Thesis for The Degree of Doctor of Philosophy

Functional Programming for Securing Cloud and Embedded Environments

Abhiroop Sarkar

Department of Computer Science and Engineering
Chalmers University of Technology | University of Gothenburg
Gothenburg, Sweden, 2024

Functional Programming for Securing Cloud and Embedded Environments

Abhiroop Sarkar

© Abhiroop Sarkar, 2024
except where otherwise stated.
All rights reserved.

ISBN xxx-xx-xxxx-xxx-x
Doktorsavhandlingar vid Chalmers tekniska högskola, Ny serie nr xxxx.
ISSN 0346-718X
Technical Report No. XXXX

Department of Computer Science and Engineering
Division of Computing Science
Chalmers University of Technology | University of Gothenburg
SE-412 96 Göteborg,
Sweden
Phone: +46(0)31 772 1000

Printed by Chalmers Digitaltryck,
Gothenburg, Sweden 2024.

"The price of reliability is the pursuit of the utmost simplicity."
- Tony Hoare

Functional Programming for Securing Cloud and Embedded Environments

Abhiroop Sarkar

Department of Computer Science and Engineering
Chalmers University of Technology | University of Gothenburg

Abstract

The ubiquity of digital systems across all aspects of modern society, while beneficial, has simultaneously exposed a lucrative attack surface for potential adversaries and attackers. Consequently, securing digital systems becomes of critical importance. In this dissertation, we address the security concerns of two classes of digital systems: (i) cloud systems, co-locating multiple applications and relying on a large trusted code base for software virtualisation, and (ii) embedded systems, resource-constrained environments that typically employ unsafe programming languages for application development.

Accordingly, we present a collection of functional programming-based tools designed for securing both cloud and embedded systems. The dissertation is organised into two parts.

Part I introduces two successive versions of a domain-specific language (DSL) designed for programming Trusted Execution Environments (TEEs), such as Intel SGX. TEEs isolate applications from low-level system software with large codebases, such as operating systems and hypervisors, thereby minimizing the trusted computing base and reducing the resultant attack surface of cloud applications. Broadly, the DSL contributes the following: (1) It facilitates automatic type-based *program partitioning* between trusted and untrusted code, (2) It supports dynamic information flow control mechanisms for ensuring *data confidentiality*, (3) It integrates with an automated *remote attestation* framework to preserve *TEE integrity*, and (4) It offers a *tierless programming model* that helps minimise errors arising from multi-tier confidential computing applications, requiring adherence to complex data exchange protocols.

Evaluations for Part I involve expressing confidential computing applications, such as (i) a privacy-preserving federated learning application, (ii) an encrypted password wallet, and (iii) a data-clean room design pattern for multiple parties to conduct data analytics.

Part II contributes a functional language runtime and a functional reactive programming language targeting embedded systems, with the goal of raising the level of abstraction and ensuring memory and type safety. The runtime offers a unified message-passing framework for handling both software messages and hardware interrupts, along with a novel timing operator to capture the notion of time. This allows for expressing classical (1) *concurrent*, (2) *I/O-bound*, and (3) *timing-aware* embedded systems applications in a declarative manner. Similarly, the reactive programming language is a declarative, pure functional language built on top of the runtime. It tracks unique side effects in its type system using a feature called *resource types*.

Evaluations for Part II ran the language and runtime on microcontrollers like NRF52, STM32, and *GRiSP* boards, microbenchmarking resource efficiency parameters including memory footprint, garbage collection latency, throughput, jitter, and interpretive load, demonstrating acceptable overheads.

The programming artifacts resulting from this dissertation comprise the *HasTEE* and *HasTEE*$^+$ DSLs for programming TEEs, the *Synchron* C99-based portable embedded systems runtime, and the *Hailstorm* reactive programming language for embedded systems. All these programming artifacts are made publicly available, along with the evaluation procedures, encouraging further experiments in securing both cloud and embedded systems.

Keywords

functional programming, trusted execution environment, information flow control, microcontrollers, real time, runtime, functional reactive programming

List of Publications

Appended publications

This thesis is based on the following publications:

[**Paper I**] **Abhiroop Sarkar**, Robert Krook, Alejandro Russo, Koen Claessen, *HasTEE: Programming Trusted Execution Environments with Haskell.*
In Proceedings of the 16th ACM SIGPLAN International Haskell Symposium, 2023 (pp. 72-88).

[**Paper II**] **Abhiroop Sarkar**, Alejandro Russo, *HasTEE$^+$: Confidential Cloud Computing and Analytics with Haskell.*
Submitted to ESORICS 2024, under review.

[**Paper III**] **Abhiroop Sarkar**, Bo Joel Svensson, Mary Sheeran, *Synchron - An API and Runtime for Embedded Systems.*
In Proceedings of the 36th European Conference on Object-Oriented Programming, ECOOP 2022,(pp. 17:1-17:29).

[**Paper IV**] **Abhiroop Sarkar**, Mary Sheeran, *Hailstorm: A Statically-Typed, Purely Functional Language for IoT Applications.*
In Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming, 2020 (pp. 1-16).

Other publications

The following publications were published during my PhD studies. However, they are not appended to this thesis, due to contents overlapping that of appended publications.

[**a**] **Abhiroop Sarkar**, Robert Krook, Bo Joel Svensson, Mary Sheeran, *Higher-Order Concurrency for Microcontrollers.*
In Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, 2021 (pp. 26-35).

Acknowledgment

Acknowledgement page. Include also funding information here!

Contents

Overview

Chapter 1

Introduction

The pervasiveness of digital systems has brought immense productivity to several aspects of modern society. Sectors such as finance, healthcare, telecommunications, information technology, retail, and manufacturing are intrinsically reliant on digital systems to enhance efficiency, data management, communication, and overall business competitiveness. However, this significant reliance on digital artifacts opens a lucrative attack surface, exposing society to unprecedented attackers and adversaries.

Consider, for instance, the classic computer worm[1] *Stuxnet* [1]. Stuxnet, identified in 2010, was crafted to compromise the Siemens Step7 software, which controls programmable logic controllers (PLCs) in industrial processes. It exploited at least four zero-day vulnerabilities in Microsoft Windows, including a remote-shell execution vulnerability (CVE-2010-2568 [2]) and a hard-coded password backdoor in the Windows driver for the Siemens PLC (CVE-2010-2772 [3]). The attack vectors enabled Stuxnet to interfere with the real-time behavior of the PLCs, causing irreparable damage to the entire control system. Similar attacks, like the Equifax data breach [4] (CVE-2017-5638 [5]), the *WannaCry* ransomware attack (CVE-2017-0143 [6]), and numerous others that are too many to list, demonstrate the prevalence of such incidents.

Aside from the above discussed instances of deliberate *cyber-espionage*, there have been cases of genuinely overlooked vulnerabilities in popular software libraries, such as the Heartbleed vulnerability in the widely popular OpenSSL project [7]. This flaw, exploiting an *unvalidated buffer over-read*, allowed attackers to retrieve sensitive data, including private keys and user credentials. Another recent example is the exploitation of a flaw within Java's Remote Method Invocation (RMI) functionality in the seemingly innocuous Java logging library *log4j* to launch a remote-code-execution attack [8].

A common theme begins to emerge from our discussions above on software vulnerabilities, illustrating a connection between software security and *software correctness*. For instance, both Microsoft and Google assert that 70% of their reported security bugs are linked with *memory safety* [9], [10] – a software correctness condition. Similarly, NASA's investigation into the 1999 Mars

[1] a software *worm*, unlike a virus, is capable of self-replication without host intervention

Climate Orbiter crash revealed that a *type-unsafe* conversion between English and metric units in the ground software [11], a *software correctness* violation, was one of the root causes behind the incident.

Complicating matters, certain attacks discussed earlier remain difficult to thwart, even when the programmer ensures memory safety and type safety throughout the entire program. In instances similar to *Stuxnet* [1], attackers exploit vulnerabilities in the host operating system and the surrounding infrastructure to mount their attacks. To differentiate between various classes of attack vectors, we use the Stuxnet example to categorise the two types of digital systems targeted by attackers:

Cloud Systems: Such systems are commonly represented by powerful machines that virtualise hardware through hypervisors and containers [12]. Typically resource-rich, these systems can host multi-tenanted, type-safe, and memory-safe software, while providing basic memory protection through the underlying hardware's memory management unit. Attacks like Stuxnet[2] tend to *exploit the memory and type unsafety of the underlying operating system and its associated drivers*, compromising critical parts of the system.

Embedded Systems: These systems employ low-powered, resource-constrained but high-volume hardware, typically in the form of microcontrollers. The resource constraints dictate the use of highly memory-efficient software, typically written in the C and C++ family of programming languages, which are both memory and type-unsafe. Additionally, microcontrollers lack memory management units for cost reduction. Exploiting the memory-unsafety of such systems, the second part of the Stuxnet attack *hijacks the control flow of the embedded-system-software and injects a malicious state-machine that degrades the real-time behavior of the control system*, causing catastrophic damages.

In this dissertation, our aim is to propose a solution based on programming language techniques to address security concerns in both classes of systems. In particular, we consider *functional programming languages* built on the concept of *pure* mathematical functions – that do not perform *side-effects* – as a means of designing more correct and, consequently, more secure software.

Functional Programming and Correctness

Functional programming focuses on expressing computation as the evaluation of pure mathematical functions. Pure functional languages, such as Haskell, emphasise *immutability*, *referential transparency*, and the *absence of side effects*.

These pleasant properties have historically cultivated a culture of equational reasoning [13], even in the presence of a variety of computational effects [14], giving rise to a category-theory-inspired *algebra of programming* [15] in functional languages. Furthermore, theoretical results [16] have justified the

[2]The Stuxnet attack was not conducted on a typical public cloud such as Amazon Web Services but on a private desktop machine infected with a USB.

informal style of equational reasoning adopted by Haskell programmers for reasoning in the presence of non-termination and the undefined $\perp$ value.

The strong type system of statically-typed functional languages, like Haskell, not only enables the development of type-safe software but also facilitates the encoding of lightweight proofs within the type system [17]. Haskell's type system is further extended by tools such as *LiquidHaskell* [18], enabling the encoding of invariants that generalise Hoare Logic [19]. Additionally, libraries like *QuickCheck* [20] allow automated property-based testing of Haskell programs.

The above discussion positions functional languages, such as Haskell, as ideal tools for designing security-critical and correct software. However, upon revisiting our preceding discussions on the classes of attacks we aim to prevent, we observe that the first class of attacks on cloud systems arises outside the safety guarantees provided by the software. Meanwhile, the second class of attacks on embedded systems exploits the fact that developers often program in a domain where functional languages (or as such any high-level languages) are uncommon or absent. In summary, we list the key challenges below:

- **Challenge 1** *Functional Programming alone cannot guarantee overall system correctness.* For instance, vulnerabilities in the underlying OS or a foreign public library are not captured in the correctness condition of a software written in Haskell.
- **Challenge 2** *Security properties are quite distinct from the category theoretic properties that functional languages primarily reason about.* For instance, *access control* properties or *data integrity* properties.
- **Challenge 3** *Functional Programming lacks abstractions for effectively expressing real-time computations, abundant in embedded systems.*
- **Challenge 4** *Functional Programming naturally evolved for bulk data-transformation computations, not I/O-heavy reactive embedded systems.*

This dissertation addresses the first two challenges by proposing a domain-specific language that combines specialised *hardware security extensions* with *language-based security* [21] to provide more concrete security guarantees to the programmer. To tackle the final two challenges, we introduce a set of *new functional programming abstractions* and a specialised *functional language runtime*, designed specifically for embedded systems.

In the next two sections, we will provide the necessary background on the safety and security of cloud and embedded systems, essential for presenting our contributions. Before that, we conclude this section with definitions of useful terms that have been and will continue to be used throughout the dissertation.

Some useful definitions

Safety. Safety is the absence of undesirable states, ensuring a system remains within defined permissible states with no violation of critical properties during execution. Examples include *memory safety*, *type safety* and *thread safety*. Security policies are often stated as safety properties [22], [23].

Correctness. The correctness of a system is defined by its precise adherence to formal specifications. Often, the formal specification takes the form of a collection of safety properties. However, the specification is not limited to safety; it may also include other properties like *liveness* and *fairness*.
Trustworthy. A system which is provably or demonstrably correct (i.e. it meets its specification) will be *trustworthy* [24] – at least within the parameters of its specification.
Trusted. An entity can be trusted if it always behaves in the expected manner for the intended purpose [24]. The expected behaviour is often closely tied to the correctness criterion of the system.
Secure. The Bell-LaPadula model [25] defines security in the form of a state-machine model for enforcing access control. A system state is defined to be *secure* if the only permitted access modes of subjects to objects are in accordance with a specific security policy.

1.1 Security in the Cloud

Starting from the early 2000s, the world has seen a massive shift in large-scale IT operations from bare-metal servers and private digital infrastructure to shared, on-demand, pay-per-use public digital services owned by tech giants such as Amazon [26], Microsoft [27], and Google [28]. This trend, known as *Cloud Computing*, heavily relies on resource pooling and virtualisation to achieve economies of scale that drive overall deployment costs on a cloud much lower than maintaining a private digital infrastructure for an organization [29].

The role of virtualisation in the cloud has been achieved through traditional isolation technologies such as hypervisors and operating systems, commonly implemented in memory-unsafe languages. This provides attackers with a fairly large and unsafe *trusted code base* that can be subjected to various types of sophisticated attacks [30]–[35].

Concerned by the alarming trend of low-level attacks against hypervisors and operating systems, hardware vendors such as Intel, ARM, and AMD have embraced an emerging security paradigm known as Confidential Computing [36]. At its core, confidential computing aims to secure what is known as *data in use*. *Data in use* refers to in-memory data on a physical machine, distributed across DRAM, cache lines, page tables, and other CPU registers. While encryption has been fairly successful in protecting secrets for *data at rest* (such as databases and file systems) as well as *data in transit* (such as networks using TLS), the need for efficiency and performance has prevented encryption from effectively protecting *data in use*.

To protect *data in use*, hardware vendors introduced the concept of a *Trusted Execution Environment (TEE)*, providing hardware-enforced *isolation* for in-memory data. A TEE unit essentially offers a *disjoint* region of code and data memory, enabling the *isolation of a program's execution and state* from the underlying operating system, hypervisor, I/O peripherals, BIOS, and other firmware. Some of the most popular Trusted Execution Environment implementations from leading hardware vendors include Intel Software Guard

Extensions (SGX) [37], ARM TrustZone [38], AMD Secure Encrypted Virtualization (SEV) [39], and Intel Trust Domain Extensions (TDX) [40]. Fig. 1.1 illustrates a comparison of the attack surface for an application running on the cloud without and with a TEE unit engaged.

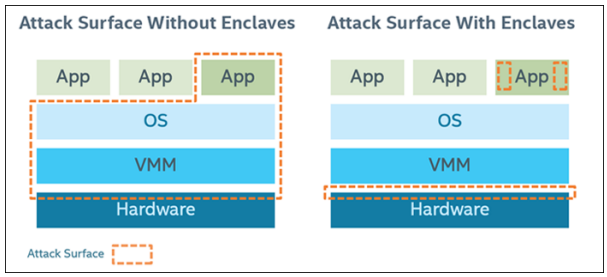


Figure 1.1: The software attack surface (trusted code base) without and with a TEE unit; Image source: Intel.

In Figure 1.1 on the right, observe that the OS and the hypervisor (also known as a virtual machine monitor or VMM) are outside the attack surface of the application stack. The hardware-protected region of code and data memory has been variously referred to as an *enclave*, *realm*, or *isolate*. This *enclave* is provided with strong guarantees of *confidentiality* and *integrity* by the underlying hardware.

The actual hardware implementation for providing said guarantees varies widely between different hardware vendors. For instance, Intel SGX implements the enclave within the virtual memory and employs encryption when the data moves to the shared cache. In contrast, ARM TrustZone provides a physically disjoint memory segment with separate memory buses.

Another key aspect of confidential computing is the presence of a hardware root of trust and a supported *remote attestation* protocol. The attestation process usually employs a cryptographic hash of the enclave code and data to produce a *measurement*, which can be used by a communicating third party to verify the integrity of the enclave.

This combination of hardware-enforced memory isolation and the remote attestation mechanism enables programmers to deploy security-critical software on a public, co-tenanted cloud without having to trust the virtualisation infrastructure of the cloud provider. However, an Achilles' heel in the large-scale adoption of confidential computing has been the low-level and awkward programming models offered by various hardware vendors [41].

From a broad perspective, the programming model demands intricate *program partitioning* that is complex and error-prone and relies on complicated vendor-specific toolchains. Adding to the woes, the majority of language support for programming TEEs is using the C/C++ language family, reintroducing the wide class of memory-unsafety-related vulnerabilities [42] that we have discussed earlier. Also, the strong confidentiality and integrity guarantee of an *enclave* does not extend to the inclusion of public libraries, such as cryptographic libraries, which can result in leaking user secrets [21].

Opportunity. *This opens up an opportunity for us to employ memory-safe functional programming abstractions to (1) implement program partitioning and (2) enforce language-based information flow control mechanisms [21] for protecting the confidentiality and integrity of security-critical software.*

Before presenting our approach to capitalise on this opportunity, we will switch gears to discuss the second type of systems mentioned earlier – embedded systems. We will provide a brief background on them and then outline our contributions.

1.2 Safety in Embedded Systems

An embedded system, unlike traditional disciplines of batch computing and data processing, is typically *embedded* within a larger system that involves interactions with the physical environment. Henzinger and Sifakis [43] defines an embedded system as *"an engineering artifact involving computation that is subject to physical constraints. The physical constraints arise through two kinds of interactions of computational processes with the physical world: (1) reaction to a physical environment, and (2) execution on a physical platform."*

The first category of interactions gives rise to behavioural requirements on an embedded system application such as *deadline*, *throughput*, *response time*, etc., that can have a tangible impact on the physical environment. The physical interaction component demands that an embedded application be *reactive* to any stimulus provided by its environment.

On the other hand, the second category results in more implementation-specific requirements such as limited power usage, memory usage, etc. These constraints dictate the economics of embedded systems, which are deployed in large numbers in most applications areas (like sensor networks and cars) and require application development platforms that prioritise resource sensitivity over high performance.

Consider a typical embedded systems application area like wireless sensor networks (WSNs) [44], where the number of deployed devices ranges from hundreds to thousands. Such large deployments are made cost-effective by reducing the price of an individual unit to be in the range of 10 to 100 dollars.

The cost of these devices is cut down by manufacturing them to be heavily resource-constrained. Such devices, often microcontrollers, have a small die area with simple circuitry, missing components like on-chip cache, transistors for superscalar execution, etc. As a result, these devices are power efficient and require little cooling. They frequently use ARM-based microcontrollers, also with constrained memory and clock speed.

Given this inclination toward resource efficiency, the embedded systems industry primarily conducts software development in the C/C++ family of programming languages. Figure 1.2 show the results of a 2019 survey of the embedded systems market conducted by EE Times [45]. The survey gathered responses from 958 participants across various sectors of the embedded systems industry, allowing for multiple responses regarding the developers' primarily used programming language.

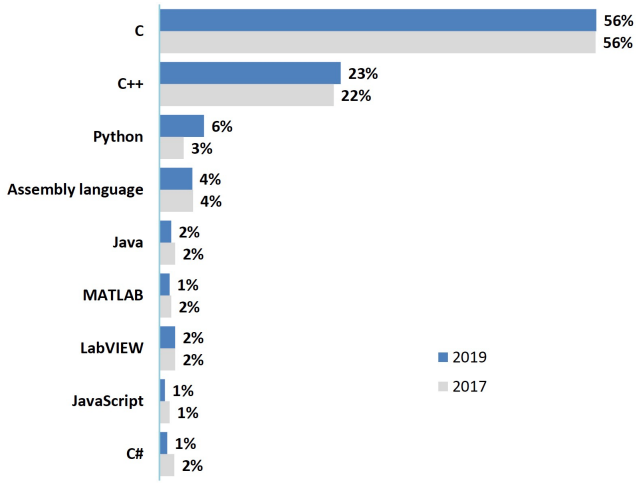


Figure 1.2: EETimes 2019 Embedded Systems Markets Study [45]

Figure 1.2 clearly shows the dominance of the C programming language in the embedded systems industry. The second-most popular language, C++, often uses a highly specialised subset of modern C++ standards. These subsets exclude several high-level features of C++ and constrain the language, effectively making it behave more like C.

The omnipresence of C can be partially attributed to microcontroller vendors exclusively supporting C compiler toolchains [46]. Today, any new microcontroller entering the market is expected to inherently support the popular ARM GNU toolchain or another vendor-specific C toolchain. One of the key benefits of C is that it is considered as a sufficiently low-level language that can enable the programmer to write resource-conscious programs. Restricted subsets of C, such as MISRA C [47], allow writing deterministic programs with statically predictable object lifetimes.

However, this perceived strength of C as a *low-level systems language* becomes a security disadvantage for the system. Examining the memory-unsafety of C, MITRE analysed the list of known exploited vulnerabilities from the American Cybersecurity and Infrastructure Agency [48]. Their findings indicate that the top three culprits—*use-after-free*, *heap-based buffer overflow*, and *out-of-bound writes*—all stem from memory unsafety.

Although the adoption of C is driven by resource efficiency, Henzinger and Sifakis' definition [43] highlights an unaddressed component in embedded systems – reaction to the physical environment or *reactivity*. Operationally, reactive applications are *I/O-intensive* due to their continual interactions with the external environment. Additionally, the external environment can supply a variety of stimuli, best handled by breaking down an application into several *concurrent* stimulus handlers.

A third property that arises as a result of interaction with the external world is the notion of being *timing-aware*. Responses to certain specific types

of stimuli often require reactions within a given deadline and at a periodic rate. Thus, we can assemble three important operational properties of reactive systems, which are embodied in embedded systems applications, as follows:

1. *I/O-intensive*
2. *Concurrent*
3. *Timing-aware*

The C programming language is not a concurrent language. There are some ad-hoc libraries, such as Protothreads [49], to mimic concurrent behavior, but the intrinsic language semantics are not concurrent. Moreover, in the presence of callback-based I/O driver APIs for embedded systems, C programs start suffering from a programming anti-pattern known as *callback hell* [50]. Also, in terms of real-time behaviors, C is deficient and often resorts to vendor-specific, bespoke real-time extensions to the original language [51].

There is a clear gap for a high-level, memory-safe, and type-safe language for embedded systems that embodies the discussed reactive properties while efficiently running programs in a resource-sensitive manner. To design such systems, we need a fundamentally concurrent language that allows structuring callback-based, low-level APIs into programs with a natural control-flow while respecting the physical timing requirements.

Opportunity. *This opens up our second opportunity to employ memory-safe functional programming abstractions and extend them with structured (1) concurrency, (2) I/O, and (3) temporal programming primitives for safer embedded systems programming.*

1.3 The Thesis

Recapitulating, in this dissertation, we are considering two classes of systems - cloud systems and embedded systems. Having discussed the challenges for the safety and security of both classes of systems, we have also identified two opportunities for contributions. With that in consideration, we now present *the thesis* underlying this dissertation.

THESIS STATEMENT

> *Functional Programming abstractions, in conjunction with modern hardware security extensions and language-based information flow control mechanisms, present a viable path to providing strong security guarantees for cloud systems. Extending these functional programming abstractions with structured concurrency and temporal programming primitives further expands these security assurances to embedded systems, thereby establishing a foundation for building safer and more secure digital systems overall.*

The rest of this dissertation consists of four papers aimed at supporting our proposed thesis. Papers I and II introduce two successive versions of a domain-specific language embedded in Haskell for programming Trusted Execution Environments in cloud systems. Papers III and IV present the design and implementation of specialised functional programming languages and runtimes targeted towards embedded systems. Figure 1.3 shows the outline of this dissertation.

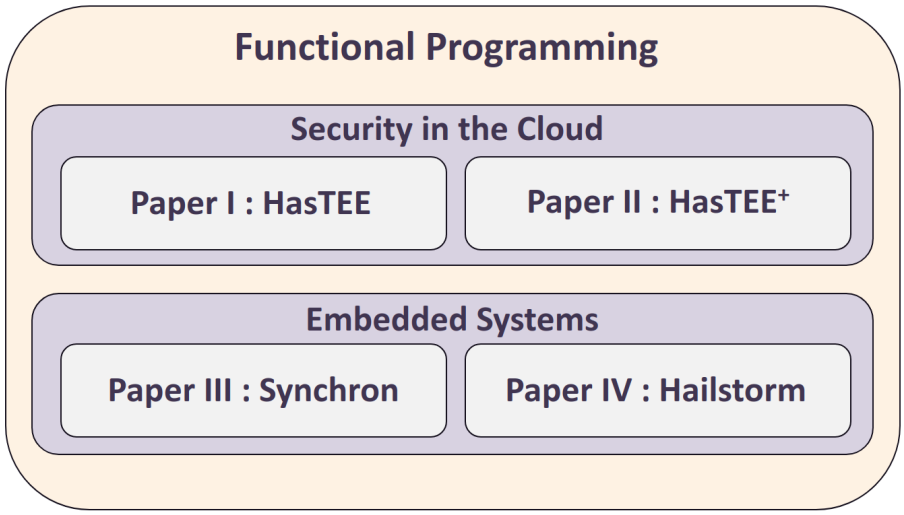


Figure 1.3: Outline of the dissertation

In the remainder of this chapter, we will provide a high-level summary of the four papers and highlight their main contributions. Note that the contributions in Papers III and IV have also been discussed in our licentiate thesis [52].

1.4 Papers I and II: HasTEE and HasTEE+

Paper I

Paper I presents HasTEE [53], a domain-specific language (DSL), designed for programming Trusted Execution Environments (TEEs) such as Intel SGX.

One of the challenges that arises when running high-level, memory-safe, and type-safe languages on a TEE like Intel SGX is the use of a restricted C standard library implementation provided by the vendor, such as *tlibc* [54]. Libraries like *tlibc* are not POSIX-compliant and severely restrict memory mapping, threading, timing, and various other APIs used for interacting with the OS and hardware. This significantly impacts the porting of standard programming language runtimes that heavily rely on POSIX compliance, especially for high-level features such as efficient and automatic memory management.

Trusted Runtime. A crucial contribution of HasTEE was enabling the Glasgow Haskell Compiler (GHC) Runtime [55] to operate on Intel SGX

hardware. This enables the execution of a language with a strong type system and automatic memory management on a TEE, which inherently offers stronger correctness guarantees than programming TEEs in the C family of languages. The implementation details are discussed in Chapter 5.

Another key challenge in TEE programming is the cumbersome multi-project programming model that necessitates partitioning the entire program and its state into two projects – a smaller trusted project for the enclave and the remaining untrusted project responsible for communication with the trusted counterpart. Every hardware vendor provides its own toolchain for accomplishing this program partitioning, and the result is often error-prone, requiring adherence to complex data-copying protocols [56] for communication between the two projects.

In addressing this challenge, our solution is creating a fairly general programming interface for HasTEE. This interface is designed to capture the *lifting* of a polymorphic Haskell function into an enclave and allowing *function application* within the *enclave*. It also features mutable references to model state. Figure 1.4 show the core HasTEE API.

```
-- mutable references for modeling state
liftNewRef :: a → App (Enclave (Ref a))
readRef    :: Ref a → Enclave a
writeRef   :: Ref a → a → Enclave ()

-- get a reference to call a function inside the enclave
inEnclave :: Securable a ⇒ a → App (Secure a)

-- runs the Client monad
runClient :: Client () → App Done

-- used for function application on the enclave
gateway :: Binary a ⇒ Secure (Enclave a) → Client a
(<@>)   :: Binary a ⇒ Secure (a → b) → a → Secure b

-- call this from `main` to run the App monad
runApp  :: App a → IO a
```

Figure 1.4: The core HasTEE [53] API

Program partitioning through a "poor man's module system". Our program partitioning is accomplished by treating the API shown in Fig. 1.4 as analogous to a module signature [57] in the ML family of languages. There are two implementations for the corresponding signature: one for the program running on the enclave (captured as the `Enclave` monad) and the second for the program communicating with the enclave (the `Client` monad). Internally, a dispatch table is constructed within our DSL, mapping a function call from the untrusted side of the program to a function application inside the enclave. The

detailed meaning and implementation of the API are provided in Chapter 5.

Note that our implementation, inspired from *Haste*[58], is quite lightweight and does not require any advanced type-level or language-level features of Haskell. The main objective of the API is to facilitate sound program partitioning rather than introducing a module system, similar to those already existing in Haskell [59], [60].

Information Flow Control. The HasTEE API also incorporates a pragmatic implementation of information flow control [21], where all computations carried out in the *enclave* are considered highly confidential. Enforcing generalised non-interference [61] on such a model would disallow any form of communication with the enclave, as all computed results would also be confidential. We relax this constraint and allow data to flow out of the enclave, but through a very restricted API – namely, the function `gateway` (Fig. 1.4). There are a number of other guardrails provided to prevent accidental information leak such as the `RestrictedIO` monad and `Binary` typeclass constraint on data flowing across the *enclave*. The details can be found in Chapter 5.

Evaluations. Our evaluations were conducted through three sample applications. One of them – a case study on Federated Learning [62] – involves the use of TEEs and homomorphic encryption [63] to emulate a zero-trust data analytics setup. Our preliminary results show acceptable memory overheads but suffer from high latency and low throughput in communication with the enclave. The performance impact is attributed to our implementation strategy, wherein we utilise two GHC runtimes to facilitate communication between the client and the enclave, a topic discussed in further detail in Chapter 5.

One of the missing features in HasTEE is the absence of any integration with the remote attestation protocols supported by various TEEs. In our follow-up Paper II, we improve upon HasTEE by adopting an alternate threat model and addressing its other shortcomings.

Paper II

Paper II presents HasTEE$^+$ [64], a DSL that builds on top of HasTEE while adopting a different threat model. Figure 1.5 contrasts the two threat models.

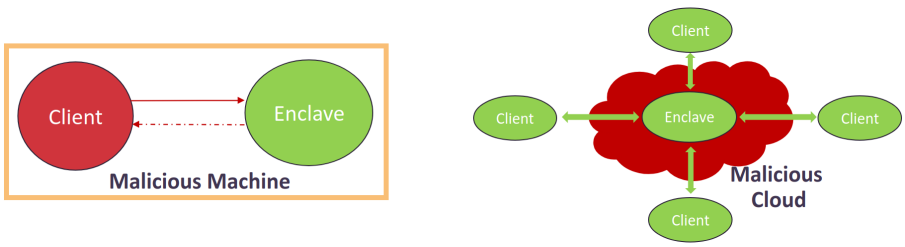


Figure 1.5: The HasTEE threat model on the left and the HasTEE$^+$ threat model on the right (not limited to four clients)

In Fig. 1.5 on the left, we observe that HasTEE's threat model closely aligns with a typical TEE threat model. In this scenario, there is a single

malicious machine, and an attacker with administrative access on the hypervisor, operating systems, and all other system software attempts to compromise the enclave, which is trusted. The red arrows indicate the input/output boundary, which is often maliciously exploited to craft spurious inputs that can compromise the enclave integrity [65].

In HasTEE$^+$, we extend the above threat model to represent multiple clients (Fig. 1.5 right). The program partitioning technique from HasTEE is generalised to allow a single program to represent multiple clients and an enclave. HasTEE$^+$ then employs multiple compilations to split the same program into several clients and one enclave program. The details of the implementation are found in Chapter 6.

Tierless DSL. This approach of using a single program to express the interactions of several clients and servers is known as *tierless programming* [66]. The automation of program partitioning in Haskell makes the entire process type-safe by capturing the types of all participants within Haskell's type system. This automation helps eliminate bugs that may arise from manual program partitioning and the adherence to complex data copying protocols.

Furthermore, this enables a transition from the HasTEE model, which uses the DSL as an SDK, to a model where separate trusted client and server programs are generated. Subsequently, the entire server program could be hosted within a TEE, aligning with the programming model of newer Intel TDX machines [40]. Further details can be found in Chapter 6.

Integrity with Remote Attestation. One of the key contributions of HasTEE$^+$ is the integration of a remote attestation infrastructure for verifying the enclave integrity. In Fig. 1.5, the arrows at the communication boundary are marked in green, as HasTEE$^+$ uses Intel's RA-TLS infrastructure [67] to establish a secure communication channel with the enclave. Furthermore, HasTEE$^+$'s design frees programmers from writing any form of cross-cutting code related to attestation protocols, which is very common in C/C++-based projects. HasTEE$^+$ also incorporates a digital signature-based scheme to verify client integrity, ensuring secure data inflow and outflow. The details are presented in Chapter 6.

Information Flow Control. HasTEE$^+$, as an improvement over HasTEE, integrates a sound and complete dynamic information flow control (IFC) mechanism inspired by the Labeled IO (LIO) monad [68]. HasTEE$^+$ introduces labeled values inside the enclave, enabling the mixing of both public and confidential data within the enclave. The basic IFC primitives include *tainting*, *labeling*, and *unlabeling* the data. Unlabeling data results in tainting the computational context, which constrains all other contexts in which the data can move while respecting generalised non-interference [61].

The labeling model adopted in HasTEE$^+$ is Disjunction Category labels [69], which itself is based on the well-known Myers-Liskov labeling model [70]. Our implementation additionally provides the client application with labeling and unlabeling primitives, as they are trusted. Declassification [71] is permitted using *privileges*, which resemble the concept of *capabilities* [72]. Further details are explained in Chapter 6.

A Confidential Data Sharing Pattern. The final contribution of HasTEE$^+$ is a data sharing design pattern that uses a combination of *privileges* or *capabilities* with standard public-key cryptography, enabling mutually distrusting parties to conduct data analytics.

Evaluations. For evaluations, the data sharing pattern discussed above is employed to model a *data clean room* [73]. We present microbenchmarks for measuring the performance overheads on the data clean room arising from the dynamic IFC mechanism, remote attestation, and client-integrity checks. The results indicate that the overheads are in the order of hundreds of milliseconds, which we suggest as acceptable overheads for security-critical software.

Multiple Enclaves. While the HasTEE$^+$ DSL accounts for multiple clients and a single enclave, it seems that there is a missing notion for *multiple* enclaves. Multiple enclaves typically refer to multiple sets of encrypted memory pages hosted in either (1) the same virtual address space, (2) different virtual address spaces, or (3) different physical address spaces.

Scenario (1) can be naturally expressed in HasTEE$^+$ using a combination of the `forkOS` and `runInBoundThread` functions from GHC's `Control.Concurrent` library [74]. Regarding scenarios (2) and (3), we deliberately decided against representing disjoint address spaces as distinct types in our DSL. This is because those scenarios are simply considered a special case of the standard client-server interaction already expressible in HasTEE$^+$.

The Question of Side channels. While HasTEE$^+$ presents itself as a viable solution for constructing secure software on the cloud, employing a combination of strongly-typed functional programming, Trusted Execution Environments, and language-based information flow control, a rising concern is the exploitation of side channels in hardware [75], [76]. In the work on HasTEE$^+$ and HasTEE, side channels were considered out of scope, as we focused on mitigating much easier-to-exploit software vulnerabilities compared to hardware side-channel attacks, which often require physical access to the hardware. Nevertheless, given the discovery of attacks like the *ÆPIC Leak* [77], which do not rely on noisy side-channels, we recognise the significant challenge of addressing side channels. Accordingly, we discuss possible future work on extending HasTEE$^+$ to counter such attacks in Chapter 3.

1.5 Paper III: Synchron

Paper III presents Synchron [78], a functional language runtime, and corresponding programming interface targeted towards embedded systems. Synchron is a specialised runtime API designed for expressing **(i) I/O-bound, (ii) concurrent and (iii) timing-aware programs**. The architecture of Synchron consists of three parts -

- Runtime - The principal component of Synchron is a specialised runtime consisting of nine built-in operations and a scheduler. The runtime allows the creation of concurrent user-level processes (green threads) and provides operators for declaratively expressing interactions between

the software processes and hardware interrupts. The power-efficient scheduling of the processes is managed by the Synchron scheduler.

- Low-level Bridge - The Synchron runtime interacts with the various hardware drivers through a low-level *bridge* interface. The interface is general enough such that it can be implemented by both synchronous drivers (like LED) as well as asynchronous drivers (like UART).

- Underlying OS - The Synchron runtime is run atop an underlying RTOS such as ZephyrOS or ChibiOS. The OS supplies the actual hardware drivers that implement the low-level bridge interface described above. We have designed our runtime interfaces in a modular fashion such that other operating systems, such as FreeRTOS, can be easily plugged in.

Our implementation of Synchron is in the form of a bytecode-interpreted virtual machine called the SynchronVM. The execution engine of SynchronVM is based on the Categorical Abstract Machine [79] (of Caml [80]), which supports the cheap creation of closures to support functional programming languages. Fig. 1.6 below provides a graphical description of the architecture of Synchron.

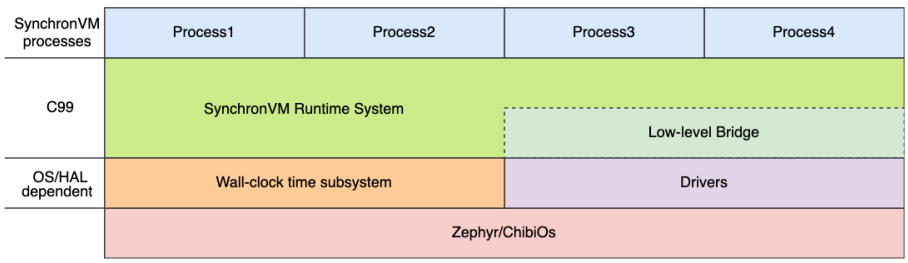


Figure 1.6: Architecture of Synchron

The Synchron API

The core API of Synchron consists of nine functions, which can be embedded within a standard *call-by-value* functional language. In Fig. 1.7, we present the complete API using the syntax of a call-by-value functional language, serving as the frontend for programming with Synchron. Syntactically, the language resembles Haskell but is semantically closer to Caml [80]. We emphasise that we use the type signature for the ease of exposition; however, the underlying virtual machine is untyped.

Concurrency. The Synchron API is built on Concurrent ML (CML) [81], a *synchronous message-passing–based concurrency model*. CML's key distinction from predecessors like Hoare's *communicating sequential processes* [82] is the separation between the *intent* and *act* of communication. This separation is captured by first-class values called *Events*.

An *event* is an abstraction to represent deferred communication. In contrast with a rudimentary protocol involving single message sends and receives, the

```
spawn    : (() → ()) → ThreadId
channel  : () → Channel a
send     : Channel a → a
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
```

Figure 1.7: The complete Synchron API

CML combinators such as `wrap` and `choose` can compose elaborate communication protocols involving multiple *sends* and *receives*.

Timing. Synchron's extension of the CML API to include the notion of timing is from the function `syncT` (Fig. 1.7). The `syncT` operation allows a programmer to specify the exact *timing window* at which an *event* synchronisation should happen. The first argument to `syncT` represents the baseline of the operation, while the second argument is the deadline. The `syncT` operator provides an opportunity to *dynamically prioritise* concurrent timed processes instead of static-priority APIs provided by typical RTOSes.

I/O. To unify the notions of I/O and concurrency, Synchron introduces the `spawnExternal` operator. The `spawnExternal` operator models the external hardware drivers as processes themselves. Modelling the drivers as processes allow programmers to apply the entire message-passing API to low-level drivers interactions such as *interrupt-handling*. The serialisation and deserialisation between software messages and hardware interrupts are handled by the runtime.

The design and implementation of SynchronVM are detailed in Chapter 7. Note that we previously introduced a subset of the API, focusing on concurrency and I/O aspects, elsewhere [83]. However, Chapter 7 provides a comprehensive explanation of the complete API.

Evaluations. Our evaluations were carried out on the NRF52840DK [84] and the STM32F4 Discovery [85] microcontroller boards with the help of a musical application, which involves some soft real-time components. Other micro-benchmarks were carried out on response times, memory usage, interpreter-overhead, and power consumption.

Our preliminary results are encouraging and show that in terms of power usage, a program running on SynchronVM has the same amount of momentary power consumption as a C program written using callback registration. Indeed, when considering integrated power usage over time, a C program tends to be more power-efficient. However, the trade-off of programming with high-level abstractions is an attractive proposition.

In terms of memory usage, a SynchronVM program occupies tens to hundreds of kilobytes, which is beneficial for memory constrained microcontrollers. The response times of our benchmarks are typically 2-3x times longer than the

C equivalents. A point to be noted here is that our execution engine is based on the categorical abstract machine, which is known to be four times slower, on average, than the Zinc abstract machine [86] (that underlies OCaml [87]).

For memory management, we use a stop-the-world, non-moving, mark-and-sweep garbage collector that employs pointer-reversal-based marking, suitable for memory-constrained embedded systems. Additionally, we employ an aggressive peephole optimization that attempts to reduce the size of the code to fit in the flash memory of microcontrollers.

A natural extension of this work would be to embed a HasTEE$^+$–style DSL in Synchron's frontend functional language and use ARM TrustZone's extensions for microcontrollers [88]. This is considered as potential future work. Additionally, in Chapter 3, we discuss our future plans to port SynchronVM to experimental memory-tagging architectures, such as CHERI [89].

1.6 Paper IV: Hailstorm

Paper IV introduces Hailstorm, a *functional reactive* programming language designed for embedded systems [90]. Chronologically, Hailstorm is the first published paper in this dissertation, preceding the work on SynchronVM. Hailstorm primarily aims to tackle the challenge of designing a programming language for I/O-intensive embedded systems applications. In a pure functional language like Haskell, these applications would reduce to a giant I/O monad capturing all external interactions.

One alternate approach to expressing such applications is Functional Reactive Programming (FRP) [91]; however, in practice, if embedded[3] in a language like Haskell, such applications would inevitably interact with the external world through monadic I/O. The problem is also alluded to by Conal Elliott, the inventor of FRP, in a blog post [92]:

> *...imperative computation still plays a significant role in most Haskell programs. Although monadic IO is wily enough to keep a good chunk of purity in our model, Haskell programmers still use the imperative model as well and therefore have to cope with the same fundamental difficulties that Backus told us about. In a sense, we're essentially still programming in Fortran,...*

Hailstorm's key contribution is introducing a purely functional programming language that incorporates side effects without relying on monads. Instead, it opts to integrate FRP into the core I/O semantics of the language. It uses the Arrowized FRP [93] formulation of FRP. The most central type in the language is that of a signal function, $SF\ a\ b$, where a and b denote polymorphic type variables. Signal functions are representations of a dataflow from type a to b.

We further extended this representation with the concept of a resource type [94]. A resource type is type-level label that can be used to uniquely identify various external resources. The new type of a signal function becomes $SF\ r\ a\ b$,

[3] refers to the *embedding* of a language, not to be confused with embedded systems

where r denote a polymorphic resource label. For instance, two sensors that can supply an *Int* and *Float* value type respectively, will have the following types in Hailstorm -

```
resource S1
resource S2

sensor1 :: SF S1 () Int
sensor2 :: SF S2 () Float
```

The unit type - () - above indicates that the sensor interacts with the external world. Hailstorm provides a family of combinators (Fig. 1.8) to declaratively compose the data flowing through the various signal functions.

```
mapSignal# : (a → b) → SF Empty a b
(>>>) : SF r1 a b → SF r2 b c → SF (r1 ∪ r2) a c
(&&&) : SF r1 a b → SF r2 a c → SF (r1 ∪ r2) a (b, c)
(***) : SF r1 a b → SF r2 c d → SF (r1 ∪ r2) (a, c) (b, d)
```

Figure 1.8: The key Hailstorm operators

The technical details about the type-level union and its semantics are described in the Chapter 8. As discussed earlier, FRP models are often embedded within a host language, making any form of interaction with the external world syntactically awkward. The introduction of resource types is done to resolve this issue, and we detail, using examples, in Chapter 8 on how a resource label can allow the *correct* composition of signal functions.

Evaluations. The Hailstorm language has an LLVM and an Erlang backend. The Erlang backend, in particular, was used to prototype experiments on the GRiSP microcontroller boards. The evaluations consisted of writing very small prototype applications in Hailstorm like a watchdog process, a simplified traffic light system and a railway level-crossing simulator.

We also carried out micro-benchmarks on the memory consumption and response time of the programs. The memory footprint of the Hailstorm programs was in the order of 2-3 MB, owing to the size of the Erlang runtime. The response time of the programs was in the range of 100-150 microseconds.

As the work on Hailstorm preceded Synchron [78], the paper uses an Erlang-based runtime. Nevertheless, the language is more naturally suited for hosting on a memory-efficient, embedded systems runtime such as Synchron.

1.7 Discussion

Revisiting our thesis statement, our goal was to use a functional programming foundation to build safer and secure digital systems. We distinguish between two classes of digital systems: cloud and embedded systems. Built upon functional programming abstractions, we employ hardware security extensions

and language-based security techniques to secure the cloud. Simultaneously, we propose concurrency, I/O, and timing primitives to enhance the safety of embedded systems.

Figure 1.9 visually illustrates how the contributions from these four papers come together to construct a secure software system.

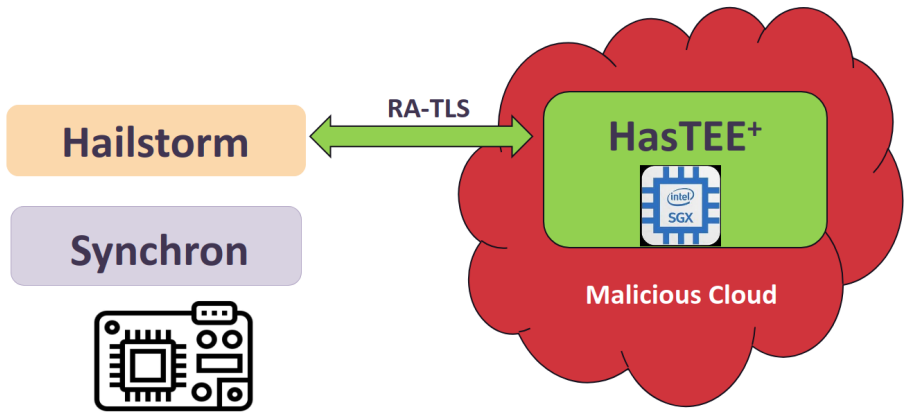


Figure 1.9: A secure digital system using HasTEE$^+$, Synchron and Hailstorm

In the figure above, the embedded systems component benefits from the declarative, memory-safe, and type-safe features of Hailstorm and Synchron. At the same time, the cloud-based software can be hosted on a potentially malicious cloud, with security guarantees ensured by HasTEE$^+$. Furthermore, the integration of our remote attestation framework with the communication protocol (TLS) ensures that we can establish secure communication channels, guaranteeing the integrity of both the client and server. In the following chapter, we offer our reflections on the role of Functional Programming in this dissertation and in the construction of secure systems in general.

Chapter 2

Reflections

This dissertation is a compilation of four papers in which we use functional programming abstractions to address a variety of problems. Reflecting on some of the more important challenges:

1. Both HasTEE [53] and HasTEE$^+$ [64] have to deal with the problem of *program partitioning* for security.

2. Both libraries, HasTEE$^+$ in particular, needs to enforce forms of *Information Flow Control (IFC)* mechanisms.

3. To provide a highly automated and type-safe programming model for TEEs, HasTEE$^+$ had to implement a multi-tier programming model, which we call *tierless programming*.

4. To address the inherent concurrency of microcontrollers, both Synchron [78] and Hailstorm [90] need to implement *structured concurrency* primitives. *Structured concurrency* [95], a general term[1] for concurrency primitives ensuring well-defined scope and structured lifetime for concurrent tasks, is implemented by the green threads in Synchron and implicitly integrated into Hailstorm's FRP primitives.

5. Synchron has to implement *temporal programming* primitives to express timing-aware computations.

6. Finally, Hailstorm needs to perform *resource tracking* to express composable I/O operations among various sensors within the FRP paradigm.

To address the above challenges, we employ various techniques and abstractions from what we refer to as the *Functional Programming Toolkit*.

[1] coined by Martin Sústrik, the author of the popular ZeroMQ library [96]

2.1 The Functional Programming Toolkit

The Functional Programming Toolkit is a metaphorical "toolkit" that offers a wide array of techniques and abstractions from a functional programming language, which we have employed for solving problems during various stages of writing this dissertation. Figure 2.1 visually illustrates the above discussed challenges (highlighted in blue) and showcases some of the tools (highlighted in green) from the toolkit that we employed to address these diverse problems.

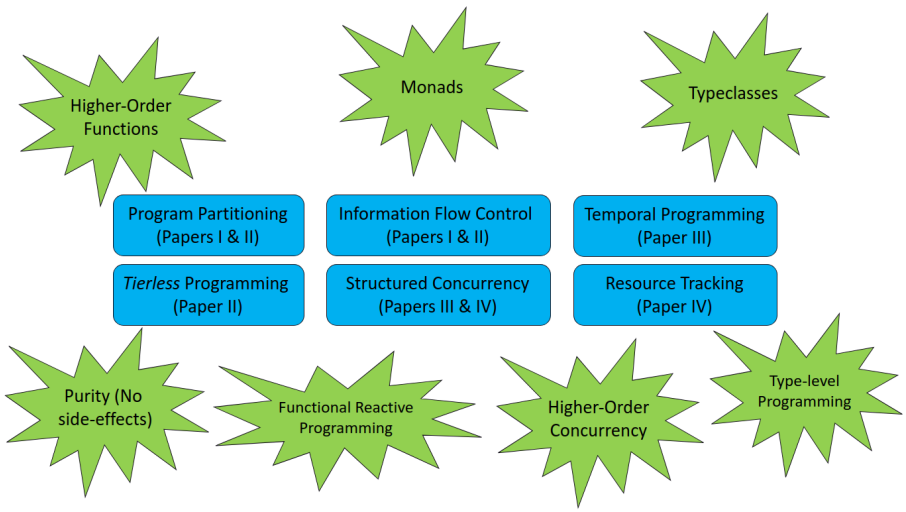


Figure 2.1: Some essential tools (green) from the *Functional Programming Toolkit* employed in this thesis and their corresponding applications (blue).

Monads. The notion of a monadic structure is employed to enforce information flow control. The `return :: a -> m a` operator can be considered as tainting a value, and the underlying monadic computation then tracks the information flow within the specified monad (Chapter 6).

For *tierless programming*, a monadic computation tracks the correct client and server, dispatching the desired computation accordingly (Chapter 6)

For program partitioning, the entire API is expressed as a combination of three monads, where the monadic type marks the location where the computation is executed (Chapters 5 and 6).

Typeclasses. Typeclasses are used to emulate remote procedure calls and *variadic* functions for a partitioned program in Chapter 5. They are extensively used in implementing disjunction category labels [69] for IFC in Chapter 6 .

Purity. Purity is an important concept in IFC, where a *pure* function can be considered *confined* by default [97]. Consequently, HasTEE$^+$ relies on the purity of Haskell functions to enforce non-interference.

Higher-Order Concurrency. Originally proposed by John Reppy [81], higher-order concurrency naturally integrates with functional languages to represent structured concurrency primitives. Our extensions in Synchron additionally allow expressing temporal programming operations in this model.

Functional Reactive Programming (FRP). FRP, in combination with resource-types [94] enable resource tracking in Hailstorm (Chapter 8).

Higher-Order Functions. Higher-Order Functions are pervasive throughout every aspect of this dissertation, with applications too numerous to enumerate. For instance, the program partitioning and tierless programming mechanisms in both HasTEE and HasTEE$^+$ rely on higher-order functions to *lift* them to the enclave and generate the internal dispatch table identifying the location of each function.

Type-level Programming. We deliberately use a limited amount of type-level programming, as it can often impact the type-inference capabilities of Hindley-Milner-style type systems [98]. In HasTEE$^+$, type-level strings are used for distinguishing between various client and servers. Also in Hailstorm, a resource type serves as an example of very simple set-theoretic operations implemented at the type-level for resource tracking.

2.2 Practicality

Reflecting on the practicality of our contributions, the HasTEE$^+$ DSL (along with its predecessor HasTEE) is implemented as a Haskell library. Along with the DSL, even the Information Flow Control aspect of HasTEE$^+$ is part of the same Haskell library.

The advantage of adopting this library-based approach is that programmers can readily access a simple and secure DSL for specialised hardware without relying on dedicated languages such as GoTEE [99] for TEE programming or Jif [100] and Flow Caml [101] for information flow control. Furthermore, the entire Haskell ecosystem is available to the programmer, as a HasTEE$^+$ program is simply a Haskell program, without any specialised language extension.

In the chronological order of publication, our earlier works on Hailstorm and Synchron represent more ground-up redesigns, which we consider essential given the propensity for memory-safety and type-safety vulnerabilities associated with C in embedded systems. Since the publication of Synchron [78], we have incorporated a foreign-function interface in Synchron to interact with C/C++ libraries, thereby opening the door to the broader embedded systems ecosystem. However, the memory and type safety guarantees do not extend to the C/C++ libraries. One possible approach to address this limitation is software *compartmentalization*, a topic we discuss as future work in Chapter 3.

2.3 Reasoning with Functional Programming

One of the benefits of functional programming is that it allows equational reasoning even in the presence of side effects [14]. As the HasTEE$^+$ library implements enclave computation within a monad, the well-known *left-identity*, *right-identity*, and *associativity* monad laws [102] apply to the *Enclave* monad.

Regarding security properties, the Information Flow Control component of HasTEE$^+$ is based on the Labeled IO monad, for which Stefan et al. have

already proven *non-interference* and *containment* (asserting that certain pieces of code cannot manipulate or have access to specific data) [68].

Reasoning becomes more challenging with Synchron as it is implemented as an impure virtual machine rather than a pure programming language. Nevertheless, the underlying API is an extension of Hoare's Communicating Sequential Processes [82], which has a substantial body of work dedicated to developing a mathematical theory [103].

Hailstorm, on the other hand, is a pure functional programming language based on Arrowized FRP formulation. Consequently, when equationally reasoning with Hailstorm, one can apply the standard monad laws as well as the additional Kleisli arrow laws [104].

What about Formal Verification? Given our aim to provide strong system security guarantees, the question naturally arises: Is it feasible to formally verify our assurances? Considering type safety, a sound [105] and complete [106] Hindley-Milner type system is able to guarantee the type safety of a program. In terms of memory safety, the memory management in our contributions is automatic, ensuring memory-safety by default.

The security properties requiring formal verification include non-interference [61]. The non-interference property of security monads and their corresponding Haskell libraries has been mechanically verified [107].

Such formal proofs often involve mechanising a *core calculus* and proving non-interference in the somewhat idealised core language. If mechanising the proof for a significantly larger, practical language were to be undertaken, it would require mechanised semantics for the entire language. However, with the exception of Standard ML [108], very few feature-rich languages have such comprehensive mechanised semantics. As a result, there are possibilities of errors creeping in the actual implementation of the library.

One of the possible solutions is writing the entire library in a proof assistant and then using the program extraction feature to generate the actual library. Such approaches often require quite elaborate proof techniques and substantial proof-engineering effort [109].

An alternative to formal verification could involve using QuickCheck [20] to encode non-interference as a property and applying property-based testing on the actual library implementation. As HasTEE$^+$ enables the execution of Haskell programs within the enclave, we consider this a potentially fruitful avenue for future research and discuss it further in Chapter 3.

2.4 Applicability beyond Functional Languages

Our final reflection is on the applicability of our contributions beyond functional programming languages. Before discussing imperative languages, if we consider other statically-typed functional programming languages like OCaml and SML, our contribution could be adopted, subject to mimicking *higher-kinded types* like monads. Similarly, while typeclasses are quite central to our implementation, the same could be implemented using ML modules.

Beyond functional languages, the natural beneficiary from the contributions

made in this dissertation would be a language like Rust [110]. Particularly in embedded systems, Rust is touted to be an eventual replacement for C. Though Rust does not intrinsically contain structured concurrency primitives, there are experimental crates [111] that leverage the basic concurrency constructs of Rust to implement various structured concurrency libraries. Rust also lacks any temporal programming operations, so the Synchron API could be something that could be naturally adopted as a Rust library.

With respect to the HasTEE$^+$ line of work, Rust, like OCaml, does not natively support higher-kinded types like monads. However, through clever usage of Rust traits, many of the core functionalities in our contributions could be implemented. There also exists a Rust SDK [112] for programming Intel SGX enclaves, so it would be quite natural to adopt the program partitioning technique of HasTEE$^+$ and use it to simplify TEE programming in Rust.

Chapter 3

Future Work

3.1 Property-based Testing for TEEs

As discussed earlier, the capability of running Haskell programs in the enclave opens up the possibility of applying QuickCheck's property-based testing to Trusted Execution Environments. There are two levels at which QuickCheck can be applied:

- Fuzz Testing. Tools such as SGXFuzz [113] and TEEzz [114] have successfully employed *fuzzing*-based approaches to discover bugs and vulnerabilities in Intel SGX and ARM TrustZone, respectively. The bugs are typically found by fuzzing the API at the enclave boundary. QuickCheck can be analogously applied as a fuzzer (as illustrated in QuickFuzz [115]) to catch vulnerabilities at the enclave boundary.
- Testing Non-interference - The more interesting and unique use case of QuickCheck could be to test if the HasTEE$^+$ library when executing a monadic computation on an enclave obeys non-interference. Unlike traditional properties, non-interference is a hyperproperty [116], meaning it is a property applied to a set of traces rather than a single trace. While this poses a challenge to express in QuickCheck, Hritcu et al. [117] have presented initial work on testing non-interference for a stack machine written in Haskell. The next challenge would be to generalise this approach for generating polymorphic HasTEE$^+$ programs, thus paving the way to employ QuickCheck for testing generalised non-interference.

3.2 Beyond Binary Attestation

Attestation, one of the core components of Confidential Computing, is concerned with proving the identity of the software (and hardware) running on a TEE to a challenging remote party. Both Intel SGX and ARM TrustZone toolchains currently rely on something called *binary attestation*, where the identity of the software essentially consists of a cryptographic hash of the software's build

artifact. One of the key limitations of binary attestation is that it provides the remote party with confidence solely in the enclave's initial state, disregarding the dynamically changing state of the program. It fails to account for the fundamental fact that software's behaviour and state, particularly in the case of long-running servers, undergoes dynamic changes throughout its runtime – something not captured by a binary hash. Moreover, the use of a static binary hash as an identity also increases the probability of *fingerprinting attacks* [118].

A more natural way to identify software could involve behavioral or property-based attestation, where the software is identified by the satisfaction of functional and security properties, rather than relying on a binary hash. The idea of property-based attestation [119] has been around since the introduction of the Trusted Platform Module [120] but deserves to be explored in the context of TEEs. An important question that needs answering is: What kind of properties best identify trusted software?

In line with our earlier discussion on property-based testing, the QuickCheck properties of the trusted software could serve as the identifier in property-based attestation. Research is needed on the infrastructure required to integrate dynamic monitoring of the software and check if it satisfies the specified properties. A potential solution would likely involve integrating a runtime monitor with HasTEE$^+$ programs, drawing parallels to related work on software privacy monitoring using runtime monitors [121].

3.3 Side-channel Attacks

There is a growing body of work on side-channel attacks against Intel SGX enclaves [122], where the speculative execution capabilities of modern x86 machines are exploited to extract secrets through timing [76], voltage [123], and other side channels. The long-term solution to these attacks would involve migrating security-critical software to computer architectures with specialised extensions [124] or to newer processor designs [125].

A more short-term solution would involve integrating software-based techniques for side-channel mitigation into HasTEE$^+$. The general strategy involves disallowing (1) branching on secrets, (2) specialized memory-access patterns depending on secrets, and (3) early termination of loops or procedures depending on secrets. In connection with this, Agat [126] proposed a simple type system, allowing statements to branch on secrets only if the branches exhibit the same memory access pattern. More recently, DSLs like FaCT [127] introduced secrecy type systems and compiler transformations to enforce the aforementioned strategies. These approaches could be implemented and experimented with using GHC compiler plugins [128]. Additionally, further research on the impact of laziness on side-channel attacks is crucial.

3.4 SynchronVM on CHERIoT

As noted earlier, the addition of the C/C++ foreign function interface to SynchronVM enables tapping into the larger embedded systems ecosystem.

However, correspondingly, the memory and type safety of the software are compromised, and even the type-safe Synchron programs themselves become vulnerable to classic memory-unsafety-based attacks [48].

A solution to this would be porting Synchron to a new architectural extension to RISC-V called CHERIoT [129]. Underneath, CHERIoT employs a capability-based system that uses a collection of techniques to perform bound-checks on memory-unsafe libraries and prevent *use-after-free* vulnerabilities, effectively compartmentalising the memory-unsafe code. Challenges would mainly involve adapting the C99-based SynchronVM code to the CHERIoT toolchain, which features a different pointer structure and introduces new pointer types compared to C99.

3.5 Conclusion

We acknowledge that security is often a moving target, and achieving a *perfectly* secure platform is a goal that demands further research. As such, our hope is that the contributions made through this dissertation would open further avenues for research, where the strong static guarantees provided by a language like Haskell can be married with newer hardware and software-based security techniques to enable the construction of a safe and secure digital system.

The programming artifacts of HasTEE, HasTEE$^+$, Synchron, and Hailstorm have all been made publicly available and are liberally licensed (MIT and BSD licenses). We hope that subsequent research will build upon these artifacts and our suggested lines of future work, enhancing our current security guarantees and moving closer to the elusive goal of a perfectly secure software system.

Chapter 4

Statement of Contributions

Paper I. HasTEE: Programming Trusted Execution Environments with Haskell

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.

Statement of Contributions

I implemented the trusted GHC runtime, developed the cabal-based library for two-project partitioning, and created the overall compiler and runtime toolchain. The idea for using the Haste-based approach for program partitioning came from Koen. The ideas on the information flow control mechanisms came from Alejandro and were implemented by Robert and myself.

For the federated learning example, the entire Paillier-based homomorphic encryption library support for IEEE-754 floating-point numbers was developed by me, and I officially maintain the Paillier library on Hackage. The server side of the federated learning example was written by me. Robert wrote the client side of the federated learning example, as well as the password wallet example and the differentially private data clean room example.

For the paper, Sections 1 to 4.3 and Section 5.1 were written by me, including the formulation of the two-memory-cell-based operational semantics. Section 4.4 was written by Alejandro. Sections 5.2, 5.3, 6.1, 8 and 9 were written in collaboration between myself and Robert. Sections 6.2 and 6.3 were written by Robert. Alejandro made several additions to the related work on IFC in Section 8.

The complete evaluations in Section 7 on the Azure machine were done by me, and the writing for Section 7 was also done by me. Alejandro made insightful comments, edits, and enhancements to the overall paper. Koen also provided feedback on the paper.

Paper II. HasTEE$^+$: Confidential Computing and Analytics with Haskell

Abstract

Confidential computing is a security paradigm that enables the protection of confidential code and data in a co-tenanted cloud deployment using specialised 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.

Statement of Contributions

The design of the tierless DSL was done in collaboration between myself and Alejandro. The implementation was done by myself. The entire remote attestation framework was designed and implemented by myself. The design of the information flow control system was inspired by LIO and COWL, whose design Alejandro asked me to study. The implementation was done by myself. The data clean room example was proposed by Alejandro, and the design was a collaboration between Alejandro and myself. The implementation was done by myself. The evaluations were also done by myself.

The entire paper was written by myself with feedback from Alejandro on the final draft.

Paper III. Synchron - An API and Runtime for Embedded Systems

Abstract

Programming embedded applications involves writing concurrent, event-driven and timing-aware programs. Traditionally, such programs are written in 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.

Statement of Contributions

I am responsible for the development and maintenance of the virtual machine discussed in the paper. I conceived the core research idea presented in the paper and designed and implemented the middleware, optimiser, assembler,

bytecode interpreter, and substantial portions of the runtime. Additionally, I proposed the timing API and implemented its core components within the runtime. Bo Joel Svensson contributed by writing the low-level bridge and timing subsystem, as well as the garbage collector, and collaborated with me on various crucial design decisions within the runtime.

I wrote the paper in collaboration with Bo Joel Svensson. Several edits and enhancements were proposed by Mary Sheeran. The experiments presented in the paper were conducted by myself and Bo Joel Svensson.

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

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.

Statement of Contributions

I was responsible for conceiving the research idea and implementing the compiler presented in the paper. I authored all the major sections of the paper, with Mary Sheeran providing suggestions for the paper's structure, making edits, and contributing final enhancements.

Bibliography

[1] R. Langner, 'Stuxnet: Dissecting a cyberwarfare weapon,' *IEEE Secur. Priv.*, vol. 9, no. 3, pp. 49–51, 2011. DOI: 10.1109/MSP.2011.67. [Online]. Available: https://doi.org/10.1109/MSP.2011.67 (cit. on pp. 3, 4).

[2] NIST. 'National Vulnerability Database CVE-2010-2568.' (2010), [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2010-2568. (accessed: 18.01.2024) (cit. on p. 3).

[3] NIST. 'National Vulnerability Database CVE-2010-2772.' (2010), [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2010-2772. (accessed: 18.01.2024) (cit. on p. 3).

[4] T. Moore, 'On the harms arising from the equifax data breach of 2017,' *Int. J. Crit. Infrastructure Prot.*, vol. 19, pp. 47–48, 2017. DOI: 10.1016/J.IJCIP.2017.10.004. [Online]. Available: https://doi.org/10.1016/j.ijcip.2017.10.004 (cit. on p. 3).

[5] NIST. 'National Vulnerability Database CVE-2017-5638.' (2017), [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-5638. (accessed: 18.01.2024) (cit. on p. 3).

[6] NIST. 'National Vulnerability Database CVE-2017-0143.' (2017), [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-0143. (accessed: 18.01.2024) (cit. on p. 3).

[7] Z. Durumeric, J. Kasten, D. Adrian *et al.*, 'The Matter of Heartbleed,' in *Proceedings of the 2014 Internet Measurement Conference, IMC 2014, Vancouver, BC, Canada, November 5-7, 2014*, C. Williamson, A. Akella and N. Taft, Eds., ACM, 2014, pp. 475–488. DOI: 10.1145/2663716.2663755. [Online]. Available: https://doi.org/10.1145/2663716.2663755 (cit. on p. 3).

[8] R. Hiesgen, M. Nawrocki, T. C. Schmidt and M. Wählisch, 'The race to the vulnerable: Measuring the log4j shell incident,' in *6th Network Traffic Measurement and Analysis Conference, TMA 2022, Enschede, The Netherlands, June 27-30, 2022*, R. Ensafi, A. Lutu, A. Sperotto and R. van Rijswijk-Deij, Eds., IFIP, 2022. [Online]. Available: https://dl.ifip.org/db/conf/tma/tma2022/tma2022-paper40.pdf (cit. on p. 3).

[9] Microsoft. 'Microsoft: 70 percent of all security bugs are memory safety issues.' (2019), [Online]. Available: https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/. (accessed: 18.01.2024) (cit. on p. 3).

[10] Google. 'Chrome: 70 percent of all security bugs are memory safety issues.' (2020), [Online]. Available: https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues/. (accessed: 18.01.2024) (cit. on p. 3).

[11] M. C. O. M. I. Board, *Mars climate orbiter mishap investigation board: Phase I report*. Jet Propulsion Laboratory, 1999 (cit. on p. 4).

[12] D. Bernstein, 'Containers and Cloud: From LXC to Docker to Kubernetes,' *IEEE Cloud Comput.*, vol. 1, no. 3, pp. 81–84, 2014. DOI: 10.1109/MCC.2014.51. [Online]. Available: https://doi.org/10.1109/MCC.2014.51 (cit. on p. 4).

[13] P. Wadler, 'A critique of abelson and sussman or why calculating is better than scheming,' *ACM SIGPLAN Notices*, vol. 22, no. 3, pp. 83–94, 1987. DOI: 10.1145/24697.24706. [Online]. Available: https://doi.org/10.1145/24697.24706 (cit. on p. 4).

[14] J. Gibbons and R. Hinze, 'Just do it: Simple monadic equational reasoning,' in *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, M. M. T. Chakravarty, Z. Hu and O. Danvy, Eds., ACM, 2011, pp. 2–14. DOI: 10.1145/2034773.2034777. [Online]. Available: https://doi.org/10.1145/2034773.2034777 (cit. on pp. 4, 23).

[15] R. S. Bird and O. de Moor, *Algebra of programming* (Prentice Hall International series in computer science). Prentice Hall, 1997, ISBN: 978-0-13-507245-5 (cit. on p. 4).

[16] N. A. Danielsson, J. Hughes, P. Jansson and J. Gibbons, 'Fast and loose reasoning is morally correct,' in *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. G. Morrisett and S. L. P. Jones, Eds., ACM, 2006, pp. 206–217. DOI: 10.1145/1111037.1111056. [Online]. Available: https://doi.org/10.1145/1111037.1111056 (cit. on p. 4).

[17] M. Noonan, 'Ghosts of departed proofs (functional pearl),' in *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*, N. Wu, Ed., ACM, 2018, pp. 119–131. DOI: 10.1145/3242744.3242755. [Online]. Available: https://doi.org/10.1145/3242744.3242755 (cit. on p. 5).

[18] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis and S. L. P. Jones, 'Refinement types for haskell,' in *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, J. Jeuring and M. M. T. Chakravarty, Eds., ACM, 2014, pp. 269–282. DOI: 10.1145/2628136.2628161. [Online]. Available: https://doi.org/10.1145/2628136.2628161 (cit. on p. 5).

[19] C. A. R. Hoare, 'An Axiomatic Basis for Computer Programming,' *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969. DOI: 10.1145/363235.363259. [Online]. Available: https://doi.org/10.1145/363235.363259 (cit. on p. 5).

[20] K. Claessen and J. Hughes, 'Quickcheck: A lightweight tool for random testing of haskell programs,' in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, M. Odersky and P. Wadler, Eds., ACM, 2000, pp. 268–279. DOI: 10.1145/351240.351266. [Online]. Available: https://doi.org/10.1145/351240.351266 (cit. on pp. 5, 24).

[21] A. Sabelfeld and A. C. Myers, 'Language-based information-flow security,' *IEEE J. Sel. Areas Commun.*, vol. 21, no. 1, pp. 5–19, 2003. DOI: 10.1109/JSAC.2002.806121. [Online]. Available: https://doi.org/10.1109/JSAC.2002.806121 (cit. on pp. 5, 7, 8, 13).

[22] F. B. Schneider, 'Enforceable security policies,' *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 1, pp. 30–50, 2000. DOI: 10.1145/353323.353382. [Online]. Available: https://doi.org/10.1145/353323.353382 (cit. on p. 5).

[23] D. A. Basin, V. Jugé, F. Klaedtke and E. Zalinescu, 'Enforceable security policies revisited,' *ACM Trans. Inf. Syst. Secur.*, vol. 16, no. 1, p. 3, 2013. DOI: 10.1145/2487222.2487225. [Online]. Available: https://doi.org/10.1145/2487222.2487225 (cit. on p. 5).

[24] A. Martin, 'The ten-page introduction to Trusted Computing,' 2008 (cit. on p. 6).

[25] D. E. Bell, L. J. LaPadula *et al.*, *Secure computer systems: Mathematical foundations*. National Technical Information Service, 1989 (cit. on p. 6).

[26] Amazon. 'Amazon Web Services.' (2006), [Online]. Available: https://aws.amazon.com/. (accessed: 18.01.2024) (cit. on p. 6).

[27] Microsoft. 'Microsoft Azure.' (2008), [Online]. Available: https://azure.microsoft.com/en-us/. (accessed: 18.01.2024) (cit. on p. 6).

[28] Google. 'Google Cloud Platform.' (2008), [Online]. Available: https://cloud.google.com. (accessed: 18.01.2024) (cit. on p. 6).

[29] B. Tak, B. Urgaonkar and A. Sivasubramaniam, 'To Move or Not to Move: The Economics of Cloud Computing,' in *3rd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'11, Portland, OR, USA, June 14-15, 2011*, I. Stoica and J. Wilkes, Eds., USENIX Association, 2011. [Online]. Available: https://www.usenix.org/conference/hotcloud11/move-or-not-move-economics-cloud-computing (cit. on p. 6).

[30] L. Adrien and D. Fenandez. 'Hyper-V bug that could crash 'big portions of Azure cloud infrastructure': Code published.' (2021), [Online]. Available: https://www.theregister.com/2021/06/02/hyperv_bug_that_until_recently/. (accessed: 18.01.2024) (cit. on p. 6).

[31] Xen. 'Improper MSR range used for x2APIC emulation.' (2014), [Online]. Available: http://xenbits.xen.org/xsa/advisory-108.html. (accessed: 18.01.2024) (cit. on p. 6).

[32] NIST. 'The VENOM Hyperjacking Vulnerability.' (2015), [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2015-3456. (accessed: 18.01.2024) (cit. on p. 6).

[33] A. Marvi, J. Koppen, T. Ahmed and J. Lepore. 'Bad VIB(E)s Part One: Investigating Novel Malware Persistence Within ESXi Hypervisors.' (2022), [Online]. Available: https://www.mandiant.com/resources/blog/esxi-hypervisors-malware-persistence. (accessed: 18.01.2024) (cit. on p. 6).

[34] Qualys. 'The GHOST Vulnerability.' (2015), [Online]. Available: https://blog.qualys.com/vulnerabilities-threat-research/2015/01/27/the-ghost-vulnerability. (accessed: 18.01.2024) (cit. on p. 6).

[35] P. Oester. '"Most serious" Linux privilege-escalation bug ever is under active exploit.' (2016), [Online]. Available: https://arstechnica.com/information-technology/2016/10/most-serious-linux-privilege-escalation-bug-ever-is-under-active-exploit/. (accessed: 18.01.2024) (cit. on p. 6).

[36] D. P. Mulligan, G. Petri, N. Spinale, G. Stockwell and H. J. M. Vincent, 'Confidential Computing - a brave new world,' in *2021 International Symposium on Secure and Private Execution Environment Design (SEED), Washington, DC, USA, September 20-21, 2021*, IEEE, 2021, pp. 132–138. DOI: 10.1109/SEED51797.2021.00025. [Online]. Available: https://doi.org/10.1109/SEED51797.2021.00025 (cit. on p. 6).

[37] Intel. 'Intel Software Guard Extension.' (2015), [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html. (accessed: 18.01.2024) (cit. on p. 7).

[38] ARM. 'ARM TrustZone.' (2004), [Online]. Available: https://www.arm.com/technologies/trustzone-for-cortex-a. (accessed: 18.01.2024) (cit. on p. 7).

[39] AMD. 'AMD Secure Encrypted Virtualization.' (2017), [Online]. Available: https://www.amd.com/en/developer/sev.html. (accessed: 18.01.2024) (cit. on p. 7).

[40] Intel. 'Intel Trust Domain Extension.' (2021), [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html. (accessed: 18.01.2024) (cit. on pp. 7, 14).

[41] H. Vault, *Intel SGX deprecation review*, 2022. [Online]. Available: https://hardenedvault.net/blog/2022-01-15-sgx-deprecated/, (accessed: 18.01.2024) (cit. on p. 7).

[42] H. Shacham, 'The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86),' in *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, P. Ning, S. D. C. di Vimercati and P. F. Syverson, Eds., ACM, 2007, pp. 552–561. DOI: 10.1145/1315245.1315313. [Online]. Available: https://doi.org/10.1145/1315245.1315313 (cit. on p. 7).

[43] T. A. Henzinger and J. Sifakis, 'The Embedded Systems Design Challenge,' in *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, J. Misra, T. Nipkow and E. Sekerinski, Eds., ser. Lecture Notes in Computer Science, vol. 4085, Springer, 2006, pp. 1–15. DOI: 10.1007/11813040_1. [Online]. Available: https://doi.org/10.1007/11813040_1 (cit. on pp. 8, 9).

[44] J. Yick, B. Mukherjee and D. Ghosal, 'Wireless sensor network survey,' *Comput. Networks*, vol. 52, no. 12, pp. 2292–2330, 2008. DOI: 10.1016/J.COMNET.2008.04.002. [Online]. Available: https://doi.org/10.1016/j.comnet.2008.04.002 (cit. on p. 8).

[45] EETimes. '2019 Embedded Markets Study.' (2019), [Online]. Available: https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf. (accessed: 18.01.2024) (cit. on pp. 8, 9).

[46] ARM. 'ARM GNU Toolchain.' (1992), [Online]. Available: https://developer.arm.com/Tools%20and%20Software/GNU%20Toolchain. (accessed: 18.01.2024) (cit. on p. 9).

[47] L. Hatton, 'Safer Language Subsets: an overview and a case history, MISRA C,' *Inf. Softw. Technol.*, vol. 46, no. 7, pp. 465–472, 2004. DOI: 10.1016/j.infsof.2003.09.016. [Online]. Available: https://doi.org/10.1016/j.infsof.2003.09.016 (cit. on p. 9).

[48] MITRE Corp. '2023 CWE Top 10 Key Exploited Vulnerabilities.' (2023), [Online]. Available: https://cwe.mitre.org/top25/archive/2023/2023_kev_list.html. (accessed: 18.01.2024) (cit. on pp. 9, 29).

[49] A. Dunkels, O. Schmidt, T. Voigt and M. Ali, 'Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems,' in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems, SenSys 2006, Boulder, Colorado, USA, October 31 - November 3, 2006*, A. T. Campbell, P. Bonnet and J. S. Heidemann, Eds., ACM, 2006, pp. 29–42. DOI: 10.1145/1182807.1182811. [Online]. Available: https://doi.org/10.1145/1182807.1182811 (cit. on p. 10).

[50] T. Mikkonen and A. Taivalsaari, 'Web applications - spaghetti code for the 21st century,' in *Proceedings of the 6th ACIS International Conference on Software Engineering Research, Management and Applications, SERA 2008, 20-22 August 2008, Prague, Czech Republic*, W. Dosch, R. Y. Lee, P. Tuma and T. Coupaye, Eds., IEEE Computer Society, 2008, pp. 319–328. DOI: 10.1109/SERA.2008.16. [Online]. Available: https://doi.org/10.1109/SERA.2008.16 (cit. on p. 10).

[51] S. Natarajan and D. Broman, 'Timed C: an extension to the C programming language for real-time systems,' in *IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2018, 11-13 April 2018, Porto, Portugal*, R. Pellizzoni, Ed., IEEE Computer Society, 2018, pp. 227–239. DOI: 10.1109/RTAS.2018.00031. [Online]. Available: https://doi.org/10.1109/RTAS.2018.00031 (cit. on p. 10).

[52] A. Sarkar, *Functional Programming for Embedded Systems*, Licentiate Thesis at Chalmers Tekniska Hogskola (Sweden), 2022. [Online]. Available: https://research.chalmers.se/publication/529325/file/529325_Fulltext.pdf, (accessed: 18.01.2024) (cit. on p. 11).

[53] A. Sarkar, R. Krook, A. Russo and K. Claessen, 'HasTEE: Programming Trusted Execution Environments with Haskell,' in *Proceedings of the 16th ACM SIGPLAN International Haskell Symposium, Haskell 2023, Seattle, WA, USA, September 8-9, 2023*, T. L. McDonell and N. Vazou, Eds., ACM, 2023, pp. 72–88. DOI: 10.1145/3609026.3609731. [Online]. Available: https://doi.org/10.1145/3609026.3609731 (cit. on pp. 11, 12, 21).

[54] Intel. 'tlibc - an alternative to glibc.' (2018), [Online]. Available: https://github.com/intel/linux-sgx/tree/master/common/inc/tlibc. (accessed: 18.01.2024) (cit. on p. 11).

[55] S. Marlow, S. L. P. Jones and S. Singh, 'Runtime support for multicore haskell,' in *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, G. Hutton and A. P. Tolmach, Eds., ACM, 2009, pp. 65–78. DOI: 10.1145/1596550.1596563. [Online]. Available: https://doi.org/10.1145/1596550.1596563 (cit. on p. 11).

[56] Intel, *Intel SGX Intro: Passing Data Between App and Enclave*, 2016. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/sgx-intro-passing-data-between-app-and-enclave.html, (accessed: 18.01.2024) (cit. on p. 12).

[57] D. B. MacQueen, 'Modules for standard ML,' in *Proceedings of the 1984 ACM Conference on LISP and Functional Programming, LFP 1984, Austin, Texas, USA, August 5-8, 1984*, R. S. Boyer, E. S. Schneider and G. L. S. Jr., Eds., ACM, 1984, pp. 198–207. DOI: 10.1145/800055.802036. [Online]. Available: https://doi.org/10.1145/800055.802036 (cit. on p. 12).

[58] A. Ekblad and K. Claessen, 'A seamless, client-centric programming model for type safe web applications,' in *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, W. Swierstra, Ed., ACM, 2014, pp. 79–89. DOI: 10.1145/2633357.2633367. [Online]. Available: https://doi.org/10.1145/2633357.2633367 (cit. on p. 13).

[59] S. Kilpatrick, D. Dreyer, S. L. P. Jones and S. Marlow, 'Backpack: Retrofitting haskell with interfaces,' in *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, S. Jagannathan and P. Sewell, Eds., ACM, 2014, pp. 19–32. DOI: 10.1145/2535838.2535884. [Online]. Available: https://doi.org/10.1145/2535838.2535884 (cit. on p. 13).

[60] E. Z. Yang, 'Backpack: Towards practical mix-in linking in haskell,' Ph.D. dissertation, Stanford University, USA, 2017. [Online]. Available: https://searchworks.stanford.edu/view/12082119 (cit. on p. 13).

[61] J. A. Goguen and J. Meseguer, 'Security Policies and Security Models,' in *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, IEEE Computer Society, 1982, pp. 11–20. DOI: 10.1109/SP.1982.10014. [Online]. Available: https://doi.org/10.1109/SP.1982.10014 (cit. on pp. 13, 14, 24).

[62] L. Li, Y. Fan and K. Lin, 'A survey on federated learning,' in *16th IEEE International Conference on Control & Automation, ICCA 2020, Singapore, October 9-11, 2020*, IEEE, 2020, pp. 791–796. DOI: 10.1109/ICCA51439.2020.9264412. [Online]. Available: https://doi.org/10.1109/ICCA51439.2020.9264412 (cit. on p. 13).

[63] C. Dwork, 'Differential privacy,' in *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II*, M. Bugliesi, B. Preneel, V. Sassone and I. Wegener, Eds., ser. Lecture Notes in Computer Science, vol. 4052, Springer, 2006, pp. 1–12. DOI: 10.1007/11787006_1. [Online]. Available: https://doi.org/10.1007/11787006_1 (cit. on p. 13).

[64] A. Sarkar and A. Russo, *HasTEE+ : Confidential Cloud Computing and Analytics with Haskell*, 2024. arXiv: 2401.08901 [cs.CR] (cit. on pp. 13, 21).

[65] J. V. Bulck, D. F. Oswald, E. Marin, A. Aldoseri, F. D. Garcia and F. Piessens, 'A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes,' in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, L. Cavallaro, J. Kinder, X. Wang and J. Katz, Eds., ACM, 2019, pp. 1741–1758. DOI: 10.1145/3319535.3363206. [Online]. Available: https://doi.org/10.1145/3319535.3363206 (cit. on p. 14).

[66] P. Weisenburger, J. Wirth and G. Salvaneschi, 'A survey of multitier programming,' *ACM Comput. Surv.*, vol. 53, no. 4, 81:1–81:35, 2021. [Online]. Available: https://doi.org/10.1145/3397495 (cit. on p. 14).

[67] T. Knauth, M. Steiner, S. Chakrabarti, L. Lei, C. Xing and M. Vij, 'Integrating remote attestation with transport layer security,' *arXiv preprint arXiv:1801.05863*, 2018 (cit. on p. 14).

[68] D. Stefan, A. Russo, J. C. Mitchell and D. Mazières, 'Flexible dynamic information flow control in haskell,' in *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*, K. Claessen, Ed., ACM, 2011, pp. 95–106. [Online]. Available: https://doi.org/10.1145/2034675.2034688 (cit. on pp. 14, 24).

[69] D. Stefan, A. Russo, D. Mazières and J. C. Mitchell, 'Disjunction Category Labels,' in *Information Security Technology for Applications - 16th Nordic Conference on Secure IT Systems, NordSec 2011, Tallinn, Estonia, October 26-28, 2011, Revised Selected Papers*, P. Laud, Ed., ser. Lecture Notes in Computer Science, vol. 7161, Springer, 2011, pp. 223–239. [Online]. Available: https://doi.org/10.1007/978-3-642-29615-4_16 (cit. on pp. 14, 22).

[70] A. C. Myers and B. Liskov, 'Protecting privacy using the decentralized label model,' *ACM Trans. Softw. Eng. Methodol.*, vol. 9, no. 4, pp. 410–442, 2000. DOI: 10.1145/363516.363526. [Online]. Available: https://doi.org/10.1145/363516.363526 (cit. on p. 14).

[71] A. Sabelfeld and D. Sands, 'Declassification: Dimensions and principles,' *J. Comput. Secur.*, vol. 17, no. 5, pp. 517–548, 2009. DOI: 10.3233/JCS-2009-0352. [Online]. Available: https://doi.org/10.3233/JCS-2009-0352 (cit. on p. 14).

[72] J. B. Dennis and E. C. V. Horn, 'Programming semantics for multiprogrammed computations,' *Commun. ACM*, vol. 9, no. 3, pp. 143–155, 1966. DOI: 10.1145/365230.365252. [Online]. Available: https://doi.org/10.1145/365230.365252 (cit. on p. 14).

[73] T. Herbrich, 'Data clean rooms,' *Computer Law Review International*, vol. 23, no. 4, pp. 109–120, 2022 (cit. on p. 15).

[74] S. Marlow, 'Parallel and Concurrent Programming in Haskell,' in *Central European Functional Programming School - 4th Summer School, CEFP 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*, V. Zsók, Z. Horváth and R. Plasmeijer, Eds., ser. Lecture Notes in Computer Science, vol. 7241, Springer, 2011, pp. 339–401. DOI: 10.1007/978-3-642-32096-5_7. [Online]. Available: https://doi.org/10.1007/978-3-642-32096-5_7 (cit. on p. 15).

[75] J. V. Bulck, F. Piessens and R. Strackx, 'SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control,' in *Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*, ACM, 2017, 4:1–4:6. DOI: 10.1145/3152701.3152706. [Online]. Available: https://doi.org/10.1145/3152701.3152706 (cit. on p. 15).

[76] J. V. Bulck, M. Minkin, O. Weisse *et al.*, 'Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution,' in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds., USENIX Association, 2018, pp. 991–1008. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck (cit. on pp. 15, 28).

[77] P. Borrello, A. Kogler, M. Schwarzl, M. Lipp, D. Gruss and M. Schwarz, 'ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture,' in *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, K. R. B. Butler and K. Thomas, Eds., USENIX Association, 2022, pp. 3917–3934. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/borrello (cit. on p. 15).

[78] A. Sarkar, B. J. Svensson and M. Sheeran, 'Synchron - An API and Runtime for Embedded Systems,' in *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany*, K. Ali and J. Vitek, Eds., ser. LIPIcs, vol. 222, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 17:1–17:29. DOI: 10.4230/LIPICS.ECOOP.2022.17. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2022.17 (cit. on pp. 15, 19, 21, 23).

[79] G. Cousineau, P. Curien and M. Mauny, 'The Categorical Abstract Machine,' in *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings*, J. Jouannaud, Ed., ser. Lecture Notes in Computer Science, vol. 201, Springer, 1985, pp. 50–64. DOI: 10.1007/3-540-15975-4_29. [Online]. Available: https://doi.org/10.1007/3-540-15975-4_29 (cit. on p. 16).

[80] P. Weis, M. V. Aponte, A. Laville, M. Mauny and A. Suárez, 'The CAML Reference Manual,' Ph.D. dissertation, INRIA, 1990 (cit. on p. 16).

[81] J. H. Reppy, 'Concurrent ML: Design, Application and Semantics,' in *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991-1992, McMaster University, Hamilton, Ontario, Canada*, P. E. Lauer, Ed., ser. Lecture Notes in Computer Science, vol. 693, Springer, 1993, pp. 165–198. DOI: 10.1007/3-540-56883-2_10. [Online]. Available: https://doi.org/10.1007/3-540-56883-2_10 (cit. on pp. 16, 22).

[82] C. A. R. Hoare, 'Communicating Sequential Processes,' *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978. DOI: 10.1145/359576.359585. [Online]. Available: https://doi.org/10.1145/359576.359585 (cit. on pp. 16, 24).

[83] A. Sarkar, R. Krook, B. J. Svensson and M. Sheeran, 'Higher-Order Concurrency for Microcontrollers,' in *MPLR '21: 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, Münster, Germany, September 29-30, 2021*, H. Kuchen and J. Singer, Eds., ACM, 2021, pp. 26–35. DOI: 10.1145/3475738.3480716. [Online]. Available: https://doi.org/10.1145/3475738.3480716 (cit. on p. 17).

[84] Nordic Semiconductors. 'nRF52840DK.' (2017), [Online]. Available: https://www.nordicsemi.com/Products/Development-hardware/nrf52840-dk (cit. on p. 17).

[85] ST Microelectronics. 'STM32F4DISCOVERY.' (2011), [Online]. Available: https://www.st.com/en/evaluation-tools/stm32f4discovery.html (cit. on p. 17).

[86] X. Leroy, 'The ZINC experiment: an economical implementation of the ML language,' Ph.D. dissertation, INRIA, 1990 (cit. on p. 18).

[87] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy and J. Vouillon, 'The ocaml system: Documentation and user's manual,' *INRIA*, vol. 3, p. 42, (cit. on p. 18).

[88] ARM. 'TrustZone for Cortex M.' (2017), [Online]. Available: https://www.arm.com/technologies/trustzone-for-cortex-m (cit. on p. 18).

[89] R. N. M. Watson, J. Woodruff, P. G. Neumann *et al.*, 'CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization,' in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, IEEE Computer Society, 2015, pp. 20–37. DOI: 10.1109/SP.2015.9. [Online]. Available: https://doi.org/10.1109/SP.2015.9 (cit. on p. 18).

[90] A. Sarkar and M. Sheeran, 'Hailstorm: A Statically-Typed, Purely Functional Language for IoT applications,' in *Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming*, 2020, pp. 1–16 (cit. on pp. 18, 21).

[91] C. Elliott and P. Hudak, 'Functional Reactive Animation,' in *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997*, S. L. P. Jones, M. Tofte and A. M. Berman, Eds., ACM, 1997, pp. 263–273. DOI: 10.1145/258948.258973. [Online]. Available: https://doi.org/10.1145/258948.258973 (cit. on p. 18).

[92] C. Elliott. 'Can functional programming be liberated from the von Neumann paradigm.' (2010), [Online]. Available: http://conal.net/blog/posts/can-functional-programming-be-liberated-from-the-von-neumann-paradigm (cit. on p. 18).

[93] H. Nilsson, A. Courtney and J. Peterson, 'Functional Reactive Programming, Continued,' in *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, 2002, pp. 51–64 (cit. on p. 18).

[94] D. Winograd-Cort, H. Liu and P. Hudak, 'Virtualizing Real-World Objects in FRP,' in *Practical Aspects of Declarative Languages - 14th International Symposium, PADL 2012, Philadelphia, PA, USA, January 23-24, 2012. Proceedings*, C. V. Russo and N. Zhou, Eds., ser. Lecture Notes in Computer Science, vol. 7149, Springer, 2012, pp. 227–241. DOI: 10.1007/978-3-642-27694-1_17. [Online]. Available: https://doi.org/10.1007/978-3-642-27694-1_17 (cit. on pp. 18, 23).

[95] M. Sustrik. 'Structured Concurrency.' (2016), [Online]. Available: https://250bpm.com/blog:71/ (cit. on p. 21).

[96] M. Sustrik. 'ZeroMQ : An open-source universal messaging library.' (2007), [Online]. Available: https://github.com/zeromq/libzmq (cit. on p. 21).

[97] B. W. Lampson, 'A Note on the Confinement Problem,' *Commun. ACM*, vol. 16, no. 10, pp. 613–615, 1973. DOI: 10.1145/362375.362389. [Online]. Available: https://doi.org/10.1145/362375.362389 (cit. on p. 22).

[98] A. M. Gundry, 'Type inference, Haskell and dependent types,' Ph.D. dissertation, University of Strathclyde, Glasgow, UK, 2013. [Online]. Available: http://oleg.lib.strath.ac.uk/R/?func=dbin-jump-full\&object_id=22728 (cit. on p. 23).

[99] A. Ghosn, J. R. Larus and E. Bugnion, 'Secured Routines: Language-based Construction of Trusted Execution Environments,' in *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, D. Malkhi and D. Tsafrir, Eds., USENIX Association, 2019, pp. 571–586. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/ghosn (cit. on p. 23).

[100] S. Zdancewic, L. Zheng, N. Nystrom and A. C. Myers, 'Secure program partitioning,' *ACM Trans. Comput. Syst.*, vol. 20, no. 3, pp. 283–328, 2002. DOI: 10.1145/566340.566343. [Online]. Available: https://doi.org/10.1145/566340.566343 (cit. on p. 23).

[101] V. Simonet, 'The Flow Caml system,' *Software release. Located at http://cristal. inria. fr/~ simonet/soft/flowcaml*, vol. 116, pp. 119–156, 2003 (cit. on p. 23).

[102] Philip Wadler, 'Comprehending monads,' in *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*, G. Kahn, Ed., ACM, 1990, pp. 61–78. DOI: 10.1145/91556.91592. [Online]. Available: https://doi.org/10.1145/91556.91592 (cit. on p. 23).

[103] S. D. Brookes, C. A. R. Hoare and A. W. Roscoe, 'A Theory of Communicating Sequential Processes,' *J. ACM*, vol. 31, no. 3, pp. 560–599, 1984. DOI: 10.1145/828.833. [Online]. Available: https://doi.org/10.1145/828.833 (cit. on p. 24).

[104] J. Hughes, 'Generalising monads to arrows,' *Sci. Comput. Program.*, vol. 37, no. 1-3, pp. 67–111, 2000. DOI: 10.1016/S0167-6423(99)00023-4. [Online]. Available: https://doi.org/10.1016/S0167-6423(99)00023-4 (cit. on p. 24).

[105] A. K. Wright and M. Felleisen, 'A Syntactic Approach to Type Soundness,' *Inf. Comput.*, vol. 115, no. 1, pp. 38–94, 1994. DOI: 10.1006/INCO.1994.1093. [Online]. Available: https://doi.org/10.1006/inco.1994.1093 (cit. on p. 24).

[106] L. Damas and R. Milner, 'Principal Type-Schemes for Functional Programs,' in *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*, R. A. DeMillo, Ed., ACM Press, 1982, pp. 207–212. DOI: 10.1145/582153.582176. [Online]. Available: https://doi.org/10.1145/582153.582176 (cit. on p. 24).

[107] M. Vassena and A. Russo, 'On Formalizing Information-Flow Control Libraries,' in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016*, T. C. Murray and D. Stefan, Eds., ACM, 2016, pp. 15–28. DOI: 10.1145/2993600.2993608. [Online]. Available: https://doi.org/10.1145/2993600.2993608 (cit. on p. 24).

[108] R. Milner, M. Tofte and R. Harper, *Definition of Standard ML*. MIT Press, 1990, ISBN: 978-0-262-63132-7 (cit. on p. 24).

[109] J. Marty, L. Franceschino, J. Talpin and N. Vazou, 'LIO*: Low Level Information Flow Control in F,' *CoRR*, vol. abs/2004.12885, 2020. arXiv: 2004.12885. [Online]. Available: https://arxiv.org/abs/2004.12885 (cit. on p. 24).

[110] N. D. Matsakis and F. S. K. II, 'The Rust Language,' in *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, M. B. Feldman and S. T. Taft, Eds., ACM, 2014, pp. 103–104. DOI: 10.1145/2663171.2663188. [Online]. Available: https://doi.org/10.1145/2663171.2663188 (cit. on p. 25).

[111] Rust Crate. 'task_scope: A runtime extension for adding support for Structured Concurrency to existing runtimes.' (2020), [Online]. Available: https://docs.rs/task_scope/latest/task_scope/ (cit. on p. 25).

[112] H. Wang, P. Wang, Y. Ding *et al.*, 'Towards Memory Safe Enclave Programming with Rust-SGX,' in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, L. Cavallaro, J. Kinder, X. Wang and J. Katz, Eds., ACM, 2019, pp. 2333–2350. DOI: 10.1145/3319535.3354241. [Online]. Available: https://doi.org/10.1145/3319535.3354241 (cit. on p. 25).

[113] T. Cloosters, J. Willbold, T. Holz and L. Davi, 'Sgxfuzz: Efficiently synthesizing nested structures for SGX enclave fuzzing,' in *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, K. R. B. Butler and K. Thomas, Eds., USENIX Association, 2022, pp. 3147–3164. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/cloosters (cit. on p. 27).

[114] M. Busch, A. Machiry, C. Spensky, G. Vigna, C. Kruegel and M. Payer, 'TEEzz: Fuzzing Trusted Applications on COTS Android Devices,' in *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, IEEE, 2023, pp. 1204–1219. DOI: 10.1109/SP46215.2023.10179302. [Online]. Available: https://doi.org/10.1109/SP46215.2023.10179302 (cit. on p. 27).

[115] G. Grieco, M. Ceresa and P. Buiras, 'Quickfuzz: An automatic random fuzzer for common file formats,' in *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, G. Mainland, Ed., ACM, 2016, pp. 13–20. DOI: 10.1145/2976002.2976017. [Online]. Available: https://doi.org/10.1145/2976002.2976017 (cit. on p. 27).

[116] M. R. Clarkson and F. B. Schneider, 'Hyperproperties,' in *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008*, IEEE Computer Society, 2008, pp. 51–65. DOI: 10.1109/CSF.2008.7. [Online]. Available: https://doi.org/10.1109/CSF.2008.7 (cit. on p. 27).

[117] C. Hritcu, L. Lampropoulos, A. Spector-Zabusky *et al.*, 'Testing noninterference, quickly,' *J. Funct. Program.*, vol. 26, e4, 2016. DOI: 10.1017/S0956796816000058. [Online]. Available: https://doi.org/10.1017/S0956796816000058 (cit. on p. 27).

[118] A. Nagarajan, V. Varadharajan, M. Hitchens and E. Gallery, 'Property Based Attestation and Trusted Computing: Analysis and Challenges,' in *Third International Conference on Network and System Security, NSS 2009, Gold Coast, Queensland, Australia, October 19-21, 2009*, Y. Xiang, J. López, H. Wang and W. Zhou, Eds., IEEE Computer Society, 2009, pp. 278–285. DOI: 10.1109/NSS.2009.83. [Online]. Available: https://doi.org/10.1109/NSS.2009.83 (cit. on p. 28).

[119] A. Sadeghi and C. Stüble, 'Property-based attestation for computing platforms: Caring about properties, not mechanisms,' in *Proceedings of the New Security Paradigms Workshop 2004, September 20-23, 2004, Nova Scotia, Canada*, C. Hempelmann and V. Raskin, Eds., ACM, 2004, pp. 67–77. DOI: 10.1145/1065907.1066038. [Online]. Available: https://doi.org/10.1145/1065907.1066038 (cit. on p. 28).

[120] T. C. Group, 'Trusted platform module (tpm) main specification,' *Trusted Computing Group*, 2020. [Online]. Available: https://trustedcomputinggroup.org/resource/tpm-main-specification/ (cit. on p. 28).

[121] F. Hublet, D. A. Basin and S. Krstic, 'User-Controlled Privacy: Taint, Track, and Control,' *Proc. Priv. Enhancing Technol.*, vol. 2024, no. 1, pp. 597–616, 2024. DOI: 10.56553/POPETS-2024-0034. [Online]. Available: https://doi.org/10.56553/popets-2024-0034 (cit. on p. 28).

[122] A. Nilsson, P. N. Bideh and J. Brorsson, 'A Survey of Published Attacks on Intel SGX,' *CoRR*, vol. abs/2006.13598, 2020. arXiv: 2006.13598. [Online]. Available: https://arxiv.org/abs/2006.13598 (cit. on p. 28).

[123] K. Murdock, D. F. Oswald, F. D. Garcia, J. V. Bulck, D. Gruss and F. Piessens, 'Plundervolt: Software-based Fault Injection Attacks against Intel SGX,' in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, IEEE, 2020, pp. 1466–1482. DOI: 10.1109/SP40000.2020.00057. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00057 (cit. on p. 28).

[124] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl and D. Gruss, 'ConTExT: A Generic Approach for Mitigating Spectre,' in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, The Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/context-a-generic-approach-for-mitigating-spectre/ (cit. on p. 28).

[125] L. Daniel, M. Bognar, J. Noorman, S. Bardin, T. Rezk and F. Piessens, 'Prospect: Provably secure speculation for the constant-time policy,' in *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, J. A. Calandrino and C. Troncoso, Eds., USENIX Association, 2023, pp. 7161–7178. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/daniel (cit. on p. 28).

[126] J. Agat, 'Transforming Out Timing Leaks,' in *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*, M. N. Wegman and T. W. Reps, Eds., ACM, 2000, pp. 40–53. DOI: 10.1145/325694.325702. [Online]. Available: https://doi.org/10.1145/325694.325702 (cit. on p. 28).

[127] S. Cauligi, G. Soeller, B. Johannesmeyer *et al.*, 'Fact: A DSL for timing-sensitive computation,' in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, K. S. McKinley and K. Fisher, Eds., ACM, 2019, pp. 174–189. DOI: 10.1145/3314221.3314605. [Online]. Available: https://doi.org/10.1145/3314221.3314605 (cit. on p. 28).

[128] M. Pickering, N. Wu and B. Németh, 'Working with source plugins,' in *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019*, R. A. Eisenberg, Ed., ACM, 2019, pp. 85–97. DOI: 10.1145/3331545.3342599. [Online]. Available: https://doi.org/10.1145/3331545.3342599 (cit. on p. 28).

[129] S. Amar, D. Chisnall, T. Chen *et al.*, 'CHERIoT: Complete Memory Safety for Embedded Devices,' in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2023, Toronto, ON, Canada, 28 October 2023 - 1 November 2023*, ACM, 2023, pp. 641–653. DOI: 10.1145/3613424.3614266. [Online]. Available: https://doi.org/10.1145/3613424.3614266 (cit. on p. 29).

PART I

FUNCTIONAL PROGRAMMING FOR SECURITY IN THE CLOUD

Chapter 5

HasTEE: Programming TEEs with Haskell

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.

CCS Concepts: • **Security and privacy → Trusted computing**; **Information flow control**; **Security in hardware**; • **Software and its engineering → Functional languages**; **Domain specific languages**.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
Haskell '23, September 8–9, 2023, Seattle, WA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0298-3/23/09.
https://doi.org/10.1145/3609026.3609731

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 terminology, we shall use the term *enclave* (adopted from Intel) to refer to the isolated region of code and data and its trusted computing base (TCB).

TEEs, despite their strong security guarantees, have seen limited adoption in software development owing to several challenges. Firstly, TEEs often present *an awkward and low-level programming model* [Decentriq 2022]. For instance, Intel provides a C/C++ interface to program SGX that requires *partitioning* the program's state into trusted and untrusted components and dividing the entire logic into two separate software projects (Section 2)—a complex and error-prone process that could lead to data leakage. From a security perspective, the use of C/C++ APIs can open further opportunities to exploit well-known memory-unsafe vulnerabilities such as return-oriented programming (ROP) [Shacham 2007]

in applications running inside TEEs [Muñoz et al. 2023]. Secondly, *current TEE programming models are insufficient to enforce security policies.* Applications should be written in a way such that they do not accidentally reveal confidential information. Furthermore, inputs and outputs to an enclave must be correctly encrypted, signed, decrypted, and verified to protect against malicious hosts. Thirdly, *little support is given to migrate legacy applications inside enclaves.* Applications inside enclaves often rely on their own Operating System (OS) since they cannot trust the one in the host machine. Library OS-based approaches exist to provide this functionality. However, for legacy applications written in high-level languages relying on non-trivial runtimes, the porting of the runtime becomes a challenging task.

Efforts have been made to address these challenges. The work by Ghosn et al. [2019] introduces GoTEE, a modification of the Go programming language with support for *secure routines* that are executed inside enclaves. In GoTEE, the authors heavily modify the Go compiler and extend the language to support new TEE-specific abstractions that helps to automatically partition an application. GoTEE does not provide any control over how sensitive information moves within the application, which could enable accidental data leaks. In a similar spirit, Oak et al. [2021] introduce J_E, a subset of Java with support for enclaves. J_E focuses on providing information-flow control (IFC) to ensure that the code does not leak sensitive data by accident or by coercion of a malicious host. J_E uses a sophisticated compilation pipeline to first partition the application and then uses another compiler to check that sensitive information is not leaked. Virtualization-based solutions, such as AMD SEV [AMD 2018], attempt to alleviate the effort required to port legacy applications. However, the trade-off is that the TCB becomes larger and the granularity to identify sensitive data becomes much coarser.

Our contribution through this paper is *HasTEE*, a domain-specific language (DSL) embedded in Haskell for programming TEE applications. HasTEE integrates TEE-specific abstraction and semantics while hiding low-level hardware intricacies making it hardware neutral! Additionally, HasTEE offers IFC to prevent accidental leakage of sensitive data. Owing to its embedding in Haskell, developers can use familiar abstractions such as high-order functions, monads, and a limited set of I/O operations to write applications in a conventional manner. This design choice enables seamless integration with all of the existing Haskell features. Compared to the previous work, HasTEE is lightweight, simple, and is provided as a *simple security library*; thus avoiding any GHC [Jones et al. 1993] compiler modifications!

HasTEE by Example

Listing 1 presents a sample password checker application written using HasTEE.

```
1  pwdChkr :: Enclave String -> String -> Enclave Bool
2  pwdChkr pwd guess = fmap (== guess) pwd
3
4  passwordChecker :: App Done
5  passwordChecker = do
6    passwd <- inEnclaveConstant "secret"
7    efunc  <- inEnclave $ pwdChkr passwd
8    runClient $ do -- Client code
9      liftIO $ putStrLn "Enter your password"
10     userInput <- liftIO getLine
11     res       <- gateway (efunc <@> userInput)
12     liftIO $ putStrLn ("Login returned " ++ show res)
```

Listing 1. A password checker written in HasTEE

The distinction between the trusted and untrusted parts of the application is done via the type system that encodes the former as the `Enclave` type (line 1) and the latter as the `Client` type (type inferred in line 8).

The function `pwdChkr` takes a sensitive string located in the enclave (`Enclave String`), a public string from the client host (`String`) and produces a sensitive Boolean in the enclave (`Enclave Bool`). Line 6 holds the secret string that we want to protect (`inEnclaveConstant`). Line 7 uses the `inEnclave` call to obtain a reference to the function `pwdChkr` located in the enclave. The function `gateway` (line 11) is responsible for transmitting the collected arguments to the enclave function, and bringing the result back to the client. The `gateway` function acts as an interface between the enclave and non-enclave environment. The untrusted *host client* is in charge of driving the application, while the *enclave* is assigned the role of a computational and/or storage resource that services the client's requests. HasTEE connects an application (`passwordChecker`) to Haskell's `main` method using the `runApp :: App a -> IO a` function that executes the application. From an IFC perspective, lines 6 and 7 correspond to labelling, i.e., establishing, which inputs are sensitive for the program—an activity that is part of the TCB. In general, HasTEE code starts by labelling the sensitive input with the `inEnclave` primitive. Subsequently, the client code is compelled to manipulate secrets in a *secure* manner. In this setting, secure means that no sensitive information in the enclave gets leaked except that it has been obtained via the primitive `gateway`. The HasTEE API is explained in Section 4.2, and the semantics are discussed in Section 4.3.

Contributions

A type-safe, secure, high-level programming model. The HasTEE library enables developers to program a TEE environment, such as Intel SGX, using Haskell - a type-safe, memory-managed language whose expressive type system can be leveraged to enforce various security constraints. Additionally, HasTEE allows programming in a familiar client-server style programming model (Section 4.2 and 5.2), an improvement over the low-level Intel SGX APIs.

Automatic Partitioning. A key part of programming TEEs, partitioning the trusted and untrusted parts of the program is done automatically using the type system (details in Section 3 and 4.3). Crucially, our approach does not require any modification of the GHC compiler and can be adapted to other programming languages, as long as their runtime can run on the desired TEE infrastructure.

Information Flow Control. Drawing inspiration from restricted IO monad families in Haskell, we designed an Enclave monad that prevents accidental leaks of secret data by TEE programmers (Section 5.3). Hence, our Enclave monad enables writing applications with a relatively low level of trust placed on the enclave programmer.

Portability of Haskell's runtime. We modify the GHC runtime, without modifying the compiler, to run on SGX enclaves. This enables us to host the complete Haskell language, including extensions, supported by GHC 8.8 (Section 5.1).

Demonstration of expressiveness. We illustrate the practicality of the HasTEE through three case studies across different domains: (1) a Federated Learning example (Section 6.1), (2) an encrypted password wallet (Section 6.2) and (3) a *differentially-private* data clean room (Section 6.3). The examples also demonstrate the simplicity of TEE development enabled by HasTEE.

2 Background

Intel Software Guard Extensions (SGX)

Intel Software Guard Extensions (SGX) [Intel 2015] is a set of security-related instructions supported since Intel's sixth-generation Skylake processor, which can enhance the security of applications by providing a *secure enclave* for processing sensitive data. The enclave is a disjoint portion of memory separate from the DRAM, where sensitive data and code reside, beyond the influence of an untrusted operating system and other low-level software.

Intel offers an SGX SDK for programming enclaves. The SDK requires dividing the application into trusted and untrusted parts, where sensitive data resides in the trusted project. It provides specialized function calls called *ecall* for enclave access and an *ocall* API for communication with the untrusted client. The boundary between the client and enclave is defined using an *Enclave Description Language (EDL).* The SDK utilizes a tool called *edger8r* to parse EDL files and generate two *bridge* files. These files ensure secure data transfer between projects through *copying* instead of sharing via pointers, preventing potential manipulation of the enclave's state. Fig 1 shows the SDK's programming model.

Application developers working with enclaves aim to minimize the Trusted Computing Base (TCB) by keeping the operating system and system software outside the enclave. The SGX SDK offers a restricted C standard library implementation (`tlibc`) for essential system software. Programming

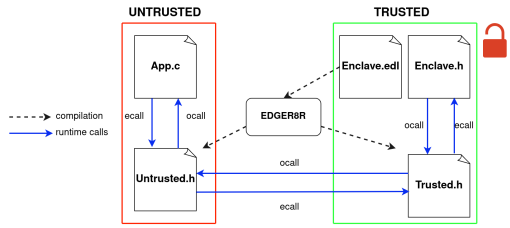


Figure 1. Intel SGX SDK Programming Model

SGX enclaves involves understanding the complex control flow between trusted and untrusted components. Enforcing SGX's programming model on typical software projects can be challenging, and the limited `tlibc` library restricts running applications beyond those written in vanilla C/C++.

3 Key Idea: A Typed DSL for Enclaves

The Programming Model and Partitioning

HasTEE supports the automatic partitioning of programs by utilizing a combination of the type system to identify the enclave and a conditional compilation tactic to provide different semantics to each component. The compilation tactic was first used in Haste.App [Ekblad and Claessen 2014], to partition a single program into a `Client` and `Server` type. Fig 2 shows the partitioning procedure at a high level.

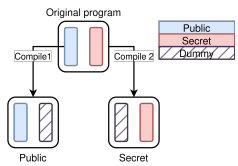


Figure 2. The HasTEE partitioning scheme

Importantly, this approach does not require any compiler extensions or elaborate dependency analysis passes to distinguish between the underlying types. The codebase involved in other complex partitioning approaches [Ghosn et al. 2019; Oak et al. 2021] becomes part of the Trusted Computing Base (TCB), creating a larger TCB. In contrast, our approach does not add any partitioning code to the TCB. Fig 3 shows the partitioned software stack in the HasTEE approach.

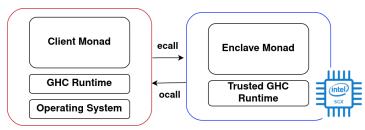


Figure 3. The untrusted (left) and trusted (right) software

Post-partitioning, the client-server-style programming model is used for programming the enclave. In this model, the client takes on the primary role of driving the program and utilizes the enclave as a computational and/or storage resource. The source program, written in Haskell, benefits from type safety, while HasTEE internally handles the message transfer between the client and enclave memory at runtime.

Information Flow Control on Enclaves

Being a Haskell library enables HasTEE to tap into the library-based Information Flow Control techniques in Haskell [Buiras et al. 2015; Russo 2015; Russo et al. 2008]. The IFC literature distinguishes between sensitive and non-sensitive computations via monads indexed with security levels [Russo et al. 2008], e.g., `Sec H` and `Sec L`, where security levels `H` and `L` are assigned to sensitive and public information, respectively. Public information can flow into sensitive entities but not in the other way around. We have a similar security-level hierarchy between the `Enclave` and `Client` monads, respectively. Accordingly, we design the `Enclave` monad such that it restricts the possible variants of I/O operations. Internally, the `Enclave` monad constrains the scope of side-effecting operations to protect the confidentiality of data within the enclave (details in Section 5.3). Furthermore, HasTEE demands to explicitly mark where information is being sent back to the client (`gateway`), thus clearly indicating where to audit and control information leakages. Due to the security-critical nature of the Enclave monad, we include a `trust` operator, which is similar to the `endorse` function found in IFC literature.

Trusted GHC Runtime

One of the key challenges in allowing Haskell programs to run on TEE platforms is to provide support for the GHC Haskell Runtime [Marlow et al. 2009] itself. A Haskell program relies on the runtime for essential tasks such as memory allocation, concurrency, I/O management, etc. The GHC runtime heavily depends on well-known C standard libraries, such as `glibc` on Linux [GNUDevs 1991] and `msvcrt` on Windows [Microsoft 1994]. In contrast, the Intel SGX SDK provides a much more restricted `libc` known as `tlibc`.

This results in the fact that several `libc` calls used by the GHC runtime such as `mmap`, `madvise`, `epoll`, `select` and 100+ other functions become unavailable. Even the core threading library used by the GHC runtime, `pthread`, has a much more restricted API on the SGX SDK. To solve this conundrum, we have patched portions of the GHC runtime and used functionalities from a library OS, Gramine [C. Tsai, Porter, et al. 2017], to enable the execution of GHC-compiled programs on the enclave.

TEE Independence

Finally, HasTEE provides an abstraction over low-level system APIs offered by TEEs. As a result, the principles applied in programming Intel SGX should translate to the programming of other popular TEEs, such as the ARM TrustZone.

4 Design of HasTEE

4.1 Threat Model

We begin by discussing the threat model of the HasTEE DSL. HasTEE has the very same threat model as that of Intel SGX. In this model, only the software running inside the enclave memory is trusted. All other application and system software, such as the operating system, hypervisors, driver firmware, etc., are considered compromised by an attacker. A very similar threat model is shared by a number of other work based on Intel SGX [Arnautov et al. 2016; Baumann et al. 2015; Ghosn et al. 2019; Lind et al. 2017].

In this work, we enhance the application-level security firstly by using a memory-safe language, Haskell, and secondly use the `Enclave` monad to introduce information flow control. Our implementation strategy of loading the GHC runtime on the enclave allows us to handle Iago attacks [Checkoway and Shacham 2013] (see Section 5.1). We trust the underlying implementation of the SGX hardware and software stack (such as `tlibc`) as provided by Intel. Known limitations of Intel SGX such as denial-of-service attacks and side-channel attacks [Schaik et al. 2022] are beyond the scope of this paper.

An ideally secure development process should include auditing the code running on the enclave either through static analyses or manual code reviews or both. The conciseness of Haskell codebases should generally facilitate the auditing process. However, the mechanisms for fail-proof audits are beyond the scope of this paper as well.

4.2 HasTEE API

We show the core API of HasTEE in Fig 4. The functions presented operate over three principal Haskell data types: (1) `Enclave`, (2) `Client`, and (3) `App`. All three types are instances of the `Monad` typeclass, which allows for the use of `do` notation when programming with them. One of the key differences in functionality provided by the `Client` and `Enclave` monads is that `Client` allows for arbitrary I/O, whereas `Enclave` only provides restricted I/O. More on the latter in Section 5.3. The `App` monad sets up the infrastructure for communication between the `Client` and `Enclave` monad. We show a simple secure counter written using most of the API in Listing 2.

Listing 2 internally gets partitioned into the trusted and untrusted components via conditional compilation. In line 3, `liftNewRef` is used to create a secure reference initialised to the value 0. Followed by that, the computation to increment this value inside the enclave is given in lines 4 - 7. Applying `inEnclave` on the enclave computation (line 4) yields the type `App (Secure (Enclave Int))`. The `Secure` type is HasTEE's internal representation of a closure. Line

```haskell
-- mutable references
liftNewRef :: a → App (Enclave (Ref a))
readRef    :: Ref a → Enclave a
writeRef   :: Ref a → a → Enclave ()

-- get reference to function inside enclave
inEnclave :: Securable a ⇒ a → App (Secure a)

-- runs the Client monad
runClient :: Client () → App Done

-- used for function application on the enclave
gateway :: Binary a ⇒ Secure (Enclave a) → Client a
(<@>) :: Binary a ⇒ Secure (a → b) → a → Secure b

-- call this from `main` to run the App monad
runApp :: App a → IO a
```

Figure 4. The core HasTEE API

```haskell
app :: App Done
app = do
  enclaveRef <- liftNewRef 0 :: App (Enclave (Ref Int))
  count <- inEnclave $ do
    r <- enclaveRef
    v <- readRef r
    writeRef r (v + 1) >> return v :: Enclave Int
  runClient $ gateway count >>=
              \v -> liftIO $ print $ "Counter's #" ++ show v

main = runApp app
```

Listing 2. A secure counter written in HasTEE (types annotated for clarity)

8 uses the critical `gateway` function to actually execute the enclave computation within the enclave memory and get the result back in the client memory. This resulting value, `v`, is displayed to the user.

The only function from Fig. 4 not used in Listing 2 is the `<@>` operator, used to collect arguments that are sent to the enclave. For example, an enclave function, `f`, that accepts two arguments, `arg1` and `arg2`, would be executed as `gateway (f <@> arg1 <@> arg2)`. Parameters to secure functions are copied to the enclave before the function is invoked, and results are copied from the enclave to the client before the client resumes execution. To do this copying, `gateway` and `<@>` has a `Binary` constraint on the types involved. This specifies that the values of the types involved have to be serialisable. Listing 1 in Section 1 shows a concrete usage of the operator. We have larger case studies in Section 6.

4.3 Operational Semantics of HasTEE

We provide big-step operational semantics of the HasTEE DSL. Note, we illustrate the semantics using an interpreter written in Haskell that shows the transition of the client as well as the enclave memory as each operators gets interpreted. We show our *expression language* and the abstract machine values to which we evaluate below:

```haskell
type Name = String

data Exp = Lit Int | Var Name | Fun [Name] Exp
  | App Exp [Exp] | Let Name Exp Exp | Plus Exp Exp
  | InEnclave Exp | Gateway Exp | EnclaveApp Exp Exp --HasTEE

data Value = IntVal Int | Closure [Name] Exp Env
  | SecureClosure Name [Value] | ArgList [Value] | Dummy
  | Err ErrState -- Error conditions
```

The `Exp` language above is a slightly modified version of lambda calculus with the restriction of allowing only fully applied function application. This restriction is done to reflect the nature of the HasTEE API, which through the type system, only permits fully saturated function applications for functions residing in the enclave. The lambda calculus language is then extended with the core HasTEE operators.

In the `Value` type, the `Closure` constructor, owing to saturated function application, captures a list of variable names and the environment. Notable in the `Value` type is the `SecureClosure` constructor that represents a closure residing in the enclave memory. This constructor does not capture the body of the closure as the body could hold any hidden state that lies protected within the enclave memory. The `SecureClosure` value is used by the `Gateway` function to invoke functions residing in the enclave.

The `ArgList` constructor supports the `<@>` operator that collects enclave function arguments. Lastly, the `Dummy` value is used as a placeholder for operators lacking semantics depending on the client or the enclave memory. For instance, the `Gateway` function has no meaning inside the `Enclave` monad, it is only usable from the `Client` monad. The `Dummy` crucially enables the conditional compilation trick in HasTEE by acting as a placeholder for meaningless functions in the respective client and enclave memory.

Our evaluators will show transition relations operating on two distinct memories that maps variable names to values - the enclave memory and the client memory.

```haskell
type ClientEnv  = [(Name, Value)]
type EnclaveEnv = [(Name, Value)]
```

Accordingly, we define two evaluators - `evalEnclave` (Fig. 5) and `evalClient` (Fig. 6). The complete evaluator run in two passes. In the first pass, it runs a program and loads up the necessary elements in the enclave memory and then in the second pass, the loaded enclave memory is additionally passed to the client's evaluator.

Two helper functions, `genEncVar` and `evalList` are not shown for concision. They generate unique variable names and fold over a list of expressions respectively. Appendix A contains the complete, typechecked semantics as runnable Haskell code.

We use Listing 3 to illustrate how the enclave, as well as the client memory, evolves as a program gets evaluated. Our

```
1  evalEnclave :: (MonadState StateVar m)
2              ⇒ Exp → EnclaveEnv → m (Value, EnclaveEnv)
3  evalEnclave (Lit n) env    = pure (IntVal n, env)
4  evalEnclave (Var x) env    = pure (lookupVar x env, env)
5  evalEnclave (Fun xs e) env =
6    pure (Closure xs e env, env)
7  evalEnclave (Let name e1 e2) env = do
8    (e1', env') ← evalEnclave e1 env
9    evalEnclave e2 ((name,e1'):env')
10 evalEnclave (App f args) env     = do
11   (v1, env1)   ← evalEnclave f env
12   (vals, env2) ← evalList args env1 []
13   case v1 of
14     Closure xs body ev →
15       evalEnclave body ((zip xs vals) ++ ev)
16     _ → pure (Err ENotClosure, env2)
17 evalEnclave (Plus e1 e2) env = do
18   (v1, env1) ← evalEnclave e1 env
19   (v2, env2) ← evalEnclave e2 env1
20   case (v1, v2) of
21     (IntVal a1, IntVal a2) → pure (IntVal (a1 + a2), env2)
22     _                      → pure (Err ENotIntLit, env2)
23 evalEnclave (InEnclave e) env = do
24   (val, env') ← evalEnclave e env
25   varname     ← genEncVar
26   let env''   = (varname, val):env'
27   pure (Dummy, env'')
28 -- the following two are essentially no-ops
29 evalEnclave (Gateway e) env = evalEnclave e env
30 evalEnclave (EnclaveApp e1 e2) env = do
31   (_, env1) ← evalEnclave e1 env
32   (_, env2) ← evalEnclave e2 env1
33   pure (Dummy, env2)
```

Figure 5. Operational Semantics of the Enclave

```
1  evalClient :: (MonadState StateVar m)
2             ⇒ Exp → ClientEnv → m (Value, ClientEnv)
3
4  -- evalClient for Lit, Var, Fun, Let, App, Plus not
5  -- shown as they behave the same as evalEnclave above
6  evalClient (InEnclave e) env = do
7    (_, env') ← evalClient e env
8    varname   ← genEncVar
9    let env'' = (varname, Dummy):env'
10   pure (SecureClosure varname [], env'')
11 evalClient (Gateway e) env = do
12   (e', env1) ← evalClient e env
13   case e' of
14     SecureClosure varname vals → do
15       enclaveEnv ← gets encState
16       let func = lookupVar varname enclaveEnv
17       case func of
18         Closure vars body encEnv → do
19           (res,enclaveEnv') ←
20             evalEnclave body ((zip vars vals) ++ encEnv)
21           pure (res, env1)
22         _ → pure (Err ENotClosure, env1)
23     _ → pure (Err ENotSecClos, env1)
24 evalClient (EnclaveApp e1 e2) env = do
25   (v1, env1) ← evalClient e1 env
26   (v2, env2) ← evalClient e2 env1
27   case v1 of
28    SecureClosure f args →
29     case v2 of
30      ArgList vals →
31           pure (SecureClosure f (args ++ vals), env2)
32       v → pure (SecureClosure f (args ++ [v]), env2)
33    v → pure (ArgList [v,v2], env2)
```

Figure 6. Operational Semantics of the Client

```
1 testProgram = let m = 3 in
2               let f = λ x -> x + m in
3               let y = inEnclave f in
4               gateway (y <@> 2)
```

Listing 3. A simple program for illustrating the operational semantics of HasTEE

$$m \longmapsto 3$$
$$f \longmapsto \mathit{Closure}\ ["x"]\ (x + m)\ [m \mapsto 3]$$
$$\mathit{EncVar}_0 \longmapsto \mathit{Closure}\ ["x"]\ (x + m)\ [m \mapsto 3]$$
$$y \longmapsto \mathit{Dummy}$$

(a) Enclave Environment

$$m \longmapsto 3$$
$$f \longmapsto \mathit{Closure}\ ["x"]\ (x + m)\ [m \mapsto 3]$$
$$\mathit{EncVar}_0 \longmapsto \mathit{Dummy}$$
$$y \longmapsto \mathit{SecureClosure}\ "\mathit{EncVar}_0"\ []$$

(b) Client Environment

Figure 7. (a) gets loaded during the first evaluator pass, and the Client Environment remains empty. In the second pass, (b) gets loaded while having access to the memory (a), as can be seen in Fig 6.

semantic evaluator operates in two passes. In the first pass, the `evalEnclave` evaluator from Fig. 5 is run. Fig. 7a shows the state of the enclave environment after the evaluator has completed evaluating Listing 3. Notably, the variable `y` maps to a value with no semantic meaning, as the evaluator is already running in the secure memory.

In the second pass, the environment from Fig. 7a is additionally passed as a state variable to the evaluator `evalClient` from Fig. 6. Note the different value mapped to the variable `y` in Fig 7b. `EnclaveApp` is evaluated on lines 25-34 in Fig 6. It generates the value `SecureClosure "`EncVar_0`" [Lit 2]`.

Notable is the evaluation of the `gateway` call on line 4 of Listing 3. The semantics for this evaluation are in lines 12-24 of Fig 6. The evaluator upon finding a reference EncVar_0 with no semantics in the client memory (Fig 7b) looks up EncVar_0 in the enclave environment (Fig 7a) and finds a `Closure` with a body. Crucially, **it evaluates the `Closure` by invoking the `evalEnclave` function on line 21 of Fig. 6 using the enclave environment.** This part models how the SGX hardware switches to the enclave memory when executing the secure function `f` rather than the client memory. An important point is generating an identical fresh variable name, EncVar_0, that the client uses to identify and call the functions in the enclave memory.

4.4 Practical security analysis

In what follows, we perform a security analysis of HasTEE. We start by making explicit that the only communication from the enclave back to the host client is primitive `gateway`. In this regard, we have the following claim capturing a (progress-insensitive [Askarov, Hunt, et al. 2008]) non-interference property. Intuitively, this property states that (side-effectful) programs do not leak information except via their termination behavior.

Proposition 4.1 (Non-interference)**.** *Given a HasTEE program* `p :: Enclave a -> App Done`*, where* `p` *does not use primitive* `gateway`*, and two enclave computations* `e1 :: Enclave a` *and* `e2 :: Enclave a`*, then* `p e1` *and* `p e2` *perform the same side-effects in the host client.*

This proposition states that in `p` the public effects on the host client cannot depend on the content of the argument of type `Enclave a`. The veracity of this proposition can be proven from the semantics of `gateway`, which is the only primitive calling `evalEnclave` from `evalClient` Fig. 6. If non-interference does not hold in the context of developing HasTEE, it could indicate the presence of vulnerabilities in the system. For example, it could suggest that data is being leaked into the host environment due to an error in the partitioning process of the HasTEE compiler. Alternatively, it might imply that certain side effects within the enclave are unintentionally revealing data back to the host, contrary to our expectations. Non-interference serves as an important initial security condition in the development of HasTEE as it helps identify and address numerous vulnerabilities that may arise during the process.

When it comes to reason about programs with the primitive `gateway`, we need to reason about IFC *declassification* primitives (or intended ways to release sensitive information) [Sabelfeld and Sands 2005] and how to avoid exploiting it to reveal more information than intended. Gollamudi and Chong [2016] utilizes *delimited release* as the security policy. This security policy extends information-flow control beyond non-interference. It allows for explicit points of controlled information release, called *escape hatches*, where sensitive information can be sent to public channels. This policy stipulates that information may only be released through escape hatches and no additional information is leaked. The function `gateway` is our escape hatch. If we apply delimited release to HasTEE, then host clients can always learn what the function `gateway e` returns, given that expression `e` evaluates to the same value in the initial states `st1, st2 :: Enclave a` given to a program `p`—a condition to avoid misusing escape hatches to reveal more information than intended. Our case studies (Section 6) satisfy delimited release.

Automatically enforcing delimited release or robust declassification [Myers, Sabelfeld, et al. 2004] imposes severe restrictions in either the information being declassified or how declassification primitives are used. Hence, we leave enforcing such security policies as future work. Instead, our DSL explicitly requires marking the points where information is sent back to the client (i.e., `gateway`), making it clear where to audit and control information leakages.

5 Implementation of HasTEE

5.1 Trusted GHC Runtime

One of the crucial challenges in implementing the HasTEE library is enabling Haskell programs to run within an Intel SGX enclave. All Haskell programs compiled via the Glasgow Haskell Compiler (GHC), rely on the GHC runtime [Marlow et al. 2009] for crucial operations such as memory allocation and management, concurrency, I/O management, etc. As such, it is essential to port the GHC runtime in order to run Haskell programs on the enclave.

The GHC runtime is a complex software that is heavily optimized for specific platforms, such as Linux and Windows, to maximize its performance. For instance, on Linux, the runtime relies on a wide variety of specialised low-level routines from a C standard library, such as `glibc` [GNUDevs 1991] or `musl` [Felker 2005], to provide essential facilities like memory allocation, concurrency, and more. The challenge lies in porting the runtime due to the limited and constrained implementation of the C standard library in the SGX SDK, called `tlibc` [Intel 2018]. Specifically, `tlibc` does not support some of the essential APIs required by the GHC runtime, including `mmap`, `madvise`, `munmap`, `select`, `poll`, a number of `pthread` APIs, operations related to timers, file reading, writing, and access control, and 100+ other functions.

Given the magnitude of engineering effort required to port the GHC runtime, we fall back on a library OS called Gramine [C. Tsai, Porter, et al. 2017]. Gramine internally intercepts all `libc` system calls within an application binary and maps them to a Platform Abstraction Layer (PAL) that utilizes a smaller ABI. In Gramine's case, this amounts to only 40 system calls that are executed through dynamic loading and runtime linking of a larger `libc` library, such as `glibc` or `musl`. Importantly, to protect the confidentiality and integrity of the enclave environment, Gramine uses a concept known as *shielded execution*, pioneered by the Haven system [Baumann et al. 2015], where a library is only loaded if its hash values are checked against a measurement taken at the time of initialisation. Shielded execution further protects applications against *Iago attacks* [Checkoway and Shacham 2013] in Gramine.

However, there are additional difficulties in loading the GHC runtime on the SGX enclave via Gramine. Owing to Gramine's diminished system ABI, it has a dummy or incomplete implementation for several important system calls that the runtime requires. For instance, the absence of the `select`, `pselect`, and `poll` functions, which are used in the GHC IO manager, required us to modify the GHC I/O manager to

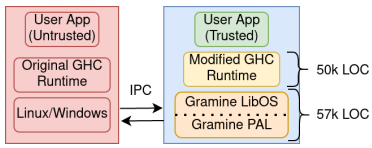


Figure 8. The high-level overview of communication between the untrusted and trusted parts of the app

manually manage the polling behavior through experimental heuristics. Similarly, the critical `mmap` operation in GHC uses specific flags (`MAP_ANONYMOUS`) that require modification. In addition, other calls, such as `madvise`, `getrusage`, and timer-based system calls, also require patching. We hope to quantify these modifications' performance in the future.

After the GHC runtime is loaded onto an enclave, communication between the untrusted and trusted parts of the application effectively occurs between two disjoint address spaces. Communication between them can happen over any binary interface, emulating a remote procedure call. Our early prototype stage implementation uses an inter-process communication (IPC) call to copy the serialised data (Fig 8). A production implementation should communicate via the C ABI using Haskell's Foreign Function Interface (FFI), as this would be significantly faster than an IPC.

The Gramine approach requires 57,000 additional lines of code in the Trusted Computing Base (TCB) [C. Tsai, Porter, et al. 2017]. However, this is still an improvement over traditional operating systems, like Linux, with a TCB size of 27.8 million lines of code [Larabel 2020].

5.2 HasTEE Library

The API of the HasTEE library was already shown (Figure 4) and discussed in Section 4.2. The principal data types, `Enclave` and `Client`, have been implemented as wrappers around the IO monad, as shown below:

```
newtype Enclave a = Enclave (IO a) -- data constructor not
    exported
type Client = IO
```

A key distinction is that the `Enclave` data type does not instantiate the `MonadIO` typeclass, as a result of which arbitrary IO actions cannot be lifted inside the `Enclave` monad. This is to ensure that the enclave does not perform leaky IO operations such as writing to the terminal. These are effectful operations that may leak information, which may not be rolled back. However, the Enclave monad does instantiate a `RestrictedIO` typeclass that will be discussed in the following section. The conditional-compilation-based partitioning technique is achieved by having dummy implementations of certain data types in one of the modules, while the concrete implementation of those types is defined in the second module. We give an example of this using two different data types from the API.

```
-- Enclave.hs
data Secure a = SecureDummy

type Ref a = IORef a
```

```
-- Client.hs
data Secure a =
  Secure CallID [ByteString]
type Ref a = RefDummy
```

A notable aspect of the API is the `Securable` typeclass, which constrains the `inEnclave` function and enables it to label functions with any number of arguments as residents of the enclave memory. The `Securable` typeclass accomplishes this using a well-known typeclass trick in Haskell, used to represent statically-typed variadic functions such as `printf` [Augustsson and Massey 2013]. In general, `Securable` characterises functions of the form $a_1 \rightarrow ... \rightarrow a_n \rightarrow \mathit{Enclave}\ b$.

The operational semantics presented in Section 4.3 should provide an intuition for the core implementation techniques used in the library. The complete HasTEE project has been open-sourced[1]. More implementation details can be found in the Haste.App paper [Ekblad and Claessen 2014].

5.3 Information Flow Control for Enclaves

The HasTEE library, being written in Haskell, allows using language-based Information Flow Control (IFC) techniques available in Haskell [Russo et al. 2008]. IFC approaches in Haskell aim to protect the confidentiality of data by encapsulating computations within a `Sec` monad. Typically, the monad employs a lattice of *labels* [Denning 1976] to model various security levels and then enforces policies on how data can flow between the levels. For a two-label lattice, where confidential data is marked with `H` and public data with `L`, a security policy known as *non-interference* is to prevent information flow from the secret to public channels [Goguen and Meseguer 1982]. In other words, $L \sqsubseteq L$, $H \sqsubseteq H$, $L \sqsubseteq H$, but $H \not\sqsubseteq L$, where $\sqsubseteq$ indicates the *flows to* relation.

A similar scenario arises in HasTEE, where the `Enclave` monad can be compared to a security-critical `Sec H` monad that attempts to prevent information leakage to a public `Sec L` channel represented by the `Client` monad. Enforcing the non-interference policy in this scenario would imply that no data can flow out of the `Enclave` monad to the `Client`, which would make the enclave very restrictive for any real-world use cases. As such, the IFC literature relaxes the non-interference policy by the means of *declassification* [Sabelfeld and Sands 2005], to allow controlled data leak from `H` to `L`.

In the HasTEE API, the `gateway :: (Binary a) => Secure (Enclave a) -> (Client a)` function is an *escape hatch* [Hedin and Sabelfeld 2012] that allows the enclave to leak *any* data to the client. We prioritise the usability of the API and trust that the enclave programmer will make the `gateway` call when they are certain they want to intentionally leak information to a public channel. However, there is a hidden line of defence in the `gateway` function. If the

[1] https://github.com/Abhiroop/EnclaveIFC

programmer wishes to send any user-defined data type to the untrusted client, they need to provide an instance of the `Binary` typeclass. Writing this typeclass instance for some confidential data type, such as a private key, equips the confidential data with the capacity to leave the enclave boundary, which should be done in a highly controlled manner.

Besides the `gateway` function, the `Enclave` monad has occasional requirements to interact with general I/O facilities like file reading/writing or random number generation. For such operations, the `Enclave` monad would need a `MonadIO` instance in Haskell to perform any I/O operations. However, as discussed in the previous section, we do not provide the lenient `MonadIO` instance to the Enclave monad but instead, use a `RestrictedIO` typeclass to limit the types of I/O operations that an `Enclave` monad can do.

`RestrictedIO`, shown in Listing 4, is a collection of typeclasses that constrains the variants of I/O operations possible inside an `Enclave` monad. For instance, if a programmer, through the usage of a malicious library, mistakenly attempts to leak confidential data through a network call, the typeclass would not allow this.

```
type RestrictedIO m = (EntropyIO m, UnsafeFileIO m) -- other
      typeclasses not shown

class EntropyIO (m :: Type -> Type) where
    type Entropy m:: Type
    genEntropyPool :: m (Entropy m)

class UnsafeFileIO (m :: Type -> Type) where
    untrustedReadFile  :: FilePath -> m (Untrusted String)
```

Listing 4. The Restricted IO typeclass

This approach is invasive in that it restricts how a library (malicious or otherwise) that interacts with a HasTEE program conducts I/O operations. For instance, we had to modify the HsPaillier library [L.-T. Tsai and Sarkar 2016] that used the `genEntropy` function for random number generation. Initially, the library could use the Haskell IO monad freely, but to interact with a package written in HasTEE, it had to be modified to use the more restricted type class constraint (`EntropyIO`) for its *effectful* operations. This limits potential malicious behaviour within the library. Notably, our changes involve only five lines of code that instantiate the type class and generalize the type signature of effectful operations.

Another aspect of IFC captured in our system is the notion of *endorsement* [Hedin and Sabelfeld 2012], which is the dual of declassification. Endorsement is concerned with the integrity, i.e., trustworthiness, of information. In HasTEE, we utilize endorsement to ensure that the integrity of secrets is not compromised by data being introduced into the enclave.

HasTEE allows file reading operations inside the `Enclave` monad, which can potentially corrupt the enclave's data integrity. To control this, HasTEE provides two forms of file reading operation - (1) untrusted file read and (2) trusted encrypted file reads. For (1), data read from untrusted files require manual endorsement via the `trust :: Untrusted a -> a` operator (where `Untrusted a` is a wrapper over the data read). This provides an additional check before untrusted data interacts with the trusted domain.

For point (2), HasTEE relies on an Intel SGX feature known as *sealing*. Every Intel SGX chip is embedded with a unique 128 bit key known as the Root Seal Key (RSK). The SGX enclave can use this RSK to encrypt trusted data that it wishes to persist on untrusted media. This process is known as sealing; HasTEE provides a simple interface to seal as well as unseal the trusted data being persisted, as shown below:

```
data SecurePath = SecurePath String

securefile :: FilePath -> SecurePath
securefile fp = "/secure_location/" <> fp -- path hidden

readSecure  :: SecurePath -> Enclave String
writeSecure :: SecurePath -> String -> Enclave ()
```

In the above, the `writeSecure` operation corresponds to *ciphertext declassification* [Askarov, Hedin, et al. 2008], while `readSecure` to an operation that applies automatic endorsement if the file can be decrypted successfully by the enclave RSK. If an attacker were to locate the secure location, the worst possible outcome would be the deletion of the file. However, the contents of the file cannot be read or modified outside the enclave, so the attacker would not be able to access the sensitive information stored within.

6 Case Studies

6.1 Federated Learning

Federated Learning is an emerging *privacy-preserving* machine learning [Al-Rubaie and Chang 2019] approach that allows multiple parties to train a model without sharing the raw training data. A typical federated learning setup involves multiple decentralized edge devices holding local datasets, training a model locally and then aggregating the trained model on a cloud server. Fig. 9 shows the desired setup.

The setup in Fig. 9 above is facilitated by a combination of TEEs and homomorphic encryption. Homomorphic Encryption (HE) [Gentry 2009] is a form of encryption that enables direct computation on encrypted data, revealing the computation result only to the decryption key owner. We emulate the very same setup for our case study where we have two mutually distrusting parties -

- **Confidential data owner.** This party wants to protect its confidential data. A real-life example would be a hospital containing confidential patient data.
- **ML model owner.** This party wants to protect their intellectual property (the ML model) from the data owners as well as the cloud provider. They encrypt their model when sending it to the data owners and allows them to use only homomorphic encryption for operating on the model.

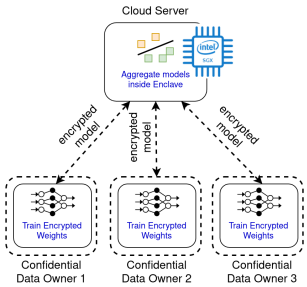


Figure 9. A Federated Learning setup where the data owners are protecting their data and the ML model owner is protecting their model. The training with encrypted weights can be done using homomorphic encryption.

```
1 data SrvSt =
2   SrvSt { publicKey :: PubKey, privateKey :: PrvKey
3         , updWts :: Vector Double, numClients :: Int
4         , wtsDict :: Map Epoch [Vector CipherText]) }
```

Listing 5. The Federated Learning server state

The above setup only requires the cloud server supporting Intel SGX technology so that even mobile devices can participate in training as a worker role. We can very conveniently model this entire setup as three clients and a server with an enclave in HasTEE. For illustration purposes, we will use GHC's threads to represent the three clients instead of three separate data owner machines.

Listing 5 models the server's state. Note that the weights are kept in plaintext form. The enclave state holds both its public and private keys. However, only the public key should be allowed to move to the client. We enforce this by not providing an instance of the `Binary` typeclass for the private key. If untrusted modules try to attack such enforcement by adding new instances to `Binary`, or even providing overlapping ones to override the behaviour of overloaded methods, then Safe Haskell [Terei et al. 2012] will indicate GHC to not compile the code. Haskell is unique in terms of having an extension like Safe Haskell. Safe Haskell enforces sandboxing for trusted code by banning extensions that introduce *loopholes* and compromise type-safety or module abstraction (often for the sake of performance). As discussed in Section 5.3, the lack of a `Binary` instance for the `privateKey` will prevent the enclave programmer from accidentally leaking the security-critical private key.

Listing 6 shows the API exposed to the client machine. Instead of the complex `SGX_ECALL` machinery, our API is expressed in idiomatic Haskell. Calling any function `f` from the record `api` with an argument `arg` in this API is expressed simply as `gateway ((f api) <@> arg)`.

```
1 type Accuracy = Double
2 type Loss     = Double
3 data API = API {
4  aggregateModel :: Secure (Epoch -> Vector CipherText ->
       Enclave (Maybe (Vector CipherText))),
5  validateModel :: Secure (Enclave (Accuracy, Loss)),
6  getPublicKey :: Secure (Enclave PubKey),
7  reEncrypt :: Secure (CipherText -> Enclave CipherText)}
```

Listing 6. The Federated Learning client API

Listing 7 shows the main ML model training loop. A few functions have been elided for brevity, but the key portions of the client-server interaction in HasTEE should be visible. The `Config` type holds the state containing encrypted weights sent from the cloud server, the learning rate, the current epoch number and the public key. After each epoch it updates the weights to the new aggregated value (Line 12). The value `x'` is the data set that the data owners are protecting and `y` is the result of the learning algorithm. The `adjustModelWithLearningRate` function (body elided, line 6) takes the computed gradient (line 5) and tries to converge on the desired result.

On line 7 the server is communicated to aggregate models spread across different clients, with the server returning the encrypted updated weights `wt'`. We use a wrapper over `gateway`, called `retryOnEnclave` (body elided), to allow the server to move in lock step with all the clients. Then in line 8, the server is communicated again to collect the accuracy and loss in the ongoing epoch number, which gets displayed in line 9. Finally, the loop continues in line 10.

```
1 handleSingleEpoch :: API -> CurrentEpochNum -> MaxEpochNum
      -> Matrix Double -> Vector Int -> Config -> Client
      Config
2 handleSingleEpoch api n m x' y cfg'
3   | n == m    = return cfg'
4   | otherwise = do
5     grad   <- computeGradient api cfg' x' y
6     cfgNew <- adjustModelWithLearningRate api
7                                   (cfg' { iterN = n }) grad
8     wt' <- retryOnEnclave $ (aggregateModel api) <@> n
9                                       <@> (weights cfgNew)
10    (acc, loss) <- gateway (validateModel api)
11    printClient $ " Iteration no: " <> show n <>
12      " Accuracy: "  <> show acc <> " Loss : " <> show loss
13    handleSingleEpoch api (n+1) m x' y
14                                  (cfgNew { weights = wt' })
```

Listing 7. The key model training loop

Listing 7 above features a complex control flow with at least two interactions visible in the loop itself. Internally, `computeGradient` and `adjustModelWithLearning` both talk to the enclave, calling the `reEncrypt` function to remove noise from the homomorphic encryption operation. HasTEE can represent a fairly complex, asynchronous control flow as simple Haskell function calls.

In terms of Information Flow Control, there are two important aspects in this case study. Firstly, the `RestrictedIO` typeclass constrains potentially malicious libraries from misbehaving. For example, consider the library HsPaillier [L.-T. Tsai and Sarkar 2016], which implements the Paillier Cryptosystem [Paillier 1999] for partial homomorphic encryption. All effectful operations from this library, such as `genKey :: Int -> IO (PubKey, PrvKey)`, need to be rewritten for them to be usable within the `Enclave` monad. The following snippet shows our typeclass instantiation and a sample type signature change needed inside the library.

```
instance (IO ~ m) => EntropyIO m where
    type Entropy m = EntropyPool
    genEntropyPool = createEntropyPool

-- genKey :: Int -> IO (PubKey, PrvKey) -- original type
genKey :: (Monad m, EntropyIO m)=> Int -> m (PubKey, PrvKey)
```

The second aspect of IFC arises when the client machine queries the server for accuracy and loss by asking it to validate the model. On the server side, the enclave has to read a file with test data. This test data resides outside of the enclave and is potentially an attack vector. In order to not inadvertently trust such an exposed source, the enclave uses the `untrustedReadFile` function from the `RestrictedIO` typeclass (Listing 4). The file is read as an `Untrusted String` and requires explicit programmer *endorsement* via the `trust` operator for the compiler to typecheck the program.

Overall the case study constitutes only 500 lines of code. It naturally fits into the client-server programming model, and the usage of Haskell provides type safety and enables IFC-based security.

6.2 Encrypted Password Wallet

For this case study, we use HasTEE to implement a secure password wallet that stores authentication tokens in encrypted form on the disk. An authentication token can be retrieved from the wallet if the right master password is supplied. The definition of a password wallet used by the case study follows in Listing 8.

```
-- | A single entry of authentication tokens
data Item = Item { title :: String, username :: String,
     password :: Password } deriving (Show, Read)
-- | The secure wallet
data Wallet = Wallet { items :: [Item], size :: Int,
     masterPassword :: Password} deriving (Show, Read)
```

Listing 8. The definition of a password wallet as a regular Haskell data type.

The `Show` and `Read` instances are used to convert a wallet to and from a string. This allows us to write the wallet to disk, and by writing to a secure file path we ensure that the stored wallet is encrypted, as described in section 5.3. By omitting a `Binary` instance we ensure that the wallet is not inadvertently leaked to the client directly. The code

```
-- | Secure file path to the wallet
wallet :: SecureFilePath
wallet = secureFile "wallet.seal"

-- | Try to load the secure wallet into the enclave
loadWallet :: Enclave (Maybe Wallet)
loadWallet = do b <- doesSecureFileExist wallet
                if b then do contents <- readSecure wallet
                             return $ readMaybe contents
                     else return Nothing

-- | Store the wallet on disk in encrypted form
saveWallet :: Wallet -> Enclave ReturnCode
saveWallet w = writeSecure wallet (show w) >> return Success
```

Listing 9. The code that storing and loading the encrypted wallet. Programmer do not need to manage encryption keys.

in Listing 9 implements the functions that store and load the wallet. We emphasize that the code does not need to explicitly reason about encryption and decryption, except for defining the secure file path.

Our password wallet has the following features - (1) adding an authentication token, (2) retrieving a password, (3) deleting a token and (4) changing the master password. It is designed as a command-line utility where the commands are handled by an untrusted client and the passwords are protected by the enclave. The complete implementation is roughly 200 lines of Haskell code.

The hardware-enforced security provided by our secure wallet makes it a natural fit for designing wallets that are protected by biometrics. A similar approach is used on modern iPhones, where passwords are stored in a secure enclave [Apple 2021] to ensure confidentiality, and the user's biometric data is used as the master password. In our case, the usage of a high-level language like Haskell enables expressing this relatively complex application concisely.

6.3 Data Clean Room with Differential Privacy

A *Data Clean Room* (DCR) [AWS 2022] is a technology that provides aggregated and anonymised user information to protect user privacy while providing advertisers and analytic firms with non-personally identifiable information to target a specific demographic with advertising campaigns and analytics-based services.

DCRs compute and release aggregated results based on the user data. To prevent attackers from compromising individual user information from aggregate data (via statistical techniques), DCRs employ *differential privacy* [Dwork 2006]. Differential privacy adds calibrated noise to the aggregate data making it computationally hard for attackers to compromise individual data. The noise calibration can be adjusted for increased privacy (more noise) or increased accuracy (less noise).

Our third case study implements a *differentially-private* DCR within an SGX enclave using HasTEE. The DCR consists of record, `User`, containing fields such as `name, occupation, salary, gender, age, etc.` The `User` record is encrypted before being provisioned to the DCR, after which we use the *Laplace Mechanism* [Dwork and Roth 2014] when performing counting queries to add noise to the result. The mechanism introduces noise by sampling a Laplace distribution. The code implementing the Laplace mechanism can be found in Appendix B.

The DCR does not provide a `Binary` instance for the `User` record to ensure that it is not transferred to the enclave via plain serialisation. Instead, we expose functions that encrypt and decrypt users.

The Laplace Mechanism for adding noise requires a source of randomness. Here, we use Haskell's `System.Random` package, which internally reads from `/dev/urandom`. For production environments, a more cryptographically secure source of randomness is required. We extend the `RestrictedIO` (Section 5.3) interface to allow this operation as long as the programmer *endorses* the file read.

Consider a sample query to test how many individuals in a data set have a salary within a specific range.

```
1 salaryWithin :: Integer -> Integer -> User -> Bool
2 salaryWithin l h u = l <= salary u && salary u <= h
```

The HasTEE code for the DCR executing this query is shown in Listing 10. Lines 3 to 8 specify the API of the data clean room. The DCR's API supports (1) initialisation, (2) fetching of the public key, (3) provisioning user data to the enclave, and (4) executing the salary query. Line 8 is used to generate some arbitrary users (for testing), after which the client code takes over. The client initializes the DCR and fetches its public key. After this, the users are encrypted and sent to the DCR. On line 15 the salary query is executed in the DCR, and then the result is printed.

Generating arbitrary users to test the setup is done purely for illustration purposes. In a more faithful implementation, the client would relay the public key to data owners that would then send already encrypted user records to the client, which provisions them to the DCR. Owing to HasTEE's client-server programming model and the use of a high-level language like Haskell, the implementation becomes very compact with roughly 200 LOC.

7 Evaluations

7.1 Discussion

In contrast to development on the Intel C/C++ SGX SDK, HasTEE's high-level programming model entirely abstracts away the complexity of dealing with the low-level `edl` files in the SGX SDK. The remote procedure calls that happen between the untrusted client and trusted enclave are typechecked in Haskell, unlike the SGX SDK. The benefits of high-level of abstraction can also be seen in the password wallet example,

```
1  app :: App Done
2  app = do
3      ref    <- liftNewRef undefined
4      initSt <- inEnclave $ initEnclave ref 0.1
5      pkey   <- inEnclave $ getPublicKey ref
6      prov'  <- inEnclave $ provisionUserEnclave ref
7      lm     <- inEnclave $ laplaceMechanism ref $
8                                  salaryWithin 10000 50000
9      dataset <- liftIO $ sequence $ replicate 500
10                                     (generate arbitrary)
11     runClient $ do
12         gateway $ initSt      -- initialize enclave state
13         key <- gateway pkey   -- enclaves public key
14         mapM_ (\u -> do ct <- encryptUser u key
15                         gateway $ prov' <@> ct) dataset
16                               -- provision users
17         result <- gateway lm  -- run the salary query
18         liftIO $ putStrLn $ concat ["res: ", show result]
```

Listing 10. The client running the salaryWithin query over the data set in the data clean room.

where functions `readSecure` and `writeSecure` (Listing 9) relieves developers from the burden of key management. Furthermore, HasTEE warns a program against accidental data leaks and can enforce stronger compile-time guarantees than Intel C/C++ SGX SDK. For instance, in all three case studies, the lack of the `Binary` type-class constraint would, by construction, prevent accidental leakage of the secret data from the enclave. All three case studies restrict the I/O operations possible in the `Enclave` monad by the type-class `RestrictedIO`. Notably, in the federated learning example, we adapted the homomorphic encryption library to limit the effects possible in the `IO` monad.

7.2 Performance Evaluations

Our evaluations were conducted on an Azure Standard DC1s v2 (1 vcpu, 4 GiB memory) SGX machine. We use the password wallet case study as the canonical example to present performance evaluations across different parameters. We chose this example as it covers all the major aspects of the HasTEE API, such as protecting the confidentiality of data across the memory as well as the disk.

Memory Overhead. We show the memory consumption of our modified GHC runtime, sampled across 100 runs, where a sample was collected every second.

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

Although the memory usage of HasTEE will certainly vary across applications, these numbers provide a general estimate of the trusted GHC runtime's space usage. The Resident Set Size (RSS) indicates that the application fits within 20 MB at peak usage. RSS is an overestimate of memory usage

as it includes the memory occupied by shared libraries as well. As a result, we can be certain that our application fits within the Enclave Page Cache limit (Section 2) of 93 MB.

Latency. We measure the latency and throughput for an instance of password retrieval, that includes - (i) an enclave crossing to call the trusted runtime, (ii) standard GHC execution time, (iii) encrypted file load, (iv) file decryption, (v) file read, and (v) a second enclave crossing to return the result.

Our measurements show that using the Linux `send/recv` call for enclave crossing results in a *60 milliseconds overall latency*. As our current socket-based communication is a proof-of-concept, it incurs a substantial overhead compared to native SGX enclave crossings. As a baseline, we measured the latency of an encrypted password retrieval in unmodified GHC (file encrypted with gpg [GNU 1999]). The baseline number comes out to be 0.6 milliseconds showing an overall 100x slowdown. Note that an average SGX ECALL operation incurs at least a 10x slowdown via the native SDK [Ghosn et al. 2019]. We believe switching to native ECALLs has the potential to improve our latencies.

Throughput. In terms of throughput, HasTEE is able to handle on average 11 requests for password retrieval per second. Again, this number has the potential for further improvemnt by switching to native SGX ECALLs.

We currently present coarse-grained measurements of the various metrics but envision future work, where more fine-grained parameters, such as the correlation between the GC pauses across the two runtimes can be presented. Appendix 7.3 provides a qualitative comparison of HasTEE against GoTEE and J_E.

7.3 Comparing HasTEE to GoTEE and J_E

Table 1 presents a comparison between HasTEE and its two closest counterparts - GoTEE [Ghosn et al. 2019] and J_E [Oak et al. 2021]. While both GoTEE and J_E had to modify the respective compilers, HasTEE required no modifications to the compiler. The specific runtime used by J_E is not mentioned in the paper [Oak et al. 2021]; however, it suggests that no modification of the runtime was required, as it was run on a large virtualized host - SGX-LKL [Priebe et al. 2019]. In contrast, the runtimes for HasTEE and GoTEE required modification. GoTEE required significant modifications to the Golang runtime system to enable communication between the trusted and untrusted memory.

Both GoTEE and J_E use sophisticated static analysis passes and program transformations to partition a program into its two components. In contrast, HasTEE's conditional compilation-based approach is much simpler, which is beneficial when it comes to security. Having less and simpler code makes it easier to verify for correctness. Notably, the purity of Haskell enables the user to inspect the type of a function and infer that it is naturally confined whenever a function is side-effect free. Inferring the confinement of a pure function is much more challenging in imperative languages like Java and Go.

8 Related Work

Managed programming languages. While there are imperative and object-oriented languages with TEE support (e.g., Go-based [Ghosn et al. 2019], and Java-based[Oak et al. 2021; C. Tsai, Son, et al. 2020], HasTEE is (to the best of our knowledge) the first functional language running on a TEE environment. The Rust-SGX [Wang et al. 2019] project provides foreign-function interface (FFI) bindings to the C/C++ Intel SGX SDK. Different from HasTEE, Rust-SGX does not aim to introduce any programming model or IFC to protect against leakage of sensitive data. Instead, Rust-SGX's main goal is application-level memory safety when programming with the low-level SGX SDK. HasTEE provides memory safety by the virtue of running Haskell, a memory-safe language, on the enclaves. TrustJS [Goltzsche et al. 2017] takes a similar FFI-based approach as Rust-SGX for programming enclaves with JavaScript. An important project in this space is the WebAssembly (WASM) initiative [Rossberg 2019]. There have been WASM projects, both academic, such as Twine [Ménétrey et al. 2021], as well as commercial, such as Enarx [Red Hat 2019], aimed at allowing WASM runtimes to operate within SGX enclaves. Our initial approach was to use the experimental Haskell WASM backend [Tweag.io 2022] to run Haskell on SGX enclaves. However, the aforementioned runtimes are not supported by GHC and lack several key features required for loading Haskell onto an enclave.

Automatic partitioning. HasTEE has a seamless program partitioning and familiar client-server-based programming model for enclaves. HasTEE's lightweight partitioning approach is inspired by the Haste.App library [Ekblad and Claessen 2014]—a library to write web applications in Haskell and deploy parts of it into JavaScript on the web browser. The most well-known automatic partitioning tool for C programs on an SGX enclave is Glamdring [Lind et al. 2017]. The general idea of partitioning a single program has been studied as multitier programming [Weisenburger et al. 2021]. Among the existing approaches to multitier programming, HasTEE provides a lightweight alternative that does not require any compiler modification or elaborate dataflow analysis to partition the program.

Application development. There have been attempts to virtualize entire platforms within the enclave memory to reduce the burden of dealing with the two-project programming model of Intel SGX. Haven [Baumann et al. 2015] virtualizes the entire Windows operating system as well as an entire SQL server application running on top of it. SCONE [Arnautov et al. 2016] virtualizes a Docker container instance within an SGX enclave. The libraryOS Gramine [C. Tsai, Porter, et al. 2017], which is used in this work, is an example of lightweight virtualization.

AMD's TEE system, AMD SEV [AMD 2018], is natively a virtualization-based approach. While it eases development, virtualization can result in drastically increasing the size of

Framework	HasTEE	GoTEE	J_E
IFC support	Standard declassification	None	Robust declassification
Partitioning scheme	Type-based	Process-based	Annotation-based
Modified compiler	No	Yes	Yes
Modified runtime system	Yes	Yes	No
Trusted Components	GHC compiler, GHC runtime, Gramine	GoTEE compiler, GoTEE runtime	Java parser and partitioner, Jif compiler, JVM, SGX-LKL[Priebe et al. 2019]
Programming model	Client-server	Synchronous Message-Passing	Using the object-framework provided by Java

Table 1. Comparison of HasTEE, GoTee, and J_E. We specify the core components involved in the Trusted Computing Base in all three frameworks.

the TCB. We chose to apply a libraryOS approach for HasTEE in order to have a TCB of 57k lines of code (Gramine). As a future work, we can move away from Gramine and make the GHC runtime a standalone library inside the SGX enclave.

Information Flow Control. HasTEE draws inspiration from the work on static IFC security libraries (e.g., [Buiras et al. 2015; Russo 2015; Russo et al. 2008]). Such approaches relies on the purity of Haskell to detect and stop malicious behaviour. HasTEE can support IFC in a dynamic manner [Stefan et al. 2011] by adapting the interpretation of the `Enclave` type to be a runtime monitor rather than just a wrapper for IO, where `gateway` performs security checks when sending/receiving information—an interesting direction for future work.

The work on IMP_E [Visser and Smaragdakis 2016] studies IFC non-interference for passive and active attackers on the host client. Gollamudi, Chong, and Arden [2019] present a calculus for reasoning about IFC for applications distributed across several enclaves. J_E [Oak et al. 2021] studies how compromised host clients can abuse gateway (declassification) primitives. Their security property and enforcement is based on the notion of robust declassification [Myers, Sabelfeld, et al. 2004; Waye et al. 2015]. Intuitively, this policy ensures that low-integrity data cannot influence the declassification of secret data. HasTEE enforces a simpler IFC policy for passive attackers—along the lines of Visser and Smaragdakis [2016]—and defer automatic analyses of the use of `gateway` for future work. Another interesting line of work is Moat [Sinha et al. 2015], which formally verifies enclave programs running on Intel SGX such that data confidentiality is respected. It uses IFC to enforce the policies and automated theorem proving to verify the policy enforcement mechanism.

9 Conclusion & Future Work

This paper presents HasTEE, a domain-specific language to write TEE programs while ensuring confidentiality of data by construction. Unlike previous work, HasTEE provides its partitioning of source code and IFC as a library! For HasTEE to work, we ported GHC's runtime to run within SGX enclaves by using the libraryOS Gramine. We demonstrate through three diverse case studies how HasTEE's IFC mechanism can help prevent accidental data leakage while producing concise code. We hope HasTEE opens future research avenues at the intersection of TEEs and functional languages.

There are several directions for future work. The IFC scheme we consider operates on two security levels - sensitive (`Enclave`) and public (`Client`) data. A natural extension is to enable multiple security levels [Myers and Liskov 2000; Stefan et al. 2011] to represent the concerns of different principals contributing data to enclaves. TEEs also provide a verifiable launch of the execution environment for the sensitive code and data, enabling a remote entity to ensure that it was set up correctly. *Remote attestation* [Knauth et al. 2018] allows an SGX enclave to prove its identity to a challenger using the private key embedded in the enclave. HasTEE does not capture attestation at the programming language level since it a property of the system components layout. Nevertheless, remote attestation can facilitate secure communication between multiple enclaves, e.g., a distributed-enclave setting; so it would be interesting to incorporate language-level support for remote attestation. Finally, GHC runtime is extensively optimized for performance. Obtaining a more compact and portable runtime, e.g., by using a restricted set of `libc` operations, could result in a considerably smaller TCB. A more portable runtime would facilitate HasTEE experiments on other TEE infrastructures such as ARM TrustZone and RISC-V PMP [RISC-V 2017].

Acknowledgements

This work was funded by the Swedish Foundation for Strategic Research(SSF) under the project Octopi (Ref. RIT17-0023) and VR. Special thanks to Mary Sheeran, Marco Vassena, Bo Joel Svensson, and Claudio Agustin Mista for their initial reviews, and Facundo Dominguez and Michael Sperber for guiding the published draft.

References

Mohammad Al-Rubaie and J. Morris Chang. 2019. "Privacy-Preserving Machine Learning: Threats and Solutions." *IEEE Secur. Priv.*, 17, 2, 49–58. DOI: 10.1109/MSEC.2018.2888775.

AMD. 2018. *AMD.* https://developer.amd.com/sev/. (accessed: 16.02.2023).

Apple. 2021. *Apple Platform Security.* https://support.apple.com/guide/security/face-id-and-touch-id-security-sec067eb0c9e/web. (accessed: 16.02.2023).

ARM. 2004. *ARM TrustZone.* https://www.arm.com/technologies/trustzone-for-cortex-a. (accessed: 16.02.2023).

Sergei Arnautov et al.. 2016. "SCONE: Secure Linux Containers with Intel SGX." In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.* Ed. by Kimberly Keeton and Timothy Roscoe. USENIX Association, 689–703. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov.

Aslan Askarov, Daniel Hedin, and Andrei Sabelfeld. 2008. "Cryptographically-masked flows." *Theor. Comput. Sci.*, 402, 2-3, 82–101. DOI: 10.1016/j.tcs.2008.04.028.

Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. 2008. "Termination-Insensitive Noninterference Leaks More Than Just a Bit." In: *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings* (Lecture Notes in Computer Science). Ed. by Sushil Jajodia and Javier López. Vol. 5283. Springer, 333–348. ISBN: 978-3-540-88312-8. DOI: 10.1007/978-3-540-88313-5_22.

Lennart Augustsson and Bart Massey. 2013. *Text.Printf - printf in Haskell.* https://hackage.haskell.org/package/base-4.17.0.0/docs/Text-Printf.html#t:PrintfType. (accessed: 16.02.2023).

Amazon AWS. 2022. *AWS Clean Room.* https://aws.amazon.com/clean-rooms/. (accessed: 16.02.2023).

Zijian Bao, Qinghao Wang, Wenbo Shi, Lei Wang, Hong Lei, and Bangdao Chen. 2020. "When Blockchain Meets SGX: An Overview, Challenges, and Open Issues." *IEEE Access*, 8, 170404–170420. DOI: 10.1109/ACCESS.2020.3024254.

Andrew Baumann, Marcus Peinado, and Galen C. Hunt. 2015. "Shielding Applications from an Untrusted Cloud with Haven." *ACM Trans. Comput. Syst.*, 33, 3, 8:1–8:26. DOI: 10.1145/2799647.

Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. 2015. "HLIO: mixing static and dynamic typing for information-flow control in Haskell." In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015.* Ed. by Kathleen Fisher and John H. Reppy. ACM, 289–301. DOI: 10.1145/2784731.2784758.

Stephen Checkoway and Hovav Shacham. 2013. "Iago attacks: why the system call API is a bad untrusted RPC interface." In: *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013.* Ed. by Vivek Sarkar and Rastislav Bodik. ACM, 253–264. DOI: 10.1145/2451116.2451145.

Decentriq. 2022. *Decentriq.* https://blog-decentriq-com.cdn.ampproject.org/c/s/blog.decentriq.com/swiss-cheese-to-cheddar-securing-amd-sev-snp-early-boot-2. (accessed: 16.02.2023).

Dorothy E. Denning. 1976. "A Lattice Model of Secure Information Flow." *Commun. ACM*, 19, 5, 236–243. DOI: 10.1145/360051.360056.

Cynthia Dwork. 2006. "Differential Privacy." In: *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II* (Lecture Notes in Computer Science). Ed. by Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener. Vol. 4052. Springer, 1–12. DOI: 10.1007/11787006_1.

Cynthia Dwork and Aaron Roth. 2014. "The Algorithmic Foundations of Differential Privacy." *Found. Trends Theor. Comput. Sci.*, 9, 3-4, 211–407. DOI: 10.1561/0400000042.

Anton Ekblad and Koen Claessen. 2014. "A seamless, client-centric programming model for type safe web applications." In: *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014.* Ed. by Wouter Swierstra. ACM, 79–89. DOI: 10.1145/2633357.2633367.

Richard Felker. 2005. *musl C Library.* https://musl.libc.org/. (accessed: 16.02.2023).

Craig Gentry. 2009. "Fully homomorphic encryption using ideal lattices." In: *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009.* Ed. by Michael Mitzenmacher. ACM, 169–178. DOI: 10.1145/1536414.1536440.

Adrien Ghosn, James R. Larus, and Edouard Bugnion. 2019. "Secured Routines: Language-based Construction of Trusted Execution Environments." In: *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019.* Ed. by Dahlia Malkhi and Dan Tsafrir. USENIX Association, 571–586. https://www.usenix.org/conference/atc19/presentation/ghosn.

GNU. 1999. *GNU Privacy Guard.* https://gnupg.org/. (accessed: 16.02.2023).

GNUDevs. 1991. *glibc - The GNU C Library.* https://www.gnu.org/software/libc/. (accessed: 16.02.2023).

Joseph A. Goguen and José Meseguer. 1982. "Security Policies and Security Models." In: *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982.* IEEE Computer Society, 11–20. DOI: 10.1109/SP.1982.10014.

Anitha Gollamudi and Stephen Chong. 2016. "Automatic enforcement of expressive security policies using enclaves." In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016.* Ed. by Eelco Visser and Yannis Smaragdakis. ACM, 494–513. ISBN: 978-1-4503-4444-9. DOI: 10.1145/2983990.2984002.

Anitha Gollamudi, Stephen Chong, and Owen Arden. 2019. "Information Flow Control for Distributed Trusted Execution Environments." In: *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019.* IEEE, 304–318. ISBN: 978-1-7281-1407-1. DOI: 10.1109/CSF.2019.00028.

David Goltzsche, Colin Wulf, Divya Muthukumaran, Konrad Rieck, Peter R. Pietzuch, and Rüdiger Kapitza. 2017. "TrustJS: Trusted Client-side Execution of JavaScript." In: *Proceedings of the 10th European Workshop on Systems Security, EUROSEC 2017, Belgrade, Serbia, April 23, 2017.* Ed. by Cristiano Giuffrida and Angelos Stavrou. ACM, 7:1–7:6. DOI: 10.1145/3065913.3065917.

Daniel Hedin and Andrei Sabelfeld. 2012. "A Perspective on Information-Flow Control." In: *Software Safety and Security - Tools for Analysis and Verification.* NATO Science for Peace and Security Series - D: Information and Communication Security. Vol. 33. Ed. by Tobias Nipkow, Orna Grumberg, and Benedikt Hauptmann. IOS Press, 319–347. DOI: 10.3233/978-1-61499-028-4-319.

Intel. 2015. *Intel Software Guard Extension.* https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html. (accessed: 16.02.2023).

Intel. 2018. *tlibc - an alternative to glibc.* https://github.com/intel/linux-sgx/tree/master/common/inc/tlibc. (accessed: 16.02.2023).

SL Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. 1993. "The Glasgow Haskell compiler: a technical overview." In: *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference.* Vol. 93.

Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. 2018. "Integrating Remote Attestation with Transport Layer Security." *CoRR*, abs/1801.05863. http://arxiv.org/abs/1801.05863 arXiv: 1801.05863.

Michael Larabel. 2020. *Linux Stats.* https://www.phoronix.com/news/Linux-Git-Stats-EOY2019. (accessed: 16.02.2023).

Christian M. Lesjak, Daniel M. Hein, and Johannes Winter. 2015. "Hardware-security technologies for industrial IoT: TrustZone and security controller." In: *IECON 2015 - 41st Annual Conference of the IEEE Industrial*

Electronics Society, Yokohama, Japan, November 9-12, 2015. IEEE, 2589–2595. DOI: 10.1109/IECON.2015.7392493.

Joshua Lind et al.. 2017. "Glamdring: Automatic Application Partitioning for Intel SGX." In: *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. Ed. by Dilma Da Silva and Bryan Ford. USENIX Association, 285–298. https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind.

Simon Marlow, Simon L. Peyton Jones, and Satnam Singh. 2009. "Runtime support for multicore Haskell." In: *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*. Ed. by Graham Hutton and Andrew P. Tolmach. ACM, 65–78. DOI: 10.1145/1596550.1596563.

Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. 2021. "Twine: An Embedded Trusted Runtime for WebAssembly." In: *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 205–216. DOI: 10.1109/ICDE51399.2021.00025.

Microsoft. 1994. *glibc - The GNU C Library*. C%20runtime%20(CRT)%20and%20C++%20standard%20library%20(STL). (accessed: 16.02.2023).

Dominic P. Mulligan, Gustavo Petri, Nick Spinale, Gareth Stockwell, and Hugo J. M. Vincent. 2021. "Confidential Computing - a brave new world." In: *2021 International Symposium on Secure and Private Execution Environment Design (SEED), Washington, DC, USA, September 20-21, 2021*. IEEE, 132–138. DOI: 10.1109/SEED51797.2021.00025.

Antonio Muñoz, Ruben Rios, Rodrigo Román, and Javier López. 2023. "A survey on the (in)security of trusted execution environments." *Comput. Secur.*, 129, 103180. DOI: 10.1016/j.cose.2023.103180.

Andrew C. Myers and Barbara Liskov. 2000. "Protecting privacy using the decentralized label model." *ACM Trans. Softw. Eng. Methodol.*, 9, 4, 410–442. DOI: 10.1145/363516.363526.

Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. 2004. "Enforcing Robust Declassification." In: *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*. IEEE Computer Society, 172–186. DOI: 10.1109/CSFW.2004.9.

Aditya Oak, Amir M. Ahmadian, Musard Balliu, and Guido Salvaneschi. 2021. "Language Support for Secure Software Development with Enclaves." In: *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*. IEEE, 1–16. DOI: 10.1109/CSF51468.2021.00037.

Pascal Paillier. 1999. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes." In: *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding* (Lecture Notes in Computer Science). Ed. by Jacques Stern. Vol. 1592. Springer, 223–238. DOI: 10.1007/3-540-48910-X_16.

Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A. Sartakov, and Peter R. Pietzuch. 2019. "SGX-LKL: Securing the Host OS Interface for Trusted Execution." *CoRR*, abs/1908.11143. http://arxiv.org/abs/1908.11143 arXiv: 1908.11143.

Inc. Red Hat. 2019. *Enarx: Confidential Computing with WebAssembly*. https://enarx.dev/. (accessed: 16.02.2023).

RISC-V. 2017. *RISC-V: Physical Memory Protection*. https://riscv.org/blog/2022/04/xuantie-virtualzone-risc-v-based-security-extensions-xuan-jian-alibaba/. (accessed: 16.02.2023).

Andreas Rossberg. Dec. 5, 2019. *WebAssembly Core Specification*. W3C, (Dec. 5, 2019). https://www.w3.org/TR/wasm-core-1/.

Alejandro Russo. 2015. "Functional pearl: two can keep a secret, if one of them uses Haskell." In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. Ed. by Kathleen Fisher and John H. Reppy. ACM, 280–288. DOI: 10.1145/2784731.2784756.

Alejandro Russo, Koen Claessen, and John Hughes. 2008. "A library for light-weight information-flow security in Haskell." In: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*. Ed. by Andy Gill. ACM, 13–24. DOI: 10.1145/1411286.1411289.

Andrei Sabelfeld and David Sands. 2005. "Dimensions and Principles of Declassification." In: *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005), 20-22 June 2005, Aix-en-Provence, France*. IEEE Computer Society, 255–269. DOI: 10.1109/CSFW.2005.15.

Stephan van Schaik et al.. 2022. "SoK: SGX.Fail: How Stuff Get eXposed." In.

Moritz Schneider, Ramya Jayaram Masti, Shweta Shinde, Srdjan Capkun, and Ronald Perez. 2022. "SoK: Hardware-supported Trusted Execution Environments." *CoRR*, abs/2205.12742. arXiv: 2205.12742. DOI: 10.48550/arXiv.2205.12742.

Hovav Shacham. 2007. "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)." In: *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*. Ed. by Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson. ACM, 552–561. DOI: 10.1145/1315245.1315313.

Rohit Sinha, Sriram K. Rajamani, Sanjit A. Seshia, and Kapil Vaswani. 2015. "Moat: Verifying Confidentiality of Enclave Programs." In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel. ACM, 1169–1184. DOI: 10.1145/2810103.2813608.

Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. 2011. "Flexible dynamic information flow control in Haskell." In: *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*, 95–106. DOI: 10.1145/2034675.2034688.

David Terei, Simon Marlow, Simon L. Peyton Jones, and David Mazières. 2012. "Safe haskell." In: *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell*, 137–148.

Chia-che Tsai, Donald E. Porter, and Mona Vij. 2017. "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX." In: *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. Ed. by Dilma Da Silva and Bryan Ford. USENIX Association, 645–658. https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

Chia-che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E. Porter. 2020. "Civet: An Efficient Java Partitioning Framework for Hardware Enclaves." In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, 505–522. https://www.usenix.org/conference/usenixsecurity20/presentation/tsai.

Li-Ting Tsai and Abhiroop Sarkar. 2016. *HsPaillier - Partial Homomorphic Encryption in Haskell*. https://hackage.haskell.org/package/Paillier. (accessed: 16.02.2023).

Tweag.io. 2022. *Haskell WASM backend*. https://www.tweag.io/blog/2022-11-22-wasm-backend-merged-in-ghc/. (accessed: 16.02.2023).

Eelco Visser and Yannis Smaragdakis, (Eds.) . 2016. *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. ACM. ISBN: 978-1-4503-4444-9. DOI: 10.1145/2983990.

Huibo Wang et al.. 2019. "Towards Memory Safe Enclave Programming with Rust-SGX." In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. Ed. by Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz. ACM, 2333–2350. DOI: 10.1145/3319535.3354241.

Lucas Waye, Pablo Buiras, Dan King, Stephen Chong, and Alejandro Russo. 2015. "It's My Privilege: Controlling Downgrading in DC-Labels." In: *Security and Trust Management - 11th International Workshop, STM 2015, Vienna, Austria, September 21-22, 2015, Proceedings* (Lecture Notes in Computer Science). Ed. by Sara Foresti. Vol. 9331. Springer, 203–219. ISBN: 978-3-319-24857-8. DOI: 10.1007/978-3-319-24858-5_13.

Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. 2021. "A Survey of Multitier Programming." *ACM Comput. Surv.*, 53, 4, 81:1–81:35. DOI: 10.1145/3397495.

Dmitry P. Zegzhda, E. S. Usov, V. A. Nikol'skii, and Evgeny Pavlenko. 2017. "Use of Intel SGX to ensure the confidentiality of data of cloud users." *Autom. Control. Comput. Sci.*, 51, 8, 848–854. DOI: 10.3103/S0146411617080284.

A Typechecked Operational Semantics of HasTEE in Haskell

```haskell
{-# LANGUAGE FlexibleContexts #-}
module HasTEEOrig where

import Control.Monad.State.Class
import Control.Monad.State.Strict

type Name = String

data Exp = Lit Int
         | Var Name
         | Fun [Name] Exp
         | App Exp [Exp]
         | Let Name Exp Exp
         | Plus Exp Exp

         -- HasTEE operators
         | InEnclave  Exp
         | Gateway  Exp
         | EnclaveApp Exp Exp -- (<@>)
         deriving (Show)

data Value = IntVal Int
           | Closure [Name] Exp Env
           -- HasTEE values
           | SecureClosure Name [Value]
           | ArgList [Value]
           | Dummy

           -- Error values
           | Err ErrState
           deriving (Show)

data ErrState = ENotClosure
              | EVarNotFound
              | ENotSecClos
              | ENotIntLit

instance Show ErrState where
  show ENotClosure  = "Closure not found"
  show EVarNotFound = "Variable not in environment"
  show ENotSecClos  = "Secure Closure not found"
  show ENotIntLit   = "Not an integer literal"

type Env = [(Name, Value)]

type ClientEnv  = Env
type EnclaveEnv = Env


type VarName = Int


data StateVar =
  StateVar { varName  :: Int
           , encState :: EnclaveEnv
           }

initStateVar :: EnclaveEnv -> StateVar
initStateVar = StateVar 0


eval :: Exp -> Value
eval e =
  let newEnclaveEnv = snd $
                      evalState (evalEnclave e
    initEnclaveEnv)
                      (initStateVar initEnclaveEnv)
  in fst $ evalState (evalClient e initClientEnv) (
    initStateVar newEnclaveEnv)
  where
    initEnclaveEnv = []
    initClientEnv  = []

genEncVar :: (MonadState StateVar m) => m String
genEncVar = do
  n <- gets varName
  modify $ \s -> s {varName = 1 + n}
  pure ("EncVar" <> show n)

evalList :: (MonadState StateVar m) => [Exp] -> Env -> [
    Value] -> m ([Value], Env)
evalList []     e vals = pure (reverse vals, e)
evalList (e1:es) env xs = do
  (v, e) <- evalEnclave e1 env
  evalList es e (v:xs)


evalEnclave :: (MonadState StateVar m)
            => Exp -> EnclaveEnv -> m (Value, EnclaveEnv)
evalEnclave (Lit n) env = pure (IntVal n, env)
evalEnclave (Var x) env = pure (lookupVar x env, env)
evalEnclave (Fun xs e) env =
  pure (Closure xs e env, env)
evalEnclave (Let name e1 e2) env = do
  (e1', env') <- evalEnclave e1 env
  evalEnclave e2 ((name,e1'):env')
evalEnclave (App f args) env = do
  (v1, env1) <- evalEnclave f env
  (vals, env2) <- evalList args env1 []
  case v1 of
    Closure xs body ev ->
      evalEnclave body ((zip xs vals) ++ ev)
    _ -> pure (Err ENotClosure, env2)
evalEnclave (Plus e1 e2) env = do
  (v1, env1) <- evalEnclave e1 env
  (v2, env2) <- evalEnclave e2 env1
```

```
  case (v1, v2) of
    (IntVal a1, IntVal a2) -> pure (IntVal (a1 + a2), env2)
    _ -> pure (Err ENotIntLit, env2)

evalEnclave (InEnclave e) env = do
  (val, env') <- evalEnclave e env
  varname     <- genEncVar
  let env'' = (varname, val):env'
  pure (Dummy, env'')
-- the following two are the essentially no-ops
evalEnclave (Gateway e) env = evalEnclave e env
evalEnclave (EnclaveApp e1 e2) env = do
  (_, env1) <- evalEnclave e1 env
  (_, env2) <- evalEnclave e2 env1
  pure (Dummy, env2)

evalList2 :: (MonadState StateVar m) => [Exp] -> Env -> [Value] -> m ([Value], Env)
evalList2 []     e vals = pure (reverse vals, e)
evalList2 (e1:es) env xs = do
  (v, e) <- evalClient e1 env
  evalList2 es e (v:xs)


evalClient :: (MonadState StateVar m)
           => Exp -> ClientEnv -> m (Value, ClientEnv)
evalClient (Lit n) env = pure (IntVal n, env)
evalClient (Var x) env = pure (lookupVar x env, env)
evalClient (Fun xs e) env =
  pure (Closure xs e env, env)
evalClient (Let name e1 e2) env = do
  (e1', env') <- evalClient e1 env
  evalClient e2 ((name,e1'):env')
evalClient (App f args) env = do
  (v1, env1) <- evalClient f env
  (v2, env2) <- evalList2 args env1 []
  case v1 of
    Closure xs body ev ->
      evalClient body ((zip xs v2) ++ ev)
    _ -> pure (Err ENotClosure, env2)
evalClient (Plus e1 e2) env = do
  (v1, env1) <- evalClient e1 env
  (v2, env2) <- evalClient e2 env1
  case (v1, v2) of
    (IntVal a1, IntVal a2) -> pure (IntVal (a1 + a2), env2)
    _ -> pure (Err ENotIntLit, env2)


evalClient (InEnclave e) env = do
  (_, env') <- evalClient e env
  varname     <- genEncVar
  let env'' = (varname, Dummy):env'
  pure (SecureClosure varname [], env'')
evalClient (Gateway e) env = do
  (e', env1) <- evalClient e env
  case e' of
    SecureClosure varname vals -> do
      enclaveEnv <- gets encState
      let func = lookupVar varname enclaveEnv
      case func of
        Closure vars body encEnv -> do
          (res,enclaveEnv') <- evalEnclave body ((zip vars vals) ++ encEnv)
          pure (res, env1)
        _ -> pure (Err ENotClosure, env1)
    _ -> pure (Err ENotSecClos, env1)
evalClient (EnclaveApp e1 e2) env = do
  (v1, env1) <- evalClient e1 env
  (v2, env2) <- evalClient e2 env1
  case v1 of
    SecureClosure f args ->
      case v2 of
        ArgList vals -> pure (SecureClosure f (args ++ vals), env2)
        v -> pure (SecureClosure f (args ++ [v]), env2)
    v -> pure (ArgList [v,v2], env2)


-- gateway (f == SecureClosure f [])
-- gateway (f <@> arg == EA f arg == SecureClosure f [arg])
-- gateway (f <@> arg1 <@> arg2 == EA f (EA arg1 arg2) == SC f [arg1, arg2])


lookupVar :: String -> [(String, Value)] -> Value
lookupVar _ [] = Err EVarNotFound
lookupVar x ((y, v) : env) =
  if x == y then v else lookupVar x env
```

Listing 11. Operational Semantics of HasTEE

B Data Clean Room Code

```
-- This function runs the query over the dataset and
-- returns the true result
countingQuery :: Enclave (Ref CleanRoomSt) -> (User -> Bool) -> Enclave Int
countingQuery refst q = do
    st <- readRef =<< refst
    return $ length $ filter id $ map q (users st)

-- Sample the laplace distribution
laplaceDistribution :: Enclave (Ref CleanRoomSt) -> Double -> Enclave Double
laplaceDistribution refst b = do
    z <- int2Double <$> getRandom refst (0,1)
    u <- ((/) 1000 . int2Double) <$> getRandom refst (1,1000)
    return $ (2 * z - 1) * (b * log u)

-- The laplace mechanism, assuming the server state is
-- already given, has type
-- (User -> Bool) -> Enclave Double
-- In the example usage in the paper, the salaryWithin
-- query is partially applied such that it is of type
```

```
-- User -> Bool
laplaceMechanism :: Enclave (Ref CleanRoomSt) -> (User ->
    Bool) -> Enclave Double
laplaceMechanism refst q = do
  st <- readRef =<< refst
  -- perform the true query
  true <- int2Double <$> countingQuery refst q
  -- sample noise from the laplace distribution
  noise <- laplaceDistribution refst (1 / (epsilon st))
  -- augment the true result with the noise
  return $ true + noise
```

Listing 12. Code for the laplace mechanism.

Received 2023-06-01; accepted 2023-07-04

Chapter 6

HasTEE$^+$: Confidential Computing and Analytics with Haskell

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.

1 Introduction

Confidential computing [28] is an emerging security paradigm that ensures the isolation of sensitive computations and data during processing, shielding them from potential threats within the underlying infrastructure. It accomplishes this by employing specialised hardware isolation units known as Trusted Execution Environments (TEEs). A TEE unit, such as the Intel SGX [24] or ARM TrustZone [1], provides 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, I/O peripherals, BIOS and other firmware.

TEEs further allow a remote party to establish *trust* by providing a measurement of the sensitive code and data, composing a signed *attestation report* that can be verified. As such, TEEs have been heralded as a leading contender to enforce a strong notion of *trust* within cloud computing infrastructure[4,33,50], particularly in regulated industries such as healthcare, law and finance [12].

However, an obstacle in the wide-scale adoption of confidential computing has been the awkward programming model of TEEs [47]. TEEs, such as Intel SGX, involve partitioning the state of the program into a trusted project linked with Intel-supplied restricted C standard library [19] and an untrusted project that communicates with the trusted project using custom data copying protocols [18]. The complexity is compounded for distributed, multi-tiered cloud applications due to the semantic friction in adhering to various data formats and protocols across multiple projects [32], resulting in increased developer effort [30].

Furthermore, a well-known class of security vulnerabilities [8,39] arise from the memory-unsafety of the TEE projects. Given the security-critical nature of TEE applications, efforts have been made to introduce Rust-based [48] and Golang-based SDKs [13] aimed at mitigating memory unsafety vulnerabilities. However, the same applications remain vulnerable to unintended information leaks [34]. Consider the following *trusted* function hosted within a TEE:

```
#include "library.h" //provides `int computeIdx(char *)`
char *secret = "...";
int public_arr [15] = ....;
void trusted_func(char *userinput, int inputsize) {
   if (memcmp(secret, userinput, inputsize) == 0) {
     int val = computeIdx(userinput);
     ocall_printf("%d\n", public_arr[val]); // handwritten printf routine
   } else { ocall_printf("0\n"); }
}
```

In the above program, at least five attack vectors are present - (1) the `inputsize` parameter can be abused to cause a buffer-overflow attack, (2) the `userinput` parameter can be tampered by a malicious operating system to force an incorrect branching, (3) even with the correct `inputsize` and `userInput`, the attacker can observe `stdout` to learn which branch was taken, (4) the trusted library function - `computeIdx` - could intentionally (if written by a malicious party) or accidentally leak `secret`, and (5) finally the attacker can use timing-based side channels to learn the branching information or even the `secret` [7].

Our contribution through this work is – *HasTEE*$^+$ – a Domain Specific Language (DSL) embedded in Haskell and targeted towards confidential computing. HasTEE$^+$ is designed to mitigate at least four of the five attack vectors mentioned above. Additionally, it offers a *tierless* programming model, thereby simplifying the development of distributed, multi-tiered cloud applications.

HasTEE$^+$ builds on the HasTEE [37] project, which crucially provides a *Glasgow Haskell Compiler (GHC)* runtime [23] capable of running Haskell on

Intel SGX machines. While using a memory-safe language like Haskell or Rust mitigates attack vector (1), all the other attack vectors remain open in the HasTEE project. HasTEE$^+$ notably adds support for a general one-time-effort remote-attestation infrastructure that helps mitigate the attack vector (2). While the underlying protocol employs Intel's RA-TLS [21], HasTEE$^+$ ensures that programmers are not required to create custom attestation code for capturing and sending measurements or conducting integrity checks [22].

For attack vectors (3) and (4), HasTEE$^+$ adds support for a dynamic information flow control (IFC) mechanism based on the Labeled IO Monad (LIO) [43]. Accordingly, we adopt a *floating-label* approach from OSes such as HiStar [51], enabling HasTEE$^+$ to relax some of the impractical I/O restrictions in the original HasTEE project. Side-channel attacks (attack vector (5)) remain outside the scope of HasTEE$^+$.

Concerning the complex programming model, modern TEE incarnations like AMD SEV-SNP [38] and Intel TDX [20] introduce a virtualization-based solution, opting to virtualize the entire project instead of partitioning it into trusted and untrusted components. At the cost of an increased trusted code base, this approach simplifies the TEE project layout. Nevertheless, it remains vulnerable to the complexities of a multi-tiered cloud application, as well as all five of the aforementioned attack vectors.

The *tierless* programming model of HasTEE$^+$ expresses multi-client-server projects as a single program and uses the Haskell type system to distinguish individual clients. A separate monadic type, such as `Client "client1" a`, demarcates each individual client, while HasTEE's multi-compilation tactic partitions the program. This programming model, evaluated on Intel SGX, remains applicable on newer Intel TDX machines.

Contributions. We summarize the key contributions of HasTEE$^+$ here:

- HasTEE$^+$ introduces a *tierless* DSL (Section 3.1), capable of expressing multi-tiered confidential computing applications as a single program, increasing an application's comprehensibility and reducing developer effort.
- HasTEE$^+$ incorporates a remote attestation design (Section 3.2) that relieves programmers from crafting custom integrity checks and attestation setups.
- HasTEE$^+$ integrates dynamic information flow control mechanisms (Section 3.3) to prevent explicit and implicit information leaks from applications.
- We use HasTEE$^+$'s IFC mechanism and cryptography in a *data clean room* case study (Section 4), showing a general data sharing pattern for conducting analytics on confidential data among mutually distrustful participants.

2 Threat Model

We build upon the threat model of the HasTEE project [37] and other related works on Intel SGX [3,4,13,48]. In such a threat model, an attacker attempts to compromise the code and data memory within the TEE. The attacker has administrative access to the operating system, hypervisor and other related system software hosted on a malicious cloud service.

We expand the above threat model to include an *active attacker* attempting to compromise the integrity of the data flowing into the TEE, as well as a *passive attacker* who observes the public channels that the trusted software interacts with to learn more about its behaviour. The threat model terminology is adopted from the J_E project [31], and related attacks are discussed in subsequent sections. Another class of potential threats emerge from the inclusion of public software libraries into TEE software, such as cryptography libraries, which might accidentally or intentionally leak secrets [25].

HasTEE$^+$'s attestation infrastructure, based on Intel's RA-TLS protocol [21] accounts for masquerading attacks [15,45]. Availability attacks such as denial-of-service and hardware side-channel attacks are outside the scope of this work.

3 The HasTEE$^+$ DSL

We illustrate the key APIs of HasTEE$^+$ using a password checker application in Listing 1, and explain the individual types and functions in the subsequent sections. Notably, the entire application, consisting of a separate client and server, can be expressed within 27 lines of Haskell code (excluding import declarations).

Listing 1 shows the user *Alice* storing her password in the TEE memory and deploying the trusted function `pwdChecker` to conduct a password check. The application uses three key types – `EnclaveDC`, `Client`, and `App`, adapted from HasTEE [37]. All three types implements a monadic interface, denoted as `m`, constructed using fundamental operations `return :: a -> m a` and `(>>=) :: m a -> (a -> m b) -> m b` (read as *bind*). The `return x` operation produces a computation returning the value of x without side effects, while the `(>>=)` function sequences computations and their associated side effects. In Haskell, we often use the *do-notation* to express such monadic computations.

The `EnclaveDC` monad represents the trusted computations that get loaded onto a TEE. The name *Enclave* alludes to an Intel SGX *enclave*, while *DC* stands for *Disjunction Category*, which we further explain in Section 3.3.

The client-side of the application is represented by the namesake `Client` monad that captures a *type-level* string - `"client"`. The type-level string allows the Haskell type-system to distinguish between multiple clients and serves as an identifier for the monadic computation runner function - `runAppRA` on line 27.

The monad `App` serves as a staging area where the enclave data (`"password"`) and the enclave computation (`pwdChecker`) are loaded into trusted memory. Also, the `client` function is provided with the `API` it will use to communicate with the TEE within the `App` monad. We provide the type signatures of a simplified subset of the HasTEE$^+$ APIs used in Listing 1 for loading trusted data and computations, and running the computations, in Fig. 1 below.

The application begins with the `app` function (lines 17-25). The functions `inEnclaveLabeledConstant` and `inEnclave` are used to load the trusted data (line 19) and the trusted function (line 21) in the enclave, respectively. The `runClient` at line 22 runs the monadic `Client` computations. Note, there is no

```
1  pwdChecker :: EnclaveDC (DCLabeled String) -> String -> EnclaveDC Bool
2  pwdChecker pwd guess = do
3    l_pwd <- pwd
4    priv  <- getPrivilege
5    p     <- unlabelP priv l_pwd
6    if p == guess then return True else return False
7
8  data API = API { checkpwd :: Secure (String -> EnclaveDC Bool) }
9
10 client :: API -> Client "client" ()
11 client api = do
12   liftIO $ putStrLn "Enter your password:"
13   userInput <- liftIO getLine
14   res <- gatewayRA ((checkpwd api) <@> userInput)
15   liftIO $ putStrLn ("Login returned " ++ show res)
16
17 app :: App Done
18 app = do
19   pwd   <- inEnclaveLabeledConstant pwdLabel "password"
20   let priv = toCNF "Alice"
21   efunc <- inEnclave (dcDefaultState priv) $ pwdChecker pwd
22   runClient (client (API efunc))
23   where
24     pwdLabel :: DCLabel
25     pwdLabel = "Alice" %% "Alice" :: DCLabel
26
27 main = runAppRA "client" app >> return ()
```

Listing 1: A password checker application in HasTEE+

```
inEnclave :: Label l => LIOState l p -> a -> App (Secure a)
inEnclaveLabeledConstant
          :: Label l => l -> a -> App (EnclaveDC (DCLabeled a))
gatewayRA :: (Binary a, Label l)
          => Secure (Enclave l p a) -> Client loc a
(<@>)     :: Binary a => Secure (a -> b) -> a -> Secure b
runClient :: Client loc a -> App Done
runAppRA  :: Identifier   -> App a -> IO a
```

Fig. 1. HasTEE+ APIs for loading data and computations on the TEE and invoking the TEE (parameterized types simplified and typeclass constraints omitted for brevity).

equivalent `runEnclave`, as in our programming model *a client functions as the main application driver, while the enclave serves as a computational service.*

The client and server communicate with each other through a user-defined `API` type (line 8) that encapsulates a remote closure, represented using the

`Secure` type constructor. This closure is constructed on line 21 with the `inEnclave` function whose type signature can be found in Fig. 1. The parameter `LIOState l p` and the typeclass constraint `Label l` are explained in Section 3.3.

The `client` function (lines 10-15) has access to the remote closure through the `API` type. The remote function is invoked on line 14 using the `<@>` operator to emulate function application and the `gatewayRA` function executes the remote function call. The respective type signatures are specified in Fig. 1.

A notable type in the `pwdChecker` function is `DCLabeled String` that captures the password string but is labeled with ownership information of user *Alice*. The labeling happens on line 19 using the `inEnclaveLabeledConstant` function and the label `pwdLabel` (lines 24,25). The body of `pwdChecker` uses certain IFC operations - `getPrivilege` and `unlabelP`, which we elaborate on in Section 3.3.

The structure of Listing 1 represents the standard style of writing HasTEE+ programs, where the monadic types `EnclaveDC`, `Client` and `App` indicate the partitioning within the program body. We discuss the partitioning tactic and the underlying representation to capture multiple clients in the next section.

3.1 Tierless client-server programming

HasTEE+ builds on the partitioning strategy of HasTEE [37] but generalizes it for multiple clients. The strategy involves compiling the program `n` times for `n` parties. The compilation of the enclave program substitutes dummy implementation for all client monads. Similarly, each client compilation substitutes a dummy value for the enclave monad. The distinction between each client is done using a type-level string identifier, such as `"client1"`. At runtime, this identifier is used to dynamically dispatch the correct client, as shown in Fig. 2.

The dynamic identifier-based dispatch of the client computation is inspired from the HasChor library [40] for choreographic programming. In this approach, a `Client` monad is parameterized with a type-level location string, which is used at runtime to execute the desired `Client` computation (see Listing 2).

```
data Client (loc :: Symbol) a where
  Client :: (KnownSymbol loc) => Proxy loc -> IO a -> Client loc a

symbolVal :: forall (n :: Symbol) proxy. KnownSymbol n => proxy n -> String

runClient :: Client loc a -> App Done
runClient (Client loc cl) = App $ do
  location <- get -- the underlying App monad captures the location
  if ((symbolVal loc) == location)
  then liftIO cl >> return Done
  else return Done -- cl not executed
```

Listing 2: The underlying Client monad in HasTEE+

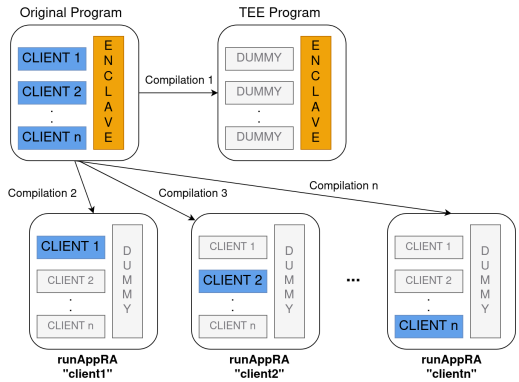


Fig. 2. HasTEE+'s partitioning uses multiple compilations to create binaries that can dynamically dispatch the code for only one concerned monad based on a string identifier

The function `symbolVal` is provided by GHC to reflect types as terms at runtime, provided the types are constrained by the `KnownSymbol` typeclass. The `runClient` function in Listing 2 queries the App monad that captures the string within the underlying `App` monad. The `runClient` implementation for the `EnclaveDC` module is simply implemented as `runClient _ = return Done`, which amounts to a dummy implementation. A case study involving multiple clients is demonstrated in Section 4.

For the remote function invocation, the `inEnclave` function internally builds a *dispatch table* mapping an integer identifier to each enclave function. The client only gets access to the integer identifier. It uses the `<@>` operator to gather the function argument and the `gatewayRA` function to serialise the arguments, make the remote function call (specifying the identifier), and obtain the result computed on the remote enclave machine. Note the `Binary` typeclass constraint (Fig. 1) on both of the remote invocation functions for binary serialisation.

The complete implementation details of HasTEE+ has been made publicly available[1]. Further details on the operational semantics of the general partitioning strategy can be found in the HasTEE[37] paper.

3.2 Remote Attestation via a Monitoring Server

A key component of confidential computing is remote attestation, which establishes *trust* on a TEE within a malicious environment. In HasTEE+, we conduct our experiments on Intel SGX enclaves, and hence our infrastructure is integrated with the SGX attestation protocol. The low-level protocol is a multi-step

[1] https://github.com/Abhiroop/HasTEE

process [21] that begins with the client sending a nonce to the TEE, the TEE then creates a manifest file that includes an ephemeral key to encrypt future communication. Next, the TEE generates an *attestation report* that summarizes the enclave and platform state. A quoting enclave on the same machine verifies and signs the report, now called a *quote*, and returns it to the client. The client then communicates with the Intel Attestation Service (IAS) to verify the quote.

The API for this interface is quite low-level and involves programming at the level of a device driver (`/dev/attestation`). Intel's RA-TLS protocol abstracts over the low-level APIs and presents an API deeply tied to the TLS protocol. RA-TLS operates by extending an X.509 certificate to incorporate the attestation report within an unused X.509 extension field. During TLS connection setup, it uses the TLS handshake to transmit the *quote*, calculated using the protocol described earlier [21]. The enclave programmer, working with RA-TLS, interacts with a modified TLS implementation such as Mbed TLS [2]. Now, the focus shifts back to dealing with low-level socket-programming APIs, such as:

```
int (*ra_tls_create_key_and_crt_der_f)(uint8_t** der_key, size_t* der_key_size,
                                       uint8_t** der_crt, size_t* der_crt_size);
void (*ra_tls_set_measurement_callback_f)(int (*f_cb)(const char* mrenclave,
          const char* mrsigner, const char* isv_prod_id, const char* isv_svn));
```

Once again, managing these APIs is *error-prone* and *memory unsafe*. Additionally, it requires constructing *underspecified protocols*. Most importantly, the programmer is burdened with handling *cross-cutting concerns* that are irrelevant to the application code.

In HasTEE+, we abstract over Intel's RA-TLS protocol. As mentioned earlier, clients always serve as the primary program *driver*, while the enclave functions as a computational *service*. The *enclave-as-a-service* model is implemented by representing the entire enclave program as an infinitely running server. The server is implemented in C using the Mbed TLS library [2], which can parse and verify the modified X.509 certificate. Internally, when the enclave runs, it spawns the C server hosted on the enclave memory but using separate memory pages. We use GHC's Foreign Function Interface to establish a communication channel between the C and Haskell heaps. Listing 3 shows a high-level overview of the implementation.

The Mbed TLS-based C server module acts as a *monitor* for the enclave application. All dataflows between the clients and the enclave pass through this module, which conducts integrity checks on incoming data at this point. Fig. 3 shows the HasTEE+ general monitoring architecture.

There are two distinct attackers targeting the dataflow - (1) a malicious OS snooping or tampering with the data flowing into the enclave, and (2) a malicious client, potentially colluding with the OS, repeatedly sending garbage inputs to observe the behaviour of the enclave. The TLS channel specifically prevents the first attack. For the second attack, we use public-key-cryptography-based digital signatures to verify the identity of the client making the request.

For instance, in Listing 1, we provision the public key for user *Alice* during enclave boot time and disallow password checks from other malicious clients.

```
runAppRA :: Identifier -> App a -> IO a
runAppRA ident (App s) = do
  (a, vTable) <- runStateT s (initAppState ident)
  flagptr  <- malloc :: IO (Ptr CInt)
  dataptr  <- mallocBytes dataPacketSize :: IO (Ptr CChar)
  _    <- forkOS (startmbedTLSSERVER_ffi tid flagptr dataptr)
  result <- try (loop vTable flagptr dataptr) -- exception handler
  -- exception handling and freeing C pointers
  return a
  where
    loop :: [(CallID, Method)] -> Ptr CInt -> Ptr CChar -> IO ()
    loop vTable flagptr dataptr = do -- body elided
      -- non-blocking loop that gets woken when data arrives;
      -- `flagptr` indicates data arrival; read data from `dataptr`
      -- invoke the correct method from the lookup table `vTable`
      loop vTable flagptr dataptr -- continue the event loop
-- implemented in C
startmbedTLSSERVER_ffi :: ThreadId -> Ptr CInt -> Ptr CChar -> IO ()
```

Listing 3: High-level template of the `runAppRA` function for the enclave

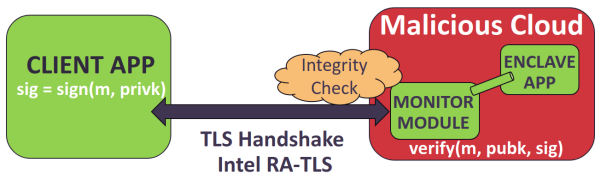


Fig. 3. HasTEE+'s remote attestation infrastructure abstracts over Intel's RA-TLS protocol and supports establishing the identity of the client and the server

While Intel-SGX also offers mutual attestation services, they depend on the client machine supporting an SGX enclave. Considering Intel's recent deprecation of SGX services on desktops and other client devices [47], our scheme aligns well.

3.3 Dynamic Information Flow Control

While programming TEEs using a memory-safe and type-safe language inherently provides stronger guarantees than programming in C/C++, such TEE applications remain vulnerable to unintended information leaks. To mitigate such explicit and implicit information leaks, HasTEE+ integrates a dynamic information flow control mechanism within the underlying `EnclaveDC` monad. For instance, the trusted function `pwdChecker` from Listing 1 operates in this monad and captures a *labeled* string - the user password. The internal representation of the types `EnclaveDC a` and `DCLabeled a` is as follows:

```
newtype Label l => Enclave l p a = Enclave (IORef (LIOState l p) -> IO a)
```

```
data Labeled l t where
  LabeledTCB :: (Label l, Binary l, Binary t) => l -> t -> Labeled l t

type EnclaveDC = Enclave DCLabel DCPriv
type DCLabeled = Labeled DCLabel

data LIOState l p = LIOState { lioCurLabel :: l, lioClearance :: l
                             , lioOutLabel :: l, lioPrivilege :: Priv p}
```

The above representation is inspired by the LIO Haskell library [43]. The `Enclave` monad is parameterized by the label type `l` and privilege type `p`, wherein the `Label` typeclass captures a lattice [9] with partial order $\sqsubseteq$ governing the allowed flows. The `Enclave` monad employs a *floating label* approach, inspired by the HiStar OS [51]. In this approach, the *computational context* retains a current label L_{cur}. Upon reading sensitive data labeled L, it *taints* the context with the label $L_{cur} \sqcup L$, where $\sqcup$ denotes the least upper bound. The floating label approach restricts subsequent effects and enforces the lattice property, thereby preventing information leaks from higher-classified data to lower contexts in the lattice—a principle known as non-interference [14]. In the `LIOState` type, `lioCurLabel` represents L_{cur}, and `lioClearance` imposes an upper bound on the upward flow of a computational context within the lattice.

The `EnclaveDC` type specialises the `Enclave` monad to use *disjunction category (DC)* labels [42]. A DC Label captures both the *confidentiality* [9] and *integrity* [6] as a tuple. It employs the notion of mutually distrusting *principals*, whose conjunction represents restrictions on both the confidentiality and integrity of the data. An example label type is found in Listing 1, where `pwdLabel` constructs a DC Label using the tuple-construction operator `%%` and a string representation of the principal *Alice* to give `"Alice" %% "Alice"` (line 25).

A notable component in the DC Label system is the notion of a *privilege*, which is the type parameter `p` in the `Enclave` monad. In most real-world scenarios, the strict enforcement of non-interference is impractical, and privileges allow relaxing this policy by defining a more flexible ordering relation, $\sqsubseteq_P$. In HasTEE+'s DCLabel implementation, we use a *conjunctive normal form (CNF)* representation for each disjunctive category of confidentiality and integrity. Hence, given a boolean formula P representing the privileges and two labels $< C_1, I_1 >$ and $< C_2, I_2 >$, the $\sqsubseteq_P$ is defined as shown in the formula.

$$\frac{P \wedge C_2 \implies C_1 \qquad P \wedge I_1 \implies I_2}{< C_1, I_1 > \sqsubseteq_P < C_2, I_2 >}$$

In Listing 1, we use the `toCNF` function (line 20) to generate a privilege for *Alice* to declassify the password, provisioning it to the `Enclave` monad at boot time (line 21). This allows the `pwdChecker` function to invoke `getPrivilege` (line 4) and then use that privilege to call the `unLabelP` function (line 5), which internally computes the $\sqsubseteq_P$ formula shown above. The `unlabelP` function and a related set of core APIs for HasTEE+'s IFC enforcement is shown in Fig. 4.

```
unlabel  :: Label l => Labeled l a -> Enclave l p a
unlabelP :: PrivDesc l p => Priv p -> Labeled l a -> Enclave l p a
label    :: (Label l, Binary l, Binary a)
         => l -> a -> Enclave l p (Labeled l a)
labelP   :: (PrivDesc l p, Binary l, Binary a)
         => Priv p -> l -> a -> Enclave l p (Labeled l a)
taint    :: Label l => l -> Enclave l p ()
taintP   :: PrivDesc l p => Priv p -> l -> Enclave l p ()
```

Fig. 4. Core HasTEE$^+$ APIs for Information Flow Control

Implementation note: for prototyping, we represent principals and corresponding privileges using `String`s. In practice, a 512-bit private-key-hash is recommended.

The operations shown above dynamically compute the $\sqsubseteq$ and $\sqsubseteq_P$ relation to determine allowed information flows. The `PrivDesc` typeclass permits delegating privileges akin to the *acts for* relation described in the Myers-Liskov labeling model [29]. The type `Labeled l a` exists to allow labeling values to labels other than L_{cur}. Labeled data can be used to indicate data ownership and hence we provide additional APIs for the `Client` monad to `label`, serialise and `unlabel` data, inspired by *labeled communication* in the COWL system [44].

An implementation challenge arises when integrating the dynamic IFC mechanism with our partitioning tactic, specifically within the `inEnclave :: Label l => LIOState l p -> a -> App (Secure a)` function. This function is used to mark a function as trusted and move it into the TEE. The polymorphic type `a` encodes any general function of the form $\texttt{a}_1$ `->` $\texttt{a}_2$ `->` `...` $\texttt{a}_n$ `-> Enclave l b`. However, the type-checker is unable to unify the `l` in `LIOState l p` and the `l` in `Enclave l b`. Due to the dynamic nature of our IFC mechanism, a user can mistakenly supply a different label type at runtime, preventing the type-checker from producing a witness. Accordingly, we use the `Data.Dynamic` module of GHC to *dynamically type* the `LIOState l p` term. Thus, before evaluating the LIO computation, our evaluator dynamically checks for matching types and, on success, executes the monadic computation. A notable aspect is that both the program partitioning and IFC enforcement are *implemented as a Haskell library*, which allows us to use these features of the language.

4 Case Study: A Confidential Data-Analytics Pattern

We present a case study in privacy-preserving data analytics, illustrating how a group of mutually distrusting parties can perform analytics without revealing their data to each other. We use the core features of HasTEE$^+$ that we have discussed so far - tierless programming, remote attestation and dynamic IFC with privileges for declassification.

Fig. 5 shows the overall confidential analytics setup. The analytics is carried out in a *data clean room (DCR)* - a TEE hosted on a public cloud that aggregates

data from multiple parties without revealing the actual data. In Fig. 5, the notation $\{a, b, ...\}$ denotes the state (in-memory and persistent) of that party. The figure shows two distinct sets of participants - the *Data Providers (P)* and the *Analytics Consumers (C)*. The setup does not limit the number of participants, allowing m data providers and n analytics consumers, where $m, n \in \mathbb{N}$. Additionally, there are *no restrictions requiring the data providers and the analytics consumer to be different*, and hence in some cases $P = C$.

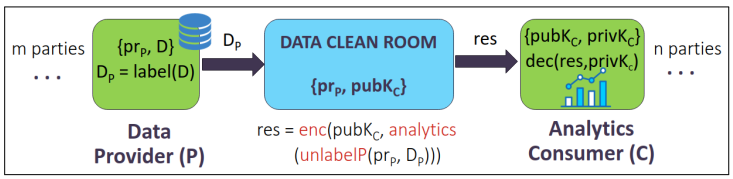


Fig. 5. A *Data Clean Room (DCR)* pattern with m data providers (P) and n analytics consumers (C). P labels its data as D_P and sends it to the DCR, which loads C's public key $pubK_C$ as well as *privilege* pr_P using a closure. The functions *enc* and *dec* handle encryption and decryption, and *analytics* refers to any general query.

In this setup, the data providers *label* their data before sending it to the DCR. The DCR is loaded with analytics queries from C after P reviews both the schema and the specific queries requested by C. The DCR is provisioned with C's public key, limiting access to the computed analytics solely to C. We notably use Haskell's partial application to load the privilege pr_P, enabling data unlabeling while restricting the locations of the `unlabelP` operation.

Consider a synthetic data analytics example with two data providers P_1 and P_2 that both store confidential data regarding COVID strains and the corresponding age of patients. An analytics consumer C_1 wishes to aggregate this data and derive the correlation between mean age and the respective COVID strain. We add a constraint that the analytics should only aggregate COVID strains that are common to both P_1 and P_2 (*private set intersection* [11]).

The DCR exposes two API calls for communication. The first, `datasend`, accepts a `DCLabeled Row` as an argument and is used by P_1 and P_2 to send a row labeled with their respective DC Label. The DCR stores the in-memory data in HasTEE+'s mutable reference type `DCRef a`. Reading and writing occur via `readRef` and `writeRef`, raising the context's label accordingly. We omit the body of `datasend` for brevity. The full Haskell program is publicly available [36].

Listing 4 (lines 1-7) shows the second interface to the DCR, `runQuery`, used by C_1 to run the analytics query. A notable operation happens in line 5 where the `unLabelFunc` function is applied to each row of the database, labeled with either P_1's or P_2's DC label. `unlabelFunc` inspects the label and accordingly uses the correct privilege to declassify the data. Note, if during the whole computation, a row's label gets tainted by both P_1 and P_2, the `unlabel` function (not `unlabelP`,

```
1  runQuery :: EnclaveDC (DCRef DB) -> PublicKey
2           -> Priv CNF -> Priv CNF -> EnclaveDC ResultEncrypted
3  runQuery enc_ref_db pubKC1 priv1 priv2  = do
4    labeled_rows <- join $ readRef <$> enc_ref_db
5    rows         <- mapM (unlabelFunc priv1 priv2) labeled_rows
6    res_enc      <- encrypt_TCB pubKC1 (toStrict $ encode $ query rows)
7    return res_enc -- encryption error-handling elided
8
9  unlabelFunc p1 p2 lrow =
10   case extractOrgName (labelOf lrow) of
11     "P1" -> unlabelP p1 lrow
12     "P2" -> unlabelP p2 lrow
13     _      -> unlabel lrow -- label will float up
14
15 data API = API { datasend :: Secure (DCLabeled Row -> EnclaveDC ())
16                , runQ     :: Secure (EnclaveDC ResultEncrypted }
17
18 app = do db     <- liftNewRef dcPublic database
19          sfunc  <- inEnclave dcDefaultState $ sendData db
20          pubKC1 <- liftIO $ read <$> readFile "ssl/public.key"
21          p1Priv <- liftIO $ privInit (toCNF p1)
22          p2Priv <- liftIO $ privInit (toCNF p2)
23          qfunc  <- inEnclave dcDefaultState $ runQuery db pubKC1 p1Priv p2Priv
24          let api = API sfunc qfunc
25          runClient (client1 api)
26          runClient (client2 api)
27          runClient (client3 api)
```

Listing 4: `runQuery` unlabels the data, runs the query and encrypts the result; The `app :: App Done` computation captures the three clients and the enclave

see line 13) is invoked, floating the context high enough that writes to public channels are no longer possible.

In the absence of privileges, the `EnclaveDC` monad obeys general non-interference [14]. Hence, privileges, which allow *declassification* and *endorsement*, must be handed out with caution and used in limited places. In `app`, the privileges are created (lines 21, 22) and are partially applied to the `runQuery` function (line 23). As a result, the enclave loads a partially applied closure, `runQuery db pubKC1 p1Priv p2Priv`, and it is limited to using the privileges solely within `runQuery` and its *callees*. An interesting future work would be using Haskell's linear type support [5] to limit the copying of privileges and make them *unforgeable*.

The `app` function demonstrates the overall *tierless* nature of our DSL. It describes three clients and the enclave as a single program without specifying complex data copying protocols or involving multi-project hierarchies. We elide the body of the clients P_1 and P_2, involving data retrieval from their databases, labeling, and sending it to DCR, while C_1 calls `runQuery` and decrypts the result. We also omit the `query` function's implementation, responsible for exe-

cuting private set intersection and returning results in a structured format. The interested reader can find the entire program hosted publicly [36]. In-transit security, enclave-integrity and client-integrity checks are implicitly enforced on all communication through HasTEE$^+$'s remote attestation infrastructure.

4.1 Security Analysis

Privacy Protection. The data clean room ensures privacy through - (1) *runtime security*, provided by the TEE's isolation of trusted code and data; (2) *in-transit security*, ensured by the RA-TLS protocol; (3) *enclave integrity*, established through remote attestation; (4) *client integrity*, provided with digital signatures checked by a monitor (Section 3.2); and (5) *information flow control*, implemented using a mix of IFC mechanisms and controlled privilege delegation.

Why is the result encrypted if HasTEE$^+$'s monitor already does client-integrity checks? This is necessary due to two distinct attacker models: *open-world attacks* and *closed-world attacks*. The digital signature verification in the monitor protects against open-world attacks, where an unknown malicious attacker outside our described system attempts to communicate with the DCR. On the other hand, in a closed-world attack, one of the participating entities, say P_1, may maliciously query the DCR for analytics, even though P_1 is intended to be merely a data provider. Although the monitor will allow this communication, the encryption will protect the data privacy.

Declassification Dimensions [35] Classifying the DCR along the four dimensions - **Who**: Integrity checks in HasTEE$^+$, RA-TLS and data labeling constrain the *who* dimension, allowing the DCR alone to declassify the data; **What**: The combination of the Haskell type system and dynamic privileges aim to allow declassification only for the analytics query; **Where**: The partial-application-based privilege loading was done to restrict this dimension, ensuring that only `runQuery` and its subsequent *callees* can declassify; **When** This is currently not captured but it is fairly straightforward to implement a *relative declassification* [35] policy where the analytics is released only after a certain set of data uploads succeeds, especially using the tierless nature of our DSL.

While HasTEE$^+$'s privacy protection mechanism provides useful guardrails against information leaks, we emphasize the importance of *auditing* the trusted code by all concerned parties to ensure the privileges are not misused.

5 Performance Evaluations

Here, we present performance microbenchmarks that allow for quantifying the overheads associated with various features in HasTEE$^+$. For benchmarking, we use the password checker example from Listing 1. We evaluate three particular sources of overheads - (1) *dynamic checks for information flow control*, (2) *remote attestation* and (3) *client-integrity checks* performed by the monitoring module.

We plot the overheads associated with each feature in Fig. 6. The X-axis captures the mean response-time in executing variants of the operation `gatewayRA`

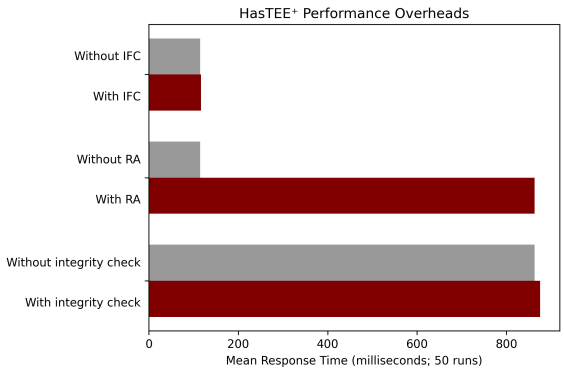


Fig. 6. Performance Overheads in HasTEE$^+$. The X-axis represents the mean response time for a query, both with and without the desired feature enabled, measured in milliseconds. The response time is averaged over 50 runs for each measure.

`((checkpwd api) <@> userInput)` (line 14) from Listing 1. A *gateway* call involves serialising the function arguments, making a remote procedure call to the enclave, executing the enclave computation, and then deserialising the result to the client. Measurement were conducted on an Azure Standard DC1s v2 (1 vcpu, 4 GiB memory) SGX machine, using the Intel SGX SDK for Linux.

Dynamic IFC Checks Overhead. In Fig. 6, the first group measures the overhead due to the runtime checks for IFC. The difference arises from extra conditional statements checking legal data flow policies. For Listing 1, the overhead is 2 milliseconds when compared to the non-IFC version. However, more complex applications (like Section 4) might incur slightly higher overheads.

Remote Attestation (RA) Overhead. The second group shows the RA overhead, demonstrating a considerable jump in response time with RA enabled. Much of the latency is attributed to the underlying *RA-TLS* protocol, implementing TLS version 1.2. In contrast, the non-RA baseline employs plain TCP for communication, using Linux's `send/recv` to enhance communication speed. The RA version's mean latency is 863 milliseconds, improvable by establishing a secure channel instead of initiating the entire handshake protocol each time.

Integrity-check Overhead. The client-integrity check is built on top of the HasTEE$^+$ RA infrastructure. As a result, for the baseline we use RA measurements from the second group and incorporate integrity checks on top of RA. The overhead on top of RA is minimal, in the order of 15 milliseconds.

Discussion. The measurements in Fig 6 show that each HasTEE$^+$ feature incurs maximum overheads in the order of hundreds of milliseconds. The significant response time increase for RA is mainly due to the complex TLS handshake involving multiple hops and communication with the Intel Attestation Service.

Given the security-critical nature of confidential computing and considering slowdowns due to general network latency, we posit that *HasTEE+'s overheads are acceptable, making it a practical choice for security-critical applications.*

6 Related Work

We have already discussed projects closely related to HasTEE+, including HasTEE [37], GoTEE [13], J_E [31], etc., in Sections 1 and 2. Here, we highlight additional related work that is relevant to the broader contributions of HasTEE+.

Tierless Programming for Enclaves To the best of our knowledge, HasTEE+ is one of the first practical programming frameworks to introduce the notion of *tierless* programming for confidential computing applications. Weisenburger et al. [49] provide a survey of general multi-tier programming approaches. Among the surveyed approaches, the HasTEE+ DSL draws inspiration from the *Haste framework* [10] and functional choreographic programming [40].

Remote Attestation Infrastructure. HasTEE+ is built on top of the Intel RA-TLS protocol [21] for *binary attestation*. In contrast, GuaranTEE [27] proposes a *control-flow attestation* technique based on two enclaves, which can be adapted quite naturally to the HasTEE+ RA infrastructure.

Information Flow Control for TEEs. Gollamudi et al. proposed the first use of IFC to protect against low-level attackers in TEEs with the IMP_E calculus [16], followed by a more general security calculus, DFLATE [17], for distributed TEE applications. In contrast to their work, HasTEE+ does not require language-level modifications or type-system extensions. Instead, it conveniently enforces *IFC as a library* in an existing programming language. At the OS level, Deluminator [46] offers OS abstractions and userspace APIs for trace-based tracking of IFC violations in compartmentalized hardware, such as TEEs. Note that Deluminator is a reporting tool and not an *enforcement* mechanism, in contrast to HasTEE+. Another application of IFC to TEEs is Moat [41], which formally verifies the confidentiality of enclave programs by proving the non-interference property [14].

Confidential Data Analytics. Referring to our confidential analytics pattern in Section 4, another proposed design pattern is Privacy Preserving Federated Learning [26], tailored to machine learning attacker models. We believe such threat models can be naturally integrated with our proposed design pattern.

7 Conclusion

We introduced HasTEE+, a *tierless* confidential computing DSL that enforces dynamic information flow control, along with strong client-integrity and enclave-integrity checks. We also proposed a general confidential analytics pattern, expressed as a single program in HasTEE+. Additionally, we presented performance evaluations that demonstrate acceptable overheads. Our evaluations, while conducted on Intel SGX, illustrate HasTEE+'s general applicability to ARM TrustZone, AMD SEV, and Intel TDX machines. Furthermore, our library-based partitioning and IFC approach is extendable to other programming languages.

References

1. ARM: ARM TrustZone (2004), https://www.arm.com/technologies/trustzone-for-cortex-a
2. ARM: Mbed TLS (2009), https://tls.mbed.org
3. Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M., Goltzsche, D., Eyers, D.M., Kapitza, R., Pietzuch, P.R., Fetzer, C.: SCONE: secure linux containers with intel SGX. In: Keeton, K., Roscoe, T. (eds.) 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016. pp. 689–703. USENIX Association (2016), https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov
4. Baumann, A., Peinado, M., Hunt, G.C.: Shielding applications from an untrusted cloud with haven. ACM Trans. Comput. Syst. **33**(3), 8:1–8:26 (2015). https://doi.org/10.1145/2799647, https://doi.org/10.1145/2799647
5. Bernardy, J., Boespflug, M., Newton, R.R., Jones, S.P., Spiwack, A.: Linear haskell: practical linearity in a higher-order polymorphic language. Proc. ACM Program. Lang. **2**(POPL), 5:1–5:29 (2018). https://doi.org/10.1145/3158093, https://doi.org/10.1145/3158093
6. Biba, K.J.: Integrity considerations for secure computer systems. Tech. rep., MITRE Corp. (04 1977)
7. Chen, G., Chen, S., Xiao, Y., Zhang, Y., Lin, Z., Lai, T.: Sgxpectre: Stealing intel secrets from SGX enclaves via speculative execution. IEEE Secur. Priv. **18**(3), 28–37 (2020). https://doi.org/10.1109/MSEC.2019.2963021, https://doi.org/10.1109/MSEC.2019.2963021
8. Cowan, C., Wagle, F., Pu, C., Beattie, S., Walpole, J.: Buffer overflows: Attacks and defenses for the vulnerability of the decade. In: Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00. vol. 2, pp. 119–129. IEEE (2000)
9. Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19**(5), 236–243 (1976). https://doi.org/10.1145/360051.360056, https://doi.org/10.1145/360051.360056
10. Ekblad, A., Claessen, K.: A seamless, client-centric programming model for type safe web applications. In: Swierstra, W. (ed.) Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014. pp. 79–89. ACM (2014). https://doi.org/10.1145/2633357.2633367, https://doi.org/10.1145/2633357.2633367
11. Escalera, D.M., Agudo, I., López, J.: Private set intersection: A systematic literature review. Comput. Sci. Rev. **49**, 100567 (2023). https://doi.org/10.1016/J.COSREV.2023.100567, https://doi.org/10.1016/j.cosrev.2023.100567
12. Geppert, T., Deml, S., Sturzenegger, D., Ebert, N.: Trusted execution environments: Applications and organizational challenges. Frontiers Comput. Sci. **4** (2022). https://doi.org/10.3389/FCOMP.2022.930741, https://doi.org/10.3389/fcomp.2022.930741
13. Ghosn, A., Larus, J.R., Bugnion, E.: Secured routines: Language-based construction of trusted execution environments. In: Malkhi, D., Tsafrir, D. (eds.) 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019. pp. 571–586. USENIX Association (2019), https://www.usenix.org/conference/atc19/presentation/ghosn

14. Goguen, J.A., Meseguer, J.: Security policies and security models. In: 1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982. pp. 11–20. IEEE Computer Society (1982). https://doi.org/10.1109/SP.1982.10014, https://doi.org/10.1109/SP.1982.10014
15. Goldman, K., Perez, R., Sailer, R.: Linking remote attestation to secure tunnel endpoints. In: Proceedings of the first ACM workshop on Scalable trusted computing. pp. 21–24 (2006)
16. Gollamudi, A., Chong, S.: Automatic enforcement of expressive security policies using enclaves. In: Visser, E., Smaragdakis, Y. (eds.) Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016. pp. 494–513. ACM (2016). https://doi.org/10.1145/2983990.2984002, https://doi.org/10.1145/2983990.2984002
17. Gollamudi, A., Chong, S., Arden, O.: Information flow control for distributed trusted execution environments. In: 32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019. pp. 304–318 (2019). https://doi.org/10.1109/CSF.2019.00028, https://doi.org/10.1109/CSF.2019.00028
18. Intel: Intel SGX Intro: Passing Data Between App and Enclave (2016), https://www.intel.com/content/www/us/en/developer/articles/technical/sgx-intro-passing-data-between-app-and-enclave.html
19. Intel: tlibc - an alternative to glibc (2018), https://github.com/intel/linux-sgx/tree/master/common/inc/tlibc
20. Intel: Intel Trust Domain Extensions (2021), https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html
21. Knauth, T., Steiner, M., Chakrabarti, S., Lei, L., Xing, C., Vij, M.: Integrating remote attestation with transport layer security. arXiv preprint arXiv:1801.05863 (2018)
22. LinuxSGX: Linux SGX Remote Attestation (2017), https://github.com/svartkanin/linux-sgx-remoteattestation/blob/master/Application/isv_enclave/isv_enclave.cpp#L152-L308
23. Marlow, S., Jones, S.L.P., Singh, S.: Runtime support for multicore haskell. In: Hutton, G., Tolmach, A.P. (eds.) Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009. pp. 65–78. ACM (2009). https://doi.org/10.1145/1596550.1596563, https://doi.org/10.1145/1596550.1596563
24. McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C.V., Shafi, H., Shanbhogue, V., Savagaonkar, U.R.: Innovative instructions and software model for isolated execution. In: Lee, R.B., Shi, W. (eds.) HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 23-24, 2013. p. 10. ACM (2013). https://doi.org/10.1145/2487726.2488368, https://doi.org/10.1145/2487726.2488368
25. Microsoft: Windows cryptoapi spoofing vulnerability (2020), https://nvd.nist.gov/vuln/detail/CVE-2020-0601
26. Mo, F., Haddadi, H., Katevas, K., Marin, E., Perino, D., Kourtellis, N.: PPFL: privacy-preserving federated learning with trusted execution environments. In: Banerjee, S., Mottola, L., Zhou, X. (eds.) MobiSys '21: The 19th Annual International Conference on Mobile Systems, Applications, and Services, Virtual Event, Wisconsin, USA, 24 June - 2 July, 2021. pp. 94–108.

ACM (2021). https://doi.org/10.1145/3458864.3466628, https://doi.org/10.1145/3458864.3466628
27. Morbitzer, M., Kopf, B., Zieris, P.: Guarantee: Introducing control-flow attestation for trusted execution environments. In: 16th IEEE International Conference on Cloud Computing, CLOUD 2023, Chicago, IL, USA, July 2-8, 2023. pp. 547–553. IEEE (2023). https://doi.org/10.1109/CLOUD60044.2023.00073, https://doi.org/10.1109/CLOUD60044.2023.00073
28. Mulligan, D.P., Petri, G., Spinale, N., Stockwell, G., Vincent, H.J.M.: Confidential computing - a brave new world. In: 2021 International Symposium on Secure and Private Execution Environment Design (SEED), Washington, DC, USA, September 20-21, 2021. pp. 132–138. IEEE (2021). https://doi.org/10.1109/SEED51797.2021.00025, https://doi.org/10.1109/SEED51797.2021.00025
29. Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. ACM Trans. Softw. Eng. Methodol. **9**(4), 410–442 (2000). https://doi.org/10.1145/363516.363526, https://doi.org/10.1145/363516.363526
30. Northwood, C.: The full stack developer: your essential guide to the everyday skills expected of a modern full stack web developer. Springer (2018)
31. Oak, A., Ahmadian, A.M., Balliu, M., Salvaneschi, G.: Language support for secure software development with enclaves. In: 34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021. pp. 1–16. IEEE (2021). https://doi.org/10.1109/CSF51468.2021.00037, https://doi.org/10.1109/CSF51468.2021.00037
32. Ramsingh, A., Singer, J., Trinder, P.: Do fewer tiers mean fewer tears? eliminating web stack components to improve interoperability. CoRR **abs/2207.08019** (2022). https://doi.org/10.48550/ARXIV.2207.08019, https://doi.org/10.48550/arXiv.2207.08019
33. Russinovich, M., Costa, M., Fournet, C., Chisnall, D., Delignat-Lavaud, A., Clebsch, S., Vaswani, K., Bhatia, V.: Toward confidential cloud computing. Commun. ACM **64**(6), 54–61 (2021). https://doi.org/10.1145/3453930, https://doi.org/10.1145/3453930
34. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Sel. Areas Commun. **21**(1), 5–19 (2003). https://doi.org/10.1109/JSAC.2002.806121, https://doi.org/10.1109/JSAC.2002.806121
35. Sabelfeld, A., Sands, D.: Declassification: Dimensions and principles. J. Comput. Secur. **17**(5), 517–548 (2009). https://doi.org/10.3233/JCS-2009-0352, https://doi.org/10.3233/JCS-2009-0352
36. Sarkar, A.: Confidential private set intersection with hastee (2023), https://github.com/Abhiroop/HasTEE/blob/lio-ifc/app/Main.hs
37. Sarkar, A., Krook, R., Russo, A., Claessen, K.: HasTEE: Programming Trusted Execution Environments with Haskell. In: McDonell, T.L., Vazou, N. (eds.) Proceedings of the 16th ACM SIGPLAN International Haskell Symposium, Haskell 2023, Seattle, WA, USA, September 8-9, 2023. pp. 72–88. ACM (2023). https://doi.org/10.1145/3609026.3609731, https://doi.org/10.1145/3609026.3609731
38. Sev-Snp, A.: Strengthening vm isolation with integrity protection and more. White Paper, January **53**, 1450–1465 (2020)
39. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Ning, P., di Vimercati, S.D.C., Syverson, P.F. (eds.) Proceedings of the 2007 ACM Conference on Computer and Communications

Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007. pp. 552–561. ACM (2007), https://doi.org/10.1145/1315245.1315313
40. Shen, G., Kashiwa, S., Kuper, L.: Haschor: Functional choreographic programming for all (functional pearl). Proc. ACM Program. Lang. **7**(ICFP), 541–565 (2023). https://doi.org/10.1145/3607849, https://doi.org/10.1145/3607849
41. Sinha, R., Rajamani, S.K., Seshia, S.A., Vaswani, K.: Moat: Verifying confidentiality of enclave programs. In: Ray, I., Li, N., Kruegel, C. (eds.) Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015. pp. 1169–1184. ACM (2015), https://doi.org/10.1145/2810103.2813608
42. Stefan, D., Russo, A., Mazières, D., Mitchell, J.C.: Disjunction category labels. In: Laud, P. (ed.) Information Security Technology for Applications - 16th Nordic Conference on Secure IT Systems, NordSec 2011, Tallinn, Estonia, October 26-28, 2011, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7161, pp. 223–239. Springer (2011), https://doi.org/10.1007/978-3-642-29615-4_16
43. Stefan, D., Russo, A., Mitchell, J.C., Mazières, D.: Flexible dynamic information flow control in haskell. In: Claessen, K. (ed.) Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011. pp. 95–106. ACM (2011), https://doi.org/10.1145/2034675.2034688
44. Stefan, D., Yang, E.Z., Marchenko, P., Russo, A., Herman, D., Karp, B., Mazières, D.: Protecting users by confining javascript with COWL. In: Flinn, J., Levy, H. (eds.) 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014. pp. 131–146. USENIX Association (2014), https://www.usenix.org/conference/osdi14/technical-sessions/presentation/stefan
45. Stumpf, F., Tafreschi, O., Röder, P., Eckert, C., et al.: A robust integrity reporting protocol for remote attestation. In: Proceedings of the Workshop on Advances in Trusted Computing (WATC). p. 65 (2006)
46. Tarkhani, Z., Madhavapeddy, A.: Information flow tracking for heterogeneous compartmentalized software. In: Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2023, Hong Kong, China, October 16-18, 2023. pp. 564–579. ACM (2023), https://doi.org/10.1145/3607199.3607235
47. Vault, H.: Intel SGX deprecation review (2022), https://hardenedvault.net/blog/2022-01-15-sgx-deprecated/
48. Wang, H., Wang, P., Ding, Y., Sun, M., Jing, Y., Duan, R., Li, L., Zhang, Y., Wei, T., Lin, Z.: Towards memory safe enclave programming with rust-sgx. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019. pp. 2333–2350. ACM (2019), https://doi.org/10.1145/3319535.3354241
49. Weisenburger, P., Wirth, J., Salvaneschi, G.: A survey of multitier programming. ACM Comput. Surv. **53**(4), 81:1–81:35 (2021), https://doi.org/10.1145/3397495
50. Zegzhda, D.P., Usov, E.S., Nikol'skii, V.A., Pavlenko, E.: Use of intel SGX to ensure the confidentiality of data of cloud users. Autom. Control. Comput. Sci. **51**(8), 848–854 (2017), https://doi.org/10.3103/S0146411617080284
51. Zeldovich, N., Boyd-Wickizer, S., Kohler, E., Mazières, D.: Making information flow explicit in histar. In: Bershad, B.N., Mogul, J.C. (eds.) 7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA. pp. 263–278. USENIX Association (2006), http://www.usenix.org/events/osdi06/tech/zeldovich.html

PART II

FUNCTIONAL PROGRAMMING FOR EMBEDDED SYSTEMS

Chapter 7

Synchron - An API and Runtime for Embedded Systems

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

Keywords and phrases real-time, concurrency, functional programming, runtime, virtual machine

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.25

1 Introduction

Embedded systems are ubiquitous. They are pervasively found in application areas like IoT, industrial machinery, cars, robotics, etc. Applications running on embedded systems tend to embody three common characteristics:

1. They are *concurrent* in nature.
2. They are predominantly *I/O–bound* applications.
3. A large subset of such applications are *timing-aware*.

This list, although not exhaustive, captures a prevalent theme among embedded applications. Programming these applications involves interaction with callback-based driver APIs like the following from the Zephyr RTOS[12]:


© Abhiroop Sarkar, Bo Joel Svensson and Mary Sheeran;
licensed under Creative Commons License CC-BY 4.0

Listing 1 A callback-based GPIO driver API

```
int gpio_pin_interrupt_configure(const struct device *port
                                 , gpio_pin_t pin
                                 , gpio_flags_t flags);
void gpio_init_callback(struct gpio_callback *callback
                        , gpio_callback_handler_t handler
                        , gpio_port_pins_t pin_mask);
int gpio_add_callback(const struct device *port
                      , struct gpio_callback *callback);
```

Programming with such APIs involves expressing complex state machines in C, which often results in difficult-to-maintain and elaborate state-transition tables. Moreover, C programmers use error-prone shared-memory primitives like *semaphores* and *locks* to mediate interactions that occur between the callback-based driver handlers.

In modern microcontroller runtimes, like MicroPython [13] and Espruino (Javascript) [39], higher-order functions are used to handle callback-based APIs:

Listing 2 Driver interactions using Micropython

```
1 def callback(x):
2 #...callback body with nested callbacks...
3
4 extint = pyb.ExtInt(pin, pyb.ExtInt.IRQ_FALLING
5                     , pyb.Pin.PULL_UP, callback)
6 ExtInt.enable()
```

The function `callback(x)` from Line 1 can in turn define a callback action `callback2`, which can further define other callbacks leading to a cascade of nested callbacks. This leads to a form of *accidental* complexity, colloquially termed as *callback-hell* [22].

We present Synchron, an API that attempts to address the concerns about callback-hell and shared-memory concurrency while targeting the three characteristics of embedded programs mentioned earlier by a combination of:

1. A message-passing–based concurrency model inspired from Concurrent ML.
2. A message-passing–based I/O interface that unifies concurrency and I/O.
3. A notion of time that fits the message-passing concurrency model.

Concurrent ML (CML) [27] builds upon the synchronous message-passing–based concurrency model CSP [17] but adds the feature of composable first-class *events*. These first-class events allow the programmer to tailor new concurrency abstractions and express application-specific protocols. Additionally, a synchronous concurrency model renders linear control-flow to a program, as opposed to bottom-up, non-linear control flow exhibited by asynchronous callback-based APIs.

Synchron extends CML's message-passing API for software processes to I/O and hardware interactions by modelling the external world as a process through the `spawnExternal` operator. As a result, the standard message-passing functions such as `send, receive` etc. are applicable for handling I/O interactions, such as asynchronous driver interrupts. The overall API design allows efficient scheduling and limited power usage of programs via an associated runtime.

For timing, Synchron has the `syncT` operator, drawing inspiration from the TinyTimber kernel [21], that allows the specification of baseline and deadline windows for invocation of a method in a class. In TinyTimber, `WITHIN( B, D, &obj, meth, 123 );` expresses the desire that method `meth` should be run at the earliest at time `B` and finish within a duration of `D`. Our adaptation of this API, `syncT B D evt`, takes a baseline, deadline and a CML *event* (evt) as arguments and obeys similar semantics as `WITHIN`.

The Synchron API is implemented in the form of a bytecode-interpreted virtual machine (VM) called SynchronVM. The bytecode instructions of the VM correspond to the various operations of the Synchron API, such that any language hosted on the VM can access Synchron's concurrency, I/O and timing API for embedded systems.

Internally, the SynchronVM runtime manages the scheduling and timing of the various processes, interrupt handling, memory management, and other bookkeeping infrastructure. Notably, the runtime system features a low-level bridge interface that abstracts it from the platform-dependent specifics. The bridge translates low-level hardware interrupts or memory-mapped I/O into software messages, enabling the SynchronVM application process to use the message-passing API for low-level I/O.

Contributions

- We identify three characteristic behaviours of embedded applications, namely being (i) concurrent, (ii) I/O–bound, and (iii) timing-aware, and propose a combination of abstractions (the Synchron API) that mesh well with each other and address these requirements. We introduce the API in Section 3.
- **Message-passing–based I/O.** We present a uniform message-passing framework that combines *concurrency* and *callback–based I/O* to a single interface. A software message or a hardware interrupt is identical in our programming interface, providing the programmer with a simpler message-based framework to express concurrent hardware interactions. We show the I/O API in Section 3.2 and describe the core runtime algorithms to support this API in Section 4.
- **Declarative state machines for embedded systems.** Combining CML primitives with our I/O interface allows presenting a declarative framework to express state machines, commonly found in embedded systems. We illustrate examples of representing finite-state machines using the Synchron API in Sections 6.1 and 6.2.
- **Evaluation.** We implement the Synchron API and its associated runtime within a virtual machine, SynchronVM, described in Section 5. We illustrate the practicality and expressivity of our API by presenting three case studies in Section 6, which runs on the STM32 and NRF52 microcontroller boards. Finally, we show response time, memory and power usage, jitter rates, and load testing benchmarks on the SynchronVM in Section 7.

2 Motivation

• **Concurrency and IO.** In embedded systems, concurrency takes the form of a combination of callback handlers, interrupt service routines and possibly a threading system, for example threads as provided by ZephyrOS, ChibiOS or FreeRTOS. The callback style of programming is complicated but offers benefits when it comes to energy efficiency. Registering a callback with an Interrupt Service Routine (ISR) allows the processor to go to sleep and conserve power until the interrupt arrives.

An alternate pattern to restore the linear control flow of a program is the event-loop pattern. As the name implies, an event-loop based program involves an infinitely running loop that handles interrupts and dispatches the corresponding interrupt-handlers. An event-loop based program involves some delicate plumbing that connects its various components. Listing 3 shows a tiny snippet of the general pattern.

Listing 3 Event Loop

```
1 void eventLoop(){
2 while (1) {
```

```
 switch(GetNextEvent()) {
    case GPIO1 : GPIO1Handler();
    case GPIO2 : GPIO2Handler();
    ....
    default : goToSleep(); // no events
}}}

GPIO1Handler(){ ... } // must not block
GPIO2Handler(){ ... } // must not block

//when interrupt arrives write to event queue and wake up the while loop
GPIO1_IRQ(){....}
GPIO2_IRQ(){....}
```

Programs like the above are an improvement over callbacks, as they restore the linear control-flow of a program, which eases reasoning. However, such programs have a number of weaknesses - (i) they enforce constraints on the blocking and non-blocking behaviours of the event handlers, (ii) programmers have to hand-code elaborate plumbings between the interrupt-handlers and the event-queue, (iii) they are highly *inextensible* as extension requires code modifications on all components of the event-loop infrastructure, and (iv) they are instances of clearly concurrent programs that are written in this style due to lack of native concurrency support in the language.

Although there are extensions of C to aid the concurrent behaviour of event-loops, such as protothreads [8] or `FreeRTOS Tasks`, the first three listed problems still persist. The main infinite event loop unnecessarily induces a tight coupling between unrelated code fragments like the two separate handlers for GPIO1 and GPIO2. Additionally, this pattern breaks down the abstraction boundaries between the handlers.

• **Time.** A close relative of concurrent programming for embedded systems is *real-time programming*. Embedded systems applications such as digital sound cards routinely exhibit behaviour where the time of completion of an operation determines the correctness of the program. Real-time programs, while concurrent, differ from standard concurrent programs by allowing the programmer to override the fairness of a *fair* scheduler.

For instance, the `FreeRTOS Task` API allows a programmer to define a static *priority* number, which can override the *fairness* of a task scheduler and customise the emergency of execution of each thread. However, with a limited set of priorities numbers (1 - 5) it is very likely for several concurrent tasks to end up with the same priority, leading the scheduler to order them fairly once again. A common risk with priority-based systems is to run into the *priority inversion problem* [33], which can have fatal consequences on hard real-time scenarios. On the other hand, high-level language platforms for embedded systems such as MicroPython [13] do not provide any language support for timing-aware computations.

Problem Statement. We believe there exists a gap for a high-level language that can express concurrent, I/O–bound, and timing-aware programs for programming resource-constrained embedded systems. We outline our key idea to address this gap below.

2.1 Key Ideas

Our key idea is the Synchron API, which adopts a synchronous message-passing concurrency model and extends the message-passing functionality to all I/O interactions. Synchron also introduces *baselines* and *deadlines* for the message-passing, which consequently introduces a notion of time into the API. The resultant API is a collection of nine operations that can express (i) concurrency, (ii) I/O, and (iii) timing in a uniform and declarative manner.

The external world as processes. The Synchron API models all external drivers as

processes that can communicate with the software layer through message-passing. Synchron's `spawnExternal` operator treats an I/O peripheral as a process and a hardware interrupt as a message from the corresponding process. Fig. 1 illustrates the broad idea.

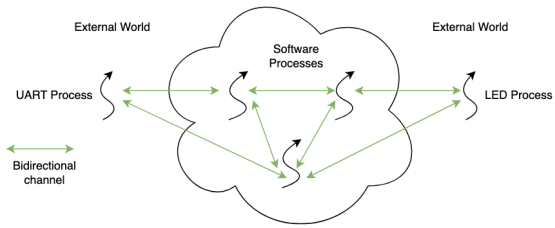


Figure 1 Software processes and hardware processes interacting

The above design relieves the programmer from writing complex callback handlers to deal with asynchronous interrupts. The synchronous message-passing–based I/O renders a linear control-flow to I/O-bound embedded-system programs, allowing the modelling of state machines in a declarative manner. Additionally, the message-passing framework simplifies the hazards of concurrent programming with shared-memory primitives (like `FreeRTOS semaphores`) and the associated perils of maintaining intricate locking protocols.

Hardware-Software Bridge. The Synchron runtime enables the seamless translation between software messages and hardware interrupts. The runtime does hardware interactions through a low-level software *bridge* interface, which is implemented atop the drivers supplied by an OS like Zephyr/ChibiOS. The *bridge* layer serialises all hardware interrupts into the format of a software message, thereby providing a uniform message-passing interaction style for both software and hardware messages. Fig. 2 shows the overall architecture of Synchron.

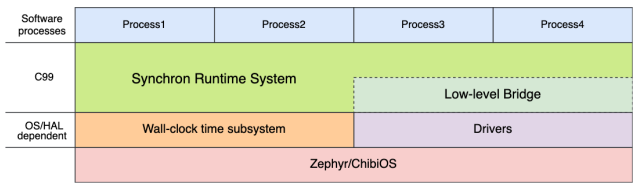


Figure 2 The Synchron Architecture

Timing. The final key component of the Synchron API is the real-time function, `syncT`, that instead of using a static priority for a thread (like Ada, RT-Java, FreeRTOS, etc.), borrows the concept of a dynamic priority specification from TinyTimber [21].

The `syncT` function allows specifying a *timing window* by stating the baseline and deadline of message communication between processes. The logical computational model of Synchron assumes to take zero time and hence the time required for communication determines the timing window of execution of the entire process. As the deadline of a process draws near, the Synchron runtime can choose to dynamically change the priority of a process while it is running. Fig. 3 illustrates the idea of dynamic priority-change.

Fig. 3 shows how a scheduler can choose to prioritise a second process over a running, *timed* process, even though the running process has a deadline in the future. In practice, a

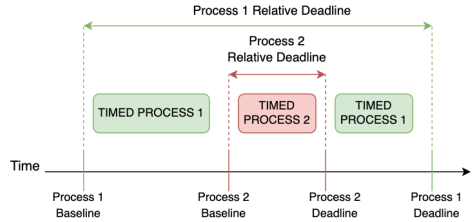


Figure 3 Dynamic priority change with `syncT`

scheduler needs to be able to pause and resume processes to support the above, which is possible in the Synchron runtime. The `syncT` function, thus, fits fairly well with the rest of the API and provides a notion of time to the overall programming interface.

The combination of `syncT`, `spawnExternal` and the CML-inspired synchronous message-passing concurrency model constitutes the Synchron API that allows declarative specification of embedded applications. We suggest that this API is an improvement, in terms of expressivity, over the currently existing languages and libraries on embedded systems and provide illustrative examples to support this in Section 6. We also provide benchmarks on the Synchron runtime in Section 7. Next, we discuss the Synchron API in more detail.

3 The Synchron API

3.1 Synchronous Message-Passing and Events

We begin by looking at a standard synchronous message-passing API, like Hoare's CSP [17] -

```
spawn   : (() -> ()) -> ThreadId
channel : () -> Channel a
sendMsg : Channel a -> a -> ()
recvMsg : Channel a -> a
```

In the above type signatures, the type parameter, `a`, indicates a polymorphic type. The call to `spawn` allows the creation of a new process whose body is represented by the $(() \rightarrow ())$ type. The `channel ()` call creates a blocking *channel* along which a process can send or receive messages using `sendMsg` and `recvMsg` respectively. A channel blocks until a sender has a corresponding receiver and vice-versa. Multi-party communication in CSP is enabled using the *nondeterministic choice* operation that races between two sets of synchronous operations and chooses the one that succeeds first.

However, there is a fundamental conflict between procedural abstraction and the *choice* operation. Using a procedure to represent a complex protocol involving multiple *sends* and *receives* hides the critical operations over which a *choice* operation is run. On the other hand, exposing the individual *sends* and *receives* prevents the construction of protocols with strict message-ordering constraints (Appendix A). Reppy discusses this issue [27] in depth and provides a solution in Concurrent ML (CML), which is adopted by the Synchron API.

The central idea is to break the act of synchronous communication into two steps:

(i) Expressing the intent of communication as an *event*-value
(ii) Synchronising between the sender and receiver via the *event*-value

The first step above results in the creation of a type of value called an `Event`. An *event* is a first-class value in the language akin to the treatment of higher-order functions in functional

languages. Reppy describes events as "first-class synchronous operations" [27]. Adapting this idea, the type signature of message sending and receiving in the Synchron API becomes :

```
send : Channel a → a → Event ()
recv : Channel a → Event a
```

Given a value of type `Event`, the second step of synchronising between processes and the consequent act of communication is accomplished via the `sync` operation, whose type is :

```
sync : Event a → a
```

Intuitively, an equivalence can be drawn between the message passing in CSP and the CML-style message passing (as adopted in the Synchron API) using function composition:

```
sync . (send c) ≡ sendMsg c
sync . recv     ≡ recvMsg
```

The advantage of representing communication as a first-class value is that event-based combinators can be used to build more elaborate communication protocols. In the same vein as higher-order functions like *function composition (.)* and *map*, events support composition via the following operators in the Synchron API :

```
choose : Event a → Event a → Event a
wrap   : Event a → (a → b) → Event b
```

The `choose` operator is equivalent to the *choice* operation in CSP and the `wrap` operator can be used to apply a post-synchronisation operation (of type $a \to b$). A large tree of *events* representing a communication protocol can be built in this framework as follows :

```
protocol : Event ()
protocol =
  choose (send c1 msg1)
         (wrap (recv c2) (λ msg2 -> sync (send c3 msg2)))
```

Using events, the above protocol involving multiple *sends* and *receives* was expressible as a procedural abstraction while still having the return type of `Event ()`. A consumer of the above protocol can further use the nondeterministic choice operator, `choose`, and *choose* among multiple protocols (see Appendix A). This combination of a composable functional programming style and CSP-inspired multiprocess program design allows this API to represent callback-based, state machine oriented programs in a declarative manner.

Comparisons between Events and Futures. The fundamental difference between events and futures is that of *deferred communication* and *deferred computation* respectively. A future aids in asynchronous computations by encapsulating a computation whose value is made available at a future time. On the other hand, an event represents deferred communication as a first-class entity in a language. Using the `wrap` combinator, it is possible to chain lambda functions capturing computations that should happen post-communication as well. However, events are fundamentally building blocks for communication protocols.

3.2 Input and Output

The I/O component of Synchron is deeply connected to the Synchron runtime. Hence, we mention parts of the low-level runtime implementation while describing the I/O interface.

In the Synchron API, I/O is expressed using the same events as are used for inter-process communication. Each I/O device is connected to the running program using a primitive we call `spawnExternal` as a hint that the programmer can think of, for example, an LED as a process that can receive messages along a channel. Each *external* process denotes an underlying I/O device that is limited to send and receive messages along one channel.

The `spawnExternal` primitive takes the channel to use for communication with software and a driver. It returns a "ThreadId" for symmetry with `spawn`.

spawnExternal : Channel a → Driver → ExternalThreadId

The first parameter supplied to `spawnExternal` is a designated fixed channel along which the external process shall communicate. The second argument requires some form of identifier to uniquely identify the driver. This identifier for a driver tends to be architecture-dependent. For instance, when using low-level memory-mapped I/O, reads or writes to a memory address are used to communicate with a peripheral. So the unique memory address would be an identifier in that case. On the other hand, certain real-time operating system (such as `FreeRTOS` or `Zephyr`) can provide more high-level abstractions over a memory address. In the Synchron runtime, we number each peripheral in a monotonically increasing order, starting from 0. So our `spawnExternal` API becomes:

```
type DriverNo = Int
spawnExternal : Channel a -> DriverNo -> ExternalThreadId
```

In the rest of the paper, we will use suggestive names for drivers like `led0`, `uart1`, etc instead of plain integers for clarity. We have ongoing work to parse a file describing the target board/MCU system, automatically number the peripherals, and emit typed declarations, like $led0 = LED\ 0$, that can be used in the `spawnExternal` API.

To demonstrate the I/O API in the context of asynchronous drivers, we present a standard example of the *button-blinky* program. The program matches a button state to an LED so that when the button is down, the LED is on, otherwise the LED is off:

Listing 4 Button-Blinky using the Synchron API

```
1  butchan = channel ()
2  ledchan = channel ()
3
4  glowled i = sync (send ledchan i)
5
6  f : ()
7  f = let _ = sync (wrap (recv butchan) glowled) in f
8
9  main =
10   let _ = spawnExternal butchan 0 in
11   let _ = spawnExternal ledchan 1 in f
```

Listing 4 above spawns two hardware processes - an LED process and a button process. It then calls the function `f` which arrives at line 7 and waits for a button press. During the waiting period, the scheduler can put the process to sleep to save power. When the button interrupt arrives, the Synchron runtime converts the hardware interrupt to a software message and wakes up process `f`. It then calls the `glowled` function on line 4 that sends a switch-on message to the LED process and recursively calls `f` infinitely.

The above program represents an asynchronous, callback-based application in an entirely synchronous framework. The same application written in C, on top of the Zephyr OS, is more than 100 lines of callback-based code [11]. A notable aspect of the above program is the lack of any non-linear callback-handling mechanism.

The structure of this program resembles the event-loop pattern presented in Listing 3 but fixes all of its associated deficiencies - (1) all Synchron I/O operations are blocking; the runtime manages their optimal scheduling, not the programmer, (2) the internal plumbing related to interrupt-handling and queues are invisible to the programmer, (3) the program is highly extensible; adding a new interrupt handler is as simple as defining a new function.

3.3 Programming with Time

Real-time languages and frameworks generally provide a mechanism to override the fairness of a *fair* scheduler. A typical fair scheduler abstracts away the details of prioritising processes.

However, in a real-time scenario, a programmer wants to precisely control the response-time of certain operations. So the natural intuition for real-time C-extensions like FreeRTOS *Tasks* or languages like Ada is to delegate the scheduling control to the programmer by allowing them to attach a priority level to each process.

The priority levels involved decides the order in which a tie is broken by the scheduler. However, with a small fixed number of priority levels it is likely for several processes to end up with the same priority, leading the scheduler to order them fairly again within each level.

Another complication that crops up in the context of priorities is the *priority inversion problem* [33]. Priority inversion is a form of resource contention where a high-priority thread gets blocked on a resource held by a low-priority thread, thus allowing a medium priority thread to take advantage of the situation and get scheduled first. The outcome of this scenario is that the high-priority thread gets to run after the medium-priority thread, leading to possible program failures.

The Synchron API admits the *dynamic* prioritisation of processes, drawing inspiration from the TinyTimber kernel [23]. TinyTimber allows specifying a *timing window* expressed as a baseline and deadline time, and a scheduler can use this timing window to determine the runtime priority of a process. The timing window expresses the programmer's wish that the operation is started at the *earliest* on the baseline and *no later* than the deadline.

In Synchron, a programmer specifies a *timing window* (of the wall-clock time) during which they want message synchronisation, that is the rendezvous between message sender and receiver, to happen. We do this with the help of the timed-synchronisation operator, `syncT`, with type signature:

syncT : Time → Time → Event a → a

Comparing the type signature of `syncT` with that of `sync` :

```
syncT : Time -> Time -> Event a -> a
sync  :                 Event a -> a
```

The two extra arguments to `syncT` specify a lower and upper bound on the *time of synchronisation* of an event. The two arguments to `syncT`, of type `Time`, express the relative times calculated from the current wall-clock time. The first argument represents the *relative baseline* - the earliest time instant from which the event synchronisation should begin. The second argument specifies the *relative deadline*, i.e. the latest time instant (starting from the baseline), by which the synchronisation should start. For instance,

```
syncT (msec 50) (msec 20) timed_ev
```

means that the event, *timed_ev*, should begin synchronisation at the earliest 50 milliseconds and the latest $50 + 20$ milliseconds from *now*. The *now* concept is based on a thread's

local view of what time it is. This thread-local time (T_{local}) is always less than or equal to wall-clock time ($T_{absolute}$). When a thread is spawned, its thread-local time, T_{local}, is set to the wall-clock time, $T_{absolute}$.

While a thread is running, its local time is frozen and unchanged until the thread executes a timed synchronisation; a `syncT` operation where time progresses to $T_{local} + baseline$.

```
process1 _ =
  let _ = s1 in -- T_local = 0
  let _ = s2 in -- T_local = 0
  let _ = syncT (msec 50) (usec 10) ev1 in
  process1 () -- T_local = 50 msec
```

The above illustrates that the *untimed* operations `s1` and `s2` have no impact on a thread's view of what time it is. In essence, these operations are considered to take no time, which is a reference to *logical* time and not the physical time. Synchron shares this logical computational model with other systems such as the synchronous languages [7] and ChucK [37].

In practice, this assumption helps control jitter in the timing as long as the timing windows specified on the synchronisation is large enough to contain the execution time of `s1`, `s2`, the synchronisation step and the recursive call. Local and absolute time must meet up at key points for this approach to work. Without the two notions of time meeting, local time would drift away from absolute time in an unbounded fashion. For a practical implementation of `syncT`, a scheduler needs to meet the following requirements:

- The scheduler should provide a mechanism for overriding fair scheduling.
- The scheduler must have access to a wall-clock time source.
- A scheduler should attempt to schedule synchronisation such that local time meets up with absolute time at that instant.

We shall revisit these requirements in Section 5 when describing the scheduler within the Synchron runtime. Next, we shall look at a simple example use of `syncT`.

Blinky

The well-known *blinky* example, shown in Listing 5, involves blinking an LED on and off at a certain frequency. Here we blink once every second.

Listing 5 Blinky with `syncT`

```
1  not 1 = 0
2  not 0 = 1
3
4  ledchan = channel ()
5
6  sec  n = n * 1000000
7  usec n = n -- the unit-time in the Synchron runtime
8
9  foo : Int -> ()
10 foo val =
11  let _ = syncT (sec 1) (usec 1) (send ledchan val) in
12  foo (not val)
13
14 main = let _ = spawnExternal ledchan 1 in foo 1
```

In the above program, `foo` is the only software process, and there is one external hardware process for the LED driver that can be communicated with, using the `ledChan` channel. Line 11 is the critical part of the logic that sets the period of the program at 1 second, and the recursion at Line 12 keep the program alive forever. Appendix B shows the details of scheduling this program. We discuss a more involved example using `syncT` in Section 6.3.

4 Synchronisation Algorithms

The synchronous nature of message-passing is the foundation of the Synchron API. In this section, we describe the runtime algorithms, in an abstract form, that enable processes to synchronise. The Synchron runtime implements these algorithms, which drives the scheduling of the various software processes.

In Synchron, we synchronise on events. **Events**, in our API, fall into the categories of *base* events and *composite* events. The base events are `send` and `recv` and events created using `choose` are composite.

```
composite_event = choose (send c1 m1) (choose (send c2 m2) (send c3 m3))
```

From the API's point of view, composite events resemble a tree with base events in the leaves. However, for the algorithm descriptions here, we consider an event to be a *set* of base events. An implementation could impose an ordering on the base events that make up a composite event. Different orderings correspond to different event-prioritisation algorithms.

In the algorithm descriptions below, a **Channel** consists of two FIFO queues, one for `send` and one for `recv`. On these queues, process identities are stored. While blocked on a `recv` on a channel, that process' id is stored in the receive queue of that channel; likewise for `send` and the send-queue. Synchronous exchange of the message means that messages themselves do not need to be maintained on a queue.

Additionally, the algorithms below rely on there being a queue of processes that are ready to execute. This queue is called the `readyQ`. In the algorithm descriptions below, handling of `wrap` has been omitted. A function wrapped around an event specifies an operation that should be performed after synchronisation has been completed. Also, we abstract over the synchronisation of hardware events. As a convention, `self` used in the algorithms below refers to the process from which the `sync` operation is executed.

4.1 Synchronising events

The synchronisation algorithm, that performs the API operation `sync`, accepts a set of base events. The algorithm searches the set of events for a base event that has a sender or receiver blocked (ready to synchronise) and passes the message between sender and receiver. Algorithm 1 provides a high-level view of the synchronisation algorithm.

The first step in synchronisation is to see if there exists a synchronisable event in the set of base events. The *findSynchronisableEvent* algorithm is presented in Algorithm 2.

If the *findSynchronisableEvent* algorithm is unable to find an event that can be synchronised, the process initiating the synchronisation is blocked. The process identifier then gets added to all the channels involved in the base events of the set. This is shown in Algorithm 3.

After registering the process identifiers on the channels involved, the currently running process should yield its hold on the CPU, allowing another process to run. The next process to start running is found using the *dispatchNewProcess* algorithm in Algorithm 4.

When two processes are communicating, the first one to be scheduled will block as the other participant in the communication is not yet waiting on the channel. However, when *dispatchNewProcess* dispatches the second process, the *findSynchronisableEvent* function will return a synchronisable event and the *syncNow* operation does the actual message passing. The algorithm of *syncNow* is given in Algorithm 5 below.

Algorithm 1 The synchronisation algorithm

```
Data: event : Set
ev ← findSynchronisableEvent(event);
if ev ≠ ∅ then
    syncNow(ev);
else
    block(event);
    dispatchNewProcess();
end
```

Algorithm 2 The findSynchronisableEvent function

```
Data: event : Set
Result: A synchronisable event or ∅
foreach e ∈ event do
    if e.baseEventType == SEND then
        if ¬isEmpty(e.channelNo.recvq) then
            return e
        end
    else if e.baseEventType == RECV then
        if ¬isEmpty(e.channelNo.sendq) then
            return e
        end
    else return ∅;                                  /* Impossible case */
end
return ∅ ;                              /* No synchronisable event found */
```

Algorithm 3 The block function

```
Data: event : Set
foreach e ∈ event do
    if e.baseEventType == SEND then
        e.channelNo.sendq.enqueue(self);
    else if e.baseEventType == RECV then
        e.channelNo.recvq.enqueue(self);
    else Do nothing;                                /* Impossible case */
end
```

Algorithm 4 The dispatchNewProcess function

```
if readyQ ≠ ∅ then
    process ← dequeue(readyQ);
    currentProcess = process;
else
    relinquish control to the underlying OS
end
```

Algorithm 5 The syncNow function

Data: A base-event value - *event*
if *event.baseEventType* == *SEND* **then**
  receiver ← *dequeue(event.channelNo.recvq)*;
  deliverMSG(self, receiver, msg) ; `/* pass msg from self to receiver */`
  readyQ.enqueue(self);
else if *event.baseEventType* == *RECV* **then**
  sender ← *dequeue(event.channelNo.sendq)*;
  deliverMSG(sender, self, msg) ; `/* pass msg from sender to self */`
  readyQ.enqueue(sender);
else Do nothing; `/* Impossible case */`

4.2 Timed synchronisation of events

Now, we look at the *timed* synchronisation algorithms. Timed synchronisation is handled by a two-part algorithm - the first part (Algorithm 6) runs when a process is executing the `syncT` API operation, and the second part (Algorithm 7) is executed later, after the baseline time specified in the `syncT` call is reached.

These algorithms rely on there being an alarm facility based on absolute wall-clock time, which invokes Algorithm 7 at a specific time. The alarm facility provides the operation `setAlarm` used in the algorithms below. The algorithms also require a queue, `waitQ`, to hold processes waiting for their baseline time-point.

Algorithm 6 The time function

Data: Relative Baseline = *baseline*, Relative Deadline = *deadline*
$T_{wakeup} = self.T_{local} + baseline$;
if $deadline == 0$ **then**
  $T_{finish} = Integer.MAX$; `/* deadline = 0 implies no deadline */`
else
  $T_{finish} = T_{wakeup} + deadline$;
end
$self.deadline = T_{finish}$;
$baseline_{absolute} = T_{absolute} + baseline$;
$deadline_{absolute} = T_{absolute} + baseline + deadline$;
$cond1 = T_{absolute} > deadline_{absolute}$;
$cond2 = (T_{absolute} \geq baseline_{absolute})\&\&(T_{absolute} \leq deadline_{absolute})$;
$cond3 = baseline < \epsilon$; `/* platform dependent small time period */`
if $baseline == 0 \vee cond1 \vee cond2 \vee cond3$ **then**
  $readyQ.enqueue(currentThread)$;
  $dispatchNewProcess()$;
  $return$;
end
$setAlarm(T_{wakeup})$;
$waitQ.enqueue(self).orderBy(T_{wakeup})$;
$dispatchNewProcess()$;

The `handleAlarm` function in Algorithm 7 runs when an alarm goes off and, at that point,

makes a process from the waitQ ready for execution. When the alarm goes off, there could be a process running already that should either be preempted by a timed process with a tight deadline or be allowed to run to completion in case its deadline is the tightest. The other alternative is that there is no process running and the process acquired from the waitQ can immediately be made active.

Algorithm 7 The handleAlarm function

```
Data: Wakeup Interrupt
timedProcess ← dequeue(waitQ);
T_now = timedProcess.baseline;
timedProcess.T_local = T_now;
if waitQ ≠ ∅ then
    timedProcess_2 ← peek(waitQ);                    /* Does not dequeue */
    setAlarm(timedProcess_2.baseline);
end
if currentProcess == ∅;                  /* No process currently running */
 then
    currentProcess = timedProcess;
else
    if timedProcess.deadline < currentProcess.deadline then
        /* Preempt currently running process                              */
        readyQ.enqueue(currentProcess);
        currentProcess = timedProcess;
    else
        /* Schedule timed process to run after currentProcess             */
        readyQ.enqueue(timedProcess);
        currentProcess.T_local = T_now;        /* Avoids too much time drift */
    end
end
```

5 Implementation in SynchronVM

The algorithms of Section 4 are implemented within the Synchron runtime. The Synchron API and runtime are part of a larger programming platform that is the bytecode-interpreted virtual machine called SynchronVM, which builds on the work by Sarkar et al. [28].

The execution unit of SynchronVM is based on the Categorical Abstract Machine (CAM) [6]. CAM supports the cheap creation of closures, as a result of which SynchronVM can support a functional language quite naturally. CAM was chosen primarily for its simplicity and availability of pedagogical resources [16].

5.1 System Overview

Figure 4 shows the architecture of SynchronVM. The whole pipeline consists of three major components - (i) the frontend, (ii) the middleware and (iii) the backend.

Frontend. We support a statically-typed, eagerly-evaluated, Caml-like functional language on the frontend. The language comes equipped with Hindley-Milner type inference. The

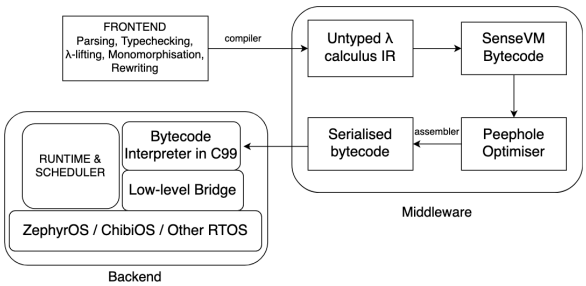


Figure 4 The compiler and runtime for SynchronVM

polymorphic types are monomorphised at compile-time. The frontend module additionally runs a lambda-lifting pass to reduce the heap-allocation of closures.

Middleware. The frontend language gets compiled to an untyped lambda-calculus-based intermediate representation. This intermediate representation is then further compiled down to the actual bytecode that gets interpreted at run time. The generated bytecode emulates operations on a stack machine with a single environment register that holds the final value of a computation. This module generates specialised bytecodes that reduce the environment register-lookup using an operational notion called *r-free* variables described by Hinze [16]. On the generated bytecode, a peephole-optimisation pass applies further optimisations, such as β-reduction and *last-call optimisation* [16] (a generalisation of tail-call elimination).

Backend. The backend module is the principal component of the SynchronVM. It can be further classified into four components - (i) the bytecode interpreter, (ii) the runtime, (iii) a low-level *bridge* interface, and (iv) underlying OS/driver support.

Interpreter. The interpreter is written in C99 for portability. Currently, we use a total of 55 bytecodes operating on a stack machine.

Runtime. The runtime consists of a fixed-size stack with an environment register. A thread/process is emulated via the notion of a *context*, which holds a fixed-size stack, an environment register and a program counter to indicate the bytecode that is being interpreted. The runtime supports multiple but a *fixed* number of contexts, owing to memory constraints. A context holds two additional registers - one for the process-local clock (T_{local}) and the second to hold the deadline of that specific context (or thread).

The runtime has a garbage-collected heap to support closures and other composite values like tuples, algebraic data types, etc. The heap is structured as a list made of uniformly-sized tuples. For garbage collection, the runtime uses a standard *non-moving*, mark-and-sweep algorithm with the Hughes lazy-sweep optimisation [18].

Low-level Bridge. The runtime uses the low-level bridge interface functions to describe the various I/O-related interactions. It connects the runtime with the lowest level of the hardware. We discuss it in depth in Section 5.4.

Underlying OS/drivers. The lowest level of SynchronVM uses a real-time operating system that provides drivers for interacting with the various peripherals. The low-level is not restricted to use any particular OS, as shall be demonstrated in our examples using both the Zephyr-OS and ChibiOS.

5.1.1 Concurrency, I/O and Timing bytecode instructions

For accessing the operators of our programming interface as well as any general runtime-based operations, SynchronVM has a dedicated bytecode instruction - `CALLRTS n`, where `n` is a natural number to disambiguate between operations. Table 1 shows the bytecode operations corresponding to our programming interface.

spawn	CALLRTS 0
channel	CALLRTS 1
send	CALLRTS 2

recv	CALLRTS 3
sync	CALLRTS 4
choose	CALLRTS 5

spawnExternal	CALLRTS 6
wrap	CALLRTS 7
syncT	CALLRTS 8; CALLRTS 4

Table 1 Concurrency, I/O and Timing bytecodes

A notable aspect is the handling of the `syncT` operation, which gets compiled into a sequence of two instructions. The first instruction in the `syncT` sequence is `CALLRTS 8` which corresponds to Algorithm 6 in Section 4. When the process is woken up by Algorithm 7, the process program counter lands at the next instruction which is `CALLRTS 4` (`sync`).

5.2 Message-passing with events

All forms of communication and I/O in SynchronVM operate via synchronous message-passing. However, a distinct aspect of SynchronVM's message-passing is the separation between the *intent* of communication and the actual communication. A value of type `Event` indicates the intent to communicate.

An event-value, like a closure, is a concrete runtime value allocated on the heap. The fundamental event-creation primitives are `send` and `recv`, which Reppy calls base-event constructors [27]. The event composition operators like `choose` and `wrap` operate on these base-event values to construct larger events. When a programmer attempts to send or receive a message, an event-value captures the channel number on which the communication was desired. When this event-value is synchronised (Section 4), we use the channel number as an identifier to match between prospective senders and receivers. Listing 6 shows the heap representation of an event-value as the type `event_t` and the information that the event-value captures on SynchronVM.

Listing 6 Representing an `Event` in SynchronVM

```
typedef enum {
   SEND, RECV
} event_type_t;

typedef struct {
  event_type_t e_type; //  8 bits
  UUID channel_id;     //  8 bits
} base_evt_simple_t;

typedef struct {
  base_evt_simple_t  evt_details; // stored with 16 bits free
  cam_value_t wrap_func_ptr; // 32 bits
} base_event_t;


typedef struct {
  base_event_t bev; // 32 bits
  cam_value_t  msg; // 32 bits; NULL for recv
} cam_event_t;

typedef heap_index event_t;
```

An event is implemented as a linked list of base-events constructed by applications of the `choose` operation. Each element of the list captures (i) the message that is being sent or received, (ii) any function that is wrapped around the base-event using `wrap`, (iii) the channel being used for communication and (iv) an enum to distinguish whether the base-event is a `send` or `recv`. Fig 5 visualises an event upon allocation to the Synchron runtime's heap.

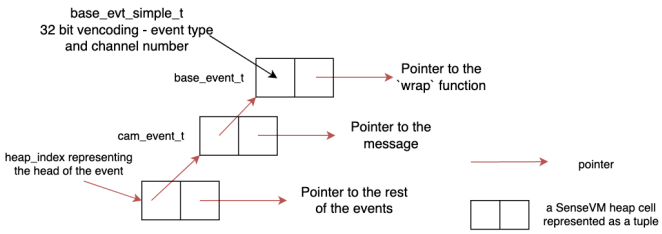


Figure 5 An event on the SynchronVM heap

The linked-list, as shown above, is the canonical representation of an `Event`-type. It can represent any complex composite event. For instance, if we take an example composite event that is created using the base-events, e_1, e_2, e_3 and a wrapping function wf_1, it can always be rewritten to its canonical form.

```
choose e1 (wrap (choose e2 e3) wf1)

-- Rewrite to canonical form --

choose e1 (choose (wrap e2 wf1) (wrap e3 wf1))
```

The `choose` operation can simply be represented as *cons*ing onto the event list.

5.3 The scheduler

SynchronVM's scheduler is a hyrbid of cooperative and preemptive scheduling. For applications that do not use `syncT`, the scheduler is cooperative in nature. Initially the threads are scheduled in the order that the main method calls them. For example,

```
main =
 let _ = spawn thread1 in
 let _ = spawn thread2 in
 let _ = spawn thread3 in ...
```

The scheduler orders the above in the order of `thread1` first, `thread2` next and `thread3` last. As the program proceeds, the scheduler relies on the threads to yield their control according to the algorithms of Section 4. When the scheduler is unable to find a matching thread for the currently running thread that is ready to synchronise the communication, it blocks the current thread and calls the *dispatchNewProcess*() function to run other threads (see Algorithm 1). On the other hand, when synchronisation succeeds, the scheduler puts the message-sending thread in the `readyQ` and the message-receiving thread starts running.

The preemptive behaviour of the scheduler occurs when using `syncT`. For instance, when a particular *untimed* thread is running and the baseline time of a timed thread has arrived, the scheduler then preempts the execution of the *untimed* thread and starts running the timed thread. A similar policy is also observed when the executing thread's deadline is later than a more urgent thread; the thread with the earliest deadline is chosen to be run at that instance. Algorithm 7 shows the preemptive components of the scheduler.

The SynchronVM scheduler also handles hardware driver interactions via message-passing. The structure that is used for messaging is shown below:

Listing 7 A SynchronVM hardware message

```
typedef struct {
  uint32_t sender_id;
  uint32_t msg_type;
  uint32_t data;
  Time timestamp;
} svm_msg_t;
```

The `svm_msg_t` type contains a unique sender id for each driver that is the same as the number used in `spawnExternal` to identify that driver. The 32 bit `msg_type` field can be used to specify different meanings for the next field, the `data`. The `data` is a 32 bit word. The `timestamp` field of a message struct is a 64 bit entity, explained in detail in Section 5.5.

When the SynchronVM scheduler has all threads blocked, it uses a function pointer called `blockMsg`, which is passed to it by the OS that starts the scheduler, to wait for any interrupts from the underlying OS (more details in Section 5.4). Upon receiving an interrupt, the scheduler uses the SynchronVM runtime's `handleMsg` function to handle the corresponding message. The function internally takes the message and unblocks the thread for which the message was sent. The general structure of SynchronVM's scheduler is shown in Algorithm 8.

Algorithm 8 The SynchronVM scheduler

Data: $blockMsg$ function pointer
$\forall threads\ set\ T_{local} = T_{absolute}$;
$svm\_msg\_t\ msg$;
while *True* **do**
  if *all threads blocked* **then**
    $blockMsg(\&msg)$; /* Relinquish control to OS */
    $handleMsg(msg)$;
  else
    $interpret(currentThread.PC)$;
  end
end

The T_{local} clock is initialised for each thread when starting up the scheduler. Also notable is the `blockMsg` function that relinquishes control to the underlying OS, allowing it to save power. When the interrupt arrives, the `handleMsg` function unblocks certain thread(s) so that when the $if..then$ clause ends, in the following iteration the *else* clause is executed and bytecode interpretation continues. We next discuss the low-level bridge connecting Synchron runtime to the underlying OS.

5.4 The Low-Level Bridge

The low-level bridge specifies two interfaces that should be implemented when writing peripheral drivers to use with SynchronVM. The first contains functions for reading and writing data synchronously to and from a driver. The second is geared towards interrupt-based drivers that asynchronously produce data.

The C-`struct` below contains the interface functions for reading and writing data to a driver as well as functions for checking the availability of data.

```
typedef struct ll_driver_s{
  void *driver_info;
  bool is_synchronous;
  uint32_t (*ll_read_fun)(struct ll_driver_s *this, uint8_t*, uint32_t);
  uint32_t (*ll_write_fun)(struct ll_driver_s *this, uint8_t*, uint32_t);
  uint32_t (*ll_data_readable_fun)(struct ll_driver_s* this);
  uint32_t (*ll_data_writeable_fun)(struct ll_driver_s* this);

  UUID channel_id;
} ll_driver_t;
```

The `driver_info` field in the `ll_driver_t` struct can be used by a driver that implements the interface to keep a pointer to lower-level driver specific data. For interrupt-based drivers, this data will contain, among other things, an *OS interoperation* struct. These OS interoperation structs are shown further below. A boolean indicates whether the driver is synchronous or not. Next, the struct contains function pointers to the low-level driver's implementation of the interface. Lastly, a `channel_id` identifies the channel along which the driver is allowed to communicate with processes running on top of SynchronVM.

The `ll_driver_t` struct contains all the data associated with a driver's configuration in one place and defines a set of platform and driver independent functions for use in the runtime system, shown below:

```
uint32_t ll_read(ll_driver_t *drv, uint8_t *data, uint32_t data_size);
uint32_t ll_write(ll_driver_t *drv, uint8_t *data, uint32_t data_size);
uint32_t ll_data_readable(ll_driver_t *drv);
uint32_t ll_data_writeable(ll_driver_t *drv);
```

The OS interoperation structs, mentioned above, are essential for drivers that asynchronously produce data. We show their Zephyr and ChibiOS versions below:

```
typedef struct zephyr_interop_s {
  struct k_msgq *msgq;
  int (*send_message)(struct zephyr_interop_s* this, svm_msg_t msg);
} zephyr_interop_t;
```

```
typedef struct chibios_interop_s {
  memory_pool_t *msg_pool;
  mailbox_t *mb;
  int (*send_message)(struct chibios_interop_s* this, svm_msg_t msg);
} chibios_interop_t;
```

In both cases, the struct contains the data that functions need to set up low-level message-passing between the driver and the OS thread running the SynchronVM runtime. Zephyr provides a message-queue abstraction that can take fixed-size messages, while ChibiOS supports a mailbox abstraction that receives messages that are the size of a pointer. Since ChibiOS mailboxes cannot receive data that is larger than a 32-bit word, a memory pool of messages is employed in that case. The structure used to send messages from the drivers is the already-introduced `svm_msg_t` struct, given in Listing 7.

The scheduler also needs to handle alarm interrupts from the wall-clock time subsystem, arising from the `syncT` operation. The next section discusses that component of SynchronVM.

5.5 The wall-clock time subsystem

Programs running on SynchronVM that make use of the timed operations rely on there being a monotonically increasing timer. The wall-clock time subsystem emulates this by implementing a 64bit timer that would take almost 7000 years to overflow at 84MHz frequency or about 36000 years at 16MHz. The timer frequency of 16MHz is used on the NRF52 board, while the timer runs at 84MHz on the STM32.

SynchronVM requires the implementation of the following functions for each of the platforms (such as ChibiOS and Zephyr) that it runs on :

```
bool      sys_time_init(void *os_interop);
Time      sys_time_get_current_ticks(void);
uint32_t  sys_time_get_alarm_channels(void);
uint32_t  sys_time_get_clock_freq(void);
bool      sys_time_set_wake_up(Time absolute);
Time      sys_time_get_wake_up_time(void);
bool      sys_time_is_alarm_set(void);
```

The timing subsystem uses the same OS interoperation structs as drivers do and thus has access to a communication channel to the SynchronVM scheduler. The interoperation is provided to the subsystem at initialisation using `sys_time_init`.

The key functionality implemented by the timing subsystem is the ability to set an alarm at an absolute 64-bit point in time. Setting an alarm is done using `sys_time_set_wake_up`. The runtime system can also query the timing subsystem to check if an alarm is set and at what specific time.

The low-level implementation of the timing subsystem is highly platform dependent at present. But on both Zephyr and ChibiOS, the implementation is currently based on a single 32-bit counter configured to issue interrupts at overflow, where an additional 32-bit value is incremented. Alarms can only be set on the lower 32-bit counter at absolute 32-bit values. Additional logic is needed to translate between the 64-bit alarms set by SynchronVM and the 32-bit timers of the target platforms. Each time the overflow interrupt happens, the interrupt service routine checks if there is an alarm in the next 32-bit window of time and in that case, enables a compare interrupt to handle that alarm. When the alarm interrupt happens, a message is sent to the SynchronVM scheduler in the same way as for interrupt based drivers, using the message queue or mailbox from the OS interoperation structure.

Revisiting the requirements for implementing `syncT` (Section 3.3), we find that our scheduler (1) provides a preemptive mechanism to override the fair scheduling, (2) has access to a wall-clock time source, and (3) implements an earliest-deadline-first scheduling policy that attempts to match the local time and the absolute time.

5.6 Porting SynchronVM to another RTOS

For porting SynchronVM to a new RTOS, one needs to implement - (1) the wall-clock time subsystem interface from Section 5.5, (2) the low-level bridge interface (Section 5.4) for each peripheral, and (3) a mailbox or message queue for communication between asynchronous drivers and the runtime system, required by the time subsystem.

Our initial platform of choice was ZephyrOS for its platform-independent abstractions. The first port of SynchronVM was on ChibiOS, where the wall-clock time subsystem was 254 lines of C-code. The drivers for LED, PWM, and DAC were about 100 lines of C-code each.

6 Case Studies

Finite-State Machines with Synchron

We will begin with two examples of expressing state machines (involving callbacks) in the Synchron API. Our examples are run on the `NRF52840DK` microcontroller board containing four buttons and four LEDs. We particularly choose the button peripheral because its drivers have a callback-based API that typically leads to non-linear control-flows in programs.

6.1 Four-Button-Blinky

We build on the *button-blinky* program from Listing 4 presented in Section 3.2. The original program, upon a single button-press, would light up an LED corresponding to that press and switch off upon the button release. We now extend that program to produce a one-to-one mapping between four LEDs and four buttons such that button1 press lights up LED1, button2 lights up LED2, button3 lights up LED3 and button4 lights up LED4 (while the button releases switch off the corresponding LEDs).

The state machine of button-blinky is a standard two-state automaton that moves from the ON-state to OFF on button-press and vice versa. Now, for the four button-LED combinations, we have four state machines. We can combine them using the `choose` operator.

Listing 8 shows the important parts of the logic. The four state machines are declared in Lines 1 to 4, and their composition happens in Line 6 using the `choose` operator. See Appendix C.1 for the full program.

Listing 8 The Four-Button-Blinky program expressed in the Synchron API

```
1 press1 = wrap (recv butchan1) (λ x -> sync (send ledchan1 x))
2 press2 = wrap (recv butchan2) (λ x -> sync (send ledchan2 x))
3 press3 = wrap (recv butchan3) (λ x -> sync (send ledchan3 x))
4 press4 = wrap (recv butchan4) (λ x -> sync (send ledchan4 x))
5
6 anybutton = choose press1 (choose press2 (choose press3 press4))
7
8 program : ()
9 program = let _ = sync anybutton in program
```

6.2 A more intricate FSM

We now construct a more intricate finite-state machine involving intermediate states that can move to an error state if the desired state-transition buttons are not pressed. For this example a button driver needs to be configured to send only one message per button press-and-release. So there is no separate button-on and button-off signal but one signal per button.

In this FSM, we glow the LED1 upon consecutive presses of button1 and button2. We use the same path to turn LED1 off. However, if a press on button1 is followed by a press of button 1 or 3 or 4, then we move to an error state indicated by LED3. We use the same path to switch off LED3. In a similar vein, consecutive presses of button3 and button4 turns on LED2 and button3 followed by button 1 or 2 or 3 turns on the error LED - LED3. Fig. 6 shows the FSM diagram of this application, omitting self-loops in the OFF state.

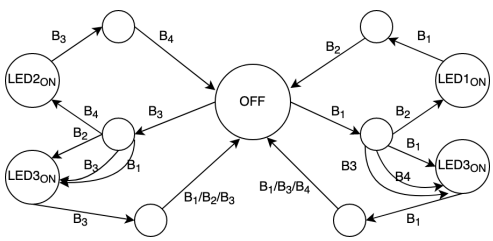


Figure 6 A complex state machine

Listing 9 shows the central logic expressing the FSM of Fig 6 in the Synchron API (see Appendix C.2 for the full program). This FSM can be viewed as a composition of two

separate finite state machines, one on the left side of the OFF state involving LED2 and LED3 and one on the right side involving LED1 and LED3. Once again, we use the `choose` operator to compose these two state machines.

Listing 9 The complex state machine running on the SynchronVM

```
errorLed x = ledchan3

fail1ev = choose (wrap (recv butchan1) errorLed)
         (choose (wrap (recv butchan3) errorLed)
                 (wrap (recv butchan4) errorLed))

fail2ev = choose (wrap (recv butchan1) errorLed)
         (choose (wrap (recv butchan2) errorLed)
                 (wrap (recv butchan3) errorLed))

led1Handler x =
   sync (choose (wrap (recv butchan2) (\x -> ledchan1)) fail1ev)

led2Handler x =
   sync (choose (wrap (recv butchan4) (\x -> ledchan2)) fail2ev)

led : Int -> ()
led state =
  let fsm1 = wrap (recv butchan1) led1Handler in
  let fsm2 = wrap (recv butchan3) led2Handler in
  let ch = sync (choose fsm1 fsm2) in
  let _ = sync (send ch (not state)) in
  led (not state)
```

In Listing 9, the `led1Handler1` and `ledHandler2` functions capture the intermediate states after one button press, when the program awaits the next button press. The error states are composed using the `choose` operator in the functions `fail1ev` and `fail2ev`.

The compositional nature of our framework is visible in line no. 21 where we compose the two state machines, `fsm1` and `fsm2`, using the `choose` operator. Synchronising on this composite event returns the LED channel (demonstrating a higher-order approach) on which the process should attempt to write. This program is notably a highly callback-based, reactive program that we have managed to represent in an entirely synchronous framework.

6.3 A soft-realtime music playing example

We present a soft-realtime music playing exercise from a Real-Time Systems course, expressed using the Synchron API. We choose the popular nursery rhyme - "Twinkle, Twinkle, Little Star". The program plays the tune repeatedly until it is stopped.

The core logic of the program involves periodically writing a sequence of 1's and 0's to a DAC driver. However, to make the produced note sound musical to the human ear, the *periodic rate* at which our process writes to the DAC driver is important, and this is where the real-time aspect of the application comes in. The human ear recognises a note produced at a certain frequency as a musical note. Our sound is generated at the 196Hz G3 music key.

Listing 10 shows the principal logic of the program expressed using the Synchron API. Note that we use `syncT` to describe a new temporal combinator `after` that determines the periodicity of this program. The list `twinkle` (line 17) holds the 28 notes in the twinkle song and the list `durations` (line 18) provides the length of each note. Appendix C provides full details of the program.

Listing 10 The *Twinkle, Twinkle* tune expressed using the Synchron API

```
msec t = t * 1000
usec t = t
```

```
after t ev = syncT t 0 ev

-- note frequencies
g = usec 2551
a = usec 2273
b = usec 2025
c = usec 1911
d = usec 1703
e = usec 1517

hn = msec 1000 -- half note
qn = msec  500 -- quarter note

twinkle, durations : List Int
twinkle   = [ g,  g,  d,  d,  e,  e,  d.... ] -- 28 notes
durations = [qn, qn, qn, qn, qn, qn, hn.... ]

dacC  = channel ()
noteC = channel ()

playerP : List Int -> List Int -> Int -> () -> ()
playerP melody nt n void =
  if (n == 29)
  then let _ = after (head nt) (send noteC (head twinkle)) in
       playerP (tail twinkle) durations 2 void
  else let _ = after (head nt) (send noteC (head melody)) in
       playerP (tail melody) (tail nt) (n + 1) void

tuneP : Int -> Int -> () -> ()
tuneP timePeriod vol void =
  let newtp =
      after timePeriod (choose (recv noteC)
                               (wrap (send dacC (vol * 4095))
                                     (λ _ -> timePeriod))) in
  tuneP newtp (not vol) void

main =
  let _ = spawnExternal dacC 0 in
  let _ = spawn (tuneP (head twinkle) 1) in
  let _ = spawn (playerP (tail twinkle) durations 2) in ()
```

The application consists of two software processes and one external hardware process. We use two channels - `dacC` to communicate with the DAC and `noteC` to communicate between the two software processes. Looking at what each software process is doing -

playerP. This process runs at the rate of a note's length. For a quarter note it wakes up after 500 milliseconds (1000 msecs for a half note), traverses the next element of the `twinkle` list and sends it along the `noteC` channel. It circles back after completing all 28 notes.

tuneP. This process creates the actual sound. Its running rate varies depending on the note that is being played. For instance, when playing note C, it will write to the DAC at a rate of 1911 microseconds-per-write. However, upon receiving a new value along `noteC`, it changes its write frequency to the new value resulting in changing the note of the song.

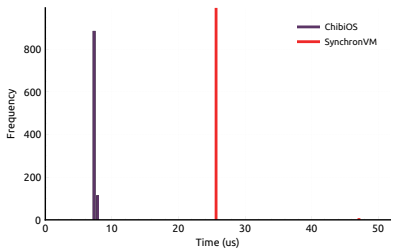
7 Benchmarks

7.1 Interpretive overhead measurements

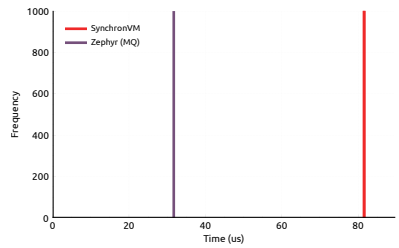
We characterise the overhead of executing programs on top of SynchronVM, compared to running them directly on either Zephyr or ChibiOS, by implementing *button-blinky* directly on top of these operating systems and measuring the response-time differences.

The button-blinky program copies the state of a button onto an LED, something that could be done very rapidly at a large CPU utilization cost by continuously polling the button state and writing it to the LED. Instead, the Zephyr and ChibiOS implementations are interrupt-based and upon receiving a button interrupt (for either button press or release), send a message to a thread that is kept blocking until such messages arrive. When the thread receives a message indicating a button down or up, it sets the LED to on or off. This approach keeps the low-level implementation in Zephyr and ChibiOS similar to SynchronVM and indicates the interpretive and other overheads in SynchronVM.

The data in charts presented here is collected using an STM32F4 microcontroller based testing system connected to either the NRF52 or the STM32F4 system under test (SUT). The testing system provides the stimuli, setting the GPIO (button) to either active or inactive and measures the time it takes for the SUT to respond on another GPIO pin (symbolising the LED). The testing system connects to a computer displaying a GUI and generates the plots used in this paper. Each plot places measured response times into buckets of similar time, and shows the number of samples falling in each bucket as a vertical bar. Each bucket is labelled with the average time of the samples it contains.



(a) Response time comparison between a C-code implementation using ChibiOS against the same program on SynchronVM (running on ChibiOS). Data obtained on the STM32F4 microcontroller. Uses 1000 samples.



(b) Response time comparison between a C-code implementation using Zephyr OS against the same program on SynchronVM (running on Zephyr). Data obtained on the NRF52 microcontroller. Uses 1000 samples.

Figure 7 Button-blinky response times comparison between C and SynchronVM

Fig. 7a shows the SynchronVM response time in comparison to the implementation of the program running on ChibiOS using its mailbox abstraction (MB). There the overhead is about 3x. Fig. 7b compares response times for SynchronVM and the Zephyr message queue based implementation (MQ), and shows an overhead of 2.6x.

7.2 Effects of Garbage Collection

This experiment measures the effects of garbage collection on response time by repeatedly running 10000 samples test for different heap-size configurations of SynchronVM. A smaller heap should lead to more frequent interactions with the garbage collector, and the effects of the garbage collector on the response time should magnify.

As a smaller heap is used, the number of outliers should increase if the outliers are due to garbage collection. The following table shows the number of outliers at each size configuration for the heap used, and there is an indication that GC is the cause of outliers.

Heap size (bytes)	256	512	1024	2048	4096	8192
Outliers NRF52 on Zephyr	3334	1429	811	491	0	81
Outliers STM32 on ChibiOs	3339	1430	810	491	0	80

Figures 8 and 9 show the response-time numbers across the heap sizes of 8192, 4096, 2048, 1024, 512 and 256 bytes. A general observable trend is that as the heap size decreases and GC increases, the response time numbers hover towards the farther end of the X-axis. This trend is most visible for the heap size of 256 bytes, which is our smallest heap size. Note that we cannot collect enough sample data for response-time if we switch off the garbage collector (as a reference value), as the program would very quickly run out of memory and terminate.

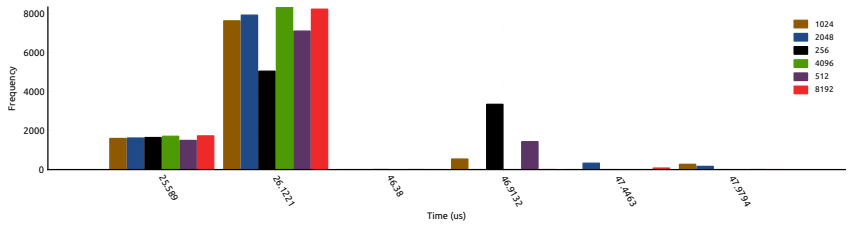


Figure 8 Response time measurements at different sizes of the heap to identify effects of garbage collection. This data is collected on the STM32F4 microcontroller running SynchronVM on top of ChibiOS. Each bucket size is approx 0.533us. Uses 10000 samples.

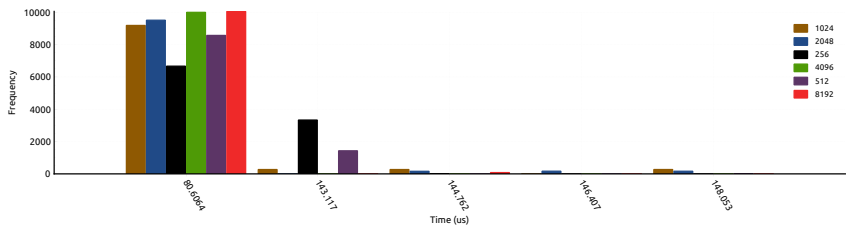


Figure 9 Response time measurements at different sizes of the heap to identify effects of garbage collection. This data is collected on the NRF52 microcontroller running SynchronVM on top of the Zephyr OS. Each bucket size is approx 1.65us. Uses 10000 samples.

7.3 Memory Footprint

SynchronVM resides on the Flash memory of a microcontroller. On Zephyr, a tiny C application occupies 17100 bytes, whereas the same SynchronVM application occupies 49356 bytes, which gives the VM's footprint as 32256 bytes. For ChibiOS, the C application takes 18548 bytes, while the SynchronVM application takes 53868 bytes. Thus, SynchronVM takes 35320 bytes in this case. Hence, we can estimate SynchronVM's rough memory footprint at 32 KB, which will grow with more drivers.

7.4 Power Usage

Fig. 10 shows the power usage of the NRF52 microcontroller running the button-blinky program for three implementations. The first is a polling version of the program in C. The second program uses a callback-based version of button-blinky [11]. The last program is Listing 4 running on SynchronVM. The measurements are made using the Ruideng UM25C ammeter. We collect momentary readings from the ammeter after the value has stabilised. Notable in Fig. 10 is the polling-based C implementation's use of 0.0175 Watts of power in a button-off state, whereas SynchronVM consumes five times less power (0.0035 Watts).

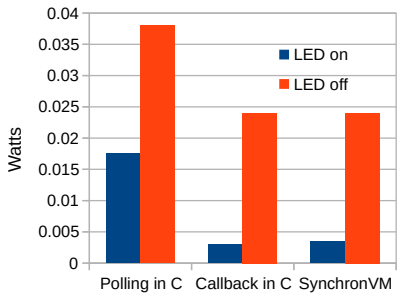


Figure 10 Power usage measured on the NRF52 microcontroller

This is comparable to the callback-based C implementation's use of 0.003 Watts. Integrating the power usage over time will likely make the difference between SynchronVM and the callback-based C version more noticeable. However, we believe that the simplicity and declarative nature of the Synchron-based code provide a fair tradeoff.

7.5 Jitter and Precision

Jitter can be defined as the deviation from true periodicity of a presumably periodic signal, often in relation to a reference clock signal. We want to evaluate how our claims of `syncT` reducing jitter pans out in practice.

Listing 11 below is written in a naive way to illustrate how jitter manifests in programs. Figure 11a shows what the oscilloscope draws, set to persistent mode drawing while sampling the signal from the Raspberry Pi outputs.

The Raspberry Pi program reads the status of a GPIO pin and then inverts its state back to that same pin. The program then goes to sleep using `usleep` for 400us. The goal frequency was 1kHz and sleeping for 400us here gave a roughly 1.05kHz signal. The more expected sleep time of 500us to generate a 1kHz signal led, instead, to a much lower frequency. So, the 400us value was found experimentally.

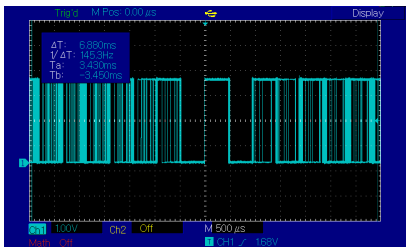
```
while (1) {
 uint32_t state = GPIO_READ(23);
 if (state) {
   GPIO_CLR(23);
 } else {
   GPIO_SET(23);
 }
 usleep(400);
 }
// main method and other setup
    elided
```

Listing 11 Raspberry Pi C code

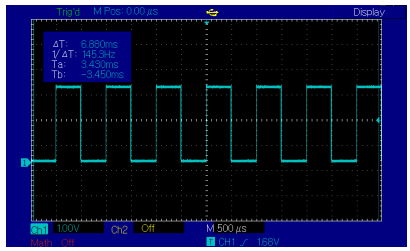
```
ledchan = channel ()

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

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

Listing 12 SynchronVM 1KHz wave code

Listing 12 shows the same 1kHz frequency generator for SynchronVM. Note that, in this case, specifying a baseline of 500us led to a 1kHz wave (Fig. 11b). In comparison, a 400us period in Listing 11 generated a roughly 1kHz wave, owing to additional delays of the system.



(a) Illustrating the amount of jitter on the square wave generated from the Raspberry Pi by setting the oscilloscope display in persistent mode.



(b) A 1kHz square wave generated using SynchronVM running on the STM32F4 with no jitter

Figure 11 A 1 kHz frequency generator on the Raspberry PI (in C) and STM32 (Synchron)

7.6 Load Test

The SynchronVM program in the previous section could produce a 1kHz-wave with no jitter. However, the only operation that the program did was produce the square wave. In this section, we want to test how much computational load can be performed by Synchron while producing the square wave. We emulate the workload using the following program.

```
21 load i n =
22  let _ = fib_tailrec n in
23  let _ = syncT 8000 0 (send
       ledchan i)
24   in load (not i)
```

```
25 loop i a b n =
26  if i == n then a
27  else loop (i+1) (b) (a+b) n
28
29 fib_tailrec n = loop 0 0 1 n
```

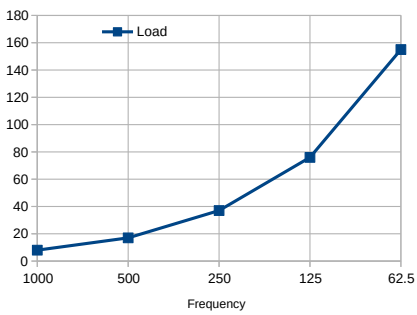


Figure 12 Load testing SynchronVM with the nth fibonacci number function

At a given frequency, it is possible to calculate only up to a certain Fibonacci number while generating the square wave at the desired frequency. For example, when generating a 62.5 Hz wave, it is only possible to calculate up to the 155th Fibonacci number. If the 156th number is calculated, the wave frequency drops below 62.5 Hz.

Fig. 12 plots the nth Fibonacci numbers that can be calculated against the square wave frequencies that get generated without jitters. Our implementation of `fib_tailrec` involves 2 addition operations, 1 equality comparison and 1 recursive call. So, calculating the 155th Fibonacci number involves $155 * 4 = 620$ major operations. The trend shows that the load capacity of SynchronVM grows linearly as the desired frequency of the square wave is halved.

7.7 Music Program Benchmarks

We now provide some benchmarks on the music program from Section 6.3. Figure 13 shows CPU usage, average time it takes to allocate data and total time spent doing allocations in a 1 minute window. The values used in the chart come from the second minute of running

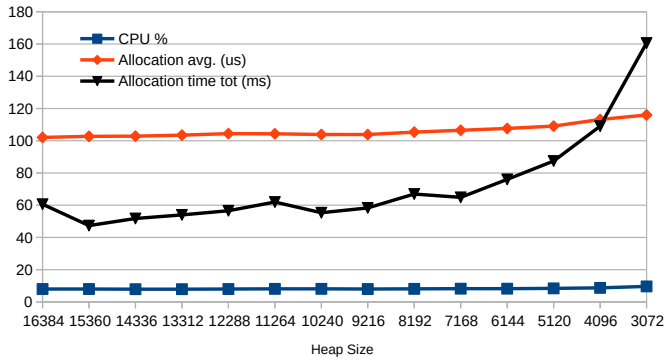


Figure 13 CPU usage and allocation trends over a 1 minute window for Listing 10

the music application. The values from the first minute of execution are discarded as those would include the startup phase of the system. The amount of heap made available to the runtime system is varied from a roomy 16384 bytes down to 3072 bytes.

The sweep phase of our garbage collector is intertwined with the allocations phase. Hence, instead of showing the GC time, the chart shows statistics related to all allocations that take a measurable amount of time using the ChibiOS 10KHz system timer. All allocations taking less than 100us are left out of the statistics (and not counted towards averaging).

The data in Fig. 13 shows that CPU usage of the music application is pretty stable at around 8 percent over the one minute window. It increases slightly for the very small heap sizes and ends up at nearly 10 percent at the smallest heap size that can house the program.

In terms of allocation, the average time of an allocation (in usecs) increases when the probability of a more expensive allocation increases, which in turn increases with small heap sizes. In the last data series, the total amount of time spent in allocations (in msecs) grows considerably as the heap size drops below 7168 bytes - an indicator of increased GC activity.

Programming Complexity. This music application ports a Real-Time Systems course exercise written in C using the TinyTimber kernel [21]. The TinyTimber-based C program (excluding the kernel) is around 600 lines of code. In comparison, our entire program (Appendix D), along with the library functions, stands at 74 lines. The most time-consuming part of the port was modifying the core logic in terms of message-passing. A major gain is that the interrupt-handling and other I/O management routines are invisible to the programmer. The user-defined timing operator, `after`, further enables the concision of the program.

7.8 Discussion

Our benchmarks, so far, show promising results for power, memory, and CPU usage. However, SynchronVM's response time is 2-3x times slower than native C code, which needs improvement. We attribute the slowness to the CAM-based execution engine, which we hope to mitigate by moving to a ZAM-based machine [20].

Our synthetic load test (Fig. 12) indicates that the VM can support around 150 operations for applications that operate around 250Hz (such as humanoid balance bots [9], autonomous vehicle platforms [36]). Our music program falls in the range of 200-500 Hz, and SynchronVM could sustain that frequency without introducing any jitter. There exist other *untimed*,

aperiodic applications with much lower frequencies where SynchronVM could be applicable. Examples include smart home applications [34], monitoring systems [14], etc.

The Synchron API chooses a synchronous message-passing model, in contrast, with actor-based systems like Erlang that support an asynchronous message-passing model with each process containing a mailbox. We believe that a synchronous message-passing policy is better suited for embedded systems for the following reasons:

1. Embedded systems are highly memory-constrained, and asynchronous send semantics assume the *unboundedness* of an actor's mailbox, which is a poor assumption in the presence of memory constraints. Once the mailbox becomes full, message-sending becomes blocking, which is already the default semantics of synchronous message-passing.
2. Acknowledgement is implicit in synchronous message-passing systems, in contrast to explicit message acknowledgement in asynchronous systems that leads to code bloat. Additionally, if a programmer forgets to remove acknowledgement messages from an actor's mailbox, it leads to memory leaks.

8 Limitations and Future Work

In this section, we propose future work to improve the Synchron API and runtime.

8.1 Synchron API limitation

Deadline miss API. Currently, the Synchron API cannot represent actions that should happen if a task were to miss its deadline. We envision adapting the negative acknowledgement API of CML to represent missed-deadline handlers for Synchron.

8.2 SynchronVM limitations

Memory management. A primary area of improvement is upgrading our stop-the-world mark and sweep garbage collector and investigating real-time garbage collectors like Schism [25]. Another relevant future work would be investigating static memory-management schemes like regions [32] and techniques combining regions with GC [15].

Interpretation overhead. A possible approach to reducing our interpretation overhead could be pre-compiling our bytecode to machine code (AOT compilation). Similarly, dynamic optimization approaches like JITing could be an area of investigation.

Priority inversions. Although TinyTimber-style dynamic priorities might reduce priority inversion occurrences, they can still occur on the SynchronVM. Advanced approaches like priority inheritance protocols [29] need to be experimented with on our scheduler.

8.3 General runtime limitations

- Power efficiency and lifetime while operating from a small battery is challenging for a byte-code interpreting virtual machine.
- Safety-critical, hard real-time systems remain out of reach with a bytecode-interpreted and garbage collected virtual machine.

9 Related Work

Among functional languages running on microcontrollers, there exists OCaml running on OMicroB [35], Scheme running on Picobit [31] and Erlang running on AtomVM [4]. Synchron

differs from these projects in the aspect that we identify certain fundamental characteristics of embedded systems and accordingly design an API and runtime to address those demands. As a result, our programming interface aligns more naturally to the requirements of an embedded systems application, in contrast with general-purpose languages like Scheme.

The Medusa [2] language and runtime is the inspiration behind our uniform framework of concurrency and I/O. Medusa, however, does not provide any timing based APIs, and their message-passing framework is based on the actor model (See Section 7.8).

In the real-time space, a safety-critical VM that can provide hard real-time guarantees on Real-Time Java programs is the FijiVM [26] implementation. A critical innovation of the project was the Schism real-time garbage collector [25], from which we hope to draw inspiration for future work on memory management.

RTMLton [30] is another example of a real-time project supporting a general-purpose language like SML. RTMLton adapts the MLton runtime [38] with ideas from FijiVM to enable handling real-time constraints in SML. CML is available as an SML library, so RTMLton provides access to the event framework of CML but lacks the uniform concurrency-I/O model and the `syncT` operator of Synchron.

The Timber language [5] is an object-oriented language that inspired the `syncT` API of Synchron. Timber was designed for hard real-time scenarios; related work on estimating heap space bounds [19] could perhaps benefit our future research.

The WebAssembly project (WASM) has spawned sub-projects like WebAssembly Micro Runtime (WAMR) [1] that allows running languages that compile to WASM on microcontrollers. Notable here is that while several general-purpose languages, like JavaScript, can execute on ARM architectures by compiling to WebAssembly, they lack the native support for the concurrent, I/O-bound, and timing-aware programs that is naturally provided by our API and its implementation. Reactive extensions of Javascript, like HipHop.js [3], are being envisioned to be used for embedded systems.

Another related line of work is embedding domain-specific languages like Ivory [10] and Copilot [24] in Haskell to generate C programs that can run on embedded devices. This approach differs from ours in the aspect that two separate languages dictate the programming model of an EDSL - the first being the DSL itself and the second being the host language (Haskell). We assess that having a single language (like in Synchron) provides a more uniform programming model to the programmer. However, code-generating EDSLs have very little runtime overheads and, when fully optimised, can produce high performance C.

10 Conclusion

In this paper, we have presented Synchron - an API and runtime for embedded systems. The API is implemented as a virtual machine called SynchronVM. We identified three essential characteristics of embedded applications, namely being concurrent, I/O–bound, and timing-aware. Correspondingly, our API is designed to address all three concerns. Our evaluations, conducted on the STM32 and NRF52 microcontrollers, show encouraging results for power, memory and CPU usage of the SynchronVM. Our response time numbers are within the range of 2-3x times that of native C programs, which we envision being improved by moving to a register-based execution engine and by using smarter memory-management strategies. We have additionally demonstrated the expressivity of our API through state machine-based examples, commonly found in embedded systems. Finally, we illustrated our timing API by expressing a soft real-time application, and we expect future theoretical investigations on the worst-case execution time and schedulability analysis on SynchronVM.

References

1 WAMR - WebAssembly Micro Runtime, 2019. URL: https://github.com/bytecodealliance/wasm-micro-runtime.

2 Thomas W. Barr and Scott Rixner. Medusa: Managing Concurrency and Communication in Embedded Systems. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 439–450. USENIX Association, 2014. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/barr.

3 Gérard Berry and Manuel Serrano. Hiphop.js: (A)Synchronous reactive web programming. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 533–545. ACM, 2020. doi:10.1145/3385412.3385984.

4 Davide Bettio. AtomVM, 2017. URL: https://github.com/bettio/AtomVM.

5 Andrew P Black, Magnus Carlsson, Mark P Jones, Richard Kieburtz, and Johan Nordlander. Timber: A programming language for real-time embedded systems. Technical report, OGI School of Science and Engineering, Oregon Health and Sciences University, Technical Report CSE 02-002. April 2002, 2002.

6 Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The Categorical Abstract Machine. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings*, volume 201 of *Lecture Notes in Computer Science*, pages 50–64. Springer, 1985. doi:10.1007/3-540-15975-4_29.

7 Robert de Simone, Jean-Pierre Talpin, and Dumitru Potop-Butucaru. The Synchronous Hypothesis and Synchronous Languages. In Richard Zurawski, editor, *Embedded Systems Handbook*. CRC Press, 2005. doi:10.1201/9781420038163.ch8.

8 Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In Andrew T. Campbell, Philippe Bonnet, and John S. Heidemann, editors, *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems, SenSys 2006, Boulder, Colorado, USA, October 31 - November 3, 2006*, pages 29–42. ACM, 2006. doi:10.1145/1182807.1182811.

9 Ahmed Elhasairi and Alexandre N. Pechev. Humanoid Robot Balance Control Using the Spherical Inverted Pendulum Mode. *Frontiers Robotics AI*, 2:21, 2015. doi:10.3389/frobt.2015.00021.

10 Trevor Elliott, Lee Pike, Simon Winwood, Patrick C. Hickey, James Bielman, Jamey Sharp, Eric L. Seidel, and John Launchbury. Guilt free ivory. In Ben Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 189–200. ACM, 2015. doi:10.1145/2804302.2804318.

11 Zephyr examples. Zephyr button blinky, 2021. URL: https://pastecode.io/s/szpf673u.

12 The Linux Foundation. Zephyr RTOS. https://www.zephyrproject.org/. Accessed 2021-11-28.

13 Damien George. Micropython, 2014. URL: https://micropython.org/.

14 R. Kingsy Grace and S. Manju. A Comprehensive Review of Wireless Sensor Networks Based Air Pollution Monitoring Systems. *Wirel. Pers. Commun.*, 108(4):2499–2515, 2019. doi:10.1007/s11277-019-06535-3.

15 Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining Region Inference and Garbage Collection. In Jens Knoop and Laurie J. Hendren, editors, *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 141–152. ACM, 2002. doi:10.1145/512529.512547.

16 Ralf Hinze. The Categorical Abstract Machine: Basics and Enhancements. Technical report, University of Bonn, 1993.

17 C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, 1978. doi:10.1145/359576.359585.

18 R John M Hughes. A semi-incremental garbage collection algorithm. *Software: Practice and Experience*, 12(11):1081–1082, 1982.

19 Martin Kero, Pawel Pietrzak, and Johan Nordlander. Live Heap Space Bounds for Real-Time Systems. In Kazunori Ueda, editor, *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings*, volume 6461 of *Lecture Notes in Computer Science*, pages 287–303. Springer, 2010. doi:10.1007/978-3-642-17164-2_20.

20 Xavier Leroy. *The ZINC experiment: an economical implementation of the ML language*. PhD thesis, INRIA, 1990.

21 Per Lindgren, Johan Eriksson, Simon Aittamaa, and Johan Nordlander. TinyTimber, Reactive Objects in C for Real-Time Embedded Systems. In *2008 Design, Automation and Test in Europe*, pages 1382–1385, 2008. doi:10.1109/DATE.2008.4484933.

22 Tommi Mikkonen and Antero Taivalsaari. Web Applications - Spaghetti Code for the 21st Century. In Walter Dosch, Roger Y. Lee, Petr Tuma, and Thierry Coupaye, editors, *Proceedings of the 6th ACIS International Conference on Software Engineering Research, Management and Applications, SERA 2008, 20-22 August 2008, Prague, Czech Republic*, pages 319–328. IEEE Computer Society, 2008. doi:10.1109/SERA.2008.16.

23 Johan Nordlander. *Programming with the TinyTimber kernel*. Luleå tekniska universitet, 2007.

24 Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: A Hard Real-Time Runtime Monitor. In Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2010. doi:10.1007/978-3-642-16612-9_26.

25 Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. Schism: fragmentation-tolerant real-time garbage collection. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 146–159. ACM, 2010. doi:10.1145/1806596.1806615.

26 Filip Pizlo, Lukasz Ziarek, and Jan Vitek. Real time Java on resource-constrained platforms with Fiji VM. In M. Teresa Higuera-Toledano and Martin Schoeberl, editors, *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES 2009, Madrid, Spain, September 23-25, 2009*, ACM International Conference Proceeding Series, pages 110–119. ACM, 2009. doi:10.1145/1620405.1620421.

27 John H. Reppy. Concurrent ML: Design, Application and Semantics. In Peter E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991-1992, McMaster University, Hamilton, Ontario, Canada*, volume 693 of *Lecture Notes in Computer Science*, pages 165–198. Springer, 1993. doi:10.1007/3-540-56883-2_10.

28 Abhiroop Sarkar, Robert Krook, Bo Joel Svensson, and Mary Sheeran. Higher-Order Concurrency for Microcontrollers. In Herbert Kuchen and Jeremy Singer, editors, *MPLR '21: 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, Münster, Germany, September 29-30, 2021*, pages 26–35. ACM, 2021. doi:10.1145/3475738.3480716.

29 Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990. doi:10.1109/12.57058.

30 Bhargav Shivkumar, Jeffrey C. Murphy, and Lukasz Ziarek. RTMLton: An SML Runtime for Real-Time Systems. In Ekaterina Komendantskaya and Yanhong Annie Liu, editors, *Practical Aspects of Declarative Languages - 22nd International Symposium, PADL 2020, New Orleans, LA, USA, January 20-21, 2020, Proceedings*, volume 12007 of *Lecture Notes in Computer Science*, pages 113–130. Springer, 2020. doi:10.1007/978-3-030-39197-3_8.

31 Vincent St-Amour and Marc Feeley. PICOBIT: A Compact Scheme System for Microcontrollers. In Marco T. Morazán and Sven-Bodo Scholz, editors, *Implementation and Application of Functional Languages - 21st International Symposium, IFL 2009, South Orange, NJ, USA, September 23-25, 2009, Revised Selected Papers*, volume 6041 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009. doi:10.1007/978-3-642-16478-1_1.

32 Mads Tofte and Jean-Pierre Talpin. Region-based Memory Management. *Inf. Comput.*, 132(2):109–176, 1997. doi:10.1006/inco.1996.2613.

33 Hideyuki Tokuda, Clifford W. Mercer, Yutaka Ishikawa, and Thomas E. Marchok. Priority Inversions in Real-Time Communication. In *Proceedings of the Real-Time Systems Symposium - 1989, Santa Monica, California, USA, December 1989*, pages 348–359. IEEE Computer Society, 1989. doi:10.1109/REAL.1989.63587.

34 Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L. Littman. Practical Trigger-Action Programming in the Smart Home. In Matt Jones, Philippe A. Palanque, Albrecht Schmidt, and Tovi Grossman, editors, *CHI Conference on Human Factors in Computing Systems, CHI'14, Toronto, ON, Canada - April 26 - May 01, 2014*, pages 803–812. ACM, 2014. doi:10.1145/2556288.2557420.

35 Steven Varoumas, Benoît Vaugon, and Emmanuel Chailloux. A Generic Virtual Machine Approach for Programming Microcontrollers: the OMicroB Project. In *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*, 2018.

36 Benjamin Vedder, Jonny Vinter, and Magnus Jonsson. A Low-Cost Model Vehicle Testbed with Accurate Positioning for Autonomous Driving. *J. Robotics*, 2018:4907536:1–4907536:10, 2018. doi:10.1155/2018/4907536.

37 Ge Wang and Perry R. Cook. ChucK: A Concurrent, On-the-fly, Audio Programming Language. In *Proceedings of the 2003 International Computer Music Conference, ICMC 2003, Singapore, September 29 - October 4, 2003*. Michigan Publishing, 2003. URL: http://hdl.handle.net/2027/spo.bbp2372.2003.055.

38 Stephen Weeks. Whole-program compilation in MLton. In Andrew Kennedy and François Pottier, editors, *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*, page 1. ACM, 2006. doi:10.1145/1159876.1159877.

39 Gordon Williams. Espruino, 2012. URL: http://www.espruino.com/.

A Appendix A - Selective Communication and Events

To enable multi-party communication, synchronous message-passing models introduce a *selective communication* operator that races between two operations and selects the one that completes first. This enables the handling of communication with multiple participants without being unnecessarily blocked by the synchronous nature of the communication.

However, Reppy identified a conflict that arises between selective communication and procedural abstraction [27]. The complication can be demonstrated via the example of a client-server communication protocol where the client follows the protocol - *first send a message along a channel, `reqCh`, and only upon the success of the send will it accept the server response along the channel, `respCh`.* Such a protocol can be expressed in synchronous models like CSP and then abstracted as a procedure like the following:

```
1 clientCall : (Channel a, Channel b) -> a -> b
2 clientCall (reqCh, respCh) a =
3   let _ = sendMsg reqCh a
4    in recvMsg respCh
```

Now imagine a scenario where the client is communicating with two servers and the first server is temporarily unavailable. In such an instance the `sendMsg` call in line no. 3 will block and as the `sendMsg` call has been abstracted away inside the procedure it is not possible to

apply the *select* operation on it. Hence, for liveness the `sendMsg` operation should not be hidden in the procedure.

However, if we expose the `sendMsg` operation, it goes against the principles of software abstraction where the internal operations of a protocol are leaked and can authorize the programmer to write unsafe operations like a sequence of two `sendMsg` calls that violates the protocol invariant (a send should always be followed by a receive). The `Event` construct of Concurrent ML resolves this issue elegantly by programming the abstraction as follows:

```
clientCallEvt : (Channel a, Channel b) -> a -> Event b
clientCallEvt (reqCh, respCh) a =
  wrap (send reqCh a) (λ _ -> sync (recv respCh))
```

The `clientCallEvt` program above represents a server as a tuple of a request channel and a response channel. The use of `wrap` in `clientCallEvt` creates an event of type `Event b` (where `b` is the type of the values sent across the `respCh`). Send events have type `Event ()` so the function `(λ _ -> sync (recv respCh))` has type `() -> b`.

The `choose` operator, of type `choose : Event a -> Event a -> Event a`, can be used now. Given two servers, `server1 = (server1ReqCh, server1RespCh)` and `server2 = (server2ReqCh, server2RespCh)`, multi-party communication can be expressed without breaking the procedural abstraction using `choose (clientCallEvt server1) (clientCallEvt server2)`.

The return type of a `choose` call will still be `Event`, allowing us to compose and *choose* among several synchronous operations like *choose* (*choose* (*choose* ev_1 ev_2) ev_3) ev_4... When `sync` is applied to such a composite event it will race among all the events, $ev_1, ev_2, ev_3, ..$, and synchronise on the operation that unblocks first.

B Appendix B - Scheduling Blinky

Fig. 14 below shows the timeline of our scheduler executing the blinky program from Listing 5. This chart involves two clocks. The actual wall-clock time, $T_{absolute}$, is represented along the X-axis while the process-local clock, T_{local}, for the process `foo` is shown inside the body of green chart representing `foo`.

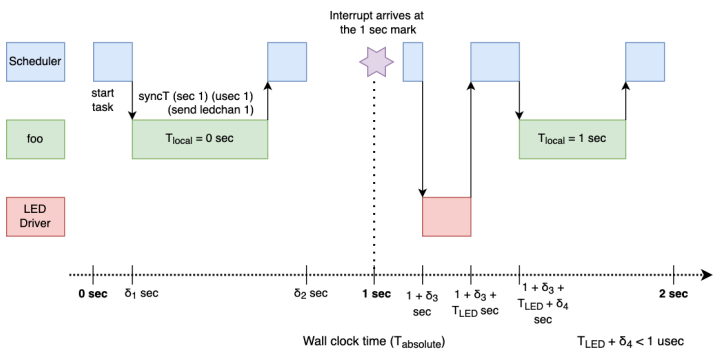


Figure 14 Scheduler timeline while executing the *blinky* program

When the program arrives at the `syncT` statement, an alarm is set for the time at which the VM should begin attempting communication with the LED driver. The alarm is set exactly at the 1-second mark, calculated from the T_{local} clock, which removes the jitter associated with other statement executions and runtime overheads.

Once the alarm interrupt arrives, communication is initiated, and the deadline counter gets activated. The LED driver takes T_{LED} seconds to execute, and the scheduler takes an additional δ_4 time units to unblock the process `foo`. So, the deadline requested to the runtime follows the relation - $\delta_4 + T_{LED} < 1$ *usec*. Finally, T_{local} is incremented by the relation $T_{local} = T_{local} + baseline$, where the baseline is 1 second in our case and the program continues.

C Appendix C - FSM Examples

C.1 Four button blinky

Listing 13 The complete Four-Button-Blinky program (Section 6.1) running on SynchronVM

```
butchan1 = channel ()
butchan2 = channel ()
butchan3 = channel ()
butchan4 = channel ()

ledchan1 = channel ()
ledchan2 = channel ()
ledchan3 = channel ()
ledchan4 = channel ()

press1 = wrap (recv butchan1) (λ x -> sync (send ledchan1 x))
press2 = wrap (recv butchan2) (λ x -> sync (send ledchan2 x))
press3 = wrap (recv butchan3) (λ x -> sync (send ledchan3 x))
press4 = wrap (recv butchan4) (λ x -> sync (send ledchan4 x))

anybutton = choose press1 (choose press2 (choose press3 press4))

program : ()
program =
 let _ = sync anybutton in
 program

main =
  let _ = spawnExternal butchan1 0 in
  let _ = spawnExternal butchan2 1 in
  let _ = spawnExternal butchan3 2 in
  let _ = spawnExternal butchan4 3 in
  let _ = spawnExternal ledchan1 4 in
  let _ = spawnExternal ledchan2 5 in
  let _ = spawnExternal ledchan3 6 in
  let _ = spawnExternal ledchan4 7 in
  program
```

C.2 Large State Machine

Listing 14 The complete complex state machine (Section 6.2) running on SynchronVM

```
butchan1 : Channel Int
butchan1 = channel ()
butchan2 : Channel Int
butchan2 = channel ()
butchan3 : Channel Int
butchan3 = channel ()
butchan4 : Channel Int
butchan4 = channel ()

ledchan1 : Channel Int
ledchan1 = channel ()
ledchan2 : Channel Int
```

```
ledchan2 = channel ()
ledchan3 : Channel Int
ledchan3 = channel ()
ledchan4 : Channel Int
ledchan4 = channel ()

not : Int -> Int
not 1 = 0
not 0 = 1

errorLed x = ledchan3

fail1ev = choose (wrap (recv butchan1) errorLed)
         (choose (wrap (recv butchan3) errorLed)
                 (wrap (recv butchan4) errorLed))

fail2ev = choose (wrap (recv butchan1) errorLed)
         (choose (wrap (recv butchan2) errorLed)
                 (wrap (recv butchan3) errorLed))

led1Handler x =
   sync (choose (wrap (recv butchan2) (\x -> ledchan1)) fail1ev)

led2Handler x =
   sync (choose (wrap (recv butchan4) (\x -> ledchan2)) fail2ev)

led : Int -> ()
led state =
  let fsm1 = wrap (recv butchan1) led1Handler in
  let fsm2 = wrap (recv butchan3) led2Handler in
  let ch = sync (choose fsm1 fsm2) in
  let _ = sync (send ch (not state)) in
  led (not state)

main =
  let _ = spawnExternal butchan1 0 in
  let _ = spawnExternal butchan2 1 in
  let _ = spawnExternal butchan3 2 in
  let _ = spawnExternal butchan4 3 in
  let _ = spawnExternal ledchan1 4 in
  let _ = spawnExternal ledchan2 5 in
  let _ = spawnExternal ledchan3 6 in
  let _ = spawnExternal ledchan4 7 in
  led 0
```

D Appendix D - The complete music programming example

We run this program on the STM32F4-discovery board that comes with a 12-bit digital-to-analog converter (DAC), which we connect to a speaker as a peripheral. We can write a value between 0 to 4095 to the DAC driver that gets translated to a voltage between 0 to 3V on the DAC output pin.

To produce a sound note we need to periodically write a sequence of 1's and 0's to the DAC driver. However, to make the produced note sound musical to the human ear, the *periodic rate* at which our process writes to the DAC driver is very important, and this is where the real-time aspect of the application comes in. The human ear recognises a note produced at a certain frequency as a musical note. Frequency is related to the periodic rate of a process by the relation:

$$Period = 1/Frequency$$

For instance, the musical note A occurs at a frequency of 440 Hz, which implies it has a time period of 2273 μseconds. From the point of view of the software, we are actually writing two values, a 1 and a 0, so we need to further divide the value by 2 to determine our rate of each individual write. If we call the rate of our writes as $Time_{Write}$, we get the relation -

$$Time_{Write} = Period/2 = 1/(2 * Frequency)$$

Now that we know how to calculate the periodicity of our write in relation to the frequency, we need to know (i) what are the musical notes that occur in the "Twinkle, Twinkle" rhyme and (ii) what are the frequencies corresponding to those notes so that we can calculate the $Time_{Write}$ value from the frequency. The musical notes of the "Twinkle, Twinkle" tune (in the key of G) are well known and is given below:

G G D D E E D C C B B A A G D D C C B B A D D C C B B A

Given the above notes, the frequency of each of these notes are also well known. In Table 2 we show our calculation of the $Time_{Write}$ value for the various musical notes.

Note	Frequency (Hz)	Period (μsec)	Time$_{Write}$ (μsec)
G	196	5102	2551
A	220	4546	2273
B	247	4050	2025
C	261	3822	1911
D	294	3406	1703
E	329	3034	1517

Table 2 Musical notes, their frequencies and time periods

Now we need to specify the time duration of each note. At the end of each note's duration period, we change the frequency of writes to the DAC driver. For instance, consider the transition from the second to the third note of the tune from G to D. If the note duration for G is 500 milliseconds then that implies our writing frequency should be 196 Hz for 500 milliseconds, and then at the 501st millisecond the frequency changes to 294 Hz (D's frequency).

When describing a musical etude, each note should be ideally mapped to its distinct duration in the program. A note duration can be a half note (1000 milliseconds) or a quarter note (500 milliseconds). The note duration of each of the 28 notes of the "Twinkle, Twinkle" tune is given below (Q implies a quarter note and H implies a half note):

Q Q Q Q Q Q H Q Q Q Q Q Q H Q Q Q Q Q Q H Q Q Q Q Q Q H

Listing 15 shows the entire program running on the SynchronVM that cyclically plays the "Twinkle, Twinkle, Little Stars" tune. The first 20 lines consists of declarations initialising a `List` data type and other standard library functions. Lines 54 - 68 consist of the principal logic

of the program. Listing 15 can be compiled and run, **unaltered**, on an STM32F4-discovery board.

Listing 15 The *Twinkle, Twinkle* tune (Section 6.3) running on SynchronVM

```
data List a where
  Nil  : List a
  Cons : a -> List a -> List a

head : List a -> a
head (Cons x xs) = x

tail : List a -> List a
tail Nil = Nil
tail (Cons x xs) = xs

not : Int -> Int
not 1 = 0
not 0 = 1

msec : Int -> Int
msec t = t * 1000

usec : Int -> Int
usec t = t

after : Int -> Event a -> a
after t ev = syncT t 0 ev

g  = usec 2551
a  = usec 2273
b  = usec 2025
c  = usec 1911
d  = usec 1703
e  = usec 1517
hn = msec 1000
qn = msec 500

twinkle : List Int
twinkle = Cons  g (Cons g (Cons d (Cons d (Cons e (Cons e (Cons d
          (Cons c (Cons c (Cons b (Cons b (Cons a (Cons a (Cons g
          (Cons d (Cons d (Cons c (Cons c (Cons b (Cons b (Cons a
          (Cons d (Cons d (Cons c (Cons c (Cons b (Cons b (Cons a Nil)
          ))))))))))))))))))))))))

durations : List Int
durations = Cons qn (Cons qn (Cons qn (Cons qn (Cons qn (Cons qn (Cons hn
            (Cons qn (Cons qn (Cons qn (Cons qn (Cons qn (Cons qn (Cons hn
            (Cons qn (Cons qn (Cons qn (Cons qn (Cons qn (Cons qn (Cons hn
            (Cons qn (Cons qn (Cons qn (Cons qn (Cons qn (Cons qn (Cons hn
             Nil))))))))))))))))))))))))))))

dacC : Channel Int
dacC = channel ()

noteC : Channel Int
noteC = channel ()

playerP : List Int -> List Int -> Int -> () -> ()
playerP melody nt n void =
  if (n == 29)
  then let _ = after (head nt) (send noteC (head twinkle)) in
       playerP (tail twinkle) durations 2 void
  else let _ = after (head nt) (send noteC (head melody)) in
       playerP (tail melody) (tail nt) (n + 1) void

tuneP : Int -> Int -> () -> ()
```

```
63 tuneP timePeriod vol void =
64   let newtp =
65       after timePeriod (choose (recv noteC)
66                                (wrap (send dacC (vol * 4095))
67                                      (λ _ -> timePeriod))) in
68   tuneP newtp (not vol) void
69
70 main =
71   let _ = spawnExternal dacC 0 in
72   let _ = spawn (tuneP (head twinkle) 1) in
73   let _ = spawn (playerP (tail twinkle) durations 2) in
74   ()
```

Our application consists of two software processes and one external hardware process. The $Time_{Write}$ values of each of the twenty eight notes are represented as the list `twinkle` on Lines 34-39 and the note durations are contained in the `durations` list (Lines 41-46). We use two channels - `dacC` to communicate with the DAC and `noteC` to communicate between the two software processes.

Owing to the different time periods of the two processes, their T_{local} clock progresses at different rates. In Figure 15 we visualise the message passing that occurs between the two software process and the hardware process when transitioning from a note C4 to a note G4.

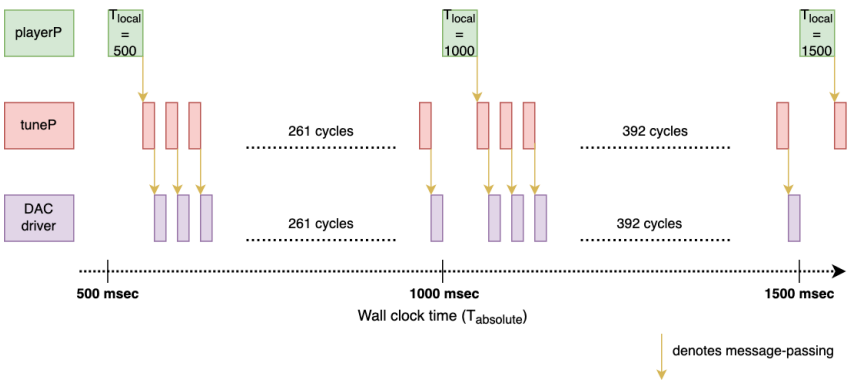


Figure 15 Moving from the note C4 to note G4

As the `playerP` process runs once every 500 milliseconds, the `tuneP` process completes $500 * 10^3/1915 = 261 cycles$ when playing the note C. For the next note, G, the $Time_{Write}$ value changes to 1432 microseconds and the corresponding write frequency changes to 392 cycles and the process cyclically carries on.

Chapter 8

The Hailstorm IoT Language

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.*

KEYWORDS

functional programming, IoT, compilers, embedded systems

ACM Reference Format:
Abhiroop Sarkar and Mary Sheeran. 2020. Hailstorm : A Statically-Typed, Purely Functional Language for IoT Applications. In *22nd International Symposium on Principles and Practice of Declarative Programming (PPDP '20), September 8–10, 2020, Bologna, Italy.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3414080.3414092

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PPDP '20, September 8–10, 2020, Bologna, Italy
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8821-4/20/09…$15.00
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 makes dataflow explicit. The purity and static typing features of the language, aside from providing a preliminary static-analysis tool, provide an essential foundation for embedding advanced language-based security techniques [54] in the future.

The programming model of Hailstorm draws inspiration from the extensive work on Functional Reactive Programming (FRP) [18]. FRP provides an interface to write *reactive* programs such as graphic animations using (1) continuous time-varying values called *Behaviours* and (2) discrete values called *Events*. The original formulation of FRP suffered from a number of shortcomings such as space-leaks [41] and a restrictive form of *stream based I/O*.

A later FRP formulation, *arrowized* FRP [46], fixed space leaks, and in more recent work Winograd-Cort et al. introduced a notion of *resource types* [66] to overcome the shortcomings of the stream based I/O model. The work on resource types is a library in Haskell, and is not suitable to run directly on resource-constrained hardware. Hailstorm uses resource types to uniquely identify each I/O

resource. It treats each resource as a signal function to track its lifetime, and prevents resource duplication through the type system. Hailstorm currently has a simple discrete time semantics, though we hope to explore extensions later.

A Hailstorm program is compiled to a dataflow graph, which is executed synchronously. The core of the language is a pure call-by-value implementation of the lambda calculus. The synchronous language of arrowized-FRP provides a minimal set of combinators to which the pure core constructs of Hailstorm can be raised. This language of arrows then enforces a purely functional way to interact with I/O, using resource types.

Hailstorm, in its current version, is a work-in-progress compiler which does not address the reliability concerns associated with node and communication failures plaguing edge devices [58]. We discuss the future extensions of the language to tackle both reliability and security concerns in Section 7. We summarize the contributions of Hailstorm as follows:

- **A statically-typed purely functional language for IoT applications.** Hailstorm provides a tailored, purely functional alternative to the current state of programming resource constrained IoT devices.
- **Resource Types based I/O.** Hailstorm builds on Winograd-Cort et al's work to provide the semantics and implementation of an alternate model of I/O for pure functional languages using *resource types* - which fits the streaming programming model of IoT applications. (Section 4.1)
- **Discrete time implementation** Hailstorm uses the combinators of arrowized FRP in a discrete time setting (Section 3).
- **An implementation of the Hailstorm language.** We implement Hailstorm as a standalone compiler, with Erlang and LLVM backends. We have run case studies on the GRiSP embedded system boards [60], to evaluate the features of the language. (Section 4). The compiler implementation and the examples presented in the paper are made publicly available[1].

2 LANGUAGE OVERVIEW

In this section we demonstrate the core features and syntax of Hailstorm using running examples. We start with a simple pure function that computes the n^{th} Fibonacci number.

```
def main : Int = fib 6

def fib : (Int -> Int)
  = fib_help 0 1

def fib_help (a : Int) (b : Int) (n : Int) : Int
  = if n == 0
    then a
    else fib_help (a + b) a (n - 1)
```

The simple program above, besides showing the ML-like syntax of Hailstorm, demonstrates some features like (1) higher-order functions (2) recursion (3) partial application (4) tail-call optimization and (5) static typing. All top-level functions in a Hailstorm program have to be annotated with the types of the arguments and the return type, which currently allows only monomorphic types. However certain built-in combinators supported by the language are polymorphic which will be discussed in the following section.

The pure core of the language only allows writing pure functions which have no form of interactions with the outside world. To introduce I/O and other side effects, we need to describe the concept of a *signal function*.

2.1 Signal Functions

A fundamental concept underlying the programming model of Hailstorm is that of a *Signal Function*. Signal Functions, derived from the work on arrowized-FRP [46], are functions that *always* accepts an input and *always* returns an output.

Signal functions are analogous to the nodes of a dataflow graph. Signal functions operate on *signals* which do not have any concrete representation in the language. A signal denotes a discrete value at a give point of time. Nilsson et al [46] use the electric circuit analogy: a signal corresponds to a wire and the current flowing through it, while signal functions correspond to circuit components. An important distinction between Hailstorm and both classic and arrowized-FRP is that signals are always treated as discrete entities in Hailstorm unlike the continuous semantics enforced by FRP.

To create larger programs Hailstorm provides a number of built-in combinators to compose signal functions. These combinators are drawn from the Arrow framework [31] which is a generalization of monads. Arrows allow structuring programs in a *point-free* style, while providing mathematical laws for composition. We start by presenting some of the core Hailstorm combinators[2] and their types for composing signal functions.

```
mapSignal# : (a -> b) -> SF a b
(>>>) : SF a b -> SF b c -> SF a c
(&&&) : SF a b -> SF a c -> SF a (b, c)
(***) : SF a b -> SF c d -> SF (a, c) (b, d)
```

Some of the built-in combinators in Hailstorm are polymorphic and the type parameters `a, b` and `c` represent the polymorphic types. `mapSignal#` is the core combinator which lifts a pure Hailstorm function to the synchronous language of arrows, as a signal function (See Fig 1).

Hailstorm then provides the rest of the built-in combinators to compose signal functions while satisfying nine arrow laws [39]. One of the advantages of having a pure functional language is that such laws can be freely used by an optimizing compiler to aggressively inline and produce optimized code. The semantics of composing signals with the arrow combinators is visually depicted in Fig 1.

[1] https://abhiroop.github.io/ppdp-2020-artifact.zip

[2] Non-symbolic built-in combinators & driver functions in Hailstorm end with #

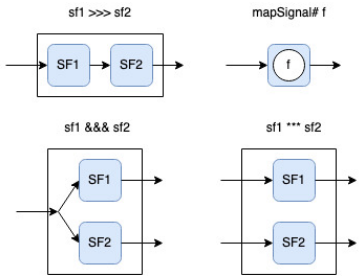


Figure 1: Arrow combinators for signal functions

Now where do signals actually come from? To answer this question - a natural extension to signal functions is using them to interact with I/O, which we discuss in the next section.

2.2 I/O

Hailstorm adopts a streaming programming model, where an effectful program is constructed by composing various signal functions in the program. The final program is embedded in a stream of input flowing in and the program transforms that into the output.

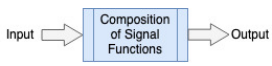


Figure 2: A Hailstorm program interacting with the real world

While this model of I/O adapts well with a pure functional language, it is reminiscent of the now abandoned stream-based I/O interface in Haskell. In Haskell's early stream-based I/O model, the type of the `main` function was `[Response] → [Request]`. The major problems with this model are :

- All forms of I/O are restricted to happen at the main function leading to non-modular programs, especially in case of applications running on IoT devices where I/O functions dominate the majority of the program.
- It is non-extensible as the concrete types of `Request` and `Response` need to be altered every time any new I/O facility has to be added [48].
- There is also no clear mapping between an individual `Request` and its `Response` [48].

Work by Winograd-Cort et al. on resource types [66] attempts to address this problem by *virtualizing* real-world devices as signal functions. What we mean here by "virtualizing" is that the scope of a program is extended so that devices like sensors and actuators are represented using signal functions. For example:

```
sensor : SF () Int
uart_rx  : SF () Byte
actuator : SF Bool ()
```

We adopt this model in Hailstorm. The type parameter `()` represents a *void* type. There are no *values* in the language which inhabit the `()` type. The `()` type always appears within a signal function. So, an example like `sensor : SF () Int` - this represents an *action* which when called, produces an integer.

One of the key aspects of designing a pure functional language is this distinction between an expression that returns a *value* and an *action*. When an action (like `sensor : SF () Int`) is evaluated, it returns a representation of that function call rather than actually *executing* the call itself. This distinction is the key to equational reasoning in a purely functional program. In the absence of such a distinction the following two expressions are no longer equivalent although they represent the same programs.

```
-- Expression 1 : accepts an input and
-- duplicates that input to return a pair
let x = getInput -- makes one I/O call
 in (x,x)

-- Expression 2 : accepts two inputs
-- returns both of them as a pair
(getInput, getInput) -- makes two I/O calls
```

After enforcing a difference between values and action in the language, we soon encounter one of the pitfalls of treating a real-world object as a virtual device - it allows a programmer to write programs with unclear semantics. For example:

```
def foo : SF () (Int, Int) = sensor &&& sensor
```

Although the above program is currently type correct, it can have two conflicting semantics - (1) either `sensor &&& sensor` implies two consecutive calls to the sensor device or (2) a single call emitting a pair. Given the type of the `sensor` function, the latter is not supported and the former is incompatible with Hailstorm's discrete, synchronous semantics.

The notion of *Resource Types* seeks to solve this problem by labeling each device with a type-level identifier, such that duplicating a device becomes impossible in the program. We change the type of `sensor` to :

```
resource S

sensor : SF S () Int
```

The *resource* keyword in Hailstorm declares a type level identifier which is used for labeling signal functions like `sensor` above. All the built-in arrow combinators introduced previously are now enriched with new *type-level* rules for composition as follows:

```
mapSignal# : (a -> b) -> SF Empty a b
(>>>) : SF r1 a b -> SF r2 b c -> SF (r1 ∪ r2) a c
(&&&) : SF r1 a b -> SF r2 a c -> SF (r1 ∪ r2) a (b, c)
(***) : SF r1 a b -> SF r2 c d -> SF (r1 ∪ r2) (a, c) (b, d)
```

In the combinators above, the type parameters r_1, r_2 represent polymorphic resource type variables, which act as labels for the effectful signal functions. The combinator `mapSignal#` lifts a pure

Hailstorm function to a signal function without any effectful operations, and as a result the resource type is `Empty`. The combinators `>>>`, `&&&` and `***` compose signal functions, and result in a disjoint-union of the two resources types. This type-level disjoint union prevents us from copying the same resource using any of these combinators. So in Hailstorm if we try this,

```
def foo : SF (S ∪ S) () (Int, Int) = sensor &&& sensor
```

we currently get the following upon compilation:

```
Type-Checking Error:
  Error in "foo":
   Cannot compose resources : S S containing same resource
   Encountered in
      sensor &&& sensor
```

The type rules associated with composing Hailstorm combinators and their operational semantics are presented formally in Section 3.2 and 3.3 respectively.

2.2.1 Example of performing I/O. We can distinguish the read and write interface of a resource using two separate resource types. For example, to repeatedly blink an LED we need two APIs - (1) to *read* its status (2) to *write* to it. The drivers for these two functions have the following types:

```
readLed# : SF R () Int
writeLed# : SF W Int ()
```

We use the integer 1 to represent light ON status and 0 for OFF. The program for blinking the LED would be:

```
def main : SF (R ∪ W) () () =
  readLed# >>> mapSignal# flip >>> writedLed#

def flip (s : Int) : Int =
  if (s == 0) then 1 else 0
```

The above program runs the function `main` infinitely. It is possible to adjust the rate at which we want to run this program, discussed later in Section 2.5. This treatment of I/O as signal functions has the limitation that each device (as well as their various APIs) has to be statically encoded as a resource type in the program.

2.3 State

Hailstorm supports stateful operations on signals using the `loop#` combinator.

```
loop# : c -> SF Empty (a, c) (b, c) -> SF Empty a b
```

The type of the `loop#` combinator is slightly different from the type provided by the `ArrowLoop` typeclass in Haskell, in that it allows initializing the state type variable `c`. The internal body of the signal function encapsulates a polymorphic state entity. This entity is repeatedly fed back as an additional input, upon completion of a whole step of signal processing by the entire dataflow graph. Fig 3 represents the combinator visually.

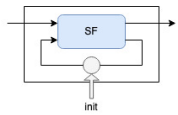


Figure 3: The stateful loop# function

The `loop#` combinator can be used to construct the `delay` function as found in synchronous languages like Lustre [26], for encoding state.

```
def delay (x : Int) : SF Empty Int Int =
  loop# x (mapSignal# swap)

def swap (a : Int, s : Int) : (Int, Int) = (s, a)
```

2.4 A sample application

We now demonstrate the use of the Hailstorm combinators in a sample application. The application that we choose is a simplified version of an earthquake detection algorithm [65] which was first used by Mainland et al. to demonstrate their domain specific language for wireless sensor networks [42]. The figure below shows the core dataflow graph of the algorithm.

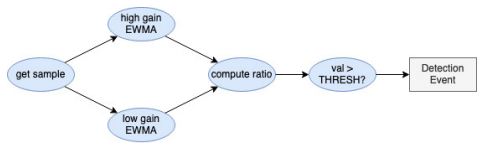


Figure 4: The earthquake detection dataflow graph

The exponentially weighted moving average (EWMA) component above is a stateful element. We assume the `getSample` input function is a wrapper around a seismometer providing readings of discrete samples. At the rightmost end, the `Detection Event` would be another stateful entity which would include some form of an *edge detector*. The entire program for the earthquake detection is given in Fig 5.

The function edge in Fig 5 is a stateful edge detector which generates an action if a boolean signal changes from `False` to `True`. To program this, we use an imaginary actuator (like an LED) in a GRiSP board, which would glow red once if the input to it is 1, signalling danger, and otherwise stay green signalling no earthquake.

2.5 Sampling rate

The combinators introduced so far execute *instantaneously* using a logical clock. In Hailstorm, one logical time step includes the following actions, in sequence -

- accepting a discrete sample of data from each of its connected input devices
- passing the discrete sample through the dataflow graph
- finally passing a discrete value to the responsible actuator

```
resource S
resource E

def main : SF (S ∪ E) () () =
 getSample >>> detect >>> edge

def detect : SF Empty Float Bool
  =   (ewma high &&& ewma low)
  >>> (mapSignal# (\(hi : Float, lo : Float) =>
                    (hi / lo) > thresh))

def ewma (α : Float) : SF Empty Float Float
  = let func = \(x : Float, x_old : Float) =>
                let x_new = (α *. x) +. (1.0 -. α) *. x_old
                 in (x_new, x_new)
    in loop# 0.0 (mapSignal# func)

-- constants
def low : Float = ...
def high : Float = ...
def thresh : Float = ...

def edge : SF E Bool () =
  loop# False (mapSignal# edgeDetector) >>>
  actuator

def edgeDetector (a : Bool, c : Bool) : (Int, Bool) =
  if (c == False && a == True)
  then (1, a)
  else (0, a)

-- getSample : SF S () Float - Erlang driver
-- actuator : SF E Int () - Erlang driver
```

Figure 5: The earthquake detection algorithm

A program returning a signal function continuously loops around, streaming in input and executing the above steps, at the speed it takes for the instructions to execute. However under most circumstances we might wish to set a slower rate for the program. The `rate#` combinator is used for that purpose,

```
rate# : Float -> SF r a b -> SF r a b
```

The first argument to `rate#` is the length of the wall clock time (in seconds) at which we wish to set the period of sampling input. This helps us establish a relation between the wall clock time and Hailstorm's logical clock. We demonstrate the utility of the *rate#* combinator using a *Stopwatch* example.

2.5.1 Stopwatch. We program a hypothetical stopwatch which accepts an input stream of `Int`s where 1 represents START, 2 represents RESET and 3 represents STOP.

```
def f (g : Float) (a : Int, c : Float) : (Float, Float) =
  let inc = c +. g in
  case a of
       1 ~> (inc, inc);
       2 ~> (0.0,0.0);
       _ ~> (c,c)

def stopwatch (g : Float) : SF Empty Int Float
  = rate# g (loop# 0.0 (mapSignal# (f g)))

def main : SF (I U O) () () =
  input >>> stopwatch 1.0 >>> output
```

In the above program, the `rate#` combinator uses the argument `g` to set the sampling rate to 1 second, which in turn fixes the granularity of the stopwatch as 1 second.

2.5.2 Limitation. In the current implementation of Hailstorm, an operation like `(rate# t2 (rate# t1 sf1))` would result in setting the final sampling rate as t_2, overwriting the value of t_1. In the programs presented here, we use a single clock, and hence a single sampling rate. Libraries like Yampa [14] provide combinators like *delay* :: $Time \rightarrow a \rightarrow SF\ a\ a$ which allow *oversampling* for dealing with multiple discrete sampling rates. As future work, we hope to adopt oversampling operators for communication among signal functions with different sampling rates.

2.6 Switches

A Hailstorm dataflow graph allows a form of *dynamic*, data-driven switching within the graph. It accomplishes this using the `switch#` combinator:

```
switch# : SF r1 a b -> (b -> SF r2 b c) -> SF (r1 ∪ r2) a c
```

The first argument to `switch#` accepts a signal function whose output data is used to switch between the various signal functions. The strict typing of Hailstorm restricts the branches of the switch to be of the same type, including the resource type. The switching dataflow is visually presented in Fig 6.

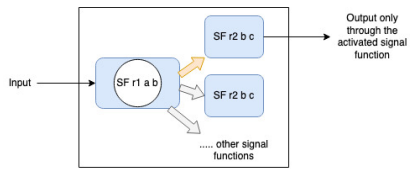


Figure 6: A switch activating only the top signal function

2.6.1 Limitation. `switch#` is a restrictive combinator with a number of known limitations:
- `switch#` constrains all of its branches to be of the same type. This is particularly restrictive when dealing with actuators where each actuator would have their own resource type. We currently deal with the notion of *choice* in the following way -

```
... >>> foo >>> (actuator1 *** actuator2)
-- instead of emitting a single value foo will emit a pair
-- of values encoded as (move actuator1, dont move actuator2)
```

The Arrow framework provides more useful combinators based on the `ArrowChoice` typeclass which are currently absent from Hailstorm.

- The switch# combinator is an experiment to describe expressions of the form:

```
switch# input_signal_function
  (\val => if <condition_1 on val>
           then SF_1
           else if <condition_2 on val>
                then SF_2
                else  ...)
```

The combinator currently supports expression only of the above form. Thus, we currently do not allow more general functions of type (b→SF b c) as the second parameter to switch#, and so avoid problems with possibly undefined runtime behaviour. However, there is no check in the compiler to enforce this restriction. As future work we hope to adopt the *guards* syntax of Haskell to represent the second parameter as a collection of boolean clauses and their corresponding actions.

3 SYNTAX, SEMANTICS AND TYPES

In the previous sections, we have given an informal treatment to the most important syntactic parts of Hailstorm, for IoT applications, and described the programming model using them. In this section, we present the core syntax, type rules and operational semantics of the main parts of Hailstorm.

3.1 Syntax

The set of types in the source language is given by the following grammar.

$$\tau ::= () \mid \texttt{Int} \mid \texttt{Float} \mid \texttt{Bool} \mid \tau_1 \rightarrow \tau_2 | \tau_1 \stackrel{r}{\Longrightarrow} \tau_2 | (\tau_1, \tau_2)$$

The type $\tau_1 \stackrel{r}{\Longrightarrow} \tau_2$ represents a signal function type from a to b with the resource type r i.e SF r a b.

The abstract syntax of the core *expressions* of Hailstorm is given by the following grammar. The meta-variable $x \in Var$ ranges over variables of the source language. Additionally we let $i \in \mathbb{Z}$ and $f \in \mathbb{R}$. We use e and e_{sf} separately to denote ordinary expressions and arrow based signal function expressions respectively.

$$\begin{aligned}
e ::= {} & x | \bar{i} | \bar{f} | \texttt{True} | \texttt{False} | i_1\ binop_i\ i_2 | f_1\ binop_f\ f_2 \\
& | e_1\ relop\ e_2 | \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 | \lambda x : \tau.\ e \mid (e_1\ e_2) \\
& | \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2 | (e_1,\ e_2) \mid \texttt{fst\#}\ e \mid \texttt{snd\#}\ e \\
e_{sf} ::= {} & \texttt{mapSignal\#}\ (\lambda x.e) | e_{sf1} >>> e_{sf2} \\
& | e_{sf1}\ \&\&\&\ e_{sf2} | e_{sf1} * * * e_{sf2} | \texttt{loop\#}\ e_1\ e_2 \\
& | \texttt{switch\#}\ e_1\ (\lambda x.e) | \texttt{read\#} | \texttt{write\#} \\
binop_i ::= {} & + \mid - \mid * \\
binop_f ::= {} & +. \mid -. \mid *. \mid / \\
relop ::= {} & > \mid < \mid >= \mid =< \mid ==
\end{aligned}$$

In the grammar above we describe two primitives for I/O called read# and write#. In practise, as Hailstorm deals with a number of I/O drivers there exists a variety of I/O primitives with varied parameters and return types. However for the purpose of presenting the operational semantics, we abstract away the complexity of the drivers and use the abstracted read# and write# to describe the semantics in Section 3.3.

3.2 Type rules

Hailstorm uses a fairly standard set of type rules except for the notion of *resource types*. The typing context of Hailstorm employs *dual contexts*, in that it maintains (1) Γ - a finite map from variables to their types and (2) Δ - a finite set which tracks all the I/O resources connected to the program.

$$\Delta; \Gamma ::= \cdot \mid \Gamma, x : \tau$$

An empty context is given by ·. Additionally $dom(\Gamma)$ provides the set of variables bound by a typing context.

In Fig 7, we show the most relevant type rules concerning signal functions and their composition. The remaining expressions follow standard set of type rules which is provided in its entirety in the extended version of this paper [55].

Looking at the rule T-Mapsignal, a signal function such as $\tau_1 \stackrel{\emptyset}{\Longrightarrow} \tau_2$ denotes the result of applying mapSignal# to a pure function. This results in an expression with an empty resource type denoted by ∅. A missing rule is the introduction of a new resource type in the resource type context Δ. The resource type context is an append-only store and a new resource is introduced using the keyword resource. It can be defined using this simple reduction semantics

$$\frac{}{\Delta; \Gamma \vdash resource\ r\ \rightsquigarrow\ \Delta \cup r; \Gamma}$$

where $\rightsquigarrow$ denotes one step of reduction which occurs at compile-time. The rules such as T-Compose, T-Fanout, T-Combine, T-Switch apply a type-level *disjoint union* to prevent resource duplication.

3.3 Big-step Operational Semantics

In this section, we provide a big-step operational semantics of our implementation of the Hailstorm language, by mapping the meaning of the terms to the lambda calculus. We begin by defining the *values* in lambda calculus that cannot be further reduced:

$$V ::= i \mid f \mid \lambda x.E$$

i and f represent integer and float constants respectively. We use n to represent variables and the last term denotes a lambda expression. The syntax that we use for defining our judgements is of the form :

$$s_1 \vdash M \Downarrow V_i, s_2$$

The variables s_1, s_2 are finite partial functions from variables n to their bound values $V_i \in V$. In case a variable n is unbound and s is called with that argument it returns ∅. The above judgement is read as *starting at state s_1 and evaluating the term M results in the irreducible value $V_i \in V$ while setting the final state to s_2.*

The first judgement essential for our semantics is this,

$$\frac{}{s \vdash V \Downarrow V, s}$$

which means that the values V cannot be reduced further.

We use a shorthand notation $\rho(n, s)$ to signify *lookup the variable n in s.* Additionally, to make the semantics more compact we

$$\frac{\Delta;\Gamma \vdash e : \tau_1 \to \tau_2}{\Delta;\Gamma \vdash mapSignal\#\ e : \tau_1 \overset{\emptyset}{\Longrightarrow} \tau_2}\ \text{(T-Mapsignal)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \overset{r_1}{\Longrightarrow} \tau_2 \qquad \Delta;\Gamma \vdash e_2 : \tau_2 \overset{r_2}{\Longrightarrow} \tau_3 \qquad r_1, r_2 \in \Delta \qquad r_1 \cap r_2 = \emptyset}{\Delta;\Gamma \vdash e_1\ >>>\ e_2 : \tau_1 \overset{r_1 \cup r_2}{\Longrightarrow} \tau_3}\ \text{(T-Compose)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \overset{r_1}{\Longrightarrow} \tau_2 \qquad \Delta;\Gamma \vdash e_2 : \tau_1 \overset{r_2}{\Longrightarrow} \tau_3 \qquad r_1, r_2 \in \Delta \qquad r_1 \cap r_2 = \emptyset}{\Delta;\Gamma \vdash e_1\ \&\&\&\ e_2 : \tau_1 \overset{r_1 \cup r_2}{\Longrightarrow} (\tau_2, \tau_3)}\ \text{(T-Fanout)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \overset{r_1}{\Longrightarrow} \tau_2 \qquad \Delta;\Gamma \vdash e_2 : \tau_3 \overset{r_2}{\Longrightarrow} \tau_4 \qquad r_1, r_2 \in \Delta \qquad r_1 \cap r_2 = \emptyset}{\Delta;\Gamma \vdash e_1\ ***\ e_2 : (\tau_1, \tau_3) \overset{r_1 \cup r_2}{\Longrightarrow} (\tau_2, \tau_4)}\ \text{(T-Combine)}$$

$$\frac{\Delta;\Gamma \vdash e : (\tau_1,\ \tau_c) \overset{\emptyset}{\Longrightarrow} (\tau_2,\ \tau_c) \qquad \Delta;\Gamma \vdash c : \tau_c}{\Delta;\Gamma \vdash loop\#\ c\ e : \tau_1 \overset{\emptyset}{\Longrightarrow} \tau_2}\ \text{(T-Loop)} \qquad \frac{\Delta;\Gamma \vdash t : Float \qquad \Delta;\Gamma \vdash e : \tau_1 \overset{r}{\Longrightarrow} \tau_2 \qquad r \in \Delta}{\Delta;\Gamma \vdash rate\#\ t\ e : \tau_1 \overset{r}{\Longrightarrow} \tau_2}\ \text{(T-Rate)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \overset{r_1}{\Longrightarrow} \tau_2 \qquad \Delta;\Gamma \vdash e_2 : \tau_2 \to \tau_2 \overset{r_2}{\Longrightarrow} \tau_3 \qquad r_1, r_2 \in \Delta \qquad r_1 \cap r_2 = \emptyset}{\Delta;\Gamma \vdash switch\#\ e_1\ e_2 : \tau_1 \overset{r_1 \cup r_2}{\Longrightarrow} \tau_3}\ \text{(T-Switch)}$$

$$\frac{r \in \Delta}{\Delta;\Gamma \vdash read\# : () \overset{r}{\Longrightarrow} \tau}\ \text{(T-Read)} \qquad \frac{r \in \Delta}{\Delta;\Gamma \vdash write\# : \tau \overset{r}{\Longrightarrow} ()}\ \text{(T-Write)}$$

Figure 7: Typing rules of signal functions in Hailstorm

$$\frac{s \vdash exp \Downarrow \lambda x.E, s}{s \vdash mapSignal\#\ exp \Downarrow \lambda x.E, s}\ \text{(eval-Mapsignal)} \qquad \frac{s_1 \vdash exp_1 \Downarrow V_1, s_2 \qquad s_2 \vdash exp_2 \Downarrow V_2, s_3}{s_1 \vdash exp_1\ >>>\ exp_2 \Downarrow \lambda x.V_2\ (V_1\ x), s_3}\ \text{(eval-Compose)}$$

$$\frac{s_1 \vdash exp_1 \Downarrow V_1, s_2 \qquad s_2 \vdash exp_2 \Downarrow V_2, s_3}{s_1 \vdash exp_1\ \&\&\&\ exp_2 \Downarrow \lambda x.\ <V_1\ x, V_2\ x>, s_3}\ \text{(eval-Fanout)}$$

$$\frac{s_1 \vdash exp_1 \Downarrow V_1, s_2 \qquad s_2 \vdash exp_2 \Downarrow V_2, s_3}{s_1 \vdash exp_1\ ***\ exp_2 \Downarrow \lambda x.\lambda y.\ <V_1\ x, V_2\ y>, s_3}\ \text{(eval-Combine)}$$

$$\frac{s_1 \vdash init \Downarrow V_i, s_2 \qquad s_2 \vdash exp \Downarrow V_1, s_3 \qquad n \notin dom(s_3)}{s_1 \vdash loop\#_n\ init\ exp \Downarrow \lambda x.fst\ (V_1\ (x,\ V_s)), s_3[n \mapsto snd\ (V_1\ (x,\ V_i))]}\ \text{(eval-Loop-Init)}$$

$$\frac{s_1 \vdash init \Downarrow V_i, s_2 \qquad s_2 \vdash exp \Downarrow V_1, s_3 \qquad n \in dom(s_3)}{s_1 \vdash loop\#_n\ init\ exp \Downarrow \lambda x.fst\ (V_1\ (x,\ \rho(n, s_3))), s_3[n \mapsto snd\ (V_1\ (x,\ \rho(n, s_3)))]}\ \text{(eval-Loop)}$$

$$\frac{s_1 \vdash e_2 \Downarrow V, s_2 \qquad s_2 \vdash e_1 \Downarrow \lambda x.E_1, s_3 \qquad s_3 \vdash E_1[x \mapsto V] \Downarrow V_f, s_4}{s_1 \vdash (e_1\ e_2) \Downarrow V_f, s_3}\ \text{(eval-App)} \qquad \frac{s_1 \vdash t \Downarrow V_t, s_2[\Psi \mapsto V_t] \qquad s_2[\Psi \mapsto V_t] \vdash exp \Downarrow V, s_3}{s_1 \vdash rate\#\ t\ exp \Downarrow V, s_3}\ \text{(eval-Rate)}$$

$$\frac{s_1 \vdash exp_1 \Downarrow V_1, s_2 \qquad s_2 \vdash exp_2 \Downarrow \lambda b.\sigma_b[\lambda c.E_1, \lambda c.E_2, ...\lambda c.E_n], s_3}{s_1 \vdash switch\#\ exp_1\ exp_2 \Downarrow \lambda a.((\lambda b.\sigma_b[\lambda c.E_1, \lambda c.E_2, ...\lambda c.E_n])(V_1\ a))\ (V_1\ a), s_3}\ \text{(eval-Switch)}$$

$$\frac{}{s \vdash read\# \Downarrow \lambda x.read, s}\ \text{(eval-Read)} \qquad \frac{}{s \vdash write\# \Downarrow \lambda x.(write\ x), s}\ \text{(eval-Write)}$$

Figure 8: Big-Step Operational Semantics of signal functions in Hailstorm

use pairs <a,b> and their first and second projections, `fst, snd`. They do not belong to V but it is possible to represent all three of them using plain lambdas and function application - shown in the extended version of this paper [55].

In Fig 8, we show the most relevant big-step operational semantics concerning signal function based combinators. The remaining expressions have standard semantics and the complete rule set is provided in the extended paper [55]. In the rule `eval-Rate`, we use Ψ to store the sampling rate. In our current implementation, when composing signal functions with different sampling rates, the state transition from s_2 to s_3 overwrites the first sampling rate.

In `eval-Loop-Init` and `eval-Loop`, the subscript n represents a variable name that is used as a key, in the global state map s, to identify each individual state.

For the rule `eval-Switch`, $\sigma_b[\lambda c.E_1, \lambda c.E_2, ...\lambda c.E_n]$ represents a conditional expression that uses the value of b i.e. $(V_1\ a)$ to *choose* one of the several branches - $\lambda c.E_i$ - and then supplies $(V_1\ a)$ again to the selected branch to actually generate a value of the stream.

Of special interest in Fig 8 are the rules `eval-Read` and `eval-Write`. We need to extend our lambda calculus based abstract machine with the operations, `read` and `write`, to allow any form of I/O. The effectful operations, `read` and `write`, are guarded by λs to prevent

any further evaluation, and as a result are treated as values. This method is essential to ensure the purity of the language - by treating effectful operations as values.

The program undergoes a *partial evaluation* transformation which evaluates the entire program to get rid of all the λs guarding the `read` operations. Given the expression $\lambda x.read$ the compiler supplies a compile time token of type `()` which removes the *lambda* and exposes the effectful function `read`. The partially evaluated program is then prepared to conduct I/O. This approach is detailed further in Section 4.1.

The big-step semantics of the language shows its evaluation strategy. However to understand the streaming, infinite nature of an effectful Hailstorm program we need an additional semantic rule. A Hailstorm `function` definition is itself an expression and a `program` is made of a list of such functions,

$$Program ::= main : [Function]$$
$$Function ::= e | e_{sf}$$

Each Hailstorm program compulsorily has a `main` function. After the entire program is *partially evaluated* (described in Section 4.1) and given that the `main` function causes a side effect (denoted by `()` below), we have the following rule:

$$\frac{s_1 \vdash main \Downarrow (), s_2}{s_1 \vdash main \Downarrow main, s_2} \text{ (eval-Main)}$$

The `eval-Main` rule demonstrates the streaming and infinite nature of a Hailstorm program when the main function is a signal function itself. After causing a side effect, it calls itself again and continues the stream of effects while evaluating the program using the semantics of Fig 8.

4 IMPLEMENTATION

Here we describe an implementation of the Hailstorm language and programming model presented in the previous sections. We implement the language as a *compiler* - the Hailstorm compiler - whose compilation architecture is described below.

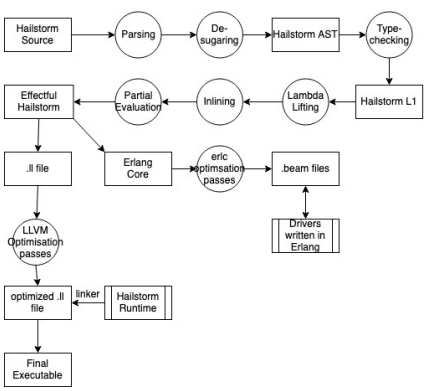


Figure 9: The Hailstorm compilation architecture

The compiler pipeline starts by parsing a Hailstorm source file and *desugaring* small syntactic conveniences provided to reduce code size (such as `case` expressions). After desugaring, the Hailstorm AST constituted of the grammar described in Section 3.1 is generated. Next, typechecking of the AST using the type rules of Fig 7 is done. After a program is typechecked, it is transformed into the Hailstorm core language called `L1`.

The `L1` language is an enriched version of the simply typed lambda calculus (STLC) with only nine constructors. Unlike STLC, `L1` supports recursion by calling the name of a global function. It is currently incapable of recursion using a `let` binding. `L1` has a simpler type system than the Hailstorm source, as it *erases* the notion of resource types - which are exclusively used during type checking.

The `L1` language being considerably simpler, forms a sufficient foundation for running correctness preserving optimization passes. `L1` attempts to get rid of all closures with free variables, as they are primarily responsible for dynamic memory allocation. Additionally, it attempts to inline expressions (primarily partial applications) to further reduce both stack and heap memory allocation. The optimization passes are described in further detail in Section 4.2.

The final optimization pass in `L1` is a *partial evaluation* pass which specializes the program to convert it into an effectful program. Before this pass, all I/O inducing functions are treated as values, by guarding them inside a λ abstraction. This pass evaluates those expressions by passing a compile-time token and turning the `L1` non-effectful program into an effectful one. This pass is discussed in further detail in Section 4.1.

Finally, the effectful Hailstorm program gets connected to one of its backends. Recursion is individually handled in each of the backends by using a global symbol table. We currently support an LLVM backend [36] and a BEAM backend [4].

4.1 I/O

The I/O handling mechanism of the Hailstorm compiler is the essential component in making the language *pure*. A pure functional language allows a programmer to equationally reason about their code. The entire program is written as an *order independent* set of equations. However, to perform I/O in a programming language, it is necessary to (1) enforce an order on the I/O interactions, as they involve *chronological* effects visible to the user and (2) interact with the real world and actually perform an effect.

To solve (1) we use the `eval-App` rule given in Fig 8. Hailstorm, being a *call-by-value* implementation of the lambda calculus, follows *ordering* in function application by always evaluating the function argument before passing it to the function. This semantics of function application allows us to introduce some form of ordering to the equations.

To solve (2) we have extended our pure lambda calculus core to involve effectful operations like `read` and `write`. Again, as observed in the semantic rules, `eval-Read` and `eval-Write` from Fig 8, the effectful operations are guarded by λ abstractions to treat them as values, rather than operations causing side-effects. This allows us to freely inline or apply any other optimization passes while preserving the correctness of the code. However, to finally perform the side effect, we need to resolve this lambda abstraction at compile

time. We do this by *partially evaluating* [33] our program. Below we describe the semantics of the partial evaluation step.

A Hailstorm program is always enforced (by the typechecker in our implementation) to contain a *main* function. We shall address the case of an effectful program i.e a *main* function whose return type is an effectful signal function such as `SF r () ()`. The `()` type in the input and output parameters reflect that this function reads from a real world source like a sensor and causes an effect such as moving an actuator. The partial application pass is only fired when the program contains the `()` type in either one of its signal function parameter.

Let us name the main function above with the return type of `SF r () ()` as $main_{sf}$. Now $main_{sf}$ is itself a function, embedded in a stream of input and transforming the input to an output, causing an effect. As it is a function we can write: $main_{sf} = \lambda t.E$.

Additionally in our implementation, after typechecking, a signal function type $\tau_1 \overset{r}{\Longrightarrow} \tau_2$ is reduced to a plain arrow type - $\tau_1 \to \tau_2$. Now, using the two aforementioned definitions, we can write the following reduction semantics:

$$\frac{main_{sf} = \lambda t.E \qquad main_{sf} : () \overset{r}{\Longrightarrow} () \qquad \tau_1 \overset{r}{\Longrightarrow} \tau_2 \rightsquigarrow \tau_1 \to \tau_2}{\dfrac{\lambda t.E : () \overset{r}{\Longrightarrow} () \rightsquigarrow \lambda t.E : () \to ()}{\lambda t.E : () \to () \rightsquigarrow E[t \mapsto \theta : ()]}\ \text{(eval-Partial)}}$$

where $\rightsquigarrow$ denotes one step of reduction and θ denotes an arbitrary compile time token of type `()`. The final step, which produces the expression $E[t \mapsto \theta : ()]$, is the partial evaluation step. We demonstrate this reduction semantics in action using an example:

```
read#  : SF STDIN () Int
write# : SF STDOUT Int ()

def main : SF (STDIN U STDOUT) () () = read# >>> write#
```

We use the `eval-Read`, `eval-Write` and `eval-Compose` rules to translate the *main* function above to

```
main = λt. (λy. write y) ((λx. read) t)
```

Given the above definition of *main*, we can apply the reduction rule `eval-Partial` and `eval-App` to get the following,

$$\frac{}{\lambda t.\ (\lambda y.\ write\ y)\ ((\lambda x.\ read)\ t)\ \rightsquigarrow_*\ (\lambda y.\ write\ y)\ (read)}\ \text{(Partial evaluation)}$$

Now the program is ready to create a *side effect* as the `read` function is no longer guarded by a λ abstraction. The `eval-App` rule guarantees that `read` is evaluated first and only then the value is fed to `write` owing to the *call-by-value* semantics of Hailstorm.

4.1.1 Limitation. The Hailstorm type system doesn't prevent a programmer from writing `write# >>> read#`. The type of such a program would be `SF (STDOUT U STDIN) Int Int`. As the types do not reflect the `()` type, the partial evaluation pass is not fired and the program simply generates an unevaluated closure - which is an expected result given the meaningless nature of the program. However, the type would allow composing it with other pure functions and producing bad behaviour if those programs are connected to meaningful I/O functions. This is simply solved by adding the following type rule:

$$\frac{r \in \Delta \qquad r \neq \emptyset \qquad \Delta;\Gamma \vdash \tau_1 = () \vee \tau_2 = ()}{exp : \tau_1 \overset{r}{\Longrightarrow} \tau_2}\ \text{(T-Unsafe)}$$

4.2 Optimizations

4.2.1 Lambda Lifting. Hailstorm, being a functional language, supports higher order functions (HOFs). HOFs frequently capture free variables which survive the scope of a function call. For example:

```
1 def addFive (nr : Int) : Int =
2   let x = 5 in
3   let addX = \(y:Int) => x + y in
4   addX nr
```

Our implementation treats HOFs as closures which are capable of capturing an environment by allocating the environment on the heap. In line no. 3 above, the value of the variable `x` is heap allocated. However, in resource constrained devices heap memory allocation, is highly restrictive and any language targeting such devices should attempt to minimize allocation.

Hailstorm applies a *lambda-lifting* [32] transformation to address this. Lambda-lifting lifts a lambda expression with free variables to a top-level function and then updates related call sites with a call to the top-level function. The free variables then act as arguments to the function. This effectively allocates them on the stack (or registers). Owing to our restricted language and the lack of polymorphism, our algorithm is less sophisticated than the original algorithm devised by Johnsson. We describe the operational semantics of our algorithm in the `lambda-lifting` rule below.

We use a slightly different notation from Section 3.3 here. P_n is used to describe the entire program with its collection of top level functions. We identify a modified program with $P_n[G(x).E]$ to mean a new program with an additional global function `G` which accepts an argument `x` and returns an expression `E`. Finally the fv function is used find the set of free variables in a Hailstorm expression.

$$\frac{P_1 \vdash exp \Downarrow \lambda x.\ E,\ P_2 \qquad fv(\lambda x.\ E) \neq \emptyset \qquad fv(\lambda x.\ E) = \{i_1, ..., i_n\} \qquad P_2 \vdash \lambda x.\ E \Downarrow E',\ P_3[G(i_1, ..., i_n, x)\ .\ E]}{P_1 \vdash exp \Downarrow E',\ P_3[G(i_1, ..., i_n, x)\ .\ E]}$$

The above rule returns a modified expression E' which consists of a function call to $G(i_1, ..., i_n, x)\ .\ E$ with the free variables $i_1, ..., i_n$ as arguments. This rule is repeatedly run on local lambda expressions until none has any free variables. It transforms the program `foo` above to :

```
def addX' (x:Int) (y:Int) = x + y

def addFive (nr : Int) : Int =
  let x = 5 in
  addX' x nr
```

4.2.2 Inlining. The inlining transformation works in tandem with the lambda lifter to reduce memory allocations. Inlining a lambda calculus based language reduces to plain β reduction. i.e $((\lambda x.\ M)\ E) \rightsquigarrow M[x \mapsto E]$. Inlining subsumes optimization passes like *copy-propagation* in a functional language. Our prototype inliner is relatively conservative in that (1) it doesn't attempt to inline recursive functions and (2) it doesn't attempt inter-function inlining.

However, it attempts to cooperate with the lambda-lifter to remove all possible sites of partial applications, which are also heap allocated, to minimize memory allocation. The program shown in the previous section, undergoes the cycle in Fig 10

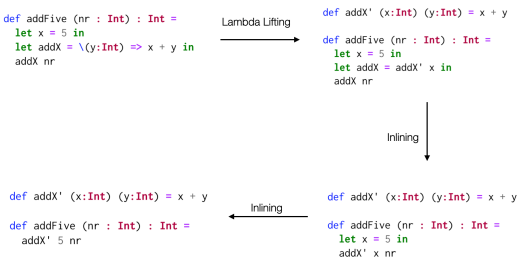


Figure 10: Lambda-lifting and inlining in action

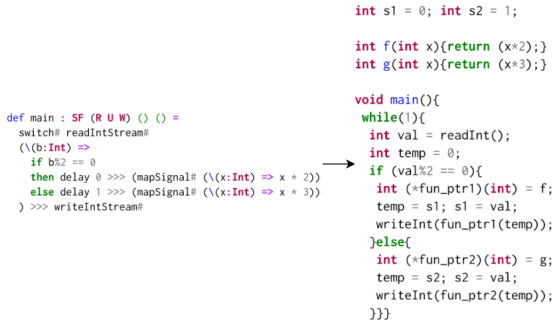


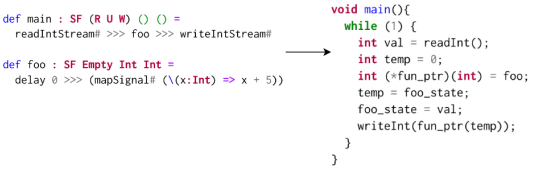
Figure 11: Code generation for switch#

The lambda lifting pass produces a partial application which is further inlined to a single function call where the arguments can be passed using registers. We show the generated LLVM code before and after the optimization passes are run in the extended version of this paper [55]. We show there the absence of any calls to `malloc` in the optimized version of the code. Our inliner doesn't employ any novel techniques but it still has to deal with engineering challenges like dealing with *name capture* [5], which it solves using techniques from the GHC inliner [34].

4.3 Code Generation

The Hailstorm compiler is designed as a linear pass through a tower of interpreters which compile away high level features, in the tradition of Reynolds [51]. Here, we show the final code generation for two of the more interesting combinators using C-like notation.

4.3.1 loop#. The `loop#` combinator models a traditional Mealy machine whose output depends on the input as well as the current state of the machine. In the following we see the code generated for the `delay` combinator described in Section 2.3 which is itself described using `loop#`.



In Erlang, the global variable of `foo_state` is modeled recursively using a global state map, which is updated on every time step. In the LLVM backend, the translation is very similar to the C-notation shown above.

4.3.2 switch#. We show an example of the `switch#` combinator when dealing with stateful branches

`switch#` type - `(b→SF b c)` - acts like a macro which is unfolded into an if-then-else expression in the `L1` core language. The core language contains primops for *getting* and *setting* state variables. The final `L1` fragment that is generated for the `(b→SF b c)` component of `switch#` (post *partial evaluation* phase) is given below:

```
... -- s1 and s2 recursively in scope
let x = readInt() in
((λ val .
   if(val % 2 == 0)
   then (λ val2 .
          let temp = s1 in
          let _ = (set s1 to val2) in
          writeInt (f temp))
   else (λ val3 .
          let temp = s2 in
          let _ = (set s2 to val3) in
          writeInt (f temp))
  ) x) x ....
```

From the above `L1` fragment, the C like imperative code shown in Fig. 11 is generated. The if-then-else expressions are translated to case expressions in Erlang and the LLVM translation is very similar to the C code. The global state map now stores two state variables which are updated depending on the value of `x`. To make the operational aspect of the combinator clearer, we show the evolution of the state variables through the various timesteps of the program:

```
INPUT:  2  3  4  5  6  7  8...
s1:     0  2  2  4  4  6  6...
s2:     1  1  3  3  5  5  7...
OUTPUT: 0  3  4  9  8  15 12...
```

4.4 Pull Semantics

In this section we discuss the semantics of data consumption in our compiler. There exist two principle approaches (1) demand-driven pull and (2) data-driven push of data. As Hailstorm's signal function semantics assume a continuous streaming flow of data triggering the dataflow graph, our compiler adopts a pull based approach which continuously *polls* the I/O drivers for data. The program blocks until more data is available from the I/O drivers.

Let us take the earthquake detection example from Section 2.4. The *getSample* input function in the dataflow graph (Fig. 4) is a

wrapper around the driver for a simulated seismometer, which when polled for data provides a reading. The rightmost edge of the graph for the `Detection Event` pulls on the dataflow graph after it completes an action and the rest of the graph in turn pulls on `getSample`, which polls the simulated seismometer.

However, in certain devices such as UART, a push based model is more prevalent, where data is asynchronously pushed to the drivers. In such cases, to avoid dropping data the wrapper function (such as `getSample`) needs to be stateful and introduce buffers that store the data. The pseudocode of the wrapper function for such a driver, written for our Erlang backend, would look like Fig 12.

```
loop(State)
  receive
    {hailstormcall, From, Datasize} ->
        (Data, NewState) = extract_Data(Datasize, State),
        From ! {ok, Data},
        loop(NewState);
    {uartdriver, Message} ->
        NewState = buffer(Message, State),
        loop(NewState);
  end.
```

Figure 12: Enforcing pull semantics on push-based data

In Fig 12 there are two separate message calls handled. The data transmission from the drivers is handled using the `uartdriver` message call, which continuously buffers the data. On the other hand, the Hailstorm program, upon finishing one cycle of computation, requests more data using the `hailstormcall` message, and proceeds with the rest of the cycle.

4.4.1 Limitation. Elliott has criticized the use of pull semantics [19] as being wasteful in terms of the re-computation required in the dataflow graph. He advocates a hybrid *push-pull* approach, which, in case of continuously changing data, adopts the pull model, but in the absence of any change in the data doesn't trigger the dataflow graph. This approach could be useful in resource constrained devices, where energy consumption is an important parameter, and we hope to experiment with this approach in future work.

4.5 The digital - analog interface

The runtime of Hailstorm has to deal with the boundary of discrete digital systems and continuous analog devices. The input drivers have to frequently discretize events that occur at some unknown point of time into a stream of discrete data. An example is the Stopwatch simulation from Section 2.5.1. The pressing of an ON button in a stopwatch translates to the stream of ones (111...) and when switched OFF the simulation treats that as a stream of zeroes. This stream transformation is handled by the wrapper functions around the input drivers.

On the contrary, for the output drivers a reverse translation of discrete to continuous is necessary. We can take the example of operating traffic lights (related demonstration in Section 5.1.2). When operating any particular signal like GREEN supplying a discrete stream of data (even at the lowest granularity) will lead to a *flickering* quality of the light. In that case the wrapper function for the light drivers employs a stateful edge detector, as discussed in Section 2.4, to supply a new signal only in case of change.

4.6 Backend specific implementation

The backend implementation includes

- *Memory management.* The compiler attempts to minimize the amount of dynamically allocated memory using lambda-lifting and inlining such that the respective garbage collectors have to work less. Future work hopes to experiment with static memory management schemes like *regions* [61].
- *Tail call optimization (TCO).* Erlang itself does TCO and LLVM supports TCO when using the `fastcc` calling convention.

5 EVALUATION

5.1 Case Studies

In this section we demonstrate examples from the synchronous language literature [30] written in Hailstorm.

5.1.1 Watchdog process. A watchdog process monitors a sequential order processing system. It raises an alarm if processing an order takes more than a threshold time. It has two input signals - (1) *order* : *SF O* () *Bool* which emits *True* when an order is placed and *False* otherwise, (2) *done* : *SF D* () *Bool* which also emits *True* only when an order is done. For output we use - *alarm* : *SF A Bool* () where an alarm is rung only when *True* is supplied. In the program below we keep a threshold time for order processing as 3 seconds.

```
def f ((order : Bool, done : Bool),
       (time : Int, openOrder : Bool)) : (Bool, (Int, Bool))
 = if (openOrder == True && time > 3)
   then (True,(time + 1, False)) -- set alarm once
   else if (done  == True)
        then (False, (0, False)) -- reset
        else if (order  == True)
             then (False, (0, True))
             else (False, (time + 1, openOrder))

def watchdog : SF (O ∪ D ∪ A) () () = (order &&& done) >>>
  (loop# (0, False) (mapSignal# f)) >>> alarm

def main : SF (O ∪ D ∪ A) () () = rate# 1.0 watchdog
```

5.1.2 A simplified traffic light system. We take the classic example of a simplified traffic light system from the Lustre literature [53]. The system consists of two traffic lights, governing a junction of two (one-way) streets. In the default case, traffic light 1 is green, traffic light 2 is red. When a car is detected at traffic light 2, the system switches traffic light 1 to red, light 2 to green, waits for 20 seconds, and then switches back to the default situation.

We use a sensor - *sensor* : *SF S* () *Bool* - which keeps returning *True* as long as it detects a car. The system, upon detecting a subsequent car, resets the wait time to another 20 seconds. We sample from the sensor every second. For the traffic lights, we use 1 to indicate green and 0 for red.

```
def lightSwitcher (sig : Bool, time : Int):((Int, Int), Int)
  = if (time > 0)
    then ((0,1), time - 1)
    else if (sig == True)
         then ((0,1), 20) -- reset
         else ((1,0), 0) -- default

def lightController : SF (S ∪ TL1 ∪ TL2) () ((), ()) =
  sensor >>> (loop# 0 (mapSignal# lightSwitcher)) >>>
  (trafficLight1 *** trafficLight2)

def main : SF (S ∪ TL1 ∪ TL2) () () =
  rate# 1.0 lightController

-- sensor : SF S () Bool
-- trafficLight1 : SF TL1 Int ()
-- trafficLight2 : SF TL1 Int ()
```

5.1.3 *A railway level crossing.*

The problem. We consider a two-track railway level crossing area that is protected by barriers, that must be closed in time on the arrival of a train, on either track. They remain closed until all trains have left the area. The barriers must be closed 30 seconds before the expected time of arrival of a train. When the area becomes free, barriers could be opened, but it's not secure to open them for less than 15 s. So the controller must be warned 45 s before a train arrives. Since the speed of trains may be very different, this speed has to be measured, by detecting the train at two points separated by a known distance. A first detector is placed 2500 m before the crossing, and a second one 100 m after this first. A third is placed after the crossing area, and records a train's leaving. We divide our solution into three programs.

The Detect Process. The passage of a train is detected by a mechanical device, producing a *True* pulse - *pulse* : *SF P* () *Bool* - only when a wheel runs on it (otherwise *False*). The `detect` process receives all these pulses, but warns the controller only once, on the first wheel. All following pulses are ignored.

```
def f (curr : Bool, old : Bool) : (Bool, Bool) =
  if (curr == True && old == False)
  then (True, curr) else (False, curr)

def detect : SF P () Bool =
  pulse >>> loop# False (mapSignal# f)
```

A Track Controller. On each track, a controller receives signals from two detectors. From `Detect1` and `Detect2`, it must compute the train's speed, and warn the barriers 45 seconds before expected time of arrival at crossing. At the maximum speed of 180 km/h, the 100 m between `Detect1` and `Detect2` are covered in 2 s. So, a clock pulse every 0.1 s would be of good accuracy.

```
def t ((d1 : Bool, d2 : Bool), time:Float) : (Float, Float)
 = case (d1, d2) of
     (True, False) ~> (0.0,0.0);
     (False, True) ~> ((24.0 *. (time +. 0.1) -. 45.0),0.0);
     _             ~> (0.0, time +. 0.1)

def timer : SF Empty (Bool, Bool) Float
 = rate# 0.1 (loop# 0.0 (mapSignal# t))

def trackController : SF (P1 ∪ P2) () Float
 = (detect1 &&& detect2) >>> timer

def detect1 : SF P1 () Bool = ...
def detect2 : SF P2 () Bool = ...
```

When a train approaches, the `trackController` calculates the time for the train to reach the barrier and sends that value. In the absence of a train it sends zeroes.

The Barriers Controller. It consists of an alarm - *alarm* : *SF* ($P_1 \cup P_2$) () *Bool* which consumes the time values from the `trackController` and returns *True* when an alarm is to be rung. The `trackController` is also sampled every 0.1 second.

```
def g (sig : Float, time : Float) : (Bool, Float) =
  if (sig > 0.0)
  then (False, sig)
  else if (time == 0.1)
       then (True, 0.0)
       else (False, time -. 0.1)

def alarm : SF (P1 ∪ P2) () Bool
  = trackController >>> rate# 0.1 (loop# 0.0 (mapSignal# g))
```

Given the alarms from the two separate tracks, the barrier controller sends an `open/close` signal represented by 0 and 1 respectively. In case a train is approaching in both of the tracks at the same speed - the barrier for only track 1 is opened.

```
def alarm1 : SF (P1 ∪ P2) () Bool = ...
def alarm2 : SF (P3 ∪ P4) () Bool = ...

def openclose (sig : (Bool, Bool)) : (Int, Int) =
  case sig of
     (True, False)  ~> (0, 1);
     (False, True)  ~> (1, 0);
     (True , True)  ~> (0, 1);
     _  ~> (0, 0) -- (False, False)

def barrierController : SF (P1 ∪ P2 ∪ P3 ∪ P4) () (Int, Int)=
    alarm1 &&& alarm2 >>> (mapSignal# openclose)
```

Finally, before sending the signal to the actuators (i.e the barriers), we need an additional system clock that keeps each barrier open for 45 seconds, and ignores other signals in the interim. The handling of the conversion of discrete signals to continuous is done by the drivers for the actuators, as discussed in Section 4.5.

```
def gate ((x : Int, y : Int),
          (t : Float, old : (Int, Int))) :
         ((Int, Int) , (Float, (Int, Int))) =
  if (old ≠ (0,0) ∧ t > 0.0)
  then (old, ((t -. 0.1), old)) -- persisting a signal
  else if (x == 1 ∨ y == 1)
       then ((x,y), (45.0, (x, y)))
       else ((x,y), (0.0, (x,y)))

def main : SF (P1 ∪ P2 ∪ P3 ∪ P4 ∪ B1 ∪ B2) () ((), ()) =
    rate# 0.1 (barrierController >>>
               (loop# (0.0, (0,0)) (mapSignal# gate)) >>>
               (barrier1 *** barrier2))
```

```
-- barrier1 : SF B1 Int ()
-- barrier2 : SF B2 Int ()
```

Note that the three instances of `rate#` all sample at the same interval of 0.1 seconds. Hailstorm currently doesn't have well defined semantics for programs with multiple clock rates.

5.2 Microbenchmarks

Here we provide memory consumption and response-time microbenchmarks for the examples presented above using the Erlang backend.

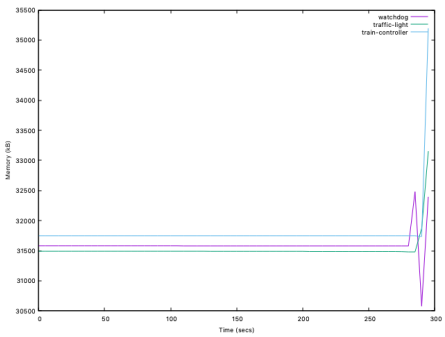


Figure 13: Memory consumption of programs

We measure the mean memory consumption for each program over five runs, each of five minutes duration. Given the I/O driven nature of the programs, the memory consumption shows little to no fluctuations. The Erlang runtime (ERTS) upon initialization sets up the garbage collector, initializes the lookup table and sets up the bytecode-interpreter which occupies 30 MB of memory on average. The actual program and its associated bookkeeping structures takes up an average of 1.5 MB of memory in the programs above. The garbage collector remains inactive throughout the program run. The memory spike visible upon termination is the garbage collector pausing the program and collecting all residual memory.

Program	Run_1(ms)	Run_2(ms)	Run_3(ms)
watchdog	7.7	8.65	11.4
traffic-light	3.81	3.04	2.12
train controller	29.72	28.05	29.8

Table 1: Response time measured in milliseconds

Table 1 shows the response time for the programs measured in milliseconds. We measure the CPU Kernel time (CPUT) - which calculates the time taken by the dataflow graph to finish one cycle of computation. We show three separate runs where the Erlang virtual machine is killed and restarted to reset the garbage collector. Each of the numbers are an average of twenty iterations of data processing. We use the `erlang:statistics` module for measuring time and in the applications I/O happens over the command line interface, which explains the overall slow behaviour (tens of milliseconds). The metrics are run on the `erts-10.6.4` runtime and virtual machine running on a Macbook-Pro with a 2.9 GHz Intel Core i9 processor. A common observation is that the computation takes less than 1% of the total wall clock time involved in the response rate, showing that the I/O reading/writing times dominate the final response rate.

An alternate benchmarking strategy which we used was to model the input from the sensors as an in-memory structure (in Erlang) and compute the total response time for processing those values using the `timer:tc` module:

Program	$\text{Run}_1(\mu s)$	$\text{Run}_2(\mu s)$	$\text{Run}_3(\mu s)$
watchdog	97.3	106.7	98.7
traffic-light	110.8	120.2	115.1
train controller	144.3	128.1	138.7

The above values are all measured in microseconds which are averaged over forty iterations each. As expected, using an in-memory structure results in graph processing times that are much lower than those in Table 1.

6 RELATED WORK

6.1 Programming Languages for IoT

There has been recent work on designing embedded DSLs (EDSLs) for IoT applications [11]. In EDSLs, I/O is handled by embedding a pure core language inside a host language's I/O model - which is an approach that Hailstorm deliberately avoids. Given the I/O dominated nature of IoT apps, we choose to focus much of our attention on designing a composable stream based I/O model, rather than only considering the pure core language, as many EDSLs do.

Other approaches like Velox VM [62] runs general purpose languages like Scheme on specialized virtual machines for IoT devices. A separate line of work has been exploring restrictive, Turing-incomplete, rule-based languages like IoTDSL [2] and CyprIOT [9].

Juniper [29] is one of the few dedicated languages for IoT but it exclusively targets Arduino boards. Emfrp [56] and its successor XFRP [59] are most closely related to the goals of Hailstorm. However, their model of I/O involves writing glue code in C/C++ and embedding the pure functional language inside it. Hailstorm has a more sophisticated I/O integration in the language.

While IoT stands for an umbrella term for a large collection of software areas, there has been research on *declarative* languages for older and *specialized* application areas like:

- *Wireless Sensor Networks (WSNs).* There exists EDSLs like Flask [42] and macroprogramming languages like Regiment [45] and Kairos [24] for WSNs.
- *Real Time Systems.* Synchronous language like Esterel [10], Lustre [26] are a restrictive set of languages designed specifically for real time systems. Further extensions of these languages like Lucid Synchrone [12], ReactiveML [44], Lucy-n [43] have attempted to makes them more expressive.

The applications demonstrated in the paper are expressible in synchronous languages, albeit using a very different interface from Hailstorm. While languages like Lustre and its extensions are *pure*

they restrict their synchronous calculus to the pure core language and handle I/O using the old stream based I/O model of Haskell [48]. Hailstorm explores the design space of *pure* functional programming with the programming model and purity encompassing the I/O parts as well.

In Lustre, a type system called the clock calculus ensures that programs can run without any implicit buffering inside the program. Strong safety properties such as determinism and absence of deadlock are ensured at compile time, and programs are compiled into statically scheduled executable code. This comes at the price of reduced flexibility compared to synchronous dataflow-like systems, particularly in the ease with which bounded buffers can be introduced and used. Mandel et al have studied n-synchronous systems in an attempt to bring greater flexibility to synchronous languages [43]. Hailstorm, in the presence of recursion, is unable to statically predict memory usage but we plan future work on type level encoding of buffer sizes to make memory usage more predictable. *Polychronous* languages like SIGNAL [8] and FRP libraries like Rhine [6] provide static guarantees on correctness of systems with multiple clocks - something that we hope to experiment with in the future.

6.2 FRP

Hailstorm draws influence from the FRP programming model. Since the original FRP paper [18], it has seen extensive research over various formulations like arrowized FRP [46], asynchronous FRP [16], higher-order FRP [35], monadic stream functions [47].

Various implementations have explored the choice between a static structure of the dataflow graph (for example Elm [15]) or dynamic structure, as in most Haskell FRP libraries [3] as well as FrTime in Racket [13]. The dynamic graph structure makes the language/library more expressive, allowing programs like sieves [25].

The higher-order FRP implementation offers almost the local maxima of tradeoffs, but at the cost of an extremely sophisticated type system, which infests into the source language, compromising its simplicity.

The `loop#` combinator in Hailstorm is similar to the μ-combinator first introduced by Sheeran [57] and to the `loopB` combinator in the Causal Commutative Arrows (CCA) paper [40]. CCA presents a number of mathematical laws on arrows and utilizes them to compile away intermediate structures and generate efficient FRP code - a promising avenue for future work in Hailstorm.

FRP has seen adoption in various application areas. Application areas related to our target areas of IoT applications include robotics [64], real-time systems [63], vehicle simulations [21] and DSLs discussed in detail in the previous section.

6.3 Functional I/O

A detailed summary of approaches to functional I/O was presented by Gordon et al. [22]. Since then monadic I/O [23] has become the standard norm for I/O in pure functional languages, with the exception of Clean's I/O system [1] based on uniqueness types.

More recently, there have been attempts at non mondaic I/O using a state passing trick in the Universe framework [20]. The latest work has been on the notion of resource types, proposed by Winograd-Cort et al. [66], which is explored as a library in Haskell. In effect, their library uses Haskell's monadic I/O model; however, they mention possible future work on designing a dedicated language for resource types. Hailstorm explores that possibility by integrating the idea of resource types natively in a language's I/O model. FRPNow! [49] provides an alternate monadic approach to integrate I/O with FRP.

7 FUTURE WORK

Hailstorm is an ongoing work to run a pure, statically-typed, functional programming language on memory constrained edge devices. As such, there are a number of open avenues for research:

- *Security.* We hope to integrate support for Information Flow Control (IFC) [28] - which uses language based privacy policies to determine safe dataflow - in Hailstorm. Given that the effectful combinators are based on the Arrow framework, we expect to build on two lines of work - (1) integrating IFC with the Arrow typeclass in Haskell [37] (2) a typeclass based technique to integrate IFC without modifying the compiler [52] but one which relies on the purity and static typing of the language.

- *Reliability.* Hailstorm currently doesn't provide any fault tolerance strategies to mitigate various node/communication failures. However, being hosted on the Erlang backend, we plan to experiment with distributed versions of the arrow combinators where the underlying runtime would use *supervision trees* to handle failures. This would be a much more invasive change as the synchronous dataflow model of the language is not practically suitable in a distributed scenario - leading to more interesting research directions on *macro-programming models* [24].

- *Memory constrained devices.* The evaluation of this paper is carried out on GRiSP boards which are relatively powerful boards. We are currently working on developing a small virtual machine which can interpret a functional bytecode instruction set and run on much more memory constrained devices like STM32 microcontrollers.

8 CONCLUSION

We have presented the design and implementation of Hailstorm, a domain specific language targeting IoT applications. Our evaluation suggests that Hailstorm can be used to declaratively program moderately complex applications in a concise and safe manner. In the future, we hope to use the purity and type system of Hailstorm to enforce language-based security constraints, as well as to increase its expressiveness to enable the description of interacting IoT devices in a distributed system.

ACKNOWLEDGMENTS

This work was funded by the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023). We would also like to thank Henrik Nilsson and Joel Svensson for their valuable feedback on improving our paper.

REFERENCES

[1] Peter Achten and Rinus Plasmeijer. 1995. The ins and outs of Clean I/O. *Journal of Functional Programming* 5, 1 (1995), 81–110.
[2] Moussa Amrani, Fabian Gilson, Abdelmounaim Debieche, and Vincent Englebert. 2017. Towards User-centric DSLs to Manage IoT Systems.. In *MODELSWARD*. 569–576.
[3] Edward Amsden. 2011. A survey of functional reactive programming. *Rochester Institute of Technology* (2011).
[4] Joe Armstrong. 1997. The development of Erlang. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*. 196–203.

[5] Hendrik Pieter Barendregt. 1985. *The lambda calculus - its syntax and semantics*. Studies in logic and the foundations of mathematics, Vol. 103. North-Holland.

[6] Manuel Bärenz and Ivan Perez. 2018. Rhine: FRP with type-level clocks. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*, Nicolas Wu (Ed.). ACM, 145–157. https://doi.org/10.1145/3242744.3242757

[7] Iulia Bastys, Musard Balliu, and Andrei Sabelfeld. 2018. If this then what? Controlling flows in IoT apps. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1102–1119.

[8] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. 1991. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of computer programming* 16, 2 (1991), 103–149.

[9] Imad Berrouyne, Mehdi Adda, Jean-Marie Mottu, Jean-Claude Royer, and Massimo Tisi. 2019. CyprIoT: framework for modelling and controlling network-based IoT applications. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. 832–841.

[10] Gérard Berry and Georges Gonthier. 1992. The Esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming* 19, 2 (1992), 87–152.

[11] Ben Calus, Bob Reynders, Dominique Devriese, Job Noorman, and Frank Piessens. 2017. FRP IoT modules as a Scala DSL. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. 15–20.

[12] Paul Caspi, Grégoire Hamon, and Marc Pouzet. 2008. Synchronous Functional Programming: The Lucid Synchrone Experiment. *Real-Time Systems: Description and Verification Techniques: Theory and Tools. Hermes* (2008).

[13] Gregory H Cooper and Shriram Krishnamurthi. 2006. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming*. Springer, 294–308.

[14] Antony Courtney, Henrik Nilsson, and John Peterson. 2003. The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*. 7–18.

[15] Evan Czaplicki. 2012. Elm: Concurrent Frp for Functional GUIs. *Senior thesis, Harvard University* (2012).

[16] Evan Czaplicki and Stephen Chong. 2013. Asynchronous functional reactive programming for GUIs. *ACM SIGPLAN Notices* 48, 6 (2013), 411–422.

[17] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. 2014. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*. 475–488.

[18] Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*. 263–273.

[19] Conal M Elliott. 2009. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*. 25–36.

[20] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2009. A functional I/O system or, fun for freshman kids. In *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, Graham Hutton and Andrew P. Tolmach (Eds.). ACM, 47–58. https://doi.org/10.1145/1596550.1596561

[21] Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. 2017. Vehicle Platooning Simulations with Functional Reactive Programming. In *Proceedings of the 1st International Workshop on Safe Control of Connected and Autonomous Vehicles*. 43–47.

[22] Andrew Donald Gordon. 1992. *Functional programming and input/output*. Ph.D. Dissertation. University of Cambridge, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.259479

[23] Andrew D Gordon and Kevin Hammond. 1995. Monadic I/O in Haskell 1.3. In *Proceedings of the haskell Workshop*. 50–69.

[24] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. 2005. Macro-programming wireless sensor networks using Kairos. In *International Conference on Distributed Computing in Sensor Systems*. Springer, 126–140.

[25] Heine Halberstam and Hans Egon Richert. 2013. *Sieve methods*. Courier Corporation.

[26] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320.

[27] Eric Haugh and Matt Bishop. 2003. Testing C Programs for Buffer Overflow Vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA*. The Internet Society. https://www.ndss-symposium.org/ndss2003/testing-c-programs-buffer-overflow-vulnerabilities/

[28] Daniel Hedin and Andrei Sabelfeld. 2012. A Perspective on Information-Flow Control. *Software safety and security* 33 (2012), 319–347.

[29] Caleb Helbling and Samuel Z Guyer. 2016. Juniper: a functional reactive programming language for the Arduino. In *Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design*. 8–16.

[30] Bernard Houssais. 2002. The synchronous programming language SIGNAL: A tutorial. *IRISA, April* (2002).

[31] John Hughes. 2000. Generalising monads to arrows. *Science of computer programming* 37, 1-3 (2000), 67–111.

[32] Thomas Johnsson. 1985. Lambda lifting: Transforming programs to recursive equations. In *Conference on Functional programming languages and computer architecture*. Springer, 190–203.

[33] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation*. Prentice Hall.

[34] Simon Peyton Jones and Simon Marlow. 2002. Secrets of the glasgow haskell compiler inliner. *Journal of Functional Programming* 12, 4-5 (2002), 393–434.

[35] Neelakantan R Krishnaswami. 2013. Higher-order functional reactive programming without spacetime leaks. *ACM SIGPLAN Notices* 48, 9 (2013), 221–232.

[36] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[37] Peng Li and Steve Zdancewic. 2010. Arrows for secure information flow. *Theoretical computer science* 411, 19 (2010), 1974–1994.

[38] Jie Lin, Wei Yu, Nan Zhang, Xinyu Yang, Hanlin Zhang, and Wei Zhao. 2017. A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications. *IEEE Internet of Things Journal* 4, 5 (2017), 1125–1142.

[39] Sam Lindley, Philip Wadler, and Jeremy Yallop. 2010. The arrow calculus. *Journal of Functional Programming* 20, 1 (2010), 51–69.

[40] Hai Liu, Eric Cheng, and Paul Hudak. 2009. Causal commutative arrows and their optimization. *ACM Sigplan Notices* 44, 9 (2009), 35–46.

[41] Hai Liu and Paul Hudak. 2007. Plugging a space leak with an arrow. *Electronic Notes in Theoretical Computer Science* 193 (2007), 29–45.

[42] Geoffrey Mainland, Greg Morrisett, and Matt Welsh. 2008. Flask: Staged functional programming for sensor networks. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*. 335–346.

[43] Louis Mandel, Florence Plateau, and Marc Pouzet. 2010. Lucy-n: a n-synchronous extension of Lustre. In *International Conference on Mathematics of Program Construction*. Springer, 288–309.

[44] Louis Mandel and Marc Pouzet. 2005. ReactiveML: a reactive extension to ML. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*. 82–93.

[45] Ryan Newton, Greg Morrisett, and Matt Welsh. 2007. The regiment macroprogramming system. In *2007 6th International Symposium on Information Processing in Sensor Networks*. IEEE, 489–498.

[46] Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. 51–64.

[47] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. 2016. Functional reactive programming, refactored. *ACM SIGPLAN Notices* 51, 12 (2016), 33–44.

[48] Simon Peyton Jones. 2001. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. *Engineering theories of software construction* 180 (2001), 47.

[49] Atze van der Ploeg and Koen Claessen. 2015. Practical principled FRP: forget the past, change the future, FRPNow!. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 302–314.

[50] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. 2015. Low latency geo-distributed data analytics. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 421–434.

[51] John C Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2*. 717–740.

[52] Alejandro Russo. 2015. Functional pearl: two can keep a secret, if one of them uses Haskell. *ACM SIGPLAN Notices* 50, 9 (2015), 280–288.

[53] Philipp Rümmer. 2014 (accessed May 13, 2020). *An Introduction to Lustre*. http://www.it.uu.se/edu/course/homepage/modbasutv/ht14/Lectures/lustre_slides_141006.pdf

[54] Andrei Sabelfeld and Andrew C Myers. 2003. Language-based information-flow security. *IEEE Journal on selected areas in communications* 21, 1 (2003), 5–19.

[55] Abhiroop Sarkar. 2020. Hailstorm - A Statically-Typed, Purely Functional Language for IoT Applications. http://abhiroop.github.io/pubs/hailstorm.pdf. [Online; accessed 19-July-2020].

[56] Kensuke Sawada and Takuo Watanabe. 2016. Emfrp: a functional reactive programming language for small-scale embedded systems. In *Companion Proceedings of the 15th International Conference on Modularity*. 36–44.

[57] Mary Sheeran. 1984. muFP, A Language for VLSI design. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*. 104–112.

[58] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE internet of things journal* 3, 5 (2016), 637–646.

[59] Kazuhiro Shibanai and Takuo Watanabe. 2018. Distributed functional reactive programming on actor-based runtime. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. 13–22.

[60] Peer Stritzinger. (accessed May 13, 2020). *GRiSP embedded system boards*. https://www.grisp.org/

[61] Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. *Information and computation* 132, 2 (1997), 109–176.
[62] Nicolas Tsiftes and Thiemo Voigt. 2018. Velox VM: A safe execution environment for resource-constrained IoT applications. *Journal of Network and Computer Applications* 118 (2018), 61–73.
[63] Zhanyong Wan, Walid Taha, and Paul Hudak. 2001. Real-time FRP. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*. 146–156.
[64] Zhanyong Wan, Walid Taha, and Paul Hudak. 2002. Event-driven FRP. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 155–172.
[65] Geoffrey Werner-Allen, Jeff Johnson, Mario Ruiz, Jonathan Lees, and Matt Welsh. 2005. Monitoring volcanic eruptions with a wireless sensor network. In *Proceeedings of the Second European Workshop on Wireless Sensor Networks, 2005.* IEEE, 108–120.
[66] Daniel Winograd-Cort, Hai Liu, and Paul Hudak. 2012. Virtualizing real-world objects in FRP. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 227–241.