# Proving Crypto Implementations Secure

Proof techniques and tools

Manuel Barbosa
HASLab – INESC TEC
Faculty of Science – University of Porto
mbb@dcc.fc.up.pt
http://www.dcc.fc.up.pt/~mbb

June 2018

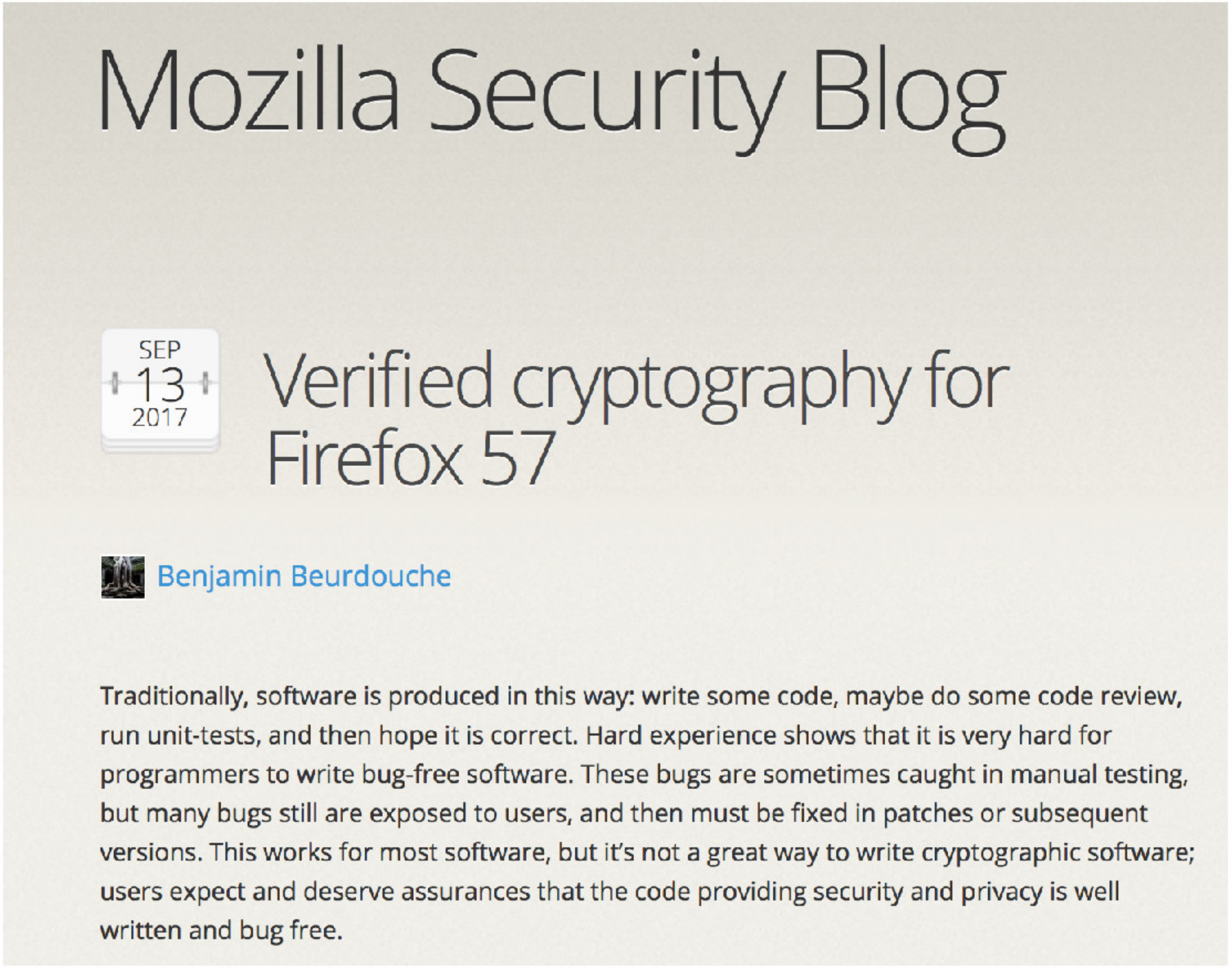
## Goal of this talk

Overview of different paths to obtain machine-level implementations provably secure against timing attackers

Selection of approaches which have been reported in the literature

Closer look at Jasmin

## Is formal verification of crypto taking off?



Mozilla Security Blog

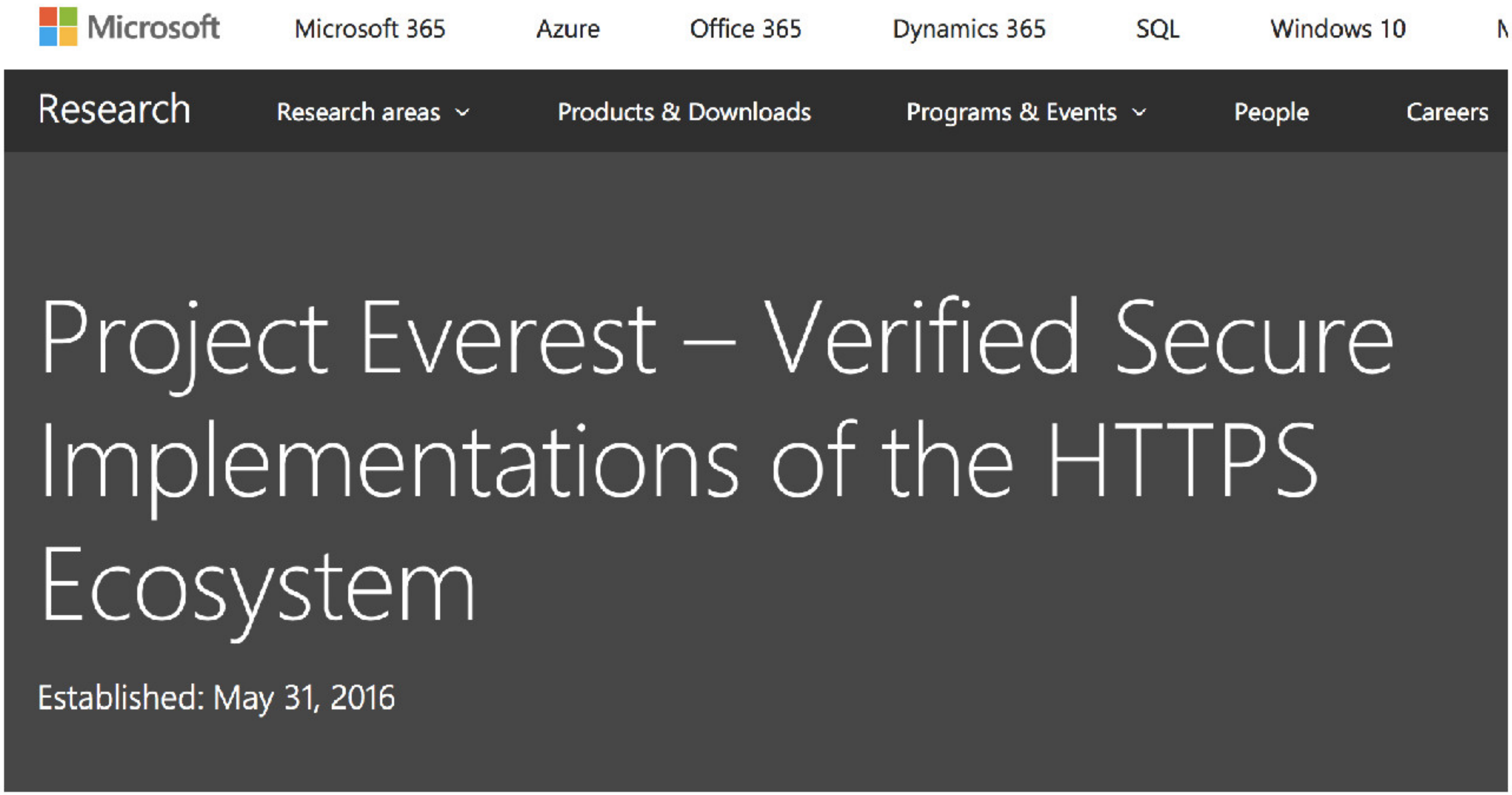
SEP 13 2017

Verified cryptography for Firefox 57

Benjamin Beurdouche

Traditionally, software is produced in this way: write some code, maybe do some code review, run unit-tests, and then hope it is correct. Hard experience shows that it is very hard for programmers to write bug-free software. These bugs are sometimes caught in manual testing, but many bugs still are exposed to users, and then must be fixed in patches or subsequent versions. This works for most software, but it's not a great way to write cryptographic software; users expect and deserve assurances that the code providing security and privacy is well written and bug free.

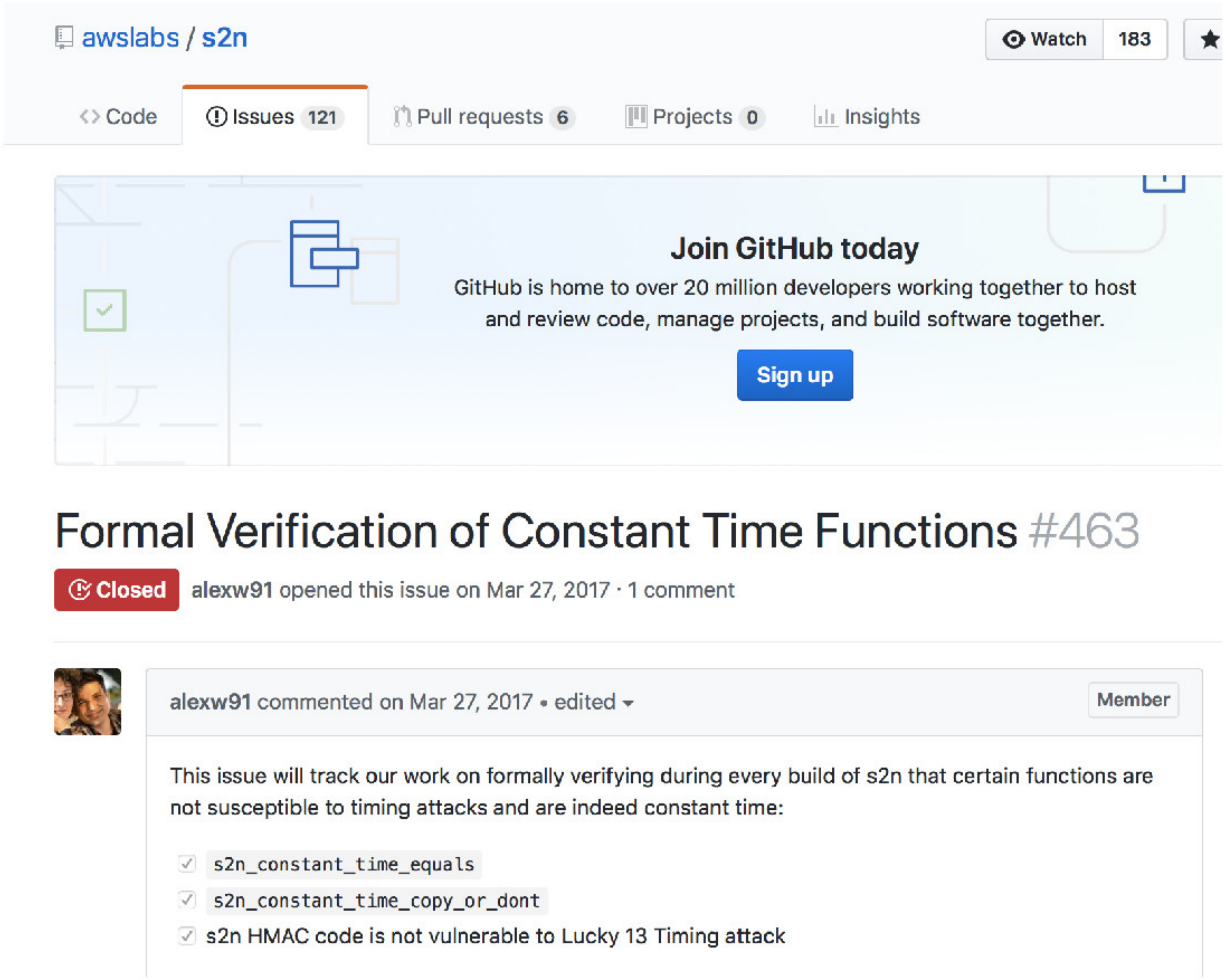
## Is formal verification of crypto taking off?



Project Everest – Verified Secure Implementations of the HTTPS Ecosystem

Established: May 31, 2016

# Is formal verification of crypto taking off?



**Join GitHub today**

GitHub is home to over 20 million developers working together to host and review code, manage projects, and build software together.

Formal Verification of Constant Time Functions #463

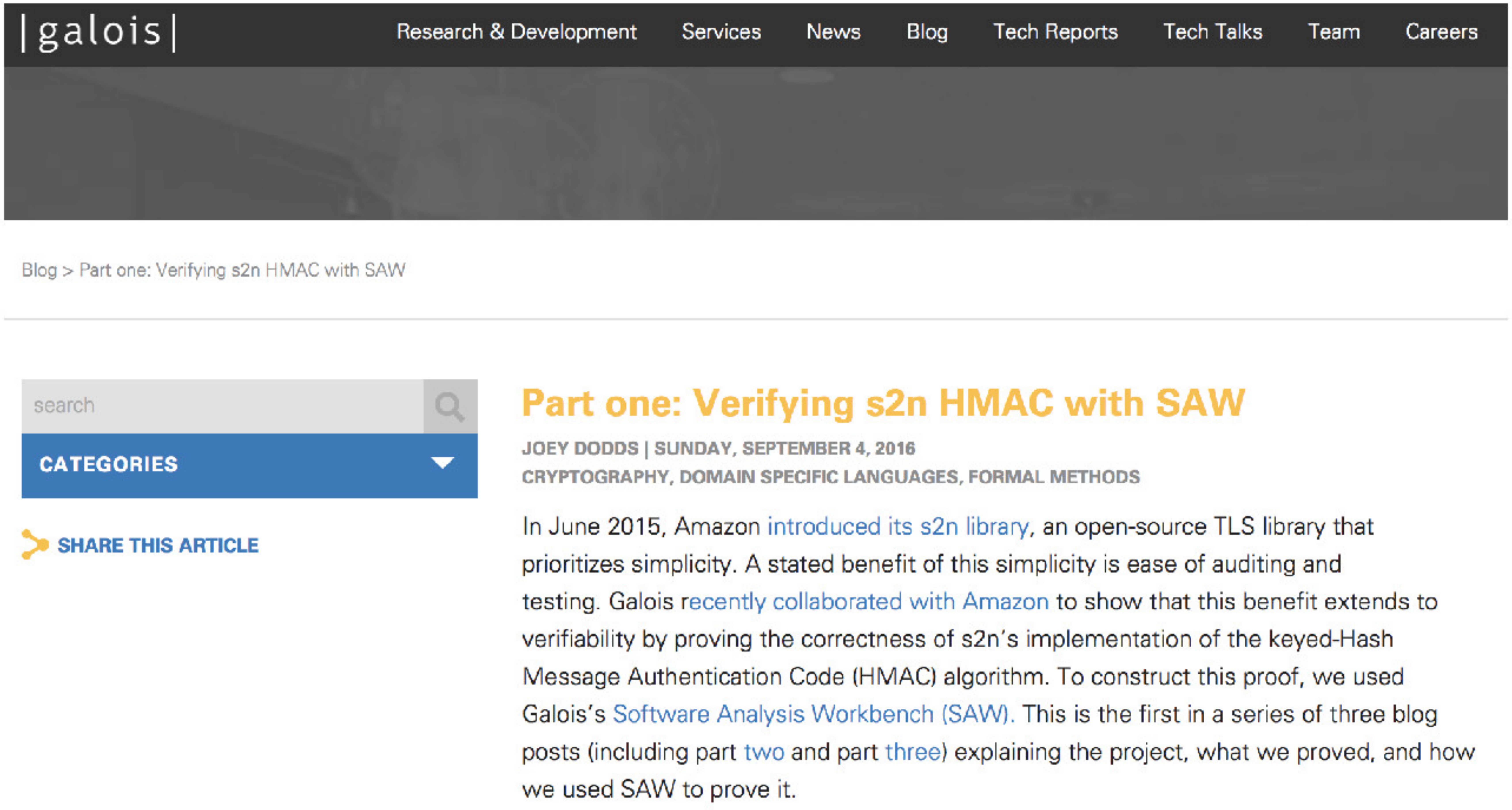
Closed alexw91 opened this issue on Mar 27, 2017 · 1 comment

alexw91 commented on Mar 27, 2017 • edited — Member

This issue will track our work on formally verifying during every build of s2n that certain functions are not susceptible to timing attacks and are indeed constant time:

- ✓ `s2n_constant_time_equals`
- ✓ `s2n_constant_time_copy_or_dont`
- ✓ s2n HMAC code is not vulnerable to Lucky 13 Timing attack

## Is formal verification of crypto taking off?



Blog > Part one: Verifying s2n HMAC with SAW

Part one: Verifying s2n HMAC with SAW

JOEY DODDS | SUNDAY, SEPTEMBER 4, 2016
CRYPTOGRAPHY, DOMAIN SPECIFIC LANGUAGES, FORMAL METHODS

In June 2015, Amazon introduced its s2n library, an open-source TLS library that prioritizes simplicity. A stated benefit of this simplicity is ease of auditing and testing. Galois recently collaborated with Amazon to show that this benefit extends to verifiability by proving the correctness of s2n's implementation of the keyed-Hash Message Authentication Code (HMAC) algorithm. To construct this proof, we used Galois's Software Analysis Workbench (SAW). This is the first in a series of three blog posts (including part two and part three) explaining the project, what we proved, and how we used SAW to prove it.

## Problem statement

Cryptographers write specifications in (very) high-level language

Several specialized tools can be used as entry point for such specifications

- EasyCrypt
- CryptoVerif
- F*
- Coq

Assume high-assurance that specifications are secure:

- Ideally machine-checked security proof
- Or paper proof and specification “looks OK”

**We want functionally correct CT-secure assembly**

## Recap

Recall from first lecture we proved $A \wedge B \Rightarrow C$

- $A$: Specification $S$ is secure in the SM
- $B$: Implementation $P$ is "valid" with respect to $S$
- $C$: Implementation $P$ is secure in the IM

We sketched a methodology to get implementation security:

- Cryptographers take care of $A$
- We want $C$: security as in $A$ plus no timing attacks
- It suffices to generate $P$ from $S$ such that
  - $P$ is functionally correct
  - $P$ is constant-time (CT) secure

Let us look at how this can be done in practice

## Is there a simple solution?

Recall our goal:

- Go from provably secure crypto specification
- To secure and timing-attack resilient machine code

Why doesn't someone write a compiler to do this?

Harder than it looks.

Next: various efforts in this direction.

## How much do we trust compilers?

Compilers designed and trusted to preserve functionality

*Not* designed to be aware of timing side-channels

Good tools exist to enforce constant-time at LLVM-level

- FlowTracker
- CT-verif (details later)

This is a good solution **if we trust clang**

- to preserve functionality from C to assembly
- to preserve CT-security from LLVM to assembly

How to get functionally correct C code?

- Trust some compiler to produce C from crypto spec
- We will see some examples below

## What if we don't trust compilers?

The CompCert compiler [Ler06] is a milestone in formal verification

It is able to produce efficient assembly code from C

It is proven correct in Coq

It has very good coverage of ANSI C

Efficiency is comparable to GCC -O1

Extensions to CompCert to handle intrinsics are subject of ongoing work

## Is CompCert enough?

CompCert is *not* proven to preserve constant-time

Almeida et al [CCS'12] gave an extension for CompCert

- If source code is PC-model secure (might not be constant-time due to memory accesses)
- Then assembly code is PC-model secure
- Uses translation validation
- Caveats: does not easily generalise to full CT security

Barthe et al [CCS'14] gave an extension for CompCert

- Includes type-based CT verifier at pre-assembly level
- Successful compilation implies non-interference
- Caveats: limited coverage and precision

**Still need to find a way to generate correct C code**

# Back to approaches and challenges

Next we will see how various works in the literature are tackling these challenges

We broadly categorize efforts in three classes:

- Code extraction
- Certified compilation
- Direct low-level proofs

Clarify what is being trusted to do what.

## Approach 1: Code extraction

EasyCrypt, Cryptoverif and F* support code extraction from high-level specification to functional language

- EasyCrypt and CryptoVerif to OCaml
- F* to F# or OCaml

Advantages:

- Specifications are written in functional operator language
- Extraction is essentially the identity function
- Compilers to C exist, so all done if we trust compilers
- Example: Yao's protocol by Almeida et al. [CCS'17]

Limitations:

- Need to trust compilers
- The generated assembly code is relatively slow

## Approach 2: Certified compilation

Several works use CompCert to go from C to assembly

They vary in how one obtains a correct C implementation

Common pattern:

- Refine specification, e.g, interactive theorem proving until it can be used as a specification (Hoare tripple) in C axiomatic semantics
- Example: specification uses integers and is proven equivalent to another specification using word-level representation.
- Prove C code satisfies the refined specification using the C axiomatic semantics (many options here, including gfverif, Frama-C, F*)

## Approach 2: Certified compilation

Instantiations:

- Almeida et al. [CCS'13]: Use EasyCrypt to refine RSA-OAEP specification and Frama-C to prove C implementation correct; mapping of EasyCrypt logic to axiomatic semantics is trusted
- Zinzindohoué et al [CCS'17] use F* to refine functional specification of NaCl and also to prove C-like imperative form correct; extraction to C is trusted
- Appel et al [Usenix'15, CCS'17] Use "Verifiable C" within Coq to do all the steps for HMAC and HMAC-DBRG.

## Approach 2: Certified compilation

Advantages:

- No trust in compilers
- Code can be faster than what is obtained from extraction

Limitations:

- A lot of human effort involved compared to extraction
- No guarantee that compilation will preserve CT-security (unclear what one can do with tools like ct-verif and FlowTracker)
- Poor support to directly CT-verify assembly code

## Technology summary for approaches #1 and #2

We have great tools for automatic

- CT-verification at pre-assembly level (LLVM)
- Generation of certified assembly code from C (CompCert)

If we trust compilers, then good solutions exist.

If we don't trust compilers we lack automatic tools to prove:

- C implementations functionally correct
- Certified assembly-level code CT-secure

Current limitations:

- Functional correctness proofs are *very* hard even at C level
- Existing C compilers, even certified ones, are not proven to preserve CT-security
- Certified pre-assembly level CT-security checkers have poor coverage/precision

## Approach 3: Direct low-level proof

Emerging approach to obtain both CT-security and functional correctness verification at low level

- **Carry out all verification at (pre-)assembly level**
- Effort involved in functional correctness proofs comparable to that required at C level
- Constant-time verification performed close to machine level
- Preliminary work demonstrated feasibility
  cf. Chen et al [CCS'14]
- Two recently proposed platforms go in this direction: Vale and Jasmin

This approach will be covered in the rest of the talk

## Approach 3: Direct low-level

Vale [Usenix'17] is a tool that permits directly verifying assembly code

- Annotated assembly code is translated to Dafny
- Dafny is a general purpose verification language
- Assembly semantics are written as Dafny code
- Dafny then works as a VCGen relying on Z3
- CT-verification is performed using data-flow analysis

## Approach 3: Direct low-level

Jasmin [CCS'17] is a framework for writing low-level high-speed cryptographic code

- New language inspired in qhasm, but with some additional high-level features
- Full programmer control, but functional verification made simpler by high-level constructs
- Certified compiler guaranteed to preserve functionality and CT-security
- ct-verif style verifier at source level provides full power of self-composition

## Part 3: Short Introduction to Jasmin

Implementing crypto requires a subtle equilibrium

- Correct
- Fast
- Secure (side-channel free)

This shows gap between C and assembly

Programmers like C

- Portable
- Convenient software-engineering abstractions
- Readable, maintainable

Often they end up programming in assembly

- Efficiency
- Control (instruction selection and scheduling)
- Precise semantics

## What is Jasmin?

A language

- with a familiar syntax
- seemingly high-level constructs
- allows for “assembly in the head” programming

A formal semantics

A **predictable** compiler proven correct in Coq

Tooling for proofs of safety, correctness and CT-security

## Jasmin team

`https://github.com/jasmin-lang/jasmin`

José Bacelar Almeida

Manuel Barbosa

Gilles Barthe

Arthur Blot

Benjamin Grégoire

Vincent Laporte

Tiago Oliveira

Hugo Pacheco

Benedikt Schmidt

Pierre-Yves Strub

## Jasmin "Hello World!" (constant-time swapping)

```
param int n = 4;

inline
fn cswap(stack u64[n] x, stack u64[n] y, reg u64 swap)
⟶ stack u64[n], stack u64[n] {
  reg u64 tmp1, tmp2, mask;
  inline int i;
  mask = swap * 0xffffffffffffffff;
  for i = 0 to n {
    tmp1 = x[i];
    tmp1 ^= y[i];
    tmp1 &= mask;
    tmp2 = x[i];
    tmp2 ^= tmp1;
    x[i] = tmp2;
    tmp2 = y[i];
    tmp2 ^= tmp1;
    y[i] = tmp2;
  }
  return x, y;
}
```

Zero-cost abstractions

- Variable names
- Global parameters
- Arrays
- Loops
- Inline functions (with custom calling conventions)

## Control down to architecture level

```
reg bool cf;
reg u64 addt0, addt1, t10, t11, t12, t13;
// ...
t10 = [workp + 4 * 8];
t11 = [workp + 5 * 8];
t12 = [workp + 6 * 8];
t13 = [workp + 7 * 8];
// ...
cf, t10 += [workp + 8 * 8];
cf, t11 += [workp + 9 * 8] + cf;
cf, t12 += [workp + 10 * 8] + cf;
cf, t13 += [workp + 11 * 8] + cf;
addt0 = 0;
addt1 = 38;
addt1 = addt0 if ! cf;
```

- Direct memory access
- The carry flag is an ordinary boolean variable

```
reg u64 i, j;
stack u64 is, js;
// ...
j = 62;
i = 3;
while (i >=s 0) {
    is = i;
    // ...
    while (j >=s 0) {
        js = j;
        // ...
        j = js; j -= 1;
    }
    j = 63; i = is; i -= 1;
}
```

- Control over loop unrolling
- Control over spilling

## Beyond correctness

Automatic proof of memory safety

- Translate to Dafny
- Reuse the Dafny · Boogie infrastructure

Automatic proof of constant-time

- Translate to Dafny then Boogie
- Adapt the technique from the CT-verif tool
- Build a product program and check that the product is safe

Jasmin for secure programming

- Infrastructure for automatic checks
- The compiler preserves functional correctness, CT
- Known verification techniques apply to Jasmin programs

## Running Jasmin programs

Jasmin programs as libraries

- Compliant with standard ABI
- Link with your own programs
  written in assembly, C, OCaml, Rust. . .