Benchmarking tools for verification of constant-time execution

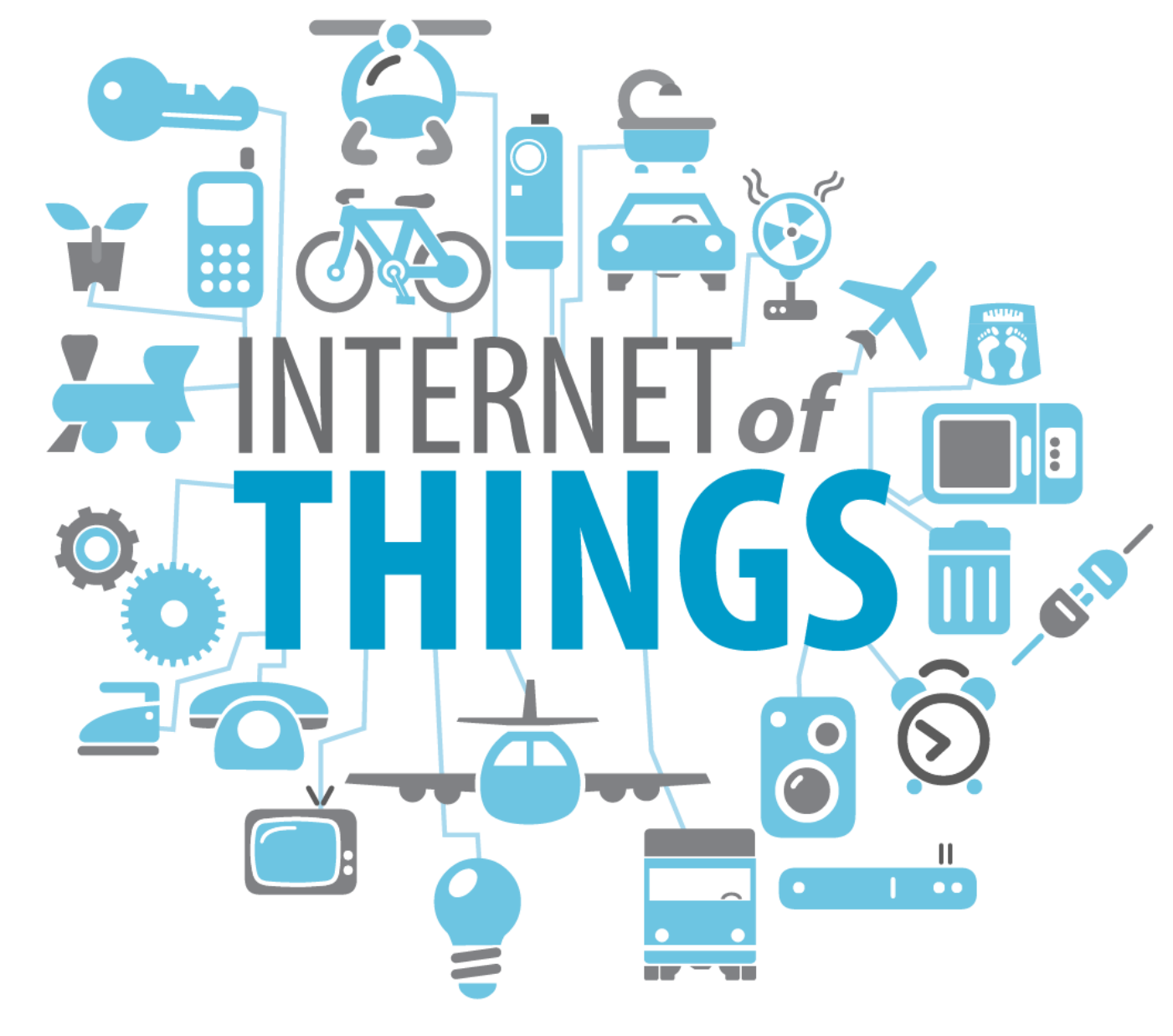
WTICG – X Workshop de Trabalhos de Iniciação Científica e de Graduação

Arthur Costa Lopes, **Diego F. Aranha**

6 de novembro de 2017

Instituto de Computação/Unicamp

Introduction



Introduction

Definition

The Internet of Things (IoT) is a scenario in which objects, animals or people are provided with unique identifiers and the ability to automatically transfer data over a network without requiring human-to-human or human-to-computer interaction.

Many applications:

- Smart cities (lighting, waste management, environment, traffic)
- Incident response (access control, detection of fire or radiation)
- Retail (supply chain control, logistics)
- Home automation (intrusion detection, smart spaces).

Important: Devices need to be small and pervasive, thus resource-constrained and limited tamper-resistance.

Motivation

Cryptography can mitigate security issues in embedded devices.

Security property	Technique	Primitive
Protecting data at rest	FS-level encryption	Block cipher
Protecting data in transit	Secure channel	Auth stream cipher
Secure software updates	Code signing	Digital signatures
Secure booting	Integrity/Authentication	Hash functions, MACs
Secure debugging	Entity authentication	Challenge-response
Device id/auth	Auth protocol	PKC
Key distribution	Key exchange	PKC

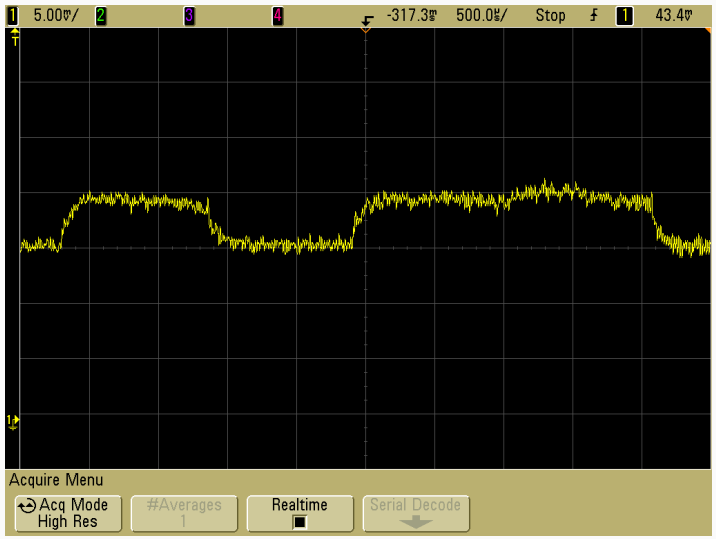
Several algorithms needed to implement primitives:

- Block and stream ciphers
- Hash functions
- Message Authentication Codes (MACs)
- Elliptic Curve Cryptography

Side-channel attacks

Computation leaks information correlated with data:

- **Execution time and cache timing**
- Power consumption and acoustic emanation
- Fault injection



Challenge

Algorithms and implementations need to be made **regular**, adding non-trivial performance penalty!

Side-channel attacks

Timing attacks

If execution time information correlates with secret bits, timing information can help an attacker to discover the secret.

Example: Hash-based password authentication.

```
EXTERN_C int __cdecl memcmp(const void *Ptr1,
        const void *Ptr2, size_t Count) {
    INT v = 0;
    BYTE *p1 = (BYTE *)Ptr1;
    BYTE *p2 = (BYTE *)Ptr2;

    while(Count-- > 0 && v == 0) {
        v = *(p1++) - *(p2++);
    }
    return v;
}
```

Defense: Constant-time execution.

Timing attacks and countermeasures

```
int util_cmp_const(const void *in, const void *pw,
        const size_t size) {
    const unsigned char *_a = (const unsigned char *) in;
    const unsigned char *_b = (const unsigned char *) pw;
    unsigned char result = 0;
    size_t i;

    for (i = 0; i < size; i++) {
      result |= _a[i] ^ _b[i];
    }

    return result; /* returns 0 if equal, nonzero otherwise */
}
```

Important: Use analysis tool to verify constant-time behavior!

Contributions

- Performance evaluation of constant-time verification tools.
- New representation format for constraint specification.
- Improvements in FlowTracker to parse new representation.
- Benchmarking database available at `https://github.com/arthurlopes/ctbench`.

Analysis tools

1. Static analysis (compile time):
   - Verifies source code, compiled binary or intermediate representation.
   - Cannot detect if instructions or external library functions are variable time.
   - Examples: *FlowTracker*, *ct-verif*.
2. Dynamic analysis (execution time):
   - Verifies code during actual execution in target platform.
   - Cannot process all possible inputs and detect corner cases.
   - Examples: *dudect*, *ctgrind*.

Static analysis

1. *FlowTracker*:
    - Creates an information flow graph from LLVM intermediate representation.
    - Checks if there is information flow from secret inputs to branches or memory accesses.
    - Limited to availability of source code (not external libraries).
2. *ct-verif*:
    - Requires source code to be annotated.
    - Converts LLVM intermediate representation to Boogie program.
    - Extracts provable statements about constant-time execution.

Dynamic analysis

1. *dudect*:
   - Executed the code a few times to sample execution time.
   - Performs statistical tests over the samples.
   - Cannot explore the full input size and might miss corner cases.

2. *ctgrind*:
   - Plug-in of the *Valgrind* memory debugger.
   - Marks secret information as not initialized.
   - Triggers memory safety issue when secret information is used in branches or array indexes.
   - Patch was ported to latest release (*3.13*).

Comparison among tools

Tool name	Type	Limitations	Advantages
ct-verif	Static	Preliminary and under development.	Provides formal guarantees at high-level.
FlowTracker	Static	Cannot check all dependencies automatically or detect microarchitecture effects.	High efficiency and full coverage through information flow analysis.
dudect	Dynamic	Cannot prove implementation is constant time or test all inputs.	Usability and detection of microarchitecture effects.
ctgrind	Dynamic	Cannot prove implementation is constant time or test all inputs.	Usability (easy to set up and run).

New representation

We developed an interoperable new representation for specifying secret inputs based on *FlowTracker* XML annotations.

Listing 1: XML file used to annotate code in Listing 1.

```
<functions>
  <sources>

    <function>
      <name>util_cmp_const</name>   <!--Function to be analyzed-->
      <return>false</return>    <!--Return value is not critical-->
      <public>
        <parameter>in</parameter>          <!--Input String -->
        <parameter>size</parameter>        <!--String length-->
      </public>
      <secret>
        <parameter>pw</parameter>          <!--Password-->
      </secret>
    </function>

  </sources>
</functions>
```

CTBench: a new benchmarking database

Library	Algorithm Type	Constant	Variant
BearSSL	Symmetric	18	4
	MAC	1	2
	Hash	3	5
	RSA	3	4
	ECC	0	4
dudect examples	AES	1	1
	ECC	1	1
	Others	1	1
NaCl	Authenticated encryption	6	0
	Hash	4	0
	Curve25519	1	0

Tabela 2: List of 60 algorithms in our benchmarking database, containing implementations from the BearSSL and NaCl cryptographic libraries and examples from the *dudect* dynamic analysis tool.

Experimental results

1. Obtain information to establish **ground truth**.
2. Start by running *FlowTracker* for all implementations.
3. Verify *dudect* results against *FlowTracker*.
4. Verify *ctgrind* results against *FlowTracker*.

Collected observations:

- Dynamic analysis tools performed as expected and detected variable-time code.
- There was a mismatch with FlowTracker due to `memcmp` function.

Conclusions and future work

Static and dynamic analysis tools can indeed help the software development process to detect timing side-channels.

Both approaches are needed for full coverage of program behavior. Annotations are an important assumption and must be done right.

For future work, we plan to **extend** the CTBench database with more implementations and improve usability to better assist the developer (i.e. integrate into development workflow and generate warning when external source code is not available).

Questions?

D. F. Aranha

dfaranha@ic.unicamp.br

@dfaranha