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

ISBN 978-91-8103-026-6
Series Number 5484
in the series Doktorsavhandlingar vid Chalmers tekniska högskola, Ny serie
(ISSN 0346-718X) .

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.

The thesis underlying our dissertation is that *digital systems can be protected from a wide range of critical attacks by employing functional programming-based techniques, ensuring software isolation in the cloud, and facilitating high-level, declarative and memory-safe abstractions in embedded systems.* Our approach here is to employ programming language tools, specifically functional programming, which focuses on building software by composing pure functions, avoiding shared state, mutable data, and side-effects, to enhance the security of both cloud and embedded systems. For cloud systems, we use functional programming abstractions to partition security-critical software into compartmentalised structures that use modern hardware protection mechanisms such as Trusted Execution Environments (TEEs) for software isolation. For embedded systems, we present high-level functional programming constructs that raise the level of abstractions and provides safety features to resource constrained embedded system. The dissertation is organised into two parts.

Part I introduces two successive versions of a domain-specific language (DSL) designed for programming 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

This dissertation is the culmination of love, support, and guidance from numerous people spread across the world, too many to name individually. Nonetheless, I will attempt to acknowledge a few who have played the most significant roles in its development. First and foremost, I must acknowledge Mary, my supervisor extraordinaire, who has provided me with a balance of freedom and gentle nudging, helping me understand what research is and how to conduct it with integrity. I will miss our weekly meetings the most, where Mary's calming presence made it comforting to share both the progress of my work and aspects of life. Next, I must thank my co-supervisor and friend Joel, who is probably one of the best hackers I have ever met (he is too humble to admit it). Joel embodies the spirit of the old-school Unix hacker, who never forgets how fun software and hardware hacking can be – a lesson I always try to carry over into my work.

I must also thank Ale, who has been like an unofficial supervisor to me in the last two years. The way Ale consistently promotes his students (including myself) in conferences, workshops, (or even during a startup pitch!) has been a lesson for me in great leadership while supporting one's team. Ale has been one of the best collaborators I have had the opportunity to work with, and I hope we have many more chances to collaborate in the future. I would also like to thank Koen, who has been a general guide in conducting good research, hacking, and consistently encouraging me to maintain a healthy work-life balance (an area I still need to work on).

Next, I would like to thank my thesis opponent and the entire grading committee – Anil Madhavapeddy, David Basin, Elisavet Kozyri, and Jeremy Singer – for taking the time out of their busy schedules to provide me with feedback and for being among the small group of people who will likely read the complete dissertation. I want to extend special thanks to David for organising the research visit at ETH Zurich and for being a fantastic host. He has been incredibly supportive and encouraging of my work. Also, I would like to thank Jeremy and his postdoc Dejice for being a wonderful hosts during my visit to the University of Glasgow. The trip to Bletchley Park to see *where it all began* will always be memorable for me. A shoutout also goes to Alexandre Joannou for organising the short research visit at University of Cambridge, and a special thanks to Graham Hutton for encouraging me to pursue Functional Programming as an area of research.

I would next like to thank my wonderful colleagues at Chalmers, for making

it one of the most welcoming workplaces that I have worked in. The experience during the pandemic turned out to be a lesson in the importance of collaboration and working with others. I have to begin with Robert, my close friend, colleague, office-mate, and collaborator, who has made my research and work twice as fun. The great times we had not only in Göteborg, but also during our visits to Seattle and Edinburgh will always be some of the highlights of my time at Chalmers. Our shared love for Pizzeria Gibraltar will remain the stuff of legend for years to come.

Thanks to Prabhat for his friendship and his *Prabhatian proverbs* that has helped me and Robert navigate through our strenuous C-debugging sessions. I promise to restore your poster in our office soon. Also, I thank Luca for being a great friend and guide about life after PhD. Naming a few other friends and colleagues at Chalmers - Nachi, Matthi, Carlos, Henrik, Therese, Beata, Hira, Luis, David, Agustín - who have made this journey so much more enjoyable.

To my old office-mates - Sola, Oskar, Elisabet, Hira, Niek, Mathis - and my current office mates - Robert, Henrik, Luis, Katya (and Carl when he is hacking with Henrik before a deadline) - thank you for making the office fun and lively, and for encouraging those much-needed, *ever-so-often* fika breaks. I must also thank the course responsibles with whom I have had great fun working as a teaching assistant—Alex, Peter, Nir, and Patrik.

Now, coming to the last survivors of the BITSian gang – my two oldest and closest friends, Ankit and Shila – have always been a great support to share a light moment during the exhausting hacking sessions. I hope we someday live up to the name of our WhatsApp group and end up doing the Svalbard trip that we have been planning for the last 5 years or so.

To my other close friends in Bangalore (and spread around now) – Debaleena, Tushar, Siddhant, Ashish (Gogna), Tamojay, Athma, Varun, Ashish (Negi) and Pooja, thanks for the wonderful times and memories at Bangalore. I hope to be back there sooner than you would imagine. Especially to Ashish and Varun, if we did not decide to skip the odd stand up meeting at work and attend the talk by John (Hughes) at Functional Conf 2016, I would probably have been at a very different place today.

Finally, I would like to thank my entire extended family for their constant support and encouragement. In particular, I would like to thank my mother, who is my first and most important teacher; my father, for setting an example of a strong work ethic; and my sister, for her constant support in all of my harebrained ideas and being a great friend.

I conclude with gratitude to the Swedish Foundation for Strategic Research (SSF) for funding my research, conference attendance, and research visits throughout these five years, providing me with the incredible opportunity to live and work in the beautiful city of Göteborg.

Contents

Abstract **iii**

List of Publications **v**

Acknowledgement **vii**

1 Introduction **3**
1.1 The Thesis 6
1.2 Security in the Cloud 7
1.3 Safety in Embedded Systems 9
1.4 Papers I and II: HasTEE and HasTEE$^+$ 12
1.5 Paper III: Synchron 16
1.6 Paper IV: Hailstorm 19
1.7 Discussion 20

2 Reflections **23**
2.1 The Functional Programming Toolkit 24
2.2 Practicality 25
2.3 Reasoning with Functional Programming 25
2.4 Applicability beyond Functional Languages 26

3 Future Work **29**
3.1 Property-based Testing for TEEs 29
3.2 Beyond Binary Attestation 29
3.3 Side-channel Attacks 30
3.4 SynchronVM on CHERIoT 30
3.5 Conclusion 31

4 Statement of Contributions **33**

Bibliography **37**

5 HasTEE: Programming TEEs with Haskell **56**

6 HasTEE$^+$: Confidential Computing and Analytics with Haskell **76**

7 Synchron - An API and Runtime for Embedded Systems **100**

8 The Hailstorm IoT Language **140**

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 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 hypervisor, operating system and its associated drivers*, compromising critical parts of the system. Section 1.2 explains the attacker model and security of the cloud in further detail.

Embedded Systems: These systems employ hardware deployed in large volumes, characterized by low power and resource constraints, often in the form of microcontrollers. The resource constraints dictate the use of highly memory-efficient and power-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. Section 1.3 explains the consequences of unsafe programming languages in embedded systems in further detail.

The Threat Model

Based on the discussed attacks, we can now craft a threat model that we want to address in this dissertation. We can divide it into two parts:

- **Threat Model 1.** For cloud systems, we assume a powerful attacker that has administrative access to the operating system, hypervisor and other related system software hosted on a malicious cloud service. The attacker could be further classified as a *passive attacker* who can learn from observing the public channels of interaction of the trusted program and the *active attacker* who actively attempts to compromise the trusted program through various malicious means discussed above.

[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.

- **Threat Model 2.** In embedded systems, the *active attacker* often exploits the memory-unsafety of C/C++ to supply malicious inputs or misuse the calling convention, causing buffer overflow or use-after-free bugs. These vulnerabilities enables *remote code injection*, *privilege escalation*, *denial-of-service attacks*, and similar exploits.

Madhavapeddy et al [13] provides a systematic classification of the various attack vectors exploitable within both of our threat models. In this dissertation, our aim is to propose a solution based on programming language techniques that addresses both of the threat models stated above. 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 convenient properties have historically cultivated a culture of equational reasoning [14], even in the presence of a variety of computational effects [15], giving rise to a category-theory-inspired *algebra of programming* [16] in functional languages. Furthermore, theoretical results [17] have justified the 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 [18]. Haskell's type system is further extended by tools such as *LiquidHaskell* [19], enabling the encoding of invariants that generalise Hoare Logic [20]. Additionally, libraries like *QuickCheck* [21] 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 considering *Threat Model 1* carefully, 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 (*Threat Model 2*) 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 arising from *Threat Model 1* by proposing a domain-specific language that combines specialised *hardware security extensions* with *language-based security* [22] to provide more concrete security guarantees to the programmer. To tackle the final two challenges arising from Threat Model 2, we introduce a set of *new functional programming abstractions* and a specialised *functional language runtime*, designed specifically for embedded systems.

1.1 The Thesis

Having discussed the challenges arising from the threat models attempting to compromise the safety and security of both cloud and embedded systems, 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 isolate sensitive code and data in the presence of powerful adversarial attacks in a cloud system. Extending these functional programming abstractions with structured concurrency and temporal programming primitives further provides high-level, declarative and memory-safe abstractions for embedded systems, thereby establishing a foundation for building safer and more secure digital systems overall.*

The rest of this dissertation is aimed at supporting our thesis statement. In the following sections, we will first provide the necessary background on the safety and security of cloud and embedded systems, essential for presenting our contributions. Followed by that, we provide the summary of four of our publications that support the hypothesis of functional programming as a foundation for building safer and secure system. Figure 1.1 shows the outline of this dissertation and how they tackle parts of our general threat model.

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 [23], [24].

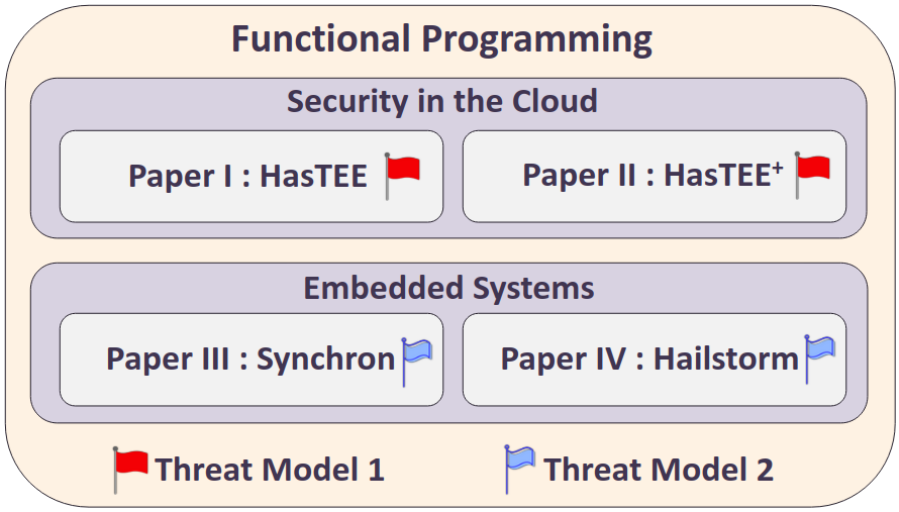

Figure 1.1: Outline of the dissertation

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* [25] – 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 [25]. The expected behaviour is often closely tied to the correctness criterion of the system.
Secure. The Bell-LaPadula model [26] 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.

1.2 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 [27], Microsoft [28], and Google [29]. 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 [30].

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 [31]–[36].

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 [37]. 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) [38], ARM TrustZone [39], AMD Secure Encrypted Virtualization (SEV) [40], and Intel Trust Domain Extensions (TDX) [41]. Fig. 1.2 illustrates a comparison of the attack surface for an application running on the cloud without and with a TEE unit engaged.

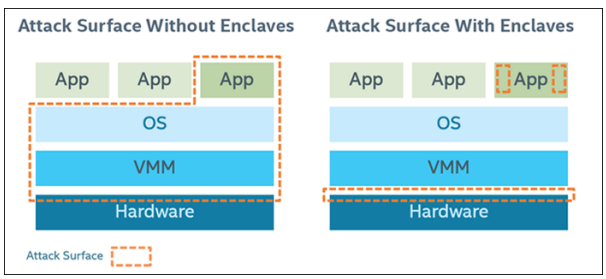

Figure 1.2: The software attack surface (trusted code base) without and with a TEE unit; Image source: Intel.

In Figure 1.2 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 both the *code integrity* and *data 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 [42].

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 [43] 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 [22].

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 [22] for protecting the confidentiality and integrity of security-critical software.*

Before presenting our approach to capitalise on this opportunity, we will now 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.

1.3 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 [44] 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) [45], 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 1.3 show the results of a 2019 survey of the embedded systems market conducted by EE Times [46]. 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.

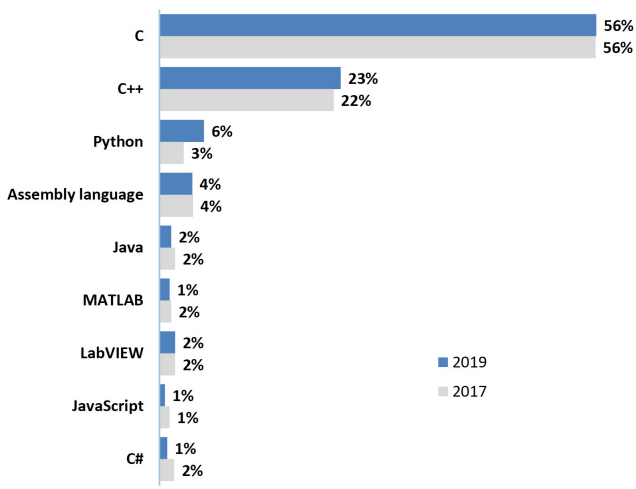

Figure 1.3: EETimes 2019 Embedded Systems Markets Study [46]

Figure 1.3 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 [47]. 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 [48], 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 [49]. 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 [44] 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 [50], 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* [51]. Also, in terms of real-time behaviors, C is deficient and often resorts to vendor-specific, bespoke real-time extensions to the original language [52].

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.

For certain application areas, such as soft real-time applications, if programmers are willing to adopt automatic memory management schemes, a trade-off emerges: memory-safe programs that occasionally miss application deadlines versus memory-unsafe programs susceptible to the wide class of memory-unsafety-based attacks [49] but with tighter timing performance. In this dissertation, we opt for the former, which gives rise to an opportunity.

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.*

Recapitulating, in this dissertation, we are considering two classes of systems – cloud systems and embedded systems. After discussing the challenges to the safety and security of both classes of systems, we have now identified two opportunities for contributions aimed at supporting our thesis of using functional programming abstractions as a foundation for safe and secure systems. In the remainder of this chapter, we will provide a high-level summary of our four papers and highlight their main contributions that support our thesis statement. 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. Note that the contributions in Papers III and IV have also been discussed in our licentiate thesis [53].

1.4 Papers I and II: HasTEE and HasTEE$^+$

Paper I

Paper I presents HasTEE [54], 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* [55]. 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 [56] 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 [57] 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 [54] 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 [58] 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*[59], 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 [60], [61].

Information Flow Control. The HasTEE API also incorporates a pragmatic implementation of information flow control [22], where all computations carried out in the *enclave* are considered highly confidential. Enforcing generalised non-interference [62] 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 [63] – involves the use of TEEs and homomorphic encryption [64] 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$^+$ [65], a DSL that builds on top of HasTEE while adopting a different threat model. Figure 1.5 contrasts the two threat models.

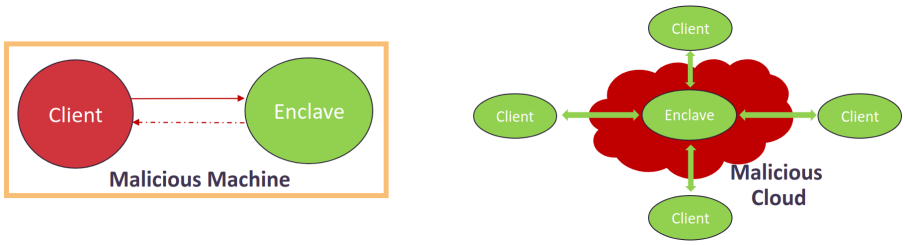

Figure 1.5: The HasTEE threat model on the left and the HasTEE$^+$ threat model on the right (not limited to four clients)

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 [66].

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* [67]. 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 [41]. 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 [68] 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 [69]. 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 [62].

The labeling model adopted in HasTEE$^+$ is Disjunction Category labels [70], which itself is based on the well-known Myers-Liskov labeling model [71]. Our implementation additionally provides the client application with labeling and unlabeling primitives, as they are trusted. Declassification [72] is permitted using *privileges*, which resemble the concept of *capabilities* [73]. 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* [74]. 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 [75]. 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 [76], [77]. 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* [78], 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 [79], 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 [80] (of Caml [81]), 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.

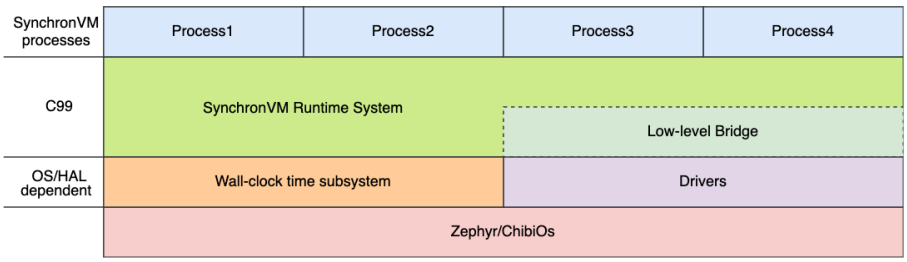

Figure 1.6: Architecture of Synchron

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 [81]. We emphasise that we use the type signature for the ease of exposition; however, the underlying virtual machine is untyped.

```
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

Concurrency. The Synchron API is built on Concurrent ML (CML) [82], a *synchronous message-passing–based concurrency model*. CML's key distinction from predecessors like Hoare's *communicating sequential processes* [83] 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 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 [84]. However, Chapter 7 provides a comprehensive explanation of the complete API.

Evaluations. Our evaluations were carried out on the NRF52840DK [85] and the STM32F4 Discovery [86] 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 [87] (that underlies OCaml [88]).

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 [89]. 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 [90].

1.6 Paper IV: Hailstorm

Paper IV introduces Hailstorm, a *functional reactive* programming language designed for embedded systems [91]. 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) [92]; 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 [93]:

> *...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 [94] 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 [95]. 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 $SF\ r\ a\ b$, 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
```

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

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 [79], 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.

In Fig. 1.9, 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)

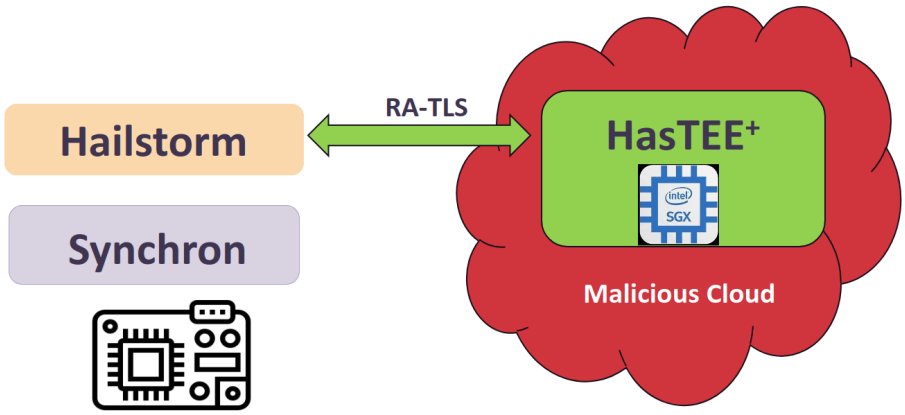

Figure 1.9: A secure digital system using HasTEE$^+$, Synchron and Hailstorm

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 [54] and HasTEE$^+$ [65] 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 [79] and Hailstorm [91] need to implement *structured concurrency* primitives. *Structured concurrency* [96], 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 [97]

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.

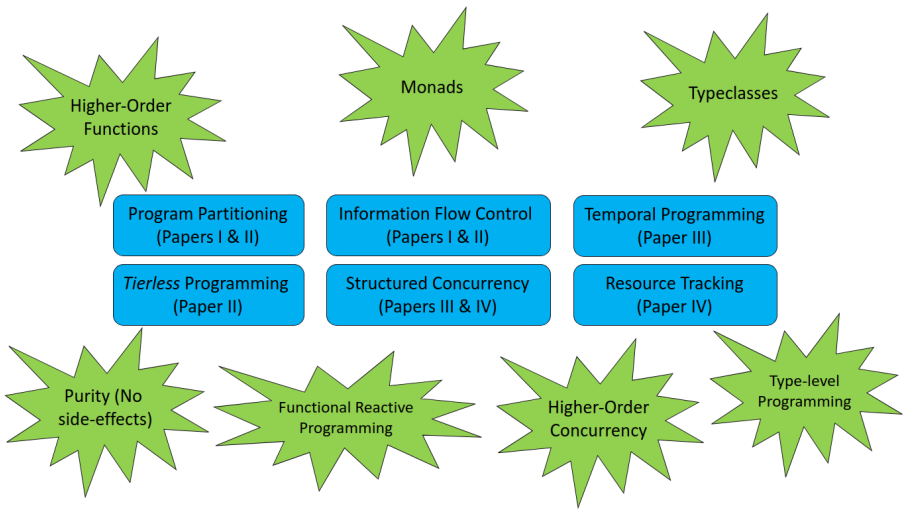

Figure 2.1: Some essential tools (green) from the *Functional Programming Toolkit* employed in this thesis and their corresponding applications (blue).

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 [70] for IFC in Chapter 6 .

Purity. Purity is an important concept in IFC, where a *pure* function can be considered *confined* by default [98]. Consequently, HasTEE$^+$ relies on the purity of Haskell functions to enforce non-interference.

Higher-Order Concurrency. Originally proposed by John Reppy [82], 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 [95] 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 [99]. 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 [100] for TEE programming or Jif [101] and Flow Caml [102] 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 [79], 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 [15]. As the HasTEE$^+$ library implements enclave computation within a monad, the well-known *left-identity*, *right-identity*, and *associativity* monad laws [103] 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) [69].

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 [83], which has a substantial body of work dedicated to developing a mathematical theory [104].

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 [105].

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 [106] and complete [107] 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 [62]. The non-interference property of security monads and their corresponding Haskell libraries has been mechanically verified [108].

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 [109], 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 [110].

An alternative to formal verification could involve using QuickCheck [21] 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 [111]. 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 [112] 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 sophisticated usage of Rust traits, many of the core functionalities in our contributions could be implemented. There also exists a Rust SDK [113] 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 [114] and TEEzz [115] 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 [116]) 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 [117], 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. [118] 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* [119].

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 [120] has been around since the introduction of the Trusted Platform Module [121] 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 [122].

3.3 Side-channel Attacks

There is a growing body of work on side-channel attacks against Intel SGX enclaves [123], where the speculative execution capabilities of modern x86 machines are exploited to extract secrets through timing [77], voltage [124], and other side channels. The long-term solution to these attacks would involve migrating security-critical software to computer architectures with specialised extensions [125] or to newer processor designs [126].

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 [127] 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 [128] introduced secrecy type systems and compiler transformations to enforce the aforementioned strategies. These approaches could be implemented and experimented with using GHC compiler plugins [129]. 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 [49].

A solution to this would be porting Synchron to a new architectural extension to RISC-V called CHERIoT [130]. Underneath, CHERIoT employs a capability-based system that uses a collection of techniques involving fat pointers 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] A. Madhavapeddy, T. Leonard, M. Skjegstad *et al.*, 'Jitsu: Just-in-time summoning of unikernels,' in *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, USENIX Association, 2015, pp. 559–573. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/madhavapeddy (cit. on p. 5).

[14] 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. 5).

[15] 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. 5, 25).

[16] 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. 5).

[17] 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. 5).

[18] 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).

[19] 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).

[20] 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).

[21] 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, 26).

[22] 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. 6, 9, 13).

[23] 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. 6).

[24] 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. 6).

[25] A. Martin, 'The ten-page introduction to Trusted Computing,' 2008 (cit. on p. 7).

[26] D. E. Bell, L. J. LaPadula *et al.*, *Secure computer systems: Mathematical foundations*. National Technical Information Service, 1989 (cit. on p. 7).

[27] Amazon. 'Amazon Web Services.' (2006), [Online]. Available: https://aws.amazon.com/. (accessed: 18.01.2024) (cit. on p. 7).

[28] Microsoft. 'Microsoft Azure.' (2008), [Online]. Available: https://azure.microsoft.com/en-us/. (accessed: 18.01.2024) (cit. on p. 7).

[29] Google. 'Google Cloud Platform.' (2008), [Online]. Available: https://cloud.google.com. (accessed: 18.01.2024) (cit. on p. 7).

[30] 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. 7).

[31] 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. 8).

[32] 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. 8).

[33] NIST. 'The VENOM Hyperjacking Vulnerability.' (2015), [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2015-3456. (accessed: 18.01.2024) (cit. on p. 8).

[34] 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. 8).

[35] 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. 8).

[36] 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. 8).

[37] 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. 8).

[38] 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. 8).

[39] ARM. 'ARM TrustZone.' (2004), [Online]. Available: https://www.arm.com/technologies/trustzone-for-cortex-a. (accessed: 18.01.2024) (cit. on p. 8).

[40] AMD. 'AMD Secure Encrypted Virtualization.' (2017), [Online]. Available: https://www.amd.com/en/developer/sev.html. (accessed: 18.01.2024) (cit. on p. 8).

[41] 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. 8, 15).

[42] 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. 9).

[43] 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. 9).

[44] 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. 9, 11).

[45] 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. 10).

[46] 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 p. 10).

[47] ARM. 'ARM GNU Toolchain.' (1992), [Online]. Available: https://developer.arm.com/Tools%20and%20Software/GNU%20Toolchain. (accessed: 18.01.2024) (cit. on p. 10).

[48] 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. 11).

[49] 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. 11, 31).

[50] 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. 11).

[51] 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. 11).

[52] 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. 11).

[53] 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. 12).

[54] 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. 12, 13, 23).

[55] 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. 12).

[56] 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. 12).

[57] 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. 13).

[58] 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. 13).

[59] 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).

[60] 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).

[61] 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).

[62] 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, 15, 26).

[63] 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. 14).

[64] 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. 14).

[65] A. Sarkar and A. Russo, *HasTEE+ : Confidential Cloud Computing and Analytics with Haskell*, 2024. arXiv: 2401.08901 [cs.CR] (cit. on pp. 14, 23).

[66] 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).

[67] 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. 15).

[68] 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. 15).

[69] 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. 15, 26).

[70] 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. 15, 24).

[71] 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. 15).

[72] 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. 15).

[73] 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. 15).

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

[75] 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. 16).

[76] 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. 16).

[77] 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. 16, 30).

[78] 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. 16).

[79] 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. 16, 20, 23, 25).

[80] 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. 17).

[81] P. Weis, M. V. Aponte, A. Laville, M. Mauny and A. Suárez, 'The CAML Reference Manual,' Ph.D. dissertation, INRIA, 1990 (cit. on p. 17).

[82] 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. 17, 24).

[83] 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. 17, 26).

[84] 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. 18).

[85] Nordic Semiconductors. 'nRF52840DK.' (2017), [Online]. Available: https://www.nordicsemi.com/Products/Development-hardware/nrf52840-dk (cit. on p. 18).

[86] ST Microelectronics. 'STM32F4DISCOVERY.' (2011), [Online]. Available: https://www.st.com/en/evaluation-tools/stm32f4discovery.html (cit. on p. 18).

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

[88] 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).

[89] ARM. 'TrustZone for Cortex M.' (2017), [Online]. Available: https://www.arm.com/technologies/trustzone-for-cortex-m (cit. on p. 19).

[90] 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. 19).

[91] 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. 19, 23).

[92] 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. 19).

[93] 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. 19).

[94] 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. 19).

[95] 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. 19, 25).

[96] M. Sustrik. 'Structured Concurrency.' (2016), [Online]. Available: https://250bpm.com/blog:71/ (cit. on p. 23).

[97] M. Sustrik. 'ZeroMQ : An open-source universal messaging library.' (2007), [Online]. Available: https://github.com/zeromq/libzmq (cit. on p. 23).

[98] 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. 24).

[99] 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. 25).

[100] 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. 25).

[101] 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. 25).

[102] 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. 25).

[103] 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. 25).

[104] 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. 26).

[105] 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. 26).

[106] 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. 26).

[107] 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. 26).

[108] 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. 26).

[109] R. Milner, M. Tofte and R. Harper, *Definition of Standard ML*. MIT Press, 1990, ISBN: 978-0-262-63132-7 (cit. on p. 26).

[110] 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. 26).

[111] 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. 27).

[112] 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. 27).

[113] 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. 27).

[114] 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. 29).

[115] 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. 29).

[116] 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. 29).

[117] 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. 29).

[118] 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. 29).

[119] 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. 30).

[120] 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. 30).

[121] 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. 30).

[122] 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. 30).

[123] 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. 30).

[124] 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. 30).

[125] 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. 30).

[126] 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. 30).

[127] 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. 30).

[128] 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. 30).

[129] 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. 30).

[130] 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. 31).