many positives were caused by loops or variable sized arrays. This means that a more precise version of ct-verif that supports its current limitations is plausible for future work on a practical software verification tool.

References

[1] David Brumley and Dan Boneh. 2005. Remote timing attacks are practical. Computer Networks 48, 5 (2005), 701–716.

[2] J. B. Almeida, M. Barbosa, and G. Barthe. Verifying constant-time implementations. In Proceedings of the USENIX Security Symposium, 2016.