PRIVATEEDGE: PRIVACY PRESERVED AUTOMATIC CODE OFFLOADING FRAMEWORK

by

Fatih Mustafa Kurt

B.S., Computer Engineering, Boğaziçi University, 2021

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2025

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor, who always helped me undertake this journey by supporting my decisions. I am grateful for the friendly conversations at our meetings. I would also like to say special thanks to each member of my family for supporting me both mentally and financially and encouraging me to continue this journey. Lastly, I am also thankful for my friends, who provided both distractions when needed and encouragement when it seemed impossible to continue.

ABSTRACT

PRIVATEEDGE: PRIVACY PRESERVED AUTOMATIC CODE OFFLOADING FRAMEWORK

By providing an enriched computation environment, Edge Computing enables many infeasible applications requiring low latency to be deployed at computationally constrained devices but connected to a network. However, as the computation moves away from devices towards the edge, data belonging to the user also moves outside the device, raising privacy concerns. Besides privacy concerns, it is a fact that moving computation from devices to edge requires support from applications, which inherently incurs development costs. Considering that these two challenges set a barrier to adopting the Edge Computing paradigm, this thesis proposes a framework named PrivateEdge that automatically analyzes and transforms applications to offload their subset of tasks to the edge without compromising user privacy by bringing advances in other concepts, technologies, and tools together. Internally, PrivateEdge uses LLVM to analyze and transform a given application at compile time. Additionally, it introduces the Confidential Computing paradigm for hosting a Trusted Execution Environment (TEE) on edge to secure and ensure the integrity of user data and computation. As a last feature, PrivateEdge employs WebAssembly technology on edge to provide a unified execution environment across different edge devices and secure them by sandboxing the execution of offloaded tasks. When the implementation of the framework is tested with a benchmark suite, a 15% reduction in overall execution time and also a considerable decrease in CPU utilization are observed. PrivateEdge can help many applications adapt to the Edge Computing paradigm without spending additional effort and compromising their users' privacy.

ÖZET

PRIVATEEDGE: GİZLİLİĞİ KORUYAN OTOMATİK KOD TAŞIMA SİSTEMİ

Hesaplama konusunda gelişmiş cihazları son kullanıcıya yakın konumlandırarak, Edge Computing daha öncesinde çalışması pek mümkün olmayan düşük gecikmelere ihtiyaç duyan uygulamaların, hesaplama gücü konusunda kısıtlı ama bir iletişim ağına bağlı cihazlarda çalışmasına imkan tanıyor. Fakat, yapılan hesaplamaların kullanıcının cihazından çıkıp uçtaki cihazlara taşınması ile kullanıcıya ait veriler de cihazın dışına çıkıyor ve kullanıcıların gizliliği konusunda endişeler ortaya çıkıyor. Gizlilik kaygılarının yanı sıra, hesaplama işlemlerini taşımak için de uygulama tarafından desteğe ihtiyaç duyuluyor ve bir geliştirme maliyeti ortaya çıkıyor. Bu iki zorluğun Edge Computing paradigmasının yaygınlaşmasının önünde bir engel oluşturduğunu düşünerek bu tez, uygulamaların belli başlı kısımlarını otomatik bir şekilde uçtaki bir cihazda çalıştıracak şekilde analiz eden ve gerekli dönüşümleri yapan, aynı zamanda kullanıcıların gizliliğini koruyan PrivateEdge adında bir framework öneriyor. Önerilen framework, öne sürdüğü faydaları diğer konsept, teknoloji ve araçlarda olan gelişmeleri Edge Computing paradigmasına entegre ederek gerçekleştiriyor. PrivateEdge'in içerisinde, uygulamaların derleme anında otomatik bir şekilde analiz edip dönüşümlerini gerçekleştirmek için LLVM kullanılıyor. Ek olarak, yapılan hesaplamaları ve kullanıcı bilgilerini korumak için Confidential Computing paradigmasından faydalanıyor. Ayrıca PrivateEdge, farklı tipteki uç cihazların teknik detaylarını gizlemek, uygulamaların hedefleyeceği tek bir cihaz tipi sunmak ve onların uç cihazlara zarar vermesini engellemek için WebAssembly teknolojisini uç cihazlarda kullanıyor. PrivateEdge'in implementasyonunu bir benchmark paketi ile test ettiğimizde genel çalışma zamanlarında %15 düşüş gözlemlendi. CPU kullanımında da ciddi düşüşler görüldü. Bu sonuçlar tez önerisinin, Edge Computing paradigmasının yaygınlaşmasında büyük katkılar yapabileceğini gösteriyor.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

$c(n_{tx})$	Cost function
$c_{new}(n_{tx}, n_{rx}, k, l, m)$	Improved cost function
C^k	Experiment configuration k
E_{ij}^k	Benchmark j's ith execution under configuration k
E_j^k	Benchmark j's overall execution under configuration k
P_j^k	CPU time reduction of benchmark j for configuration k
S_j^k	Speedup measured in benchmark j for configuration k
T_{ij}^k	Benchmark j's ith execution time under configuration k
T_j^k	Benchmark j's overall execution time under configuration k
U_j^k	Benchmark j's overall CPU time under configuration k

LIST OF ACRONYMS/ABBREVIATIONS

ABI	Application Binary Interface
API	Application Programming Interface
CCA	Confidential Compute Architecture
COM	Common Object Model
EPC	Enclave Page Cache
I/O	Input/Output
IP	Infrastructure Provider
IR	Intermediate Representation
ISA	Instruction Set Architecture
JIT	Just In Time
OS	Operating System
RPC	Remote Procedure Call
SEV-SNP	Secure Encrypted Virtualization-Secure Nested Paging
SGX	Software Guard Extensions
SSA	Single Static Assignment
TCB	Trusted Computing Base
TCP	Transmission Control Protocol
TDX	Trust Domain Extensions
TEE	Trusted Execution Environment
TLS	Transport Layer Security
VM	Virtual Machine
WAMR	WebAssembly Micro Runtime

1. INTRODUCTION

Many widespread computing devices, such as mobile phones and IoT devices, do not always possess the required computation power to perform the tasks they are designed for as a consequence of many factors, including limited energy consumption, mobility, and cost. The lack of necessary power forbids many practical solutions from being deployed on computationally weak devices. On this matter, Edge Computing offers excellent flexibility for the devices connected to a network by providing much more powerful computing devices at the edge of the network [1]. By offloading computationally heavy tasks to a powerful device located at the edge, many applications running on weak devices can extend their set of features with new ones that were not possible previously. Besides providing a powerful computation environment, Edge Computing can also enable devices to reduce their power consumption when they intelligently choose whether to offload or not depending on the task itself [2].

Compared to the Cloud Computing paradigm, Edge Computing mitigates many shortcomings encountered in a centralized computation environment, such as high latency, congested network, and scaling issues [3]. Contrary to the cloud, applications that utilize Edge Computing can reach edge devices, positioned one hop away, without traveling the global network, and benefit provided great computation power at lower latencies. Nonetheless, even though Cloud Computing offers excellent opportunities for many use cases for weak devices, and while Edge Computing mitigates many shortcomings, they both still face some challenges and problems that might become blockers in their adoption. These challenges, which are the main topics of this thesis, can be specified as privacy of users and development cost [3–8]. They are the challenges not inherent in Cloud or Edge Computing but are the nature of moving the computation as well as its related data from one device to an external one.

As a requirement of performing computation on a different device, offloading a task to an edge device mandates sending data related to the client or its environment,

posing the risk of disclosing client data to unauthorized entities while being transferred to or computed at the edge device. For instance, an entity with access to the network can capture transferred data and obtain any information that leaves the client's device. Moreover, Infrastructure Provider (IP) of edge devices can view both the computation and data, such as by taking snapshots of the computation's state periodically. It is also possible for an attacker who gains access to edge devices to obtain information about clients and their data. Lastly, a much worse case can be described as one of the entities with sufficient access can alter the results by modifying the computation or transferred data, which may cause the application running on the client device to misbehave. The lack of trust for the client against its environment, including the network and edge device, sets a high barrier for many applications, especially with privacy concerns, to embrace the Edge Computing paradigm.

In addition to privacy concerns, the development cost of adapting existing applications to utilize an external computing device and designing new applications from scratch with the Edge Computing paradigm in mind can be another barrier. This cost can include the development of both the application itself and also its edge counterpart. On the application side, developers must first identify a possible offloadable task set. Additionally, developers have to rewrite the identified tasks to support execution on both client and edge devices. On the edge part, a service needs to manage the edge device, handle offloaded tasks, and provide an execution environment for them. Considering the diversity of the edge devices, the service also needs to provide a unified execution environment for the client regardless of the hardware and software details of the edge device. A protocol similar to typical Remote Procedure Call (RPC) mechanisms has to be defined between client and edge devices to perform offloading cooperatively. On top of all these requirements, all offloading operations must be robust against possible failures so as not to downgrade the user experience. Satisfying all these requirements incurs a development cost.

Thanks to developments in other concepts, tools, and technologies, bringing advancements from these fields and incorporating them into Edge Computing can solve

the abovementioned problems. To start with privacy concerns, Confidential Computing [9] emerges as a new secure computing concept where the combined power of both hardware and software ensures the computation's confidentiality and integrity while preserving the data's privacy. As many hardware designers realized the necessity of trust in the performed computation, they started integrating necessary hardware changes and software support into their chips [10–13]. By providing a Trusted Execution Environment (TEE) inside the edge device, it is possible to eliminate privacy concerns that arise for the computation. Secondly, to alleviate the development cost, a tool that can enhance an application and bring task offloading support by applying some transformations to it can be developed. The primary purpose of this tool is to automate embedding Edge Computing support into applications without requiring developer effort. Additionally, this tool should provide all the requirements mentioned about the development cost above.

Following the mentioned problems and possible solutions, this thesis proposes and implements a framework named PrivateEdge, which combines different concepts, tools, and technologies. It integrates itself into an application's compile time and runtime stages to remove barriers encountered in Edge Computing with the advancements in other fields. This thesis makes three different contributions to Edge Computing by delivering them under one framework.

(i) Reduce development cost. By utilizing LLVM compiler infrastructure, while an application is being compiled, the framework automatically identifies offloadable tasks and applies a set of transformations to make the execution of identified tasks possible both on client and edge devices.

(ii) Preserve confidentiality of user at outside of the client device. The framework executes offloaded tasks on edge inside a TEE to provide a confidential execution and prevent unauthorized entities from obtaining information about user data and applied computation.

(iii) Provide a unified execution environment on edge. The framework employs WebAssembly to provide a unified execution environment on edge for applications.

By offloading the tasks in WebAssembly binary form, applications can run on a wide range of different edge devices. Internally, the framework hosts a WebAssembly runtime inside the TEE and executes offloaded tasks.

When the framework is employed on a benchmark suite with many different kinds of benchmarks, a speedup of up to seven times and some slowdowns are observed. By improving the cost function with additional information obtained after profiling the benchmarks, it is shown that slowdowns can also be eliminated.

This thesis is organized as follows. Firstly, Chapter 2 makes an introduction to the technologies utilized in the PrivateEdge to have a background about them. In Chapter 3, many research studies and works related to the contributions made by the framework will be investigated. Additionally, how the proposed framework differs from others will be discussed. Chapter 4 starts presenting PrivateEdge in detail by first describing the system model it assumes. Afterward, how it is architected to achieve its contributions will be explained. As last section, its behavior in compile time and runtime when it is applied to an imaginary application will be shown. Following the framework's presentation, Chapter 5 evaluates the framework in terms of its performance in compile time and runtime. Finally, Chapter 6 will conclude the thesis by expressing the challenges faced during the framework development and possible future works.

2. BACKGROUND

While the topic of this thesis is mainly centered around Edge Computing, it also brings multiple different concepts, tools, and technologies together with the intention of improving Edge Computing by eliminating common hurdles and problems. Because of the addition of these concepts, they deserve a little mentioning in order to provide a background and allow tracking of the upcoming chapters easily. The following sections will mention three topics. First, they will introduce confidential computing and its available implementations. Afterward, the necessity and benefits of having LLVM will be explained. Lastly, how WebAssembly's features and properties can enhance Edge Computing will be discussed.

2.1. Confidential Computing

Employing cryptographically strong encryption methods, such as AES-256 [14], can be used as a measure to secure in-transit or stored user data and prevent unauthorized access. However, most of them are not usable to perform computation on the data without first decrypting them, thereby not being suitable for preserving the confidentiality of user data. As an exception to the decryption requirement, fully homomorphic encryption schemes [15] can be utilized to apply desired computation on the encrypted form of the data without first decrypting it. Even though the utilization of this kind of method helps secure user data while being computed, a user cannot trust the output due to the lack of trust in the applied computation. This disadvantage can become an important factor in environments where computing devices are not under the control of the user, such as third-party IP.

To accommodate this requirement and establish a trust relation between the IP and the user, an execution model under the Confidential Computing name has emerged. This model provides a TEE to the user where data and computation are not observable by any entities, hence becoming a confidential computing environment. Besides the

execution environment, the user is also able to verify the integrity of the code running inside the TEE by performing remote attestation, which eventually guarantees that the applied computation is the same as the desired one. An essential element of TEE is its TCB, which contains all the components that play a role in the computation, ranging from hardware to software, including the user code. Any security issue that may arise inside the Trusted Computing Base (TCB) can compromise the TEE and endanger the confidentiality of computation and data. Considering the importance of TCB's security as a countermeasure to possible security issues, keeping the size of TCB small can be regarded as good practice.

Regarding implementations of Confidential Computing, there are multiple hardware-based examples from different hardware vendors and microchip designers. Such example comes from Intel named as Software Guard Extensions (SGX) [11], [16] that is an extension to x86 Instruction Set Architecture (ISA). The basic idea behind the SGX is separating a process into two parts: untrusted and trusted. The untrusted part runs on the processor the same as an ordinary process. In contrast, the trusted part runs in a secured enclave in which no authorities have permission to inspect its execution and memory. The application developer defines communication between these two parts and has clear boundaries. Any data passed between these parts is copied in and out since the untrusted part cannot see the trusted part's memory. At the hardware level, the enclave's content is encrypted while storing it to memory and decrypting it when it is loaded from the memory. Protecting the trusted partition's memory is not the only benefit provided by SGX solution. As a means for verifying the integrity of code running inside the enclave, SGX provides a remote attestation procedure that enables application developers to verify whether the code running on the SGX capable system is the expected one.

As another example, AMD has Secure Encrypted Virtualization-Secure Nested Paging (SEV-SNP) [17] that also extends the x86 ISA and focuses on the TEE at the Virtual Machine (VM) level, unlike SGX, instead of process level. Providing a TEE at VM level has many benefits compared to the process level, such as not requiring any

special kind of support from the application itself as long as underlying guest Operating System (OS) and hypervisor support SEV-SNP. Apart from AMD, in addition to SGX, Intel also has introduced Trust Domain Extensions (TDX), which provides a TEE at the VM level similar to SEV-SNP implementation. One downside of VM level bounded TEEs, unfortunately, is the increase in the TCB size caused by additional components added to TEE such as hypervisor and the guest OS's kernel. Due to the increase in the size of TCB, the attack vector also increases proportionally.

In addition to x86, which is extended by several different TEE implementations, considering the prominence of Arm based processors in the embedded systems and IoT field, Arm also extends its ISA to provide isolation and integrity by introducing TrustZone. TrustZone defines two worlds, Secure and Non-Secure, separated from each other with the help of hardware. By placing applications with sensitive data or resources in the Secure world, direct access to applications from the Non-Secure world can be prevented. Simultaneously, Arm enhances TrustZone with the addition of Arm Confidential Compute Architecture (CCA) [12]. Arm CCA brings code verification at multiple levels, such as VM which provides a TEE similar to SEV-SNP and TDX implementations.

As RISC-V is an ISA that grows in popularity due to its open-standard ISA and royalty-free license, multiple efforts are put towards bringing Confidential Computing to RISC-V [18–20]. For example, Multizone [18] provides a solution similar to TrustZone in terms of isolation and integrity. On the other hand, KeyStone [20] proposes an open source framework that presents a customizable TEE where hardware vendors are required to provide only basic primitives. As a more tailored solution towards providing a TEE with attestation capabilities, ACE-RISCV [19] implements RISC-V CoVE specification [13].

Another Confidential Computing implementation that is worth mentioning by differing from others belongs to NVIDIA's H100 GPU solutions [21, 22]. NVIDIA expands common TEE implementations' TCB by including their GPUs to prevent

unauthorized entities from accessing the GPU's data and enabling attesting the GPU's state. This is achieved by either assigning one or more GPUs to a single VM or partitioning a single GPU across multiple VMs while isolating each operation of VMs from each other.

2.2. LLVM Compiler Infrastructure

LLVM started its life as a research project [23] to provide a modern and Single Static Assignment (SSA) based compilation. It hosts various other sub-projects ranging from a compiler to multiple different tools to provide a compiler infrastructure. The essential one, commonly used as a base for other programming languages and the most important for this thesis is the LLVM Core project with its libraries and tools. Inside the project, there is a target-independent optimizer and a machine code generator for popular CPUs, which is built around a code representation named LLVM Intermediate Representation (IR). With the help of having a common IR that represents the code for an abstract machine, many programming languages can utilize the optimizer and code generators by outputting their source codes in LLVM IR format. Thereafter, the same optimizer can be used on the generated LLVM IR codes. Applications that exist in LLVM IR format are composed of modules representing a single translation unit similar to one source file of a C project. Inside the Module, there are global variables and functions, as well as metadata definitions that can be used during the optimizer processes and code generations.

Under the hood, the optimizer consists of many passes that perform various analyses and apply transformations to the codes, such as loop unrolling and function inlining. Owing to its pass architecture and Application Programming Interface (API) provided by the libraries, the optimizer allows itself to be extended by running custom passes on the codes. This property opens a broad range of solutions that can be applied at the LLVM IR level and benefits supporting programming languages that are able to produce such codes. Moreover, these passes can run at different granularities, such as running directly on the whole module or specifically on the function. Thanks to the

extensibility of the optimizer by adding custom passes, some of the barriers encountered in Edge Computing can be eliminated.

2.3. WebAssembly

Thanks to its portable binary instruction format, WebAssembly has become an applicable choice for applications that want to target a generic execution environment without having to consider the underlying hardware or software details. Being supported by multiple programming languages such as C, C++, Rust, or Go also helps its popularity increase. As a complementary addition to JavaScript and part of major browsers, WebAssembly enables the execution of many applications written in supported languages on the browsers. Additionally, with the help of the Emscripten compiler toolchain [24], which utilizes LLVM to perform compilation, applications can be compiled to WebAssembly to run on browsers, including necessary JavaScript glue code.

Another advantage of WebAssembly is its strong sandboxing feature in which an application is not able to interact with its environment without having access to necessary APIs imported into its execution environment [25]. By virtue of Memory Safety and Control Flow Integrity guarantees of WebAssembly, user-provided and untrusted applications can be executed inside the sandbox without risking the host environment. By protecting the call stack of execution and performing boundary checks for every access to its linear memory, WebAssembly prevents applications from accessing memory or executing a code not in its available memory regions. Furthermore, sandboxed execution helps provide faster cold time starts without inquiring about the cost of VM initialization to isolate the application from the host environment. In addition to having a faster startup time, WebAssembly also aims for an execution speed near native with the help of mostly available hardware features to reduce the cost of its sandboxing feature.

Since WebAssembly is essentially a specification of a binary instruction format, it does not have only one implementation that provides a runtime and an execution environment. However, there are multiple implementations that target different host environments, including embedded systems, edge computing environments, browsers, and many others. An example of such implementation is WebAssembly Micro Runtime (WAMR) [26], which is capable of running on a very diverse range of hardware. On the tooling side, LLVM provides code generation for WebAssembly from its IR. As a result of running on a wide range of different hardware while providing a secure sandboxed execution environment, the PrivateEdge utilizes WebAssembly as its execution environment on the edge devices.

3. RELATED WORKS

The framework employs many different research topics to accomplish its contributions. Even though it does not introduce any additional improvements to employed topics, their simultaneous uses provide a great opportunity for Edge Computing uniquely. To name these topics, PrivateEdge combines the power of Automatic Code Partitioning, Confidential Computing, and Heterogenous Computing. Accordingly, in addition to examining the studies that combine the same topics, it is also logical to look at previously done works on each topic individually.

Automatic Code Partitioning is an important research topic for avoiding human-related errors and time-saving features by performing predefined steps automatically. There are many examples of code partitioning that are operated at different levels of application representation at either compile time or runtime, such as binary or source code [27–30]. For instance, Galen C. Hunt proposes Coign [28] as an automatic distributed partitioning system to partition applications composed from distributable Microsoft's Common Object Model (COM) binaries. By utilizing the components' interface information provided by COM, the solution partitions the components into subsets by constructing a graph, representing the components as nodes and their interactions as edges. Coign defines a cost function that uses communication cost as a metric to minimize it while forming the partitions. When partitions are formed, components inside them are distributed across the available compute resources. Any communication between partitions is proxied by Coign runtime to provide a unified view of the components.

As a different code partitioning tool that defines cost function in terms of consumed power and tries to reduce overall energy consumption, Ulrich Kremer [29] suggests a compilation framework that operates at the source code level for C or Java programming languages and moves partitioning responsibility to the compiler. The framework partitions the application tasks into two different sets: client and server.

While the client partition contains all the tasks, the server partition only includes tasks that can be executed remotely on a server device. Whether a task is remotely executable depends on the communication time it takes to offload the task to the server and its execution time on the server. When the application is partitioned and compiled successfully, two binaries for client and server devices are produced. Additionally, the compiler provides necessary RPC like communication mechanism inside both binaries while compiling the application. On top of the RPC like mechanism, the compiler inserts necessary instructions to put the device into sleep to conserve energy while a task is executed remotely. Kremer reports that the prototype shows promising results, such as providing energy saving up to 10 times.

Another compiler-derived code partitioning and offloading solution is proposed for systems where client and server devices can have different architectures. Native Offloader [30] provides identifying heavy tasks of a native application and offloading these tasks to a server device that may have a different architecture. Native Offloader uses LLVM Compiler Infrastructure to first compile native application into LLVM IR, identify offloadable tasks, and then produce two separate IRs for server and client devices. Afterward, each IR is compiled with respect to its corresponding target compiler to obtain executables for both devices. Due to the architectural differences, running a native application on multiple architectures requires considering the memory layout differences and pointers. Native Offloader uses a unified memory space to support the sharing of pointers between client and server devices. Besides that, memory layout differences such as padding or alignment requirements are satisfied by forcing both executables to use the same layout.

Whether it is fully automatic or guided by the application developer, code partitioning has been an interesting research topic not only for performing task offloading but also in terms of security perspective [31–33]. Considering a security flaw inside one part of the application that runs in a single process, exploiting the flaw can compromise other parts containing sensitive information, such as private keys or passwords stored in plain text. Privtrans [31] partitions a given application in its C source code form

into two subsets according to the annotations and policies provided by the developer. These annotations are in the form of C attributes, which mark variables or functions as privileged or unprivileged. Then Privtrans propagates these annotations to the whole source code and forms two partitions called monitor and slave, where monitor contains the sensitive variables and functions. While annotations affect partitioning, policies define how the slave can interact with the monitor, such as calling functions. Policies are helpful to prevent arbitrary function execution by slave, considering that they may get compromised. Separation of an application into two privileged and unprivileged can definitely help in reducing the attack surface. However, it is unfortunately unable to protect the sensitive information when the OS itself gets compromised.

Providing isolation at the OS process level may look sufficient for the purpose of protecting sensitive information. However, this isolation does not give protection against a compromised or untrustworthy system. In this regard, TEE provides both isolation from the rest of the running system and a clear interface that defines how the isolated and remaining parts can interact with each other. Given the clear advantages of TEEs, many implementations and research about the integration of TEE into current microchips can be found [11], [12], [17], [18]. Although there are many benefits of TEEs, using these technologies can require extra considerations while integrating them into an application, such as modifying the application to run on more than one TEE implementation and ensuring correct usage of TEEs. In this aspect, Hans Winderix [34] proposes a set of enhancements and additions to LLVM compiler infrastructure that unifies different TEEs by providing a set of generic constructs for different programming languages. The developer uses these constructs to define the security properties of the application's modules inside the source code, which are then reduced by the compiler into IR that is generic over different TEEs. Afterward, this representation is compiled into a set of libraries that are appropriate for running inside the target TEE. Hans Winderix states that the current implementation supports C and Rust as source programming languages and Sancus and Intel SGX as target TEE implementations.

When usage of TEEs is moved from local devices towards edge or cloud, they can open up a broad range of solutions which enable performing computation while preventing unauthorized access to sensitive information without incurring significant computation cost [35–39]. In this regard, SCONE [35], a secure Linux container technology, utilizes Intel SGX to isolate containers running inside the Docker [40] runtime from its underlying environment. Due to the inabilities of SGX enclaves that cannot execute system calls directly, SCONE provides a C standard library interface, which implements encrypted Input/Output (I/O) and userspace threading to be used by the enclave. One disadvantage of SCONE implementation is the lack of remote attestation. As a result of the lack of remote attestation, clients who communicate with a service running inside the SCONE cannot trust its integrity, which may pose a risk of disclosing sensitive information.

It is also possible to look at the security from the IP perspective. Protecting edge devices against a malicious application is as important as securing the application from its environment. For example, an application can perform denial of service by consuming all available resources or inspect private information on the edge device if necessary safeguards are not in place, such as using sandboxing technologies. In this regard, WebAssembly [41] is increasingly becoming a popular choice for sandboxing to provide resource accounting and performing access control [42–45], especially in Cloud or Edge Computing contexts. As an example usage of WebAssembly, which targets edge environments, Sledge [42] implements a serverless framework utilizing the WebAssembly as a function execution environment. Compared to VM or container-based environments, Sledge aims to keep startup time and resource consumption low to hold properties known for Edge Computing, such as low latency. By combining the sandboxing feature of WebAssembly with the confidentiality and integrity protection of TEEs, AccTEE [43] offers a two-way sandbox that isolates the computation and its environment from each other while accounting for the resources consumed by computation. With the environment provided by AccTEE, a trust relationship between client and edge devices can be established, which enables numerous possibilities such as volunteer computing or renting compute devices by individuals. In addition to the sandboxing

feature, WebAssembly also provides a unified execution environment. For instance, the same WebAssembly binary can be executed on any supported hardware as long as functions defined in its import interface are provided. EdgeDancer [44] harnesses this unified execution environment to provide seamless migration of services between heterogeneous edge devices. Besides seamless migration, EdgeDancer employs TEEs to deliver a secure execution environment and secure migration.

All the proposed solutions examined up to this point share similar ideas with the PrivateEdge, the framework proposed by this thesis. Contrary to their proposals and implementations, which mostly focus on one or two aspects of the topics, the framework provides a fully fledged implementation. To start with automatic code partitioning, the most similar solution to the framework is the NativeOffloader. Like the framework, NativeOffloader utilizes LLVM to transform applications and generate two different binaries for client and edge devices. However, NativeOffloader does not consider the confidentiality of user data when it leaves the client device. Additionally, it needs to generate a separate binary for each possible kind of edge device, which makes the distribution of the binaries difficult and prohibits the utilization of possible future edge devices. On the other hand, the framework secures user data outside of the client device while also providing a unified execution environment on edge devices to take future devices into account.

On the edge side of the applications, AccTEE and EdgeDancer use WebAssembly to provide an execution environment for applications, as is the case for the framework. They also embrace TEE to provide a confidential execution on edge devices. However, they do not consider the client part of the application and only expect a WebAssembly binary to be supplied. Compared to AccTEE and EdgeDancer, the framework also automatically transforms applications to offload some of their tasks to edge devices. While performing task offloading, the framework ensures that the confidentiality of user data is always held when it leaves the client device.

PrivateEdge makes extensive use of improvements in different topics to enhance the current status of Edge Computing by resolving privacy concerns and also reducing the required development cost. While many works focus on a subset of these topics, the framework combines them uniquely and provides a complete solution. PrivateEdge has the potential of introducing Edge Computing to many kinds of applications that were not possible to accomplish previously.

4. METHODOLOGY

In this chapter, PrivateEdge framework is presented with all details about its architecture and behavior. The framework aims to automatically enable an application, defined in a known code representation, to utilize an edge device for offloading some of its tasks without compromising user confidentiality. Under the hood, the framework utilizes LLVM compiler infrastructure to analyze the application, identify offloadable tasks, and transform them to enable execution on client and edge devices. It also provides a service running on the edge device inside a TEE to secure user data and computation from environmental threats. Lastly, it uses WebAssembly as an execution environment inside TEE to abstract and unify the hardware and software details of edge devices.

Outlines of this chapter are given as follows. To begin with, section 4.1 models the system considered by the framework. The model is first defined by specifying its problem space. It also specifies the framework's goals to be accomplished at the end of its implementation. Following that, section 4.2 first gives a clear picture of the framework's architecture to depict a generalized overview and then dives into details by describing the internal workings of its components. Finally, section 4.3 examines the framework's behavior separately at compile time, startup, and runtime.

4.1. System Model

PrivateEdge considers a system with an application available in its source code form and a client device appropriate for running the application when it is compiled for the device. Besides that, it is assumed that the application's architecture only considers execution on the client device. In this system model, there are two different stages. The first one, the compile time stage, consists of compiling application source code to its final executable form for the client device. The second stage, the runtime stage, happens when this executable form of application starts running on the client

device. Continuing with this system model, when it is thought to be inside an edge computing environment, some additional main actors take part in the system, namely the edge device and the communication channel. Aside from the addition of these two main actors, for the application to utilize the edge device by offloading some of its tasks to enhance its computational capabilities, some set of modifications need to be applied to the application to enable task offloading. Figure 4.1 and 4.2 display both compile time and runtime stages, respectively, in detail.

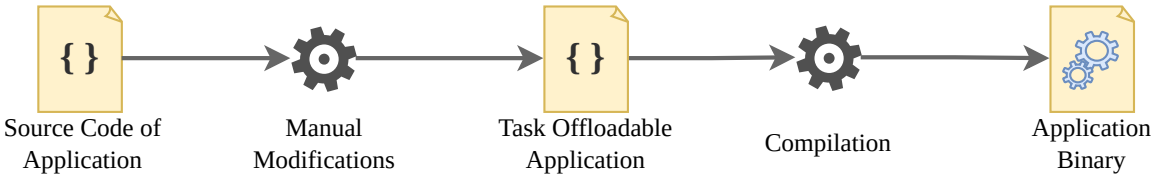

Figure 4.1. Compile time stage of system.

This model has a few disadvantages and barriers to adapting an application specifically designed for running on a single device that cannot perform task offloading to an edge device. First of all, manual modifications that enable applications to support task offloading require a development effort, which creates a high barrier to introducing edge computing. Secondly, during a task offloading and its execution on the edge device, private information located on the client device may also become observable to any authorities controlling the communication channel or edge device. Even though the communication channel can be secured by employing end-to-end encryption between the client and edge devices, the computation performed on the edge device is still not secured from IP. This disadvantage is a vital fact, especially for privacy-oriented applications. As a last disadvantage of this model, applications should be aware of both hardware and software details of the edge device in order to ensure that the edge device is able to execute the offloaded tasks.

In order to eliminate disadvantages and remove barriers, PrivateEdge introduces multiple improvements to both compile time and runtime stages. The framework is expected to transform application source code into another one capable of offloading

some parts to an edge device while preserving the confidentiality of data that leaves the client device. These expectations are specified as goals of PrivateEdge below.

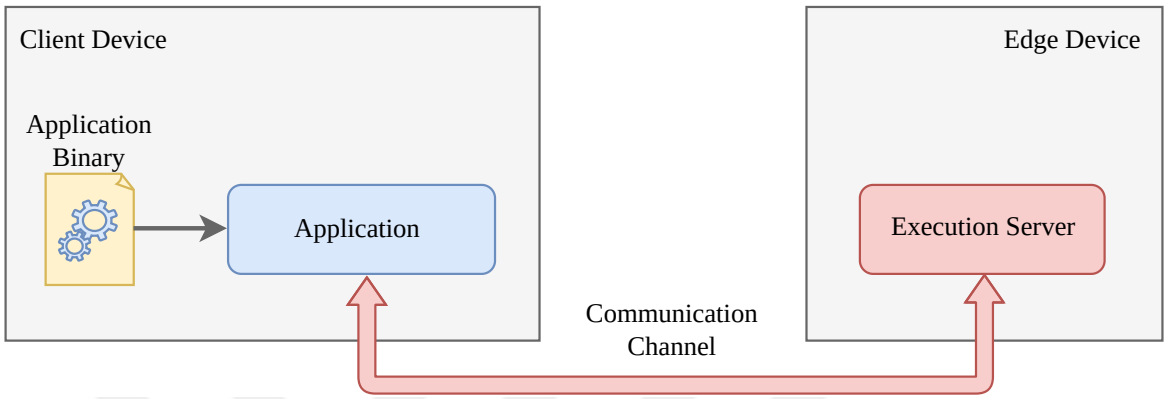

Figure 4.2. Runtime stage of system.

(i) Identify tasks that are candidate for offloading. To keep the developer interaction minimal, the framework shall be able to analyze applications in a predefined format, such as LLVM IR, and automatically identify tasks that can be a candidate for being executed on the edge device. Under which conditions a task should be considered as a candidate is left as an implementation detail and is not specified in the goals.

(ii) Transform identified tasks to make them offloadable to edge device when certain conditions are met. Following the identification of candidate tasks, the framework shall transform each of these tasks separately into a form that enables the same computation to be performed by the task in the edge device. Transformed tasks should also check task and environment-specific conditions, such as the cost of offloading the current task or the state of connection between local and edge devices, before deciding if offloading should take place or not.

(iii) Offloaded tasks shall be generic over different kind of devices to increase diversity of edge devices. With the increasing number of diverse edge devices, utilizing those devices in all possible conditions can only enhance the practicality of the application by providing a unified experience across all edge devices. With this usability feature in mind, the identified and transformed tasks shall be offloadable

to supported edge devices by the framework. For example, utilizing a VM, which is supported by targeted edge devices, can satisfy this requirement.

(iv) Confidentiality of user data shall be preserved when it leaves the client device. For the purpose of preventing one of the downsides of edge computing, which is moving user data outside the client device, the framework shall preserve the confidentiality of any data transmitted from and to the client device. Commonly used solutions such as Transport Layer Security (TLS) [46] can provide confidentiality for in-transit data. As an extension to the confidentiality preservation of data while being in transit, the framework shall be able to prevent any unauthorized access to the data while being computed on the edge device in order to keep the confidentiality of data fully intact.

(v) Offloading shall be robust against possible failures and be transparent to both user and developer. Because of numerous possible issues in offloading, such as network problems or an outage in the edge device, offloading a task is prone to failure. The failure can happen at any part of the offloading procedure, for example, while initiating the offloading or waiting for a response from the edge device. However, the framework needs to provide a transparent execution to both the user and developer, regardless of whether a task is offloaded. Thus, it shall be robust against possible failures and fall back to local execution if offloading fails at some point.

Besides the framework's goals, it is also important to specify the boundaries of the data confidentiality it provides to the application. For this purpose, the framework defines an adversarial model aiming to obtain information about data when it leaves the client device, travels over the communication channel to the edge device, and gets computed on the edge device. A clear definition of an adversary points out possible ways of leaking confidential data to the outer world of the client device. According to the runtime stage of the system displayed in Figure 4.2, the execution of an application consists of four actors inside of an edge computing environment: client device, edge device, communication channel, and IP. Over these four actors, it is assumed that an adversary has full control over the communication channel and IP, such as performing

man-in-the-middle attacks or denial of service. It is also assumed that the edge device is able to provide a TEE to process the tasks offloaded from the client device.

In this adversarial model, the framework is responsible for ensuring continuity of service and guarding confidential data while it travels from the client device through the channel and IP to the TEE inside the edge device. In other words, the framework does not provide any continuity or confidentiality guarantees inside the client device or TEE boundaries and assumes that these two actors are clear from threats. For example, since the execution of the application does not differ from other usual applications, an adversary with access to the client device can inspect the application state and obtain user data. Additionally, a security flaw in the implementation of a TEE, such as side-channel attacks [47–49], is also out of scope of the framework.

4.2. Architecture

After defining the System Model considered by the framework, an overview of its architecture can be given to understand its contributions. For the framework to meet its goals specified in section 4.1, it proposes additional improvements that span from compile time to runtime stages of the model displayed previously in Figures 4.1 and 4.2. On top of the improvements, to reduce the complexity and make it modular, the framework's architecture encompasses four different components that have distinctive responsibilities, interact with each other over a clear interface, and participate in the application's compile time and runtime stages. These four components are named as Edge Transformer (EdgeT), Bridge, Communication Server (ComS), and Execution Server (ExeS).

To start with the framework's first improvement, it replaces *Manual Modifications* step of the compile time stage with EdgeT component, which accepts an application in the LLVM IR form and automatically identifies the offloadable tasks. Additionally, it transforms these identified offloadable tasks to make them both executable on the client device and edge device, depending on conditions determined before running the

task at runtime. With this replacement, the framework is able to achieve its (i) and (ii) goals.

For a second improvement, the framework introduces the WebAssembly binary instruction format to provide a sandboxed generic execution environment for the offloaded tasks on the edge device. This instruction format is utilized by EdgeT and ExeS components by first extracting instructions of offloadable tasks into another LLVM IR, compiling it to WebAssembly, sending this binary to ExeS component at application startup, and then executing them inside ExeS when a task is offloaded to edge device. With the introduction of WebAssembly to the model, the framework fulfills the (iii) goal.

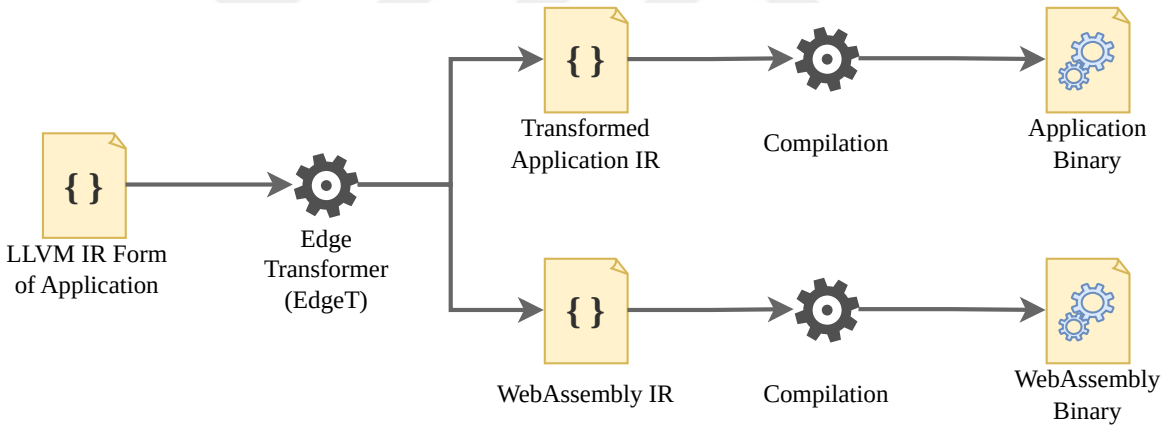

Figure 4.3. Compile time stage of improved system.

As a last improvement to the model, the framework proposes embedding a TEE into the edge device in order to prevent inspecting the offloaded task computation. With the help of TEE, data can be computed on the edge device without sacrificing its confidentiality. To complement data confidentiality outside of the client device, the framework also integrates end-to-end encryption between client and edge devices over the communication channel by employing TLS protocol. The addition of TEE and TLS helps meet the (iv) goal of the framework.

Aside from the improvements, the (v) goal also specifies that any operation performed by the framework should not disrupt the user experience regardless of the environmental conditions it faces. To satisfy that, when the application wants to offload a task, the Bridge component first checks the state of the channel and whether it can communicate with the edge device. Unless the channel is healthy, it prevents task offloading, and the execution of the same task continues on the client device. Moreover, if any failure happens during the offloading procedure, it immediately aborts the offloading and task execution starts on the client device as usual. Figure 4.3 and 4.4 display the overall system model after improvements introduced by the framework are applied to the original model.

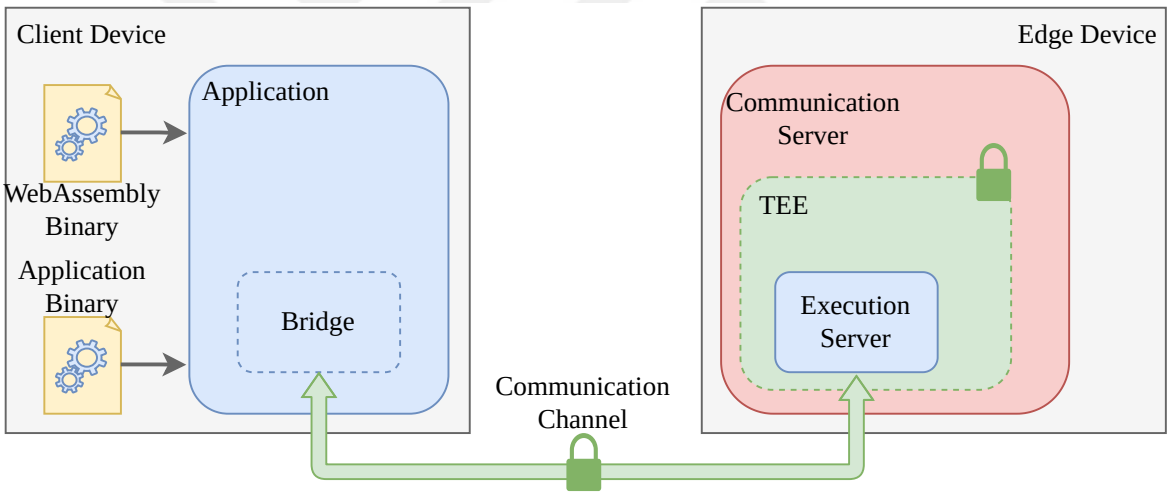

Figure 4.4. Runtime stage of improved system.

Considering that framework architecture requires both client and edge devices to work cooperatively together to accomplish a successful task offloading, it consists of components that run both on client and edge devices. Each component interacts with another one to perform a task offloading when the transformed form of the application reaches the execution of an offloadable task. Moreover, since they run in different stages and locations, they have different codebases implemented with a programming language that fits their context the most. Table 4.1 presents implementation details of components.

From automatically transforming a given application in the LLVM IR form to setting up a private edge computing environment in the runtime without requiring any additional development effort and downgrading the user experience, the framework is able to achieve all of its goals. Following the architectural overview, before understanding how the framework behaves at each stage, the upcoming sections will explain each component in detail.

Table 4.1. Component implementations.

Component	Language	LoC	Stage	Location
Edge Transformer	C++	1.3K	Compile Time	Client
Bridge	Rust	1K	Runtime	Client
Communication Server	Rust	0.4K	Runtime	Edge
Execution Server	C	1.4K	Runtime	Edge

4.2.1. Edge Transformer

An analysis that identifies remotely executable parts needs to be conducted on the application to make a subset of an application executable on more than one compute device without requiring a development effort. After conducting the analysis, corresponding transformations can be applied to the application to enable the execution of the identified parts on the appropriate computing device. It is possible to perform these two steps at either compile time or runtime, depending on various trade-offs. To name a few of those trade-offs, runtime has the advantage of having additional information such as execution environment and current user's application usage pattern. This additional information can be helpful for the analysis part and improve identifying the parts that will be the most useful if they are executed on different devices. However, performing these steps in runtime means shipping necessary tools alongside the application itself, which may significantly increase the application's binary size. Furthermore, runtime also requires supporting all possible devices clients can have. These devices can possess a processor with a different ISA and an OS with a different

Application Binary Interface (ABI). These disadvantages make runtime an infeasible option for conducting analysis and applying the transformations.

Compared to runtime, the compile time does not require any additional tools to be executed in the runtime and shipped with the application, excluding the transformed application itself. Nonetheless, compile time also experiences the same problem of supporting different configurations, like programming languages or devices, depending on at which stage of the compilation procedure the analysis and transformations will be performed. As a solution to overcome the requirements of supporting multiple configurations, a commonly supported IR format can be employed for the analysis and transformation steps. With the help of an IR format, many different programming languages and devices can be supported by first compiling the application's source code into the IR format, conducting the analysis, applying necessary transformations, and finally compiling the transformed IR to target devices. This is the point where LLVM starts playing an important role inside the framework.

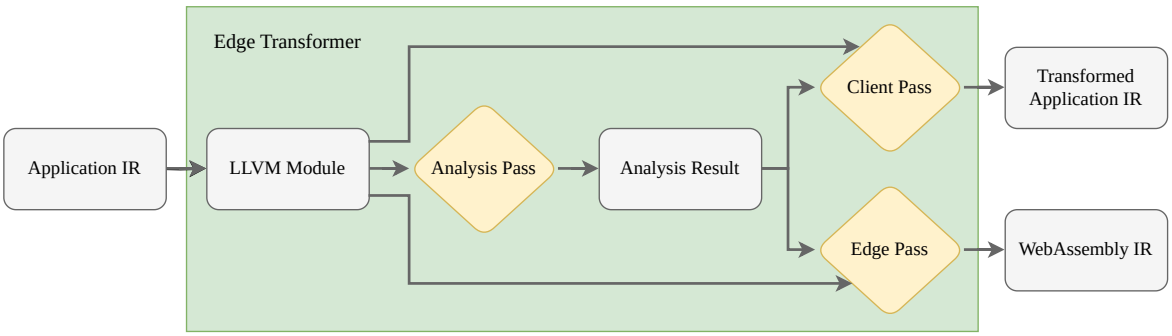

Figure 4.5. Edge transformer pipeline.

LLVM, as a modular compilation framework, offers building blocks for analyzing and transforming an application with the help of its LLVM IR format and Passes. Since LLVM IR provides an abstract machine that hides details of an actual device, many programming languages use LLVM IR as their target output to support multiple devices without spending additional effort. Considering these features of LLVM, EdgeT component utilizes LLVM to analyze the application and apply transformations at compile time. Under the hood, this component is actually composed of multiple LLVM

Passes operating at the LLVM Module level. These passes are called *Analysis Pass*, *Client Pass*, and *Edge Pass*. They perform various analyses and transformations, as their names clearly indicate. Figure 4.5 presents the relation between these passes as a pipeline.

As it is depicted in Figure 4.5, EdgeT component expects the application in the LLVM IR format as a single module to start the pipeline. When the pipeline starts running, first step is analyzing the given IR in order to identify offloadable tasks that is done by *Analysis Pass*. An important property of current implementation is that an offloadable task is defined as a function located inside the IR. Therefore, *Analysis Pass* marks the functions as either offloadable or not. Additionally, offloading a task actually means offloading a function call. After running *Analysis Pass* successfully on the IR, next step is to run both *Client Pass* and *Edge Pass* passes with the result obtained from analysis. Speaking of *Client Pass*, it is responsible for transforming the given IR by replacing any function call instructions targeting the identified offloadable functions with a set of new instructions that utilizes the Bridge component to cooperatively handle task offloading. Lastly, main objective of *Edge Pass* is generating a new IR that contains only the offloadable functions. This newly generated IR will be compiled to WebAssembly binary at compile time and sent to ExeS at startup of the application.

4.2.2. Bridge

Inside the application that is transformed by the EdgeT component, various places can trigger a function offloading. Each of them needs to know how to maintain a connection to the edge device, initialize the necessary components on it, and track the execution of the offloaded function while also handling all possible failure scenarios at each step. Embedding these capabilities into each possible place will increase the code size significantly and result in unnecessarily duplicated logic. Likewise, since they do not share a state between them, it becomes impossible to maintain a single connection to the edge device across different function offloading encountered during the execution of the whole application. To improve this situation and reduce the duplicated logic

spread around these places, *Bridge* component, which maintains a single state for the application, creates a bridge between the application and the edge device, and provides common operations via a set of exported functions, is introduced.

Bridge, in its basic form, is a dynamic library linked to the application during compile time and loaded automatically at the startup of the application. When the application starts up and *Bridge* is loaded, it first initializes its single global state that is not directly accessible from other parts of the application. Furthermore, it also manages establishing a communication channel to the edge device, keeping it alive, initializing the ExeS, and performing remote attestation to verify whether ExeS running on the edge device is the expected one. For the EdgeT component to utilize *Bridge* component and perform function offloading in the runtime, it exports four functions: `acquire_context`, `release_context`, `offloadable`, and `run_task`. These four functions are the only way to utilize the *Bridge* component and offload a task.

To start with `acquire_context`, its main objective is to provide exclusive access to the state inside the component. Exclusive access is required due to a limitation of the framework's architecture, where both edge and client only support offloading one function at a time. Hence, concurrently offloading multiple functions is not supported. However, an application can contain multiple concurrently running threads whose executions can reach an offloadable function and try to use *Bridge* component. To prevent possible race conditions and unexpected behaviors, whenever the execution of the application reaches an offloadable function, it first needs to acquire exclusive access to the state by calling `acquire_context`. This function is safe to call from any thread at any time. If the state has not already been acquired, it atomically acquires it by setting a global flag stored next to the state and returns the state to the caller. Otherwise, offloading does not start at all, and the application continues with local execution. As complementary to successfully acquiring exclusive access to the state, it should be released by calling `release_context` function. This requirement is satisfied by EdgeT component by placing necessary function call instructions to release exclusive access when the application acquires it.

After ensuring the call to `acquire_context` has succeeded and before starting the offloading, the application must decide the feasibility of the current offloading operation by calling `offloadable` function. This decision depends on a cost function, which is calculated inside the `offloadable` by taking the size of the offloading operation in bytes. Moreover, it also checks the state of the communication channel and whether it is able to communicate with the edge device. If offloading is decided as feasible, `run_task` can be used to start offloading the current function call to the edge device and receive its result.

4.2.3. Communication Server

Due to some of the TEE implementations that exhibit limited ability to interact with their environments, e.g., SGX, the edge side of the framework needs to be partitioned into two disjoint parts: insecure and secure. With this approach, the insecure part is able to interact with its environment and perform common I/O operations such as sending and receiving messages to the client. On the other hand, inside the secure part, actual computation is carried out while preserving the confidentiality of data. These two disjoint parts interact with each other via a predefined interface. With this interface, the secure part is able to communicate with the client by calling the insecure part's respective functions to perform I/O. In this framework, *Communication Server* (ComS) component implements the responsibilities of the insecure part.

For secure part to perform I/O, ComS component provides three different functions: `read_exact`, `write_all`, and `flush`. These functions are responsible for receiving incoming messages from the client, sending messages to the client, and flushing any internal buffers to ensure that they are sent. Under the hood, ComS hosts a simple Transmission Control Protocol (TCP) server where it first initializes the secure part and then starts listening on a given IP address and port at the startup. One thing to note is that the discovery of ComS's connection details is out of the framework's scope. Thus, clients have to be correctly configured to establish a connection successfully. Since the primary motivation of ComS is only to provide a communication

channel between the client and the secure part, it is designed to be a generic service over different kinds of TEE implementations. Depending on the TEE implementation edge device has, ComS will load and initialize the appropriate library to create the secure part where ExeS component is also embedded.

4.2.4. Execution Server

As complementary to ComS, Execution Server (ExeS) implements the secure part of the service running in the edge device. To assist the clients on their offloading requests by dispatching necessary functions with appropriate parameters, ExeS provides a WebAssembly runtime that is able to run inside of a TEE and only requires functionalities provided by ComS to interact with its environment. This WebAssembly runtime is powered by WAMR [26], which provides a WebAssembly execution environment with its compiler. WAMR is a high-performance and highly portable WebAssembly implementation compatible with Intel SGX. Furthermore, thanks to its Just In Time (JIT) compiler, the received WebAssembly binary is compiled to native machine code for edge device and executed near to native speed. This WebAssembly runtime inside the ExeS is initialized with the binary received from the client.

Architectural-wise, ExeS is split into two parts to support multiple TEEs easily. There is a core part where an event loop and the WAMR are located. Functionalities related to TEE are located on the other part, the wrapper part, which actually wraps the core part. Each different TEE implementation has a dedicated wrapper. The expected behavior from these wrapper implementations is to provide a channel that enables reading and writing messages to be exchanged with the clients while securing and conserving the confidentiality of the messages. In fact, they encapsulate the three I/O functions provided by the insecure part with a TLS channel whose keys are derived from a trusted source that the client can verify. As an example, wrapper implementation belonging to Intel SGX can integrate TLS setup into remote attestation procedure [50] to secure the communication channel while enabling remote attestation.

When the client completes remote attestation and initialization of ExeS with the WebAssembly binary distributed with itself, ExeS starts an event loop in which an offloading request is awaited from the client. After receiving an offloading request, ExeS locates the requested function in the binary and calls it with the parameters received from the client. Following the successful execution, ExeS sends the outputs produced by the function back to the client. This operation resembles a typical RPC mechanism specialized for the framework.

As the final component in the architecture and giving an overview of it, a more detailed explanation of how the framework behaves in the compile time and runtime can be given by demonstrating how the four components work internally and interact with each other.

4.3. Behavior

4.3.1. Compile Time

The framework starts applying its improvement beginning from the compile time stage by introducing the EdgeT component to the system. As shown in Figure 4.5, EdgeT component starts its phases by running the *Analysis Pass* on the LLVM Module, which is constructed from the given application IR. In addition to the module, *Analysis Pass* expects an entrypoint function, which is the first function to start analyzing the offloadability. Starting from the entrypoint function, the pass traverses and analyzes all the functions that are reachable from the entrypoint by being the target of either `invoke` or `call` instructions.

When the pass starts analyzing a function to decide whether it is offloadable or not, it basically checks if it has any instructions that can alter the state of the application other than the parameters given to it while being executed. To make it more concrete, the pass marks a function as not offloadable if it has one of the conditions defined below. Although all these restrictions may not be necessary and it is possible

to lift some of them in order to make more functions offloadable, solutions that lift them also significantly increase the complexity of analysis and transformation parts.

(i) Being only a declaration. In a given module, not all functions have to be defined. A function can also have only a declaration and be called by other functions defined in the module. More specifically, a function may lack the body that contains the instructions. Since a function with only a declaration cannot be analyzed for its offloadability, *AnalysisPass* directly marks them as not offloadable. This case is shown in Figure 4.6.

```
1 declare i32 @draw(i32 %0)
2
3 define i32 @main(i32 %0, ptr %1) {
4   %2 = call i32 @draw(i32 %0)
5   ret i32 %2
6 }
```

Figure 4.6. A function with only declaration is not offloadable.

(ii) Calling not offloadable functions. In order to offload a function to the edge device, all functions it calls have to be offloadable, too. Otherwise, a function located at the deeper part of the call stack may try to alter the application state while being executed on the edge.

(iii) Modifying or accessing a global variable that is not constant. When a function modifies or accesses a global variable that is not constant, executing it remotely also requires always sending the latest value of this variable and receiving it in order to keep the application state intact in the client part. The exact requirement is also valid for other functions located inside the call stack of the offloaded function. Hence, an offloaded function needs all the global variables it and its callee functions access regardless of whether or not they will be used for current execution. To ease the complexity of keeping the application state intact, *AnalysisPass* marks functions, which modify any global variables or access a non-constant global variable, as not offloadable.

(iv) Having an indirect call. Indirect calls are similar to normal function calls except that the called function is stored behind a pointer whose value is only known at runtime. Since it is impractical to know all possible functions that can be stored behind this pointer, *AnalysisPass* cannot decide whether or not the function behind the pointer is offloadable. As a consequence of this fact, any function which contains an indirect call is marked as not offloadable.

(v) Containing any target specific instructions. While an IR file is generally target-independent, it can also contain target-specific instructions, which are mainly assembly instructions or function calls made against target-specific LLVM intrinsics. Since it is not possible to support target-specific instructions in WebAssembly runtime, the pass marks these functions as not offloadable.

(vi) Having a variable amount of parameters. Functions defined in the module can accept a variable amount of parameters which is not supported by EdgeT component. A function that has a variable amount of parameters is marked as not offloadable.

(vii) Having at least one parameter or return value that is not serializable. EdgeT component does not support transferring all kinds of values between client and edge devices. Even though most of the primitive types are serializable, such as integers and floating point numbers with different bit sizes, only a few special kinds of pointers are serializable by the component. For example, a pointer parameter that does not have a `noalias` attribute cannot be used safely on the edge due to the possibility of pointing to the same memory region by other parameters. Since any modification applied on one pointer should also be visible on the other pointer parameter, and tracking these aliased memory regions can become very hard, functions with possibly aliasing parameters are marked as not offloadable.

Another case where serialization is not supported is pointer parameters whose pointed value also contains another pointer. EdgeT component currently does not support nested pointer serialization.

As a final restriction for serialization, all parameter sizes must be known at compile time. As a consequence, the framework cannot serialize the pointer parame-

ters whose size cannot be determined statically.

(viii) Diverging the execution. A function does not always have to return back to its caller when called. These kinds of functions are indicated by a `noreturn` attribute on their declaration. Since there is no use in offloading functions with `noreturn` attribute, they are marked as not offloadable.

After successfully completing the *Analysis Pass*, EdgeT component starts running *Client Pass*, which uses the analysis result and transforms all the `invoke` and `call` instructions of the given module that target the identified offloadable functions. When an instruction is found, it is transformed by *Client Pass* by replacing it with the instructions performing the following steps.

(i) Calculate the size of offloading in terms of number of bytes. Before performing the operation regarding offloading the current call instruction, the operation's whole cost should be identified first. Moreover, identifying the cost of this operation also requires determining the number of bytes that should be transferred to the edge. This is accomplished by iterating over each parameter passed to call instruction while checking their types. In LLVM IR, nearly all types have a fixed size, which is useful for finding the size of the buffer required for serialization. Besides that, the *AnalysisPass* ensures that no function with a parameter whose size is not known is marked as offloadable.

To illustrate that, consider the `i32` and `i64` types, which represent the 32-bit and 64-bit signed integers, respectively. As their names indicate, they have a known and fixed size that is not dependent on the underlying device. At first, one can imagine that summing the size of each parameter passed to the call instruction can yield the correct number of bytes. However, a few factors need to be considered when considering the total size, namely the alignment requirements of each parameter and pointer parameters.

Starting with the pointer parameters, their type is special compared to other types. The value of a pointer type is essentially a memory address pointing to another value whose type may or may not be known at compile time. As a con-

sequence of not knowing the type of pointed value, the size cannot be determined by looking at the type. Although most of the pointer parameters' sizes cannot be found, there are a few cases where necessary information exists and can be used. For example, pointer parameters can have `derefenceable(<n>)` [51] attribute, indicating that the pointer is dereferenceable for `n` number of bytes starting from the pointed address. As long as the pointed value also does not contain another pointer, which is assured by checking whether any pointer is derived from this parameter at *AnalysisPass* stage, the `dereferenceable(<n>)` attribute can be used for determining the size of the pointed value. After determining the size, performing a simple memory copy operation from pointed value to serialization buffer is sufficient to serialize it while keeping the alignment requirement specified with `align <n>` in mind.

In addition to `dereferenceable(<n>)` attribute, pointer parameters can also have `align <n>` attribute, which implies the expected alignment of pointed values in the memory. Each pointer parameter must have the specified alignment `n` when an offloaded call instruction is executed on the edge. Otherwise, the behavior is undefined. Alignment requirements can be fulfilled by either ensuring that all parameters have necessary alignments in the serialization buffer while serializing them in the client or copying each value in the buffer to other memory locations with required alignment while deserializing them on the edge. For optimization purposes, the latter solution, copying values from the buffer to another location on the edge, is not preferred. Instead, all parameters with `align <n>` attribute serialized into a buffer while preserving their alignment requirements by putting necessary padding bytes in front of them. Furthermore, on the edge side, the start address of each serialized pointer value inside the buffer is passed to call instruction. Consequently, no copying operation needs to be done, and the called function directly uses values located inside the buffer.

To sum it up, a relatively simple operation, calculating the required buffer size to serialize the parameters, can become complicated when pointers and alignment requirements are considered. While keeping these factors in mind, the pass calculates the size at compile time and uses it in the next step.

(ii) Decide whether offloading is possible and also feasible. Following the calculation of the required buffer size for serialization, a decision about offloading the current call instruction to the edge or executing it on the client can be made depending on a few factors. Details of these factors are hidden inside the `offloadable` function located inside the *Bridge* component. *Client Pass* inserts a call instruction for `offloadable` function with the required buffer size as a parameter. If the call returns a non-null pointer, which indicates offloading is possible and the returned pointer should be used as a serialization buffer, execution continues to the next step to start serializing the parameters. Otherwise, execution falls back to local execution.

(iii) Serialize the parameters. When a non-null pointer is returned from `offloadable` call, which points to a byte buffer, *Client Pass* can insert necessary instructions to serialize parameters used for the current call into this buffer. Moreover, the capacity of this buffer is guaranteed to be at least as large as the required buffer size. This ensures that overflowing the given buffer while serializing the parameters is not possible.

Starting from the first parameter specified in the offloaded function declaration, each parameter is serialized into the buffer, and the necessary padding bytes are put in place to meet the alignment requirements. Considering primitive types such as `i32` or `double`, their serializations only contain a simple `store` instruction. For pointer type, a call to `llvm.memcpy` intrinsic is performed to copy bytes from the pointed value to the buffer. An example for illustrating the serialization of such types is shown in Figure 4.7. It should be mentioned that this example lacks necessary instructions incrementing the `buffer` pointer after serializing each parameter.

```
1 store int 10, %buffer, align 4
2 store double 10.0, %buffer, align 8
3 ; Copy a 128 bytes from %parameter into %buffer
4 call void @llvm.memcpy(ptr %buffer, ptr %parameter, i64 128, i1 0)
```

Figure 4.7. Serialization of primitive types.

(iv) Execute the function on edge. By the time all parameters get serialized into a buffer, a call instruction is inserted for the `run_task` function. This function handles most of the common operations related to sending an offloading request to the edge and receiving its response while also handling the possible failure scenarios. By utilizing a single function for common operations and not inserting many similar instructions, the size of both code and, naturally, the size of final outputs can be kept identical to its unmodified form.

As parameters for `run_task` function call, the size of serialization buffer and also `ID` of the offloaded function are passed. In this case, the `ID` is an `i32` constant automatically generated by the pass and unique across the offloaded functions. Up on completion, `run_task` function returns a boolean value indicating whether the offloaded function is executed on the edge successfully or not. For a successful offloading case, the execution continues to the next step, where deserialization of the buffer received from the edge happens. On the contrary, if offloading fails for any reason, the execution falls back to local execution, where offloaded call instruction is executed on the client.

(v) Deserialize the returned values. As the next step after the successful return of `run_task` function call, necessary instructions that perform deserialization of values returned from the edge can be inserted. When offloading is completed, these values now reside in the buffer previously used for serialization purposes. Obviously, the buffer includes the return value of the offloaded call unless its type is `void`, indicating that nothing will be returned. Besides the return value, it can also contain the parameters passed to call instruction due to the possibility of modification applied on the edge.

One case where a modification is applied to a parameter on the edge that also needs to be reflected on the client part is pointer type parameters. Their values are serialized on the edge, similar to how they are serialized on the client side. Afterward, for each pointer parameter, a `llvm.memcpy` instruction is inserted on the client side to copy from the buffer to the pointed address by the parameter. One notable exception for the deserialization of pointer parameters compared to their serialization on the client is that not all of them need to be received from

edge and deserialized. Another useful attribute, called `readonly` and only used for pointer parameters, indicates that no writes will be performed through this parameter. By combining the power of both `readonly` and `noalias` attributes, it can be assured that the value pointed by this parameter will not be modified during the execution of the offloaded function. Therefore, there is no need to send these values back to the client since they will not be modified in any case. In its final form, the deserialization buffer only contains the return value of the offloaded function and pointer parameters without `readonly` and `noalias` attributes. *Client Pass* inserts necessary instructions to perform memory copy operations for pointer parameters and deserialize the return value to use it in the next step.

(vi) Use the execution result. As a final step, all of the uses belonging to the call instruction's return value need to be replaced by a new value that is conditionally determined by the successfulness of the offloading operation. To accomplish that, *Client Pass* replaces all uses of the original return value with a new `phi` [51] instruction. This `phi` instruction accepts two basic blocks as its predecessor. These two basic blocks are where deserialization occurs, or local execution of original call instruction. Depending on which basic block has branched into the `phi` instruction, its result will be used. With the `phi` instruction, *Client Pass* successfully completes its transformation for one call instruction.

By the end of the transformation of all `call` and `invoke` instructions that target an offloadable function, *Client Pass* completes its task and produces an IR that is able to offload some part of its execution to an edge device with the help of *Bridge* component. As the last step of the EdgeT component in the compile time stage, *Edge Pass* generates a new IR which contains only the offloadable functions to compile them afterward to a WebAssembly binary. While generating new IR, *Edge Pass* applies the following steps.

(i) Copy each offloadable function from original IR to new IR module. At first, copying the definition of a function between different modules can be viewed as

an easy operation. However, due to the values located outside of the function and accessed by an instruction in the function, such as indexing a global variable with `getelementptr` instruction, these definitions also need to be copied to the new module. Furthermore, because a value can also hold other values, the copy operation needs to be done in a depth-first fashion. *Edge Pass* does this by iterating over each instruction in the function and recursively checks whether it has a value that needs to be copied too until all of the values used by this instruction are copied. While most of the values can be copied directly between modules, values with pointer type need additional care due to possible pointer width differences between client devices and WebAssembly runtime.

Regarding the example specified in Figure 4.8, a global variable with a packed struct type `%packed_type` where the client device's pointer width is 8 bytes, a compiler can generate instructions that access the second field of this variable by multiple ways specified in the same listing. The first generated `getelementptr` instruction is valid for both the client device and WebAssembly. However, while the second `getelementptr` instruction is valid for the client device, it becomes invalid when it is moved to the new module. Due to the 4-byte pointer width in the WebAssembly virtual machine, the offset of the second field inside the packed struct becomes 4 bytes instead of 8 bytes. To accommodate this difference, *Edge Pass* adds additional padding bytes to a global variable whose type is a struct and also contains pointer types just after its pointer types. For example, the type in the listing would look `<{<{ptr, [4 x i8]}>, i32, <{ptr, [4 x i8]}>}>` just after it is copied to new module.

```
1 %packed_type = type <{ptr, i32, ptr}>
2 @global_var = constant %packed_type <null, 0, null>, align 8
3
4 getelementptr %packed_type, ptr %global_var, i64 1
5 getelementptr i8, ptr %global_var, i64 8
```

Figure 4.8. Multiple ways of accessing second field of a packed struct.

Another possible risk of copying values between modules can happen in the runtime. If one of the values gets modified during the execution of the application in the client or edge, the other part will observe the old value, which can result in an unexpected state. Nonetheless, this risk is avoided inside the *Analysis Pass* by not including functions that can access or modify a global variable that is not constant. The state of the application in both the client and edge parts will not diverge, and there is no need to provide synchronization of global variables between the client and edge. Once all offloadable functions are copied to the new module, the pass continues with the next step, which provides a generic entrypoint for each copied function.

(ii) For each offloadable function, create a new wrapper function callable by ExeS component. Considering that the ExeS component should be able to call a specific function requested by the client and also located inside the WebAssembly binary, it needs to know the function's return value, number of parameters, and their types to perform the call successfully. Equivalently, for each offloaded function, a corresponding wrapper function that implements a known interface and handles deserialization, function calling, and serialization parts can be generated by *Edge Pass*, which then can be called by ExeS. The second solution keeps the ExeS simple, in terms of knowing only one type of function to call, and moves the responsibility to compile time. Additionally, knowing that *Client Pass* already handles the serialization and deserialization of parameters at compile time, the effort of performing similar operations in *Edge Pass* is reduced significantly.

A wrapper function for an offloaded function accepts two pointers as parameters, specifically the deserialization and serialization buffers. It also returns a value with `i32` type in order to tell back to ExeS about the number of bytes written into the serialization buffer. As opposed to the client, edge uses different buffers for serialization and deserialization due to the additional return value that should also be serialized. In the runtime, when ExeS receives an offloading request from the client, it finds the corresponding wrapper function and calls it with the buffer received from the client and another buffer it stores in its state.

Inside the body of a wrapper function, *Edge Pass* inserts necessary instructions that deserialize the parameters the offloaded function needs. During deserialization, values of pointer parameters are calculated as the absolute address of their starting index inside the deserialization buffer. Since the client guarantees that the alignment requirement is already met for each pointer, the server does not need to consider alignment as long as the deserialization buffer is aligned correctly. This is a considerable optimization that eliminates unnecessary memory allocation and also copy operation. Following the deserialization, a simple call instruction for the offloaded function with deserialized parameters is inserted. Subsequently, the returned value from the call is first serialized into the serialization buffer, followed by the pointer parameters without a `readonly` attribute. Finally, the number of written bytes into this buffer is returned from the wrapper function.

(iii) Create a function that returns the list of offloadable functions. In order for the ExeS component to identify which functions are exported from the WebAssembly binary that is sent from the client and dispatch incoming offloading requests to respective function for the given `ID`, ExeS calls a predefined function named `entry_point` after the initialization of runtime environment. This function is automatically created by *Edge Pass*. This function returns a pointer value pointing to a constant string consisting of exported functions' names. ExeS component parses the string and identifies names of offloaded functions in addition to their `ID`s.

With the addition of `entry_point` function, *Edge Pass* ensures that the new module is valid and completes generating new IR. After two IRs have been produced successfully, they are compiled to their respective targets, namely client device and WebAssembly. First of all, the transformed form of the application is compiled into an executable suitable for the client device. Additionally, during the compilation, the dynamic library form of *Bridge* component is linked with the application to make it loaded at the startup. Lastly, the generated IR, which only contains offloadable functions, is compiled into a WebAssembly binary. These two compilations mark the

end of compile time stage for the framework.

4.3.2. Startup

When the transformed form of the application is compared to its original form, it performs a few additional steps at its startup to initialize the *Bridge* component. In addition to the client, edge also has a few startup steps performed independently from the client, such as starting the ComS component. In addition to how they interact with each other, the startup steps of both client and edge devices are presented in Figure 4.9 as a sequence diagram. Once both devices reach successfully to the end of the diagram, offloading can be performed when the application execution reaches an offloadable function.

To start with the edge device, in order to let the client establish a connection, it first starts a new instance of ComS component. When ComS starts running, it initializes a suitable TEE for the edge device by loading a dynamic library and calling its initialization function. During the initialization of TEE, an instance of ExeS component is also created and initialized. As a final step in the startup of ComS, once the ExeS component is ready for use, it starts a TCP server by listening on a port for the client to connect and create a communication channel. Since the discovery of ComS's connection details is out of the framework's scope, clients must be correctly configured to establish a connection successfully.

Moving to application startup on the client, when it starts running and *Bridge* component is loaded by the dynamic linker of the OS, *Bridge* starts initializing its single internal state inaccessible from other parts of the application. Inside this state, a new buffer is allocated to store both function parameters while offloading and also output values while receiving the execution result. The buffer size is determined at startup and kept constant throughout the application execution. In addition to setting up the state, it also creates a new OS thread, called *Connection Thread*, to establish a connection to the server and manage its lifecycle, such as reconnecting when the connection gets

broken without interfering with the application itself. To send an offloading request and receive the response from the server, the application communicates with the connection thread via a channel. These parts are abstracted away inside the library, and the application does not need to consider details of it.

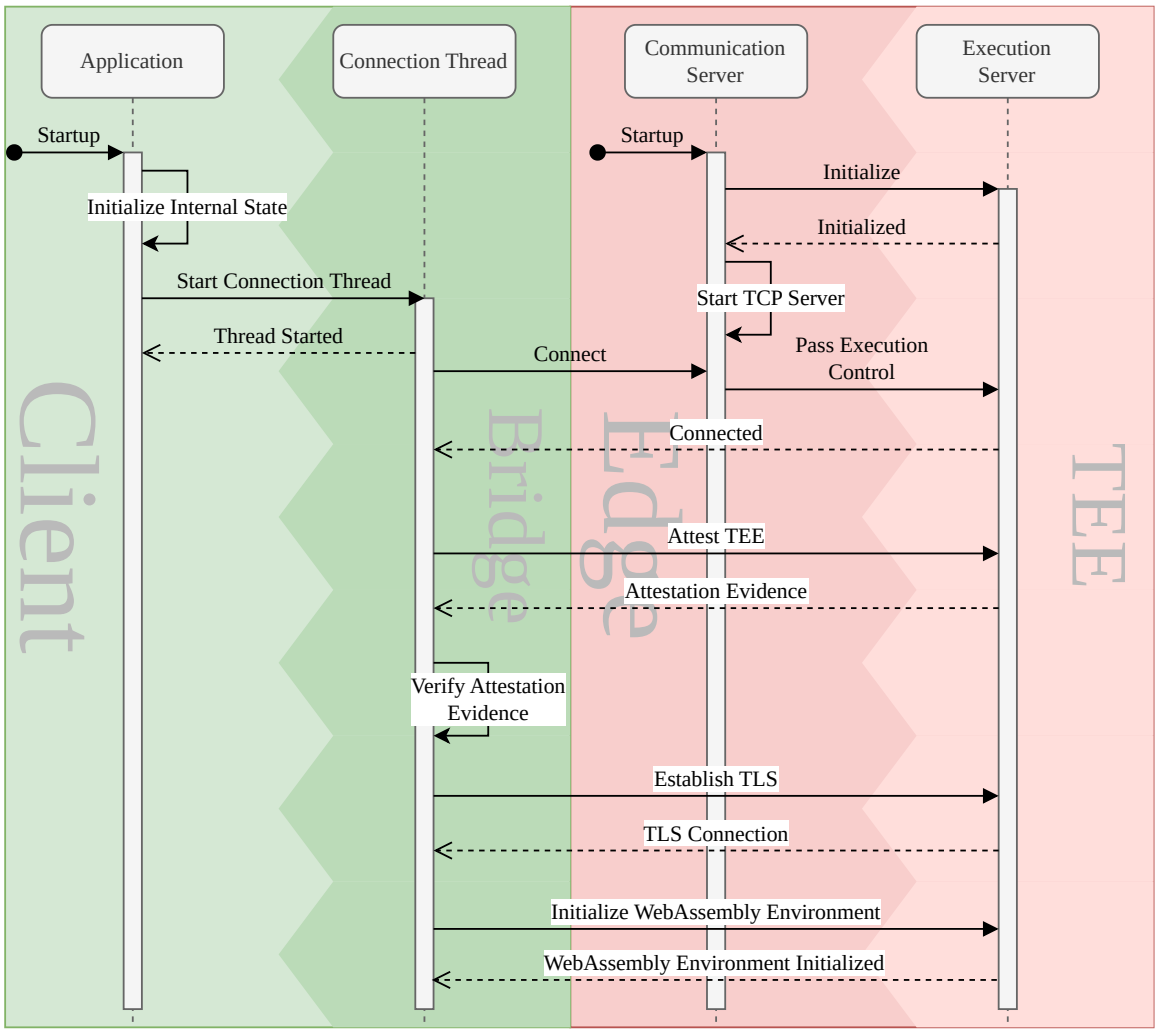

Figure 4.9. Startup stage.

As both devices complete their independent startup steps, *Connection Thread* starts interacting with the edge by sending a connection request to ComS component. When ComS accepts the incoming connection request, it passes execution control from itself to ExeS component by calling the entrypoint function provided by ExeS. Besides execution control, it also passes an opaque pointer pointing to a stream object

that holds the underlying connection to the client. ExeS component passes this opaque pointer back to ComS to communicate with the client when it calls one of the three I/O functions provided by ComS. As ExeS receives the execution control, it notifies *Connection Thread* about the successful connection. Following that, *Connection Thread* starts performing remote attestation by requesting an attestation evidence from ExeS. Upon receiving the evidence, it verifies it by following the attestation procedure of the edge device's TEE. To complete setting up a secure computation environment and communication channel, *Connection Thread* and ExeS establish a TLS channel over the existing one. As an enhancement, when the underlying TEE implementation is considered to be Intel SGX, establishing a TLS channel can be integrated into the remote attestation procedure [52] to secure the communication channel while enabling remote attestation.

To finalize the startup stage, *Connection Thread* sends the WebAssembly binary distributed with itself to the ExeS component in order to initialize a WebAssembly execution environment. When ExeS receives the binary, it creates and initializes an instance of WAMR. In addition to WAMR initialization, ExeS also calls the `entry_point` function created at compile time by EdgeT component and located inside the binary to identify the names of offloadable functions and their `ID`s. Considering all steps are successfully completed, ExeS notifies the client about successful initialization and starts its event loop, where it waits for offloading request from the client. Otherwise, when a step fails or the connection gets broken, all the steps, starting from sending a connection request, will be repeated after waiting a predefined amount of time. With the successful initialization of the ExeS component, the startup stage is finished by establishing a secure communication channel and setting up an execution environment on the edge for task offloading.

4.3.3. Runtime

In the runtime, the application continues its execution as usual until it reaches a function transformed by EdgeT component at compile time. At this point, the

application starts making multiple calls to *Bridge* component's functions to perform the offloading operation. Each step of this flow is visually displayed in Figure 4.10 as a flow chart.

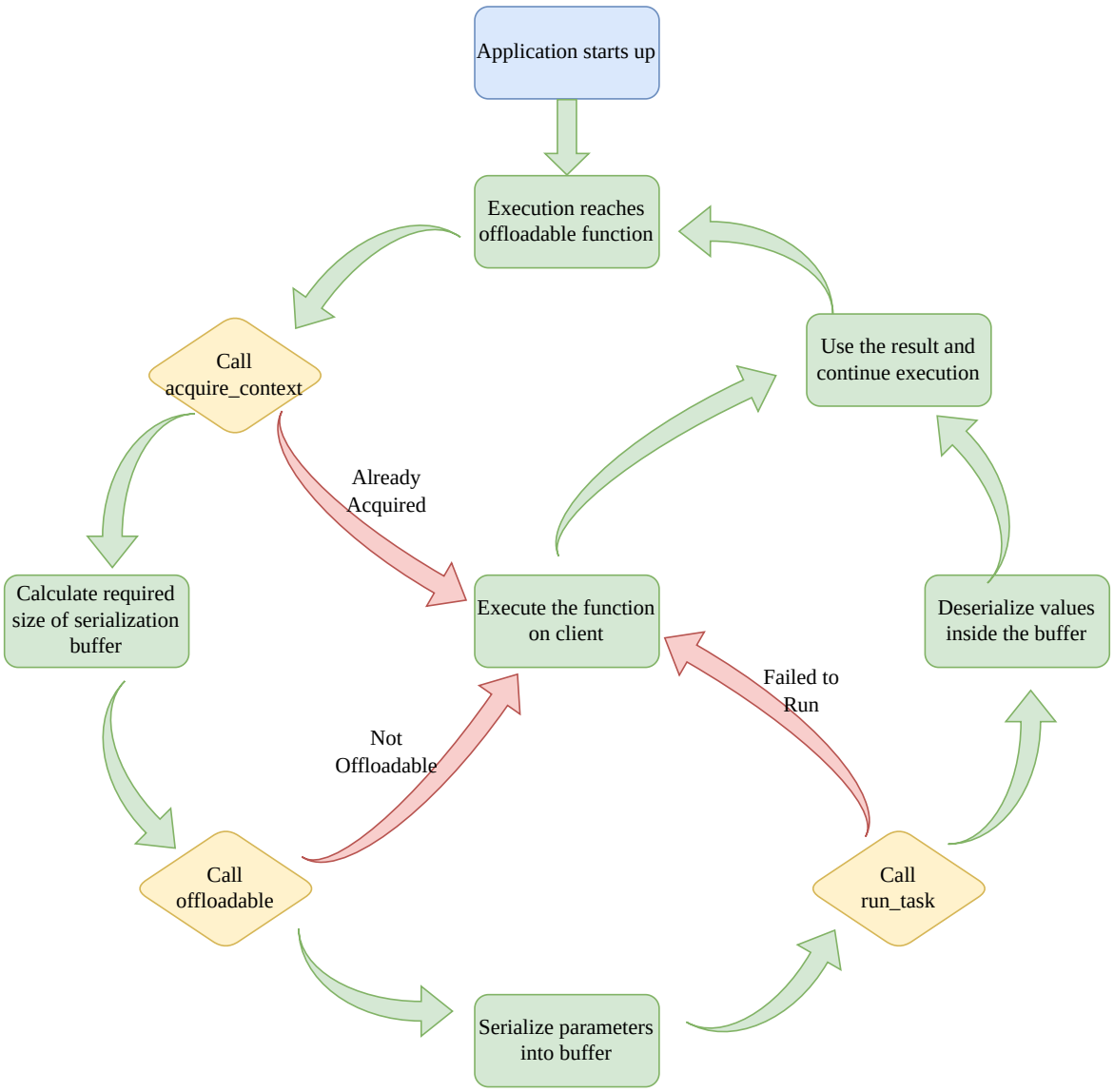

Figure 4.10. Execution of offloadable task on client.

First of all, since the framework does not support concurrent task offloading, the application needs to acquire exclusive access to the internal state of *Bridge* component to offload one task at a time by performing a call to `acquire_context` function. Once exclusive access is acquired, the size of the buffer, which will be used for serializing the

values required by the function, is calculated and used while calling the `offloadable` function. Inside this function, the feasibility of offloading is decided with respect to two factors. The first factor is whether the required size of the serialization buffer is larger than the currently allocated buffer's size. If it exceeds the existing buffer size, offloading is considered not feasible. Even though this limitation can be solved by reallocating the buffer to meet the size requirements, because of possible failures in reallocation and its cost, the initially allocated buffer is not changed during the application's lifetime. On the other hand, the second factor is related to the health of the communication channel. Considering that establishing a connection to the edge device and initializing its runtime can take longer than executing the current task locally, offloading is considered not feasible when there is no active connection. Additionally, establishing a connection is not always guaranteed to succeed due to multiple reasons, such as long-lasting network failures. Because of multiple possible failures that can happen during the connection setup, the time to obtain an active connection cannot be predicted. This behavior plays an important role in avoiding possible slowdowns in the application and keeps it continuing its functionalities under all conditions.

If current function offloading is decided as feasible, `offloadable` function returns a pointer pointing to the serialization buffer. The application can now start serializing the offloaded function's parameters into this buffer and call the `run_task` function to start offloading to the edge. `run_task` handles all the details of tracking the offloaded function's execution on the edge to catch any possible failures that can happen. Once the execution completes successfully on the edge and `run_task` receives the output into its buffer, it notifies the application about successful results. Then, the application starts deserializing the values inside the buffer to use it, calls `release_context`, and continues its execution as usual. Contrary to successful offloading, when offloading fails or is stopped due to failing to acquire context or being decided as not feasible, the application executes the offloadable function locally and continues as usual. This case is highlighted with red arrows in Figure 4.10. Additionally, excluding the failed context acquiring case, a call to `release_context` function is made to ensure that it stays available to other function offloadings.

To illustrate what happens when `run_task` is called, Figure 4.11 displays a flow chart that presents steps taken in both client and edge devices. When the application calls the `run_task` with the `ID` of the offloaded function, the size of the serialized form of parameters, and the buffer, it simply forwards these values to *Connection Thread* through a channel. Upon *Connection Thread* receives the values, it sends them through the secure communication channel to ExeS component. Then, ExeS component searches the name of the function with given `ID`, locates its wrapper function inside the WebAssembly environment, which handles the execution of the requested function, and calls it with the values received from the client. Beneath the surface, the wrapper function deserializes the parameters and passes them to the requested function. Additionally, it also serializes both the function's return value and the parameters that might be modified during the execution into the buffer given by the ExeS component. As the final step on the edge side, the ExeS component sends the buffer filled by the wrapper function to *Connection Thread*. Lastly, *Connection Thread* sends the buffer received from ExeS back to the application.

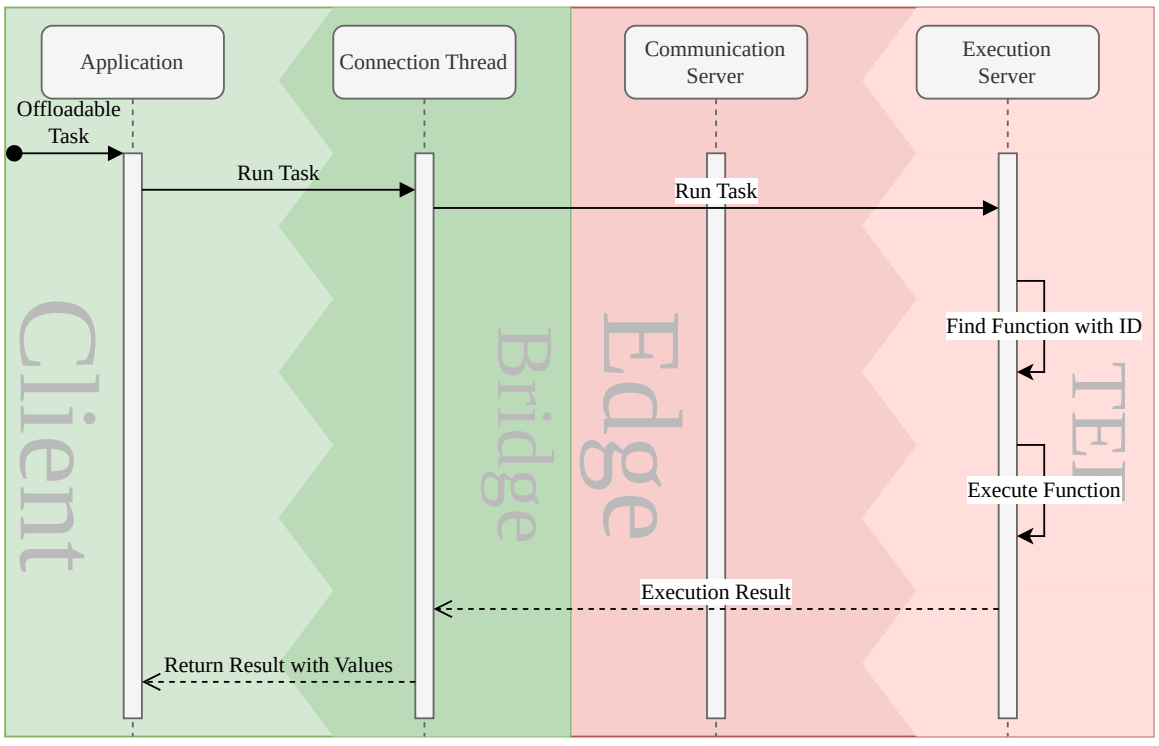

Figure 4.11. Task offloading between client and edge.

In terms of possible failure cases that can happen during function offloading, a few of them are handled by *Connection Thread*. One of the failure cases is related to network problems, which prevent communication. In this case, *Connection Thread* aborts the function offloading and notifies the application about the failure. Secondly, execution of WebAssembly binary can throw an exception such as trying to access out-of-bound memory or executing an `unreachable` instruction. In either case, the ExeS component notifies the failure to *Connection Thread*, which forwards it to the application.

With details of the interaction that happens between the client and edge devices during function offloading, the behavior of the framework in the runtime stage can be considered as explained. In the runtime stage, thanks to cooperation between the components, the framework handles all the operations related to function offloading without requiring any additional development efforts and causing any deterioration in the user experience. To understand the possible impacts of the framework on the user experience, the next chapter will evaluate the framework by performing a set of experiments.

5. EVALUATION & EXPERIMENTS

PrivateEdge implements a framework with an extendable architecture that helps applications adapt to the Edge Computing paradigm without incurring any development cost while resolving privacy concerns. Its extendable architecture consists of multiple components running at compile time and runtime, as well as on client and edge devices. Thanks to clear interfaces inside the components, extending or replacing a component with another without requiring substantial changes in the architecture is easy. For instance, the framework right now only supports Intel SGX for ExeS component to provide a TEE on edge devices. However, it is possible to change Intel SGX with another TEE implementation, such as AMD SEV-SNP or Intel TDX, by implementing a wrapper for ExeS to run inside the specific TEE. Besides the required wrapper on edge, the client also needs to support the remote attestation procedure of the TEE implementation to verify the integrity of the service running on the edge.

In addition to extending the framework with other TEE implementations, it also does not limit the platforms used on client and edge devices, as long as WAMR and a TEE supported by the framework support the edge platform. For example, Windows or Linux can be used on the edge device since Intel SGX supports both platforms. Moreover, the framework can be used on any client platform supported by LLVM and Rust's standard library, such as Windows, Linux, or macOS, with many different CPUs. Astonishingly, LLVM support enables utilizing the framework on a WebAssembly application with essential environmental supports, such as threading, in place. This results in a very portable application even after it starts adapting the Edge Computing paradigm with the help of a framework.

While the framework improves Edge Computing by making it an easy solution to utilize, it is also important to examine the benefits delivered to both the application and user in terms of performance and resource utilization. In order to assess the impacts of the framework on applications, a set of experiments on a benchmark suite called

polybench-rs [53], a Rust implementation of *PolyBench/C* [54], is carried out. *PolyBench/C* is a benchmark suite in which there are 30 different benchmarks related to numerical computations with different characteristics, such as differences in input sizes and computation requirements. It enables testing the framework on applications with different characteristics. Although the benchmark suite has 30 benchmarks, *gesummv* has failed to complete and has been excluded from results due to insufficient resources on the client device. Aside from the decision in the benchmark suite, the primary motivation behind using Rust implementation over its initial C implementation is the benefits of the ownership model of the Rust programming language.

Thanks to the Rust ownership model, in addition to references and borrowing rules, mutating a value via multiple different references is not possible. Besides, while having only a read access via a reference to a value, mutating it via another reference is also impossible. These benefits also continue impacting the IR side when an application written in Rust is compiled to LLVM IR, such as having `noalias` attribute on most of the pointer parameters. They improve the ability to perform static analysis to understand whether or not a function's pointer parameter can alias with another one. Rust is not the only language that can influence the aliasing of pointers; however, being default for Rust and enforced by Rust makes most of the applications provide these benefits.

Since the framework runs at both compile time and runtime stages, it will be examined separately for each stage. The following section first introduces the Edge Computing environment, where experiments are performed, by describing the devices and their properties in detail. Afterwards, compile-time experiments, which assess the performance of offloadable function identification and produced binary sizes, are explored. For more interesting results, the last section presents runtime experiments with improvements recorded in execution time and resource utilizations under different configurations.

5.1. Environment

To conduct the compile time and runtime experiments, an Edge Computing environment is constructed where the client and edge devices have a direct connection between them. In this environment, the client is represented by a Raspberry Pi 4 Model B device, whereas edge is an x86_64 desktop machine that hosts a more powerful CPU than the client's. Both devices are connected to each other over a wired gigabit ethernet connection to provide a stable connection during the experiments and minimize the variance. Hardware details of these two devices are displayed in Table 5.1.

Table 5.1. Details of client and edge hardwares.

	Client	Edge
CPU	4 core Cortex-A72@1.8 GHz	6 core AMD Ryzen 5600X@3.7 GHz
Memory	4GB LPDDR4@3200 MHz	32GB DDR4@3200 MHz
Network	1 Gigabit Ethernet	1 Gigabit Ethernet

Contrary to very different hardware details, both devices share similar software, excluding the ISA differences, in terms of OS and compilers they harness. However, this does not mean that using different software is impossible. They use Debian as a Linux distribution in addition to the same version of Rust, LLVM, and GCC compilers to compile respective components to their final forms. To start with the EdgeT component, this C++ project is compiled by clang compiler distributed inside the LLVM. Followed by that, ExeS, as a C project, is compiled by GCC's C compiler. Lastly, both *Bridge* and ComS components are compiled by rustc Rust compiler on client and edge devices, respectively. Exact versions of utilized software are displayed in Table 5.2.

Apart from the hardware and software details, the TEE, employed on the edge device while performing the experiments, also impacts the results. Since the current implementation of the framework provides support for Intel SGX as TEE, the edge device

runs the ExeS component inside a secure SGX enclave. However, in order to simplify the experimenting and remove specialized hardware requirements, SGX enclave is run in *Simulation* mode instead of *Hardware* mode. *Simulation* mode enables testing and experimenting without requiring supported hardware. While using *Simulation* mode can be considered as preventing realizing the impact of TEE on the execution time, it should also be noted that different TEE implementations can show different impacts on the execution. In fact, even the microchips belonging to the same manufacturer but different generations can exhibit different performance characteristics. For instance, the size of Enclave Page Cache (EPC) can dramatically affect the performance of an application running inside the Intel SGX enclave, as pointed out by AccTEE [43]. Additionally, the size of EPC can change between different processors. For this reason, the impact of *Simulation* and *Hardware* modes will be investigated in section 5.3.3.

Table 5.2. Details of client and edge softwares.

	Client	Edge
OS	Linux 6.1 - Debian 12 - arm64	Linux 6.1 - Debian 12 - x86_64
Rust	1.78.0	1.78.0
LLVM	18.1.2	18.1.2
GCC	12.2.0	12.2.0

To prepare the benchmarks for executing them in the client, each of them is first compiled to their respective LLVM IR form by utilizing the Rust compiler located on the client device. Following that, in order to apply transformations on the benchmarks and obtain a WebAssembly binary for the edge device, each of the produced IR file is given to the EdgeT component one by one. When the transformed IR file is obtained as the output of EdgeT component, it is compiled to object code with the help of the llc application located inside the LLVM toolchain. Afterward, the object code is converted to an executable using the system linker. For the edge part, the second IR file, produced additionally by the EdgeT component, is first compiled to WebAssembly object code by utilizing the same llc application. To obtain an appropriate WebAssembly binary,

the object code is then linked by wasm-ld, which LLVM also provides.

5.2. Compile Time Experiments

During the compilation of a benchmark, the framework runs EdgeT component on its LLVM IR form, which applies multiple passes and performs analysis and transformations. Inside these passes, *Analysis Pass* has the highest importance in terms of affecting the benchmark's overall performance since it decides whether a function is offloadable. Having as many functions identified as offloadable can provide greater offloading options for the framework to perform at runtime. Consequently, to assess the framework's performance at the compile time stage, the number of offloadable functions identified by the framework needs to be investigated. Because of how *polybench-rs* implemented, all benchmarks nearly utilize the same functions except their kernels where real computation happens. This results in having a similar analysis for all benchmarks. For this reason, Table 5.3 only displays one of the benchmark's analysis results.

Table 5.3. Offloadability analysis of benchmark's functions.

Reason	Offloadable	Count
Intrinsic	Yes	10
Pure	Yes	3
Only Declaration	No	6
Calling Not Offloadable Function	No	1
Accessing Mutable Global Var	No	2
Having Indirect Call	No	0
Having Target Specific Instruction	No	1
Having Variable Amount of Parameters	No	1
Having Unserializable Parameter	No	6
Diverging Execution	No	4

According to Table 5.3, there are 13 different functions suitable for offloading to the edge. However, 10 out of 13 functions are LLVM intrinsics, which are not transformed for offloading by EdgeT component since they are mostly replaced with optimized routines while compiling to target machine code by LLVM. On the other hand, the remaining three functions are appropriate for offloading since they are considered as *Pure* functions, which only perform computations without causing any side effect on the application state. As a matter of fact, these three functions are actually monomorphized versions of the same generic Rust function. Benchmarks call a generic kernel function three times with different generic parameters that result in the generation of three different functions inside the IR due to monomorphization.

Contrary to offloadable functions, 21 different functions are identified as not offloadable due to unsatisfied conditions of *Analysis Pass*. Table 5.3 displays how many functions have failed to satisfy a corresponding condition specified inside the section 4.3.1. For instance, six functions are not offloadable due to only having a declaration and lacking a body. Regarding possible improvements for analysis, not offloadable functions with *Having Unserializable Parameter* reason can be made offloadable by enhancing serialization of unsupported types, especially some pointers. However, they may still fail to satisfy other conditions.

The size of the produced WebAssembly binary that will be sent to the edge device to prepare the execution environment is an important metric for estimating the startup time of the edge device, from sending the binary to completing the initialization of the execution environment. Startup of components on the edge device depends on the upload time of binary and also compilation time spent in JIT compiler. Figure 5.1 displays binary sizes for each benchmark as broken down into their main contributors. The most notable contribution to the binary size is made by the functions identified as offloadable. In addition to offloadable functions, their corresponding wrapper functions also impact the size. Depending on the number of parameters, the contribution of a wrapper function to the final binary size can vary owing to deserialization and serialization instructions. As a last contributor, all binaries have a similar-sized data

section containing information about exported functions, global variables, and other WebAssembly related info.

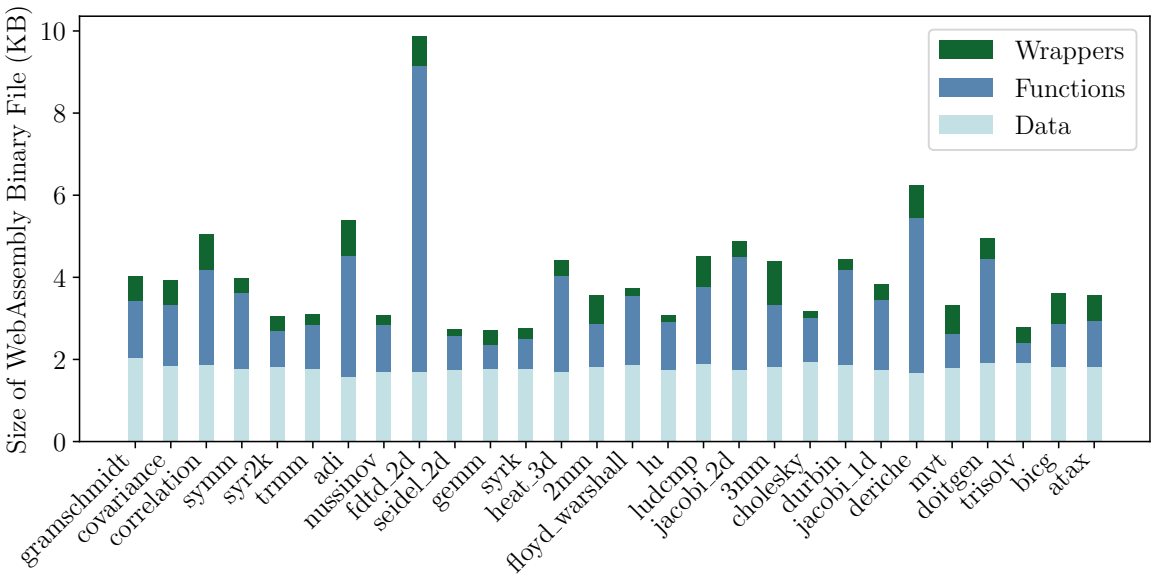

Figure 5.1. WebAssembly binary sizes of each benchmark.

Aside from the WebAssembly binary, sizes of transformed benchmark binaries did not change significantly compared to their original size due to the minimal amount of instructions inserted into the benchmark. Because of their insignificance, they are not shown separately.

5.3. Runtime Experiments

The primary goal of the framework is to improve the overall execution time of applications while reducing the required computation power expected from the device without compromising user privacy and degrading the user experience. As a result of this goal, the application's runtime is where the framework starts demonstrating the advantages of its transformations, applied at compile time. With the aim of analyzing and identifying the performance of the framework at runtime, all benchmarks are executed under three distinct configurations named *Local*, *Edge* and *Edge(NoEnc)*. Firstly, *Local* describes the configuration where all execution, including the offloadable func-

tions, happens at the client while excluding the edge completely. *Local* sets a baseline to compare with the framework's performance. Secondly, *Edge* defines the default behavior of the framework where execution of offloadable functions can happen either at the client or edge. Finally, *Edge(NoEnc)* configuration disables the encryption, which is done during client and edge communication.

Formally, to examine the performance at runtime, each benchmark is executed to completion, E_{ij}^k, for 5 times under different configurations, C^k, to obtain a time, T_{ij}^k, where $i \in \{1, ..., 5\}, j \in \{gramschmidt, ..., atax\}, k \in \{Local, Edge, Edge(NoEnc)\}$. Furthermore, during benchmark execution, offloading a function is decided with respect to a boolean cost function expressed as

$$c(n_{tx}) = n_{tx} < B \tag{5.1}$$

where n_{tx} indicates the number of bytes that will be sent from client to edge, and B is the size of the buffer allocated inside *Bridge* at startup. To obtain the overall execution time for a benchmark under a configuration, T_j^k, median of all its runs is taken

$$T_j^k = med(\{T_{1j}^k, ..., T_{5j}^k\}). \tag{5.2}$$

Inherently, E_j^k corresponds to the execution of T_j^k. To compare each configuration for each benchmark against C^{Local}, their speedup, S_j^k, is measured as

$$S_j^k = \frac{T_j^{Local}}{T_j^k}. \tag{5.3}$$

Visually, Figure 5.2 displays the S_j^k of each benchmark run under each configuration, sorted in a descending order. When the default behavior of the framework, C^{Edge}, is considered, the figure unveils that the framework helped 17 out of 29 benchmarks, from *gramschmidt* to *ludcmp*, reduce their execution time considerably. Besides that, there are five benchmarks, from *jacobi_2d* to *durbin*, without having been impacted noticeable by the framework. On the other hand, seven benchmarks, from *jacobi_1d* to *atax*, exhibit a slowdown in their execution time. Figure 5.2 also shows an important fact about the framework: the cost of encryption. Disabling the encryption improves speedup nearly across all benchmarks.

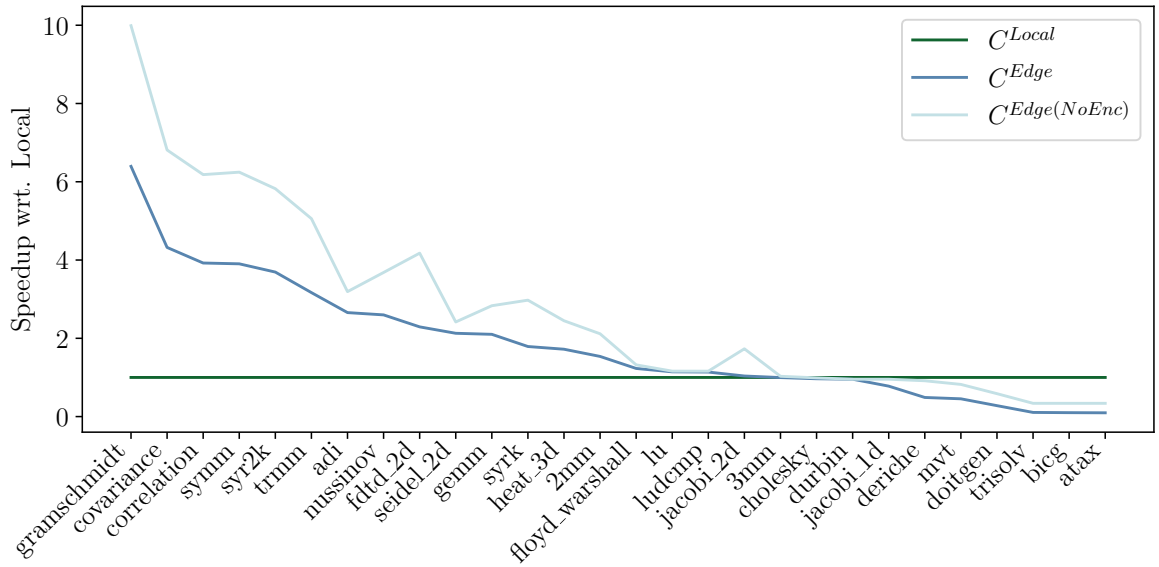

Figure 5.2. Speedup, S_j^k, of each benchmark run under each configuration.

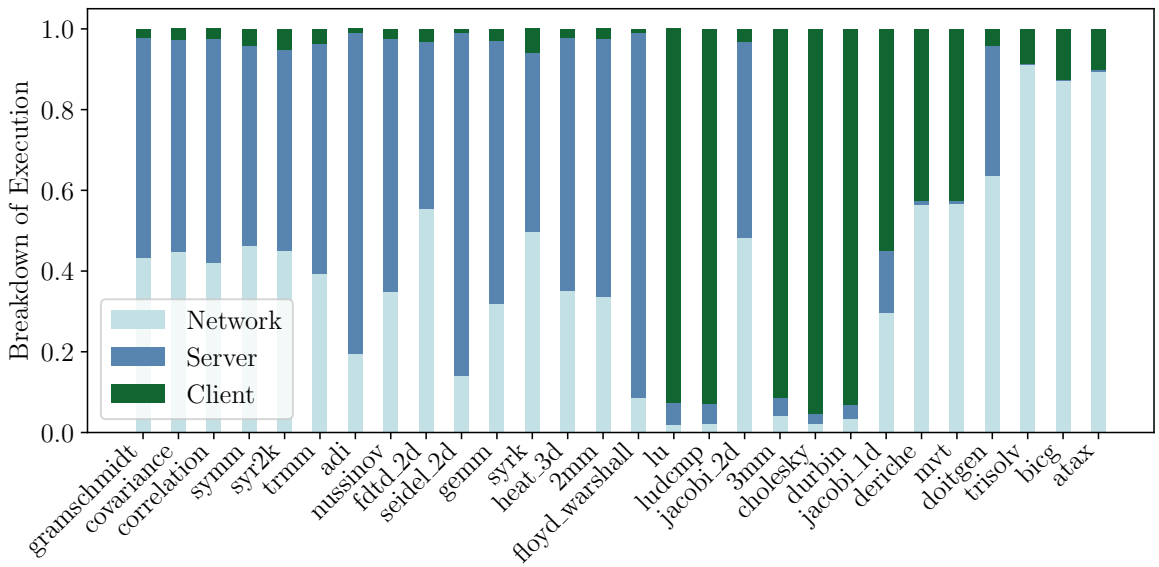

Figure 5.3. Execution time breakdown of each benchmark for C^{Edge}.

Understanding execution stages can provide greater insights into the speedups observed in Figure 5.2. For this reason, Figure 5.3 shows the breakdown of each T_j^{Edge} in terms of time spent in client, edge, and network. This figure clearly indicates that when most of the execution happens on edge, there is a considerable amount of speedup compared to C^{Local}. On the other hand, when there is no network activity

and execution on edge, C^{Edge} performs similarly to C^{Local}. However, slowdowns start happening as the network, which includes encryption and decryption, constitutes the most of execution. Moreover, the cost of encryption can also be verified with Figure 5.3. As both C^{Edge} and $C^{Edge(NoEnc)}$ get close to each other in Figure 5.2 such as for *adi* and *seidel_2d*, network part in Figure 5.3 also starts taking small part of overall execution, indicating less encryption and decryption is happening.

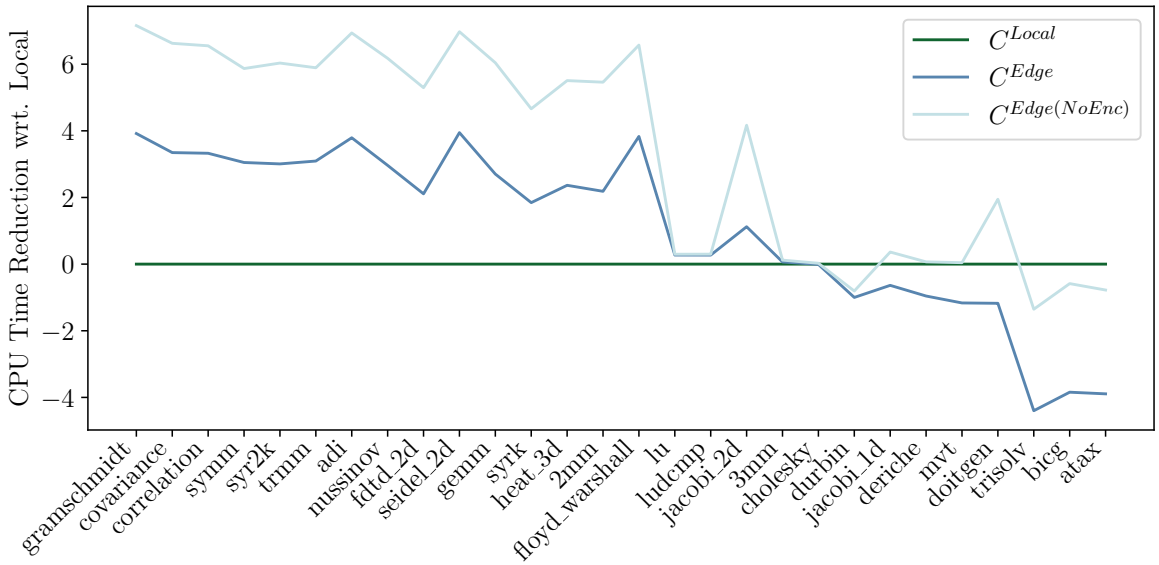

Figure 5.4. CPU time reduction, P_j^k, of each benchmark run under each configuration.

Apart from execution times, a metric in terms of resource usage that should be compared between configurations is the CPU utilization of the client device. Furthermore, to evaluate a benchmark's overall CPU utilization, the CPU time, U_j^k, spent utilizing one processor core can be measured. By reducing the CPU time consumed on the client device, earned CPU time can be spared for other tasks or even for sleeping to reduce the overall energy consumption. Similar to S_j^k, reduction in CPU time, P_j^k, of a configuration C^k compared to C^{Local} is measured as

$$P_j^k = \frac{U_j^{Local}}{U_j^k}. \tag{5.4}$$

Figure 5.4 presents P_j^k of each benchmark for each configuration in logarithmic scale by applying $\log_2(P_j^k)$. Compared to S_j^k where max value reaches 7 for C^{Edge}, P_j^k is able to reach a factor of 2^4. However, it also causes more CPU usage, up to 2^4 times,

when slowdowns start happening.

To summarize the overall performance of the framework and make a meaningful comparison between configurations, geometric means [55] of all benchmarks' execution times and CPU times for a configuration are displayed side by side in Figure 5.5. According to Figure 5.5, C^{Edge} and $C^{Edge(NoEnc)}$ have reduced the overall execution time by 15% and 44% across the 29 benchmarks, respectively. In terms of CPU time, this results in 51% and 90% reduction. Even though the framework causes slowdowns in some of the benchmarks, it accomplishes improving overall execution times of benchmarks in addition to providing significant CPU time savings.

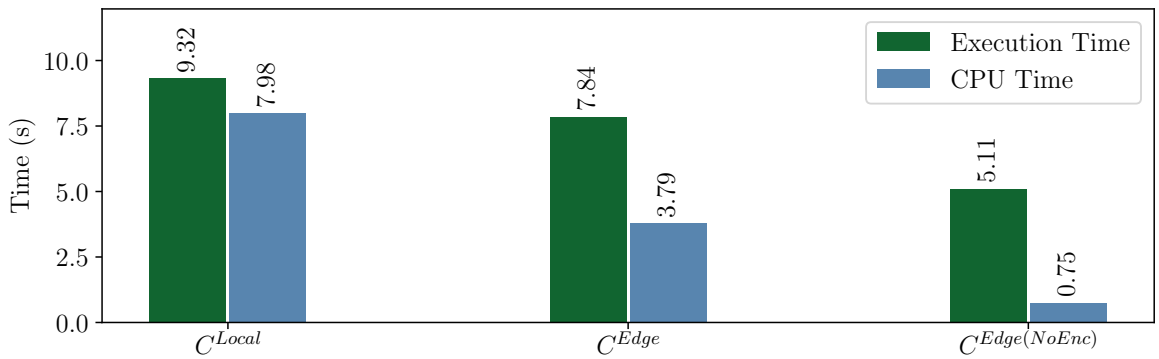

Figure 5.5. Geometric mean of execution times and CPU times.

5.3.1. Individual Execution Analysis

Beyond the overall CPU time, dissecting CPU utilization of specific execution of a benchmark is helpful in obtaining an idea about how and when resources are used. As an example, Figure 5.6 displays CPU utilization of $E^{Local}_{seidel\_2d}$, belonging to *seidel_2d* benchmark. To give a notice about figures highlighting CPU utilization, they display utilization in terms of percentage where the percentage scales with the number of cores that exist in the device's processor. For instance, when utilization hits 100%, it means that one core is fully utilized. Therefore, a device with four cores in its processor can have 400% as its maximum utilization. Since all execution only happens on the client device for $E^{Local}_{seidel\_2d}$, it nearly utilizes one core as expected by staying steady around

100%. However, this is not true when *seidel_2d* is inspected under C^{Edge}. When Figure 5.7 for $E^{Edge}_{seidel\_2d}$ is inspected, it can be noticed that the client device no more hits the 100% utilization constantly throughout the execution.

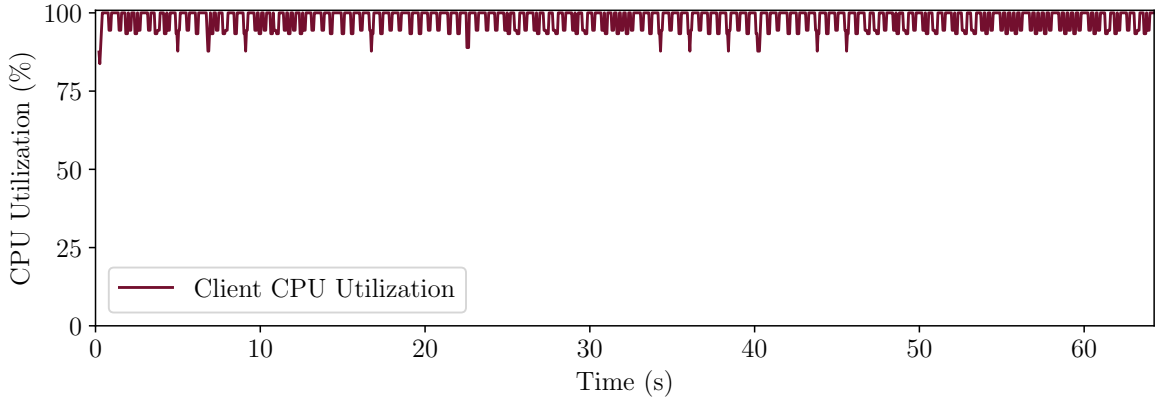

Figure 5.6. CPU utilization of $E^{Local}_{seidel\_2d}$.

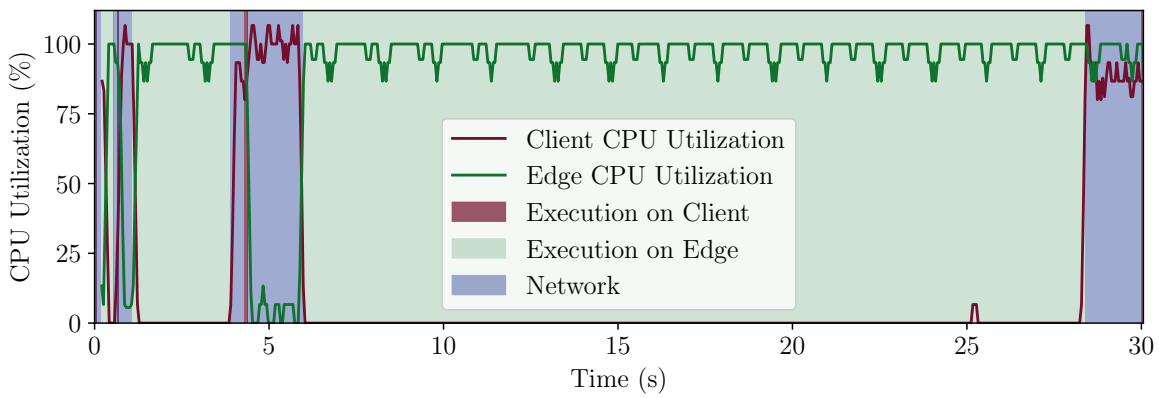

Figure 5.7. CPU utilization of $E^{Edge}_{seidel\_2d}$.

As an extension to the CPU utilization metric, Figure 5.7 carries additional information highlighting the stages of offloading when the edge device also starts playing a role. Since the execution can switch between client and edge devices, and both can spend their time communicating with each other, the background of the figure is painted with different colors depending on what happens at a specific time. To be more precise, the background is painted with a light red color when execution happens on the client, whereas it is painted with light green if the edge carries the execution.

Besides these two possible stages, when both devices communicate with each other, it is painted with a light blue color. In addition to the figure, Table 5.4 displays the exact duration of each stage for all of the offloaded functions of $E^{Edge}_{seidel\_2d}$.

Table 5.4. Offloaded functions of $E^{Edge}_{seidel\_2d}$.

#	ID	Start Time (s)	Execution Duration (s)	Network Duration (s)	Sent / Received Bytes (MB)
1	2	0.632	0.347	0.345	2 / 2
2	0	1.358	2.794	1.366	8 / 8
3	1	5.613	22.401	5.452	32 / 32

Both Figure 5.7 visually and Table 5.4 numerically show three functions offloaded to edge device for remote execution. Moreover, since all offloaded functions have different `ID`s, it also means that they belong to different functions. In Figure 5.7, up until the first offloading, the edge device exhibits very high CPU utilization while the client shows no utilization at all. High CPU utilization results from WebAssembly runtime initialization inside the edge device with the binary received from the client. Since the edge device is first initialized at the application startup by the client, compiling the received WebAssembly binary to native machine code with a JIT compiler causes high CPU utilization during the initialization. Excluding the initialization part, execution mostly happens on the edge device, other than sending and receiving messages, which can be deduced by tiny light red colored time slices in the figure.

Following the edge initialization, the client starts offloading the function with `ID 2` to the edge device around 0.632 seconds in the execution. Because first offloading took a small amount of time for both communication and execution, it may not be easy to derive a conclusion about utilization behaviors of devices under C^{Edge}. However, the other two offloaded functions share a pattern that happens during offloading of a function call to the edge device. First of all, the client shows moderate utilization while sending the offloading request, whereas the edge shows light utilization. Even

though they mostly perform I/O without much need on the CPU, encrypting and decrypting the messages still needs computation power. After altogether receiving the offloading request, edge starts the execution and consumes one core by hovering at 100% utilization. In the meantime, the client sits idle, which is observed by no utilization until the edge completes the execution and starts sending the result. Contrary to the client, which shows similar utilization while sending and receiving messages, edge exhibits much higher utilization while sending than receiving messages.

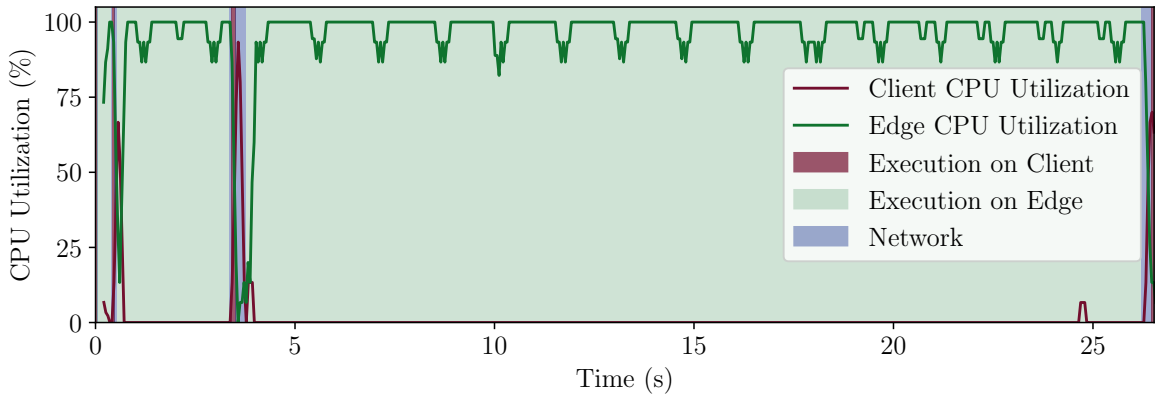

Figure 5.8. CPU utilization of $E^{Edge(NoEnc)}_{seidel\_2d}$.

For the purpose of identifying the cost of encryption, the same benchmark is run under $C^{Edge(NoEnc)}$. Figure 5.8 highlights that both client and edge devices exhibit much less CPU utilization than C^{Edge} when they communicate with each other, verifying that encryption has definitely a cost in terms of computation power requirement. This requirement also affects execution times significantly, as it is shown in Figure 5.5.

5.3.2. Framework Overhead on Edge

As a consequence of additional security measures taken at the edge, there is an expected performance loss when the execution of offloaded functions inside framework boundaries is compared to natively executing the same functions on bare metal. To identify the expected loss, another configuration, $C^{Edge(Native)}$, is introduced and compared to C^{Edge}. This configuration describes compiling the benchmark directly for the

edge device and executing them natively. Additionally, to compare only the execution of offloaded functions and not the whole benchmark, only the execution time of offloadable functions is included. Figure 5.9 displays speedup of $C^{Edge(Native)}$ against C^{Edge} with a formula similar to Equation (5.3). This figure clearly signifies the additional overhead introduced by the framework at the edge. All benchmarks except *deriche* have shown an improvement when executed natively. This situation can be the result of many reasons, such as missed optimizations by JIT compiler and overheads introduced by WebAssembly runtime as well as Intel SGX. It is also possible that the edge device's target code generator can vectorize many instructions and perform much better. Considering that geometric means of $C^{Edge(Native)}$ and C^{Edge} are 0.40 and 1.18 seconds, this investigation can be concluded that the framework has slowed down the execution of offloaded functions by exactly 2.95 times compared to their native executions.

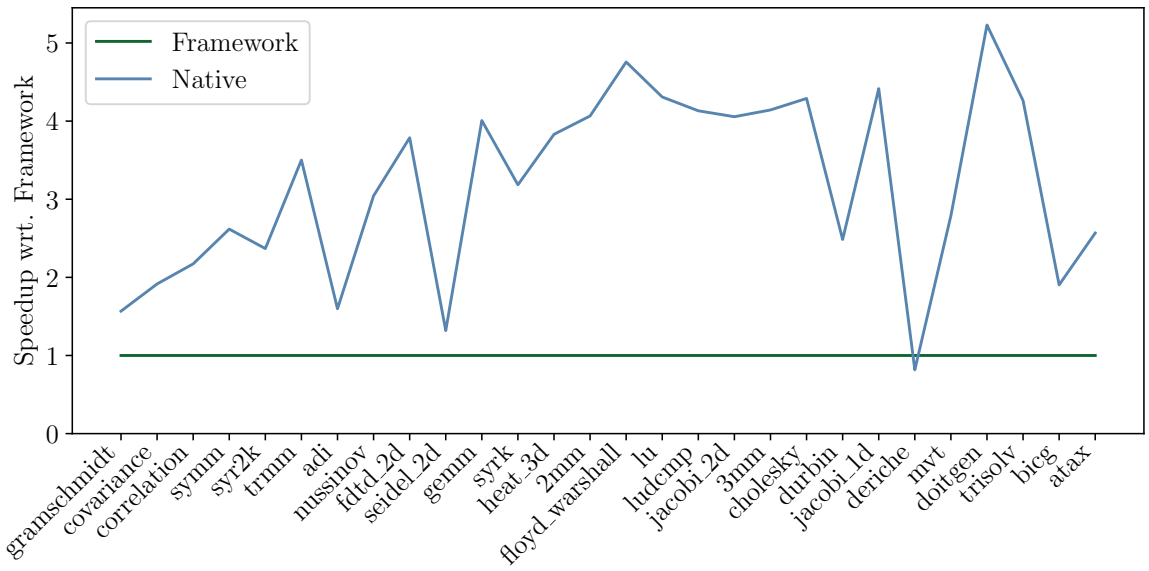

Figure 5.9. Native vs framework execution times.

5.3.3. Impact of Intel SGX Modes

To make experimenting straightforward and rule out the special hardware requirements, experiments are performed in an environment where *Simulation* mode of

Intel SGX is used instead of *Hardware* mode. Using *Simulation* mode has the possibility of preventing realizing the cost of a real TEE implementation, though different implementations can show different performance characteristics. In order to identify the performance difference between *Simulation* and *Hardware* modes, the same experiments are performed on hardware that is capable of running enclaves in *Hardware* mode, with 2 core Intel i7 6500U@2.50Ghz as CPU and 8GB DDR4@2133Mhz as memory. Regarding software details, the device shares the same software stack as the default edge device described in section 5.1. Additionally, both client and edge components are run on the same device to lift the possible network bottleneck, stress the device computationally, and emphasize the differences between modes. In other words, the client and edge are represented by the same device where they still talk with each other over TCP through the loopback interface of the OS.

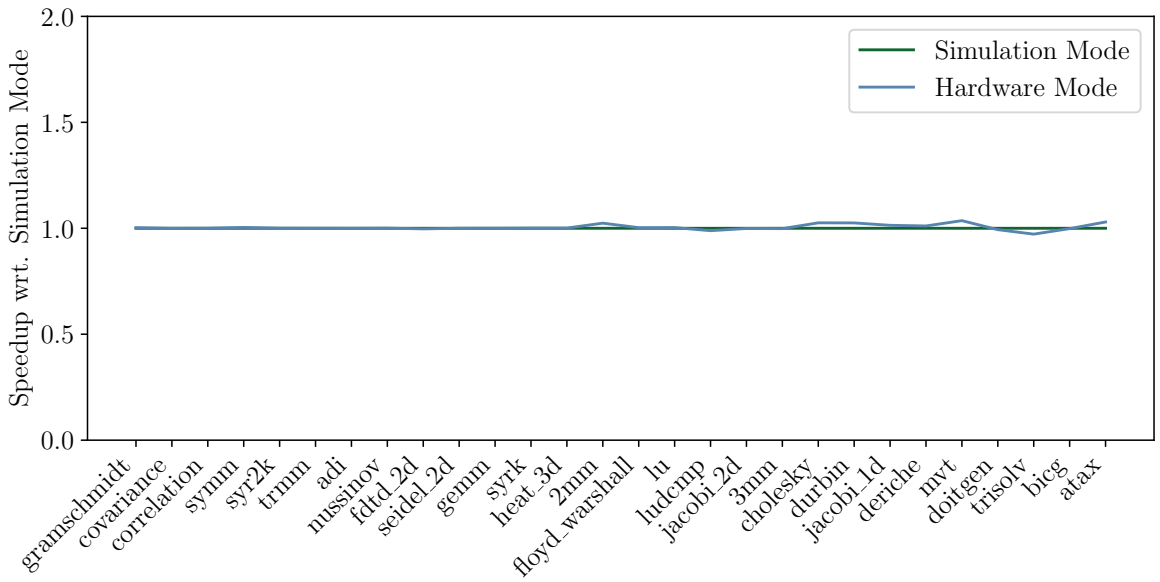

Figure 5.10. Intel SGX Hardware mode vs Simulation mode.

After conducting the experiments, Figure 5.10 reveals no significant gap between the two modes. Contrary to findings in AccTEE [43], where two modes have shown very significant differences, our framework did not experience the same difference. As mentioned in the AccTEE, the main reason for the difference between the two modes is due to EPC size. Since AccTEE utilizes Node.js [56] as its WebAssembly

runtime, which also hosts a JavaScript runtime, it is easy to pass EPC limit of hardware. However, our framework is able to stay below the limit, thanks to low memory usage of WAMR, and not face many cache misses.

5.3.4. Improved Cost Function

Although the framework can reduce the overall execution time of benchmarks when geometric means of T_j^{Edge} and T_j^{Local} are compared, some slowdowns may be prevented by improving the cost function $c(n_{tx})$. By embedding additional information into $c(n_{tx})$, such as estimating the possible execution time of the function on the client and edge as well as necessary communication time, functions that will not be executed faster when offloaded can be executed locally instead. For this reason, a new boolean cost function, $c_{new}(n_{tx}, n_{rx}, k, l, c)$, is developed and tested on the best and the worst performing benchmarks, namely *gramschmidth* and *atax*. Since c_{new} requires knowledge about functions' execution times with different parameters, which can be obtained by profiling the application, it requires further effort and hence not included in the main framework at the moment. The c_{new} consists of many other functions defined as

$$t_{local}(n_{tx}, k, l, m) = k(n_{tx})^l + m \tag{5.5}$$

$$t_{edge}(n_{tx}, n_{rx}, k, l, m) = \frac{n_{tx} + n_{rx}}{M} + \frac{t_{local}(n_{tx}, k, l, m)}{S} \tag{5.6}$$

$$c_{feasible}(n_{tx}, n_{rx}, k, l, m) = t_{edge}(n_{tx}, n_{rx}, k, l, m) < L * t_{local}(n_{tx}, n_{rx}, k, l, m) \tag{5.7}$$

$$c_{new}(n_{tx}, n_{rx}, k, l, m) = c(n_{tx}) \wedge c_{feasible}(n_{tx}, n_{rx}, k, l, m) \tag{5.8}$$

where $k, m \in R$ and $l, n_{tx}, n_{rx}, L, M, S \in N$. In terms of functions, t_{local} estimates the local execution time of the offloadable function, t_{edge} estimates the total time required for sending, receiving, and executing the offloadable function on edge, $c_{feasible}$ decides whether it will be executed faster on edge, and c_{new} extends old cost function with $c_{feasible}$.

To start with t_{local}, it tries to fit a simple polynomial curve with parameters k, l, and m over the execution times of function depending on the number of bytes, n_{tx},

that is passed to offloadable function. Whereas t_{edge} estimates the execution time of the function on edge by dividing t_{local} with a speedup variable S. It also considers the required communication time by dividing the number of bytes that will be sent and received with network bandwidth M. After calculating estimated execution times, $c_{feasible}$ checks whether t_{edge} is smaller than t_{local} by a factor L. In this case, L is helpful to eliminate estimation-related errors by ensuring that t_{edge} is definitely smaller than t_{local} by a large margin. Lastly, c_{new} checks whether both conditions are satisfied and decides offloadability.

Table 5.5. Execution time of kernels with different parameters.

Benchmark	n_{tx} (MB)	n_{rx} (MB)	Local Exec. Dur. (s)	Edge Exec. Dur. (s)	Network Dur. (s)
atax	1.92	0.008	0.0011	0.0007	0.132
	7.65	0.016	0.0045	0.0026	0.368
	30.48	0.032	0.0158	0.0099	1.46
gramschmidt	1.83	1.83	0.131	0.056	0.208
	7.32	7.32	2.487	0.457	0.785
	29.3	29.3	55.99	4.495	3.135

As each benchmark uses its own unique kernel and calls it three times with parameters sized differently, each call's execution time can be used as a datapoint for fitting a curve. Table 5.5 specifies the obtained execution time of three calls for both two kernel functions as well as their size of input and output parameters, n_{tx} and n_{rx}. The table also reveals why *atax* performs poorly on edge. Even though execution time on the edge is much better than local, time spent on the network makes it worse. When a polynomial regression is applied to both benchmarks' data points, corresponding values for parameters k, l, and m are obtained and displayed in Table 5.6. In addition to parameters, a reasonable value for L is chosen to take possible estimation errors into account. M's value corresponds to 1Gbit network speed, which

corresponds to a 125 MB per second transfer rate. Finally, the value of S is selected according to the geometric mean of each offloadable function's speedup on edge against their local execution.

Table 5.6. Parameters of c_{new} for each benchmark.

Benchmark	k	l	m	L	M	S
atax	4.85e-10	1	3.36e-4	0.8	125e6	4
gramschmidt	6.64e-14	2	3.25e-1	0.8	125e6	4

After obtaining values of c_{new} for each benchmark, their t_{local}, t_{edge} and $c_{feasible}$ for each call, in addition to the old cost function c, are calculated and presented in Table 5.7. According to these results, $c_{feasible}$ has prevented offloading of all *atax* calls, which takes longer to execute on edge, while still allowing *gramschmidt* to continue offloading. In terms of estimated times, t_{local} of *atax* is able to match its real values owing to being a linear function. However, t_{local} of *gramschmidt* exceeds its real value because of lacking components with less order in the polynomial curve. On the other hand, t_{edge} values of both benchmarks do not match their real values because of the simplicity of t_{edge}. For instance, it does not consider encryption costs, which affects network time.

Table 5.7. Estimated execution time and feasibility of each function call.

Benchmark	n_{tx} (MB)	n_{rx} (MB)	t_{local}	t_{edge}	$c_{feasible}$	c
atax	1.92	0.008	0.0013	0.0165	False	True
	7.65	0.016	0.0042	0.0653	False	True
	30.48	0.032	0.0158	0.260	False	True
gramschmidt	1.83	1.83	0.570	0.173	True	True
	7.32	7.32	4.243	1.184	True	True
	29.3	29.3	63.02	16.24	True	True

To conclude, when the improved cost funciton is compared to old one, it makes more intelligent decisions about whether the current function call should be offloaded to the edge or not. It considers additional parameters about computation speed and also network speed. Since it requires profiling data about offloadable functions' execution time beforehand and the framework does not obtain them automatically, the improved cost function is not included in the framework's default implementation and left as possible future work. However, it proves that the framework can perform much better than its current performance when it can utilize profiling tools to obtain necessary information.

6. CONCLUSION

Edge Computing offers solutions to many impractical problems that are not feasible on client devices by providing a powerful computation environment located near clients. However, it also comes with its problems besides its solution. A significant problem is the privacy issue that arises when user data is moved from clients through a channel to edge devices. As the data moves outside the client, it is susceptible to observations of unauthorized entities. Additionally, the kind of computation applied to the user data can also be observed. In fact, it may not be possible to verify that the computation result is legit and not altered. Because of the lack of trust in the outer world, many solutions embracing Edge Computing solution can become unsuitable in terms of securing user privacy and ensuring the integrity of the computation.

Embracing Edge Computing as a solution might also not be practical due to the required additional development effort to bring it into existing applications. Even designing applications from scratch with the existence of a powerful edge device in mind can become challenging owing to the necessity of integrating fault tolerant RPC like mechanisms and developing a remote part of the application. Adjusting existing applications and developing new ones to utilize edge devices and enhance their capabilities can become a demanding process.

Apart from the development effort required on the application, managing and supporting a diverse range of edge devices to run the remote part of the application poses a challenge. An application that mainly targets mobile devices must consider possible edge devices that can be encountered due to frequent location changes. It has to know how to initialize them and execute their remote parts because of differences in their hardware and software details. Conversely, this also limits the list of edge devices an IP can provide in a backward compatible manner to not break the applications.

All the aforementioned problems and challenges can be lifted by introducing advancements in other concepts, tools, and technologies to Edge Computing. For instance, proposing Confidential Computing to provide a TEE on the edge devices can help establish a trust relationship between the clients and IPs in terms of the confidentiality and integrity of both user data and computation. Moreover, developing a tool that is able to transform an application received in a supported form into another one capable of utilizing an edge device can alleviate the additional development cost. To hide hardware and software details of different edge devices, a unified execution environment with a fixed set of details for clients can be designed. Incorporating all these enhancements under a framework can make Edge Computing a more practical solution for all supported applications.

In this regard, the thesis proposed PrivateEdge as a framework consisting of multiple components, each with different responsibilities at compile time or runtime of applications. By utilizing the LLVM project and its tools at compile time, it analyzes applications in the LLVM IR form to identify tasks suitable for offloading to an edge device. Following the result obtained at the analysis step, it transforms identified tasks to be executed either at the client or edge decided at runtime. As a result of the analysis and transformation, the framework can reduce the development cost by automating application transformation and providing a custom RPC mechanism. In order to provide a unified execution environment independent from the hardware and software details of edge devices, it uses WebAssembly technology, which delivers a virtual machine that aims to be portable, safe, and fast as near its native speed. At compile time, in addition to transforming the application, it generates a WebAssembly binary containing all the offloadable tasks distributed with the application itself. As complementary to the binary, it hosts a WebAssembly execution environment, powered by WAMR, on the edge device. This environment is initialized with the binary when the client establishes a connection and sends it to the edge device. Lastly, to mitigate the trust issue against the edge device and its IP, and consequently to preserve the confidentiality of user data, the framework runs the WebAssembly execution environment inside a TEE whose integrity is verified by performing remote attestation when

the client successfully connects. The connection between client and edge devices is also end-to-end encrypted with TLS. Currently, the framework only supports Intel SGX to provide TEE on the edge device. However, the framework's architecture is very modular, making it easy to extend with different Confidential Computing technologies.

The first implementation of PrivateEdge is evaluated with the *polybench-rs* benchmark suite to assess its impact on applications at compile time and runtime. By running the compile time stage of the framework on each benchmark in the suite, it is shown that PrivateEdge can automatically identify a subset of the benchmarks' tasks and successfully transform them to make them offloadable to the edge device. Additionally, to understand the framework's effect on the benchmarks' performance, their transformed forms are run under different configurations inside an edge computing environment. When the total execution times of 29 benchmarks' transformed forms are compared to their original form, significant speedups are observed in 17 benchmarks and some noticeable slowdowns in seven benchmarks. For the five benchmarks, the framework did not cause major execution time differences. After many executions across the 29 benchmarks, the framework reduced overall execution times by 15% in its default configuration thanks to using a powerful edge device while preserving the confidentiality of user data. An improved cost function is defined as an additional bonus for the framework, which prevents many unfeasible offloadings from happening and results in much better performance.

6.1. Challenges

Developing a framework that combines different concepts, tools, and technologies to lift some of the problems encountered in Edge Computing is demanding. The addition of any new concept brings its own set of problems, complicates overall behavior, and limits possibilities. In this journey, a few challenges are encountered while implementing the framework. Firstly, due to the inability to interact with the environment freely, restrictions imposed by Intel SGX make it impossible for some libraries to run inside the enclave without applying any modifications. For instance, using the

upstream version of OpenSSL, a popular cryptography toolkit that enables TLS, inside the TEE is not possible. Instead, Intel provides its patched version [50], which only makes OpenSSL usable inside SGX environment.

The second challenge was designing the overall architecture of the framework. From compile time to runtime, as well as from client to edge devices, the framework needs multiple components with clear interfaces to interact with each other successfully. Furthermore, their boundaries must be well defined to ensure that the confidentiality of user data is held at all stages of application execution. For example, any operation related to user data on the edge device must be performed inside the Execution Server component that resides inside the TEE. As is the case in most of the developments, this part has evolved iteratively until it found its final form. However, architecture is still open to additional improvements.

Another challenge encountered in the compile time was ensuring the correctness of transformations applied to the application and the correctness of the generated binary. The correctness in this context means without causing any observable behavior changes on the application compared to its original form. For example, serializing two pointer parameters without having `noalias` attribute can alter application behavior due to the possibility of pointing to the exact same memory location. While any write operation performed from the first pointer will be visible from the second pointer in the client, this behavior is not true anymore on edge since they point to very different memory locations. Even though this problem can be resolved by calculating aliased regions at runtime during offloading, it significantly increases the complexity of offloading analysis and serialization.

6.2. Future Work

The current implementation of the framework is able to provide significant benefits in terms of easing the adoption of Edge Computing. However, there are still plenty of possible improvements that can push the framework further ahead in terms

of its benefits and features. As its architecture is very portable across different kinds of devices with different CPUs, one possible future work is extending the framework's range of supported TEEs by adding implementation for AMD SEV-SNP and Intel TDX. In addition to manufacturer-specific TEEs, this support can be extended to cloud provider-specific solutions such as AWS Nitro Enclave. With broad support for different TEEs, an IP can utilize a different kind of device to provide service to its clients.

The current behavior of function analysis imposes stringent rules for a function to be offloadable, which results in very few of them being identified as offloadable. Some of these rules keep the offloading procedure simple, while others are necessary for the transformations not to cause behavioral changes. One example of such a strict rule to reduce complexity is the serialization of pointer parameters. Right now, the framework is not able to serialize pointers whose size cannot be determined statically at compile time. Even though the bound of a pointer parameter can be calculated at runtime, this prevents functions that accept dynamically sized values stored behind the pointers, such as array-like data types, from being identified as offloadable. To give an example, let us consider a compiler that may choose to generate a code for passing array data types to a function by passing two parameters. These parameters represent a pointer and an integer to indicate both the start address of the array and its length. This relation can be established at compile time and calculated at runtime by performing additional analysis on the function and its parameters. In addition to dynamically calculating the size of pointer parameters, nested pointer parameters containing other pointers can also be serialized by additional analysis.

Another notable future work is about increasing the throughput of multithreaded applications. Due to the limitation of the framework, offloading tasks concurrently is impossible. For multithreaded applications, this may result in offloading only from one thread at a time, while others continue their execution locally until ongoing offloading is completed. This improvement needs concurrent execution support from both the client and edge sides. Contrary to its benefit, it can require substantial changes in how

offloading is performed in runtime.

The results obtained in Chapter 5 showed that some of the benchmarks spend considerable time in the network to send and receive data. Moreover, the cost of end-to-end encryption in computation time can become significant. For the purpose of reducing the time spent in the network and the encryption, data can be compressed before encrypting and decompressed after decrypting. As the compression can reduce overall execution time in Edge Computing scenarios [57], it can be helpful for improving the overall performance of applications. Even though compression has its own computation cost, but it can be neutralized or even surpassed by the reduction in network and encryption time.

REFERENCES

1. Cao, K., Y. Liu, G. Meng and Q. Sun, "An Overview on Edge Computing Research", *IEEE Access*, Vol. 8, pp. 85714–85728, 2020.

2. Masoudi, M. and C. Cavdar, "Device vs Edge Computing for Mobile Services: Delay-Aware Decision Making to Minimize Power Consumption", *IEEE Transactions on Mobile Computing*, Vol. 20, No. 12, pp. 3324–3337, 2021.

3. Shi, W., J. Cao, Q. Zhang, Y. Li and L. Xu, "Edge Computing: Vision and Challenges", *IEEE Internet of Things Journal*, Vol. 3, No. 5, pp. 637–646, 2016.

4. Abbas, N., Y. Zhang, A. Taherkordi and T. Skeie, "Mobile Edge Computing: A Survey", *IEEE Internet of Things Journal*, Vol. 5, No. 1, pp. 450–465, 2018.

5. Ranaweera, P., A. D. Jurcut and M. Liyanage, "Survey on Multi-Access Edge Computing Security and Privacy", *IEEE Communications Surveys & Tutorials*, Vol. 23, No. 2, pp. 1078–1124, 2021.

6. Xiao, Y., Y. Jia, C. Liu, X. Cheng, J. Yu and W. Lv, "Edge Computing Security: State of the Art and Challenges", *Proceedings of the IEEE*, Vol. 107, No. 8, pp. 1608–1631, 2019.

7. Zeyu, H., X. Geming, W. Zhaohang and Y. Sen, "Survey on Edge Computing Security", *International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering*, Fuzhou, China, pp. 96–105, 2020.

8. Zhang, J., B. Chen, Y. Zhao, X. Cheng and F. Hu, "Data Security and Privacy-Preserving in Edge Computing Paradigm: Survey and Open Issues", *IEEE Access*, Vol. 6, pp. 18209–18237, 2018.

9. Feng, D., Y. Qin, W. Feng, W. Li, K. Shang and H. Ma, "Survey of Research

on Confidential Computing", *IET Communications*, Vol. 18, No. 9, pp. 535–556, 2024.

10. Arm Limited, "Arm TrustZone for Cortex-A", `https://www.arm.com/technologies/trustzone-for-cortex-a`, accessed on December 30, 2024.

11. "Intel® Software Guard Extensions Developer Guide", Intel Corporation, 2018.

12. Mulligan, D. P., G. Petri, N. Spinale, G. Stockwell and H. J. M. Vincent, "Confidential Computing—A Brave New World", *International Symposium on Secure and Private Execution Environment Design*, Washington, DC, USA, pp. 132–138, 2021.

13. Sahita, R., V. Shanbhogue, A. Bresticker, A. Khare, A. Patra, S. Ortiz, D. Reid and R. Kanwal, "CoVE: Towards Confidential Computing on RISC-V Platforms", *Proceedings of the 20th ACM International Conference on Computing Frontiers*, Bologna, Italy, p. 315–321, 2023.

14. Dworkin, M. J., E. Barker, J. Nechvatal, J. Foti, L. E. Bassham, E. Roback and J. D. Jr., "Advanced Encryption Standard", National Institute of Standards and Technology, 2001.

15. Gentry, C., *A Fully Homomorphic Encryption Scheme*, Ph.D. Thesis, Stanford University, 2009.

16. Costan, V. and S. Devadas, "Intel SGX Explained", *International Association for Cryptologic Research Cryptology ePrint Archive*, Vol. 2016, p. 86, 2016.

17. "AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More", Advanced Micro Devices Incorporated, 2020.

18. Hex Five Labs, "MultiZone Security TEE for RISC-V Processors", `https://github.com/hex-five/multizone-sdk`, accessed on December 18, 2024.

19. Ozga, W., "Towards a Formally Verified Security Monitor for VM-based Confidential Computing", *Proceedings of the 12th International Workshop on Hardware and Architectural Support for Security and Privacy*, Toronto, Canada, p. 73–81, 2023.

20. Lee, D., D. Kohlbrenner, S. Shinde, K. Asanović and D. Song, "Keystone: An Open Framework for Architecting Trusted Execution Environments", *Proceedings of the 15th European Conference on Computer Systems*, Heraklion, Greece, pp. 1–16, 2020.

21. Apsey, E., P. Rogers, M. O'Connor and R. Nertney, "Confidential Computing on NVIDIA H100 GPUs for Secure and Trustworthy AI", `https://developer.nvidia.com/blog/confidential-computing-on-h100-gpus-for-secure-and-trustworthy-ai/`, accessed on December 18, 2024.

22. "NVIDIA H100 Tensor Core GPU Architecture", NVIDIA Corporation, 2023.

23. Lattner, C. and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation", *International Symposium on Code Generation and Optimization*, San Jose, CA, USA, pp. 75–86, 2004.

24. Emscripten Contributors, "A Complete Compiler Toolchain to WebAssembly", `https://emscripten.org/`, accessed on December 18, 2024.

25. World Wide Web Consortium, "WebAssembly Security", `https://webassembly.org/docs/security/`, accessed on December 18, 2024.

26. Bytecode Alliance Contributors, "WebAssembly Micro Runtime", `https://github.com/bytecodealliance/wasm-micro-runtime`, accessed on December 18, 2024.

27. Cuervo, E., A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra and P. Bahl, "MAUI: Making Smartphones Last Longer with Code Offload", *Pro-*

ceedings of the 8th International Conference on Mobile Systems, Applications, and Services, San Francisco, California, USA, p. 49–62, 2010.

28. Hunt, G. C. and M. L. Scott, "The Coign Automatic Distributed Partitioning System", *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, LA, p. 187–200, 1999.

29. Kremer, U., J. Hicks and J. Rehg, "A Compilation Framework for Power and Energy Management on Mobile Computers", *Languages and Compilers for Parallel Computing*, Berlin, Heidelberg, pp. 115–131, 2003.

30. Lee, G., H. Park, S. Heo, K.-A. Chang, H. Lee and H. Kim, "Architecture-Aware Automatic Computation Offload for Native Applications", *48th Annual IEEE/ACM International Symposium on Microarchitecture*, Waikiki, HI, USA, pp. 521–532, 2015.

31. Brumley, D. and D. Song, "Privtrans: Automatically Partitioning Programs for Privilege Separation", *13th USENIX Security Symposium*, San Diego, CA, p. 5, 2004.

32. Liu, S., G. Tan and T. Jaeger, "PtrSplit: Supporting General Pointers in Automatic Program Partitioning", *Proceedings of ACM SIGSAC Conference on Computer and Communications Security*, Dallas, Texas, USA, p. 2359–2371, 2017.

33. Qiang, W. and H. Luo, "AutoSlicer: Automatic Program Partitioning for Securing Sensitive Data Based-on Data Dependency Analysis and Code Refactoring", *IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, Wuhan, China, pp. 239–247, 2022.

34. Winderix, H., *Security Enhanced LLVM*, M.S. Thesis, KU Leuven, 2018.

35. Arnautov, S., B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers,

R. Kapitza, P. Pietzuch and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX", *12th USENIX Symposium on Operating Systems Design and Implementation*, Savannah, GA, pp. 689–703, 2016.

36. Chen, H., H. H. Chen, M. Sun, K. Li, Z. Chen and X. Wang, "A Verified Confidential Computing as a Service Framework for Privacy Preservation", *32nd USENIX Security Symposium*, Anaheim, CA, pp. 4733–4750, 2023.

37. Lee, H., S. Lee, Y. C. Lee, H. Han and S. Kang, "iEdge: An IoT-assisted Edge Computing Framework", *IEEE International Conference on Pervasive Computing and Communications*, Kassel, Germany, pp. 1–8, 2021.

38. Lind, J., C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer and P. Pietzuch, "Glamdring: Automatic Application Partitioning for Intel SGX", *USENIX Annual Technical Conference*, Santa Clara, CA, pp. 285–298, 2017.

39. Zobaed, S. and M. A. Salehi, "Confidential Computing across Edge-to-Cloud for Machine Learning: A Survey Study", ArXiv:2307.16447 [cs], 2023.

40. Merkel, D., "Docker: Lightweight Linux Containers for Consistent Development and Deployment", *Linux Journal*, Vol. 2014, p. 2, 2014.

41. World Wide Web Consortium, "WebAssembly Core Specification", `https://www.w3.org/TR/wasm-core-2/`, accessed on December 18, 2024.

42. Gadepalli, P. K., S. McBride, G. Peach, L. Cherkasova and G. Parmer, "Sledge: A Serverless-first, Light-weight Wasm Runtime for the Edge", *Proceedings of the 21st International Middleware Conference*, Delft, Netherlands, p. 265–279, 2020.

43. Goltzsche, D., M. Nieke, T. Knauth and R. Kapitza, "AccTEE: A WebAssembly-based Two-way Sandbox for Trusted Resource Accounting", *Proceedings of the 20th International Middleware Conference*, Davis, CA, USA, p. 123–135, 2019.

44. Nieke, M., L. Almstedt and R. Kapitza, "Edgedancer: Secure Mobile WebAssembly Services on the Edge", *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, Edinburgh, UK, p. 13–18, 2021.

45. Ménétrey, J., M. Pasin, P. Felber and V. Schiavoni, "Twine: An Embedded Trusted Runtime for WebAssembly", *IEEE 37th International Conference on Data Engineering*, Chania, Greece, pp. 205–216, 2021.

46. Rescorla, E., "The Transport Layer Security Protocol Version 1.3", RFC 8446, 2018, `https://www.rfc-editor.org/info/rfc8446`, accessed on December 30, 2024.

47. Götzfried, J., M. Eckert, S. Schinzel and T. Müller, "Cache Attacks on Intel SGX", *Proceedings of the 10th European Workshop on Systems Security*, Belgrade, Serbia, pp. 1–6, 2017.

48. Wang, J., Y. Cheng, Q. Li and Y. Jiang, "Interface-Based Side Channel Attack Against Intel SGX", ArXiv:1811.05378 [cs], 2018.

49. Wang, W., M. Li, Y. Zhang and Z. Lin, "PwrLeak: Exploiting Power Reporting Interface for Side-Channel Attacks on AMD SEV", *Detection of Intrusions and Malware, and Vulnerability Assessment: 20th International Conference*, Berlin, Heidelberg, p. 46–66, 2023.

50. Intel Corporation, "Intel® Software Guard Extensions SSL", `https://github.com/intel/intel-sgx-ssl`, accessed on December 18, 2024.

51. LLVM, "LLVM Language Reference Manual", `https://llvm.org/docs/LangRef.html`, accessed on December 18, 2024.

52. Knauth, T., M. Steiner, S. Chakrabarti, L. Lei, C. Xing and M. Vij, "Integrating Remote Attestation with Transport Layer Security", ArXiv:1801.05863 [cs], 2019.

53. Ferrer, J. R., "Rust Port of PolyBench Benchmark Suite", `https://github.com/JRF63/polybench-rs`, accessed on December 18, 2024.

54. Pouchet, L.-N. and et all, "PolyBench Benchmark Suite", `https://sourceforge.net/projects/polybench/`, accessed on December 18, 2024.

55. Touati, S., J. Worms and S. Briais, "The Speedup-Test: A Statistical Methodology for Program Speedup Analysis and Computation", *Concurrency and Computation: Practice and Experience*, Vol. 25, No. 10, pp. 1410–1426, 2013.

56. OpenJS Foundation, "Node.js — Run JavaScript Everywhere", `https://nodejs.org/`, accessed on December 18, 2024.

57. Kurt, F. M. and B. A. Özgövde, "Edge Computing for Computer Games by Offloading Physics Computation", *Gazi University Journal of Science Part A: Engineering and Innovation*, Vol. 10, No. 3, p. 310–326, 2023.