Probabilistic modelling and attack simulations on AWS Connected Vehicle Solution: An Application of the Meta Attack Language

LOVE ALMGREN

JOHAN HOLM ÅSTRÖM
Probabilistisk modellering och attacksimuleringar på AWS Connected Vehicle Solution: En tillämpning av Meta Attack Language

DD142X Examensarbete inom datalogi
Handledare: Robert Lagerström

Johan Holm Åström & Love Almgren
930203 - 5135 & 980714- 6213
johaast@kth.se & loveal@kth.se
Abstract

This work is focused on investigating if the Meta Attack Language (MAL) can be used to create an integrating layer between two different applications of the MAL, and thus being able to model a new domain. In this case vehicleLang and awsLang were chosen as candidate applications of the MAL. While the domain chosen was to model the AWS Connected Vehicle Solution Infrastructure. This domain therefore modelled a service that is quickly becoming popular among car manufacturers. The two languages were successfully compiled into one language using the MAL, and also able to model a leak within AWS that could potentially lead to greater exposure of the infrastructure as a whole. On the other hand some limitations in the MAL compiler has lead to suggestions of how to improve it for better support of integration of different MAL applications.

Sammanfattning

Contents

1 List of Definitions .................................................. 4

2 Introduction ......................................................... 5
   2.1 Problem statement ......................................... 6
   2.2 Scope of the work ........................................ 6

3 Theoretical Background ........................................... 8
   3.1 Attack Graphs: formalism for threat modeling ....... 8
   3.2 The MAL syntax ........................................... 10

4 Related Work ....................................................... 12
   4.1 Threat modeling languages ............................... 12
   4.2 Vehicle cyber attacks ..................................... 12
   4.3 coreLang .................................................. 13

5 Application of MAL in vehicleLang ......................... 14
   5.1 Overview of the project .................................. 14
   5.2 Components in vehicleLang ............................. 14

6 Application of MAL in AWSLang ............................... 16
   6.1 Overview of the project .................................. 16
   6.2 Components in AWSLang ................................ 16

7 AWS Connected Vehicle Infrastructure ...................... 18
   7.1 Overview .................................................. 18
   7.2 How to connect vehicles to AWS? ..................... 18
   7.3 AWS components ....................................... 20

8 Method ............................................................... 22
   8.1 The process of integration ............................... 22
   8.2 Choice of testcases ...................................... 22
   8.3 Test evaluation ......................................... 24

9 The awsVehicleLang meta model .............................. 25
   9.1 Integrating classes ....................................... 25
   9.2 How to use awsVehicleLang ............................ 27

10 Result ............................................................. 29
   10.1 The model's performance ............................... 29
10.2 The compromised parts of the connected vehicle and the AWS network 30

11 Evaluation 31
11.1 Usefulness of the MAL 31
11.2 Struggles 31

12 Conclusions 33
A awsVehicleLang integration code 35
B The testcase for Honda leak 37
1 List of Definitions

**DSL** Domain-Specific Language. Programming languages created for use in a specific domain, for example HTML is a DSL for web sites.

**IoT** Internet of Things

**Smart city** A city that takes use of IoT devices to gather data about events and surroundings in order for resources such as power and water to be used efficiently and where needed

**Telematics** Data from vehicles that cover things such as GPS navigation, engine information and fuel use.

**Connected car** A car that is connected with internet access in order to send telematics over internet.

**MAL** Meta Attack Language

**coreLang** A DSL created with use of MAL that can be used by other applications of MAL in order to ease the development.

**AWS** Amazon Web Services

**ECU** Engine Control Unit

**awsLang** a DSL created using MAL which has the domain of AWS network and some of it’s key services.

**vehicleLang** a DSL created using MAL which has the domain of a connected vehicle.
2 Introduction

IoT devices are becoming increasingly popular and as of 2017 8.4 billion connected devices are in use, and this figure is estimated to rise to 20.4 billion by 2020\(^1\). This means that IT systems and devices are becoming more connected to each other. According to a survey by Gallup\(^2\), Americans are more afraid of being hacked on their smartphone or computer than any other form of crime. Therefore this poses the question whether the increased connectivity between devices increases the risk of being hacked, and according to Lagerström et al., 2017 the complexity of an IT system is a significant factor leading to security risks.

Connected vehicles are a type of IoT device which Gartner\(^3\) predicts there will be 250 million of them by 2020. Recently there has been many attacks aimed at connected vehicles, both by criminals with malicious intent, so called black-hat hackers and research groups, so called white-hat hackers. One example of such an attack performed by white-hat hackers made the engine of a Jeep Cheroke stop while it was driving on a highway by connecting to its entertainment system through cellular connection. This attack forced Fiat Chrysler to recall 7 of their models, about 1.4 million vehicles in total. This is only one of many attacks which demonstrates their dangers and costs. Moreover the number of malicious hacks by black-hat hackers has now eclipsed the number of hacks performed by white-hat hackers, and the number of attacks against connected vehicles has increased by six times over the last four years (Security, 2019).

The most common attacks against connected vehicles are server attacks (Security, 2019). An example of such an attack happened in 2018 against Tesla Motors, where black-hat hackers broke into one of their Amazon Web Services Server to 'mine' cryptocurrency and exposed private data belonging to Tesla's customers\(^4\).

---


\(^4\)fortune.com (2018). “Tesla Hackers Hijacked Amazon Cloud Account to Mine Cryp-
AWS is a cloud computing platform which allows customers to rent virtual computer clusters. They also provide a foundation for connected vehicles called AWS Connected Vehicle Solution which include tools for autonomous driving, electric drive, shared mobility and much more. Amazon has released a guide to how this should be implemented\(^5\). Therefore the increased complexity of connected vehicles with AWS should be investigated to find the threats, in order to be able to ensure the safety of drivers against hacks.

The Meta Attack Language (MAL) is a newly developed domain specific language (DSL) for probabilistic threat modelling and attack simulations (Johnson, Lagerström, and Ekstedt, 2018). At the moment MAL has been used to model cyber attacks on AWS and vehicles separately in AWSLang and vehicleLang. This work will use the applications of MAL in vehicleLang and AWSlang in order to create an integrating layer that connects the two. The idea behind creating this as a DSL is that it can be reused by any manufacturer of AWS connected vehicles with their knowledge on the specific security specifications of that car and AWS application. Moreover by showing that it is possible to connect two other domains of MAL can contribute to a greater application of MAL that has a smartcity as domain.

### 2.1 Problem statement

Does an integrating layer between vehicleLang and awsLang written using MAL potentially expose a new set of vulnerabilities for the AWS connected vehicle solution?

### 2.2 Scope of the work

This work will aim to create a DSL for modelling attacks on connected cars using AWS only as a cloud service. There is a possibility that there can be many other cloud services available for connected cars but AWS is the most common cloud service available. Only the MAL will be considered for creating the attack graphs in this report because it creates a language that users should easily be able to use. Also there are good resources to

\(^{\dagger}\)Amazon (2017d). “Introducing the AWS Connected Vehicle Solution”. In:
help with the use of MAL here at KTH. This report will focus on creating attack graphs that show the connection between AWS and the car, and will therefore not aim to model the full extent of attacks within the car or AWS cloud done by vehicleLang and awsLang respectively. The attacks that this report cover can also include cyber attacks which first target a users phone or computer connected to the same AWS connected vehicle solution.
3 Theoretical Background

This section is meant to explain the formalism for threat modeling and then explain the syntax of MAL.

3.1 Attack Graphs: formalism for threat modeling

In order to facilitate the possibility of threat modeling there is a need to formalise it, the usefulness of attack graphs has been presented by (Sheyner et al., 2006). The creation of attack graphs involve three steps: modelling the system, creating an attack graph using attack steps and analysis of the attack graph. The analysis is most often performed based on two metrics if there is post compromise service availability or the time to compromise (Johnson, Lagerström, and Ekstedt, 2018). Moreover for attack graphs to give a more casual relationship between vulnerabilities in a attack graph it is proposed that probabilistic metrics should be used (Wang et al., 2008), although a metric such as the common vulnerability score system which is most commonly used for assessing software is also seen as trustworthy (Johnson et al., 2016). This paper will only present a brief formalization of threat modelling the system used for attack graphs, it is important to note that threat modelling has other uses than attack graphs.

Attack graph consist of domain entities, such as myLaptop, that are partitioned into sets of classes \( X = \{x_1, x_2, ..., x_n\} \). If \( X \) is the computer class then \( myLaptop \in \text{Computer} \). Each class has a set of associated attack step \( A(X_i) \) where \( X.A \) denotes a specific attack step, such as \( \text{Computer}.login \). Moreover there can exist link relationships which is a binary tuple of objects from different classes, denoted \( \lambda = (x_i, x_j) \), for example myLaptop might be a containment link to rootUser. These links are then partitioned into sets of associations \( \Lambda = \{\Lambda_1, \Lambda_2, ..., \Lambda_n\} \) which describe the relationship between classes, in such a way that

\[
\forall x_i, x_k \in X_m, x_j, x_l \in X_n | \lambda_1 = (x_i, x_j) \in \Lambda \land \lambda_2 = (x_k, x_l) \in \Lambda
\]

The classes then have roles in associations such that \( \Gamma(X_i, \Lambda) \) where \( X_i \) then has a containment link to it’s container, for a Users container is the Computer therefore \( \text{User.container} = \text{Computer} \). Then on an instance level this would be \( \gamma(x_i, \Lambda) = \{x_j | x_i, x_j \in \lambda \land \lambda \in \Lambda\} \), i.e. \( myLaptop.contained = \{\text{rootUser, johan, guest}\} \).
Associations can lead to attack steps being connected by directed edges, $e \in E$, implying that the compromise of one attack step leads to the compromise of other objects attack step. For example $e = (\text{myLaptop.login, myLaptop.contained.access})$ would mean that login leads to access on all user contained by myLaptop. Each attack step can be of type OR or AND, which means that if it is OR an attacker can start to execute an attack step immediately as one of their parent attack steps is compromised. While for AND the attacker needs to compromise all it’s parent attack steps. Moreover, classes can also contain defenses which are denoted $D(X_i)$ that can be set to true or false. This means that if $X_i.D$ has an direct edge to another attack step of $X_j$ then that attack step cannot be performed or the time to compromise is affected. For example, myLaptop could have a defense passwordRequired that leads to myLaptop.login and if set to true the attacker’s time to compromise will be slowed down in order to find the password.
3.2 The MAL syntax

For a more detailed specification of the MAL syntax read Johnson, Lagerström, and Ekstedt, 2018. This will only be a general explanation with examples related to this report’s subject.

Each domain created using MAL consists of assets which are similar to classes in object-oriented languages. We create an asset for an entity such as a Server that we want to instantiate (i.e. `ubuntuServer`). There is also the possibility to create `abstractAsset` which can be used to create an abstract class to be extended and instantiated by other assets.

In each of these assets there is one or more attack steps, with there being two different types of attack steps which are called AND and OR denoted by \& and |. An OR attack step can be initiated as soon as one parent attack step is finished, while an AND attack step requires all parent attack steps to be finished. An asset can have an attack step such as `ubuntuServer.login`. You can also specify the time it takes to complete an attack step by declaring a probability distribution beside the name. If no distribution is present, the time to compromise is instant.

There are also defenses denoted by # which can act as optional attack steps if set to true they are active. Let’s say the attack step login was an & step then if there was a defense `ubuntuServer.passwordRequired` set to true which leads to login, then also `passwordRequired` has to be finished before login can be reached.
category Hardware {
    AbstractAsset Computer {
        | login
        -> access
        & access
        -> compromise
        # passwordRequired
        -> access
    }

    Asset Ubuntu extends Computer {
        & access [ExponentialDistribution(10.0)]
        -> terminalAccess
        | terminalAccess
    }
}

Lastly there are associations which are used to determine the relationship between assets in order to expose all the pathways through the modelled domain.

Associations {
    Computer [container] 1 <- Users -> * [contained] User
}

4 Related Work

Motivation for why MAL is choosen over other threat modeling languages, and introduces some work done previously on vehicle cyber attacks.

4.1 Threat modeling languages

There are a handful of related works involving MAL and languages written in MAL, like the ones already mentioned: vehicleLang, awsLang, and coreLang. There are also other languages specifically designed for writing attack/defense graphs or trees or to analyze the security of system architectures. Some of these are DrAGON and CySeMoL. The Cyber Security Modeling Language(CySeMoL) uses a Probabilistic Relational Model to assess the security of a system. Similar to how Bayesian Networks work, each attribute in the system is associated with a conditional probability table that defines the attribute’s value given all possible combinations of states in the attribute’s parents (Sommestad, Ekstedt, and Holm, 2013). P³CySeMoL was created to improve some limitations of CySeMoL such as the scope of assets, maximum time spent by attacker on a attack step and decrease computational cost (Hannes Holm, 2015). This is similar to how MAL works but according to Johnson, Lagerström, and Ekstedt, 2018 it is hard-coded and inflexible and therefore MAL should be easier to use for modelling several different domains. Moreover Johnson, Lagerström, and Ekstedt, 2018 also mentions that there are languages which only do either modelling, probabilistic simulation or attack graphs, and these languages are therefore seen as not being user-friendly as they require too many different tools in order to be used to analyze the security risks. Thus all this leads to MAL being the most interesting candidate for a threat modeling language to use and discover it's potential.

4.2 Vehicle cyber attacks

Other relevant work include general analyses of the state of cyber security risks regarding vehicles. Some of these analyze the increasing security risks of having carry-in devices in cars such as smartphones and USB-memory-devices as well as Global Navigation Satellite Systems and vehicle-to-vehicle communication (Onishi et al., 2017; Onishi, 2012). This
work helps determine the severity of a potential hack on for example the in-
fotainment system and therefore what set of vulnerabilities are potentially
exposed.

4.3 coreLang

coreLang is developed as part of the MAL project and comes with the
compiler. It is a set of classes that correspond with the fundamental
parts of common IT elements. The classes it consists of are: Machine,
Account, Vulnerability, Network, Data, Dataflow, and User. These are the
most fundamental classes and mimic some of the basic behaviours of
these, therefore these can be used to extend into new classes that has full
functionality wanted for the modelled domain.
5 Application of MAL in vehicleLang

5.1 Overview of the project

vehicleLang is a domain specific language built in MAL with the purpose of being a language for constructing attack graphs to model the possible cyber security threats against vehicles. The language was made by gathering resources and information from vehicle security research, security engineering and attack graphs. The result is an instance of MAL with various classes representing mostly relevant software parts and attacks that are related. Some classes are also derived from coreLang.

Figure 2: Model of vehicleLang (Katsikeas, 2018)

Some of the assets in the language include ECU (Engine Control Unit), Firmware, and IDPS.

5.2 Components in vehicleLang

In vehicleLang the only components that this language should connect to is the outward-facing part, which is the vehicleLang public interfaces
module. The important parts are:

**InfotainmentSystem** models a infotainment system in a car that can be used to receive remote updates over internet connection, can patch the updates to different parts of the car and collect telemetry from the car parts and send it to server.
6 Application of MAL in AWSLang

6.1 Overview of the project

AWSlang is a domain specific language aimed at AWS domain and created by using the MAL. It was completed using a Design Science Research Process Model (DSR cycle). First a domain survey was performed then a feature matrix on that domain, after this a MAL specification could be produced with those features that was thereafter tested. Virdi determined which features of AWS was used the most first in order to determine what the design should focus on completing. The MAL specification was created with some assets from coreLang, a few modified from coreLang and some new assets solely related to AWS specification such as IAMAccount, Role, Group, AccessKey, etc (Virdi, 2018).

6.2 Components in AWSLang

**Instance** models all types of virtual machines available through the Amazon Elastic Cloud computer service to build and host software. This is an extension of the abstract Machine from coreLang where the main difference is that the Instance require user to authenticate using AccessKey.

**OperatingSystem** models the operating system being run on any Instance used on AWS, and the user can pick the OperatingSystem from a list provided by Amazon or community customized Linux images. The custom images expose the user to more security threats.

**Application** models any additional software that the user wants on their AWS Instance in order to build their sought after software system. Application can use a firewall to increase the effort needed to compromise it.

**Bucket** models the Simple Storage Service of AWS which can be made either private or public. If made publically available it runs the risk of being compromised by a dictionary attack, which is a brute force attack where
commonly used words are combined in order to break a password, and also 
since there is no additional access control on Bucket then authenticate via 
the name of the bucket leads to a compromise.

**IAM account** models the account used to log on to AWS and a compromise 
of this will lead to access of all Bucket, Instance and more that it has 
privileges for.

**Role** models that users can be given temporary credentials to access 
some resource on AWS.

**Group** models that several IAM accounts can be put together and then 
the administrator can easily give all in the Group the same access privi-
leges.

**AccessKey** models that each instance require a private key to be accessed 
by a user, the instance has a private key pair.

**CryptographicKey** models the other key used in AWS to encrypt and 
decrypt data in AWS environment.

**Network** interacts with other entities and models the possible attacks 
against cloud networks. Networks can be created by an IAM account.

**Gateway** allows for a representation of Internet Gateways or NAT Gate-
ways in the AWS Cloud environment to provide routing rules between two 
subnets.

**SecurityGroup** models the ability to create virtual firewalls on VPC to 
control the traffic.
7 AWS Connected Vehicle Infrastructure

7.1 Overview

AWS enables automotive companies to build serverless IoT applications that collect, process, analyse, act and store data from connected vehicles without having to manage their own infrastructure. With AWS IoT customers can connect their vehicles and devices to AWS cloud securely, in order to with help of other AWS applications build event-triggered applications that, for example can track vehicle diagnostics and health, predict maintenance requirements and provide recommendations to the driver\(^6\).

7.2 How to connect vehicles to AWS?

Amazon has provided detailed documentation on how vehicles should be connected to AWS and what different uses there can be with this. As previously mentioned there are a lot of use cases for AWS connected vehicle solution infrastructure, and AWS has provided a way to create a infrastructure with default parameters that builds an environment in the AWS cloud shown in figure 3.

In this we can see that there is a AWS Greengrass core in the vehicle that connects with AWS IoT to provide authentication, and make sure the vehicle sends it’s data to correct endpoint. Each application usually contains AWS lambda that perform some function on data and then a database to store the processed telematics. These application can then use other parts of AWS library to create wanted functionality.

---

\(^6\)Amazon (2017d). "Introducing the AWS Connected Vehicle Solution". In:
Figure 3: AWS Connected vehicle solution
7.3 AWS components

This section provides a brief explanation of what each important AWS components used in the AWS connected vehicle solution does.

AWS IoT core is a AWS cloud service that lets you connect IoT devices to AWS application securely and that is easily scalable. In this case it will receive requests from your vehicle and route it to the correct AWS endpoint, or the vehicle can subscribe to endpoints through the IoT core.

AWS Greengrass Core is a computer in vehicle that preprocesses telematics data and then sends requests to the IoT core. It can also receive responses and be used to manage updates on different vehicle devices such as infotainment system.

AWS Lambda A service which allows users to upload and execute code on the cloud without having to manage the servers. Users pay for the
total compute time their code has been executing. The code can later be invoked from any web app\(^\text{12}\).

**Amazon S3**  A simple and flexible data storage service that allows for any type of data to be stored through a web service interface. The service proved strong guarantees for scalability, availability, security and performance\(^\text{13}\).

**Amazon Kinesis Analytics**  A service that provides built-in functions to analyze streaming data so that the user can act on real-time data. Like most AWS it will scale automatically based on the computing power needed\(^\text{14}\).

**Amazon Kinesis Streams**  is data streaming that can handle data from several different sources and provides the user with ability to perform analytics, for example with Amazon Kinesis Analytics for real time visualization of large sets of data\(^\text{15}\).

**Amazon Kinesis Firehose**  A tool for loading streaming data and uploading it to other data analytics services like Amazon S3 and Amazon RedShif. It is a fully managed service which automatically scales to the data throughput. It can also batch, compress, transform and encrypt the data before loading it\(^\text{16}\).

**Amazon DynamoDB**  A fully managed NoSQL Database service which stores key-value and document structures\(^\text{17}\).


\(^{13}\)Amazon (2019c). “Amazon S3”. In: URL: https://aws.amazon.com/s3.


\(^{15}\)Amazon (2017b). “Amazon Kinesis Data Streams”. In: URL: https://aws.amazon.com/kinesis/data-streams/.


\(^{17}\)Amazon (2019a). “Amazon DynamoDB”. In: URL: https://aws.amazon.com/dynamodb.
8 Method

This chapter will describe the process used to create the DSL in other words how the two languages were integrated, how we created our testcases and how they were evaluated.

8.1 The process of integration

The process of integration started with understanding how both awsLang and vehicleLang were built. Thereafter both languages were tested and compiled separately to make sure they worked with the latest MAL compiler. This took much effort since awsLang was no longer maintained and therefore wasn’t compatible with latest MAL compiler, therefore those issues had to be addressed first. Then once both languages worked separately the parts that should be connected were identified, which in vehicleLang was the public interfaces module and in awsLang it was the network client that a client is meant to use to connect to all AWS services. In the connected vehicle solution presented by AWS there are a lot of possible assets to model as presented in 3. The final decision was made though that most of those assets were meant to be modelled in the aws domain and therefore should not be created for this language, and therefore can instead be used for testing purposes. Some few assets were still created specifically for this language and the integration layer. Lastly the integration layer was made using MAL and by in one malfile include the other two languages, and then creating relevant extensions of present classes and associations to model the connection.

8.2 Choice of testcases

The choice of testcases was focused within the scope of the work meaning that testcases inside the AWS cloud or the car are left for respective language.

In 2018 there was a attack on Honda Connect App using publically available Amazon S3 buckets. This attack exposed 50000 users connect ID’s, VIN and more which could later to control some parts of their cars using the
app by attackers. For example, the location of the car is exposed to the attacker\textsuperscript{18}.

There was another similar attack against a Tesla-owned Amazon cloud S3 bucket where the hackers got access to telemetry data, mapping, and vehicle service data and managed to install a cryptocurrency mining service on their servers. This attack was unknown until a group of security researches performed a similar attack where they discovered the harmful software that the hackers had install\textsuperscript{19}. This report will model the potential compromise that can be made by an attacker if the manufacturer decides to use public Amazon S3 buckets for their connected vehicle solution.

This work has focused on creating a general test of the code and not unit tests. For one the code base for just this project is quite small and the general test of the code will therefore tests most parts of it, and then awsLang and vehicleLang are separately unit tested. A version of the proposed test will be made where the defenses are active to make sure that they work as intended.

Keywords used for searching for testcases:

1. AWS Connected vehicle solutions
2. AWS
3. connected vehicle cyber attacks
4. connected vehicle attacks
5. AWS leak
6. AWS vehicle hack

The keywords have been used in a number of search engines, such as Google, DuckDuckGo and National Vulnerability Database. Most of the attacks discovered through these keywords lead to attacks that were outside the scope of this work. Those attacks were instead better modelled by awsLang and vehicleLang, and would not show the connection between the two languages.


8.3 Test evaluation

Our test with the publicly available S3 containing sensitive user data that can be used to login to the app used by the car. This S3 bucket is prone to a dictionary attack and this means that the attack shall take quite high effort to begin with, but thereafter the app should be connected with vehicles OBD2 connector which in turn can access the infotainment system. After this we have made the assumption that the infotainment system can be seen as the AWS greengrass core that has a account which can be used to compromise the AWS connected vehicle solution.
9 The awsVehicleLang meta model

This section will present our final version of awsVehicleLang and what classes it has implemented, and how they are meant to be used. Example of test code will be included in appendix.

![Graph diagram of awsVehicleLang](image_url)

Figure 5: Graph diagram of awsVehicleLang

9.1 Integrating classes

**GreenGrass** is an extension of the infotainment system that can easily access the AWS network through the IoT core and request/send data. It also could be used to run AWS lambda by itself in order to manipulate requested data.
AWSIoTCore is meant to represent the fact that the greengrass core is a client in the AWS network, and therefore has an IAM account to be used as verification when it wants to send requests to the AWS connected vehicle solutions different endpoints. This extends the AWSNetworkClient and adds the attack step to compromise IAM account since this is already in use on the greengrass core, and therefore access on the greengrass core leads to compromise of the account.

CarApp is meant to represent an application used to remotely access a connected vehicle and that can also be used to request data from the car through the greengrass core. The car application can be used by a client remotely using an app in smartphone or on computer as shown in 4.

awsLang a lot of assets had same names as in vehicleLang and therefore compilation was not possibly until they were renamed. They are still functionally same as in awsLang but with other asset names, a list will be provided in appendix.
9.2 How to use awsVehicleLang

From the AWS side any existing model of application in the connected vehicle solution should be associated with the AWSIoTCore in order to connect to the car. While the car and its infotainment system should then be associated with the CarApp. In the CarApp there is a defense available Infotainment2FA, and represents that some cars infotainment system applications can require more than just account id and password to login.

In order for our application layer to work there were small changes made to a few assets in awsLang. These assets were IAMAccount, AWSData and InfotainmentSystem, and also some new associations with these were made in the integration layer. Unfortunately also needed for the integrating layer is a special adopted version of awsLang available on this projects GitHub repo. Below is an example of test code using awsVehicleLang.

```java
@Test
public void testBucket_withDefenses() {
    System.out.println("~~~~~ DEFENSES IN PLACE ~~~~~");

    //true because this is a private bucket and dictionary attack should not be possible
    Bucket bucket = new Bucket(true, true);
    IAMaccount iamaccount = new IAMaccount();
    AWSIoTCore awsConn = new AWSIoTCore();
    GreenGrass infoSys = new GreenGrass();

    iamaccount.addAccessedBuckets(bucket);
    iamaccount.addAwsConn(awsConn);
    awsConn.addGreencore(infoSys);

    Attacker attacker = new Attacker();
    attacker.addAttackPoint(bucket.attemptConnectPublicBucket);
    attacker.attack();

    bucket.bruteForceAttack.assertUncompromised();
    bucket.compromisedAccess.assertUncompromised();
    bucket.authenticate.assertUncompromised();
    bucket.authenticatedAccess.assertUncompromised();
}```
bucket.access.assertUncompromised();
bucket._machineAccess.assertUncompromised();
bucket.denialOfService.assertUncompromised();
iamaccount.compromisedAccess.assertUncompromised();
awsConn.requestData.assertUncompromised();
infoSys.access.assertUncompromised();
10 Result

This section will present how successful the language was at modelling test cases and what type of vulnerabilities the AWS connected vehicle solution potentially exposes.

10.1 The model’s performance

The test that we wrote that would simulate the Honda attack got the results that we had expected. The attacker’s predicted time to compromise the bucket was 3 days, while the rest were instantaneous after that step was fulfilled. By setting the starting AttackPoint of the Attacker as one of the attack steps of an S3 Bucket with its defense step “dictionaryattack” set to false, it managed to in the end compromise and access both the Greengrass Core and the IAMAccount used to log into the AWS Cloud. This compromise shows a set of vulnerabilities that includes parts of both awsLang and vehicleLang, and therefore it is a new set of vulnerabilities. An illustration of the attack is shown below:

Figure 7: Possible attack path of the Honda leak on AWS connected vehicle solution
10.2 The compromised parts of the connected vehicle and the AWS network

There are many ways a hacker getting access to the infotainment system of a vehicle could cause harm and danger. This has been demonstrated in several known vehicle hacks, including against the brands Volkswagen, Audi, Tesla and more. In many of today’s modern cars, the infotainment system is closely linked to the car’s mechanical features. A hacker can thus remotely perform various operations on the vehicle, such as turning off the lights, accessing the breaking system, or even turning off the vehicle.\textsuperscript{20,21}

Hackers getting access to the AWS IAM Account may also be able to access any of the data that is stored on its AWS Cloud, depending on the privilege levels of the account. This data could be sensitive information which might enable a more extensive hack of the car, or even other cars.

\textsuperscript{20}Wired, 2015.
\textsuperscript{21}Security, 2018.
11 Evaluation

This section will evaluate the usefulness of MAL, how well integration between domains of MAL and what further research is recommended.

11.1 Usefulness of the MAL

MAL is a language that is relatively simple to understand and to learn as well as to test and create attack models with. It is a language that also doesn’t require the user to write so many lines of code. It elegantly compiles into readable and structured Java code. As shown in the result a small integrating layer can be used to make two different applications of MAL work together, and thus expose more vulnerabilities within the system as a whole.

11.2 Struggles

In the beginning of creating a integrating layer there was a need to make sure that both awsLang and vehicleLang worked separately. vehicleLang was easy and fast to compile and test since it was maintained. awsLang on the other hand took a lot of effort to compile this was due to it not being maintained, the syntax the compiler supported had changed a bit. Here the biggest issue was caused by the fact that the newer MAL compiler couldn’t handle whitespace within distribution definitions, for example:

& bruteForceAttack [UniformDistribution(0, 100)]

The whitespace between , and 100 could not be handled by the new Attacker parser. This was therefore quite time consuming for the process of integrating.

On top of that the most difficult part of integration was the fact that when including several MAL files into one MAL lead to multiple declarations of assets with same name. This issue lead to questioning if the correct way to integrate two MAL is by using MAL since the compiler couldn’t handle multiple declarations of same asset names. Also both languages had quite different definitions within the assets and therefore they couldn’t just be removed from one language. The solution in this case was to rename all
the duplicates in awsLang by adding "AWS" in front of asset name. This is not a optimal solution and in even larger integrations of MAL this would not be a viable solution at all. Thus in order to fix this issue it would be great if the MAL compiler supports something similar to namespaces in object-oriented programming, which can be used when including in order to avoid ambiguity. Another way of handling it would be to make sure that all MAL languages follow some standard for naming assets that make sure ambiguity is avoided but again this is unlikely and harder to enforce.

Another way would be that the integrating layer would have been built using something else than the MAL. For example building Java classes that could be used to connect the compiled versions of different MAL implementations. This would also lead to problems because if either language grows it would harder to maintain the integrating layer since it wouldn’t automatically match the new features or slight changes. Thus it was decided that doing the integrating layer still in the MAL was important. Moreover this decision has lead to the possibility for hopefully some improvements in the MAL that can help further it’s modularity so that it can be used to model more threats.

During the writing of this report, various attempts have been made to gather examples of relevant attacks to use as test cases for the awsVehicleLang. There could be many reasons for why these sorts of attacks seem to be rare. One potential reason is that the companies that the attacks have been aimed at want to keep them private and don’t want to disclose them to the public media. For instance, it has been reported that Uber Technologies once paid hackers $100,000 to keep a data breach secret.

The potential negative outcomes of disclosing them are plenty, for example it could worsen the reputation of the company, or lead to costumers requiring a large scale retrieval of their cars (similar to what Jeep decided to do), or it could encourage other hackers to attempt similar attacks.
12 Conclusions

The work shows that it is possible to build an integrating layer between two applications of MAL. Also that it can potentially expose more vulnerabilities within the larger modelled domain. As previously mentioned from Lagerström et al., 2017 a more complex software architecture leads to increased security risks, and according to Noel and Jajodia, 2014 a larger attack graph means that there will be more ways the system can be compromised. This is shown by that a relatively simple attack on the S3 bucket leads to leaking of information which in turn lets an attacker gain access to both the AWS network and the vehicles network layer.

As of now many vehicle companies are only in development stage of using the AWS connected vehicle solution for their cars. Therefore it would be beneficial if these companies at this stage decide to perform audit of their infrastructure choices in order to assess most pressing security flaws ahead of production. This work showed that the AWS connected vehicle solution can potentially expose the manufacturers to new set of vulnerabilities they might not be used to. Thus it would benefit them if they can use tools such as awsVehicleLang which should be easy to adapt to their specifications.

As vehicles become more computerized with infotainment and navigation systems, wi-fi, automatic software updates, vehicle-to-vehicle communication and more, it is clear that the potential attacks against cars multiply. One can only speculate in what types of attacks might be possible in the future. For instance a large scale attack could involve hackers uploading harmful software to many vehicles, perhaps by first accessing a central AWS node.

This work is not complete in the sense that current state of awsVehicleLang only adds a small integrating layer that can only model a small set of vulnerabilities within the AWS connected vehicle solution. At the moment, awsVehicleLang can be used to show that your configuration of aws or vehicle security separately are secure in the sense, that the compromise of one part doesn’t lead to compromise of the other. Therefore further research would be needed into more detail what potential attacks can be made on the AWS connected vehicle solution on it’s own, and in turn this can help explain what attack steps can be potentially exploited within the integrating layer. Also further work is recommended on the MAL compiler.
as explained in the evaluation, for example a new version of awsLang is being developed that will make the current version obsolete. In the current state of integration it would take a lot of work to make this new version compatible.
A /awsVehicelang integration code

include vehicleLangEthernet.mal
include vehicleLangPublicInterfaces.mal
include vehicleLang.mal
include AWSLang.MAL

category AWSIoTCore {
  asset AWSIoTCore extends AWSNetworkClient
  info: "Represents the iot core which is used to handle the connection to and from the
  {   |
        requestData
        -> accounts.compromise
  }
  }

category Application {

  //Through this app you are mainly able to get information about the car
  //and store it in an S3 bucket as well as get access to the InfotainmentSystem
  asset CarApp
  info: "Represents that most connected vehicles have some form of way to be remotely accessible
  {   |
        login
        -> accessInfotainment,
        getCarInfo

        & accessInfotainment
        -> greencore.gainNetworkAccess

        & getCarInfo

        Infotainment2FA
        -> accessInfotainment
  }

  asset GreenGrass extends InfotainmentSystem
  info: "Extends the infotainment system and the greengrass core let's you run AWS lambda
  {   |
| gainNetworkAccess
    -> awsConn.requestData

associations {
    AWSIoTCore [awsConn] 1 <-- AWSIoT --> 1 [greencore] GreenGrass
    AWSIoTCore [awsConn] 1 <-- AWSClient --> * [accounts] IAMaccount
    CarApp [app] 1 <-- InfotainmentConn --> 1 [greencore] GreenGrass
    CarApp [appAccount] 1 <-- CarAppAcc --> 0-1 [appData] AWSData
}
B The testcase for Honda leak

/**
 * This test is meant to model the leak of Honda users personal information through unsecured S3 buckets.
 * Source: https://kromtech.com/blog/security-center/honda-leaked-personal-information
 * The attack makes the assumption since account ids and passwords were leaked then the
 */
@Test
public void testBucket_noDefenses() {

System.out.println("~~~~~ NO DEFENSES IN PLACE ~~~~~");

//false because this is a public bucket and dictionary attack should be possible
Bucket bucket = new Bucket("Bucket", false, true);
IAMaccount iamaccount = new IAMaccount("IAMacc");
AWSConnection awsConn = new AWSConnection("awsconn");
AWSData awsData = new AWSData("awsdata");
CarApp carApp = new CarApp("carapp");
InfotainmentSystem infoSys = new InfotainmentSystem("infosys");

bucket.addData (awsData);
awsData.addAppAccount (carApp);
infoSys.addApp (carApp);
iamaccount.addAwsConn (awsConn);
awsConn.addGreencore (infoSys);

Attacker attacker = new Attacker();
attacker.addAttackPoint(bucket.attemptConnectPublicBucket);
attacker.attack();

bucket.bruteForceAttack.assertCompromisedWithEffort();
bucket.authenticate.assertCompromisedWithEffort();
bucket.authenticatedAccess.assertCompromisedWithEffort();
bucket.access.assertCompromisedWithEffort();
bucket._machineAccess.assertCompromisedWithEffort();
bucket.denialOfService.assertCompromisedWithEffort();
awsData.read.assertCompromisedWithEffort();
carApp.login.assertCompromisedWithEffort();
infoSys.access.assertCompromisedWithEffort();
awsConn.requestData.assertCompromisedWithEffort();
iamaccount.compromisedAccess.assertCompromisedWithEffort();
}


Johnson, Pontus et al. (2016). *Can the Common Vulnerability Scoring System be Trusted? A Bayesian Analysis*. IEEE.


