Enclave Host Interface for Security

Anmol Sinha

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 29.7.2022

Supervisor

Adj. Prof. Jan-Erik Ekberg, Aalto University
Prof. Ben Slimane, KTH Royal Institute of Technology

Advisor

Antti Rusanen

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Anmol Sinha

Title Enclave Host Interface for Security

Degree programme Security and Cloud Computing

Major Security and Cloud Computing **Code of major** SCI3113

Supervisor Adj. Prof. Jan-Erik Ekberg, Aalto University
Prof. Ben Slimane, KTH Royal Institute of Technology

Advisor Antti Rusanen

Date 29.7.2022 **Number of pages** 52 **Language** English

Abstract

Secure enclave technology has during the last decade emerged as an important hardware security primitive in server computer cores, and increasingly also in chips intended for consumer devices like mobile phones and PCs. The Linux Confidential Compute Consortium has taken a leading role in defining the host APIs for enclave access (e.g. OpenEnclave APIs). Earlier solutions for security isolation in mobile phones relied on so called Trusted Execution Environments, which are similar in hardware isolation, but serve primarily OEM device security use-cases, and the environments are access controlled by remote trust roots (code signatures).

This thesis examines the security requirements for enclaves, visible through APIs and SDKs. An augmented IDE / SDK interface that accounts for security, including legacy considerations present with TEEs is also proposed. This thesis also attempts to improve developer experience related to development of trusted application by providing a tight integration with IDE and an expressive way to select methods which can be carved out of an existing rust application into a seperate trusted application. Furthermore, this thesis also discusses some common pitfalls while developing code for trusted applications and attempts to mitigate several of the discussed risks.

The work plan includes a background study on existing TEE and enclave SDKs, a novel SDK augmentation that accounts for the features listed above, and a prototype implementation that highlights the enclave security needs beyond mere isolated execution. An IDE plugin is also implemented, that exemplifies how software engineers (with potentially limited security knowledge) can implement a trusted application service with enclave support such that the end result (enclave code) will run without information leakage or interface security problems.

Keywords Trusted Execution Environment, Enclave, Trusted Applications, SDK, Visual Studio Code

Preface

This paper is written with experiences learnt after working on a project in collaboration with Huawei Technologies.

I want to thank my supervisor Jan-Erik Ekberg for his extremely deep and broad knowledge about Trusted Execution Environments and Enclaves.

I am also like to grateful my Examiner at KTH Royal Institute of Technology - Ben Slimane for his guidance and help in navigating and writing an academic thesis.

I would also like to thank my advisor - Antti Rusanen, whom I owe the most. His intense cooperation was throughout the entire semester was impetus for shaping this thesis. He guided me against all of my engineering challenges and answered my relentless questions so that I can reach a greater level of understanding.

I would also like the rest of the team at Huawei Technologies for their constant guidance.

Espoo, 29.7.2022

Anmol Sinha

Contents

Abstract **3**

Preface **4**

Contents **5**

Symbols and abbreviations **7**

1 Introduction **9**

2 Background **11**
2.1 JavaScript . . . 11
2.2 TypeScript . . . 11
2.3 Visual Studio Code . . . 12
2.4 Parsing . . . 12
2.5 Abstract Syntax Trees . . . 13
2.6 Grammars . . . 14
2.7 Interpretations of Context-Free Grammars . . . 15
2.7.1 Chomsky Normal Form . . . 15
2.7.2 Greibach Normal Form . . . 15
2.7.3 Backus Naur Form and Regular Expressions . . . 16
2.7.4 Parsing Expression Grammar . . . 17
2.8 ANTLR and other parser generator tools . . . 17
2.9 Trusted Execution Environments . . . 18
2.10 Trusted Applications . . . 18
2.11 Enclaves . . . 19
2.12 Difficulty with creating Trusted Applications . . . 20
2.13 Rust . . . 20

3 Research Question and Method **22**
3.1 Problem Definition . . . 22
3.2 Research Questions . . . 22
3.3 Research Method . . . 22

4 Implementation **23**
4.1 Visual Studio Code . . . 23
4.2 Code structure . . . 24
4.3 Code Scaffolding . . . 24
4.4 Handlebars . . . 25
4.5 Parsing and Tokenization . . . 26
4.6 Parser Combinators . . . 27
4.7 Abstract Syntax Trees . . . 28
4.8 Code Analysis . . . 29
4.9 Code Generation . . . 30

4.10 Code Transformations . . . 32
4.11 Macro Expansion and Code Compilation . . . 33
4.12 Proxy Call Workflow . . . 33
4.13 Direct Circular dependencies . . . 35
4.14 Enclavifying new methods . . . 36
4.15 Smarter Serialization and Deserialization . . . 37
4.16 Usable Security : Error Handling and Recovery . . . 38

5 Evaluation **40**
5.1 Execution of example code . . . 40
5.1.1 On executing command : Enclavify . . . 40
5.1.2 Macro Expansion . . . 42
5.2 Takeaways . . . 43

6 Future work **45**
6.1 Unification of templates . . . 45
6.2 Using compiled language . . . 45
6.3 Test Cases . . . 46
6.4 Using no-std libraries in Rust . . . 46
6.5 Complete Parser . . . 46
6.6 Packrat Parser . . . 47

7 Final Remarks **48**

Symbols and abbreviations

Abbreviations

TEE	Trusted Execution Environment
REE	Rich Execution Environment
VSCode	Visual Studio Code
IDE	Integrated Development Environment
JS	JavaScript
TS	TypeScript
LL(k)	Left to Right, Left-most deriving parsers with k look-aheads
LR(k)	Left to Right, Right-most deriving parsers with k look-aheads
AST	Abstract Syntax Tree
CFG	Context-Free Grammar
CNF	Chomsky Normal Form
PEG	Parser Expression Grammar
RE	Regular Expression
SDK	Software Development Kit
API	Application Programming Interface
RPC	Remote Procedure Call
REST	Representational state transfer

Terminology

- **Token :** A character or a group of characters that together form a meaningful bit of information related to the process of parsing.
- **Parsing :** The process of converting a stream of tokens into a larger all-encompassing object which represents a meaningful construct.
- **Stream :** A array or series of of objects. In this thesis, stream is referred to as a stream of tokens or a stream of characters.
- **Memoization :** A concept commonly used in dynamic programming. It is the process of storing a result of a computation in memory such that it can be looked up in subsequent computations.
- **Direct Dependency :** Here, direct dependency refers to all the methods that are used in a method under observation. For example, if a method, A calls methods B, C and D; then method A has B, C and D as direct dependencies.
- **Transient Dependency :** Here, transient dependency refers to methods which are calling a method under observation.
- **Crates :** External library files used in Rust applications.
- **Crates Dependency:** Dependencies which are result of using crates.

- **Cargo :** A file responsible defining crate dependencies and other paramters of a Rust application.
- **Serialization :** The process of converting an object into a string.
- **Deserialization :** Opposite of serialization. The process of converting a string into an object.
- **Marshal :** The process of converting an object into byte array.
- **Enclavification :** The process of carving out a series of methods from an existing Rust application and then converting these methods into a Trusted Application.
- **Operating System (OS) :** A piece of software providing basic IO, interrupt, threading and other functionalities which other user applications utilize. They also act as a medium between hardware and the user level software.
- **Trusted Execution Environment :** A special mode present in chips which provide security and integrity guarantees towards computation and storage.
- **Rich Execution Environment :** The regular execution environment where OS resides and rest of the normal user level applications are running.
- **Enclave :** A special virtual sandbox which can be created inside an application which guarantees security and integrity towards computation and storage. It relies on Trusted Execution Environment.
- **Trusted Application :** An application specially designed to be run inside a Trusted Execution Environment or Enclave.
- **Host Application :** The application which is executed in Rich Execution Environment and invokes a Trusted Application to perform secure computation.
- **REST :** It is a method of exchanging states between two different applications. This thesis discusses REST in the context of having two separate applications for communicating with VSCode and performing the Enclavification operation.

1 Introduction

Secure computation is a technology that enables computation while keeping data encrypted. It is usually used for managing secret keys and are very common in mobile phones to facilitate DRM, identity management and containing other secret parameters [1].

Traditionally, this has been achieved using Trusted Execution Environments, which is an isolation mechanism which exists as a special area on the main processor. It guarantees protection of data loaded in terms of integrity and confidentiality and offers isolated execution of code inside it. There are multiple levels of execution and each one of them have different privileges. Trusted execution environments exist at a higher privilege level than kernel levels and operates closer to the hardware [1]. Some of the popular TEEs are Intel SGX and ARM Trustzone.

Enclaves exist as a modern take on TEEs. While applications that run on TEEs are almost exclusively reserved for hardware manufacturers, Enclaves allow this by executing parts of the application in a virtual environment, It has a smaller trusted computing base and the applications do not need to be signed.

However, creating trusted applications running in enclaves and TEEs is a difficult process and tedious. It is often ridden with security vulnerabilities [2] and bugs which are overlooked. It also relies on third party tools and frameworks [3, 4] which do not have a strong integration with current text editors which further dampens the developer experience.

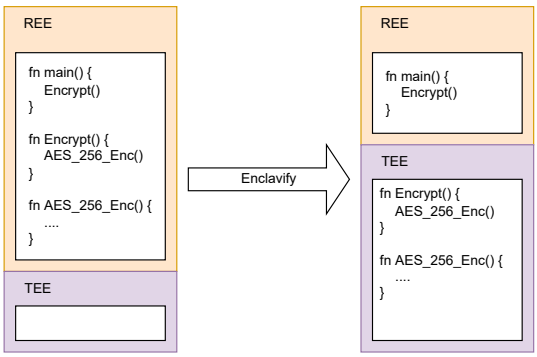

Figure 1: Enclavification Process

This thesis attempts to automating the process of carving out pieces of existing Rust applications into trusted and non-trusted code as *Enclavify*. The *Enclavification* process can be represented by the figure 1. This is done by creating a Visual Studio Code extension which generates code related to a smarter serialization and deserialization process as well creating a creating a channel for communication between the two Rust Applications.

Structure of Thesis

This paper is structured to have 7 sections.

- Section 2 attempts to provide some history and background knowledge about the literature of various components. In certain cases, various alternatives are also mentioned and a rationale for decision making process with references to the implementation wherever required is provided.
- Section 3 tries to condense the knowledge obtained in previous section into a problem statement and asks several important questions related to developer experience and security of building trusted applications. It also attempts to define the expected outcomes of this project.
- Section 4 provides insights into the implementation details and workflows of various sub-components of the projects. It also explains how these sub-components work together to generate the necessary code.
- Section 5 evaluates this work based on how well it answers questions posed to this thesis. It also demonstrates the *enclavification* process at different stages to evaluate how well the system works as whole.
- Section 6 discusses about various possibilities to improve the project in its current state and the future direction it can take.
- The last section, section 7 concludes this thesis by providing some final remarks.

2 Background

2.1 JavaScript

JavaScript (or JS for short) is an interpreted programming language, most widely known for creating and manipulating content on the web [5]. Based on the environment, it can be just-in-time compiled such as in case of Google's V8 JavaScript and WebAssembly Engine [6, 7].

Mozilla Developer Network describes JavaScript as a prototype-based (relies on object and functional prototypes), multi-paradigm, single-threaded, dynamic language, supporting object-oriented, imperative, and declarative (e.g. functional programming) styles [5].

JavaScript also has many non-web related usages, such as, used in Node.js, Adobe Acrobat, Apache CouchDB among others [5]. It is also a choice of scripting language for many web frameworks. It is often combined with other frameworks, such as Handlebars [8], to provide rich experience.

With the rise of frameworks such as ElectronJS [9], JavaScript is now used in may different non-web contexts. A popular example of this is Visual Studio Code (section 2.3).

JavaScript standard, also known as ECMAScript standard, is maintained by ECMA international [10, 11]. This international body is responsible for maintaining and adding more to JavaScript. However, to maintain backwards compatibility with older browsers, newer versions of ECMAScript are often compiled into older variations using tools like Babel.

2.2 TypeScript

In their conference paper, the author states that TypeScript is a superscript of JavaScript (section 2.1) built with the intention of creating large scale applications by adding types to the language [12, 13]. It provides additional syntax to JavaScript which aids in analyzing compile time errors by providing type safety to the language. Furthermore, TypeScript also provides other advanced features such as union and intersection types which helps in transitioning between the dynamic typing of JavaScript and stricter static typing of TypeScript smooth [14]. TypeScript also implements additional constraints related to importing dependencies which helps to avoid recursively importing or initializing other types and objects [12]. However, this type safety only exists in the context of TypeScript and as soon as the code is transpiled into regular JavaScript, those type safety guarantees are lost. This means, TypeScript does not provide run-time type safety since the code as executed as Just-in-time compiled JavaScript.

Furthermore, TypeScript also provides configuration files which allows selectively disabling currently undesired features which further aids in transitioning between the two languages.

According to Stack Overflow developer survey of years 2020 and 2021 [15, 16], there has been additive 4.79% increase in popularity of TypeScript with a corresponding

2.74% additive decrease in JavaScript's popularity.

2.3 Visual Studio Code

In their book Visual Studio Distilled [17], the author Alessandro Del Sole describes Visual Studio Code as an open source and cross-platform development tool that allows code editing throughout different code development scenarios. These scenarios often include web, mobile or cloud and also in a more traditional systems application development.

According to Microsoft, Visual Studio Code provides Editor, Debugger, Build systems, tasks and extensions among other things out of the box [18]. Through its impressive API system [19], Visual Studio Code provides an extensive set of methods which can aid developers to write variety of extensions [20]. These extensions can then be distributed further to different users. The extensions often utilize integrated command palette (accessed by *Ctrl + P* or *Ctrl + Shift + P* for command mode) to expose functionalities. These extensions can also use HTML based web page as interactive screens to provide further functionalities, configurations and customization options. Furthermore, Visual Studio Code also boasts build tasks, debugger support and integrated terminals to make development experience smoother.

Visual Studio Code is built upon ElectronJS [9] which leverages Google's V8 and Chromium engine [21, 6, 9] to provide web-experience as a desktop application. ElectronJS utilizes TypeScript (and by extension JavaScript) (sections 2.1 and 2.2) as a means for writing applications. Visual Studio Code expands upon this by also utilizing JavaScript and TypeScript for the purpose of developing extensions.

According to the Stack Overflow developer survey of 2021 [16], Visual Studio Code is the most popular IDE (Integrated Development Environment) amongst all the professional and amateur developers.

2.4 Parsing

In their book Compilers: Principles, Techniques and Tools, the authors Aho, Sethi and Ullman describe parsing as a process of converting and analyzing a formatted text or stream of symbols conforming to the rules of a formal grammar into a data structure. A parser is an algorithm which generates the aforementioned data structure by the process of parsing.

In the context of parsing programming languages, a parser converts text into AST (short for Abstract Syntax Tree) (section 2.5) [22]. Converting to ASTs provides benefits such as code analysis, comparison or editing which this project uses to implement all of the features discussed in this paper.

Parsers can often be categorized in two types [23] -

- **Top-down parser :** These parsers attempt to identify tokens as they are processed. Such parsers describe the rules as they appear, i.e. in the order of tokens. Furthermore, these parsers utilize inclusive choice (section 2.6) to handle ambiguity in grammar.

It is often believed that an implementation of such parsers cannot handle direct and indirect left recursion, and may require exponential time and space complexity when parsing ambiguous context-free grammars. However, a more sophisticated algorithm for top-down parsing was developed by Frost, Hafiz. , And Callaghan [24, 25] which also handles ambiguity and left recursion in polynomial time and generate a polynomial-sized representation of a potentially exponential number analysis tree.

These parsers often take form of recursive descent parsers or LL(k) parsers. Here, LL(k) can be described as left-to-right parsing, left-most deriving parsers with k look-aheads. LL(k) parsers can complete their execution in linear time, or $O(n)$ whereas recursive descent parsers take $O(k^n)$ (where n is the input size) to complete their execution. .

- **Bottom-up parser :** These parsers attempt to find the most basic elements first and then attempting to build a larger container element containing previously parsed smaller elements. This is done by putting currently unidentified tokens into a stack of tokens and when an identifiable token is found, stack is utilized to retrieve previously unidentifiable tokens to build an encompassing data structure.

  These parsers often take form of LR parsers, which be described as left-to-right parsing, right-most deriving parsers. An example of such class of parsers would be CLR (canonical LR) or LALR (LR with Look-Aheads) or SLR (Simple LR). Such parsers complete their execution in linear time, or $O(n)$.

This project utilizes recursive descent parsers implemented using parser combinators (section 4.6) to build ASTs. While this increases the time complexity of generating the ASTs, recursive descent parsers provide greater flexibility as they handle a larger set of grammar (LL(*)) and ambiguous grammar. This is because a recursive-descent parser is an implementation of PEGs, or Parser Expression Grammar(s) [26, 27]. This means that recursive descent parsers model the grammar they are parsing very closely. This provides an advantage in area of maintainability which creates a compelling case to use this parser.

2.5 Abstract Syntax Trees

ASTs or Abstract Syntax Trees are a type of data structure, typically used to represent stream of symbols conforming to the rules of a formal grammar, usually a language. A predefined construction of AST does not exist. However, it can be constructed as a minimum data structure which can be used to recreate original stream of symbols. An example of this can be visualized in the figure 2, where the rectangles in purple represent different possibilities of nodes of an AST.

ASTs are excellent tools for analysing different features of the language they are representing [22]. In their paper [22], the author discuss using abstract syntax trees for code comparison. However, ASTs can also be used to provide contextual insights about the language they are representing such as dependencies and types. ASTs also

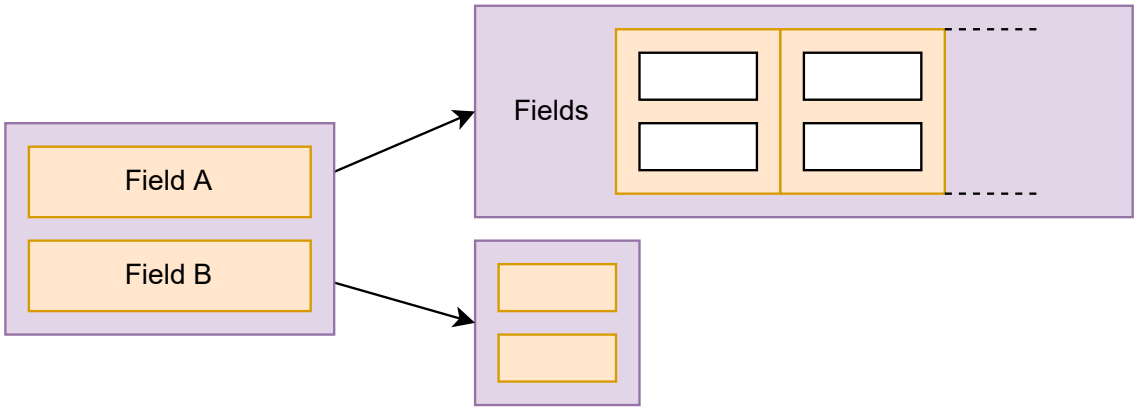

Figure 2: Example of a system of ASTs

make it possible to prune off different pieces of code or insert new pieces of code in an existing file.

ASTs can also expose features such as serialization, deserialization or recursive search or transformation. This project often transforms one form of the ASTs into another to generate enclave code or for augmenting existing code (section 4) and exposes these features in the form of interfaces which different AST nodes implement (section 4.7).

2.6 Grammars

In formal language theory, a grammar is described as rules related formation of strings from a language's alphabet [28]. It should be noted that a grammar does not describe the meaning of the strings. It does this by describing productions in the form of terminals and non-terminals. A production can be described as equation 1. According to this equation, this language (if it starts from A) accepts strings like $\beta\beta\alpha$ or $\gamma\beta\gamma\alpha$ and so on.

$$\begin{aligned} A &\rightarrow A\alpha|B \\ B &\rightarrow \beta|\gamma \end{aligned} \tag{1}$$

Grammar can be called ambiguous if it is not clear which part of the production should be used to generate the desired string. Equation 2 is an example of this. In this example, it is impossible to determine which part of the production is utilized if the input token is just α. This can be resolved by making the defined grammar unambiguous by extracting common parts of the production, as illustrated in the later part of the equation 2. This ambiguity can also be resolved by using a look-ahead in parsers (section 2.4).

$$\begin{aligned} A &\rightarrow \alpha A|\alpha B \\ B &\rightarrow \beta|\epsilon \end{aligned} \xrightarrow{transformation} \begin{aligned} A &\rightarrow \alpha(A|B) \\ B &\rightarrow \alpha B|\epsilon \end{aligned} \tag{2}$$

Furthermore, Mutually recursive functions often lead to infinite recursion by following left hand recursions. An example of this would be - an argument to a function call can be an expression but an expression can also be a function call. This is resolved by the method of replacing productions, i.e.:

$$A \to A\alpha|\beta \begin{cases} A & \to \beta A' \\ A' & \to \alpha A'|\epsilon \end{cases} \tag{3}$$

2.7 Interpretations of Context-Free Grammars

According to formal language theory, context-free grammars (or CFG for short) are formal grammar which follow the form as illustrated in equation 4 where A is a non-terminal and α is a terminal. Syntactically, the left hand part of the equation must be a non-terminal. The right hand part of the equation can be a terminal, non-terminal, option or a null(ϵ) production. All the grammars discussed in this paper are context-free grammars.

$$A \to \alpha \tag{4}$$

There are many interpretations of CFGs. However, only the following are relevant to this paper.

2.7.1 Chomsky Normal Form

Chomsky, in an attempt to create non-ambiguous grammar, tried to constraint CFGs by applying a set of rules [29]. According to these rules, all the productions must satisfy one of the following conditions -

- A non-terminal should should generate terminal, i.e. $A \to \alpha$.
- A non-terminal should generate a sequence of two non-terminals, i.e. $A \to BC$.
- The start symbol can be a null production, i.e. $S \to \epsilon$.

However, CNF is very restrictive and requires more productions to describe a language. Furthermore, there no real need for utilizing just two non-terminals. A more succinct grammar can be described using a production which produces more than two non-terminals. Furthermore, mixing terminals and non-terminals do not always produce ambiguous grammars. However, if the CFG has an ambiguous grammar, it can be made unambiguous by utilizing the solutions menetion in the section 2.6.

2.7.2 Greibach Normal Form

Sheila Greibach made an attempt to avoid left-hand recursions in CFGs by constraining the CFG to a set of rules [30].According to these rules, all the productions must satisfy one of the following conditions -

- A non-terminal should generate a terminal, i.e. $A \rightarrow \alpha$.
- A non-terminal should generate a terminal which can be followed by any number of non-terminals, i.e. $A \rightarrow \alpha BCD$.
- The start symbol can be a null production, i.e. $S \rightarrow \epsilon$.

However, this leads to very restrictive form where the CFG no is no longer close to the language it attempts to model. Left recursions can be solved as mentioned in section 2.6.

2.7.3 Backus Naur Form and Regular Expressions

Backus Naur Form (or BNF for short), is one of the best-known example of a meta language where one syntactically describes a programming language [31]. They are represented as symbol expanding into an expression as illustrated in the equation 5. This form allowed representing productions in a more expressive manner by representing non-terminals as $< symbol >$. Terminals were represented by enclosing them in quotations.

$$< symbol > ::= < expression1 > \text{``.''} < expression2 > \mid < expression3 > \tag{5}$$

According to formal language theory, regular expressions are succinct way of describing the grammar of a regular language. They have the same expressive power as regular grammars. However, regular expressions provide additional symbols such as - * for zero or many, + for one or many and ? for optional - which allow them to represent a grammar very succinctly [32]. Modern implementations of regular expressions (such as Perl Compatible Regular Expressions) even provide match groups and certain predefined groups which further expands upon the succinctness of regular expressions.

Newer versions of BNF were added later (extended BNF and augmented BNF) which borrowed features from regular expressions such as grouping, zero or many, one or many and optional which further increased the succinctness of BNFs and made them a popular choice for representing syntax as a meta language.

However, BNFs cannot handle ambiguous grammars. BNFs cannot distinguish between different parsing trees just by analyzing the strings. A popular example of this is the dangling-else problem [33]. This is represented by the following snippet1

Listing 1: Ambiguous BNF

```
IfElse ::= "if" <expr> "then" <expr>
    | "if" <expr> "then" <expr> "else" <expr>
```

Listing 2: Ambiguous BNF Interpretation

```
if a then (if b then s) else s2
if a then (if b then s else s2)
```

For the string "if a then if b then s else s2" , both the syntax trees mentioned in the code snippet 2 are equally possible and there is no way to determine which parsing tree generated the string.

2.7.4 Parsing Expression Grammar

In his paper, Parsing expression grammars: a recognition-based syntactic foundation [34], author and creator Bryan Ford argues that Context-free grammars and regular expressions makes it unnecessary difficult to express and parse machine oriented languages using CFGs. They provide an alternate, PEGs which solve the ambiguity problem by not introducing the ambiguity in the first place. CFGs become non deterministic because all the alternatives are equally viable and they propose that a system (PEG) which uses prioritized choice instead produces only a single parsing tree.

PEG grammars choose the first match in production. Therefore, there is only a single parsing tree possible from a given string. While CFGs lack the ability provide control over selecting the correct parsing tree over alternatives, PEG provides this control by reordering the alternatives. Secondly, this also closely resembles how code executes in reality. The execution is greedily short-circuited as soon as the first match is found. This makes PEGs a good candidate to represent the flow of execution of code as well as expressively describe the language.

Furthermore, PEGs can be directly transcribed into a recursive descent parser [27, 26]. The inherent backtracking provided by recursive descent parsers also remove the LL(1) constraint.

Unlink BNFs, PEGs never have the "Dangling Else" problem. This is because PEGs select the first successful option. Therefore, the generated string is always generated from the first tree.

2.8 ANTLR and other parser generator tools

While trivial, writing parsers is a time taking and error prone process. For something as extensive as a programming language, the problem is compounded. Therefore, this process is automated by utilizing parser generators. These utilize a metadata representation of the grammar (such as PEG or some form of BNF) to generate parsers and optionally integrated tokenizers.

ANother Tool for Language Recognition (or ANTLR for short) is a popular parser generator tool [35, 36], often used to build analysers or programming languages or language servers by generating a parser scaffolding. They work by generating static code which as a pre-build step. Another popular tool is GNU Bison [37] which works in a similar way.

However, as discussed in Implementation section (section 4), automated tools might not provide all the features desired from ASTs. Therefore, a recursive descent parser utilizing parser combinators is used for this project.

2.9 Trusted Execution Environments

In their paper [1], Ekberg et al. define Trusted Execution Environment as a secure, integrity-protected environment, consisting of processing, memory, and storage capabilities. It implements a logically isolated security mode using just one chip. This also includes introduction of a status bit in the processor, which is used to determine the status of data stored in memory and whether or not the processor is in a secure mode.

Trusted Execution Environments are isolated from the normal environment called the Rich Execution Environment (REE), where the device OS and various other applications run. This setup allows for possibility to execute sensitive operations of an application such as encryption/decryption or other use of cryptographic keys inside the secure Trusted Execution Environment whereas user facing and non-critical parts of the application to be executed in REEs.

Before the 1990s, the security of mobile devices was rarely ever recognized as a problem. This was usually because strict government regulations and the mobile phones themselves were just a closed system. However, in the mid 1990s, mobile phones started turning into open systems where developers were encouraged to publish their ideas in the form 3rd party Java applications. These mobile phones quickly started resembling small hand-held computers. There was an explosion in mobile phone usage and soon there was a demand for secure storage for SIM locks, device identity (International Mobile Equipment Identity, IMEI) and other parameters for radio frequency transmission. Security of such earlier phones were usually implemented as software solutions and protected by secrecy within organizations. The essential weakness of this kind of “security through obscurity” design was the loss of protection after the design was revealed. Introduction of hardware based platform security marked a groundbreaking switch from “security through obscurity” towards “security through transparency”. This also transformed security as a feature to security as an integral part of the platform design.

In 2009, Open Mobile Terminal Platform described a specification for Advanced Trusted Environment [38]. In 2010, GlobalPlatform published its first TEE standard, TEE client API 1.0 [4]. These days, TEEs are widely available and used on mobile devices, both iOS and Android smartphones.

2.10 Trusted Applications

Trusted Applications are applications designed specifically to be executed inside a Trusted Execution Environment. They execute on top of a minimal runtime environment, called the trusted OS. This OS provides an internal TEE API which the trusted applications can use to access secure storage inside TEE, or to perform cryptographic operations and to communicate with the applications currently running in rich execution environment.

However, improperly designed applications also open the door towards critical security vulnerability. For example, the paper "BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments" [2] discusses the possibility where

a user-level application can leverage a trusted application to read from or write to a part of memory in REE that it does not own. It is also desirable that the TAs should not manipulate the memory of the applications which are invoking the TAs in the first place. Furthermore, trusted applications can utilize a very reduced set of standard IO functionalities which is available through the trusted OS APIs. Therefore, most of the functionalities available in normal applications getting executed in REE do not get translated well into an equivalent counterpart. An example of this would be utilization of only "NoSTD" library functions in TAs.

This project attempts to rectify these issues by pointer sanitization and smarter serialization and deserialization in which the rust macro system generates scaffolding for communication between the host application and the TA. It also creates recursive serialization and deserialization functionalities which are invoked automatically and seamlessly when the appropriate method is called. The communication between the host application running in REE and TA is facilitated using remote procedure calls (RPC). See section 4 for more details.

2.11 Enclaves

The authors of the paper "Innovative Instructions and Software Model for Isolated Execution" [39] describe Enclaves as a protected area in an Application's address space (as illustrated in figure 3), which provides confidentiality and integrity even in the presence of privileged malware. Attempted accesses to the enclave memory area from software not resident in the enclave are prevented even from privileged software such as virtual machine monitors, BIOS, or operating systems.

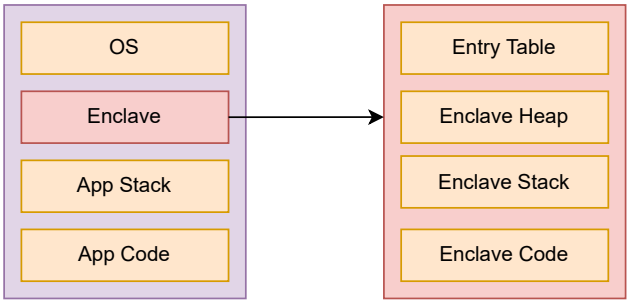

Figure 3: Enclave within application's virtual address space (derived from [39])

Enclave code and data is usually free for analysis before the enclave is built. However, once the application's code and data is loaded into an enclave, it is protected against all external software access. An application can also prove its identity to a remote party and be securely provisioned with keys and credentials. An application can request platform or enclave specific key that it can use to protect data that is to be stored outside the enclave.

This project generates code and necessary scaffolding required to build the trusted application (section 2.10) which is loaded into an enclave for execution. This paper discusses some of the implementation details in section 4.

2.12 Difficulty with creating Trusted Applications

We believe that creating trusted applications is a difficult, tedious and error prone process. There are multiple caveats that must be kept in mind while writing trusted applications. Furthermore, existing standard APIs are not ergonomic enough for creating interface between host and enclave.

- Global Platform Trusted Execution Environment Client API [4] requires passing method parameters and return values as byte buffers, forcing the developer to write ad hoc protocols. This is especially true of TEEC parameters and other methods, as evident by the documentation.
- SGX SDK [3] provides automated serialization of parameters and return values, but developer needs to define the interface with an interface definition language (IDL) which only supports relatively simple C based types.
- We also believe that developing for rich execution environment first, and then converting parts of application into trusted application is an easier endeavour compared to switching contexts and writing both the applications at the same time.

Some of these factors contribute towards developers building their own ad hoc protocol for communication between their host application. This adds risk of introducing critical security vulnerabilities such as Boomerang vulnerability [2] to the application, if the developer does not have prior experience with this. We fix this with macro based automatic rpc interface generation.

Furthermore, this project also provides a tight integration with Visual Studio Code, where the developer can expressively select parts of the application that needs to be converted into trusted application by simply highlighting the desired methods, thus contributing towards better ergonomics and smoother developer experience.

2.13 Rust

The authors Matsakis and Flock, in their paper The Rust Language [40], describe Rust as a programming language designed with concurrency in mind. Rust also provides a static type system which is safe, expressive and provides strong guarantees about isolation, concurrency, and memory safety. Rust also provides a strong control over memory representation, allocation and de-allocation with direct support for continuous record storage and stack allocation.

Furthermore, Rust is a low level programming language, which can also be used for systems development. Rust introduces new concepts over C++, such as compiler enforced ownership and borrowing which enables the language to provide memory safety without the need of garbage collection.

Rust also boasts a powerful macro system which accepts a stream of tokens and can expand those streams into meaningful code. This project leverages this feature of rust to generate code which enables automatic smart serialization/deserialization and automated RPCs. It also generates proxy clients for hosts and service for the

trusted application. The extension system then generates extra code to inject these clients into the original rust application (section 4).

3 Research Question and Method

3.1 Problem Definition

Creating trusted applications for enclaves is a difficult and tedious process. It is often ridden with security vulnerabilities and bugs which are overlooked and often not identified unless the developer has the correct background and experience. The author, in their paper [2], describe one such critical security vulnerability where a host application currently running in rich execution environment can leverage a trusted application in TEE to write to, or read from, a chunk of memory that does not belong to the application.

Secondly, creating trusted applications require some degree of boilerplate code and often involves using third party tools which do not necessarily have a strong integration with current text editors which further dampens the developer experience.

- Objective to explore and investigate how to use IDE automation and rust macros to easily and safely split a program into user mode application and an enclave.

3.2 Research Questions

This paper attempts to explore the area of *Usable Security*. It attempts to answer the following questions.

- How can we mitigate common security vulnerabilities related to communication between host application and trusted applications running in enclaves?
- How can we avoid writing boilerplate code and build trusted applications from minimum code as a minimal viable product?
- How can we convert parts of a normal rust application into a trusted application?
- How can we improve the developer experience?

3.3 Research Method

This project attempts to answer these research questions using the process of automation by code generation and strong integration with an IDE, specifically visual studio code.

The common vulnerabilities associated with communication between a host application and a trusted application can be avoided by using a smarter serialization and deserialization method. This code is generated as the user selects parts of the application that needs to be *Enclavified*. The code generation also generates the required boilerplate code necessary for gluing the two parts of the application and their communication channels. The whole experience is made more enjoyable for developer due to strong integration with visual studio code, where the developer does not need to leave the IDE for the results and developer can select the parts of the application to be *enclavified* expressively by simple code selection.

4 Implementation

This project mainly follows two workflows :

- **Create Scaffolding :** To generate an empty project with minimum viable enclave. This can be considered as a starting point if the developer wishes to create a new application while leveraging the advantages of enclaves.
- **Enclavify** - This command allows to convert selective parts (such as methods) of an existing rust application into enclaves. The generalized workflow of this command can be visualized using the figure 4.

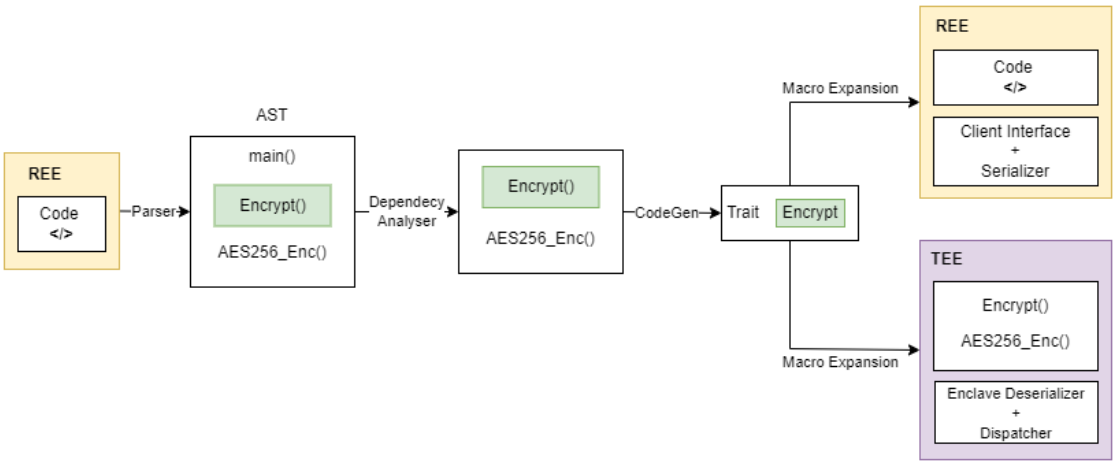

Figure 4: Enclavification Workflow

4.1 Visual Studio Code

Architecturally, Visual Studio Code combines the best of web, native, and language-specific technologies. Using Electron, VS Code combines web technologies such as JavaScript and Node.js with the speed and flexibility of native apps [18]. Among various tools such as integration with terminals and web browsers, Visual Studio Code also offers a rich ecosystem of extensions and plugins which leverages upon the extensibility of visual studio code to introduce 3rd party features into the code development ecosystem [20]. This allows users to cherry-pick selected features among a vast array of language specific or use case specific 3rd party features developed by independent developers.

This project leverages TypeScript to develop a visual studio code extension [19]. The project exposes multiple functionalities to the end user using Visual Studio Code's command palette functionality. The command palette can be accessed using the hotkeys *ctrl + shift + P*. Visual Studio code command palette exposes the following commands -

- Enclavify - This command allows to convert entire or selective parts (such as methods) to an existing rust application into enclaves. The generalized workflow of this command can be visualized using the figure 4.
- Create Empty Scaffold - This command is utilized to generate a blank template or scaffold to assist in enclave development.

Furthermore, visual studio code also allows bundling of extension so that it can be further distributed or installed into other instances of visual studio code. This integration with the extension ecosystem further promotes the popularity of the text editor.

4.2 Code structure

Code is expected to follow the structure depicted in the figure 5. The client code and the host code follow an identical structure for ease of comprehension.

- The boxes colored in yellow are considered client code or host code. This part of the code is responsible for invoking the enclave functions.
- The boxes colored in purple are considered to be the enclave code. This code is responsible for executing various pieces of code in an enclave in a trusted execution environment.
- The box in red is for other miscellaneous purposes. It might contain other user or generated files. As a part of the project, this folder contains various configurations required for the enclave to function. There is also a root cargo file present at the root of the project. This file is responsible for smooth gluing the host and the trusted parts of the application.

Creating an empty scaffold creates the expected code structure 5 as depicted above. Whereas for the purpose of enclavification, the system expects client code to be present in a structure as depicted above. The system cannot proceed if there is a mismatch in in the location of Cargo.toml and main.rs (for client). This error is displayed in the form of a message in a pop up box provided by visual studio code for general notification.

During code generation, all the necessary files are generated in the expected locations. Furthermore, if there are missing files (such as lib.rs in case of client), such files are also generated as depicted in the figure 5 as part of the code generation step.

4.3 Code Scaffolding

This project is bundled with two different copies of the bundled files. While similar, they are not identical and serve different purposes. Here, they are referred to as Base and Template.

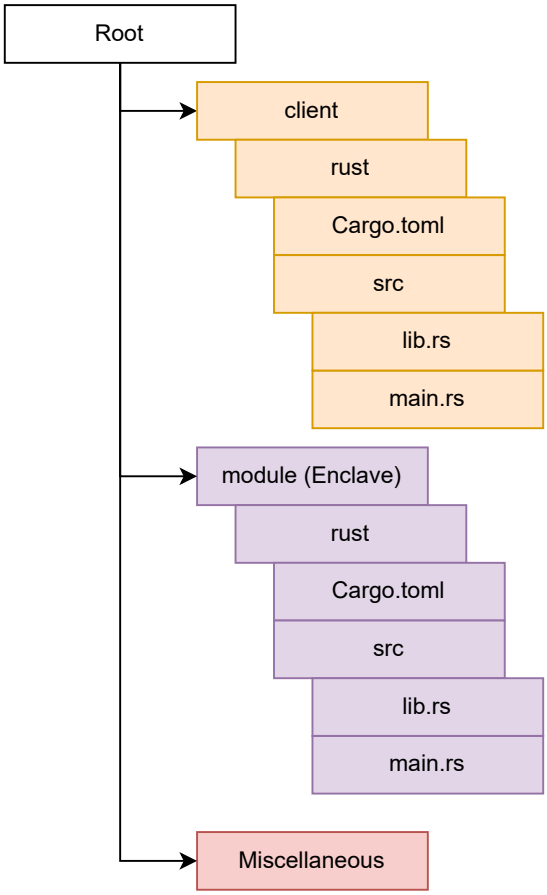

Figure 5: Expected Code Structure

- Base files are a form of a minimum viable enclave and provides just enough content and code to start coding for an enclave. In this project, they are used to generate the missing files (such as lib.rs in clients) as depicted in the figure 5 or provide augmentative data (such as required dependencies for enclave) required for the code analysis step 4.

- Template(s) are similar to the Base scaffolding files but they differ in the sense that they do not contain all the necessary files for building an enclave and thus are not a part of minimum viable enclave. Furthermore, they also contain different *slots* in the Handlebar syntax [8]. These *slots* are identifiers regarding the places where various generated code can fit into so that they follow a more intelligible code structure. Furthermore, it is more efficient to generate contents of a file by slotting in enclavified code rather than converting Base template into AST, inserting the new enclavified nodes and generating the entire file.

4.4 Handlebars

Handlebars are a JavaScript based minimal templating engine [8]. It is commonly used to provide slotting mechanism to various popular web frameworks. On the most

basic level, it works by having double opening and closing braces to provide named slots into a document which appear as {{Named_Slot}}.

Handlebars templating engine then replaces these named slots with appropriate text provided as part of arguments while calling the conversion function, thus generating the complete document. Handlebars expects an object with fields corresponding to different named slots. It emits *undefined* in case a required field is missing. Thus, to generate a document which follows some rules, it is important for all the fields to be present.

This project uses handlebars because it provides an easy mechanism to slot in generated code into different places between templates which results the generated document to be more intelligible and arranged.

In addition to named slots, Handlebars also provides some other text processing features such as mapping objects into texts or allowing custom processors in the form of block helpers. Handlebars also offers nesting different templates by using partials which also promote template reusing.

4.5 Parsing and Tokenization

In their book Compilers: Principles, Techniques and Tools, the authors Aho, Sethi and Ullman describe parsing as a process of converting and analyzing a formatted text or stream of symbols conforming to the rules of a formal grammar into a data structure. In case of this project, the document being parsed is a rust code and it is being parsed into an AST (or Abstract Syntax Tree).

Conventionally, parsing and tokenization are done as separate steps. While tokenization, many of the non-important details, such as presence of white space, is discarded from the parsed data and the result of different tokens is converted into different data structures called Abstract Syntax Trees (or AST for short) [23]. However, discarding such non-trivial details also removes the ability of controlling the flow of the document by allowing or disallowing white spaces at different parts of the document.

Therefore, this project parses and tokenizes in a single step. This is done by providing tighter ignoring and expectation rules provided in the form of parser combinators (section 4.6).

This project implements a recursive descent parser implemented as a LL(1) parser, which means the process of parsing goes from left to right, has left most derivative and utilizes 1 look-ahead. It is a top-down parser which depends on a set of mutually recursive methods where each of such methods implements one of the non-terminals of the grammar. Thus, they very closely model the structure of the programming language. They allow developers to compare the structure of a code with the parser implementation with ease.

The output of every parsing step is either an *undefined* which signifies that the parsing failed and the parser can move onto next step or it provides an AST of the resulting step. In case there is no next step available for the parser, it returns a tuple of an array of different abstract syntax trees it successfully managed to parse and the remaining text. In case everything was parsed successfully, the remaining text

would be an empty string.

4.6 Parser Combinators

Parsers are often implemented by using various toolkits or parser generators such as ANTLR [35, 36] or GNU Bison [37]. However, strong serialization capabilities were required in this project which prompted writing a parser by hand. However, writing parser by hand is a tedious and error prone endeavour and while the overall structure of parser resembles the code it is modelling, the individual functions become difficult to understand.

A parser combinator is a higher-order function that accepts several parsers as input and returns a new parser as its output [25, 24]. Parser combinators abstract away code such as error handling and look-ahead adjustments into easy to use higher order functions such as *Aggregate*, *Loop*, *Optional*, *Flush*, *ExpectButNotTokenize*, *Then* which allow various parsers and parser combinators to be chained with each other. As a result, describing the specification of text to be parsed can be done in a much more succinct way.

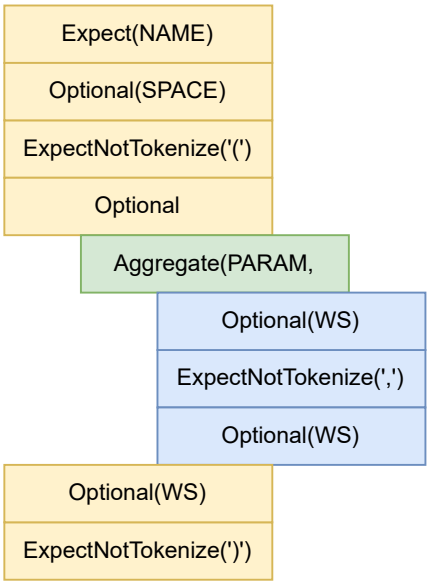

Figure 6: Example of Combinators for parsing a simple function call

Figure 6 demonstrates an example where using parser combinators instead of using simple parsers provides a much more developer friendly experience. It also allows describing the exact specification to be more succinct. In this case, *NAME*, *SPACE*, *WS* (or white space) and *PARAM* are separate higher order functions.

One may notice in the figure 6 that the delimiter part of the Aggregate step is a composite combinator which allows the flexibility for grouping white space and the comma as a composite combinator group. Furthermore, the individual higher order functions (such as *PARAM*) can also call each other mutually recursively to parse the entire text and thus closely resembling to the language specification.

However, mutually recursive functions often lead to infinite recursion by follwing left hand recursions. An example of this would be - an argument to a function call can be an expression but an expression can also be a function call. This is resolved by the method of replacing productions, i.e.:

$$A \to A\alpha|\beta \begin{cases} A & \to \beta A' \\ A' & \to \alpha A'|\epsilon \end{cases} \tag{6}$$

The implementation of parser combinators in this project also allows for custom tokenization features using which, it is possible to re-arrange or process the tokens. These are called *assigners* in the implementation, and they are using to convert a token stream into an abstract syntax tree. The implementation also supports consuming multiple ASTs in an assigner to generate a greater node and thus building a tree structure (section 4.7).

4.7 Abstract Syntax Trees

Abstract Syntax Trees (or ASTs for short) are data structural representation of a code that has been parsed. They are implemented as different nodes created by a recursive descent parser which are nested into each other to create a tree structure. Figure 7 depicts an example of an AST which represents a function call whose arguments are a variable with some value and some other function call. Note that these arguments have their own fields and values which are hidden in the figure for simplicity.

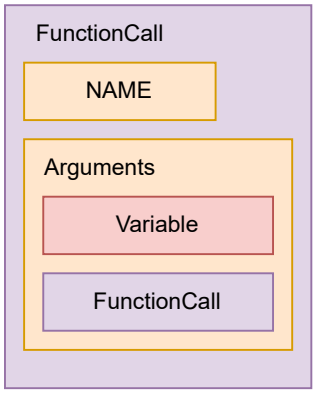

Figure 7: Example of Abstract Syntax Tree

In this project, AST nodes also provide multiple functionalities. These functionalities are implemented in the form of inheritance, which further enforces the contract that these nodes are guaranteed to provide these functionalities. They are -

- **IAssignable :** It is an interface which exposes the method *Assign*. This method accepts a token stream, and returns an array of tokens or other ASTs (specifically other IAssignables). It was a deliberate decision to return an array of IAssignables since in certain cases (such as tokenization of crates and its submodules), a token stream can return multiple assignables.

- **ISerializable :** It is an interface which exposes the method *Serialize*. Every node is responsible for converting itself into a compatible string. The string might not be exactly same as the input string of the parser, but this is due to multiple tokens which are ignored and thus not important to actual processing of the code. An example of this would be whitespace or simply space. As a result, a multi-line function call can be serialized into a single-lined function call. However, this can easily be rectified by changing the description of the parser in the parser combinator setup 4.6.

- **IExtractable :** It is an interface which exposes the method *ExtractAll*. It provides a mechanism to recursively traverse the AST nodes and perform a specified operation on all the nodes of a specified type. A use-case for this is to convert all the direct function calls into proxied function call via injected client.

4.8 Code Analysis

During the process of enclavification, the system performs an extensive code analysis of a client's file to obtain various insights into the methods which are to be enclavified. This is a multi-step process to identify various transitive (direct and indirect) and crate dependencies of the function to be carved into enclave.

In the figure 8, Function B (colored in green), has been selected to be converted into an enclave. Function D (colored in in yellow), is an independent function that is not affected by the enclavification process. Function A and C are affected by the enclavification process. Furthermore, there are crate dependencies that Function B and C rely upon.

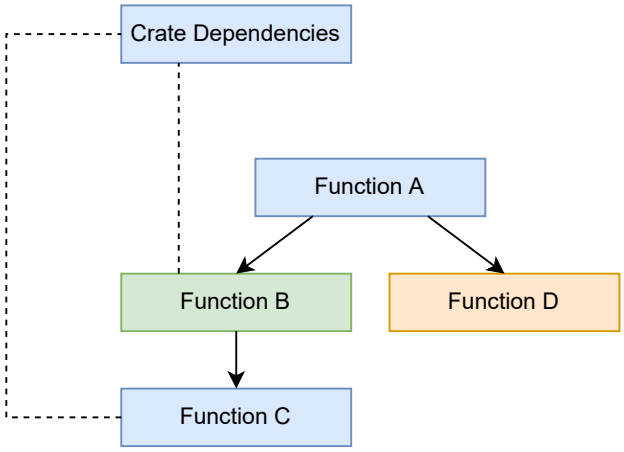

Figure 8: Example of Abstract Syntax Tree

The code analysis is a 3 step process which can be explained as follows -

- **Identifying transient dependencies :** This step identifies all the methods which a function to be enclavifed depends on. Referring to the figure 8, this

step identifies Function C as the affected function. his step utilizes *IExtractable* (section 4.7) to identify such dependencies

- **Identifying crate dependencies :** This step utilizes *IExtractable* (section 4.7) to locate all the namespace pathing which have a direct counterpart in the uses sections of the ASTs. This step is repeated for all the functions which are identified in the previous step.

- **Creating proxy of enclavified function in callers :** This step utilizes *IExtractable* (section 4.7) to identify the Function B call in all the other methods which are not sub-nodes of the function to be enclavified. In the example in figure 8, function A is a caller to function B, and is therefore eligible for this step. A client to the enclave is injected into the nodes of Function A. Furthermore, all the call sites are replace with a proxy call routed through the client injected.

In the end, these insights are passed onto the code generation (section 4.9) step of the workflow to generate the scaffolding, rest of the enclave code and to modify existing code.

4.9 Code Generation

The code generation steps utilizes insights obtained in the code analysis step (section 4.8. In this project, the code generation step is not only responsible to serialize ASTs, but it is also responsible for transforming one form of AST into another and also to generate the required scaffolding. This is a multi-step process which works as follows

- **Create Scaffolding :** The project comes bundled with two sets of template code (section 4.3). This step utilizes the bundled scaffolding code to generate all the additional files required for functioning of the enclave. This scaffolding is further augmented by rest of the generated code to build a meaningful unit of code. It should be noted that creating an empty scaffolding for a new project utilizes a different work flow.

- **Enclave Generation :** This step involves converting nodes of function that need to be enclavified into different formats as well as importing the crates and transient dependencies into the enclave. In the figure 9, different colors represent the relation between how they were generated. A trait's function signature can be derived from the function to be enclavified.

  In trait implementation, enclavified function can be utilized with minimal changes. This step requires transformation of AST since the serialized results are different and thus they are two different types of function declarations.

  This step also creates a new copy of methods which are direct dependencies of the enclavified method, retaining the original in the client code.

- **Enclave Cargo Generation :** The code generator analyzes the scaffolding created for the cargo file, which at this point simply contains dependencies

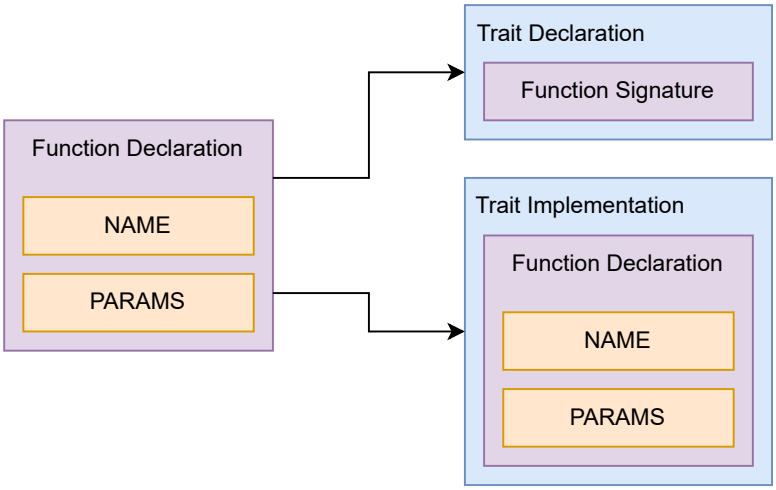

Figure 9: Conversion of Abstract Syntax Trees to other forms during code generation

required for building a minimum viable enclave. The cargo dependencies obtained in the code analysis step (section 4.8) is used to augment the overall cargo file.

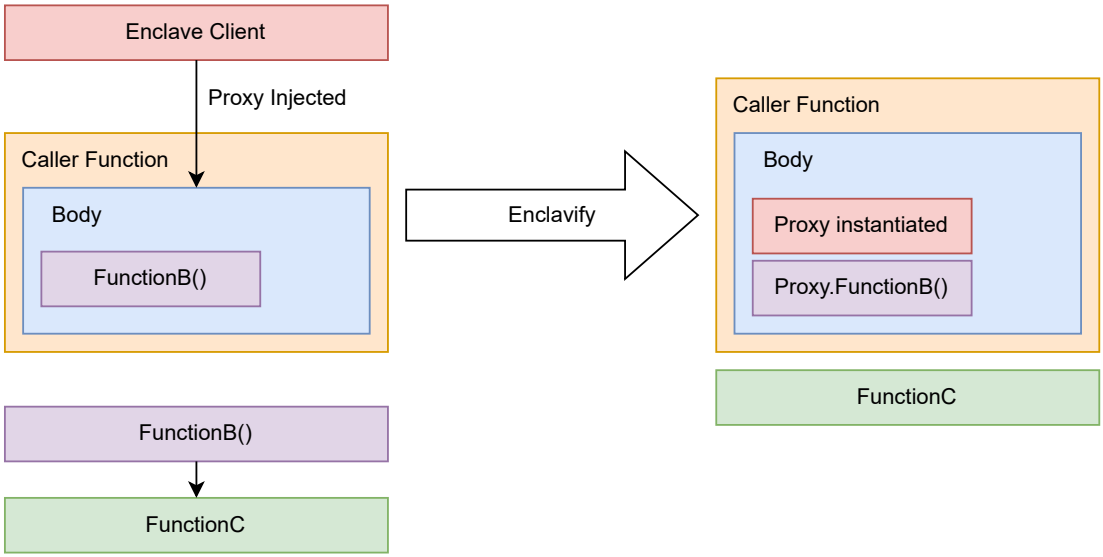

Figure 10: Injection of enclave proxy into client's caller function after code generation and subsequent removal of enclavified function from client.

- **Client Main File Augmentation :** Figure 10 demonstrates multiple sub-steps that are involved in executing this step. This step removes the enclavified method (Function B) from the client but retains its direct dependencies (Function C) in the client file. The code generator retains another copy of Function C in enclave code to avoid any conflicts,

  Furthermore, this step also injects enclave client proxy into the caller method

and replaces all the relevant call-sites using *IExtractable* (section 4.7) with calls routed via the proxy. Figure 10 demonstrates the entire process.

- **Client Cargo File Augmentation :** The code generator uses details in existing client cargo file and augments it with the client cargo file from the Base scaffolding (section 4.3). The result is a cargo file which has the necessary imports for the generated enclave as well has the imports for all the direct crate dependencies of the client main file.

Overwriting existing code is a dangerous process and often poses security issues, Simply replacing code might also remove pieces of code which were important. Therefore, code generation step comments out existing pieces of code which needs to be overwritten. This is the case for enclave main and cargo and client main and cargo files. This also provides an opportunity to recover code which were unintentionally enclavified.

4.10 Code Transformations

As mentioned in code analysis (section 4.8) and code generation (section 4.9), this project relies heavily on transforming various ASTs (Abstract Syntax Trees) into other forms. An example of this is the transformation of method to be enclavified into trait and its implementation.

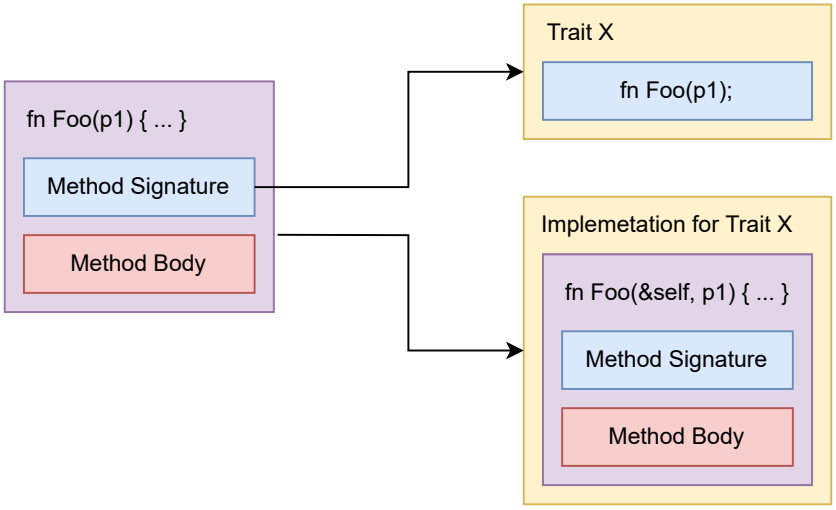

Figure 11: Proxy Call Workflow

Figure 11 illustrates this process. A method to be enclavified, colored in purple here, is split into its signature, which is used to create the trait. The method (purple) is also used to create the AST for implementation of the trait. In the illustration, all the related parts are colored in the same shade of color.

These transformations happen in the form of ASTs. While the method and the implementation might look similar, but they have different kind of serializations. Note the "&self" as parameter to the method in implementation. This means, both

the AST nodes have different JavaScript prototype and are different classes and the transformation is involves merely copying the different properties.

It should be noted that transformation of macros into proxy client and service (section 4.11) is provisioned using the rust macro system and the macro expansion is different type of code transformation and expansion.

4.11 Macro Expansion and Code Compilation

This project leverages the powerful feature of rust - macro expansion, to generate extra code related to communication between the host application and the trusted application. Once the code is generated (section 4.9), the resulting code can be compiled to expand macros and create actual enclaves. To facilitate this process, the code generation process also produces a build file named *build.sh* in the root directory of the project.

In case of manual compilation, the enclave needs to be compiled first, to generate the necessary stubs for the enclave. Thereafter, client file can be compiled and lastly, the code can be executed.

The macro is expanded into proxy code which can be injected into the host as well as a service which the trusted application can utilize. The proxy code is generated as extra files which can be used directly into the host.

Figure 12 illustrates this process appropriately. The parts colored in purple stay in the trusted part of the application whereas the part colored in green is sent to the original host application code. The processes of code analysis (section 8) and code generation (section 4.9) injects the proxy client code into the appropriate methods and then attempts to replace the call sites wherever required. The parts colored in yellow in TA code (trait and its implementation) are used to generate the parts colored in yellow in client macro expansion (structure and implementation) and trusted application macro expansion (structure and implementation).

4.12 Proxy Call Workflow

The steps mentioned previously - code analysis (section 4.8) and code generation (section 4.9) replaces the original call sites of the enclavified method with proxy clients which contain a method with same name.

Refer to illustration in figure 13. The setup works as follows

- Proxy method (replaced at original callsites) is called.
- This method utilizes smart serialization process (section 4.15) to convert the passed arguments into a byte array. Communication between the host application and the trusted parts can only be done in byte array. This also creates copies of the values that are being pointed to by various pointers. These values are also moved into enclave memory (section 2.11).
- The proxy method invokes a dispatch method in the trusted application service generated by macro expansion (section 12).

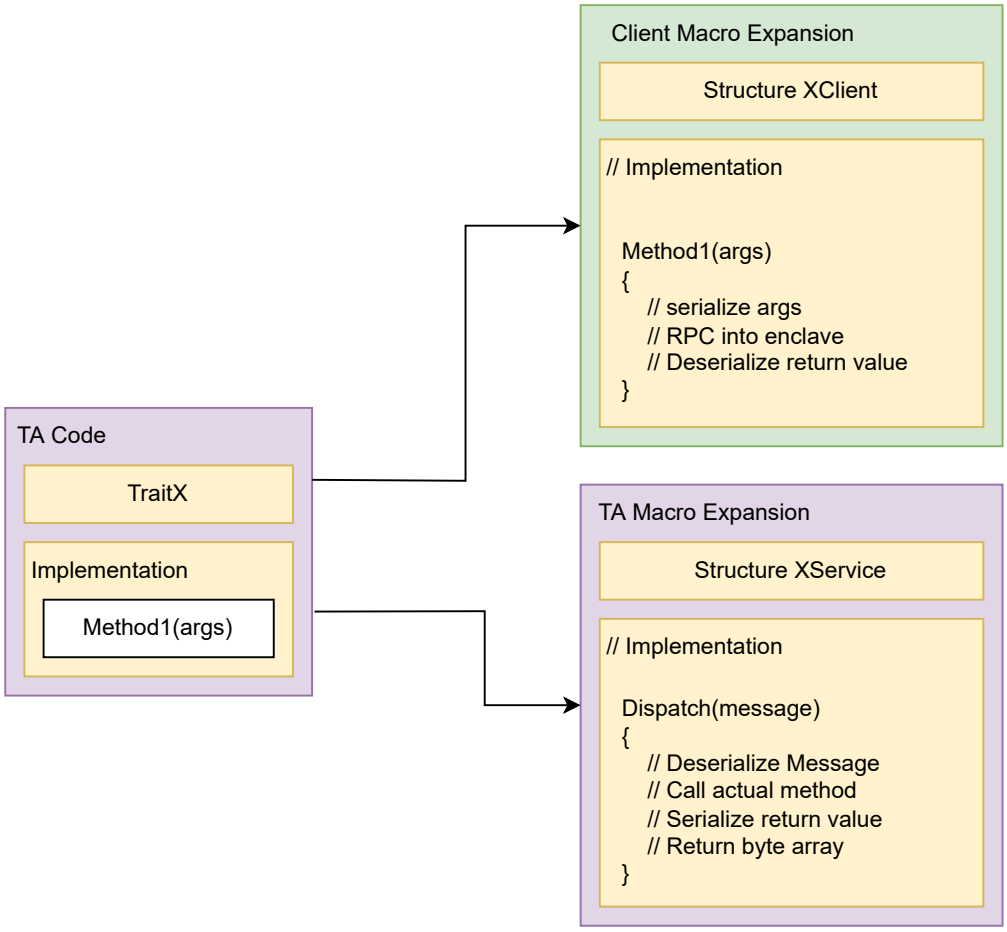

Figure 12: Macro Expansion Process

- The dispatch attempts to deserialize the byte array to reconstruct the original arguments.
- The original methods (implemented as a trait implementation) still exists in the trusted application. The dispatch calls this method while also passing all the parameters required. To the original method, it appears as if this is a normal method call.
- The return value of the original method is converted again in a byte array using smart serialization (section 16). This time, this copies all the values being pointed to by various pointers into rich execution environment memory.
- This byte array is returned as a response to the remote procedure call.
- At the proxy method, this response is deserialized into actual values, and the result is returned as the return value of proxy method call.

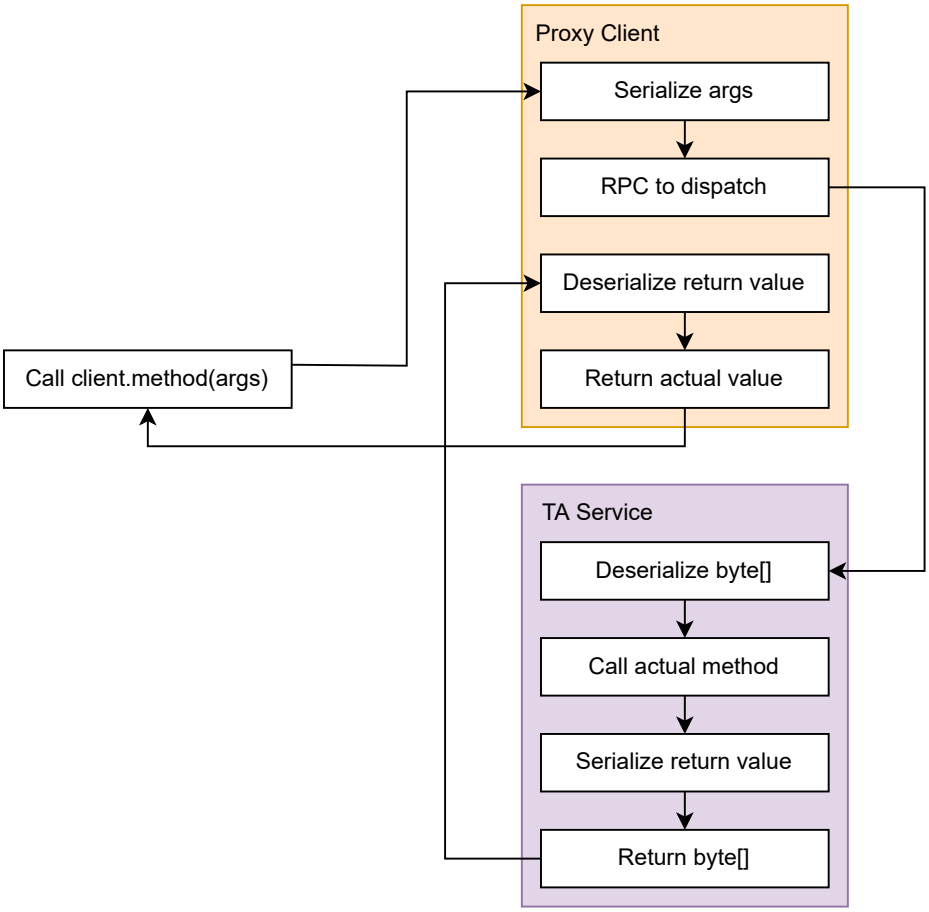

Figure 13: Proxy Call Workflow

4.13 Direct Circular dependencies

Circular dependencies are created when two functions or types are co-dependent and thus end up executing or instantiating each other in an endless loop. Given that this project extensively uses recursive descent parsers, circular dependencies are expected.

An example of such circular dependency is - an argument to a function call can be an expression, but an expression can also be a function call, which ultimately results in types and methods invoking each other. The official typescript guide [41] suggests using Type-only imports and exports. However, this does not provide any suggestions for co-dependent functions.

A commonly suggested solution to this to create indirect dependencies by creating an identical type or method. It should be noted that this is still a circular dependency. However, direct dependency has been eliminated and Typescript can now be compiled into JavaScript successfully.

This solution has also been employed in the project to create a copy of *Expressions* as *Expressions2* to eliminate direct circular dependency.

4.14 Enclavifying new methods

During the process of re-enclavification, where a new method is selected to be enclavified, there are two possible actions.

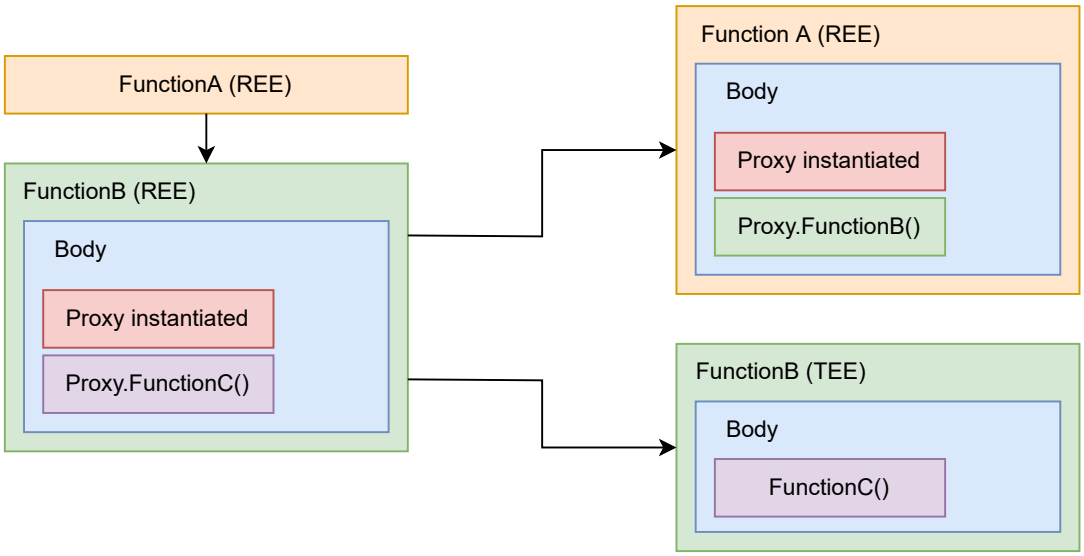

Figure 14: Enclavifying a method which already contains the Enclave's client's proxy

- **The new method does not call any enclavified method :** In this case, original workflow (section 4) can be followed.
- **The new method depends on previously enclavified method :** As represented in figure 14, to enclavify a method which was already dependent on previously enclavified method; firstly, proxy must be removed from the newly enclavified method and all references to the proxy must be replaced with regular method calls. Furthermore, proxy must now be injected in the new caller (Function A) using the same workflow as mentioned in section 4.

We recognize that the developer might need to perform some extra tweaks to the code of trusted application. This might come in the form of changing some lines of code of methods which were exported during the *enclavification* process. However, we do not wish the method to be overwritten during *re-enclavification*. Therefore, during the *re-enclvavification* process the system tries to respect that were made in the trusted application.

The figure 15 illustrates the process. During *re-enclavification*, the code analyzer also generates the abstract syntax tree (AST) of the main file for trusted application in enclaves. These ASTs are then augmented with the ASTs of new methods and their dependencies. The system attempts to locate the best place to insert these ASTs. The criteria is -

- If the same method is selected, it is to be overwritten in the trusted application.

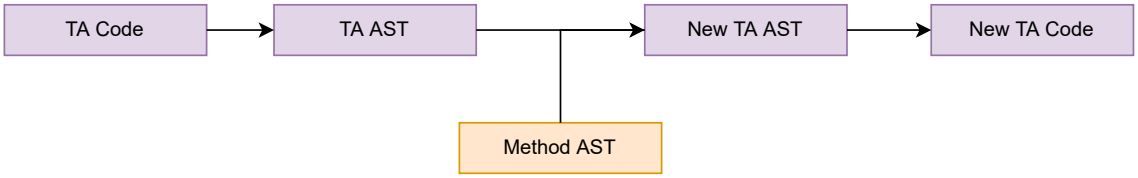

Figure 15: Respecting new changes during re-enclavification

- If a method or a group of methods are selected and they do not contain the previously enclavified, method, the system will attempt to insert those methods and their signatures as implementations and as parts of the traits.

This guarantees that the newly generated trusted application file contains all the expected methods and details.

4.15 Smarter Serialization and Deserialization

The project employs a smart serialization and deserialization system to convert any parameters passed to a method into a string while making a remote procedure call. Calling methods or sending data to trusted applications running inside enclaves require using byte buffers (which is just marshalled parameters).

However, while doing so, we risk creating security vulnerabilities [2]. Trusted applications, while running in trusted execution environment, can access memory of rich execution environment. Therefore, an ad hoc approach of naively marshalling a data structure into byte array, or serialization, creates a risk of having critical security vulnerabilities.

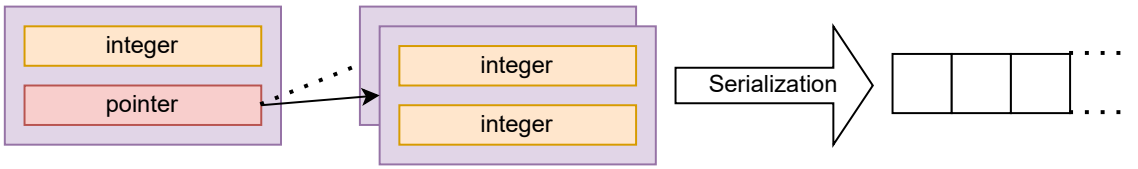

Figure 16: Smart Serialization Process

This project implements its own serialization protocol. This is illustrated in the figure 16. Members of data structures are serialized recursively and if a rust pointer is encountered, a copy of the value is created recursively and moved into enclave memory (section 2.11). A similar process is repeated in reverse during deserialization. Rust pointers which were used in the trusted application are sanitized by creating a copy of the values being pointed to in rich execution environment memory.

This creates a virtual boundary around the trusted application, so that even if the trusted application returns a pointer or parts of memory from the rich execution environment, it can no longer be manipulated - since it is a separate copy from the original value.

However, this does not guarantee that there can be no interactions between the trusted application and the host application. A malicious developer can design

the trusted application to still probe and manipulate the memory in rich execution environment. However, when the data (or pointer) is sent back to the host application, it can no longer be manipulated, even if the software is running at privilege. Another unavoidable shortcoming of this method is that a malicious developer can still attempt to hide pointers as unsigned integers and then interpret them as pointers at the trusted application level. It is impossible to determine whether an unsigned integer is actually a pointer in disguise or just some arbitrary value it is representing.

4.16 Usable Security : Error Handling and Recovery

We recognize that there is always a possibility that any of the operation can fail due to bad input, or the program being stuck in a bad state, or for some unforeseen use-case. There is always a risk associated with automated tools modifying an important file and as a result, the user might lose work. This project was built with usable security in mind and provisions certain degree of protection against some scenarios. This was done using some simple, yet effective error handing and recovery methods. Some of these scenarios are -

- **Method was not selected for enclavification :** In case some other rust programming construct was selected or no method was selected, the initial parsing step will fail. As a result, the scaffolding for trusted application will never be generated and the error will be displayed as a warning message in a Visual Studio Code popup. Note that the workflow to generate just the scaffolding for the trusted application is different from the one described above.
- **Parsing failed due to unknown programming construct :** While this project attempts to stay as true to the rust grammar as possible, there might be certain scenarios where the parser fails to recognize something. In such cases, an error message is shown in a pop-up and execution is halted. This behavior can be modified using a flag - *Parse unknown blocks* - which can potentially successfully parse such blocks. However, any context that might be associated with such blocks in original code is lost and the block is exported as it is, as part of the enclavified method. Another way this can be fixed is by implementing the parser for the correct constructs.
- **NoSTD crates for enclavified method :** The project expects the crate dependencies of the method being exported are NoSTD in nature. Trusted Execution Environments have their own OS called the trusted OS and these operating systems may not support all the functionalities that are expected from the OS residing in rich execution environment. Therefore, it is required that the crate dependencies are NoSTD in nature. Currently, the project exports these dependencies without vetting them for their nature. However, a feasible solution is proposed by this paper as part of direction this project can take in future.
- **Human error in enclavifying wrong method :** Instead of simply removing the method or removing the methods which depend the method getting

enclavified, this project attempts to re-insert old methods into the main host file as part of the comments. This also holds true for the cargo files. Any piece of code which is getting replace re-appears in the form of comments. The user may then restore these comments as actual code at a later time. These also serves as reference and can be considered part of usable security.

- **VSCode reaches at a bad state and cannot recover :** This project avoids inserting or editing any code until all the computation has been done. This means that, even if visual studio get stuck at a bad state, the entire process can be started again to arrive at the desired conclusion.

5 Evaluation

5.1 Execution of example code

For the purpose of this demonstration, we will be assuming that the extension is installed into VSCode. We intend on using a small and easy to understand code snippet as an example to demonstrate the entire workflow. Some non-significant details are also removed as this demonstration is to observe various code transformations. We will be using the code snippet 3. The method described accepts a single 32 bit integer parameter, but returns the number 321 as a 32 bit integer.

Listing 3: Method to be enclavified with main function containing its callsite

```
fn main()
{
    print("{}", foo(123));
}

fn foo(_num : i32) -> i32
{
    321
}
```

5.1.1 On executing command : Enclavify

The enclavify command can be accessed by using the command palette (*Ctrl + Shift + P* or *Ctrl + P* for the extended version) (section 4.1). On executing this command, following things happen in order

- Code Analyzer attempts to find crate dependencies and direct and transient dependencies. In this case, there is no direct or transient dependencies, nor there is any crate dependency.
- Code analyzer finds the callsite in the main function, and replaces it with a proxied call site. It also adds dependency for the new proxy. Code generator produces the code as represented in the code snippet 4.

  Listing 4: Main method changed with injected proxied callsite

  ```
  use hello_world::EnclaveClient;

  fn main()
  {
      let (runtime, transport) =
          hello_world_client::get_transport();
      let client = EnclaveClient::new(transport);

      print("{}", client.foo(123).unwrap());
  }
  ```

- Code analyzer transforms the method to be enclavified method into trait and its implementation. Their serialized versions is demonstrated in listing 5. This step also adds relevant macro on the trait. This is part of the trusted application. This step also adds the crate, transient and direct dependencies as part of the trusted application and its cargo file. However, in this scenario, we do not have any such dependencies.

Listing 5: Enclavified method transformed into trait and its implementation

```
#[EnclaveService]
pub trait Enclave
{
    fn foo(_num : i32) -> i32;
}

impl Enclave for Node
{
    fn foo(\&self, _num : i32) -> i32
    {
        321
    }
}
```

- Code Analyzer adds the dependency as part of host's cargo file so that it can be used as a crate. When serialized by code generator, the dependency is added as in snipped 6.

Listing 6: New dependency added in host's cargo file

```
[dependencies]
hello_world = "*"
```

- The next step adds the necessary scaffolding, which generates a structure as described in the section 4.2. This also includes the client proxy which is available in the same directory as the host main file. This step also adds the root cargo file, which connects the two applications together. Snippet 7 illustrates the root cargo.

Listing 7: New dependency added in host's cargo file

```
[workspace]
members = [
    "client/rust/",
    "module/rust/"
]

[patches.crates-io]
# some TA dependencies
hello_world_client = { path = "client/rust" }
```

```
hello_world = { path = "module/rust/" }
```

5.1.2 Macro Expansion

On compiling the project using either "build.sh" or by manually compiling (section 12), macros are expanded into rust code. This step also injects the expanded code at appropriate positions.

- On host end, the traits are expanded into a snippet shown in listing 8. This is responsible for serializing the original arguments, thus converting them into byte array. Then the proxy RPCs into the TA service and returns the response after deserialization.

Listing 8: Proxy client after macro expansion

```
impl <T: Transp> EnclaveClient<T>
{
    #[allow(unused)]
    pub fn foo(&self, _num: i32) -> Option<i32> {
        let request = EnclaveRequest::Foo{_num,};
        let ser = request.serialize();
        let resp =
            self.call("Enclave.foo", ser).unwrap();
        deserialize(resp)
    }
}
```

Listing 9: Trusted Application service after macro expansion

```
impl <T: Enclave> grpc::ServerNode for Dispatcher<T>
{
    fn invoke(&mut self, method: &str, req: &[u8],
        mut writer: grpc::ChannelResponseWriter)
    {
        let r: EnclaveRequest =
            Deserialize(req.get_value()).unwrap();

        EnclaveRequest::Foo { _num } =>
        {
            let res =
                EnclaveResp::Foo(self.0.foo(_num));
            serialize(res)
        }
    }
}
```

- Listing 9 demonstrates how traits and their implementations in the trusted application were expanded into a service. This service deserializes the request from client proxy. After calling the original method, it serializes the result and sends it as a response.

5.2 Takeaways

As discussed in the section 3, this project and this paper attempts to answer the following questions -

- How can we mitigate common security vulnerabilities related to communication between host application and trusted applications running in enclaves?

  We attempted to resolve this issue by serializing and deserializing the parameters in a smarter way. We tried to sanitize pointers by recursively copying the values the pointers are pointing to into enclave memory. This ensures that as long as the developer is not malicious and is using rust pointers, the resulting TA cannot access memory from REE.

  Unfortunately, this project fails to address that pointers can be hidden as unsigned integers which can again be interpreted as pointers at the trusted part of the application.

  Secondly, since the project relies on code generation, a malicious developer already has the access to the trusted application code and therefore, they can modify the TA code maliciously.

- How can we avoid writing boilerplate code and build trusted applications from minimum code as a minimal viable product?

  We attempted to resolve this issue by automating code generation. The user can either create a new and empty scaffold, which creates a minimum viable trusted application; or the user can select parts of an existing rust application to be carved into a trusted application, in which case, the boilerplate code will be generated.

- How can we convert parts of a normal rust application into a trusted application?

  We attempted to answer this question by developing extensive code analysis system 4.8). The system analyses direct and transient method dependencies along with crates required for code execution. The code generation part then utilizes these insights to generate a trusted application code.

- How can we improve the developer experience?

  We attempted to improve developer experience by providing a strong integration with Visual Studio Code. Instead of relying on commandline tools and trying to figure the appropriate way and syntax to utilize such tools, the developer can use a nicer user interface and interact with the system using Visual Studio Code's command palette which offers rich options.

Furthermore, the user can also expressively select parts of the applications that needs to be converted into a trusted application by simply highlighting pieces of code which needs to part of the TA. The extensive code analysis system handles direct and transient dependencies for the developer to provide an overall smooth experience.

6 Future work

While this project attempts to solve a problems as described in section 3, there are still other features that can be added, or quality of life changes that can be made, to improve the overall experience. Some of them are listed below.

6.1 Unification of templates

This project currently uses two different sets of files used as templates -

- **Base :** Used as minimum viable code to generate the trusted application and to generate parts of the scaffolding for the enclave.
- **Template :** Files that contains named slots, in which generated code can be slotted in to produce the main file for trusted parts of the application.

Instead of using two different sets, it is possible to use just the base files to be used as templates as well. It will work as follows in different scenarios of the workflow -

- In case of creating an empty scaffold (or minimum viable code for the trusted application), we can use the base as-is.
- In case of carving parts of a rust application to for an enclave, we can build other analyzers which converts the base files into ASTs and inserts the generated ASTs into appropriate position. Serialization of these ASTs should yield text equivalent to text generated using slotted templates.

In the end, this would further reduce the maintenance required by the extension.

6.2 Using compiled language

This project uses TypeScript (and by extension, JavaScript) to build the entire system. This is due to restrictions imposed by visual studio code as it uses TypeScript as its choice of language for extension development.

However, a more sophisticated setup is possible which uses some form of compiled language (like Java or C#) which exposes its various functionalities using REST APIs. The extension itself can just be a simple wrapper around this which acts as a bridge between visual studio code and the actual code running as a compiled language.

While Visual Studio Code (and by extension, ElectronJS) uses Google's V8 engine as its JavaScript engine [9] to generate just-in-time compiled code to get some performance, a better performance can be obtained by using a compiled language with stricter types (as TS types are lost while transpiling). Many compiled languages also offer compiling binaries ahead-of-time, which can further improve the start up time of the application.

TypeScript also suffers from circular dependency issues [12], as discussed in section 4.13. These issues, while not completely mitigated, can be handled in a much more feasible way in compiled languages.

Furthermore, implementing all the features in a separate entity from extension also decouples the actual functionality and the extension parts necessary for interaction with Visual Studio Code. This allows development of other wrappers for other environments based on the compiled binary. This means development of similar extension for other IDEs can now be built around a central codebase, which converts this from $m*n$ problem to $m+n$ problem, which further aids in terms of maintenance.

6.3 Test Cases

While being a well researched topic, building a parser is error prone. For reasons discussed in sections 2.8, it was decided that a custom parser was to be written instead of using other parser generator tools. However, this is an error prone endeavour and it is common to make or even commit breaking changes.

This can be avoided by covering essential parts of the project with proper unit and functional test cases. JavaScript and TypeScript world offers wide variety of different testing frameworks, such as Mocha or Jest.

In the end, adding test cases further contributes towards reliability of the project.

6.4 Using no-std libraries in Rust

Currently, this project relies on the user using no-std libraries in the original application from which TA has to be carved. This is because the dependency analyser (section 4.8) uses the crates found in the code analysis step as-is. This means that the generated enclave would simply fail to execute if the trusted OS does not support that functionality.

A better solution for this would be to provide a curated list of replacements for the most commonly used functionalities. This can be implemented as a simple dictionary or a hashmap. In case a crate is used which does not exist in the curated list, a warning message can be displaced using the pop-ups supported by Visual Studio Code.

This will aid greatly towards the reliability of the project and will enable the developers to automatically convert a wider array of code into trusted applications.

6.5 Complete Parser

Currently, this project supports a parser which does not completely conforms to the rust grammar as described in the official rust language documentation [42]. The current parser supports a significant, but smaller subset. The implementation allows for a easy future expansion of the parser which was implemented using parser combinators (section 4.5, 4.6).

This will aid greatly towards the reliability of the project and will enable the developers to automatically convert a wider array of code into trusted applications.

6.6 Packrat Parser

While Recursive Descent Parsers can support a wide array of grammars (LL(*)) due to their inherent backtracking capabilities, they are also infamous for their poor performance ($O(k^n)$). This can be alleviated by implementing Packrat Parsers [43] which allows parsing in linear time ($O(n)$) at the cost of higher memory usage (for memoization).

Packrat parsers are top-down parsers which are one of the valid implementations of PEGs (section 2.7.4) which utilizes dynamic programming to memoize results and thus provide better performance.

This gain in performance will ensure that bigger code bases can be parsed quickly and thus contribute towards a smoother development experience.

7 Final Remarks

This thesis attempted to explore usable security to answer the questions -

- How can we mitigate common security vulnerabilities related to communication between host application and trusted applications running in enclaves?
- How can we avoid writing boilerplate code and build trusted applications from minimum code as a minimal viable product?
- How can we convert parts of a normal rust application into a trusted application?
- How can we improve the developer experience?

We believe that we answered all of these questions appropriately. A system was built which automates code generation for a trusted application written in Rust. We also attempted to mitigate some common security vulnerabilities using smarter serialization and deserialization. Introduction of commonly overlooked bugs was minimized by generating the scaffolding required by a trusted application to run in a TEE and to communicate with its host application. Furthermore, adding an interface which allows expressively selecting parts of application to be transformed into parts of a trusted application adds to a smoother developer experience. Lastly, all of these experiences and features come together by providing a strong integration with the IDE - Visual Studio Code.

There are also some suggested improvements which can add to the quality and reliability of the project. They also provide an insight into a possible direction this project might take in future.

In the end, we believe this thesis adds a huge variety of quality of life features and experiences to trusted application development.

References

[1] J.-E. Ekberg, K. Kostiainen, and N. Asokan, "The untapped potential of trusted execution environments on mobile devices," *IEEE Security & Privacy*, vol. 12, no. 4, pp. 29–37, 2014.

[2] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Kruegel, and G. Vigna, "Boomerang: Exploiting the semantic gap in trusted execution environments." in *NDSS*, 2017.

[3] Intel, "Intel SGX Developer Guide." [Online]. Available: https://download.01.org/intel-sgx/sgx-linux/2.17/docs/Intel_SGX_Developer_Guide.pdf

[4] GlobalPlatform, "TEE Client API." [Online]. Available: https://globalplatform.org/specs-library/tee-client-api-specification/

[5] "Javascript by mdn," https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[6] "V8 javascript and webassembly engine," https://v8.dev/.

[7] J. K. Martinsen, H. Grahn, and A. Isberg, "Combining thread-level speculation and just-in-time compilation in google's v8 javascript engine," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 1, p. e3826, 2017. doi: https://doi.org/10.1002/cpe.3826 E3826 cpe.3826. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3826

[8] "Handlebars," https://handlebarsjs.com/.

[9] "Electronjs," https://www.electronjs.org/.

[10] "Ecmascript," https://262.ecma-international.org/11.0/.

[11] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu, "Safe: Formal specification and implementation of a scalable analysis framework for ecmascript," in *International Workshop on Foundations of Object-Oriented Languages (FOOL)*, vol. 10. Citeseer, 2012.

[12] G. Bierman, M. Abadi, and M. Torgersen, "Understanding typescript," in *ECOOP 2014 – Object-Oriented Programming*, R. Jones, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. ISBN 978-3-662-44202-9 pp. 257–281.

[13] "Typescript," https://www.typescriptlang.org/.

[14] G. Castagna and V. Lanvin, "Gradual typing with union and intersection types," *Proceedings of the ACM on Programming Languages*, vol. 1, no. ICFP, pp. 1–28, 2017.

[15] "Stack Overflow Developer Survey 2020, Most Popular Technologies." [Online]. Available: https://insights.stackoverflow.com/survey/2020#technology

[16] "Stack Overflow Developer Survey 2021, Most Popular Technologies." [Online]. Available: https://insights.stackoverflow.com/survey/2021#technology

[17] A. Del Sole and D. Sole, *Visual Studio Code Distilled*. Springer, 2019.

[18] "Why visual studio code," https://code.visualstudio.com/docs/editor/whyvscode.

[19] "Visual studio code api," https://code.visualstudio.com/api.

[20] "Visual studio code extensions guide," https://code.visualstudio.com/docs/editor/extension-marketplace.

[21] "The Chromium Project." [Online]. Available: https://www.chromium.org/Home/

[22] B. Cui, J. Li, T. Guo, J. Wang, and D. Ma, "Code comparison system based on abstract syntax tree," in *2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT)*, 2010. doi: 10.1109/ICBNMT.2010.5705174 pp. 668–673.

[23] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, & tools*. Pearson Education India, 2007.

[24] R. Frost, R. Hafiz, and P. Callaghan, "Modular and efficient top-down parsing for ambiguous left-recursive grammars," in *Proceedings of the Tenth International Conference on Parsing Technologies*, 2007, pp. 109–120.

[25] R. A. Frost, R. Hafiz, and P. Callaghan, "Parser combinators for ambiguous left-recursive grammars," in *International Symposium on Practical Aspects of Declarative Languages*. Springer, 2008, pp. 167–181.

[26] R. R. Redziejowski, "Parsing expression grammar as a primitive recursive-descent parser with backtracking," *Fundamenta Informaticae*, vol. 79, no. 3-4, pp. 513–524, 2007.

[27] ——, "Some aspects of parsing expression grammar," *Fundamenta Informaticae*, vol. 85, no. 1-4, pp. 441–451, 2008.

[28] J. P. Kimball, *The formal theory of grammar*. Prentice-Hall Englewood Cliffs, NJ, 1973.

[29] G. Hotz, "Normal-form transformations of context-free grammars," *Acta Cybernetica*, vol. 4, no. 1, pp. 65–84, 1978.

[30] S. A. Greibach, "A new normal-form theorem for context-free phrase structure grammars," *Journal of the ACM (JACM)*, vol. 12, no. 1, pp. 42–52, 1965.

[31] D. D. McCracken and E. D. Reilly, "Backus-naur form (bnf)," in *Encyclopedia of Computer Science*, 2003, pp. 129–131.

[32] K. Thompson, "Programming techniques: Regular expression search algorithm," *Communications of the ACM*, vol. 11, no. 6, pp. 419–422, 1968.

[33] P. W. Abrahams, "A final solution to the dangling else of algol 60 and related languages," *Communications of the ACM*, vol. 9, no. 9, pp. 679–682, 1966.

[34] B. Ford, "Parsing expression grammars: a recognition-based syntactic foundation," in *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2004, pp. 111–122.

[35] T. Parr, "ANTLR: ANother Tool for Language Recognition." [Online]. Available: https://www.antlr.org/

[36] ——, *The Definitive ANTLR 4 Reference*. The Pragmatic Bookshelf, 2013.

[37] "Gnu bison," https://www.gnu.org/software/bison/.

[38] OMTP, "Advanced Trusted Environment." [Online]. Available: http://www.omtp.org/OMTP_Advanced_Trusted_Environment_OMTP_TR1_v1_1.pdf

[39] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution." *Hasp@ isca*, vol. 10, no. 1, 2013.

[40] N. D. Matsakis and F. S. Klock, "The rust language," *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 103–104, 2014.

[41] "Type-only imports and exports," https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-8.html.

[42] "Official rust language documentation." [Online]. Available: https://doc.rust-lang.org/stable/reference/

[43] B. Ford, "Packrat parsing: simple, powerful, lazy, linear time, functional pearl," *ACM SIGPLAN Notices*, vol. 37, no. 9, pp. 36–47, 2002.