Functional Programming for Securing Cloud and Embedded Environments

Abhiroop Sarkar
Chalmers University

My Research

“Securing Digital Systems through Programming Language Techniques”

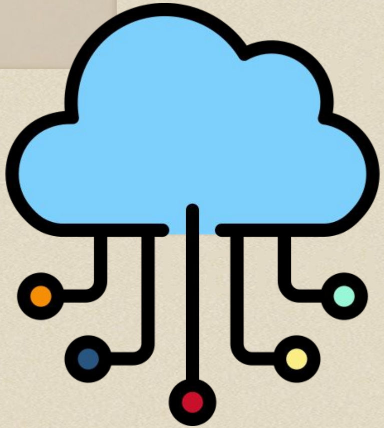
My Research

"Securing Digital Systems Programming Language Techniques"

My Research

Digital Systems

My Research



Digital Systems

Cloud Computing
- Multi-Tenant
- Large *Trust* Boundary

My Research

Digital Systems

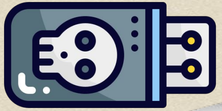
Embedded Systems

- Resource Constrained
- *Reactive* systems
- Time-bound systems

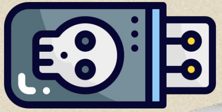
My Research

"Securing Digital Systems

Stuxnet
computer worm




MacBook



Microsoft Security Bulletin MS10-046 - Critical

Vulnerability in Windows Shell Could Allow Remote Code Execution (2286198)

Microsoft Security Bulletin MS10-061 - Critical

Vulnerability in Print Spooler Service Could Allow Remote Code Execution (2347290)

OS Remote Code Execution

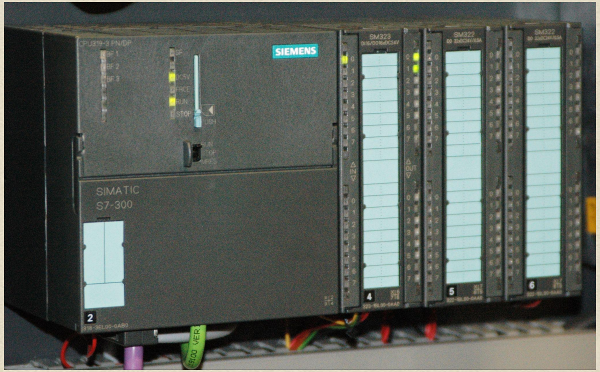
OS Remote Code Execution

Detect vulnerable PLC Driver otherwise hide

OS Remote Code Execution

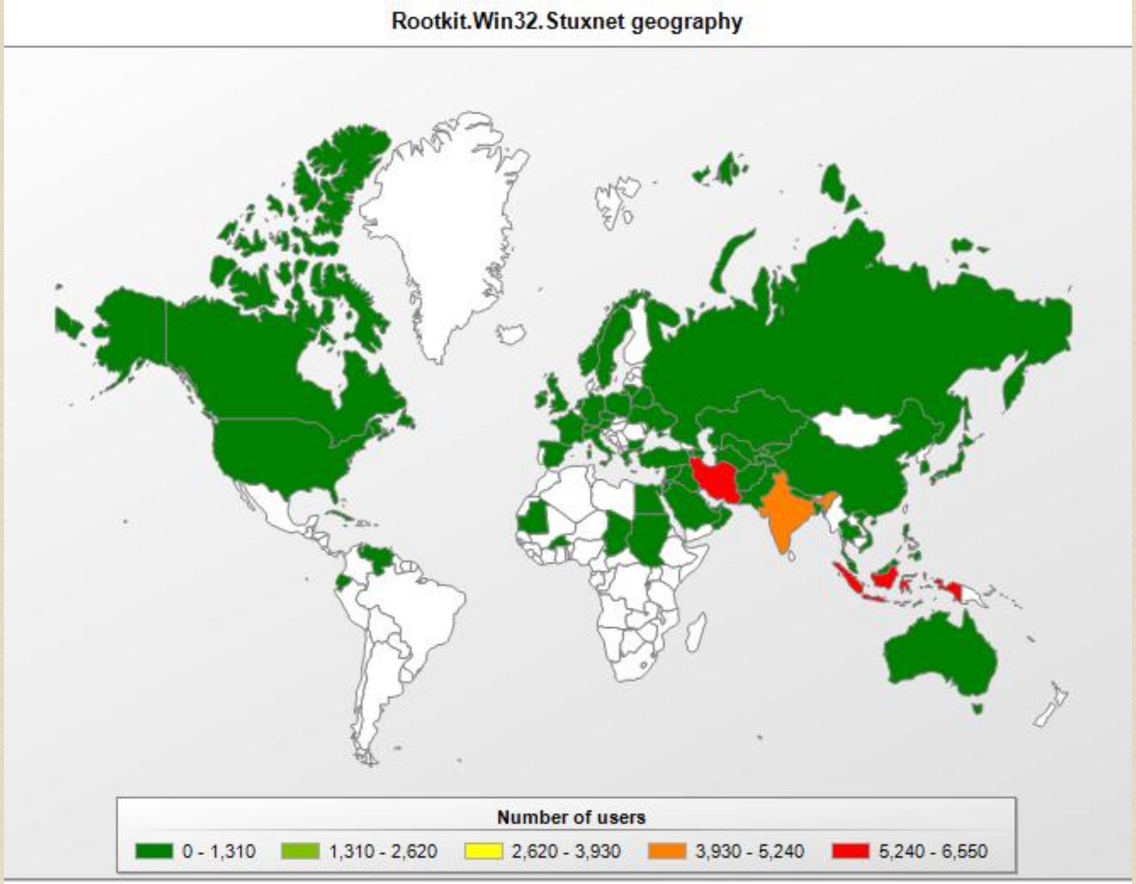
SIEMENS WinCC® Professional

Detect vulnerable PLC Driver otherwise hide



Attack Siemens PLC 807 - 1210Hz

Stuxnet Impact



ATTACKER MODELS

Attacker Model 1

TRUST

in the OS and other low-level software

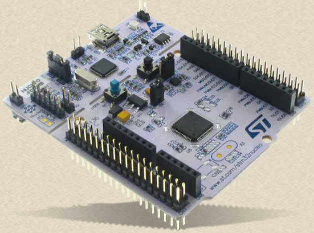
ATTACKER MODELS

Attacker Model 1

TRUST

in the OS and other low-level software

Attacker Model 2



MEMORY UNSAFETY

to accommodate resource constraints

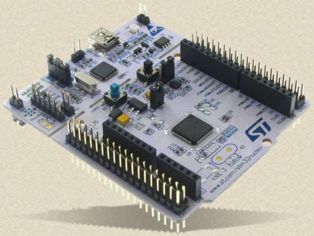
ATTACKER MODELS

Attacker Model 1

TRUST

in the OS and other low-level software

Attacker Model 2



MEMORY UNSAFETY

to accommodate resource constraints

My Research

“Securing Digital Systems

”

My Research

"Securing Digital Systems through Programming Language Techniques"

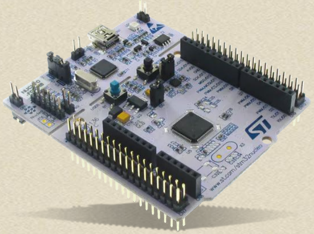
ATTACKER MODELS

Attacker Model 1

TRUST

in the OS and other low-level software

Attacker Model 2

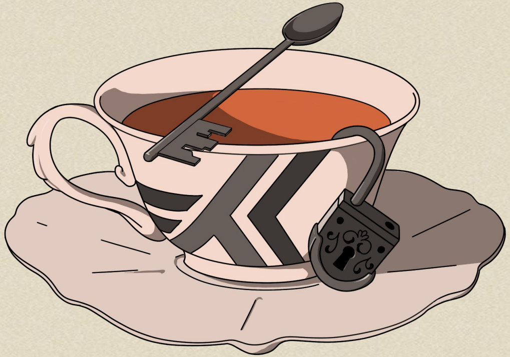


MEMORY UNSAFETY

to accommodate resource constraints

Contributions

Attacker Model 1



HasTEE⁺

for reducing ***trust*** on low-level software

Attacker Model 2

SynchronVM

for ***memory-safe***, ***soft real-time*** embedded systems

Part I

HasTEE⁺

HasTEE: Programming Trusted Execution Environments with Haskell

Abhiroop Sarkar
Chalmers University
Gothenburg, Sweden
sarkara@chalmers.se

Robert Krook
Chalmers University
Gothenburg, Sweden
krookr@chalmers.se

Alejandro Russo
Chalmers University
Gothenburg, Sweden
russo@chalmers.se

Koen Claessen
Chalmers University
Gothenburg, Sweden
koen@chalmers.se

Abstract

Trusted Execution Environments (TEEs) are hardware enforced memory isolation units, emerging as a pivotal security solution for security-critical applications. TEEs, like Intel SGX and ARM TrustZone, allow the isolation of confidential code and data within an untrusted host environment, such as the cloud and IoT. Despite strong security guarantees, TEE adoption has been hindered by an awkward programming model. This model requires manual application partitioning and the use of error-prone, memory-unsafe, and potentially information-leaking low-level C/C++ libraries.

We address the above with *HasTEE*, a domain-specific language (DSL) embedded in Haskell for programming TEE applications. HasTEE includes a port of the GHC runtime for the Intel-SGX TEE. HasTEE uses Haskell's type system to automatically partition an application and to enforce *Information Flow Control* on confidential data. The DSL, being embedded in Haskell, allows for the usage of higher-order functions, monads, and a restricted set of I/O operations to write any standard Haskell application. Contrary to previous work, HasTEE is lightweight, simple, and is provided as a *simple security library*; thus avoiding any GHC modifications. We show the applicability of HasTEE by implementing case studies on federated learning, an encrypted password wallet, and a differentially-private data clean room.

Keywords: Trusted Execution Environment, Haskell, Intel SGX, Enclave

ACM Reference Format:
Abhiroop Sarkar, Robert Krook, Alejandro Russo, and Koen Claessen. 2023. HasTEE: Programming Trusted Execution Environments with Haskell. In *Proceedings of the 16th ACM SIGPLAN International Haskell Symposium (Haskell '23), September 8–9, 2023, Seattle, WA, USA*. ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3609026.3609731

1 Introduction

Trusted Execution Environments (TEEs) are an emerging design of hardware-enforced memory isolation units that aid in the construction of security-sensitive applications [Mulligan et al. 2021; Schneider et al. 2022]. TEEs have been used to enforce a strong notion of *trust* in areas such as confidential (cloud-)computing [Baumann et al. 2015; Zegzhda et al. 2017], IoT [Lesjak et al. 2015] and Blockchain [Bao et al. 2020]. Intel and ARM each have their own TEE implementations known as Intel SGX [Intel 2015] and ARM TrustZone [ARM 2004], respectively. Principally, TEEs provide a *disjoint* region of code and data memory that allows for the physical isolation of a program's execution and state from the underlying operating system, hypervisor, and I/O peripherals. For

Haskell Symposium 2023

HasTEE^{+}: Confidential Cloud Computing and Analytics with Haskell

Abhiroop Sarkar[0000-0002-8991-9472] and Alejandro Russo[0000-0002-4338-6316]

Chalmers University, Gothenburg, Sweden
{sarkara,russo}@chalmers.se

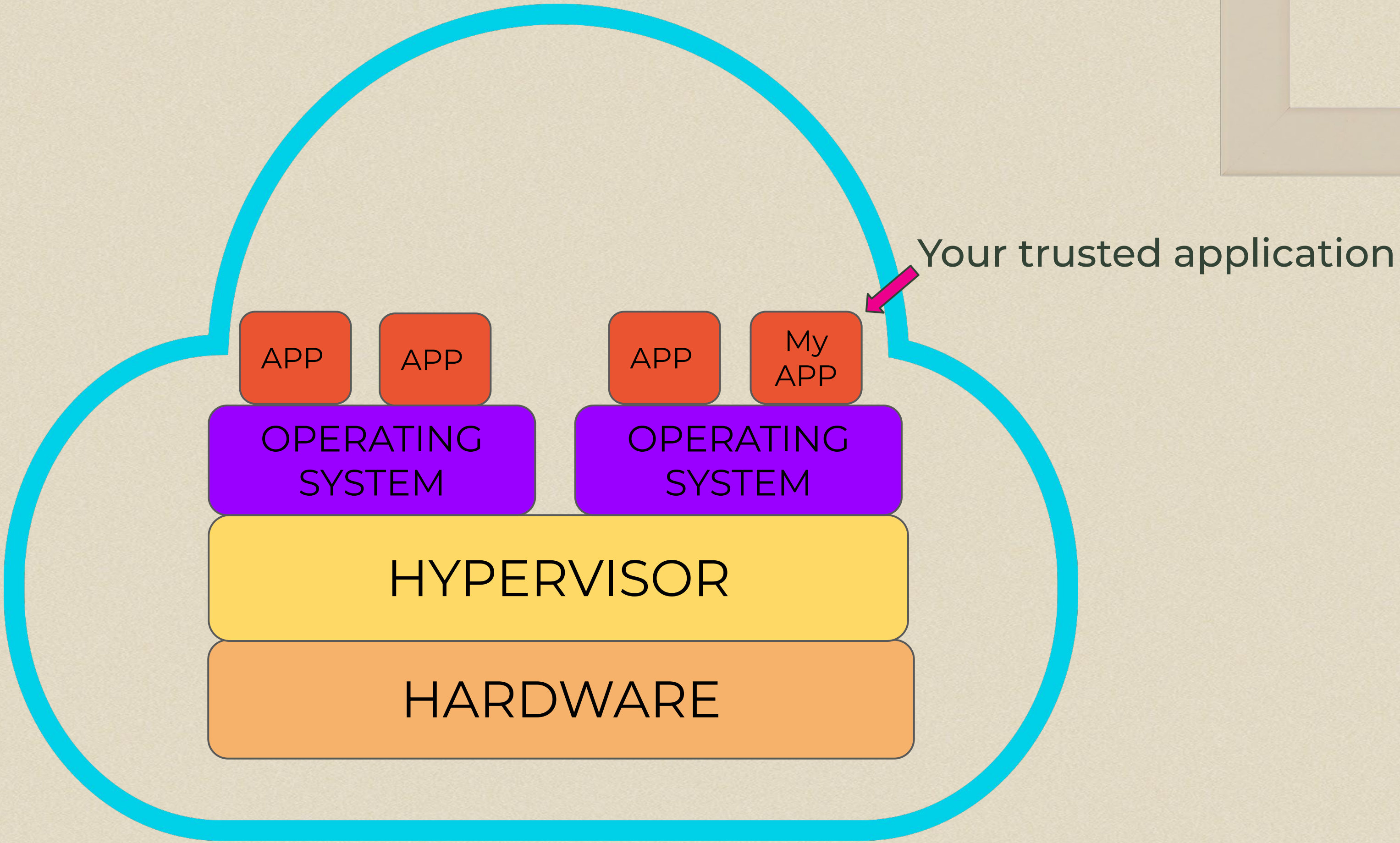
Abstract. Confidential computing is a security paradigm that enables the protection of confidential code and data in a co-tenanted cloud deployment using specialized hardware isolation units called Trusted Execution Environments (TEEs). By integrating TEEs with a Remote Attestation protocol, confidential computing allows a third party to establish the integrity of an *enclave* hosted within an untrusted cloud. However, TEE solutions, such as Intel SGX and ARM TrustZone, offer low-level C/C++-based toolchains that are susceptible to inherent memory safety vulnerabilities and lack language constructs to monitor explicit and implicit information-flow leaks. Moreover, the toolchains involve complex multi-project hierarchies and the deployment of hand-written attestation protocols for verifying *enclave* integrity.
We address the above with HasTEE^{+}, a domain-specific language (DSL) embedded in Haskell that enables programming TEEs in a high-level language with strong type-safety. HasTEE^{+} assists in multi-tier cloud application development by (1) introducing a *tierless* programming model for expressing distributed client-server interactions as a single program, (2) integrating a general remote-attestation architecture that removes the necessity to write application-specific cross-cutting attestation code, and (3) employing a dynamic information flow control mechanism to prevent explicit as well as implicit data leaks. We demonstrate the practicality of HasTEE^{+} through a case study on confidential data analytics, presenting a data-sharing pattern applicable to mutually distrustful participants and providing overall performance metrics.

Keywords: Confidential Computing · Trusted Computing · Trusted Execution Environments · Information Flow Control · Attestation · Haskell.

Under Submission ESORICS 2024

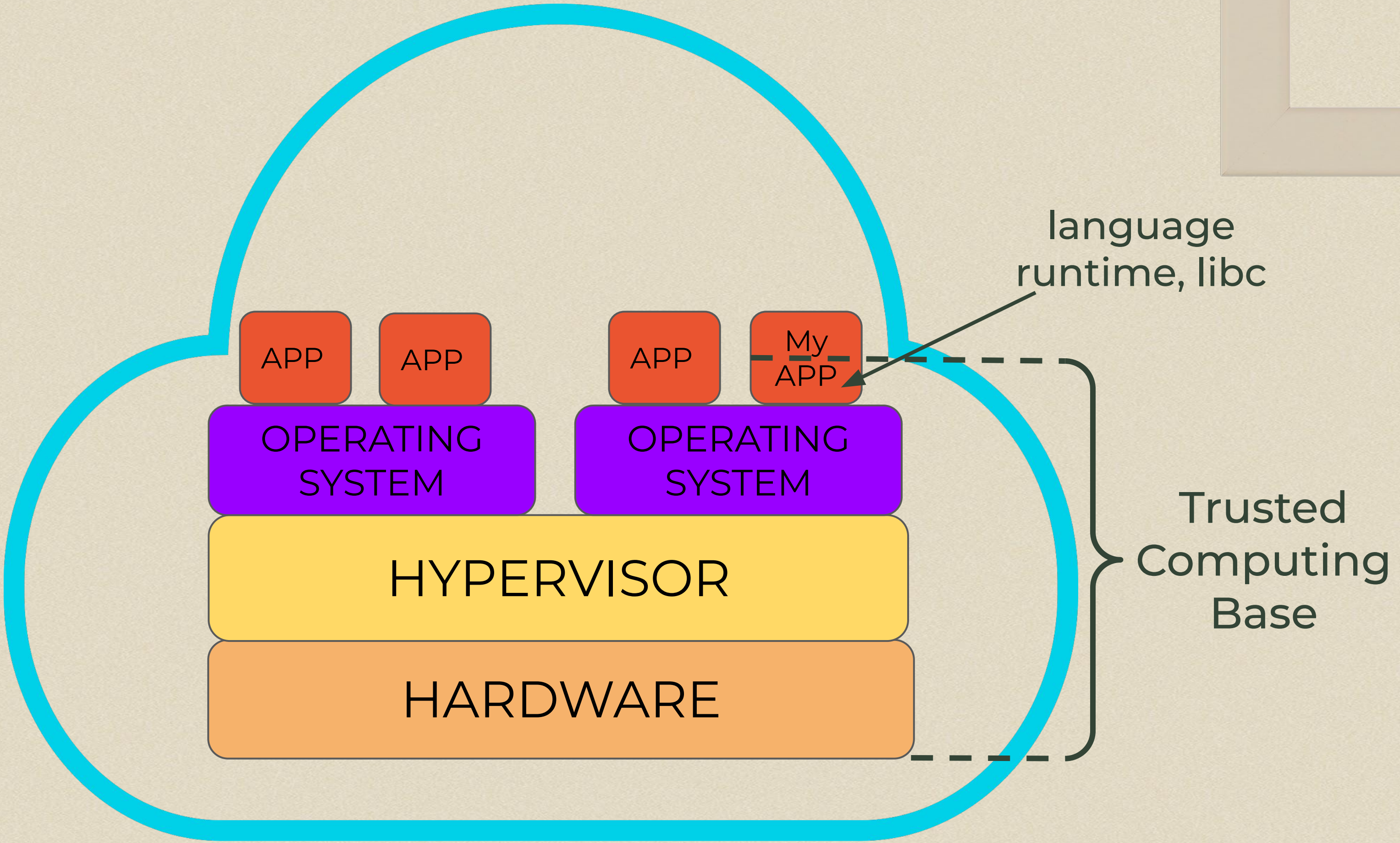
aws

Microsoft Azure



aws

Microsoft Azure


language
runtime, libc
APP
APP
APP
My
APP
OPERATING
SYSTEM
OPERATING
SYSTEM
HYPERVISOR
HARDWARE
Trusted
Computing
Base

Hypervisor/OS Vulnerabilities

{* VIRTUALIZATION *}

Hyper-V bug that could crash 'big portions of Azure cloud infrastructure': Code published

Now patched … dereference …

Tim Anderson Wed …

"Most serious" Linux privilege-escalation bug ever is under active exploit (updated)

Lurking in …

DAN GOODIN - …

VULNERABILITIES

Decade-Old VENOM Bug Exposes Virtualized Environments to Attacks

Security firm Cro…

NEWS

Critical Xen hypervisor flaw endangers virtualized environments

The vulnera…

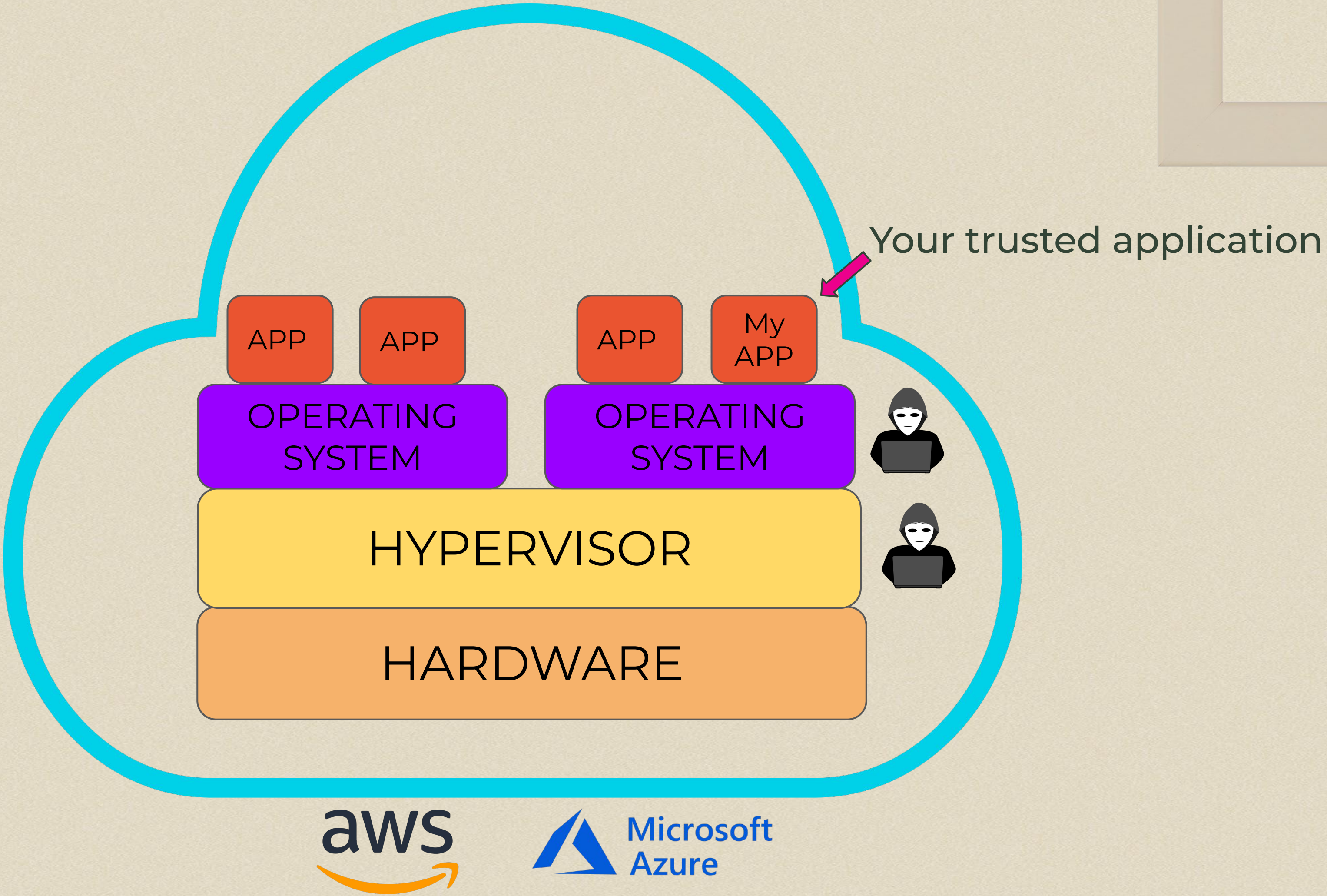
NETWORK SECURITY

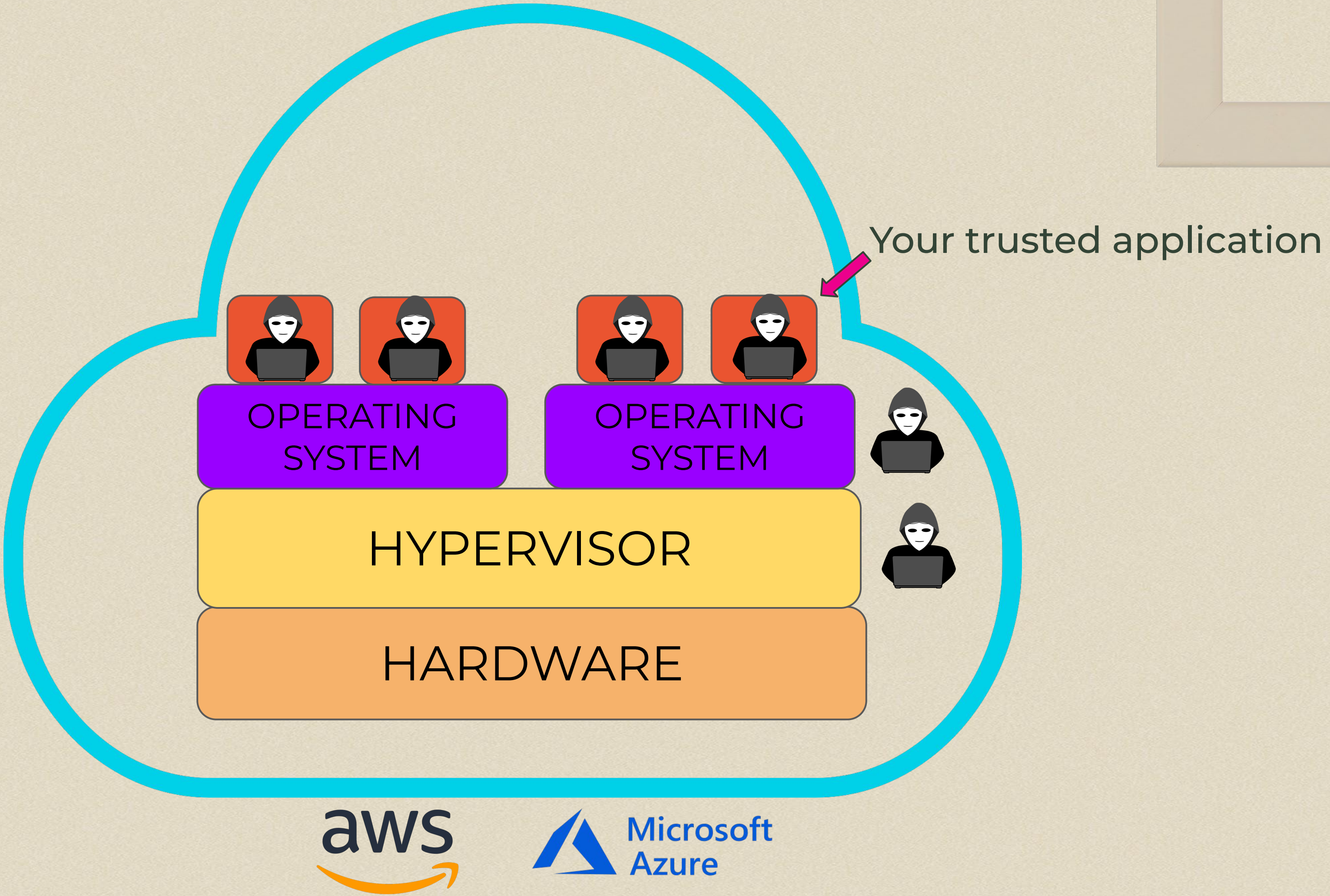
Microsoft Ships Urgent Fixes for Critical Flaws in Windows Kerberos, Hyper-V

Patch T…

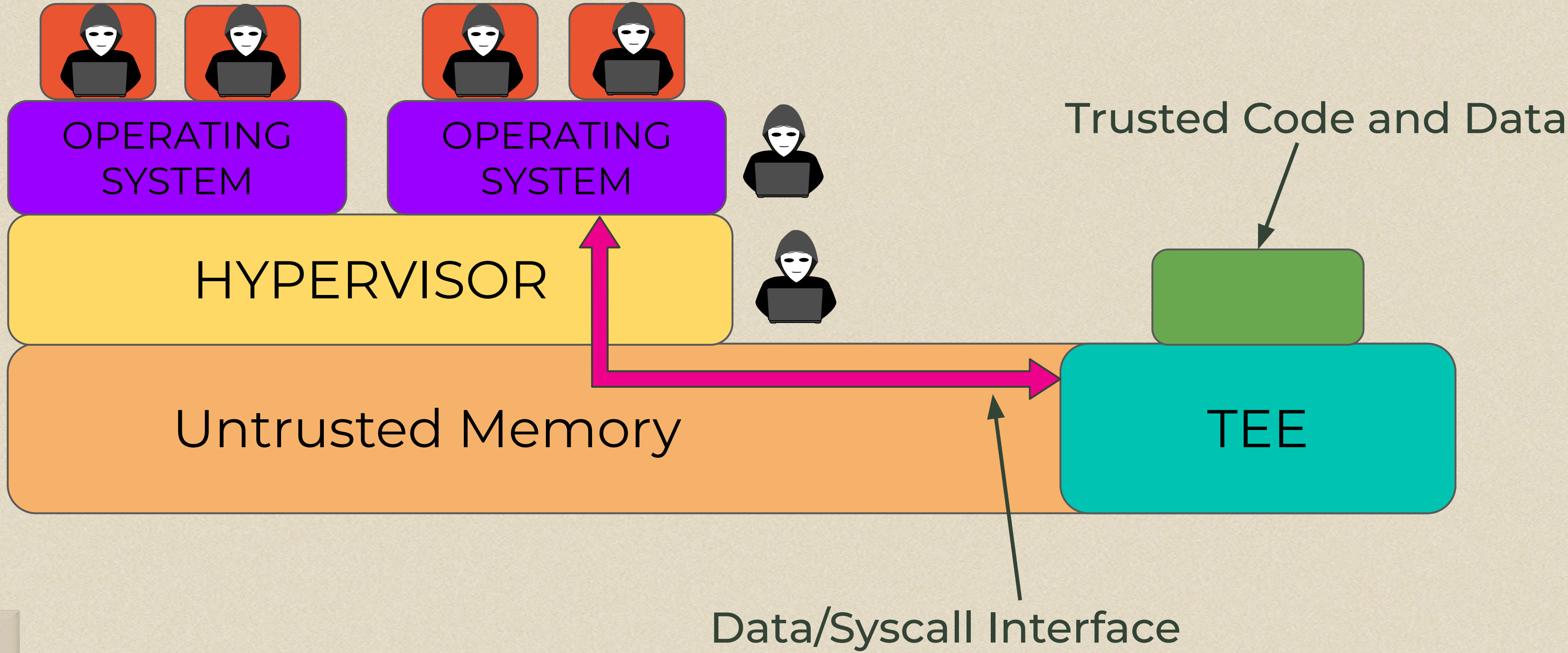
Home > News > Cloud

Hypervisor security flaw could expose AWS, Azure

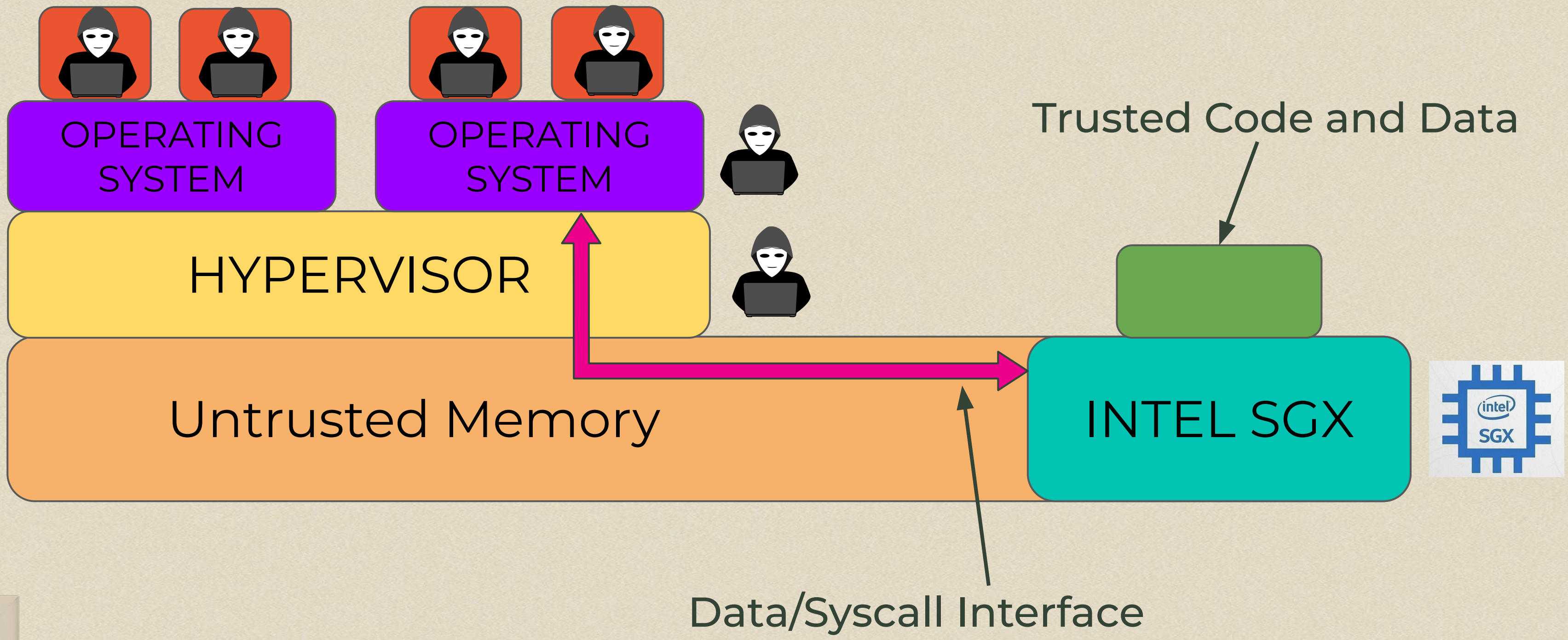

Your trusted application
APP
APP
APP
My APP
OPERATING SYSTEM
OPERATING SYSTEM
HYPERVISOR
HARDWARE
aws
Microsoft Azure


Your trusted application
OPERATING SYSTEM
OPERATING SYSTEM
HYPERVISOR
HARDWARE
aws
Microsoft Azure

Trusted Execution Environments (TEE)



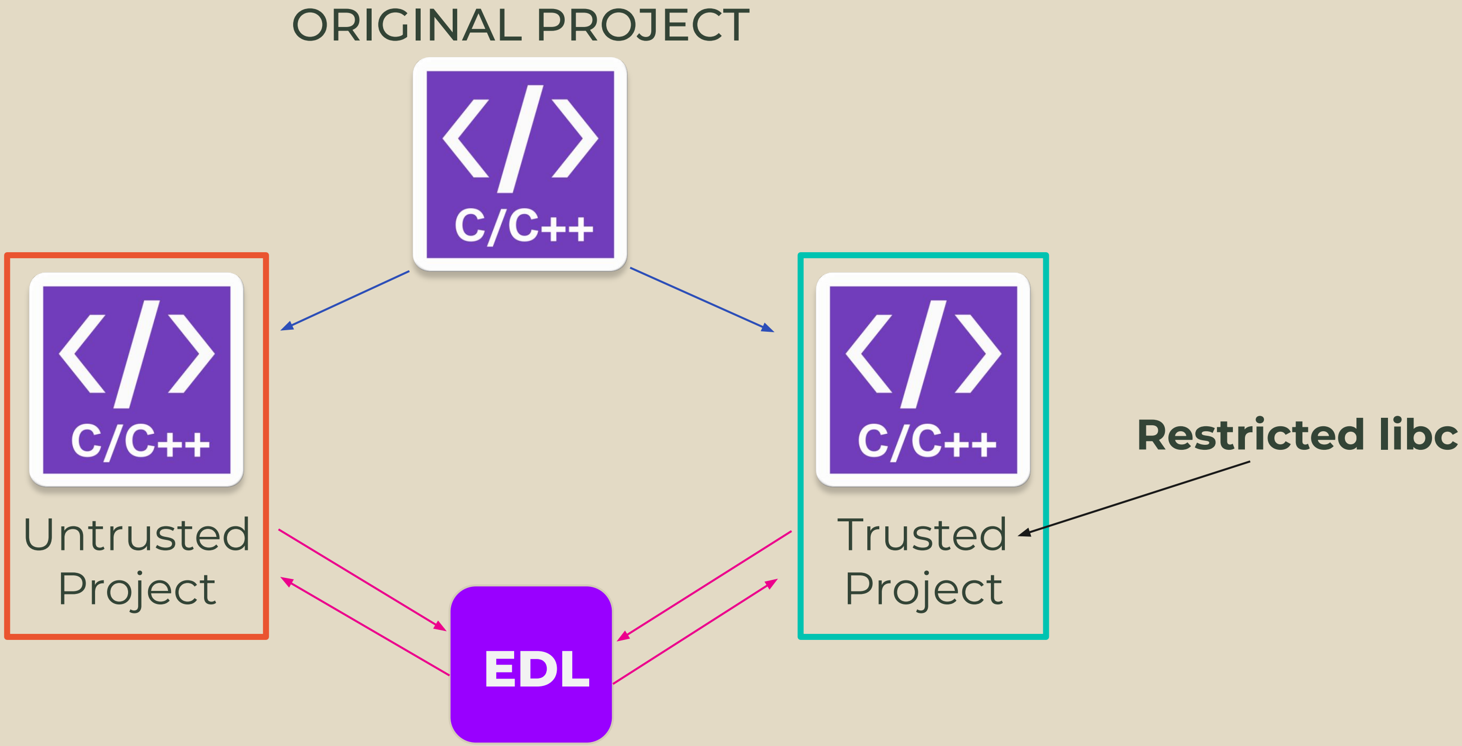
Trusted Execution Environments (TEE)



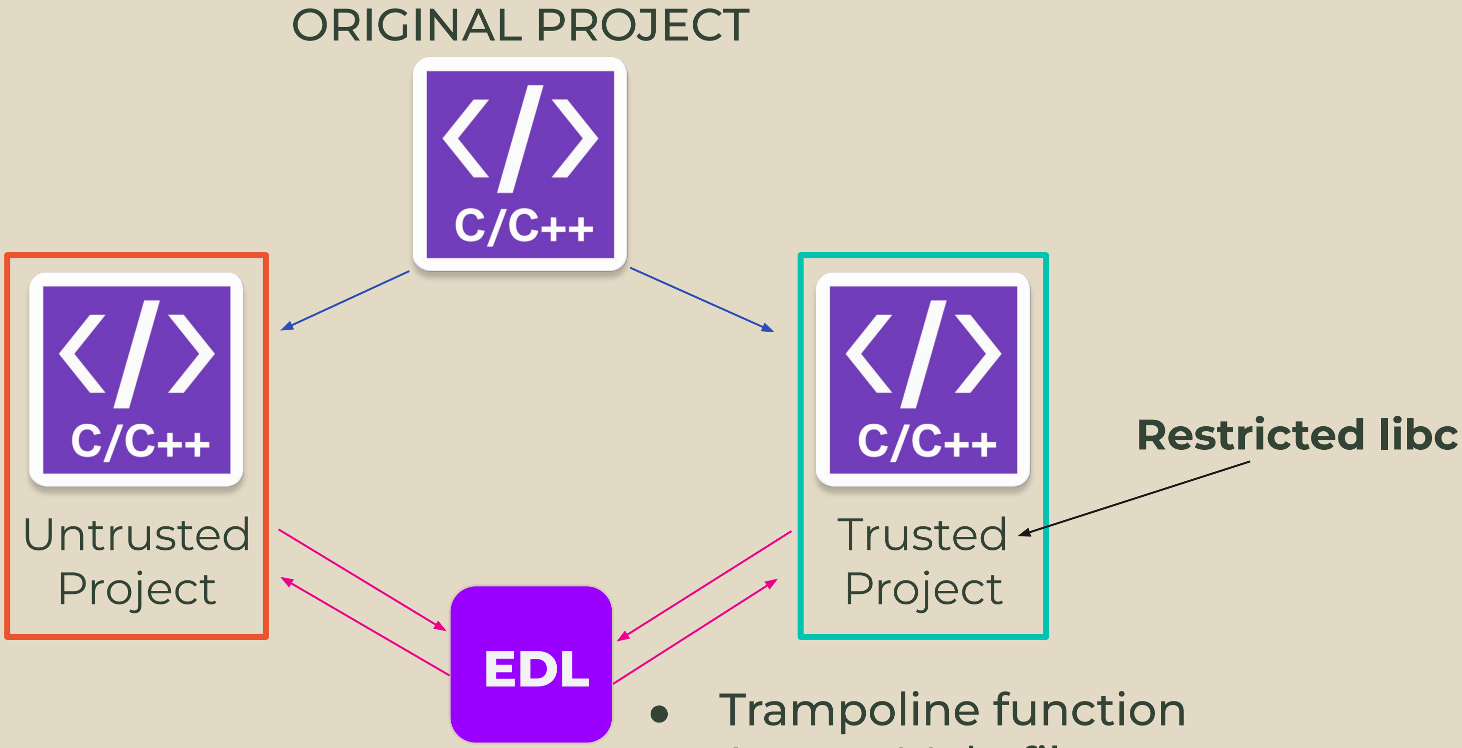
Programming TEEs

ORIGINAL PROJECT

Programming TEEs

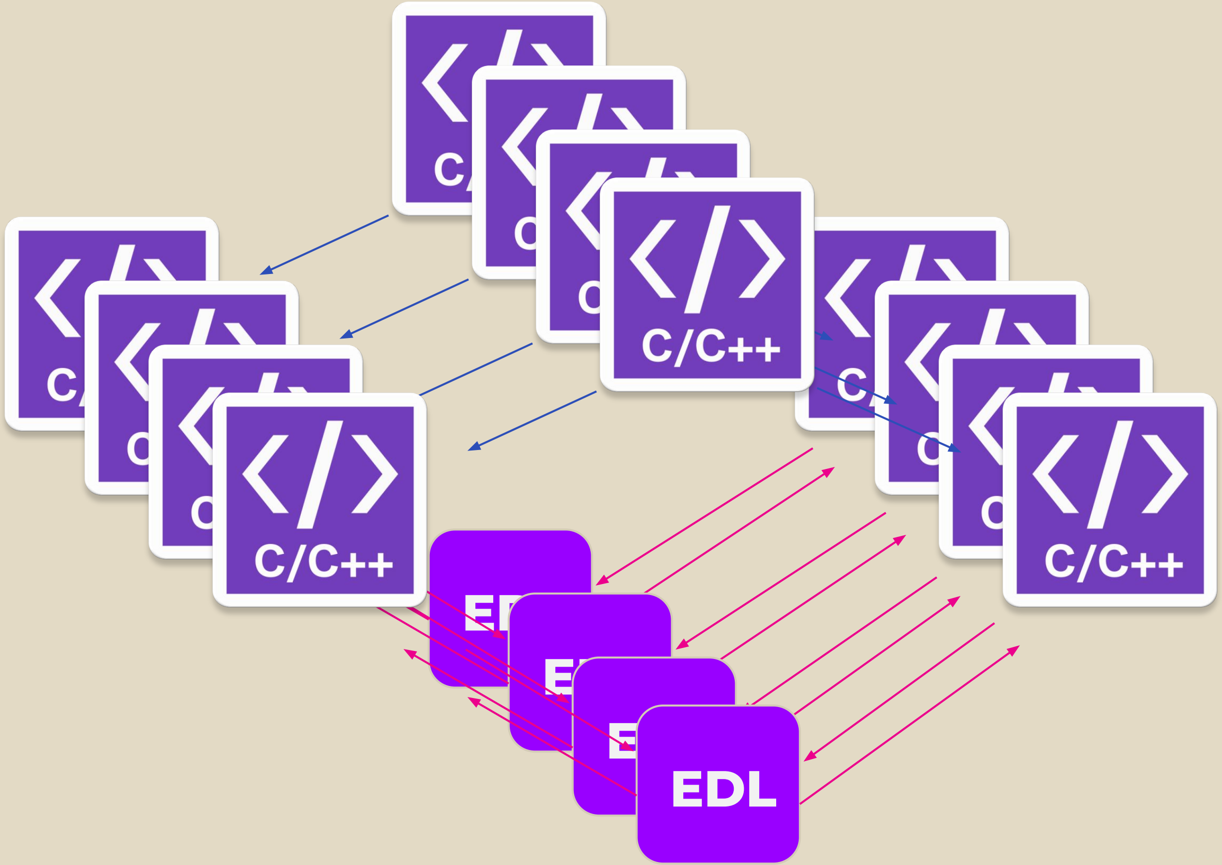


Programming TEEs

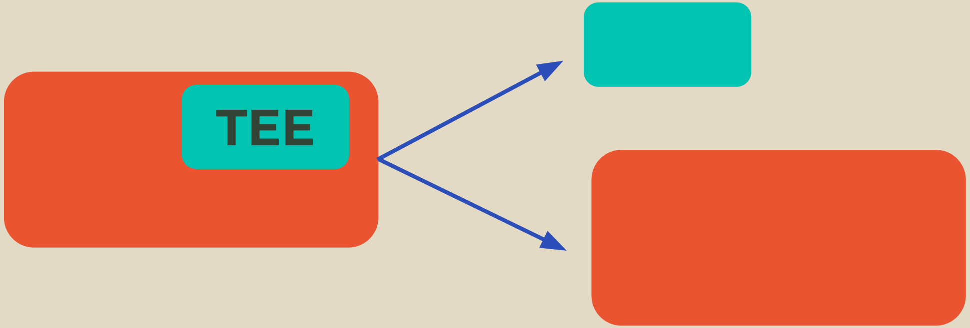


- Trampoline function
- Arcane Makefiles
- Complex data copying protocol
- Limited app porting (libc)

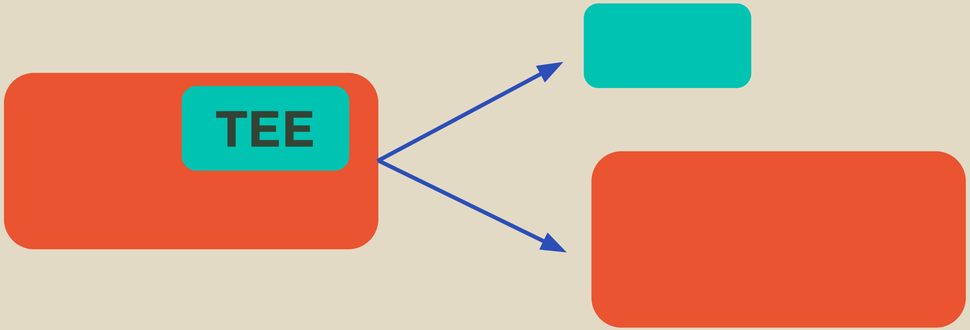
Distributed TEE Applications



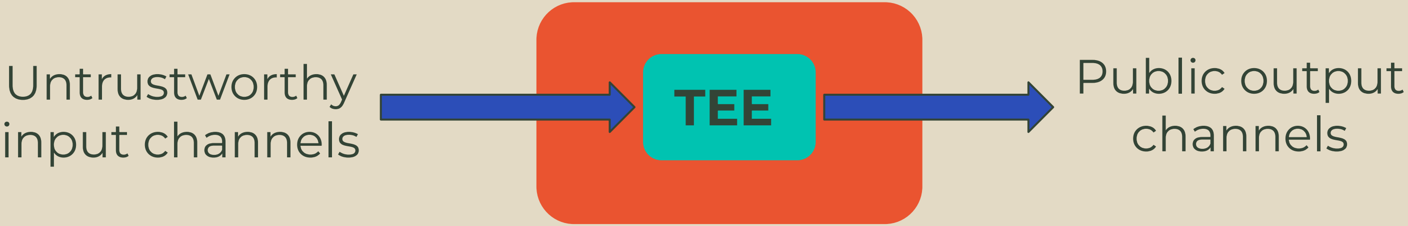
AUTOMATIC PROGRAM PARTITIONING



AUTOMATIC PROGRAM PARTITIONING



INFORMATION FLOW CONTROL



TCS ‘02

Secure Program Partitioning

STEVE ZDANCEWIC, LANTIAN ZHENG, NATHANIEL NYSTROM, and ANDREW C. MYERS
Cornell University

Language Support for Secure Software Development with Enclaves

CSF ‘21

Aditya Oak, *TU Darmstadt*
Amir M. Ahmadian, *KTH Royal Institute of Technology*
Musard Balliu, *KTH Royal Institute of Technology*
Guido Salvaneschi, *University of St.Gallen*

First, **seamless integration** of enclave programming into software applications remains challenging. For example, Intel provides a C/C++ interface to the SGX enclave but no direct support is available for managed languages. As managed languages like Java and Scala are extensively used for developing distributed applications, developers need to either interface their programs with the C++ code executing in the enclave (e.g., using the Java Native Interface [12]) or compile

PtrSplit: Supporting General Pointers in Automatic Program Partitioning

CCS ‘17

Shen Liu
The Pennsylvania State University
University Park, PA
sxl463@cse.psu.edu

Gang Tan
The Pennsylvania State University
University Park, PA
gtan@

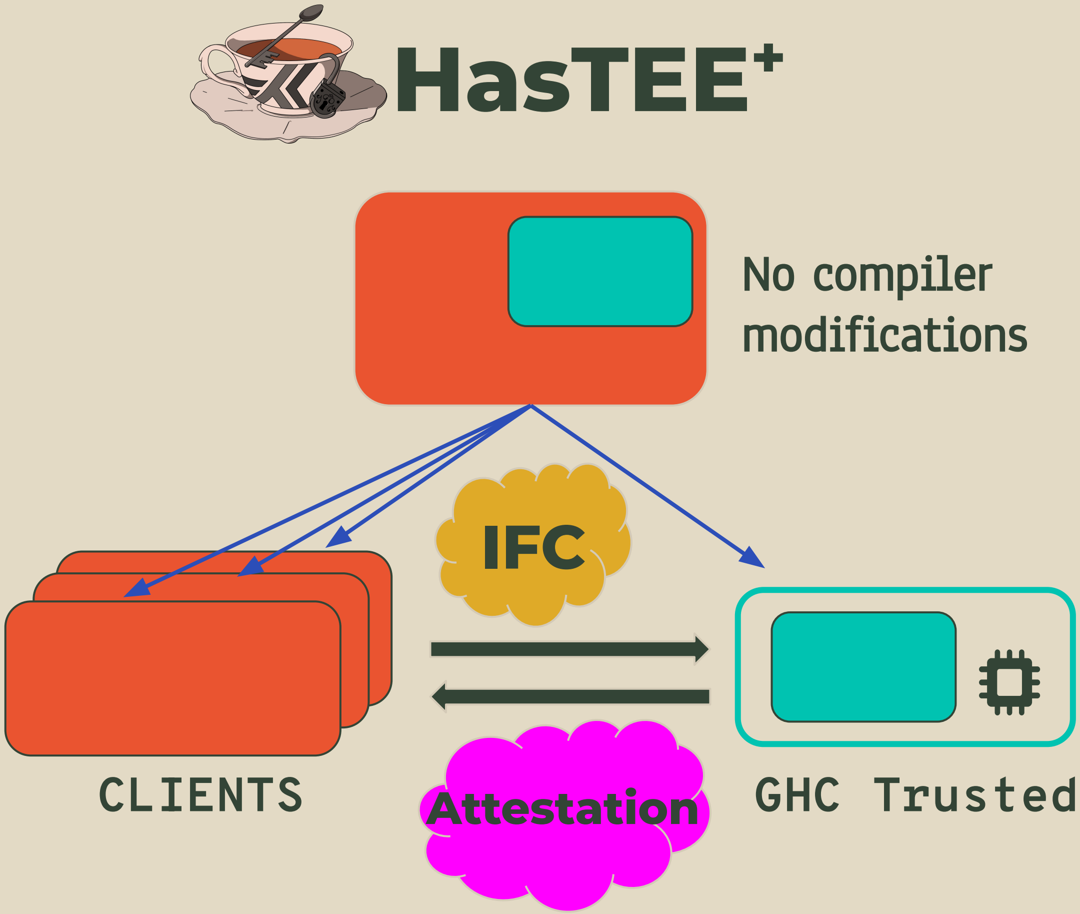
Trent Jaeger
The Pennsylvania State University
University Park, PA

ATC ‘17

Glamdring: Automatic Application Partitioning for Intel SGX

Joshua Lind, *Imperial College London*
Christian Priebe, *Imperial College London*
Divya Muthukumaran, *Imperial College London*
Dan O’Keeffe, *Imperial College London*
Pierre-Louis Aublin, *Imperial College London*
Florian Kelbert, *Imperial College London*
Tobias Reiher, *TU Dresden*
David Goltzsche, *TU Braunschweig*
David Eyers, *University of Otago*
Rüdiger Kapitza, *TU Braunschweig*
Christof Fetzer, *TU Dresden*
Peter Pietzuch, *Imperial College London*

- Significantly changes base language/compiler/runtime
- “Most interesting dynamic properties of programs are undecidable” (Rice’s theorem)
- Either lack information flow control or runtime integration with TEEs or attestation support


HasTEE+
No compiler
modifications
IFC
CLIENTS
Attestation
GHC Trusted

MONAD

```
return :: a -> m a

(>>=) :: m a -> (a -> m b) -> m b
```

MONAD

tainting

return :: a → m a

(>>=) :: m a → (a → m b) → m b

MONAD

computation builder

return :: a → m a

(>>=) :: m a → (a → m b) → m b

TAINT TRACKING

MONAD

return :: a → m a

(>>=) :: m a → (a → m b) → m b

TAINT TRACKING

ALTERNATE SEMANTICS

Illustration : Password Checker

```haskell
pwdChkr :: Enclave String -> String -> Enclave Bool
pwdChkr pwd guess = fmap (== guess) pwd
```

Enclave monad

```haskell
pwdChkr :: Enclave String -> String -> Enclave Bool
pwdChkr pwd guess = fmap (== guess) pwd
```

```haskell
passwordChecker :: App Done
passwordChecker = do
  passwd <- inEnclaveConstant "secret"
  efunc <- inEnclave $ pwdChkr passwd
```

App monad

```haskell
pwdChkr :: Enclave String -> String -> Enclave Bool
pwdChkr pwd guess = fmap (== guess) pwd
```

```haskell
passwordChecker :: App Done
passwordChecker = do
  passwd <- inEnclaveConstant "secret"
  efunc <- inEnclave $ pwdChkr passwd
```

Load code

Load data

```haskell
pwdChkr :: Enclave String -> String -> Enclave Bool
pwdChkr pwd guess = fmap (== guess) pwd

passwordChecker :: App Done
passwordChecker = do
  passwd <- inEnclaveConstant "secret"
  efunc <- inEnclave $ pwdChkr passwd
  runClient $ do -- Client code
    liftIO  $ putStrLn "Enter your password"
    userInput <- liftIO getLine
    res       <- gateway (efunc <@> userInput)
    liftIO $ putStrLn ("Login returned " ++ show res)

main = runApp passwordChecker
```

```haskell
pwdChkr :: Enclave String -> String -> Enclave Bool
pwdChkr pwd guess = fmap (== guess) pwd

passwordChecker :: App Done
passwordChecker = do
  passwd <- inEnclaveConstant "secret"
  efunc <- inEnclave $ pwdChkr passwd
  runClient $ do -- Client code
    liftIO  $ putStrLn "Enter your password"
    userInput <- liftIO getLine
    res       <- gateway (efunc <@> userInput)
    liftIO $ putStrLn ("Login returned " ++ show res)

main = runApp passwordChecker
```

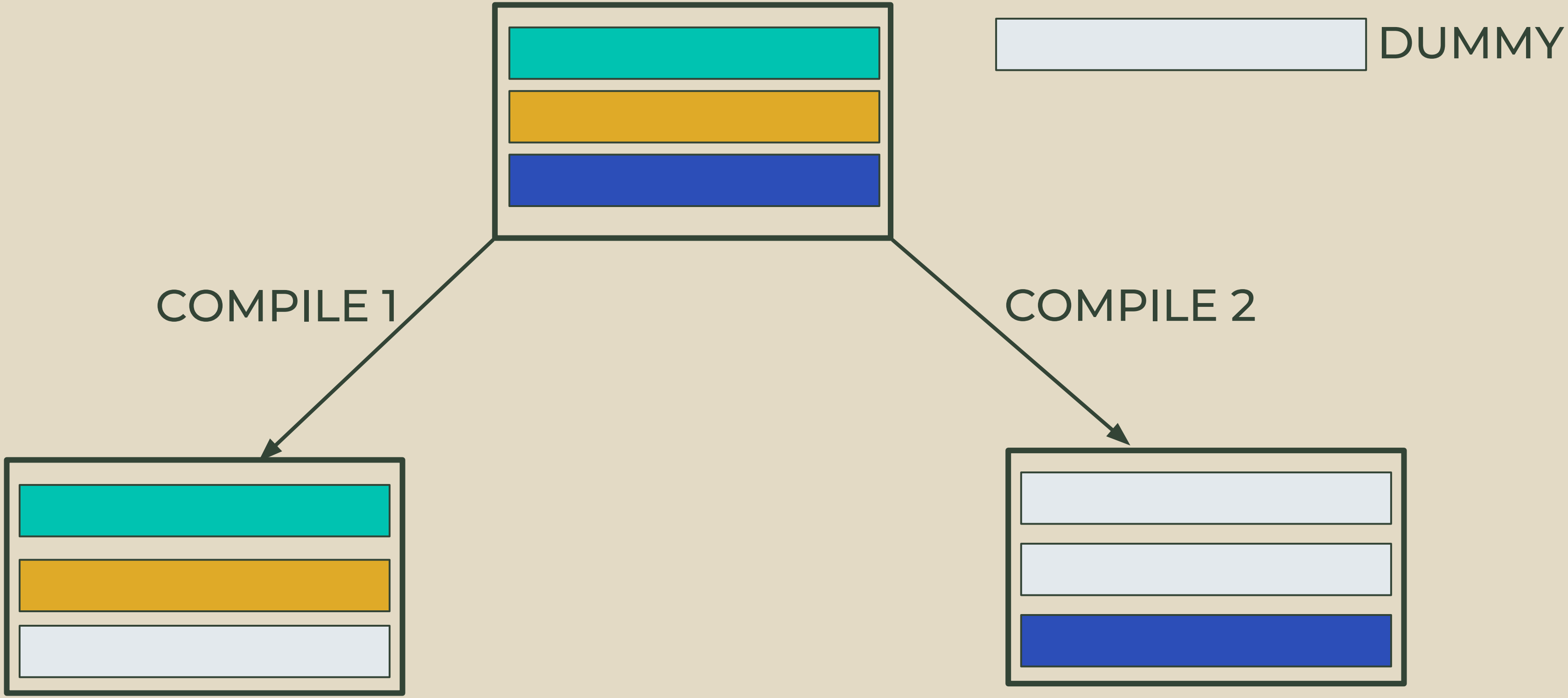
Remote function invocation

Remote function application

```haskell
pwdChkr :: Enclave String -> String -> Enclave Bool
pwdChkr pwd guess = fmap (== guess) pwd

passwordChecker :: App Done
passwordChecker = do
  passwd <- inEnclaveConstant "secret"
  efunc <- inEnclave $ pwdChkr passwd
  runClient $ do -- Client code
    liftIO  $ putStrLn "Enter your password"
    userInput <- liftIO getLine
    res       <- gateway (efunc <@> userInput)
    liftIO $ putStrLn ("Login returned " ++ show res)

main = runApp passwordChecker
```

```haskell
pwdChkr :: Enclave String -> String -> Enclave Bool
pwdChkr pwd guess = fmap (== guess) pwd

passwordChecker :: App Done
passwordChecker = do
  passwd <- inEnclaveConstant "secret"
  efunc <- inEnclave $ pwdChkr passwd
  runClient $ do -- Client code
    liftIO $ putStrLn "Enter your password"
    userInput <- liftIO getLine
    res       <- gateway (efunc <@> userInput)
    liftIO $ putStrLn ("Login returned " ++ show res)

main = runApp passwordChecker
```

COMPILED TWICE



Ekblad A, Claessen K. A seamless, client-centric programming model for type safe web applications. Haskell Symposium 2014.

Compilation 1

Compilation 2

```
-- Enclave
pwdChkr :: Enclave String -> String -> Enclave Bool
pwdChkr pwd guess = fmap (== guess) pwd

passwordChecker :: App Done
passwordChecker = do
  passwd <- inEnclaveConstant "secret"
  efunc <- inEnclave $ pwdChkr passwd
  return DONE

-- wait for calls from Client
main = runApp passwordChecker
```

DUMMY

GHC Trusted

INTEL SGX

Compilation 1

```haskell
-- Enclave
pwdChkr :: Enclave String -> String -> Enclave Bool
pwdChkr pwd guess = fmap (== guess) pwd

passwordChecker :: App Done
passwordChecker = do
  passwd <- inEnclaveConstant "secret"
  efunc <- inEnclave $ pwdChkr passwd
  return DONE

-- wait for calls from Client
main = runApp passwordChecker
```

GHC Trusted

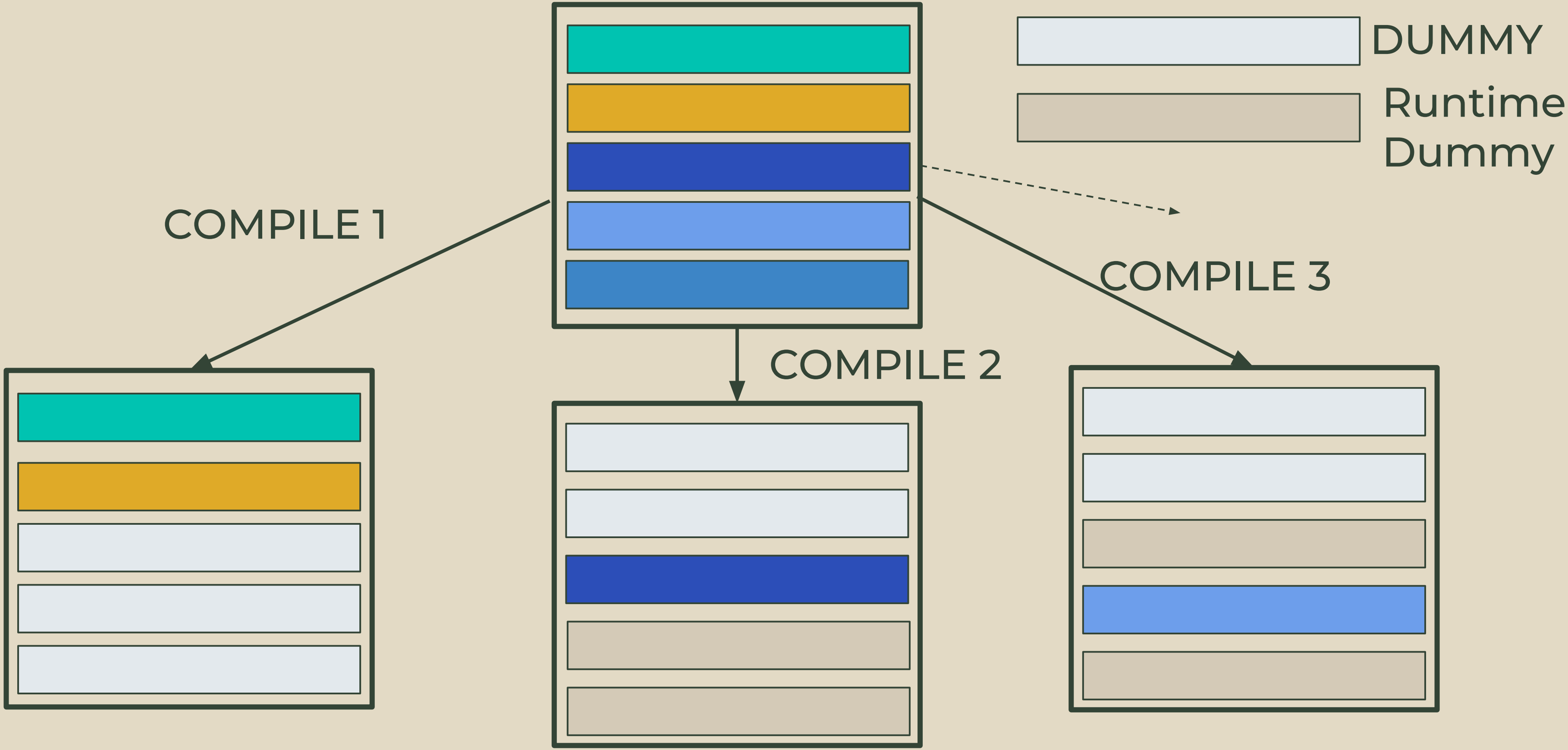
INTEL SGX

Compilation 2

```haskell
-- Client
pwdChkr = -- gets optimised away

passwordChecker :: App Done
passwordChecker = do
  passwd <- return Dummy
  efunc <- inEnclave $ -- ignores pwdChkr body
  runClient $ do -- Client code
    liftIO  $ putStrLn "Enter your password"
    userInput <- liftIO getLine
    res       <- gateway (efunc <@> userInput)
    liftIO $ putStrLn ("Login returned " ++ show res)

-- drives the application
main = runApp passwordChecker
```

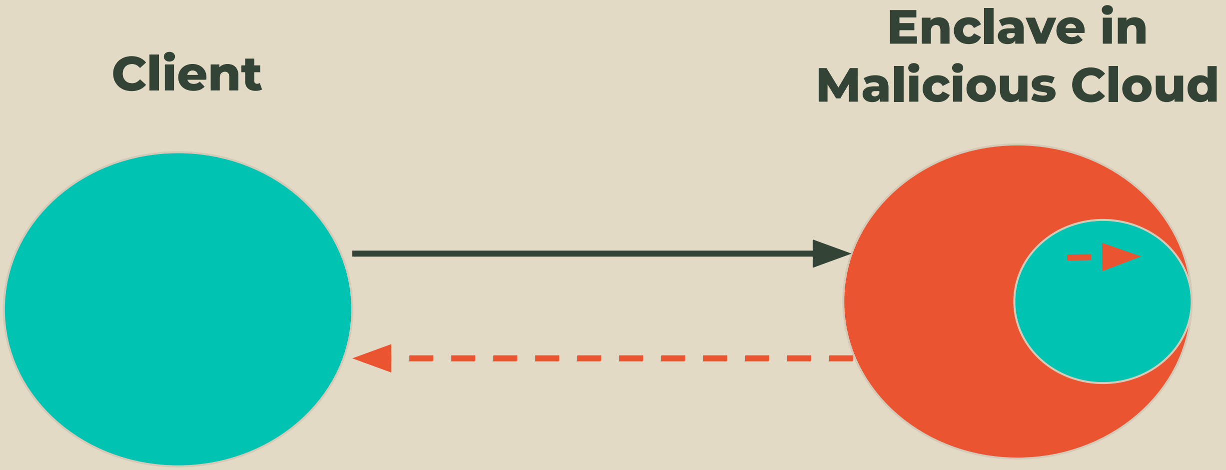
DUMMY

Generalisation for multiple clients

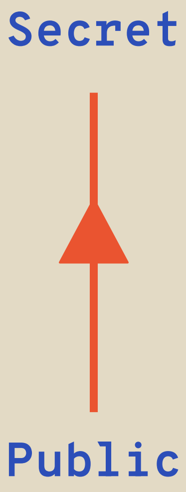


Shen G, Kashiwa S, Kuper L. Haschor: Functional Choreographic Programming for All. ICFP 2023

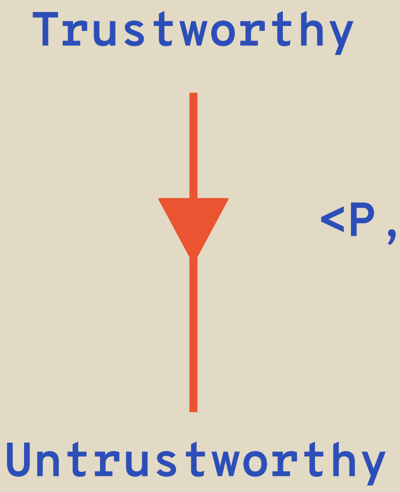
Information Flow



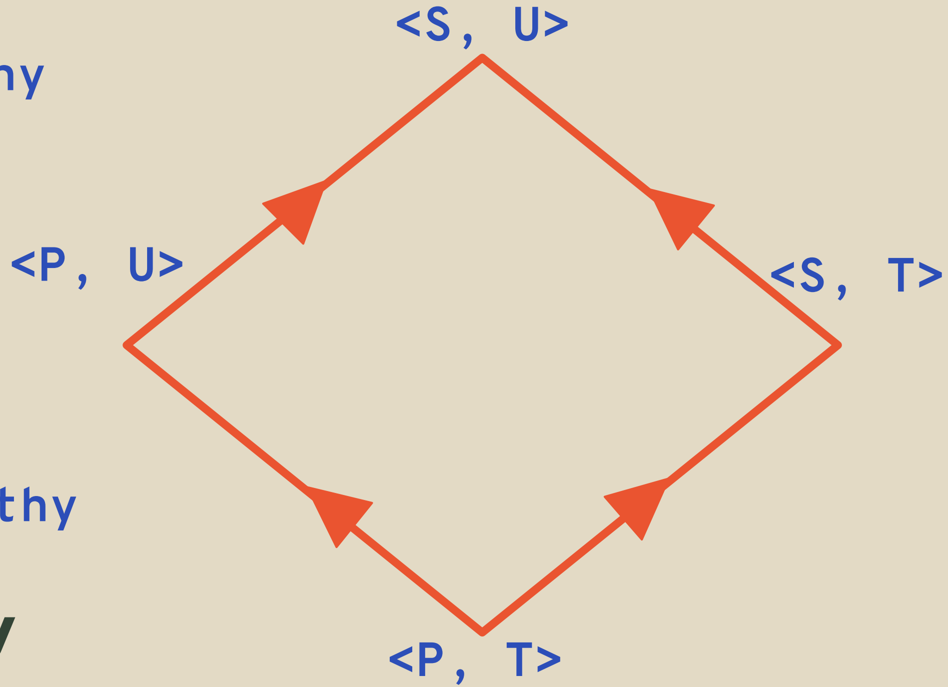
Information Flow Control



Confidentiality

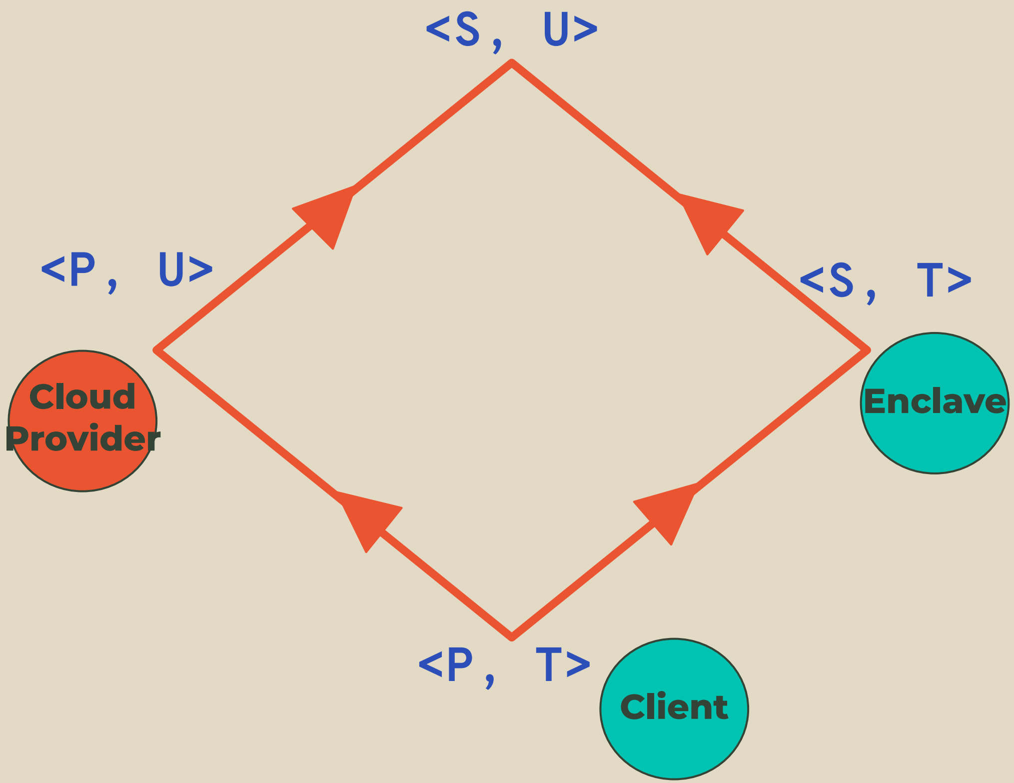


Integrity

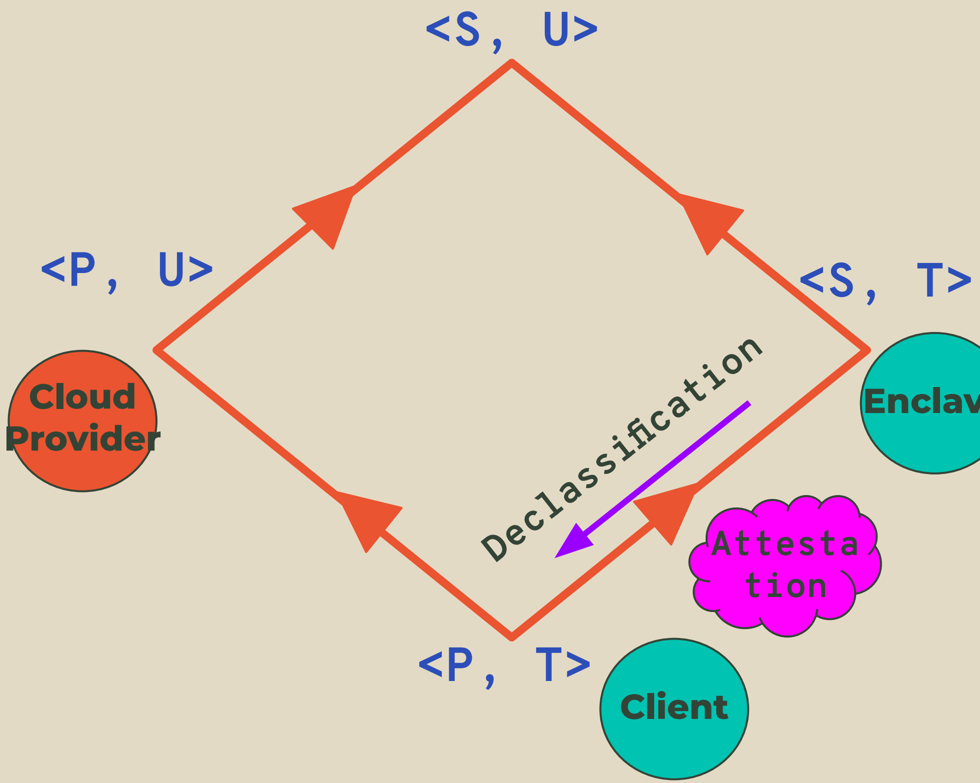


1. Denning, Dorothy E. "A lattice model of secure information flow." *Communications of the ACM* 19.5 (1976).
2. Biba, K.J. Integrity considerations for secure computer systems. Technical Report. April 1977.

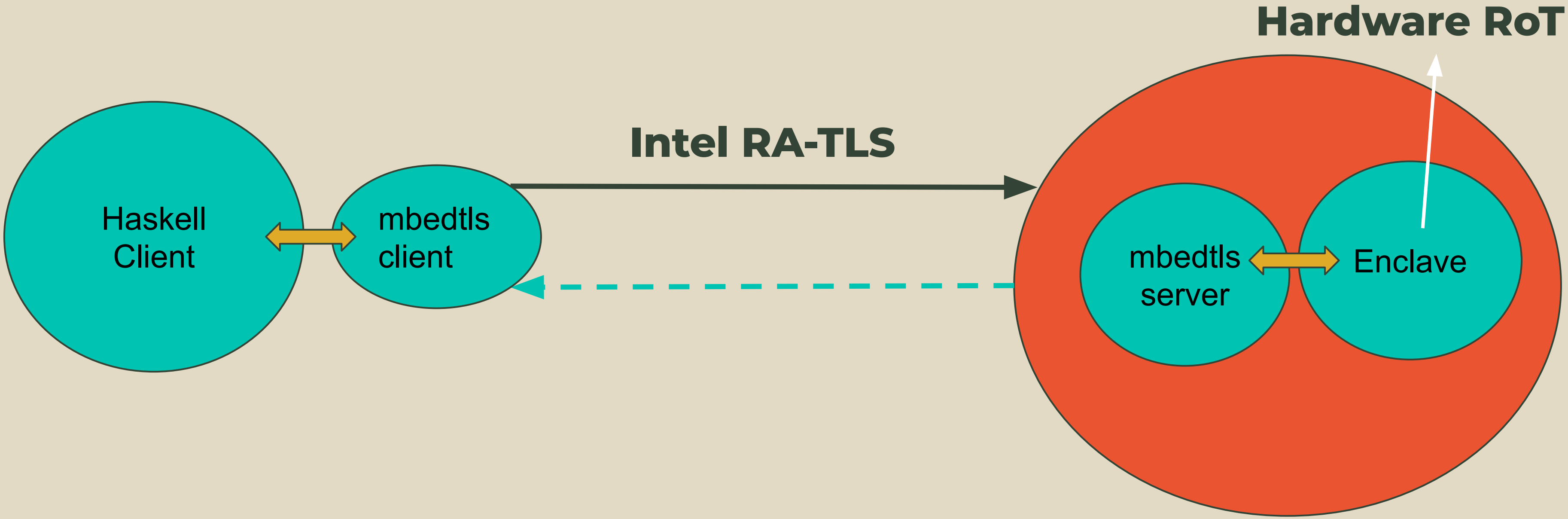
Information Flow Control



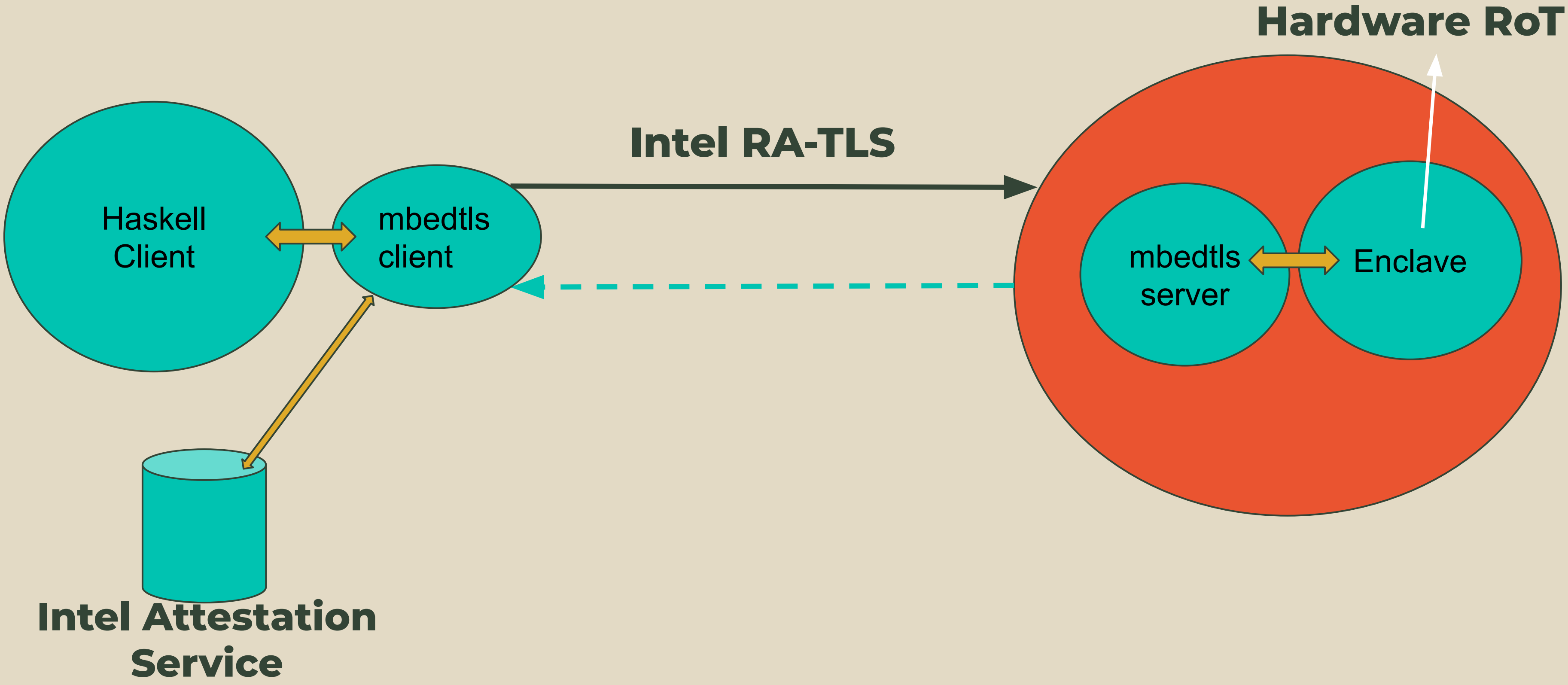
Information Flow Control



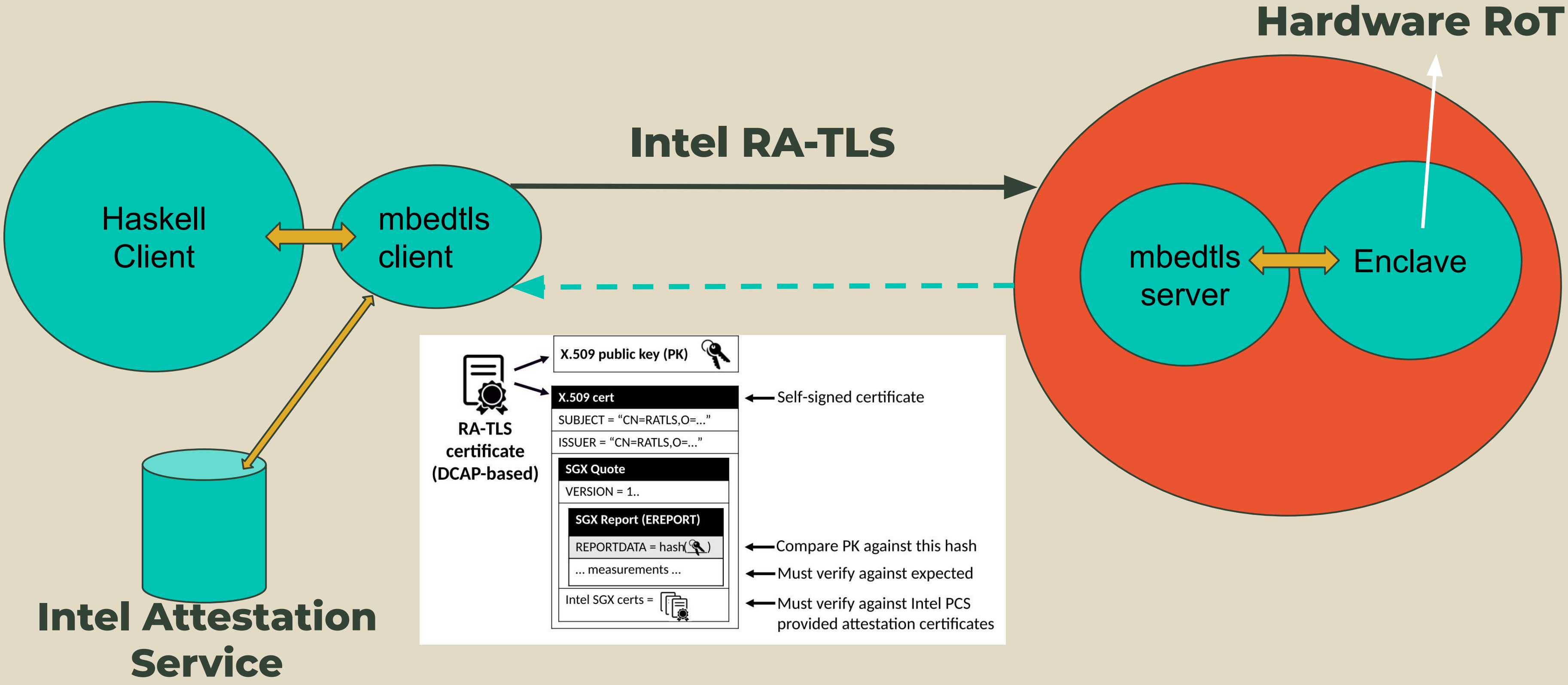
Attestation



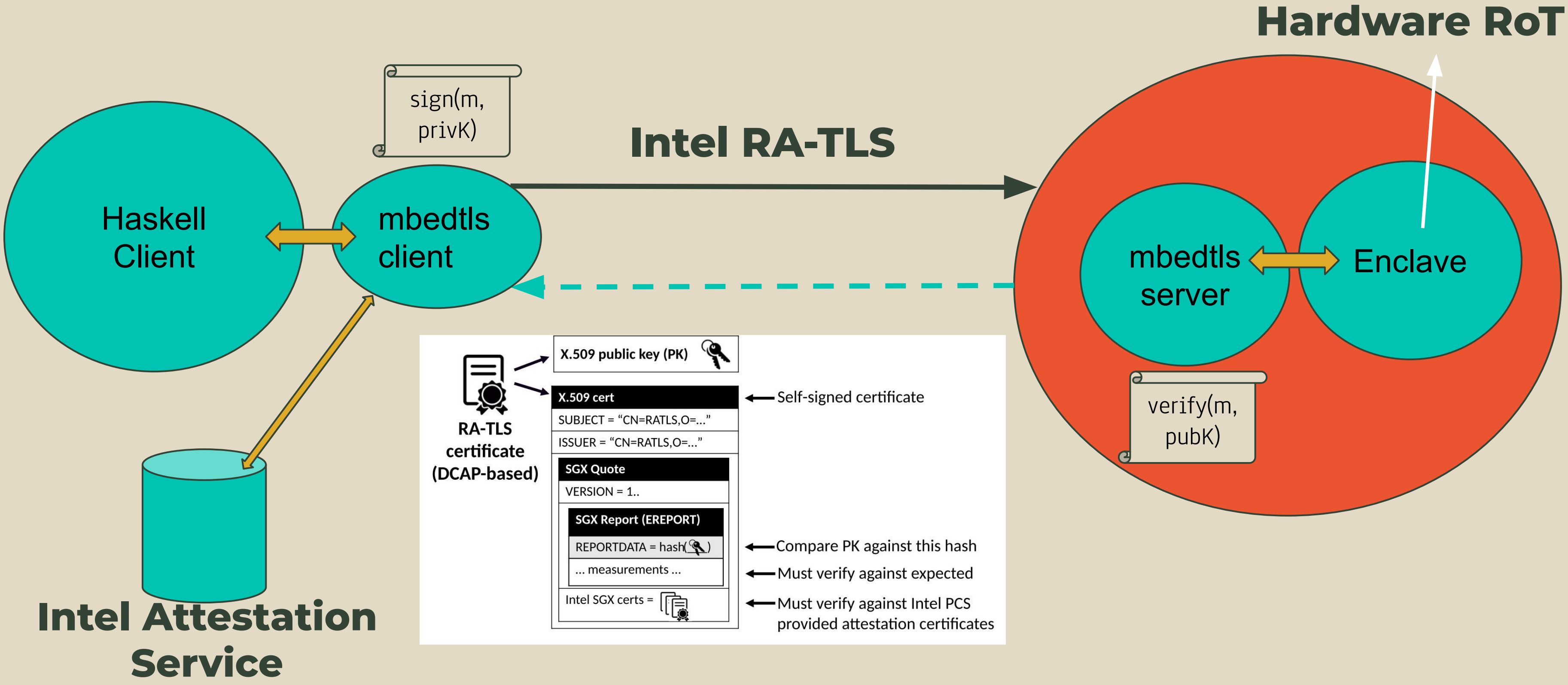
Attestation



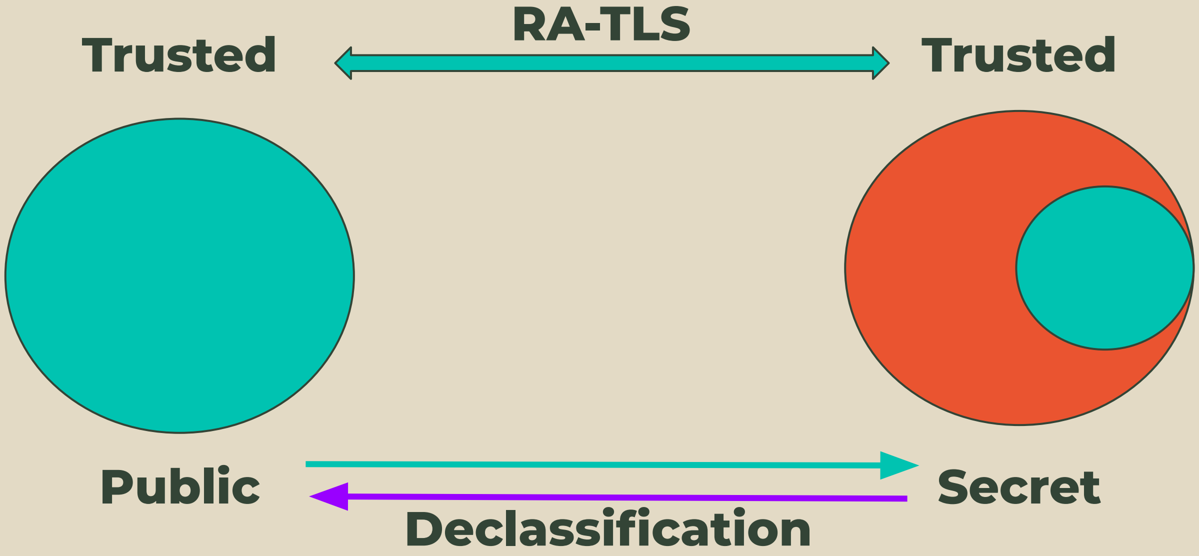
Attestation



Attestation



Information Flow



Declassification

Enclave l p a

Labeled l a

1. Stefan D, Russo A, Mitchell JC, Mazières D. Flexible Dynamic Information flow control in Haskell. Haskell Symposium 2011.
2. Stefan, D., Russo, A., Mazières, D., & Mitchell, J. C. (2012). Disjunction Category Labels. *NordSec 2011*

Declassification

Disjunction Category Labels

Enclave l p a

Privileges or capabilities

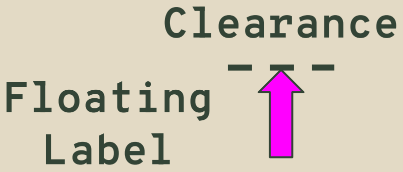
Labeled l a

1. Stefan D, Russo A, Mitchell JC, Mazières D. Flexible Dynamic Information flow control in Haskell. Haskell Symposium 2011.
2. Stefan, D., Russo, A., Mazières, D., & Mitchell, J. C. (2012). Disjunction Category Labels. *NordSec 2011*

Declassification

Enclave l p a

Labeled l a



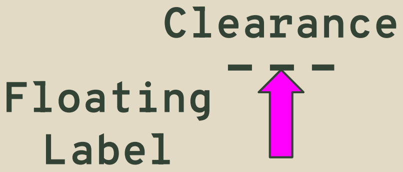
$L \sqcup L_{cur}$ $\Gamma \vdash \dots$

1. Stefan D, Russo A, Mitchell JC, Mazières D. Flexible Dynamic Information flow control in Haskell. Haskell Symposium 2011.
2. Stefan, D., Russo, A., Mazières, D., & Mitchell, J. C. (2012). Disjunction Category Labels. *NordSec 2011*

Declassification

Enclave l p a

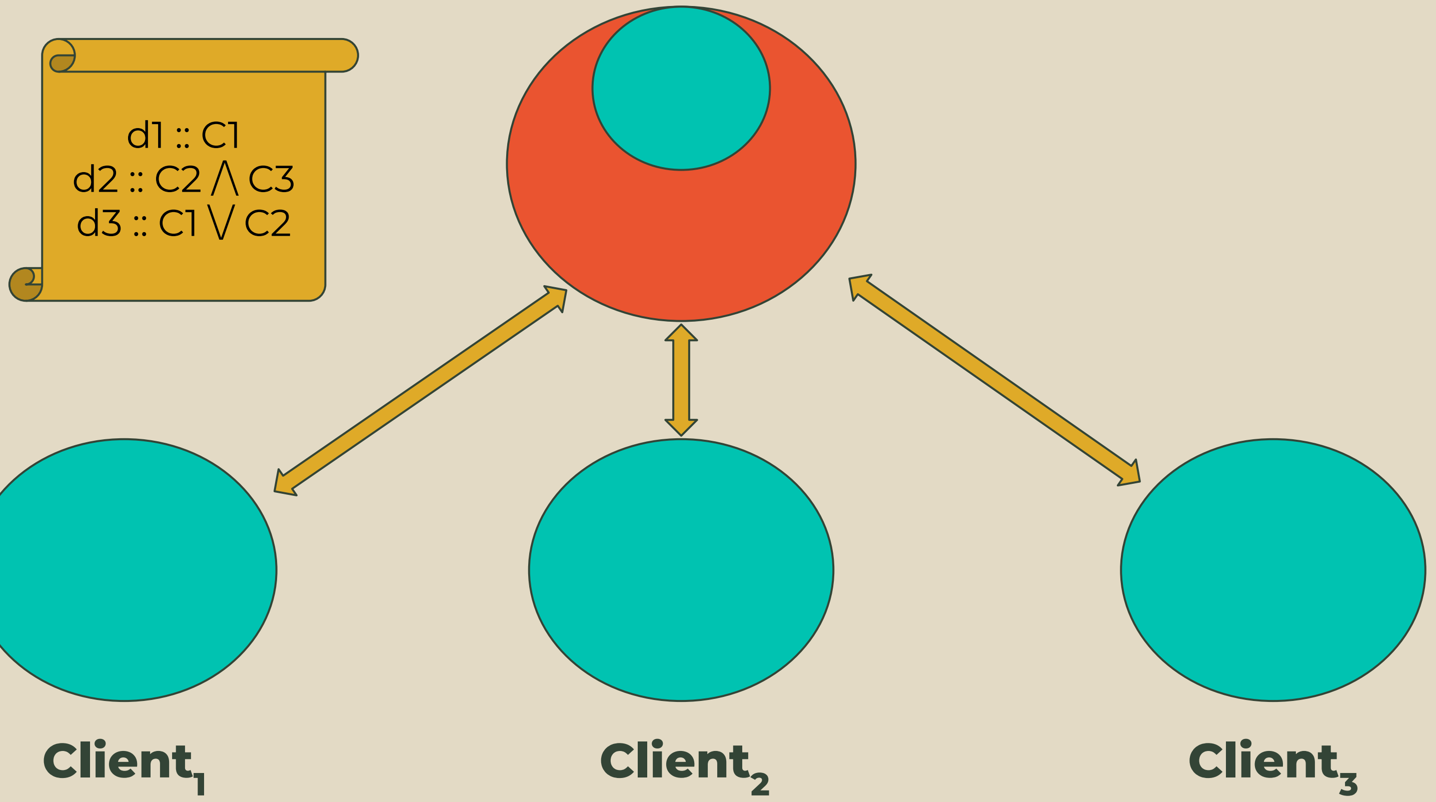
Labeled l a



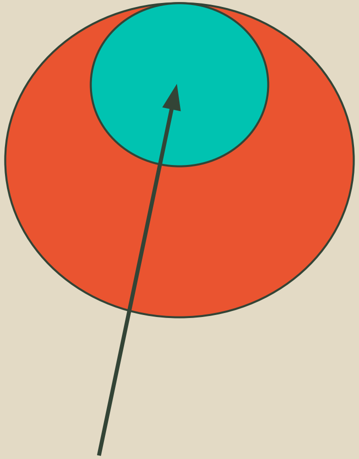
$L \sqcup L_{cur} \downarrow \quad \Gamma \vdash \ldots$

1. Stefan D, Russo A, Mitchell JC, Mazières D. Flexible Dynamic Information flow control in Haskell. Haskell Symposium 2011.
2. Stefan, D., Russo, A., Mazières, D., & Mitchell, J. C. (2012). Disjunction Category Labels. *NordSec 2011*

Data Clean Room



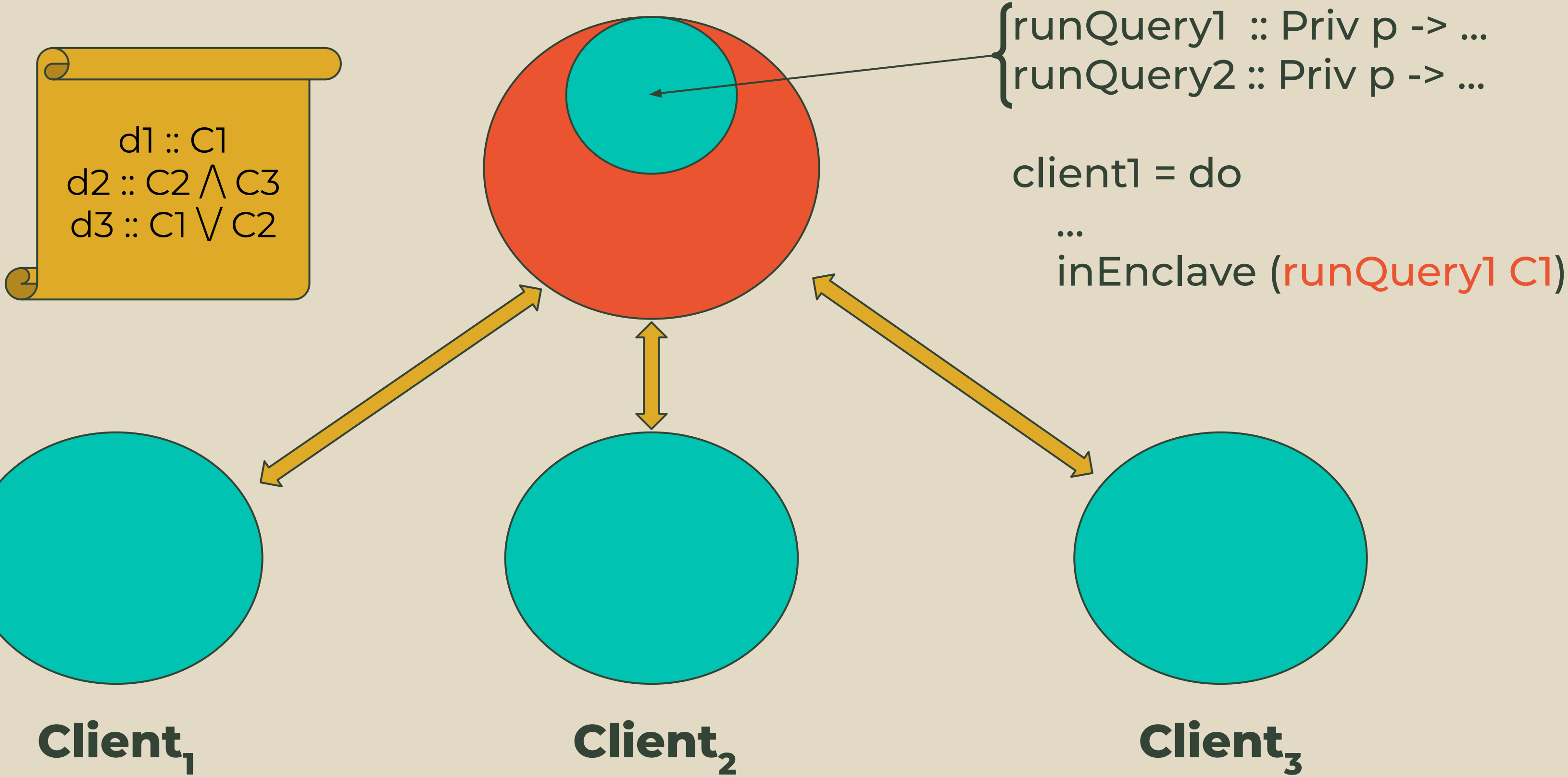
Data Clean Room



$f1_{C1}$ :: ... Carries privilege to declassify C1

$f2_{C2}$ :: ...

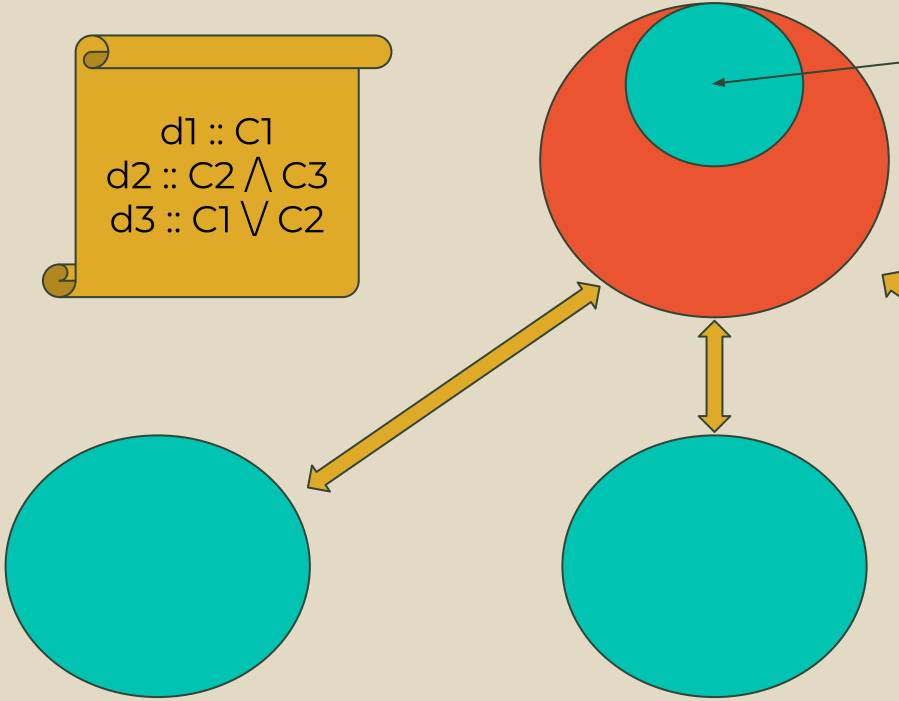
Data Clean Room



Data Clean Room

Linear arrow for “unforgeability”

```
runQuery1  :: Priv p -o ...
runQuery2 :: Priv p -o ...

client1 = do
   ...
   inEnclave (runQuery1 C1)
```

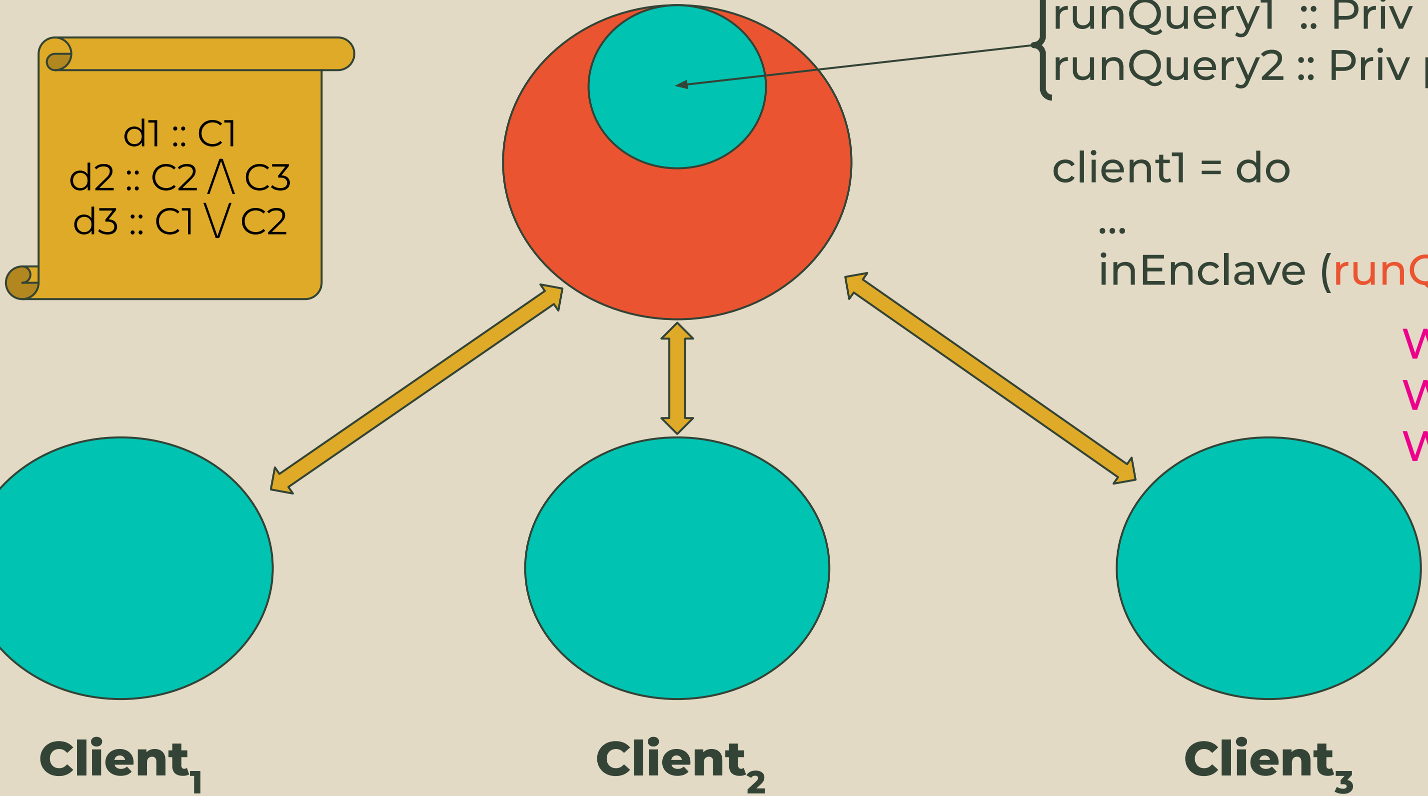


Client$_1$ **Client$_2$** **Client$_3$**

Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. Bernardy et al. POPL 2017

Data Clean Room

Linear arrow for "unforgeability"

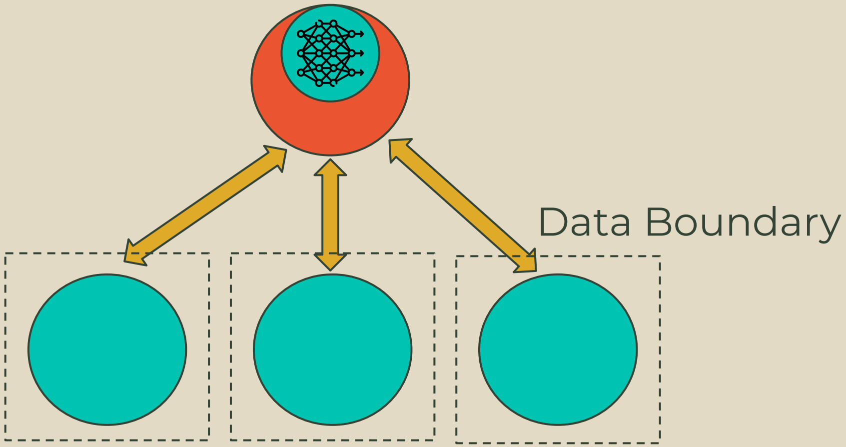



```
runQuery1  :: Priv p -o ...
runQuery2 :: Priv p -o ...

client1 = do
   ...
   inEnclave (runQuery1 C1)
```

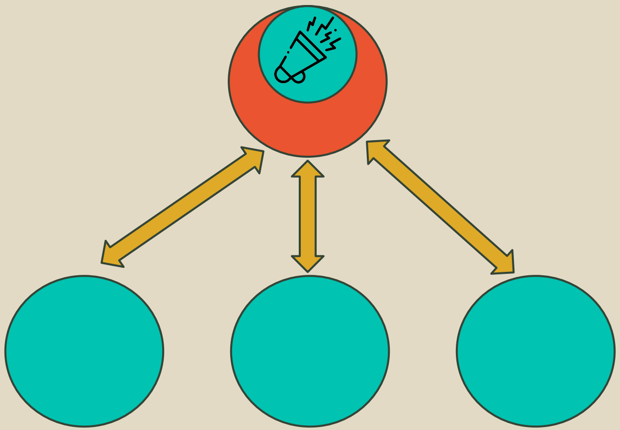
Who?
What?
Where?

Declassification: Dimensions and Principles. Sabelfeld and Sands. Journal of Computer Security. 2009.

More case studies...

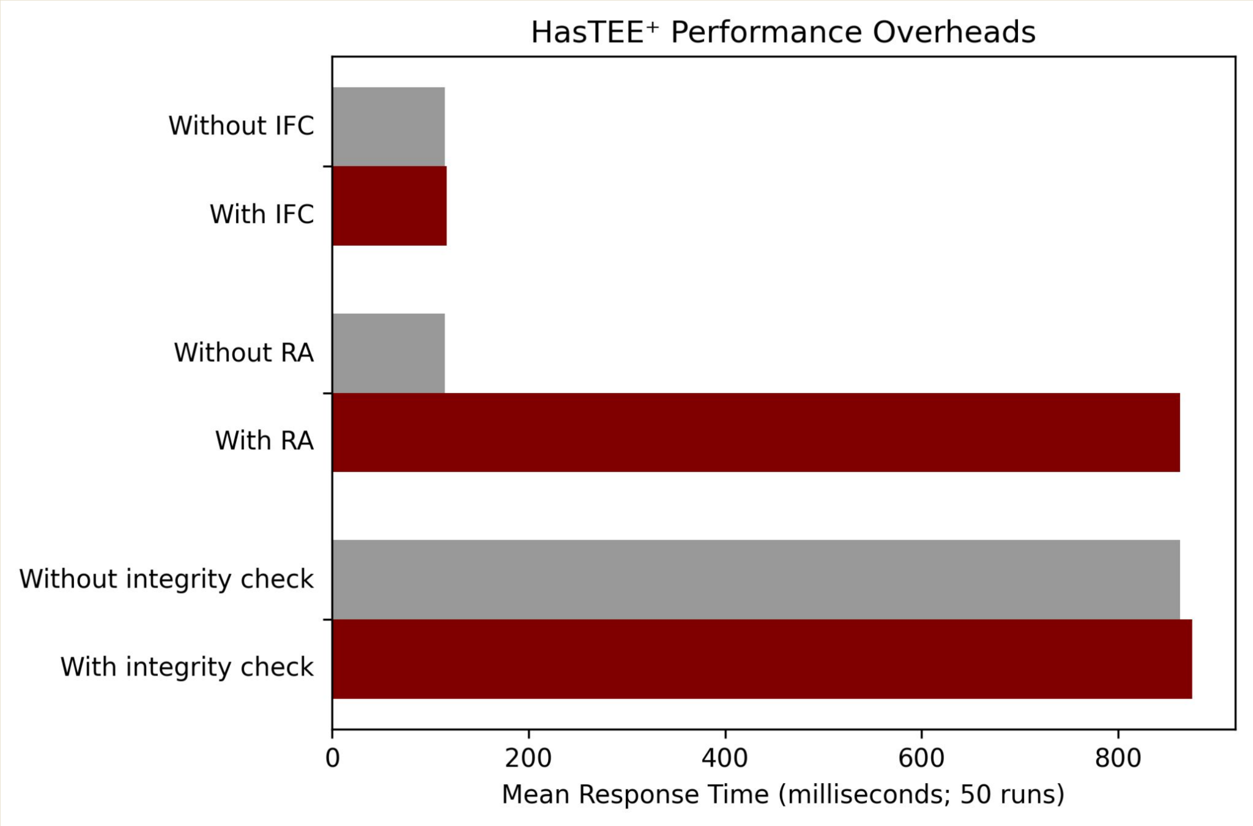


Federated Learning with TEEs
and homomorphic encryption

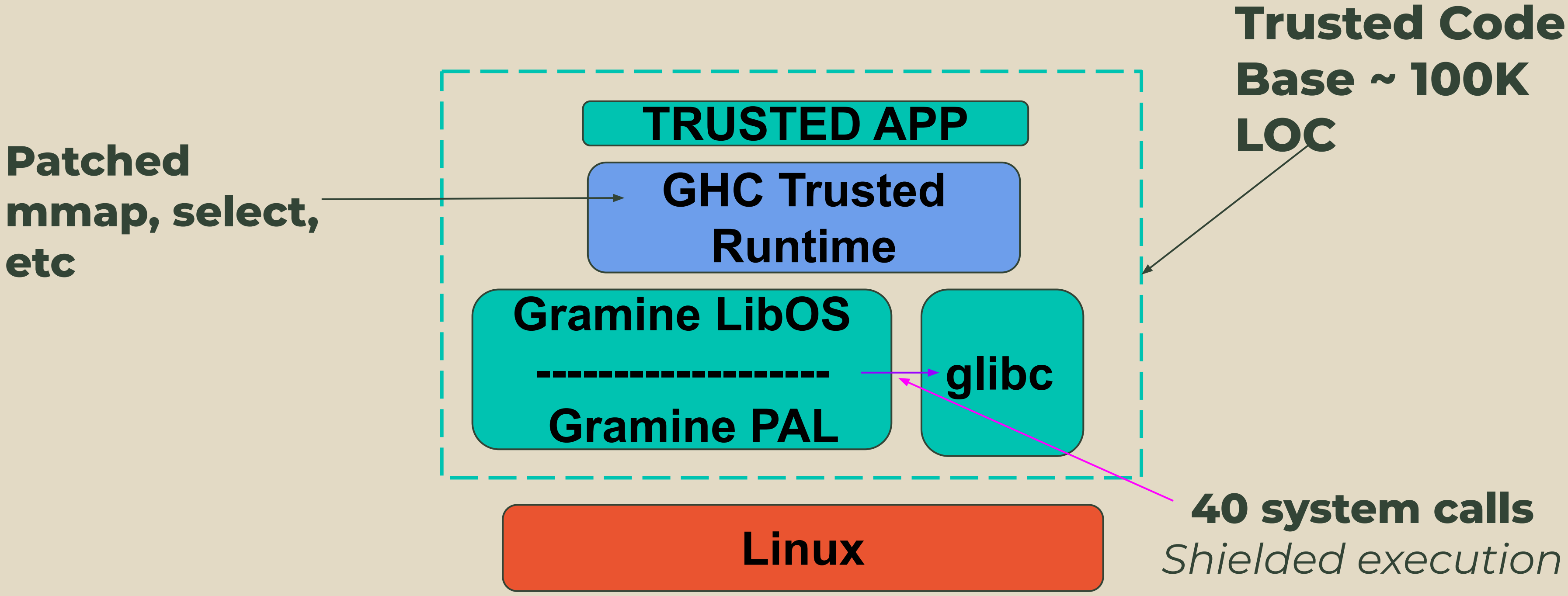


Data Clean Room with
differential privacy

Performance Overheads



Trusted GHC



Tsai CC, Porter DE, Vij M. Graphene-SGX: A practical library {OS} for unmodified applications on {SGX}. Usenix ATC 2017.

Trusted GHC

Memory	RSS	Virtual Size	Disk Swap
At rest	19,132 KB	287,920 KB	0 KB
Peak	20,796 KB	290,032KB	0 KB

LATENCY ~ 60 ms
vs
0.6 ms in native SDK

Part II

SynchronVM

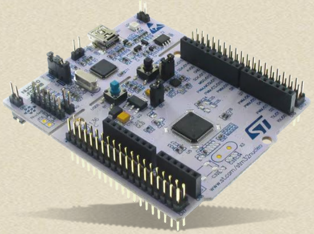
ATTACKER MODELS

Attacker Model 1

TRUST

in the OS and other low-level software

Attacker Model 2



MEMORY UNSAFETY

to accommodate resource constraints

Synchron - An API and Runtime for Embedded Systems

Abhiroop Sarkar
Chalmers University, Sweden
Bo Joel Svensson
Chalmers University, Sweden
Mary Sheeran
Chalmers University, Sweden

Abstract

Programming embedded systems applications involves writing concurrent, event-driven and timing-aware programs. Traditionally, such programs are written in low-level machine-oriented programming languages like C or Assembly. We present an alternative by introducing Synchron, an API that offers high-level abstractions to the programmer while supporting the low-level infrastructure in an associated runtime system and one-time-effort drivers.

Embedded systems applications exhibit the general characteristics of being (i) concurrent, (ii) I/O–bound and (iii) timing-aware. To address each of these concerns, the Synchron API consists of three components - (1) a Concurrent ML (CML) inspired message-passing concurrency model, (2) a message-passing–based I/O interface that translates between low-level interrupt based and memory-mapped peripherals, and (3) a timing operator, `syncT`, that marries CML's `sync` operator with timing windows inspired from the TinyTimber kernel.

We implement the Synchron API as the bytecode instructions of a virtual machine called SynchronVM. SynchronVM hosts a Caml-inspired functional language as its frontend language, and the backend of the VM supports the STM32F4 and NRF52 microcontrollers, with RAM in the order of hundreds of kilobytes. We illustrate the expressiveness of the Synchron API by showing examples of expressing state machines commonly found in embedded systems. The timing functionality is demonstrated through a music programming exercise. Finally, we provide benchmarks on the response time, jitter rates, memory, and power usage of the SynchronVM.

2012 ACM Subject Classification Computer systems organization → Embedded software; Software and its engineering → Runtime environments; Computer systems organization → Real-time languages; Software and its engineering → Concurrent programming languages

ECOOP 2022

Hailstorm : A Statically-Typed, Purely Functional Language for IoT Applications

Abhiroop Sarkar
sarkara@chalmers.se
Chalmers University
Gothenburg, Sweden

Mary Sheeran
mary.sheeran@chalmers.se
Chalmers University
Gothenburg, Sweden

ABSTRACT

With the growing ubiquity of *Internet of Things* (IoT), more complex logic is being programmed on resource-constrained IoT devices, almost exclusively using the C programming language. While C provides low-level control over memory, it lacks a number of high-level programming abstractions such as higher-order functions, polymorphism, strong static typing, memory safety, and automatic memory management.

We present Hailstorm, a statically-typed, purely functional programming language that attempts to address the above problem. It is a high-level programming language with a strict typing discipline. It supports features like higher-order functions, tail-recursion, and automatic memory management, to program IoT devices in a declarative manner. Applications running on these devices tend to be heavily dominated by I/O. Hailstorm tracks side effects like I/O in its type system using *resource types*. This choice allowed us to explore the design of a purely functional standalone language, in an area where it is more common to embed a functional core in an imperative shell. The language borrows the combinators of arrowized FRP, but has discrete-time semantics. The design of the full set of combinators is work in progress, driven by examples. So far, we have evaluated Hailstorm by writing standard examples from the literature (earthquake detection, a railway crossing system and various other clocked systems), and also running examples on the GRiSP embedded systems board, through generation of Erlang.

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; **Domain specific languages**; • **Computer systems organization** → **Sensors and actuators**; *Embedded software*.

September 8–10, 2020, Bologna, Italy. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3414080.3414092

1 INTRODUCTION

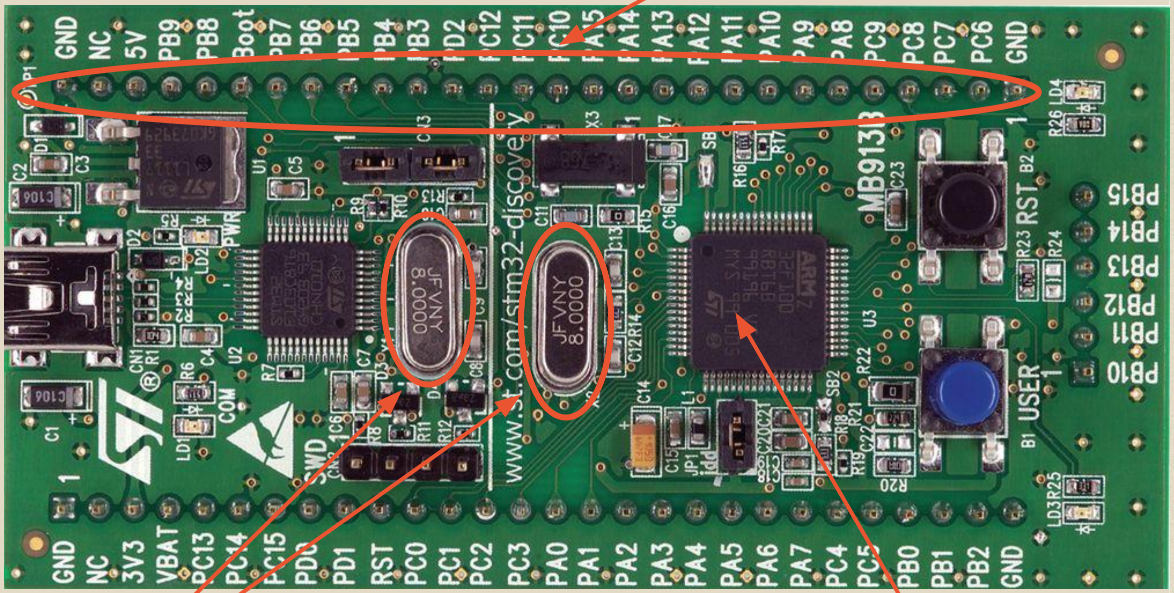
As the density of IoT devices and diversity in IoT applications continue to increase, both industry and academia are moving towards decentralized system architectures like *edge computing* [38]. In edge computation, devices such as sensors and client applications are provided greater computational power, rather than pushing the data to a backend cloud service for computation. This results in improved response time and saves network bandwidth and energy consumption [50]. In a growing number of applications such as aeronautics and automated vehicles, the real-time computation is more robust and responsive if the edge devices are compute capable.

In a more traditional centralized architecture, the sensors and actuators have little logic in them; they rather act as data relaying services. In such cases, the firmware on the devices is relatively simple and programmed almost exclusively using the C programming language. However with the growing popularity of edge computation, more complex logic is moving to the edge IoT devices. In such circumstances, programs written using C tend to be verbose, error-prone and unsafe [17, 27]. Additionally, IoT applications written in low-level languages are highly prone to security vulnerabilities [7, 58].

Hailstorm is a domain-specific language that attempts to address these issues by bringing ideas and abstractions from the functional and reactive programming communities to programming IoT applications. Hailstorm is a *pure*, *statically-typed* functional programming language. Unlike *impure* functional languages like ML and Scheme, Hailstorm restricts arbitrary side-effects and

PPDP 2020

192 KB RAM
168 MHz clock



I/O-Bound

Clocked

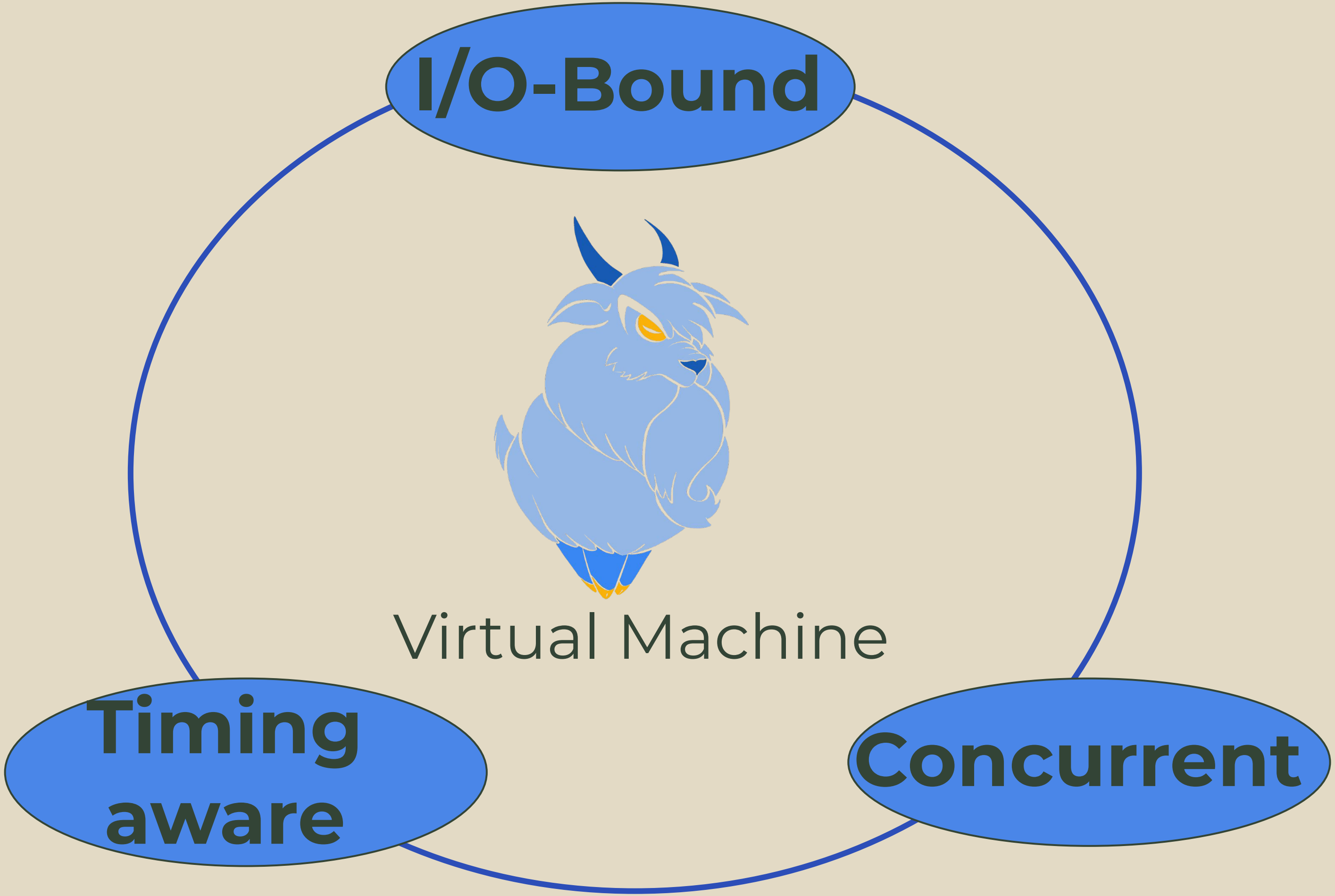
Bare-metal concurrent

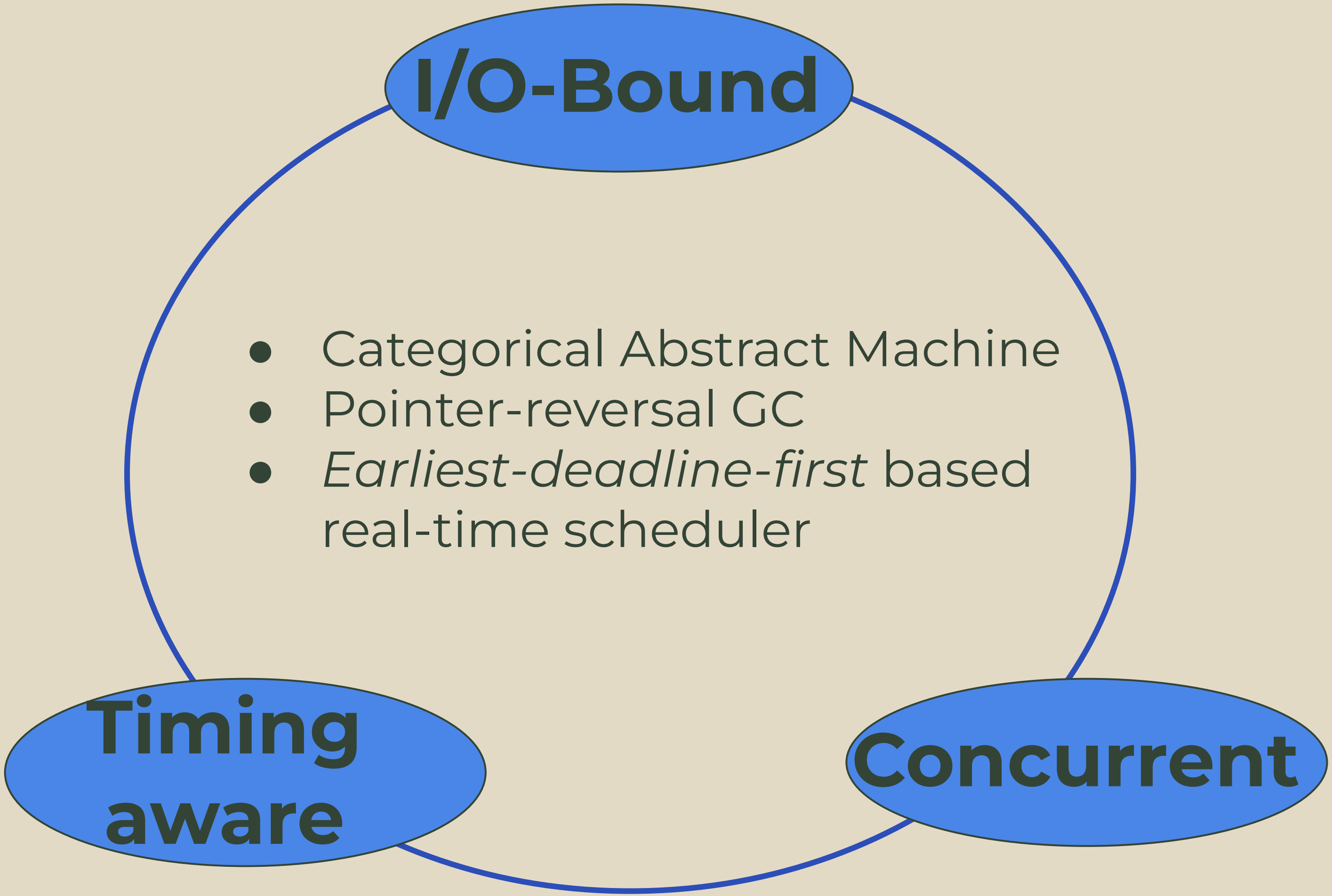
Programming Microcontrollers

Memory Unsafe

no real-time
constructs

!Concurrent


I/O-Bound
Virtual Machine
Timing aware
Concurrent



Cousineau G, Curien PL, Mauny M. The Categorical Abstract Machine. Science of computer programming. 1987

Complete Synchron API

```
spawn    : (() -> ()) -> ThreadId
channel  : ()  -> Channel a
send     : Channel a -> a -> Event ()
recv     : Channel a -> Event a
choose   : Event a   -> Event a  -> Event a
wrap     : Event a   -> (a -> b) -> Event b
sync     : Event a   -> a
syncT    : Time -> Time -> Event a -> a
spawnExternal : Channel a -> Driver -> ExternalThreadId
```

CML: A Higher-order Concurrent Language. John Reppy. PLDI 1991.

Complete Synchron API

```
spawn    : (() -> ()) -> ThreadId
channel  : ()  -> Channel a
send     : Channel a -> a -> Event ()
recv     : Channel a -> Event a
choose   : Event a   -> Event a  -> Event a
wrap     : Event a   -> (a -> b) -> Event b
sync     : Event a   -> a
syncT    : Time -> Time -> Event a -> a
spawnExternal : Channel a -> Driver -> ExternalThreadId
```

CML: A Higher-order Concurrent Language. John Reppy. PLDI 1991.

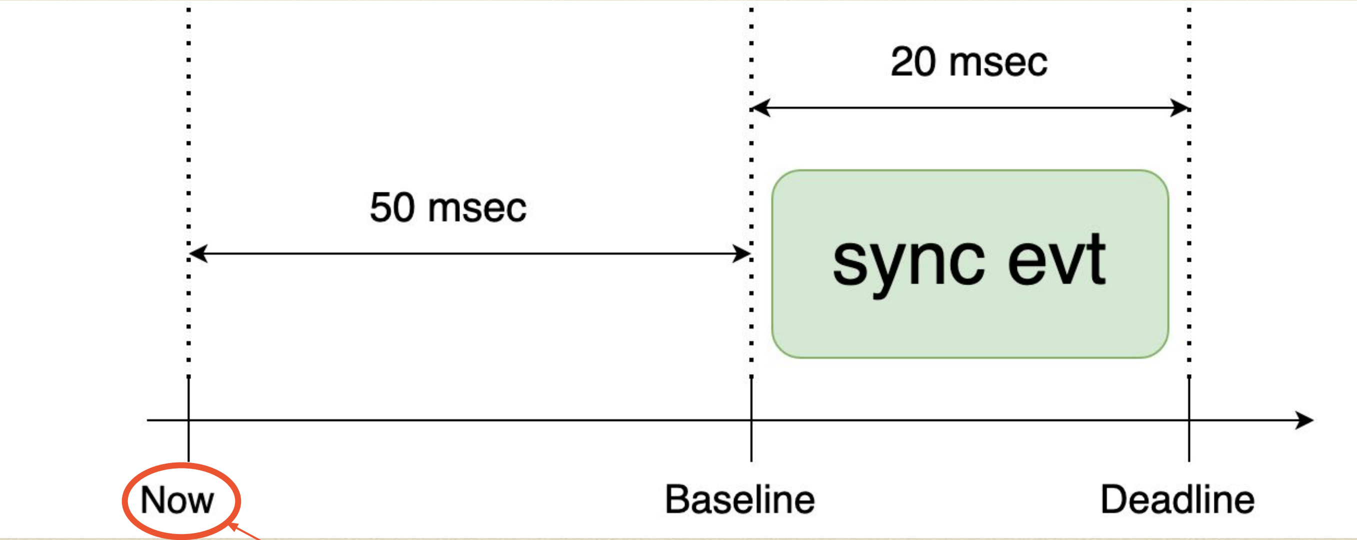
Timed Synchronisation

syncT : Time -> Time -> Event a -> a

Relative Baseline

Relative Deadline

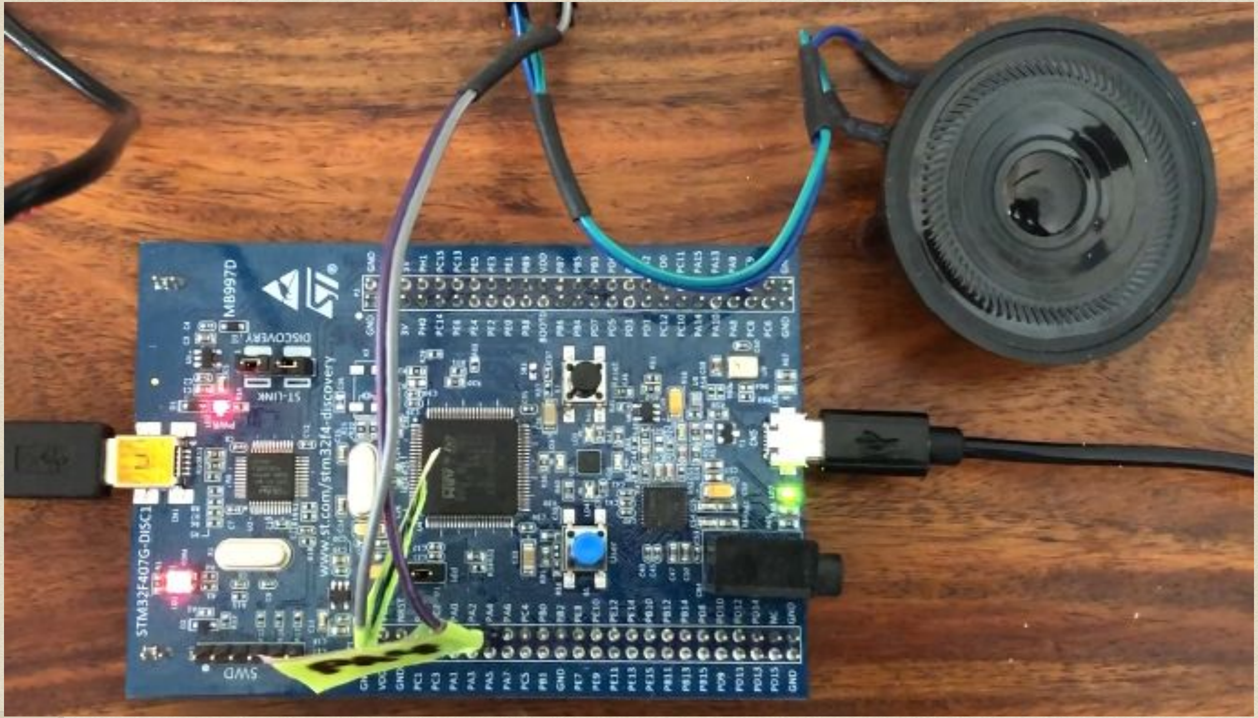
syncT (msec 50) (msec 20) evt



Logical “now”

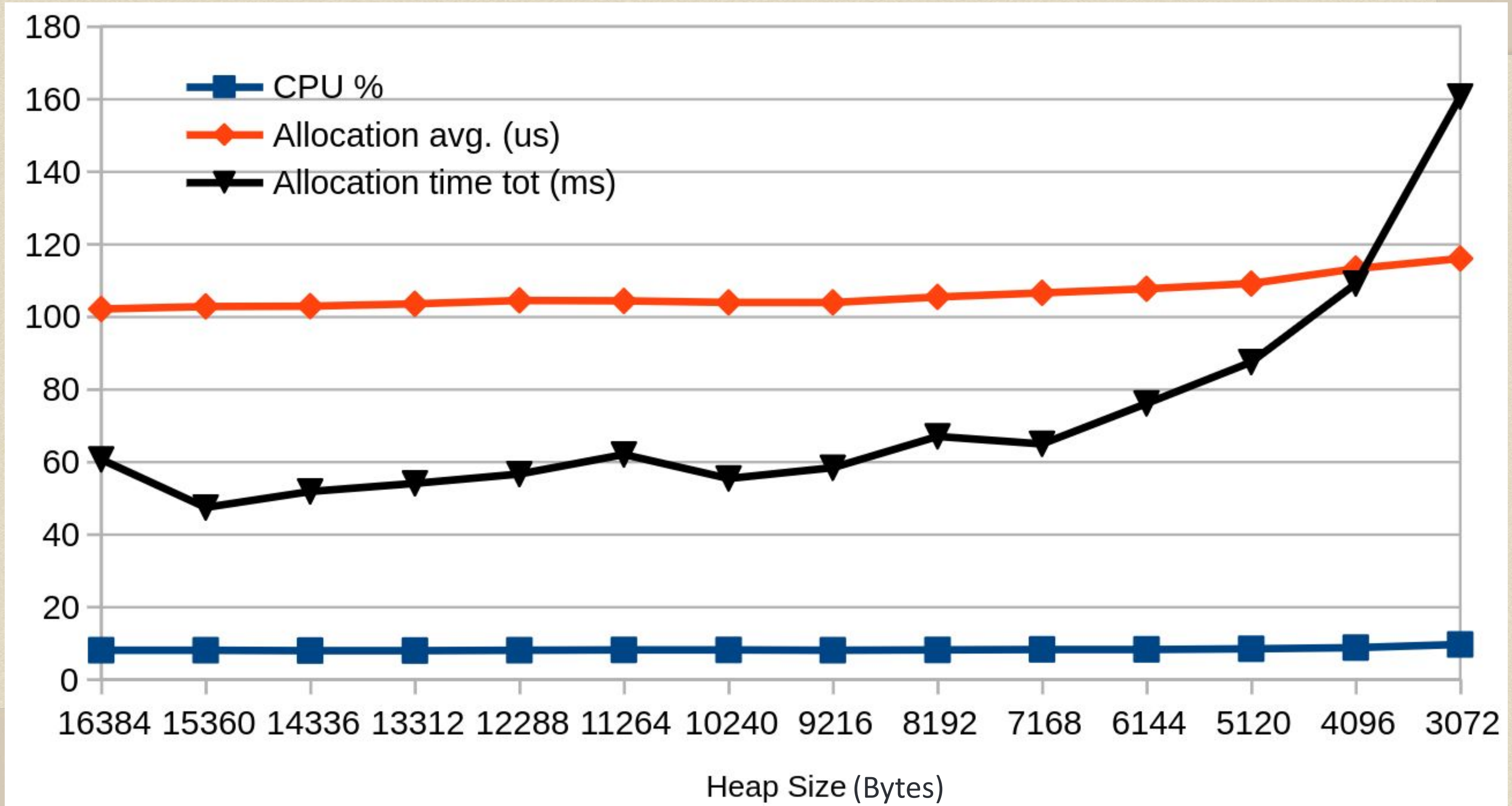
1. Berry G. The Foundations of Esterel. MIT Press 2000.
2. Nordlander J et al. Timber: A programming Language for Real-Time Embedded Systems. Technical Report 2002.

CASE STUDY

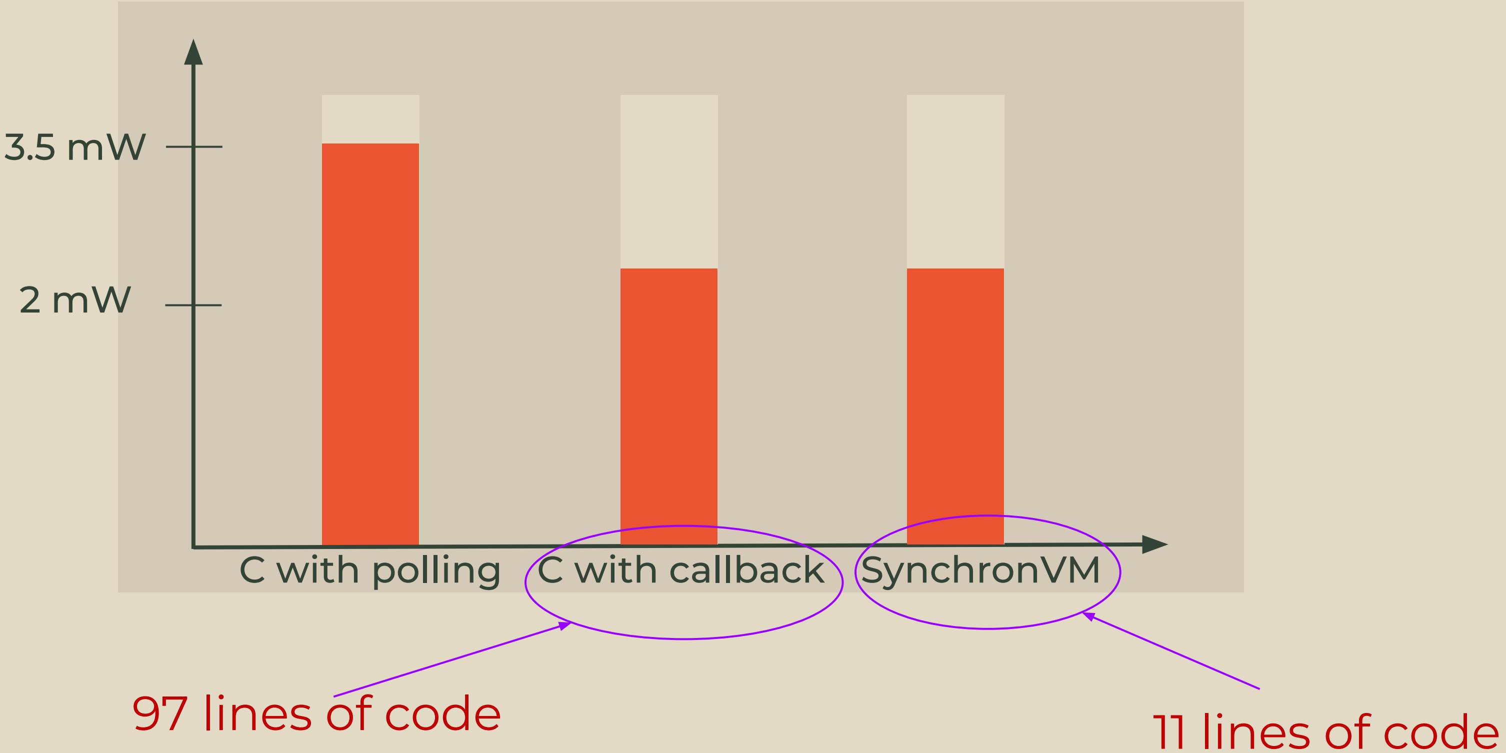


- Soft real-time
- ~440 Hz note frequency

Measurements



Button Blinky Power Usage



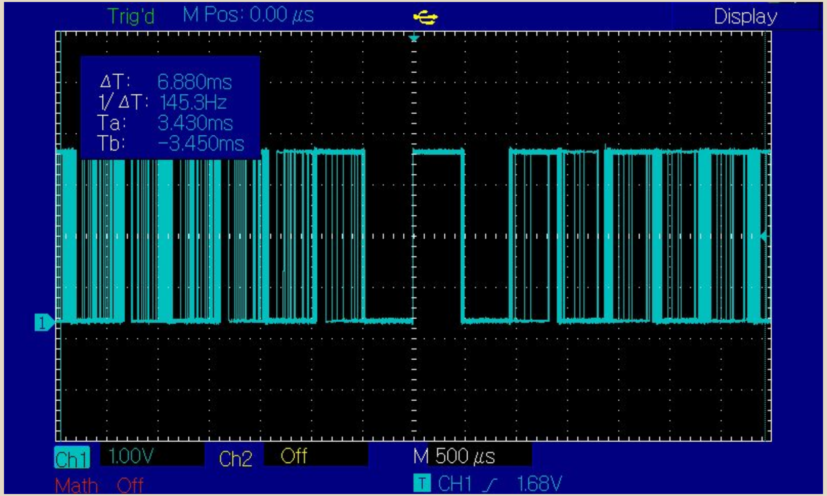
Jitter and Precision

```c
while (1) {
 uint32_t state = GPIO_READ(23);
 if (state) {
   GPIO_CLR(23);
 } else {
   GPIO_SET(23);
 }
 usleep(400);
 }
// main method and other setup elided
```

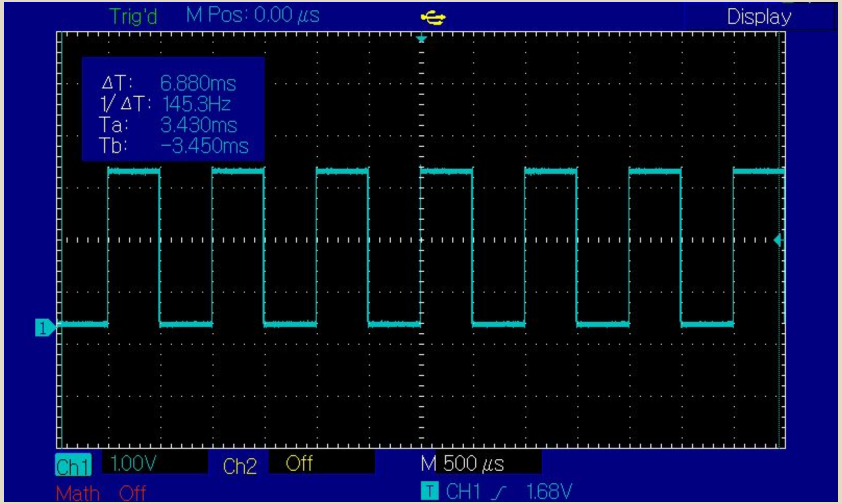
```
ledchan = channel ()

foo : Int -> ()
foo val =
 let _ = syncT 500 0 (send ledchan val)
  in foo (not val)

main =
 let _ = spawnExternal ledchan 1
  in foo 1
```

1 KHz Square Wave

Jitter and Precision



C / Raspberry Pi

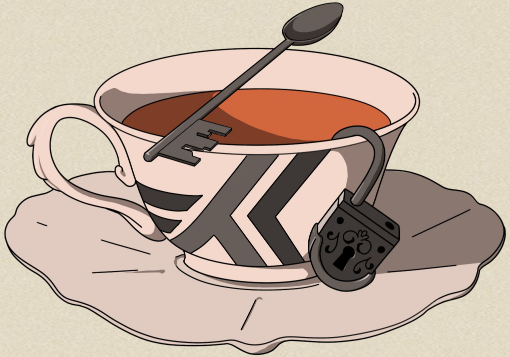


Synchron/STM32F4

1 KHz Square Wave

Contributions

Attacker Model 1



HasTEE⁺

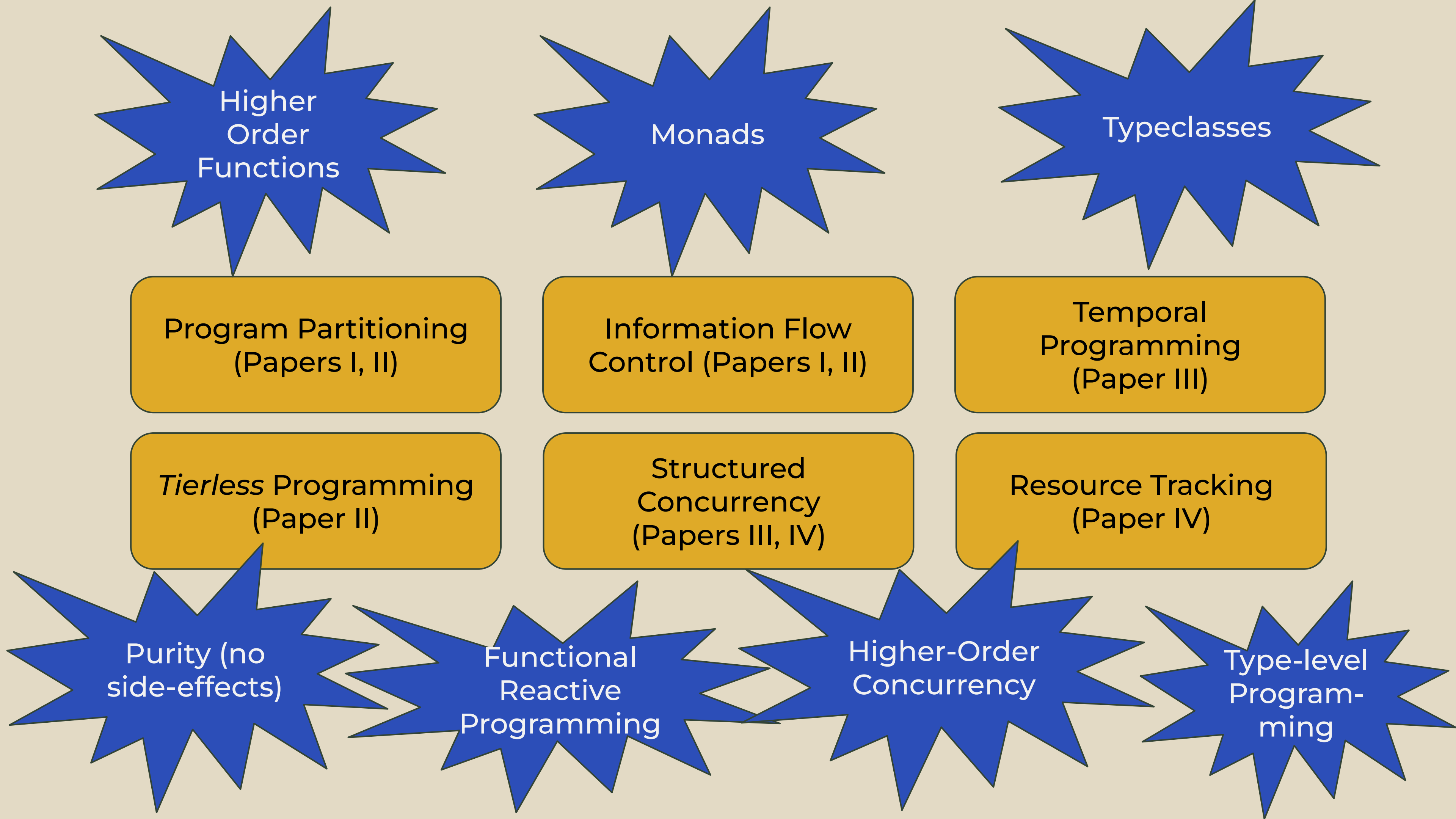
for reducing ***trust*** on low-level software

Attacker Model 2

SynchronVM

for ***memory-safe***, ***soft real-time*** embedded systems

“Securing Digital Systems through Programming Language Techniques”

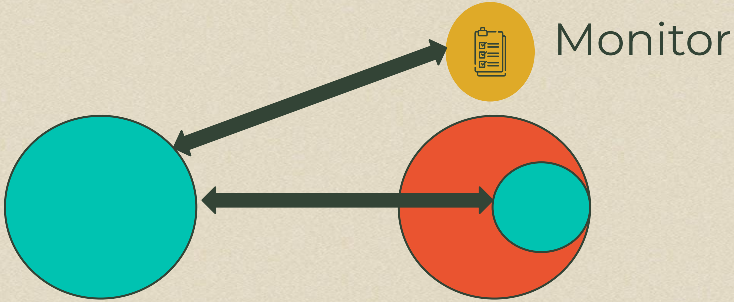

Higher Order Functions
Monads
Typeclasses
Program Partitioning (Papers I, II)
Information Flow Control (Papers I, II)
Temporal Programming (Paper III)
Tierless Programming (Paper II)
Structured Concurrency (Papers III, IV)
Resource Tracking (Paper IV)
Purity (no side-effects)
Functional Reactive Programming
Higher-Order Concurrency
Type-level Program-ming

“Securing Digital Systems through Functional Programming Abstractions”

Future Work

Future Work

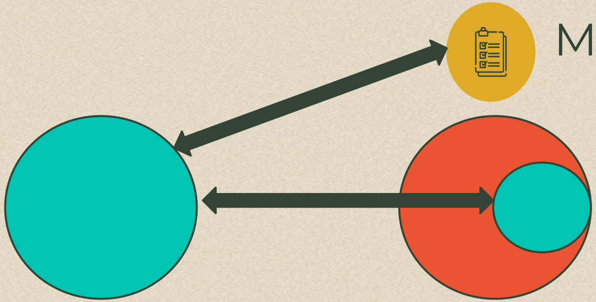
$$\square(p \to \lozenge q) \wedge \square(q \to \square q)$$



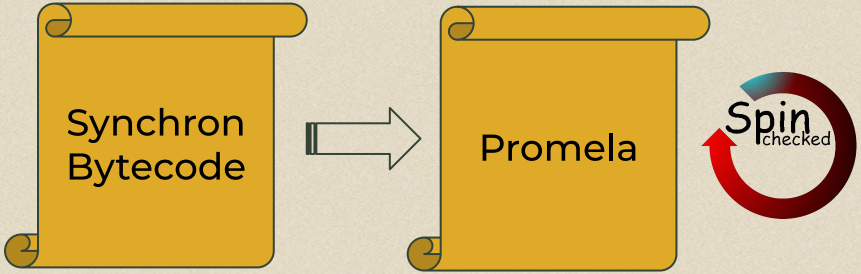
Property-based Attestation

Future Work

$\square(p \to \Diamond q) \land \square(q \to \square q)$



Property-based Attestation



Model-checking Synchron

ACKNOWLEDGEMENT

Thanks to Andrea Svensson for the cool logos.

ACKNOWLEDGEMENT

Thanks to Andrea Svensson for the cool logos.

Thanks to the SSF Octopi project for funding this research.

ACKNOWLEDGEMENT

Thanks to Andrea Svensson for the cool logos.

Thanks to the SSF Octopi project for funding this research.

Thanks to my co-authors.

Mary

Joel

Alejandro

Robert

Koen

THANKS!

https://github.com/Abhiroop/HasTEE

https://github.com/SynchronVM/SynchronVM