A Smart Contract Grammar to Protect IoT Firmware Updates using Hyperledger Fabric

Xinchi He
Tandy School of Computer Science
The University of Tulsa
Tulsa, OK, USA
xinchi-he@utulsa.edu

Rose Gamble
Tandy School of Computer Science
The University of Tulsa
Tulsa, OK, USA
gamble@utulsa.edu

Mauricio Papa
Tandy School of Computer Science
The University of Tulsa
Tulsa, OK, USA
mauricio-papa@utulsa.edu

***Abstract*—Securing firmware updates for IoT devices is a challenging undertaking because of their limited hardware resources. Most of the existing solutions are based on centralized architectures that may expose a single point of failure. Blockchain technology is largely accepted as a secure, robust and distributed platform for a number of different applications. This paper proposes the use of a blockchain platform to facilitate firmware updates for IoT devices. The distributed nature of the framework helps secure the firmware update process against single points of failure. In addition, the use of smart contracts can further strengthen the process by specifying firmware update conditions, verifying firmware update legitimacy, and protecting against potential cyber-attacks. Previous work in this area is extended by introducing a grammar and compiler to assist stakeholders in generating smart contracts. To validate the approach, a web-based prototype has been implemented to directly generate smart contracts in chaincode format that can be deployed in Hyperledger Fabric, an open source and permission–based blockchain framework. The grammar and compiler were evaluated for different use cases of the firmware update operations. Preliminary results with a prototype implementation show potential in simplifying the smart contract development process and reducing the amount of work needed to generate a working chaincode.**

***Index Terms*—Blockchain, Internet of Things, firmware updates, smart contract, grammar, Hyperledger Fabric**

I. Introduction

Firmware is non-volatile piece of software that resides on the hardware to provide basic device functionality [1]. Firmware updates are frequently pushed by device manufacturers to address security vulnerabilities, fix bugs, add new features, and support new standards and protocols [2]. IoT (Internet of Things) is an emerging paradigm with great potential for growth and adoption in different domains such as smart environments (homes, cities, buildings, offices), healthcare and transportation [3]. The term "Things" in this context refers to embedded devices (i.e objects, sensors and actuators) that are interconnected through the Internet [4].

Firmware updates for IoT devices are typically conducted through local or remote communication channels (using either wired or wireless connections) [5]. In wired IoT firmware updates, a serial (UART) connection is directly established between the IoT device and a host (for instance using a USB port). Then the firmware update binary is uploaded to the target IoT device. More commonly, IoT firmware updates, are often conducted through a wireless connection and are also called over–the–air (OTA) updates. An OTA firmware update is preferred because it does not require physical contact, it is cost-effective and convenient. However, there are two major challenges that need to be addressed to secure OTA firmware updates. The first one is associated with the limited computational power available in IoT devices [6]. The second is associated with the centralized nature of most firmware update solutions [7], [8], [9] and the fact that they may expose a single point of failure.

To address those challenges, we have proposed a blockchain-based solution [10]. This solution not only inherits the proven security properties of a well designed blockchain framework (by eliminating the single point of failure of other approaches) but it also adds a layer of intelligence through the use of smart contracts. Among others, use of blockchain allows participants to monitor all firmware update activities by querying the distributed immutable ledger, to specify firmware update conditions using smart contracts, and to deploy components that are more resilient to network failures and cyber-attacks. The proposed framework defines a layered architecture for the entire solution, utilizes smart contracts to verify the legitimacy of firmware updates pushed by vendors, and has been evaluated against MitM (man-in-the-middle) and DoS (denial of service) threat models.

Smart contracts are often used in blockchain frameworks to define the business logic required by the application domain [11]. Using smart contracts, it is possible to dynamically specify firmware update conditions (e.g. tasks IoT devices need to fulfill after the firmware updates) within the blockchain network. In other words, a smart contract enhances the firmware update process with extended functionality that includes the use of traditional certificate-based solutions [8]. Since our previous work targets a consortium involving different device manufacturers, the firmware update process had to be flexible enough to incorporate different requirements. This is achieved by using smart contracts. Re-writing a smart contract from scratch when a manufacturer needs to adopt a new update policy requirement is inefficient. Furthermore, different manufacturers may have different coding patterns and styles that may result in inconsistencies across a deployed solution.

To address this issue and to bring some uniformity across the framework, we propose an extension that relies on the use

978-1-7281-2530-5/19/ $31.00 ©2019 IEEE

of a grammar specifically designed to regulate IoT firmware updates. To be of practical use, a compiler designed for this grammar should be able to produce smart contracts that can be immediately deployed in an existing blockchain infrastructure. For instance, in cases where the Hyperledger Fabric is used, the expected outcome is a complete chaincode file that is ready to be deployed without any further end-user input. The main three tasks in our proposed extension are: (1) defining a firmware update specification grammar, (2) parsing grammar strings and (3) and generating a smart contract code workflow. The proposed grammar and associated compiler were validated with a prototype implementation that uses a Vue.js front-end GUI, a Golang back-end server and a compiler service. We evaluated both the grammar and associated compiler with different use cases of firmware update specifications. The smart contract file generated by the compiler was then deployed to a Hyperledger Fabric network for validation. Preliminary results show that the proposed solution is effective in generating smart contract files for securing the IoT firmware update process within Hyperledger Fabric.

II. Related Work

Numerous research efforts in the area of smart contracts are being conducted and new applications continue to be proposed. O'Connor [12] introduces an all-new language called Simplicity that has been designed specifically for blockchains. The language targets cryptocurrencies and blockchain applications without using loops and recursions. It also aims to improve existing problems in Bitcoin Script (e.g. limited arithmetic) and the Ethereum Virtual Machine (must have enough gas to cover worst-case use scenarios). However, Simplicity is designed as a low-level language that would require interpretation from high-level languages for more sophisticated applications. In other words, Simplicity, in its current form, cannot be directly applied to smart contract development.

Schrans et al. [13] propose a new smart contract language called Flint. The language is type-safe and capabilities-secure. As such, it aims to provide access control mechanisms and protect the system from unintentional loss of cryptocurrencies. Flint has three novel features: (i) ability to control Ethereum accounts making calls to sensitive functions, (ii) assets are protected from accidental creation, duplication and removal, and (iii) labels to declare functions with write privilege. Flint has been designed specifically for use in Ethereum. Flint can be seen as a variation of Solidity which still requires end-user programming for smart contract developments.

Zafar et al. [14] demonstrate Sol2js, a translator tool to aims to fully map Solidity smart contracts to JavaScript-based Hyperledger Fabric smart contracts. The authors claim that Sol2js is currently capable of translating 65-70% of the Solidity keywords, which shows significant promise in delivering smart contract cross-platform compatibility. However, Sol2js is limited to only one mapping, from Solidity to Hyperledger Fabric. Hyperledger Fabric now supports Java, JavaScript and Golang implementations of smart contracts. Golang is now the most popular option in the community and Javascript support may not be as strong (which may be a risk for Sol2js).

Zhang et al. [15] introduce a blockchain-based compliance model for multinational corporations (MNCs) that allows for inter-company transactions. Two tools are used to translate regulations into a smart contract that can be embedded in the blockchain. First, compliance rules, parameters, and enforcement policies are defined with a domain-specific language (DSL) tool. Then a smart contract translator is built to convert the assets defined in DSL template into chaincode in Hyperledger Fabric. However, the DSL is not generic nor powerful enough to be directly adopted into our proposed framework to specify firmware update conditions.

III. Approach

Our approach aims to extend the service layer of the previously proposed blockchain-based OTA IoT firmware update architecture [10]. This section describes a grammar and compiler designed to generate smart contracts that are compatible with the Hyperledger Fabric open-source permissioned blockchain framework [16]. Together, the grammar and compiler, enable IoT device manufacturers to generate smart contract files that articulate firmware update specifications using the proposed grammar.

The idea is to use the proposed grammar to specify firmware update requirements which are then fed into the compiler (Figure 1). Output from the compiler is a ready-to-use chaincode (a smart contract implementation in Hyperledger Fabric) file.

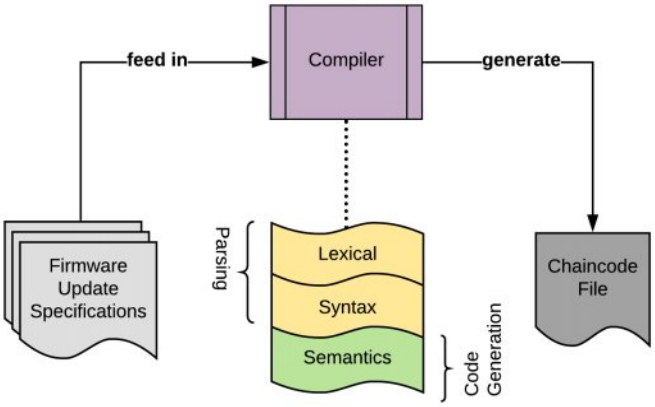

Fig. 1. Smart contract generation process overview

A. Grammar Overview

In developing the grammar, efforts were made to incorporate IETF (Internet Engineering Task Force) recommendations for IoT firmware updates [17]. These include multiple manifest fields that describe information (such as the size and type of the firmware update binary) needed for the firmware update process. It is worth mentioning that not all the IETF-recommended manifest fields are used in the proposed grammar. This omission was made because we needed to tailor it to our firmware update framework [10]. Our grammar only incorporates manifest fields and additional attributes that are

considered essential to allow a device manufacturer to specify firmware update policies and requirements. The grammar has five major sections describing requirements that support the workflow that IoT devices must follow from pre-update to post-update states: ***global***, ***pre-conditions***, ***transaction***, ***protection*** and ***post-conditions***.

The ***global*** section defines attributes that have a global scope within the smart contract. In particular, this section defines (i) the contract issuer, (ii) targeted devices, (iii) contract end-of-life and (iv) alliance entities that share use of the contract.

The ***pre-conditions*** section defines attributes that need to be validated prior to initiating the firmware update process. This section provides an opportunity to check minimum update requirements and dependencies (also known as pre-cursors in the IETF recommendations document). For instance, a device firmware may be outdated and the current incoming firmware update may not be compatible (which could crash or brick the device). Instead of applying firmware updates directly to target IoT devices, a pre-cursor compares itself with the existing firmware version on the device. The update will be aborted if the existing firmware does not meet the pre-cursor requirement. In addition, optional dependencies can be used to implement a gradual firmware update process (if needed). For example, `set pre_condition PRECURSOR 1.5 depend sequence by 1;` specifies the minimum current running firmware version while the dependency statement specifies incremental update steps for the IoT device. In this case, if the current version is older than 1.5, the incoming firmware update process will be aborted. Otherwise, the update process will gradually proceed by updating the minor version by one for each step (e.g 1.6→1.7, and then 1.7→1.8). While its use may not be common (e.g., without a depend construct, a device running version 1.5 could be directly updated to version 1.8), the construct would provide contract issuers with more granular control over the update process.

The ***transaction*** section defines attributes that must be included in a blockchain transaction. In our previous work [10], eight attributes (transaction id, timestamp send, timestamp receive, device type, firmware version, firmware hash, status, failed attempts) were defined. The goal of this section is to enable IoT device manufacturers to define blockchain transaction attributes in a scalable manner when deploying smart contracts for different use case scenarios. For example, `build transaction transaction_id type string order 1;` specifies that the blockchain attribute $transaction\_id$ is needed in the distributed ledger and that the data type must be string. The "order 1" statement indicates the index to be used when that parameter is later used in blockchain requests and activity queries.

The ***protection*** section defines threshold attributes designed to prevent cyber-attacks during the firmware update process. Two types of thresholds are defined: (i) time limit threshold to protect against a DoS attack and ensure update freshness and (ii) a failed attempts threshold to detect and prevent possible brute force attacks. For example, `set protection THRESHOLD.ATTEMPTS 5;` would limit the number of failed attempts to five.

The ***post-conditions*** section defines action attributes related with tasks that an IoT device is obligated to fulfill after a successful firmware update. Firmware updates are expected to be a common occurrence throughout the IoT device lifetime. As such, post-actions are defined to help ensure availability across multiple updates. For instance, they can be used to require the removal of residual information in on-board storage, to send a confirmation message back to the device manufacturer and to enable heartbeat message functionality (periodic messages). For example, `set post_condition HEARTBEAT ON weekly at 12:00 MON;` specifies a periodic call from the IoT device to the manufacturer.

B. ANTLR Grammar Implementation

ANTLR (ANother Tool for Language Recognition) [18] is a powerful Java-based language parsing tool. This tool helped implement our grammar using an ANTLR template. It can then be used to generate lexical and syntax analyzers in a selection of various target programming languages (e.g. Java, Python and Golang). A section of the ANTLR template defining the main sections of the proposed grammar can be seen in Figure 2.

The ***global*** section (lines 11-17), uses **set** (a reserved word in our proposed grammar) to define the **GLOBAL**, **ISSUER**, **EOL**, **RANGE** and **ALLIANCE** sections. **IDENTIFIER**, **IDENTIFIER_EOL**, **MAC** and **UUID** are regular expressions used to specify input formats. In ANTLR, `;` is a symbol identifying the end of a line, `|` is an option operation (an or) that provides alternatives for a section, and + defines a section that can appear one or more times. Line 12 defines allows the issuer of the smart contract to be assigned as either an **IDENTIFIER** (character string) or a **UUID** (Universally Unique Identifier) formats. Line 13 defines that the end-of-life (EOL) of the smart contract can be assigned using the **IDENTIFIER_EOL** format (mm/dd/yyyy). Line 14 helps describe the target of this firmware as a MAC address or a MAC address range or a combination of both. Line 15 defines the alliance of manufacturers that may share the same smart contract. This construct is useful when the same platform is used by different manufacturers. For example, the ESP8266 SoC is designed and manufactured by Espressif, but it is so popular that it has been adopted by various IoT boards manufacturers such as D1 Mini from Wemos and Feather HUZZAH from Adafruit. In this particular case, Espressifs alliances are Wemos and Adafruit. Thus, defining alliances enables Wemos and Adafruit to update ESP8266 boards with one smart contract initially deployed by Espressif. It is worth mentioning that Espressif, Wemos and Adafruit must all be participants within the same blockchain network.

In the ***pre-conditions*** section (lines 19-24), **PRE_CONDITION**, **PRE_CURSOR**, **DEPEND**, **SEQUENCE** and **by** are keywords. **IDENTIFIER_VERSION**, **INT** and **WS** are defined by regular expressions and they are used to specify the firmware version, the incremental

```
1   grammar contract;
2
3   top: contract* EOF;
4
5   contract: (global |
6             preCondition |
7             transaction |
8             protection |
9             postCondition)+;
10
11  global:
12      'set' GLOBAL ISSUER (IDENTIFIER | UUID) ';' |
13      'set' GLOBAL EOL IDENTIFIER_EOL ';' |
14      ('set' GLOBAL RANGE ( macRange | MAC) ';')+ |
15      'set' GLOBAL ALLIANCE alliances ';';
16  macRange: MAC '-' MAC;
17  alliances: (IDENTIFIER | UUID)+;
18
19  preCondition:
20      'set' PRE_CONDITION PRE_CURSOR
21          IDENTIFIER_VERSION ( WS | DEPEND dependencies)';';
22  dependencies: (sequence | discrete);
23  sequence: SEQUENCE 'by' INT;
24  discrete: '[' IDENTIFIER_VERSION+ ']';
25
26  transaction: 'build' TRANSACTION IDENTIFIER TYPE
27      (STRING | NUMBER) ORDER INT ';';
28
29  protection:
30      'set' PROTECTION THRESHOLD_TIME INT ';' |
31      'set' PROTECTION THRESHOLD_ATTEMPTS INT ';';
32
33  postCondition:
34      'set' POST_CONDITION LOCATION (operations)+ ';' |
35      'set' POST_CONDITION CONFIRMATION (ON | OFF) ';' |
36      'set' POST_CONDITION HEARTBEAT (ON conditions | OFF) ';';
37  operations: (SPIFFS | EEPROM) PURGE (ON | OFF);
38  conditions: (routine | interval);
39  routine: DAILY 'at' TIME |
40           WEEKLY 'at' TIME DAY;
41  interval: 'every' INT (MINUTE | HOUR);
```

Fig. 2. ANTLR grammar snippet

step and white space. Lines 20 and 21 define the pre-cursor of the firmware update with optional dependencies. The dependencies can be a collection of firmware versions specified as a continuous range (line 23) (defined by a sequence of step-sized increments) or a discrete collection (line 24).

In the ***transaction*** section (lines 26 and 27), **build**, **TRANSACTION**, **TYPE**, **STRING** and **NUMBER** are keywords. The transaction attribute then can be specified with a pair of attribute name and its data type. Currently, only string (character), double and integer (number) is supported. More complex data types will be incorporated in the future.

In the ***protection*** section (lines 29 and 31), **PROTECTION**, **THRESHOLD_TIME** and **THRESHOLD_ATTEMPTS** are keywords. **INT** is a regular expression for integer numbers.

In the ***post-conditions*** section (lines 33-41), **POST_CONDITION**, **LOCATION**, **CONFIRMATION**, **HEARTBEAT**, **ON**, **OFF**, **SPIFFS**, **EEPROM**, **PURGE**, **DAILY**, **WEEKLY**, **TIME**, **DAY**, **MINUTE**, **HOUR** are keywords. Configuration files and IoT device states are usually stored in SPIFFS and EEPROM. The residual information can be either retained or purged after a new firmware update. Line 34 is used to control the on-device storage after a firmware update. In addition, two active reporting mechanisms can be enabled through the smart contract. These mechanisms can be used to ask devices to (i) actively send confirmation messages after a successful firmware update (line 35) and/or (ii) send interval-based heartbeats indicating online status (line 36) back to device manufacturers directly. Even though manufacturers can later query the blockchain ledger to monitor and manage the firmware update status, these active responses from IoT devices provide first-hand insight for the status of a device as well as a mechanism for device manufacturers to engage their devices. Lines 38-41 define conditions under which a heartbeat message must be sent. IoT devices can send their heartbeats to device manufacturers either at defined intervals (e.g. every 5 minutes) or routinely at a pre-defined time (e.g. daily at 7 am or weekly at 9 pm Tuesday).

The railroad diagram (Figure 3) is a visual representation that helps understand the underlying syntax. It can be seen as a modern replacement of traditional Backus-Naur Forms (BNF) diagrams [19].

C. Parsing

The parsing process is used to retrieve attributes and their values from the firmware update specifications. The generated lexical analyzer provides listener callback functions that can be used for context retrieval. ANTLR makes it very easy to obtain a full parser tree from the grammar template.

Figure 4 shows a sample firmware update specification using the grammar defined in the previous section:

- The ***global*** section (lines 1-6) defines: the issuer, end-of-life of the smart contract, the targeted IoT devices, and partner device manufacturers that share the smart contract.
- The ***pre-conditions*** section (lines 8-9) defines the precursor of the firmware update as well as the dependency of certain firmware update versions.
- The ***transaction*** section (lines 11-15) defines several blockchain transaction attributes along with their data types.
- The ***protection*** section (lines 17 and 18) defines the numerical value of both time-based and count-based thresholds.
- The ***post-conditions*** section (lines 20-24) defines the location to purge residual storage, one-time confirmation as well as periodical heartbeat to a designate host.

Lines 8, 20 and 23 are used to show possible alternatives (they are commented out). It is important to note that comments are part of the proposed grammar (both single line and block comments). The actual mapping to smart contract generation semantics and implementation is described in the following sections.

A partial parse tree for the specifications can be seen in Figure 5. From top to bottom, they match the rules in lines 3 and 4, 15, 9 and 21 in Figure 4.

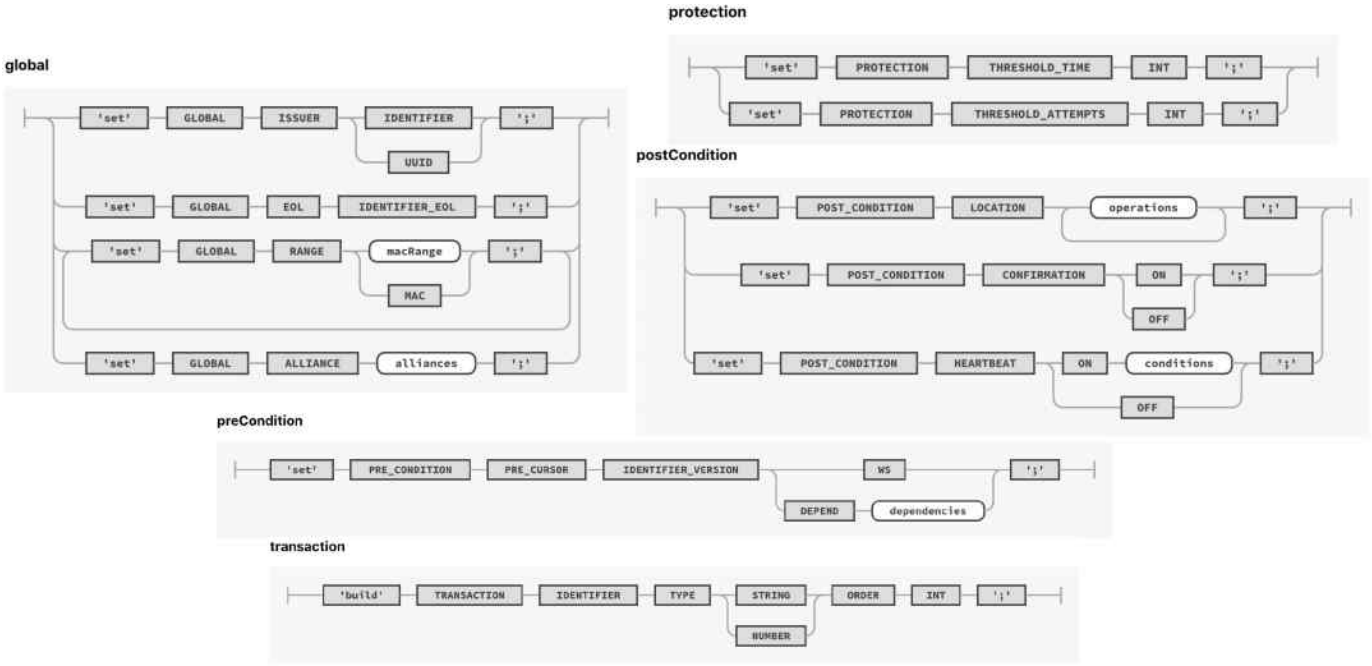

Fig. 3. Railroad diagrams

```
1   set global ISSUER d47f6632-87de-4d02-994a-0713b8cf8ebf;
2   set global EOL 8/8/2020;
3   set global RANGE 4d:02:01-4d:02:99;
4   set global RANGE 5e:02:01;
5   set global ALLIANCE 6c6df860-8623-429b-88f7-3a4ffdbb3702
6       7ec12033-f0f9-41cd-8357-1de545382fca;
7
8   //set pre_condition PRECURSOR 1.5 depend sequence by 1;
9   set pre_condition PRECURSOR 1.5 depend [1.8 2.0];
10
11  build transaction transaction_id type string order 1;
12  build transaction device_type type string order 3;
13  build transaction firmware_version type string order 4;
14  build transaction timestamp type string order 2;
15  build transaction manufacturer type string order 5;
16
17  set protection THRESHOLD.TIME 3600;
18  set protection THRESHOLD.ATTEMPTS 5;
19
20  //set post_condition LOCATION SPIFFS purge ON EEPROM purge ON;
21  set post_condition LOCATION SPIFFS purge ON EEPROM purge OFF;
22  set post_condition CONFIRMATION ON;
23  //set post_condition HEARTBEAT ON weekly at 12:00 MON;
24  set post_condition HEARTBEAT ON daily at 23:00;
```

Fig. 4. Sample firmware update specifications

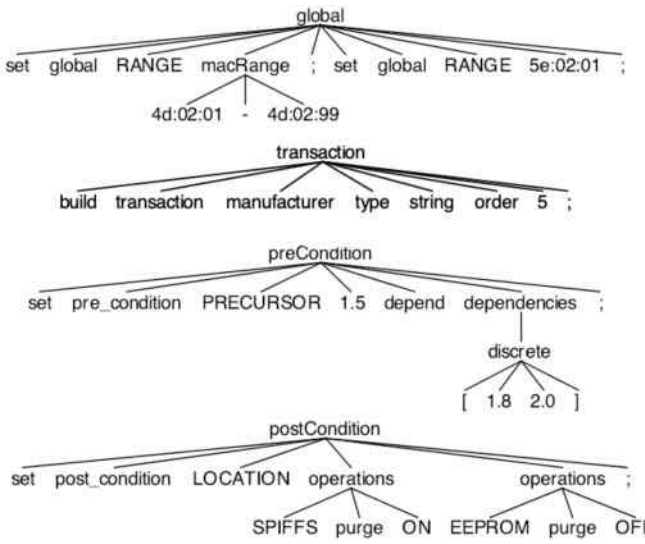

Fig. 5. Partial leaf nodes on parse tree

D. Chaincode Generation Compiler

In Hyperledger Fabric, chaincode is the smart contract that runs in an isolated secured Docker container. Chaincode manages the states in the distributed ledger through blockchain transactions. Chaincode can be seen as general Golang code using Hyperledger Fabric packages, such as *shim* and *peer*. The *shim* package provides APIs to access state variables, transaction context and calling other chaincodes. The *peer* package defines blockchain peer node information associated with RPCs and utilities.

The chaincode always includes a header section that declares the package name, imports the associated libraries, and defines data types. The `Init()` function is a fixed constructor to initialize the chaincode. The `Invoke()` is an index and navigation function to reference other sub-functions. The `main()` function must always be present as it represents the entry point for the chaincode. The remaining section contains customized functions that need to be generated by the semantics. Figure 6 shows how the main elements of a barebones chaincode template (left half) maps to a snippet of actual chaincode written in Golang (right half).

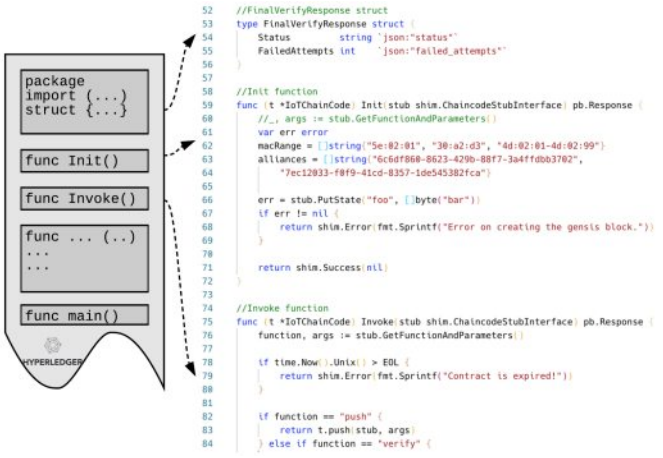

Fig. 6. Chaincode template (left) and sample chaincode (right)

The ANTLR grammar template defined in Section B helps create a compiler from the generated the syntax and lexical analyzers. The following procedure (Figure 7) is enforced by the compiler to produce the final output with the inputs retrieved from the associated parse tree.

(a) overwrite keyword and literal listener functions in the lexical analyzer to retrieve assets by iterating through the parse tree
(b) store the extracted assets into a dictionary with complex data structures (map with interfaces in Golang)
(c) break the minimal working firmware update chaincode into multiple snippet files and place anchor points
(d) iterate through the snippet files looking for anchor points in sequence and generate code as defined by the semantics
(e) merge all code snippet files and finalize the smart contract file generation process

Once the smart contract generation is finalized, a smart contract file with .go extension is created. The blockchain network administrator of the device manufacturer consortium will be able to directly deploy the smart contract to the network.

IV. Implementation

This section describes the implementation of a proof-of-concept system to generate chaincode. The system has three major components as shown in Figure 8: (i) a Vue.js based graphical user interface (GUI), (ii) a Golang based back-end server with RESTful APIs for integration, and (iii) a compiler to generate chaincode from firmware update specifications.

The chaincode generation process starts with specifying the firmware update conditions using the defined grammar. Then the graphical interface passes the specifications to the compiler through the backend server. Once the process is completed, the user can directly download the generated chaincode using a web browser.

A. Front-End GUI

Vue.js is a powerful JavaScript framework for front-end development [20]. It has been adopted by a number of websites,

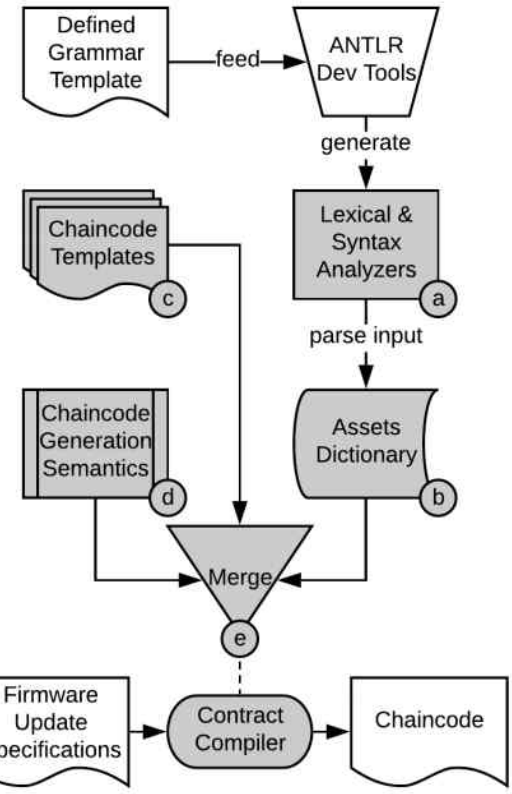

Fig. 7. Chaincode generation workflow

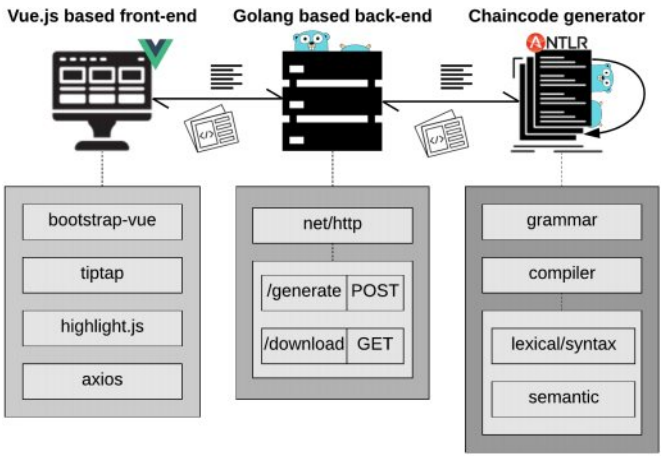

Fig. 8. Implementation overview

such as Gitlab and Netflix. Vue has a small footprint which is good for performance and flexible service integration.

Four external libraries are used in building the front-end of the proof-of-concept system: (1) *bootstrap-vue* for form and button CSS, (2) *tiptap* for in-browser text editing, (3) *highlight.js* for highlighting keywords and literals in our defined grammar, and (4) *axios* for managing HTTP requests.

Figure 9 displays the implemented front-end component in a Safari browser. The front-end component exposes a text editor and, initially, it displays the firmware update specifications in

Figure 4. Once a user is done editing the file, the following procedure is needed to finalize the chaincode generation:

1) Click the "Save" button to preserve the firmware update specifications in the text editor
2) Click the "Generate" button to pass the specification to the compiler
3) Click the "Download" button to fetch the generated chaincode from the back-end server

It is worth mentioning that the "Reset" button will clear any changes in the text editor and revert the specifications to the default one.

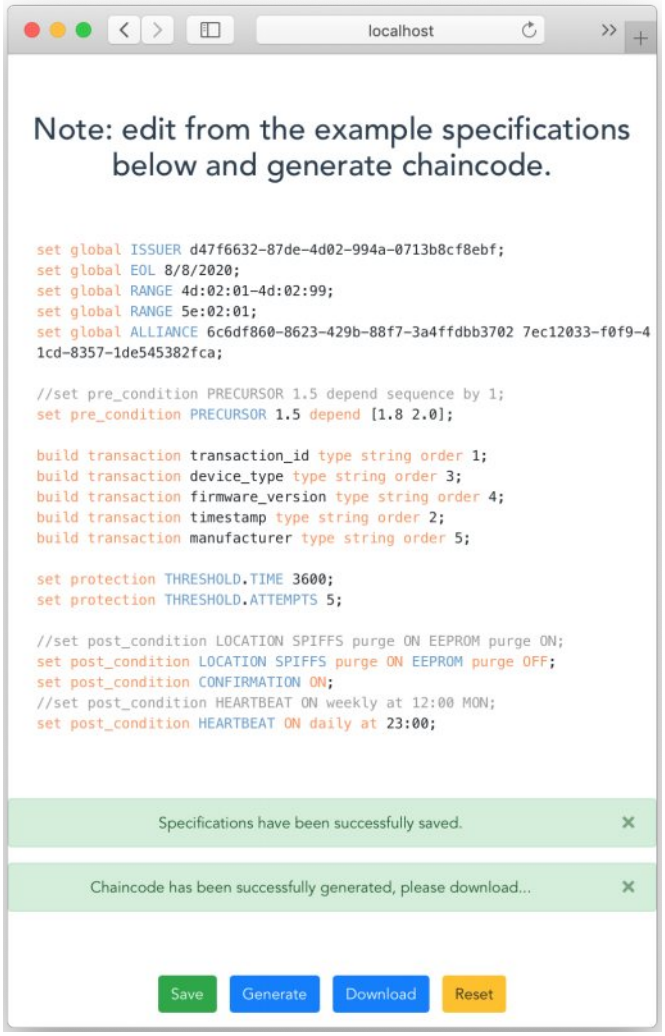

Fig. 9. Front-end GUI

B. Backend Server

The role of the back-end server is to provide the necessary glue to connect the front-end GUI and the chaincode generation compiler. The Golang *net/http* library is used to provide two API calls (center of Figure 8): (i)/*generate* handles HTTP POST requests of incoming firmware update specifications from the front-end GUI. The back-end server saves the specifications and feeds it to the compiler. (ii)/*download* processes HTTP GET requests from the front-end GUI, enable in-browser direct file downloading for the generated chaincode file.

C. Compiler

The compiler is the main engine of the entire prototype system. It receives firmware update specifications from the front-end GUI and responds back with generated chaincode. Golang structs were defined for each section in the grammar. Then a `map[string]interface{}` type of dictionary is used to store retrieved assets from the parser tree.

Based on the chaincode template layout (Figure 6 left) and smart contract functions used for firmware update verification, a minimal firmware update chaincode is partitioned into nine different snippet files:

- *header.snippet*, *init.snippet*, *invoke.snippet*, and *main.snippet* for the fixed layout of Hyperledger Fabric chaincode
- *push.snippet*, *verify.snippet*, *update.snippet*, and *query.snippet* for the firmware update verification
- *helper.snippet* is a helper snippet code that assists in specifying and validating device MAC ranges

Then code sections in each snippet file is replaced with the corresponding generated code. Figure 10 shows an example of the generated code in *verify.snippet* and the semantics needed to generate firmware update precursor codes in the compiler.

```
if IsInRange(macAddress) == false {
    return shim.Error(fmt.Sprintf(
        "Device is not eligible to update."))
}

//pre_conditions_anchor

transactionBytes, err := stub.GetState(transactionID)
if err != nil {
    return shim.Error(
        fmt.Sprintf(
            "Error on retrieving state " + transactionID + "."))
}
```

```
} else if strings.Contains(
    verifyScanner.Text(), "//pre_conditions_anchor") {
    preCursor := dict["pre_condition"].(
        PreConditionAsset).PreCursor
    output.WriteString(
        "\t\tif currentVersion < " + preCursor + " {\n")
    output.WriteString(
        "\t\treturn shim.Error(fmt.Sprintf(\"Firmware is
        too old, does not meet update pre-conditions!\"))\n")
    output.WriteString("\t}\n\n")
```

Fig. 10. Generated snippet code (top) and semantics (bottom)

V. Experimental Results

In order to validate the proposed grammar and compiler, the implemented prototype was evaluated against different use cases. We started with testing the firmware update specifications in Figure 4. The result showed that the prototype system

was able to generate 402 lines of firmware update chaincode from only 16 lines of specifications.

Four more test cases were also evaluated: (i) reproduce firmware update chaincode that was used in our previous work [10], (ii) chaincode with various transaction attributes, (iii) chaincode that can validate minimum version requirement for the firmware updates, and (iv) chaincode with post firmware updates heartbeat callbacks.

The generated chaincode in each case was validated by deploying to the Hyperledger Fabric blockchain network.

A. Case 1: Reproducing Previous Use Case

The first use case starts with generating the chaincode that had been used in our previous work. For such a basic use case, the MAC address of only one device was specified (line 1). Only minimum attributes (lines 3-6) along with protection thresholds (lines 8 and 9) were defined.

```
1  set global RANGE a0:12:b4;
2
3  build transaction transaction_id type string order 1;
4  build transaction timestamp type string order 2;
5  build transaction device_type type string order 3;
6  build transaction firmware_version type string order 4;
7
8  set protection THRESHOLD.TIME 60;
9  set protection THRESHOLD.ATTEMPTS 3;
```

By using the specification above, a chaincode file with 381 lines was successfully generated.

B. Case 2: Multiple Transaction Attributes

In this use case scenario, we test against a large number of potential transaction attributes that device manufacturers may want to store in the blockchain distributed ledger.

MAC addresses of multiple IoT devices were specified by a range (line 1) with protection thresholds (lines 14 and 15). Ten sample blockchain transaction attributes (lines 3-12) were also defined with different data types and orders for later use.

```
1  set global RANGE b1:00:21-b1:00:f0;
2
3  build transaction transaction_id type string order 1;
4  build transaction timestamp type string order 2;
5  build transaction device_type type string order 3;
6  build transaction firmware_version type string order 4;
7  build transaction foobar_alpha type number order 5;
8  build transaction foobar_bravo  type number order 6;
9  build transaction foobar_charlie type number order 7;
10 build transaction foobar_delta type string order 8;
11 build transaction foobar_echo type string order 9;
12 build transaction foobar_foxtrot type string order 10;
13
14 set protection THRESHOLD.TIME 600;
15 set protection THRESHOLD.ATTEMPTS 10;
```

By using the specification above, a chaincode file with 393 lines was successfully generated.

C. Case 3: Conditional Updates

The minimal required firmware version for incoming updates is tested in this use case. Similar specifications were used, except for an additional entry to set the minimum firmware version to 2.0 (line 4)

```
1  set global RANGE b1:00:21-b1:00:f0;
2  set global EOL 12/31/2025;
3
4  set pre_condition PRECURSOR 2.0;
5
6  build transaction transaction_id type string order 1;
7  build transaction firmware_version type string order 2;
8
9  set protection THRESHOLD.TIME 600;
10 set protection THRESHOLD.ATTEMPTS 10;
```

By using the specification above, a chaincode file with 388 lines was successfully generated.

D. Case 4: Post Update Callbacks

The last use case scenario tests generated chaincode with post-conditions, which are operations the IoT device must perform after a successful firmware update event. The post-conditions satisfy the following requirements: (i) preserve local SPIFFS and EEPROM contents (line 9) and (ii) send back a confirmation message but not a heartbeat (lines 10 and 11).

```
1  set global RANGE a0:12:b4;
2
3  build transaction transaction_id type string order 1;
4  build transaction firmware_version type string order 2;
5
6  set protection THRESHOLD.TIME 600;
7  set protection THRESHOLD.ATTEMPTS 10;
8
9  set post_condition LOCATION SPIFFS purge OFF EEPROM
       purge OFF;
10 set post_condition CONFIRMATION ON;
11 set post_condition HEARTBEAT OFF;
```

By using the specification above, a chaincode file with 378 lines was successfully generated.

When post-conditions are present, the blockchain sends a JSON response that describes them to the IoT device In this particular case, the JSON `{"confirmation":true, "heartbeat":{"frequency":"", interval:"", "moment":"", "state":false}, "purge":{"EEPROM":false, "SPIFFS":false}}` was returned to the IoT device after a successful firmware update. Reception of the JSON object ensures the target IoT device will execute the post-update tasks.

VI. Conclusions and Future Work

This paper presents a grammar and compiler used to extend the service layer of an IoT over-the-air firmware update architecture [10]. They both are designed to facilitate smart contract generation from firmware update specifications. A prototype system with three major components was implemented: a front-end GUI, a back-end server and a compiler service. Furthermore, smart contract generation was validated by using five different use case scenarios. For each one of them, we verified that the chaincode could be immediately deployed to Hyperledger Fabric without any changes. Preliminary results show that the proposed approach is effective in simplifying the smart contract development for securing the IoT firmware update process using Hyperledger Fabric. In most cases, less than 20 lines of firmware update specification generate chaincode with approximately 400 lines.

Our future work will focus on (i) expanding the grammar to cover more use cases, (ii) integrating our prototype with a management system to deploy and instantiate generated smart contracts to the blockchain network, and (iii) utilizing model checking and formal methods to verify the legitimacy of the generated smart contracts.

REFERENCES

[1] C. J. Tan, J. Mohamad-Saleh, K. A. M. Zain and Z. A. A. Aziz, Review on Firmware, in Proceedings of the International Conference on Imaging, Signal Processing and Communication (ICISPC 2017), Penang, Malaysia, 2017, pp. 186-190.

[2] J. Kim and P. H. Chou, Energy-Efficient Progressive Remote Update for Flash-Based Firmware of Networked Embedded Systems, in ACM Transactions on Design Automation of Electronic Systems (TODAES), vol.16, no.1, pp. 1-26, November 2010.

[3] A. Giri, S. Dutta, S. Neogy, K. Dahal and Z. Pervez, Internet of Things(IoT): A Survey on Architecture, Enabling Technologies, Applications and Challenges, in Proceeding of the 1st International Conference on Internet of Things and Machine Learning (IML '17), Liverpool, United Kingdom, 2017, pp.1-12.

[4] T. L. Koreshoff, T. Robertson and T. W. Leong, Internet of Things: a review of literature and products, in Proceeding of the 25th Australian Computer-Human Interaction Conference: Augmentation, Application, Innovation, Collaboration (OzCHI '13), Adelaide, Australia, 2013, pp. 335-344.

[5] C. E. Andrade, S. D. Byers, V. Gopalakrishnan, E. Halepovic, M. Majmundar, D. J. Poole, L. K. Tran and C. T. Volinsky, Managing Massive Firmware Over-The-Air Updates for Connected Cars in Celluar Networks, in Proceedings of the 2nd ACM Internation Workshop on Smart, Autonomous, and Connected Vehicular Systems and Services (CarSys '17), Snowbird, Utah, USA, 2017, pp. 65-72.

[6] M. N. Aman, K. C. Chua and B. Sikdar, Secure Data Provenance for the Internet of Things, in Proceedings of the 3rd ACM International Workshop on IoT Privacy, Trust, and Security (IoTPTS '17), Abu Dhabi, United Arab Emirates, 2017, pp. 11-14.

[7] B.-C. Choi, S.-H. Lee, J.-C. Na and J.-H. Lee, Secure Firmware Validation and Update for Consumer Devices in Home Networking, in IEEE Transactions on Consumer Electronics, vol.62, no.1, pp. 39-44, February 2016.

[8] L. Kvarda, P. Hnyk, L. Vojtech and M. Neruda, Software Implementation of Secure Firmware Update in IoT Concept, in Advances in Electrical and Electronic Engineering, vol.15, no.4, pp. 626-632, November 2017.

[9] M. A. Prada-Delgado, A. Vzquez-Reyes and I. Baturone, Trustworthy Firmware Update for Internet-of-Things Devices using Physical Unclonable Functions, in Proceedings of 2017 Global Internet of Things Summit (GIoTs), Geneva, Switzerland, 2017, pp. 1-5.

[10] X. He, S. Alqahtani, R. Gamble and M. Papa, Securing Over-The-Air IoT Firmware Updates using Blockchain, in Proceedings of the International Conference on Omni-Layer Intelligent Systems (COINS'19), Crete, Greece, 2019, pp. 164-171.

[11] S. Wang, L. Ouyang, Y. Yuan, X. Ni, X. Han and F. Wang, Blockchain-Enabled Smart Contracts: Architecture, Applications, and Future Trends, in IEEE Transactions on Systems, Man, and Cybernetics: Systems, 2019, pp. 1-12.

[12] R. O'Connor, Simplicity: A New Language for Blockchains, in Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security, Dallas, Texas, USA, 2017, pp. 107-120.

[13] F. Schrans, S. Eisenbach, and S. Drossopoulou, Writing safe smart contracts in Flint, in Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, Nice, France, 2018, pp. 218-219.

[14] M. A. Zafar, F. Sher, M. U. Janjua, and S. Baset, Sol2js: Translating Solidity Contracts into Javascript for Hyperledger Fabric, in Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers, Rennes, France, 2018, pp. 19-24.

[15] W. Zhang, Y. Yuan, Y. Hu, K. Nandakumar, A. Chopra and A. De Caro, Blockchain-Based Distributed Compliance in Multinational Corporations Cross-Border Intercompany Transactions, in Advances in Information and Communication Networks, vol. 887: Springer, 2018, pp. 304-320.

[16] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C.Stathakopoulou, M. Vukolić, S. W. Cocco and J. Yellick, Hyperledger Fabric: a Distributed Operating System for Permissioned Blockchains, in Proceedings of the Thirtenth EuroSys Conference (EuroSys '18), Porto, Portugal, 2018, pp. 1-15.

[17] B. Moran, M. Meriac and H. Tschofenig, A Firmware Update Architecture for Internet of Things Devices (2018), [Online]. Available: https://tools.ietf.org/id/draft-moran-suit-architecture-02.html

[18] T. Parr, The Definitive ANTLR 4 Reference, Pragmatic Bookshelf, 2013.

[19] L. M. Braz, Visual Syntax Diagrams for Programming Language Statement, in ACM SIGDOC Asterisk Journal of Computer Documentation, vol.14, no. 4, pp. 23-27, 1990.

[20] H. Djirdeh, N. Murray and A. Lerner, Fullstack Vue: The Complete Guide to Vue.js, CreateSpace Independent Publishing Platform, 2018.