Attack Surface and Vulnerability Assessment of Automotive Electronic Control Units

Martin Salfer and Claudia Eckert
Technische Universität München, München, Germany

Keywords: Security Metrics, Embedded Systems, Cyber-Physical Systems, Exploit Engineering Cost Assessment.

Abstract: Modern vehicles are controlled by an on-board network of ECUs (Electronic Control Units), which are specially designed computers that contain tightly tailored and customized software. Especially the trends for ECU connectivity and for semi-autonomous driver assistance functions may have an impact on passenger safety and require thorough security assessments, yet the ECU divergence strains those assessments. We therefore propose an easily automated, quantitative, probabilistic method and metric based on ECU development data and software flash images for the attack surface and vulnerability assessment automation. Our method and metric is designed for the integration into an (iterative) engineering process and the facilitation of code reviews and other security assessments, such as penetration tests. The automotive attack surface comprises especially internal communication interfaces, including diagnosis protocols, external and user-accessible interfaces, such as USB sockets, as well as low-level hardware interfaces. Some exemplary indicators for the vulnerability are access restrictions, casing tamper-resistance, code size, previously found vulnerabilities; strictness of compilers, frameworks and application binary interfaces; conducted security audits and deployed exploit mitigation techniques. This paper's main contributions are I) a method and a metric for collecting attack surface and predicting the engineering effort for a code injection exploit from ECU development data and II) an application of our metric and method into our graph-based security assessment.

SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

The automotive industry drives ECU (Electronic Control Unit) consolidation and connectivity for economic, functional and environmental reasons. Yet, the complexity of deep integration and internetworking also brings hardly foreseeable security implications. Some were revealed by practical attack studies, e.g., (Koscher et al., 2010; Checkoway et al., 2011). Security researchers demand more objective security engineering instead of mere expert intuition (Schneier, 2012). The German National Road Map for Embedded Systems explicitly demands reliable, quantified security statements for embedded systems (Damm et al., 2010).

A vehicle's attack surface increases due to the internetworking of control units with the environment and due to the integration of abundant services and functionality, e.g., for highly automated driving. The attack motivation rises due to asset accumulation: A common car will bear payment credentials for tolling, parking and electric charging and have access to cloud services, including sensitive and personal data.Security fixes are rather expensive to create and to deploy for the automotive industry, even compared to enterprise systems: Every change requires thorough and lengthy testing for guaranteed side-effect free safety quality. A vehicle's life cycle spans over more than a decade, which makes security support extra costly. Hence, the automotive industry invests in comprehensive engineering and security assessments with a long-term foresight. One method is the assessment of an ECU's attack surface. A high grade of automation and efficiency is necessary for handling the many, deeply customized automotive ECUs. "There is a pressing need for practical security metrics and measurements today" (Manadhata and Wing, 2011).

Data collection for security analysis is a tedious task, which could be facilitated greatly by automation. Yet, typical corporate IT (Information Technology) data collection software and methods are not applicable for automotive IT or CPSs (Cyber-Physical Systems). ECUs/CPSs usually communicate heterogeneously and combine a large, fast-changing and obscure variety of real-time and general-purpose operating systems, application binary interfaces and file

Salfer M. and Eckert C.
Attack Surface and Vulnerability Assessment of Automotive Electronic Control Units.
DOI: 10.5220/0005550003170326
In *Proceedings of the 12th International Conference on Security and Cryptography (SECRYPT 2015)*, pages 317-326
ISBN: 978-989-758-117-5
Copyright © 2015 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

systems for the sake of maximum dependability and efficiency and thus sustainability. While corporate IT usually communicates homogeneously with IP (Internet Protocol), automotive IT communicates with a combination of CAN (Controller Area Network), MOST (Media Oriented Systems Transport), LIN (Local Interconnect Network), FlexRay, ByteFlight, BroadR-Reach, etc. While corporate IT usually operates homogeneously on x86 with POSIX-(Portable Operating System Interface)-compliant operating systems, automotive IT operates on TriCore, Super-H, PowerPC, V850, x86, ARM, etc. with OSEK (Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug) / VDX-(Vehicle Distributed Executive)-, AUTOSAR-(AUTomotive Open System ARchitecture)-, sometimes POSIX-compliant operating systems or even merely on bare metal. Online data collection methods can therefore hardly cover all available security measures that are integrated into automotive systems.

This paper focuses on the threat of injecting malicious code using vulnerabilities of the actual implementation. Malicious code injection is one of the most subtle and critical threats as it potentially endangers all three main security goals at once: confidentiality, integrity and availability. A general security assessment has to, of course, also consider other threats, including the misuse of valid communication. This paper presents a method for assessing the attack surface and vulnerability likelihood for individual ECUs. However, it does not cover implications of the on-board network architecture or the attractiveness of individual ECUs or functions for an attacker. Our research on the complexity problem of comprehensive and reliable security assessments of an automotive on-board network inspired us for this paper. The contributions of this paper are:

I. We formalized a method and a metric based on development data for systematically assessing the attack surface and attacker effort for automotive ECUs. The method and metric arose in cooperation with automotive OEM security experts and is designed for a seamless integration into a graph-based on-board network security assessment.

II. We applied and integrated the method and metric in our attack graph-based assessment and show exemplary results.

Section 2 defines, describes and discusses our method. Section 3 shows an application of our method on a complete on-board network with an attack graph construction algorithm. Section 4 gives an overview over related work. Section 5 discusses ideas for optimizing this assessment. And Section 6 concludes our automotive attack surface and vulnerability assessment method.

2 METHOD

This section introduces a systematic approach for assessing the attack surface and vulnerability of an automotive ECU. The anticipated threat is the injection of malicious code. We will systematically walk through our method with the following steps or subsections.

1. *Attack Surface Collection* – How much and what attack surface bears an ECU? An ECU can only be exploited if it bears interfaces to the outside. We collect accessibility information mainly by sourcing different documents and databases.
2. *Vulnerability Prediction* – How likely is attack surface vulnerable?
3. *Vulnerability Finding Effort*– How much effort is presumably necessary to find a vulnerability on the attack surface?
4. *Exploit Creation Effort* – How much effort would a potential vulnerability consume, considering the effort for accessing the attack surface, engineering a basic exploit against it and finally counteracting possibly deployed exploit mitigation techniques?

2.1 Attack Surface Collection

The first step for the attack surface estimation is the collection of the attack surface. The attack surface is the sum of interaction opportunities with an ECU, i.e., everything that can potentially influence the control flow of the software. Such attack surface is typically incoming data that is parsed and processed. Incoming data can be found by monitoring the interfaces of an ECU in use or by evaluating specification documents, binary or source code. The attack surface comprises a set of accessible services and inbound communication as well as the system's I/O and hardware interfaces.

Each software consists of program code and provides services. Each of these services could be potentially abused for injecting code. An ECU's "attackability" correlates positively with its attack surface, i.e., the more accessible a service is, the easier it is to attack. Attack surface is obligatory for any attack. All attack surfaces are treated as complementary, i.e., one surface must be considered isolated from other attack surface. Attack surfaces are not mentioned comprehensively as an attacker with enough creativity can reveal new, unlisted surface, so a residue called *Others* will always exist. Specialized *ECUs* can have extra

features and therefore attack surface not mentioned here or not yet introduced into mass market *ECUs*.

We classify an *ECU's* attack surface as below or as seen in Figure 1.

- *Incoming Data* is any data arriving at an ECU.
  - *Job Parameters* are input fields in ECU jobs. *Jobs* are services that can be called via the on-board network and are useful for maintenance and diagnosis. Access to a subset of jobs (and therefore to the on-board network) is legally obligatory over the OBD-II (On-Board Diagnostics) port and therefore represents attack surface prescribed in all cars by law.
  - *User Input* is data supplied directly by the user, e.g., files, streams or user interface input.
  - *Signals* are routinely received data chunks on an on-board network for regular operation, e.g., wiper commands or speed information.
  - *Meta Data* represents any supplementary data, which could influence parsers or drivers, e.g., CAN or ethernet frame IDs.
- *Apps* are third party-supplied software packets.
- *Reflash Routines* are ways for replacing an *ECU's* software with a supplied flash image.
- *Hardware Attack Surface Spots* are spots that can be tampered with only with physical access.
  - *Boot Memory* comprises any storage that could allow execution of attacker supplied code, e.g., an EEPROM (Electrically Erasable Programmable Read-Only Memory) or external flash memory.
  - *Debug Interfaces* are any ports that are not used in regular operation or maintenance, but were used during development and might be used again for advanced maintenance, e.g., JTAG (Joint Test Action Group) interface. Debug interfaces intentionally allow very deep analysis and manipulation.
  - *Inter-Chip Communication Channels* are any data exchange links inside an ECU (usually in between semiconductors) that could be revealed and tampered with, e.g., UART (Universal Asynchronous Receiver/Transmitter), I2C (Inter-Integrated Circuit) and SPI (Serial Peripheral Interface). Such interfaces could be revealed and tampered with.
  - A *Side Channel Attack* is measuring or manipulating an ECU physically for information gathering or control flow change, e.g., power glitches or laser pulses can jump the program counter or let operations silently fail.
- *Other* represents any other not explicitly mentioned attack surface. This listing is not exhaustive and is subject to continuous extension.

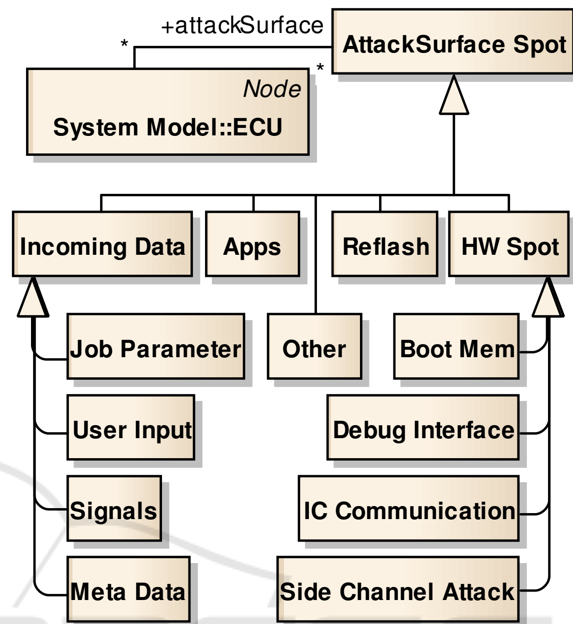

Figure 1: Attack Surface Taxonomy – ECUs bears attack surface that can be divided into *Attack Surface Spots* of either *Incoming Data*, installed *Apps*, a *Reflash* mechanism, *HW Spots* or any *Other*, yet to be found, attack surface.

Cars often also have a single exposed ECU for communication and have Bluetooth, Wi-Fi, RDS (Radio Data System), TMC (Traffic Message Channel), DAB (Digital Audio Broadcasting), GSM (Groupe Spécial Mobile), CDMA (Code Division Multiple Access), UMTS (Universal Mobile Telecommunications System), LTE (Long Term Evolution), GPS (Global Positioning System) and other antennas and services that count to their externally accessibly attack surface. Some ECUs may also have 802.11p (vehicular WiFi) for ITS (Intelligent Transportation System) support. Typical ECUs are equipped with intra-vehicular communication. Some may use radio communication for internal communication, e.g., for comfort access radio keys or tire pressure control radio sensors on the TPMS (Tire Pressure Monitoring System). These attack surfaces are real, but are atypical for an ECU, i.e., only one ECU out of about 50 has such an feature and attack surface. Previous publications named already many attack spots: from tire pressure monitoring sensors up to Bluetooth and music CDs (Francillon et al., 2010; Ishtiaq Rouf et al., 2010; Koscher et al., 2010; Checkoway et al., 2011).

2.2 Vulnerability Prediction of the Attack Surface

Attack surface per se is only vulnerable if there is an exposed, security-relevant defect. If we cannot asses the isolated attack surface by itself that is responsible for a certain attack surface, someone can still approximate by assessing the overall software of an entire ECU. We can estimate a surface's vulnerability by looking at the ECU's vulnerability density v, which is the number of security relevant defects per ECU code size, i.e.,

$$v = \frac{\text{“security relevant defects”}}{\text{“code size”}}. \quad (1)$$

A typical code size unit is the number of statements. Any vulnerability originates from either accident (unintentional software defects) or a backdoor (intentional software defects). A backdoor can be either maliciously for later exploitation or benign for later maintenance. So the vulnerability density sums up the number of intentional and unintentional vulnerabilities. For automotive applications, we assume the density of intentional vulnerabilities (backdoors) being insignificant compared to the density of unintentional defects. The neglect of intentional vulnerabilities seems true even for most open source products as vulnerability advisories are to be found often and back-door warnings are to be rarely found. We therefore approximate the vulnerability density by the density of unintentional vulnerabilities.

Quantifying the vulnerability density directly from code is hard as it is not decidable for a machine whether a control flow change was intended or not.But the vulnerability density v can be approximated by the software's defect rate d. The vulnerability density correlates strongly with the defect density as carefully designed and implemented code typically contains less vulnerabilities. The better software quality, the less defects and the less vulnerabilities. We approximate the vulnerability density v from a software's defect density d (or fault density) as statistical analysis in (Alhazmi et al., 2005) shows that usually 1 % to 5 % of all defects are security relevant defects, i.e.,

$$v \approx x * d \text{ with } x = [0.01, 0.05]. \quad (2)$$

One way to obtain defect density values straight forward is deriving those from process metrics, e.g., bug tracker statistics. A approximation can be done assuming $x = 0.03$, the middle of Alhazmi's found security relevance rate, and $d = 0.001$ for mature and secure software in the productive phase:

$$v \approx 0.001 * 0.03 = 0.00003. \quad (3)$$

Typical indicators for an ECU's individual defect density are mentioned below.

- The number of *previously found defects* and the *code size* of similar ECUs are an approximation basis for the expected defect density.
- The *ASIL (Automotive Safety Integrity Level)* is an indicator for thoroughly checked code and therefore a lower expectation for defects.
- *Safety measures* avoid any defects and raise software quality, e.g., strict type checking, assertions, boundary checking and input sanitation.
- A *conducted code audit* is an indicator for a lower defect density expectancy as a check by several people ensures a higher code quality.
- A *conducted penetration test* is an indicator for a lower defect density as such tests reveal defects before a release and allow efficient hardening.

The likelihood for the existence of a vulnerability can be modelled with a Bernoulli process $P_B(X > 0)$ with P_B being a Bernoulli process distribution function, X being the number of vulnerabilities, q being the probability of a single attack surface spot of being free from vulnerabilities, v being the vulnerability density, and i the number of available *Attack Surface Spots*, and x the average code size of the attack surface spots:

$$P_B(X > 0) = 1 - P_B(X = 0) = 1 - q^{ix} = 1 - (1 - v)^{ix}. \quad (4)$$

Assuming an ECU would have about 200 attack surface spots with an average code size of 5 lines, the ECU's likelihood for a vulnerability is according to a Bernoulli process about

$$P_B(X > 0) = 1 - (1 - 0.00003)^{200*5} \approx 3\%. \quad (5)$$

2.3 Vulnerability Finding Effort Estimation

An attacker has to search on the attack surface for vulnerabilities, before being able to exploit it. One method to reveal exposed vulnerabilities is fuzz testing, also called fuzzing. Fuzzing stimulates existing attack-surface with intentionally modified payload in order to trigger malfunction. Seeing a certain reaction, the attacker can try exploiting the reaction for malicious code execution. Another way is binary analysis, i.e., inspecting the firmware code for identifying input channels and its data processing directly in the code. Any method requires an initial set-up effort; A fuzzing set-up requires a functional ECU with equipment to modify input data and a good understanding of used protocols. The analysis of binary

code requires an extraction of the firmware. Subsequently, all of the attack surface needs to be checked. We assume an attacker to only search as long for a vulnerable attack surface until one is found. With our above assumption of a vulnerability density v, we can assess the success likelihood of an attacker.

We define the vulnerability finding effort f as the sum of both the initial effort f_0 and the sum of all individual attack surface probing efforts f_i from with n attack surface spots, i.e.,

$$f = f_0 + \sum_{i=1}^{i=n} (f_i). \quad (6)$$

Alternatively, the vulnerability finding effort can also be estimated by the expected number of vulnerability finding tries $\bar{n}$ multiplied with the average vulnerability finding effort $\bar{f}_i$, i.e.,

$$f = f_0 + \bar{n} * \bar{f}_i. \quad (7)$$

The expected number of vulnerability finding tries $\bar{n}$ can be derived from the stochastic expected value of a Bernoulli process P_B. The special case compared to standard Bernoulli processes is that the process ends as soon as the first success has occurred, i.e., "vulnerability found". The possible outcomes therefore are in Table 1.

Table 1: **Bernoulli Tries and Results.**

Tries	Results
0	-
1	1
2	01
3	001
...	...
n	0...01
n	0...00

The expected value $\bar{n}$ for the number of tries in such a Bernoulli process P_B with success probability p, failure probability q, stochastic random variable X for the number of successes and a maximum number of tries k (equalling the number of attack surface spots to test for vulnerabilities) can therefore be computed with

$$\bar{n} = 0 + \sum_{i=1}^{i=k} \left(i * q^{i-1} * p\right) + k * P_B(X = 0). \quad (8)$$

With some simplification and replacing the p and q (how they are usually called in stochastic literature) with v and $(1-v)$ as we use it, the expected number $\bar{n}$ of tries till a vulnerability is found can be computed with

$$\bar{n} = v \sum_{i=1}^{i=k} \left(i\,(1-v)^{i-1}\right) + k\,(1-v)^k. \quad (9)$$

The expected value for the finding effort $\bar{f}$ in combination with an assumption about the average individual finding effort value $\bar{f}_i$ therefore is

$$\bar{f} = f_0 + \bar{f}_i \left(v \sum_{i=1}^{i=k} \left(i\,(1-v)^{i-1}\right) + k\,(1-v)^k \right). \quad (10)$$

2.4 Attack Surface Exploitation Effort

The exploitability of a vulnerability of given attack surface depends on various factors such as the processor architecture, compilers, input parser types and exploit mitigation techniques. We also consider the use of authenticated functions, so that additional effort is necessary to access and subsequently exploit vulnerabilities of attack surface. The effort we try to estimate is on finding and exploiting then unknown ("zero day") vulnerabilities. As soon as details of vulnerabilities or the assessed platform are published, the effort drops. We systematically walk through an assessment method for an ECU with the following steps or subsections.

1. *Preliminaries* – An attacker profile needs to be defined to be referenced to for a homogeneous attack effort quantification.
2. *Access Effort* – Attack surface can require effort for overcoming software-based access or authentication checks or protective hardware means.
3. *Basic Exploitation Effort* – Attack surface spots require different basic techniques and thus effort.
4. *Counter Exploit Mitigation Effort* – Even vulnerable attack surface can be inherently robust against exploitation due to exploit mitigation techniques, e.g., stack canaries.

2.4.1 Preliminaries – Reference Attacker Profile Definition and Quantification Units

For estimating all attacker efforts quantitatively, we need to define an *Attacker Profile* as a reference. Our *Attacker Profile* represents a human attacker with economic reasoning. Having a reference *Attacker Profile* will allow us later to easily transpose an assessment onto a different *Attacker Profile*. *Capabilities* represent an attacker's knowledge, techniques and tools relevant for exploiting vulnerabilities. Each *Capability* is proportionally weighted with a floating point number $x \in \mathbb{R}_{>0}$ as a *Grade*. It rates an *Attacker Profile's Capability* in comparison to the corresponding *Capability* of the reference *Attacker Profile*. The reference *Attacker Profile Grades* therefore

are always defined as 1. The *Grade* definitions allow a directly proportional effort estimation for related *Attacker Profiles*. The *Attacker Profile Capabilities* can be arbitrarily defined, yet this definition must be constant throughout an entire security assessment for comparability and must be precisely known to the ones who define further *Attacker Profiles* and to the ones who assess elementary exploit efforts. Therefore, a reference attacker profile has to be documented and all later steps must be able to refer to the initially defined reference attacker. One exemplary inspiration for an *Attacker Profile* is an informatics study program curriculum due to the familiarity and the global standardization for a well understood *Attacker Profile*.A formal definition of an *Attacker Profile* can be seen in Figure 2.

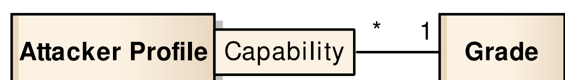

Figure 2: Attacker Profile – Our *Attacker Profile* represents human attackers with economic reasoning. Each *Capability* yields a *Grade* quantifier, i.e., how skilled or well equipped an attacker is.

We define the effort e as a tuple of a capability tag t and its corresponding effort amount r, i.e., $e \in E = \{(t,r) | t \in T \wedge r \in R\}$. The effort amount $r \in R$ is defined here as the set of positive real numbers, i.e., $R = \mathbb{R}_{>0}$, and represents a currency value for comparability reasons. Typical effort measurements are consumed time or money. Some vehicle insurances research institutes measure effort in amount of time required (for a reference attacker profile) and demand cars being secure enough in terms of minimum time effort required. Even though values can not be precise, we prefer using a currency unit as these allow a more versatile comparison. A currency effort can be compared with man-hour estimates with the help of a reference *Attacker Profile* including a well defined capabilities set, and a currency effort can also be compared with labour and black market prices for vulnerabilities. Capability tags $t \in T$ can be mapped on the capabilities of attacker profiles.

2.4.2 Access Effort

Attack surface differs in accessibility, i.e., different access surface requires different effort for reaching it. We define an attack surface's access effort as $a \in E$. The access effort determines how much to spend for overcoming access restrictions.

Software security means for protecting or unlocking attack surface are credential checks or firewall rules. Authorization functions often challenge the client with a code that has to be answered with the correct response code. OEMs (Original Equipment Manufacturer) put different effort in securing the authentication. Depending on the used cryptographic algorithms and key strengths, the access factor a varies greatly. An ECU will always have at least some (publicly) accessible attack surface as the authentication routine represents attack surface by itself. Software unlocking of attack surface can occur by many ways, e.g., snooping or brute-forcing a key or gaining knowledge about it from social engineering. Sensitive jobs (including the reflash mechanism) are secured with a 16 bit key challenge-response authentication and a 10 s retry blocker, as discovered by (Koscher et al., 2010). Such an ECU can be brute forced within 15 days; Power cycling can reset the retry blocker and speed up the brute force to reveal one key within 1.5 days.The reflash routine is usually strongly protected due to its sensitivity and therefore bears a high access effort a and differs starkly with the used cryptographic algorithms and key size. The signal parameters are received without authentication over CAN; The signals' access effort is $a_{signals} = \emptyset$.

Hardware means for protecting attack surface are, for example, resistive covers and being installed in hardly accessible space. If an ECU is integrated in a hardly accessible installation space, an attacker has to remove many parts and therefore invest much effort into accessing the ECU eventually. Resistive Covers help in blocking access to internal ports, especially development or debug ports, or to parts that contain an ECU's firmware. Chips can have cryptographic features for protecting inter-chip communication or cryptographically check the firmware image. Hardware-based attack surface protection can also be supported by software means; For example, certain side-channel attacks can be rendered ineffective by special software obfuscation techniques.

2.4.3 Basic Exploitation Effort

Vulnerable attack surface requires at least a minimum effort for injecting code, which we call the basic exploitation effort $b \in E$. The effort differs starkly, because ECUs are designed and built with such different fundamental hard- and software. Some hard- and software implicitly and carefully checks all incoming or used data, some other hard- and software uses incoming data naively. Whilst some ECUs are programmed with automatically input checking programming languages like Java (or certain ASCET compilers), many ECUs are directly programmed in C/C++, which does not automatically check buffer boundaries and hence

is more prone to buffer overflows[1]. Depending on the input's data type, a vulnerability is harder or easier to exploit for malicious code execution. Buffer overflows typically allow a direct injection of code and often an indirect manipulation of the CPU's (Central Processing Unit's) program counter. Besides the popularity and the practicality of buffer overflow attacks, any input (also primitive input) can cause code to react unintentionally, albeit much harder to inject and run code. We define four classes of input data regarding the expected exploitation effort.

- *Unchecked complex* types are rather easy to exploit, e.g., strings/arrays or any buffered data on C/C++. We define the basic exploitation effort for unchecked complex types as b_u.
- *Checked complex or primitive* types are rather hard to exploit, e.g., a single int, bool and float in C/C++/Java or strings/arrays in Java. Checked complex types are immune to typical buffer overflow attacks, but can still provoke unintentional code behaviour similar to single primitive types. We define the basic exploitation effort for checked complex or primitive types as b_c.
- *Unprivileged code* is input that will be executed with few permissions, e.g., apps.
- *Privileged code* is input that will be executed with full permissions, e.g., firmware.

The classification will be relevant for possible exploit mitigation techniques.

2.4.4 Exploit Mitigation Counter Effort

A vulnerable and accessible attack surface can be non-exploitable due to explicit and implicit exploit mitigation techniques. Exploit mitigation techniques do by definition never avoid the exploitation entirely, but raise the effort for making an exploit that is able to successfully execute injected code. Some exploit mitigation techniques found in state of the art ECUs are described below.

- *Stack canaries*: Stack canaries are extra variables on the stack that are checked before critical actions, such as loading the program counter with the return address. A simple buffer overflow would overwrite (and usually invalidate) the stack canary, too. Certain implementations even allow sidestepping stack canaries by manipulating the exception handler. We define the exploit mitigation counter effort against stack canaries as c_s.
- *NX (No eXecute bit):* Memory can be marked as non-executable (also called "modified Harvard Architecture"), so injected code cannot be executed. We define an attacker's extra exploit effort for counteracting NX as c_n.
- *ASLR (Address Space Layout Randomization)*: Memory addresses can be randomized, so attackers cannot easily predict addresses and exploit code might fail. We define the exploit mitigation counter effort against ASLR as c_a.
- *MPU/MMU/PS (Memory Protection Unit / Memory Management Unit / Privilege Separation)*: Most ECUs have processors that watch memory accesses and have operating systems that subdivide code into isolated processes, so a successful attacker cannot access the whole ECU. Once inside a running process, there is a lot more of attack surface to interact with. This depends not only on the software compilation and operating system, but also on the processor security architecture behind the application binary interface as certain ones have a fine grained permission system. We define the effort for counteracting any memory and privilege separation as c_p.
- *Other*: New exploit mitigation techniques are already researched on, e.g., control flow graph checking, see (Kayaalp et al., 2014).

The reflash routine cannot be protected by exploit mitigation techniques as firmware has to be per se executable and highly privileged, i.e., $c = \emptyset$[2].

We define the extra exploit creation effort that is necessary due to exploit mitigation techniques as c. Exploits on complex and primitive input types can be counteracted with all mentioned techniques, i.e., $c \in (c_a \cup c_n \cup c_s \cup c_p)$. Malicious apps can only be counteracted with privilege separation techniques, i.e., $c \in c_p$. And injected malicious privileged code can not be counteracted, i.e., $c = \emptyset$.

2.5 Attack Surface Summary

We conclude the method and metric with summarizing the formulae above. An ECU's vulnerability can be modelled as seen in Section 2.2 with

$$P_B(X > 0) = 1 - (1 - v)^{ix}. \quad (11)$$

The expected finding effort $\bar{f}$ can be computed as seen in Section 2.3 with

$$\bar{f} = f_0 + \bar{f}_i \left(v \sum_{i=1}^{i=k} \left(i\,(1-v)^{i-1} \right) + k\,(1-v)^k \right). \quad (12)$$

[1]C/C++ is still often used, e.g., for time critical deterministic behaviour. Many safety verification and certification tools and methods exist for C/C++.

[2]A reflash routine can still be well protected by strong cryptographic authentication, which implies a high access effort a.

The expected effort $\bar{g}$ for creating a software exploit is the sum of the expected access effort $\bar{a}$, the expected basic effort $\bar{b}$ and the expected countermeasure mitigation effort $\bar{c}$ as seen in Section 2.4 and combined in

$$\bar{g} = \bar{a} + \bar{b} + \bar{c}. \tag{13}$$

Finally, the expected overall ECU exploitation value $\bar{o}$ is

$$\bar{o} = \bar{f} + \bar{g}. \tag{14}$$

3 APPLICATION IN ATTACK GRAPH CONSTRUCTION

The presented method and metric are designed for and integrated into our attack graph generation for vehicular on-board networks. Attack graphs are a well established security engineering method, but the plethora of attack combinations makes manual construction cumbersome and likely to miss relevant attack paths. The automatic inclusion and assessment of the attack surface and vulnerability based on development data raises the accuracy of constructed attack graphs and helps keeping attack graph-based security analyses viable. A missing attack vector might enable extra, unseen attack paths that drastically facilitate compromising a security goal. Security experts can enrich and override automatically compiled information for better soundness; certain factors are better judged by humans, e.g., trending attack vectors or the level of publicly available information on given control units or software components. The attack graph algorithm eventually constructs attack graphs with given and harvested information and will show most likely attack paths; see a run-time example in Figure 3.

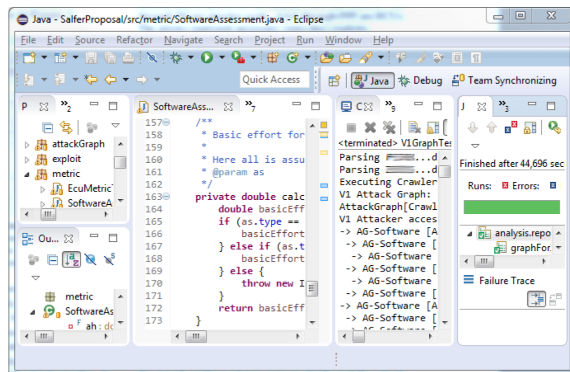

Figure 3: Automated Attack Graph Construction – The integration of our metric and method into our graph-based security analysis facilitates the automatic construction of attack graphs from development data.

We have applied the method and metric of this paper together with our attack tree construction algorithm onto an ECU network and depicted the resulting, simplified, altered and anonymized attack tree in Figure 4.

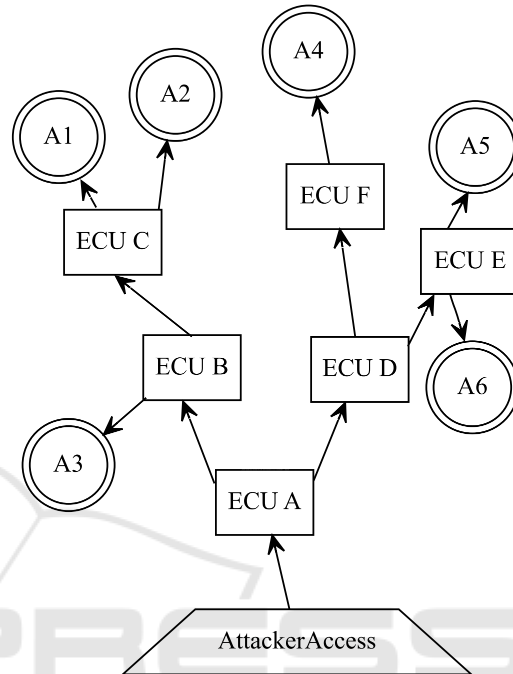

Figure 4: Resulting Attack Tree – The depicted attack graph is a simplified, altered and anonymized assessment result of a modern vehicle's ECU network, based on this paper's method. The *AttackerAccess* is the defined access point for an attacker into the vehicle. *ECU A* through *ECU F* are ECUs. Arrows represent potential attack surface exploitations that are most likely to happen according to the assessment. *A1* through *A6* are assets that an attacker targets.

4 RELATED WORK

Howard et al. started in 2003 identifying Windows services and sockets as well as similar constructs as attacker entry points and weighted them according to their (system or user) privilege and evaluated his method on several Windows versions, as can be seen in (Howard et al., 2005). Manadhata and Wing formalized an attack surface metric for software source code in (Manadhata and Wing, 2011). They measure the attack surface of three classes: methods, channels and files. A typical ECU does not have TCP (Transmission Control Protocol), SSL (Secure Sockets Layer), TLS (Transport Layer Security) or UNIX socket channels, so the channel attack surface is 0. The file system of an ECU is typically not accessible,

so the file attack surface is 0, too. Yet, their attack surface metric is made under the premiss of having access to source code, which we do not have unfortunately. Our metric instead is designed for the outer attack surface of an ECU, where an attacker can only judge from available visible surface such as communication interfaces and firmware code.

Miller and Valasek assessed the attack surface of a couple more cars from publicly available sources by enumerating mentioned features and giving a subjective rating for the criteria outer accessibility, network separation and incorporated features in (Charlie Miller and Chris Valasek, 2014).

The CVSS (Common Vulnerability Scoring System) quantifies how severe a vulnerability is with a base, temporal and environmental metric set. Each criteria is value that can be selected from an array of 3 to 5 numbers according to the verbally given severity. CVSS works only for scoring already existing and found vulnerabilities, yet it cannot assess prospective vulnerability.

Vulnerability databases allow to directly reason about a system's vulnerability in case of a positive finding. Rich research projects and results exists for corporate IT systems, e.g., (Roschke et al., 2009) at the HPI (Hasso-Plattner-Institut) worked on unifying vulnerability information as an input for attack graphs. They sourced the NVD (National Vulnerability Database), parsed textual descriptions and eventually produced homogeneous, machine-readable vulnerability information ready for attack graph construction. Many vulnerability prediction models cover the change in security advisories, i.e., they analyse the advisory output over time, e.g., (Alhazmi et al., 2007). Yet, we cannot use those databases or methods as automotive ECUs are custom made and heavily modified for optimized performance. We needed to use a method that works with black box software.

5 FUTURE WORK

An attack surface estimation becomes more accurate by considering more information. This section describes some additional sources and methods that could be included.

Semi-automated penetration test tools and techniques could reveal further attack surface and estimate its vulnerability. Many penetration test tools specialised for embedded systems were created in recent years and are actively maintained by the security community. For example, the open source tools *binwalk*[3] and *BAT* (Binary Analysis Tool)[4] are firmware software analysis tools and might reveal further attack surface assessment data, e.g., ECU-internal software architecture or run-time information. Collecting data about real-word exploitation attempts may reveal new and additional attack surface.

The method presented in this paper assumes a reasonable mid-level, but fixed, effort for each attack surface. This means an attacker is assumed *not* gaining experience, and vulnerability finding efforts are averaged out evenly over the attack surface. Yet, the effort for testing an attack surface's vulnerability might differ greatly, and an attacker usually gains experience with every attack surface he or she tests. A successor method could differentiate the vulnerability finding effort f for each attack surface type and create formulae and an attacker model for growing experience.

Security is a software attribute that is hard to measure and even harder to validate. Analogously, effort estimations need to be adjusted regularly. One validation method is consulting security experts, as (Manadhata and Wing, 2011) and we did for our attack surface metric; Unfortunately is consulting experts rather subjective. Validation data can be collected from systematic security assessments like penetration tests; especially from their data sets about attacker capabilities and corresponding actual effort for exploiting an ECU. A successor method could include said penetration test data and the defect density predictions of software growth models. A big enough data set about measured actual effort and capabilities for a given ECU would allow a regression analysis for predicting the effort for similar ECUs. Some more data might be generated by software growth models, which source defect trackers and try to predict a software's later defect density. Yet, any data set is only valid for a short time as advances in IT security research and development (especially about reverse engineering) may change the necessary actual effort starkly.

6 CONCLUSION

The method and metric that we presented in this paper shows an approach to a structured evaluation of the attack surface and the potential vulnerability of automotive control units. We compiled and assessed attack surface by collecting input interfaces and estimating vulnerability likelihoods, vulnerability finding efforts, attacker finding probabilities for revealing

[3]binwalk: http://binwalk.org

[4]BAT: http://binaryanalysis.org

a vulnerability and finally an effort estimate for exploiting found vulnerabilities. We presented an attack surface taxonomy and definition that can be applied to the heterogeneous combination of automotive communication channels; We completely abstracted an ECU's communication as potential attack surface that might be susceptible to code injection. We used a vulnerability density (ideally for every input channel separately) for estimating the overall vulnerability likelihood of an ECU. We modelled a concept of reference attacker profile and attack efforts for an extrapolation on other, later-defined attacker profiles. Finally, we gave an attacker effort estimation method for overcoming active exploit mitigation techniques and successfully exploiting an ECU. The result of our method and metric serves as an input for our graph-based security analysis. The integration into it was demonstrated as a proof of concept in Section 3. We thereby showed that our contribution is not purely academic but has also an industrial application. The created method, metric and software helps assessing the security of embedded controller networks.The metric implicitly suggests certain ways of securing ECUs: covering attack surface with firewalls or authentication checks, shrinking an ECUs attack surface by removing services and inbound data and by hardening attack surface with more secure software (stricter compiler and programming languages, more defensive programming and exploit mitigation techniques). The resulting attack surface assessment facilitates a construction of attack graphs for an overall automotive system security assessment.

Table 2: **Symbols Definition**.

Symbol	Definition
$a \in E$	Access Effort
$b \in E$	Basic Exploitation Effort
$c \in E$	Counter Exploit-Mitigation Effort
$d \in]0,1[$	Defect Density
$e \in E$	Effort $E = \{(t,r) \mid t \in T \wedge r \in R\}$
$f \in E$	Vulnerability Finding Effort
$g \in E$	Overall Exploit Creation Effort
$i, j \in \mathbb{N}_{>0}$	A Positive Natural Number
$k, n \in \mathbb{N}_{>0}$	A Positive Natural Number
$o \in E$	Overall ECU Exploitation Effort
P	Probability Distribution Function
$q \in [0,1]$	Probability
$r \in R$	Effort Amount $\in \mathbb{R}_{\geq 0}$
$t \in T$	Capability Tag (a Label)
$v \in]0,1[$	Vulnerability Density
$x \in \mathbb{R}$	An Arbitrary Real Number
X	Probability Random Variable

REFERENCES

Alhazmi, O., Malaiya, Y., and Ray, I. (2005). Security vulnerabilities in software systems: A quantitative perspective. In *Data and Applications Security XIX*, number 3654 in Lecture Notes in Computer Science, pages 281–294. Springer Berlin Heidelberg.

Alhazmi, O. H., Malaiya, Y. K., and Ray, I. (2007). Measuring, analyzing and predicting security vulnerabilities in software systems. *Computers & Security*, 26(3):219–228.

Charlie Miller and Chris Valasek (2014). A survey of remote automotive attack surfaces.

Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., Koscher, K., Czeskis, A., Roesner, F., and Kohno, T. (2011). Comprehensive experimental analyses of automotive attack surfaces. *Proceedings of the 2011 Usenix Security*.

Damm, W., Achatz, R., Beetz, K., Broy, M., Daembkes, H., Grimm, K., and Liggesmeyer, P. (2010). Nationale roadmap embedded systems. In Broy, M., editor, *Cyber-Physical Systems*, acatech DISKUTIERT, pages 67–136. Springer Berlin Heidelberg.

Francillon, A., Danev, B., and Capkun, S. (2010). Relay attacks on passive keyless entry and start systems in modern cars. In *Proceedings of NDSS*.

Howard, M., Pincus, J., and Wing, J. M. (2005). Measuring relative attack surfaces. In *Computer Security in the 21st Century*, pages 109–137. Springer US.

Ishtiaq Rouf, R. M., Mustafa, H., Travis Taylor, S. O., Xu, W., Gruteser, M., Trappe, W., and Seskar, I. (2010). Security and privacy vulnerabilities of in-car wireless networks: A tire pressure monitoring system case study. In *19th USENIX Security Symposium, Washington DC*, pages 11–13.

Kayaalp, M., Ozsoy, M., Ghazaleh, N., and Ponomarev, D. (2014). Efficiently securing systems from code reuse attacks. *IEEE Transactions on Computers*, 63(5):1144–1156.

Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., and Savage, S. (2010). Experimental security analysis of a modern automobile. In *IEEE Symposium on Security and Privacy*, pages 447–462.

Manadhata, P. and Wing, J. (2011). An attack surface metric. *IEEE Transactions on Software Engineering*, 37(3):371–386.

Roschke, S., Cheng, F., Schuppenies, R., and Meinel, C. (2009). Towards unifying vulnerability information for attack graph construction. In *12th International Conference on Information Security*, ISC 2009, pages 218–233, Berlin, Heidelberg. Springer-Verlag.

Schneier, B. (2012). The importance of security engineering. *IEEE Security & Privacy*, 10(5):88–88.