

Introducción a la Verificación Formal

Clase 1 – 26/08/2025

Administrivia

- Única materia previa: ALP
 - MUY recomendada: IS1
 - Siempre pueden cursar. Para rendir tienen que tener ALP aprobado.
- Nos vemos todos los **jueves a las 3PM.**
 - Las clases se graban. En teoría tienen acceso con la invitación.
 - Luego de cada clase hay una parte del libro para leer y ejercicios para hacer.
- Final de materia: TP final a (semi)elección, a definir más adelante
- Clases invitadas: vayan viendo si les interesa algo en particular

Software Confiable

- Objetivamente, la ingeniería de software es la peor ingeniería
 - Sin garantías reales
 - Fallos frecuentes (evidentemente)
 - Poca sistematización
 - Testing como método de calidad fundamental
- Por otro lado, el *bloat* es innegable
 - La containerización, chequeos dinámicos, etc, no son gratis

```
hashOut.data = hashOut + SSL_SHA1_DIGEST_LEN,  
hashOut.length = SSL_SHA1_DIGEST_LEN;  
if ((err = SSLFreeBuffer(&hashCtx)) != 0)  
    goto fail;  
  
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)  
    goto fail;  
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)  
    goto fail;  
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)  
    goto fail;  
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)  
    goto fail;  
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)  
    goto fail;  
  
err = sslRawVerify(ctx,  
                  ctx->peerPubKey,  
                  dataToSign,                               /* plaintext */  
                  dataToSignLen,                           /* plaintext length */  
                  signature,  
                  signatureLen);
```

uu-:---F1 sslKeyExchange.c 30% L602 (C/l Abbrev Isearch)-----

[Apple finally fixes 'gotofail' OS X security hole - CNET](#)

Salón de la Fama de Bugs

- 1994: Bug FDIV del Pentium
 - Error de cálculo en divisiones de punto flotante, sin detección por >1 año.
- 1996: Vuelo Ariane V88
 - Overflow entero (+ otros factores) causa destrucción del cohete + 4 satélites.
- 2012: Heartbleed
 - Falla de validación de entrada + bug de memoria filtra secretos remotamente.
- 2014: Hackeo y Bancarrota de MtGox
 - Maleabilidad del protocolo de Bitcoin (+ otros factores) lleva a “robo” de 650,000 BTC.
- 2018: Spectre y Meltdown
 - Side channel en el procesador afecta aislamiento entre procesos. (Todavía estamos pagando las mitigaciones con pérdida de perf.)

“Prendimos fuego la casa y ¿adivinen qué?
Nosotros somos los bomberos.”
- Charles E. Leiserson

¿Qué hacemos?






- Gran parte de la industria se enfoca en mejorar la calidad del software
 - Testing sistematizado (fuzzing, property-based testing)
 - Testing de penetración
 - Analizadores estáticos
 - Chequeos dinámicos
 - Nuevos lenguajes (funcionales, DSLs, Rust)
- Esto tiene algún éxito
 - Ver este [informe de la casa blanca](#) (Feb 2024) sugiriendo el uso de lenguajes memory-safe y métodos formales.
- Pero, estos métodos **no dan garantías de correctitud**





Formal Methods

Even if engineers build with memory safe programming languages and memory safe chips, one must think about the vulnerabilities that will persist even after technology manufacturers take steps to eliminate the most prevalent classes. Given the complexities of code, testing is a necessary but insufficient step in the development process to fully reduce vulnerabilities at scale. If correctness is defined as the ability of a piece of software to meet a specific security requirement, then it is possible to demonstrate correctness using mathematical techniques called *formal methods*. These techniques, often used to prove a range of software outcomes, can also be used in a cybersecurity context and are viable even in complex environments like space. While formal methods have been studied for decades, their deployment remains limited; further innovation in approaches to make formal methods widely accessible is vital to accelerate broad adoption. Doing so enables formal methods to serve as another powerful tool to give software developers greater assurance that entire classes of vulnerabilities, even beyond memory safety bugs, are absent.

Métodos Formales

- Los métodos formales son formas **estáticas** de analizar software basadas en sistemas formales de lógica.
 - Formal: basado en reglas de inferencia. E.g. Si “ $P \rightarrow Q$ ” y “ P ” entonces “ Q ”. No puede apelarse a conocimiento sobre “qué” son P/Q ni a razonamientos fuera del sistema.
- Hay dos campos principales
 - Sistemas de tipos Fuertes (Tipado Dependiente, IFC, ...)   
 - El lenguaje de programación tiene un sistema de tipos que permite expresar propiedades más complejas
 - E.g. “ f es una función monótona” “ n es primo”
 - El sistema de tipos luego acepta o rechaza el programa
 - Lógicas de Programa (e.g. Lógica de Hoare, Lógica de Separación, ...)  
 - El programador escribe el programa, tal vez con anotaciones, y la herramienta computa *condiciones de verificación* (VCs) que implican la correctitud.
 - El programador luego intenta demostrar esas VCs, ya sea con métodos manuales o automáticos, o ajustando el programa y sus anotaciones
- En ambos casos el programa se desarrolla junto a su prueba
 - La prueba se **chequea por máquina**
- Se eliminan **clases** enteras de bugs posibles, con certeza
- Sigue siendo “caro”: requiere expertos y mucho tiempo

Algunos éxitos en Verificación Formal

- Teorema de los cuatro colores
 - Demostrado por Wolfgang Haken y Kenneth Appel (padre del Appel que conocen) en la primera prueba matemática por computadora. [Video](#)
- Feit-Thompson: teorema sobre grupos, ~250 páginas
- [Mathlib](#): librería de matemática en Lean
- [CompCert](#): Compilador de C verificado
- [seL4](#): Microkernel
- [DeepSpec](#)
- [HACL*](#): Criptografía verificada 
- [EverParse](#): Parsers y serializadores verificados 

Program Proofs in F* for Billions of Unsuspecting Users

Automated parsing
untrusted data with proofs
in Hyper-V/VMSwitch



Cloud infrastructure

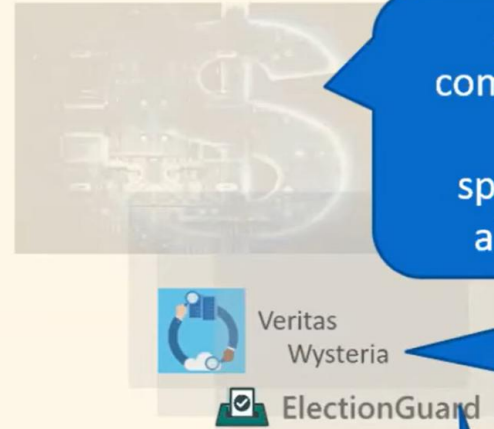
Verified Merkle trees
for Enterprise
blockchains

Verified
cryptography in the
Linux kernel



Secure communications

Quic transport,
MSQuic in Windows,
Verified crypto in Firefox,
mbedTLS,
Signal in Wasm,
Wireguard, ...



High-value domains:
Fintech, verifiable
computing, ...

Some Fintech
companies rewriting
core trading
specifications and
algorithms in F*

Correctness of
Verifiable
computations

Core crypto in
ElectionGuard



F*: Programación orientada a las pruebas



- F* es un lenguaje funcional de la familia ML
 - Estricto (lo opuesto a lazy)
 - Alto orden (lambdas)
 - Polimórfico, con inferencia de tipos (Hindley-Milner)
- Lo nuevo:
 - Refinamientos
 - Tipos dependientes
 - Descarga de VCs por SMT solver
 - Extensionalidad
 - Sistema de efectos

```
(** The type of non-negative integers *)  
type nat = i: int{i >= 0}
```

Sintaxis básica

- Similar a OCaml
 - **val**: declaración de tipo
 - **let**: definición
 - **list a**: tipo de listas de **a**
 - **match** para pattern matching (en lugar de ecuaciones como en Haskell/Agda)
- Recursión explícita con **let rec**
 - El archivo va en orden.
- No hay comprensión de listas

```
val map : (int -> bool) -> list int -> list bool
let rec map f xs =
  match xs with
  | [] -> []
  | x::xs -> f x :: map f xs
```

Factorial en F*

```
let rec fac (x:nat) : nat =  
  if x = 0 then 1 else x * fac (x-1)
```

- Parece simple, pero requiere:
 - Demostrar que 1 es natural
 - Demostrar que si x es natural y no es cero (condición de rama), $x-1$ es natural
 - Si x es natural y **fac (x-1)** es natural, entonces $x * \mathbf{fac (x-1)}$ es natural
 - Terminación: la recursión eventualmente termina
- La mayor parte de este trabajo lo realiza [Z3, un SMT solver](#).

Refinamientos

- Subtipos definidos por una base y una formula l3gica
- Para chequear que un **x** tiene tipo **(y:t{phi})**
 - Primero, **x** debe tener tipo **t** (que puede ser un refinamiento)
 - La proposici3n **phi[x/y]** debe ser cierta
- Dualmente, si tenemos **x** de tipo **(y:t{phi})**, podemos asumir ambos puntos
 - Esto implica que, por ejemplo, siempre puede usarse un natural donde se espera un entero. No hacen falta coerciones (vs Coq, Agda).

```
(** The type of non-negative integers *)
type nat = i: int{i >= 0}

(** The type of positive integers *)
type pos = i: int{i > 0}

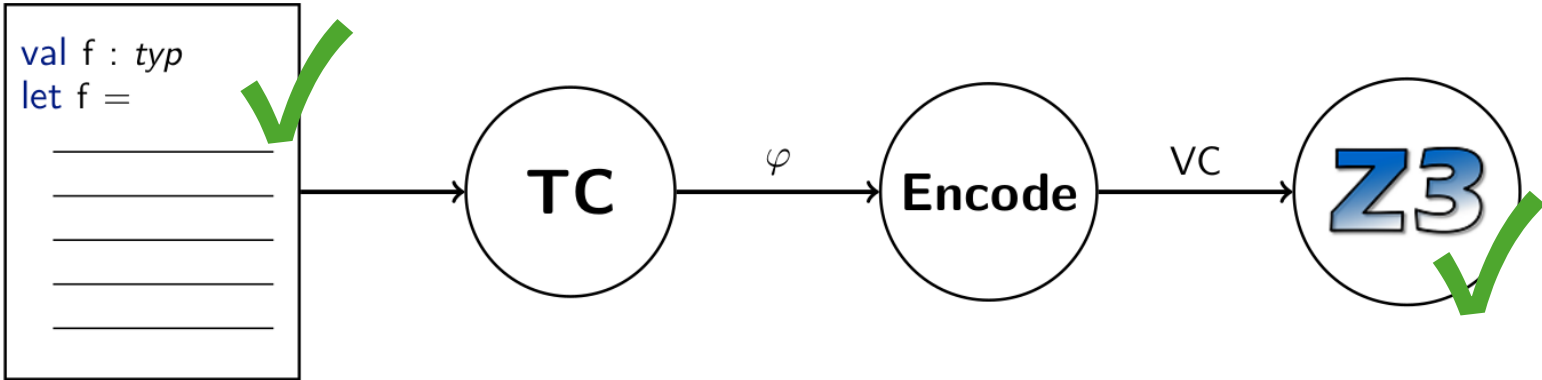
(** The type of non-zero integers *)
type nonzero = i: int{i <> 0}
```

```
let addnat (x y : nat) : int = x + y
```

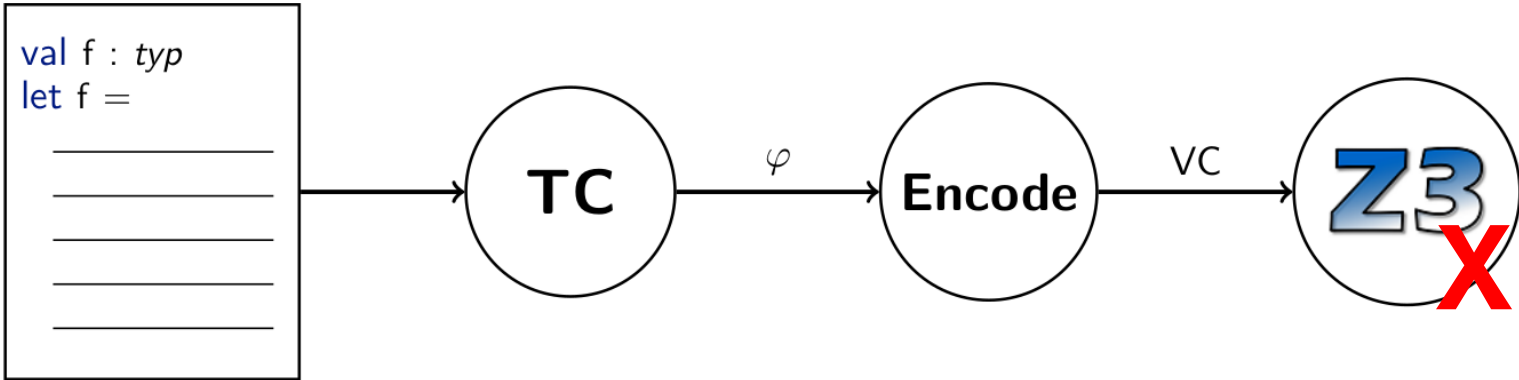
```
let debilitar (f : nat -> nat) : int -> int
```

```
let debilitar (f : int -> nat) : nat -> int = f
```

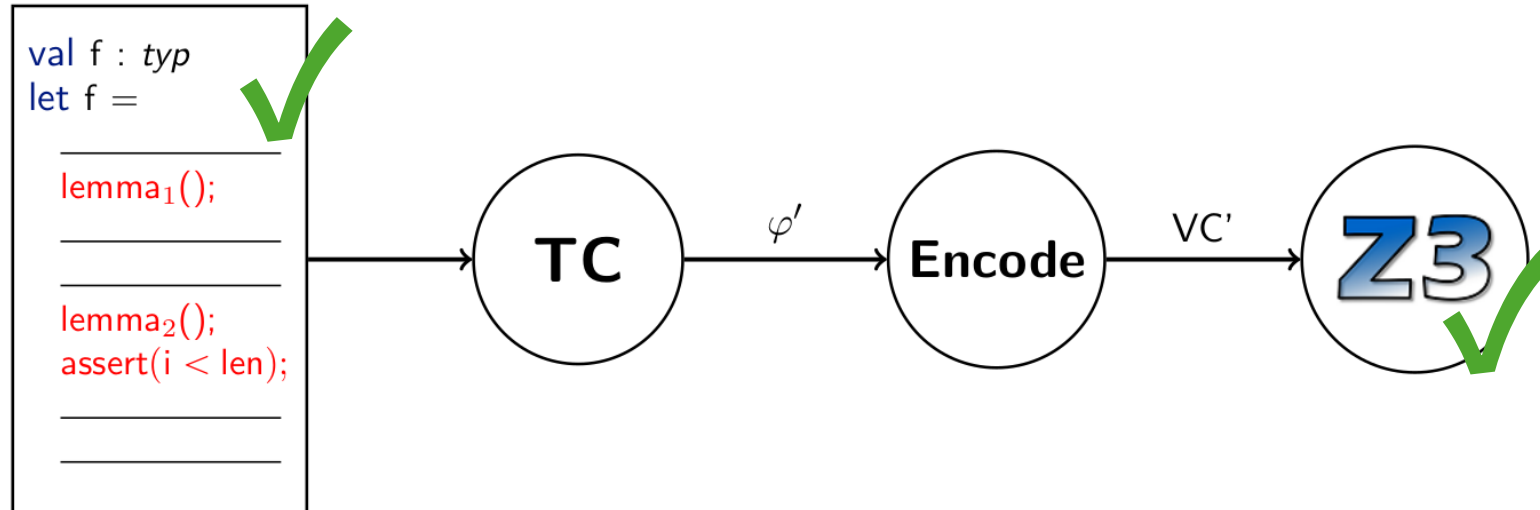
Verificación “auto-activa”



Verificación “auto-activa”



Verificación “auto-activa”



(demo)

Extracción: ejecutando F*

- (No vamos a ver mucho.)
- Los programas en F* se pueden *extraer* a OCaml, y luego usar una toolchain normal para compilar y ejecutar.
 - Uso de garbage collector
- Low*: un DSL de “bajo-nivel” embebido en F*
 - Código de primer orden (e.g. sin lambdas)
 - Pruebas arbitrariamente complejas (total se borran)
 - Manejo manual de memoria
 - Extrae a C via [Karamel](#), sin garbage collector
- Steel y Pulse:
 - Lógica de separación en F*
 - De alguna forma la evolución de Low*, pero más escalable
 - Steel extrae a C. Pulse extrae a OCaml/C/Rust

Disclaimer: TCB

¿Es correcta la lógica de F*?

¿Es correcta la *implementación* de F*?

¿Es correcto el SMT solver?

¿Es correcto el proceso de extracción?

- Vamos a asumir que **sí**. Responder estas preguntas está fuera del alcance de la materia.
- Esto pone a F* y Z3 en nuestra *trusted computing base* (TCB). Si algo en la TCB no cumple su especificación, perdemos las garantías sobre el programa. En general la TCB también incluye al hardware, librerías, runtime, etc.
- Así y todo, es extremadamente raro que un fallo en, e.g., la lógica de F* se alinee perfectamente para *secondar* un bug *real*. Las garantías que se obtienen en la práctica siguen siendo altas aun sin verificar las herramientas.

- [Self-certification: Bootstrapping certified typecheckers in F* with Coq](#)

- [A Verified Implementation of the DPLL Algorithm in Dafny](#)

- [MetaCoq | Website of the MetaCoq Project](#)

- [Gagallium : How I found a bug in Intel Skylake processors](#)

Tareas

- Preparar VS Code siguiendo instrucciones
 - [mtzguido/intro-verif-25](https://github.com/mtzguido/intro-verif-25)
- Leer: capítulos 1 y 2 del [libro](#)
- Completar archivo Clase1.fst