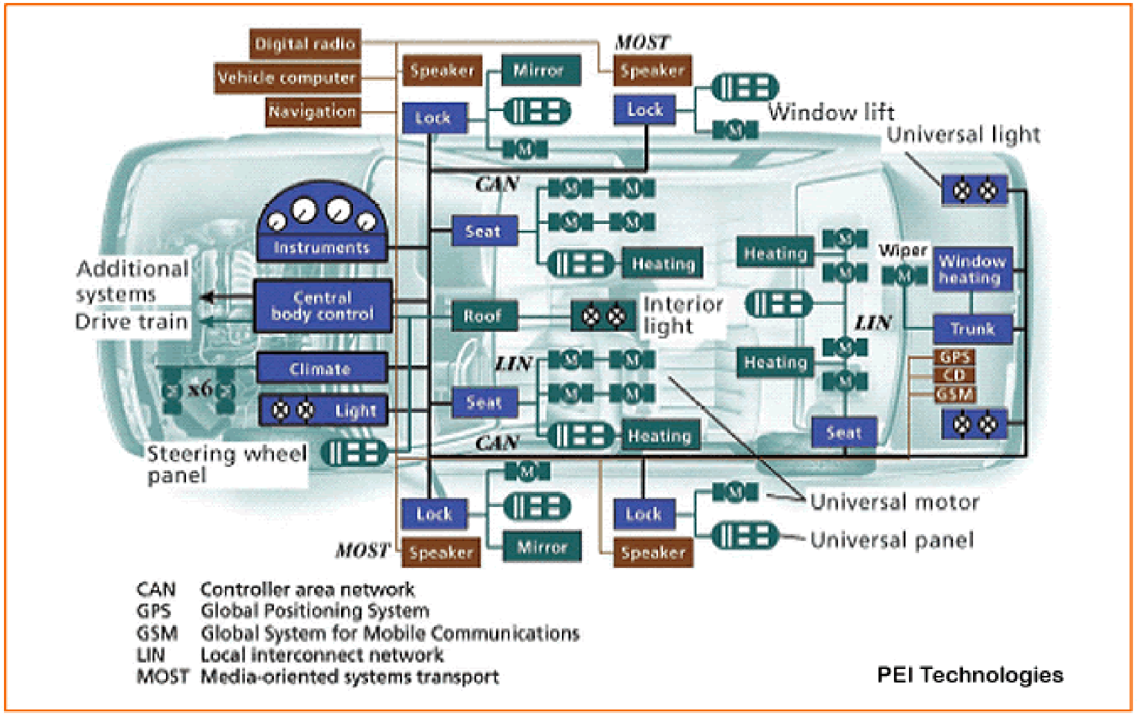
Hacking cars in the style of Stuxnet

András Szijj[1], Levente Buttyán[1], Zsolt Szalay[2]
[1] CrySyS Lab, Department of Networked Systems and Services
[2] Department of Automobiles and Vehicle Manufacturing
Budapest University of Technology and Economics

Introduction

- modern cars are full of embedded controllers (ECUs)
- they are connected by internal networks (e.g., CAN)
- they have a number of external interfaces (e.g., Bluetooth, GPS, ...)

→ cyber attacks against cars became a plausible threat



Checkoway et al. (Usenix Security, 2011)

Comprehensive Experimental Analyses of Automotive Attack Surfaces

Stephen Checkoway, Damon McCoy, Brian Kantor,
Danny Anderson, Hovav Shacham, and Stefan Savage
University of California, San Diego

Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno
University of Washington

Abstract

Modern automobiles are pervasively computerized, and hence potentially vulnerable to attack. However, while previous research has shown that the *internal* networks within some modern cars are insecure, the associated threat model — requiring *prior physical access* — has justifiably been viewed as unrealistic. Thus, it remains an open question if automobiles can also be susceptible to *remote* compromise. Our work seeks to put this question to rest by systematically analyzing the *external* attack

This situation suggests a significant gap in knowledge, and one with considerable practical import. To what extent are external attacks possible, to what extent are they practical, and what vectors represent the greatest risks? Is the etiology of such vulnerabilities the same as for desktop software and can we think of defense in the same manner? Our research seeks to fill this knowledge gap through a systematic and empirical analysis of the remote attack surface of late model mass-production sedan.

We make four principal contributions:

Checkoway et al. (Usenix Security, 2011)

- the paper shows that cars can be compromised remotely
  - systematic overview of the attack surface
    - indirect physical access (e.g., mechanics tools, CD players)
    - short range wireless access (e.g., Bluetooth, WiFi, wireless TPM)
    - long range wireless access (e.g., cellular)
  - proof-of-concept demonstrations for all possible attack vectors
    - vulnerable diagnostics equipment widely used by mechanics
    - media player playing a specially modified song in WMA format
    - vulnerabilities in hands-free Bluetooth functionality
    - calling the car’s cellular modem and playing a carefully crafted audio signal encoding both an exploit and a bootstrap loader for additional remote-control functionality

Work by Charlie Miller and Chris Valasek

MONDAY, AUGUST 5, 2013

Car Hacking: The Content

By Chris Valasek *@nudehaberdasher* ***and Charlie Miller*** *@0xcharlie*

Hi Everyone,
As promised, Charlie and I are releasing all of our tools and data, along with our white paper. We hope that these items will help others get involved in automotive security research. The paper is pretty refined but the tools are a snapshot of what we had. There are probably some things that are deprecated or do not work, but things like ECOMCat and ecomcat_api should really be all you need to start with your projects. Thanks again for all the support!

Content: http://illmatics.com/content.zip

Paper:
http://www.ioactive.com/pdfs/IOActive_Adventures_in_Automotive_Networks_and_Control_Units.pdf

→

Adventures in Automotive Networks and Control Units

By Dr. Charlie Miller & Chris Valasek

Work by Charlie Miller and Chris Valasek

Work by Charlie Miller and Chris Valasek

WIRED Hackers Remotely Kill a Jeep on the Highway—With Me in It SUBSCRIBE

BUSINESS | DESIGN | ENTERTAINMENT | GEAR | SCIENCE | SECURITY

ANDY GREENBERG SECURITY 07.21.15 6:00 AM

HACKERS REMOTELY KILL A JEEP ON THE HIGHWAY—WITH ME IN IT

- remote wireless control over the Internet
- exploiting a bug in the car's WiFi hotspot

Other examples

IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY, VOL. 8, NO. 6, JUNE 2013

FM 99.9, Radio Virus: Exploiting FM Radio Broadcasts for Malware Deployment

Earlence Fernandes, Bruno Crispo, *Senior Member, IEEE*, and Mauro Conti, *Member, IEEE*

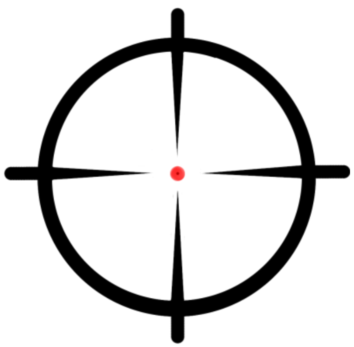
Security and Privacy Vulnerabilities of In-Car Wireless Networks: A Tire Pressure Monitoring System Case Study

Ishtiaq Rouf[a], Rob Miller[b], Hossen Mustafa[a], Travis Taylor[a], Sangho Oh[b]
*Wenyuan Xu[a], Marco Gruteser[b], Wade Trappe[b], Ivan Seskar[b] **

Relay Attacks on Passive Keyless Entry and Start Systems in Modern Cars

Aurélien Francillon, Boris Danev, Srdjan Capkun
Department of Computer Science
ETH Zurich
8092 Zurich, Switzerland
{aurelien.francillon, boris.danev, srdjan.capkun}@inf.ethz.ch

Other examples



remote attacks

Putting things in perspectives

- remote attacks
  - are intriguing and scary
  - can attract media attention
  - but their real risk is unclear...
    - need exploitable vulnerability in an interface (e.g., GSM module)
    - finding such vulnerabilities is far from being trivial
      - reverse engineering embedded software (→ difficult)
      - very limited availability of information (→ frustrating)
      - risk of bricking relatively expensive equipment (→ expensive)
    - may not scale
      - a vulnerability in one brand may not work in any other brands of cars

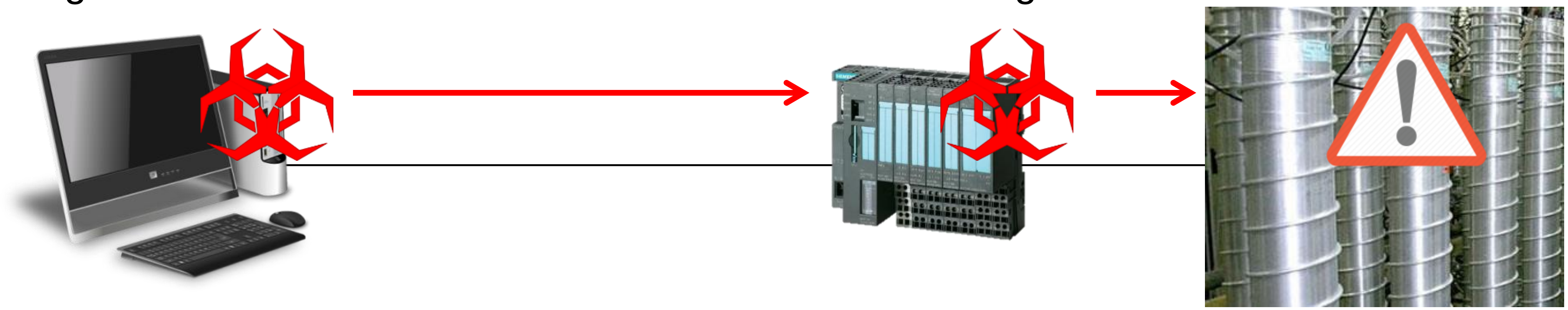
- is there some fruits hanging lower than remote attacks?

How Stuxnet worked?

PC running WinCC PLC management software

PLC controlling the uranium centrifuges

uranium centrifuges

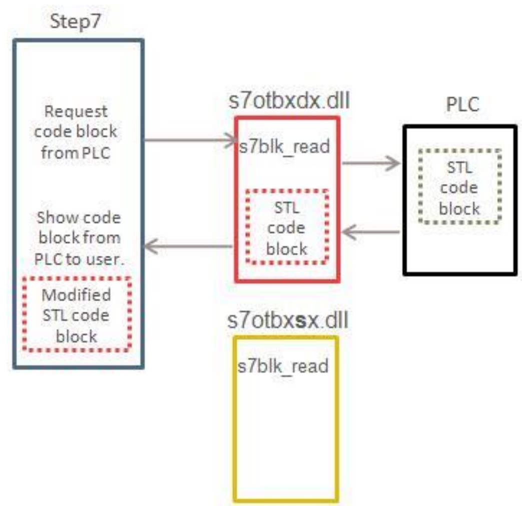


Stuxnet infected PCs, and took over the communication between the PC and the PLC

then modified the PLC program

modified program destroyed centrifuges

- exploited vulnerabilities in Windows
- replaced the DLL responsible for communications with the PLC

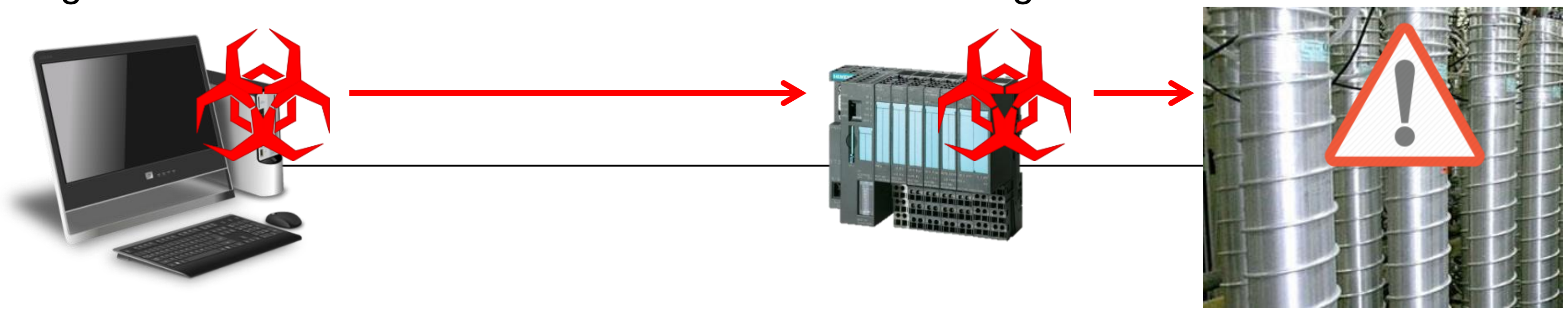


A blueprint for attacking embedded system

PC running WinCC PLC management software

PLC controlling the uranium centrifuges

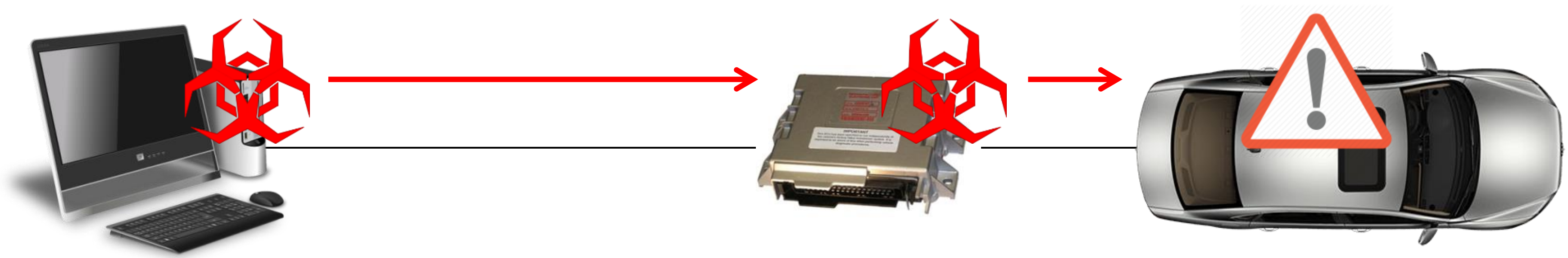
uranium centrifuges



PC running a vehicle diagnostic software

ECU controlling some function of the vehicle

vehicle



Why is this worrisome?

- PCs in repair shops and garages are vulnerable
  - probably connected to the Internet
  - probably allow for connecting USB sticks
  - probably poorly maintained and administered
  - probably used not only for running diagnostic programs

  → it is relatively easy to infect them even with known malware

- malware can compromise diagnostic applications, and implement stealth functionality
  - almost direct access to internal components (via the OBD2 interface)
  - mainly needs standard reverse engineering skills in a PC environment
  - does not require special car electronics know-how

- this scales better than remote attacks
  - same software is usually compatible with multiple different car brands
  - every car is taken to the repair shop regularly

Proof of Concept

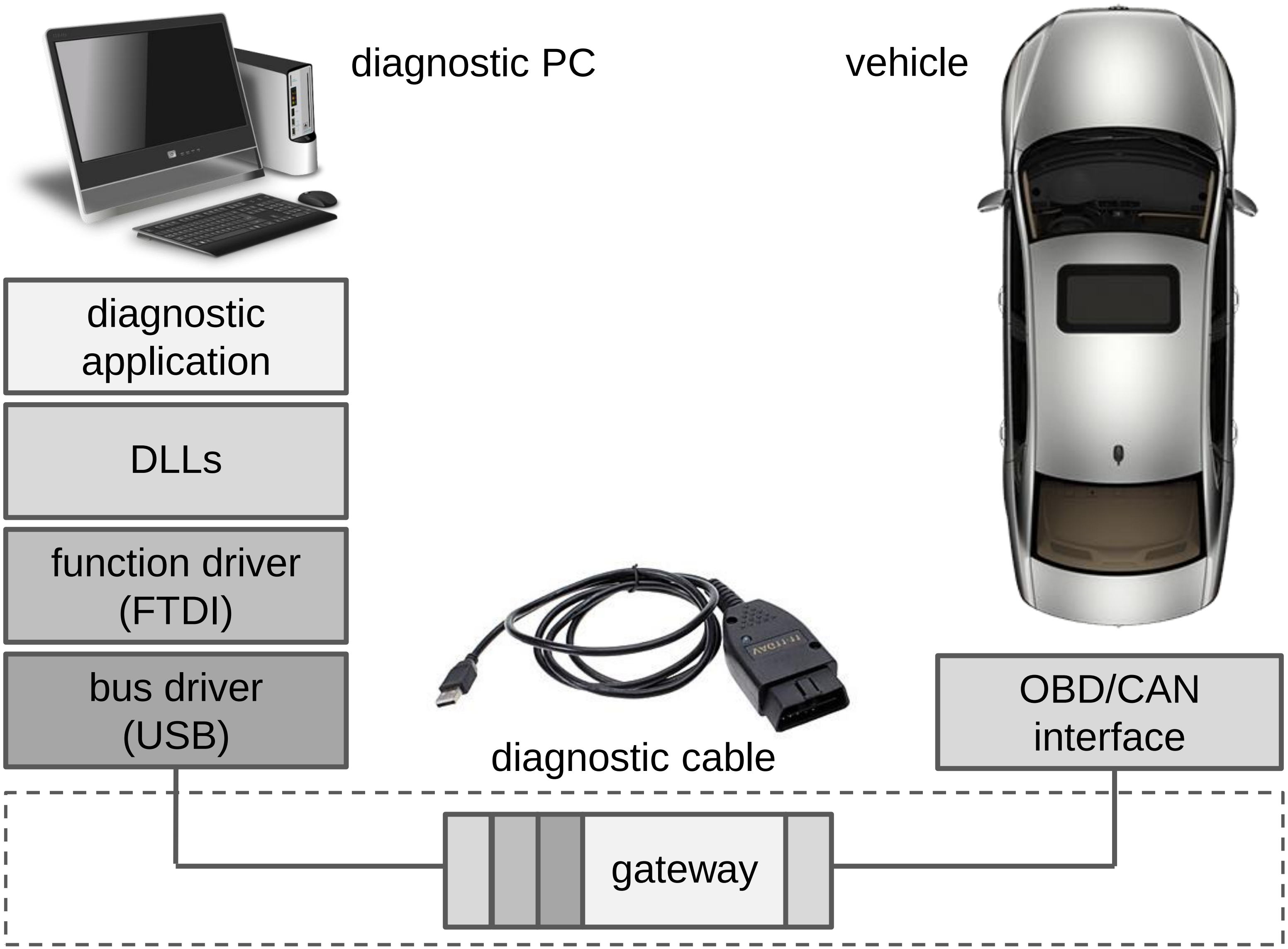
- objective:
  - demonstrate *in practice* that a Stuxnet-style attack is easy to implement against cars by minimal modification of a diagnostic application
    - in our test environment we had access to an Audi TT
    - we have chosen a widely-used, third-party diagnostic application that is compatible with cars from the Volkswagen group
  - the modifications should allow for Man-in-the-Middle attacks between the application and the car (i.e., eavesdropping and modifying messages stealthily)

- assumptions:
  - we assume that the PC that runs the diagnostic application is already infected by malware
  - the malware can carry out the modifications we propose on the diagnostic application
  - the diagnostic application has the necessary licenses and credentials to access the car when connected via the appropriate diagnostic cable (available in the repair shop)

Outline

- system model
- protection mechanisms
- attack techniques
  - our DLL replacement attack
  - protocol reverse engineering
    - message formats
    - checksum computation
    - encryption scheme
  - man-in-the-middle attacks
    - logging and replaying sessions
    - modifying messages on-the-fly
  - experiments
- conclusions and outlook

System model



System model

diagnostic PC

vehicle

diagnostic application

DLLs

function driver (FTDI)

bus driver (USB)

holds necessary keys and protocol implementations for accessing a wide variety of cars via the OBD interface

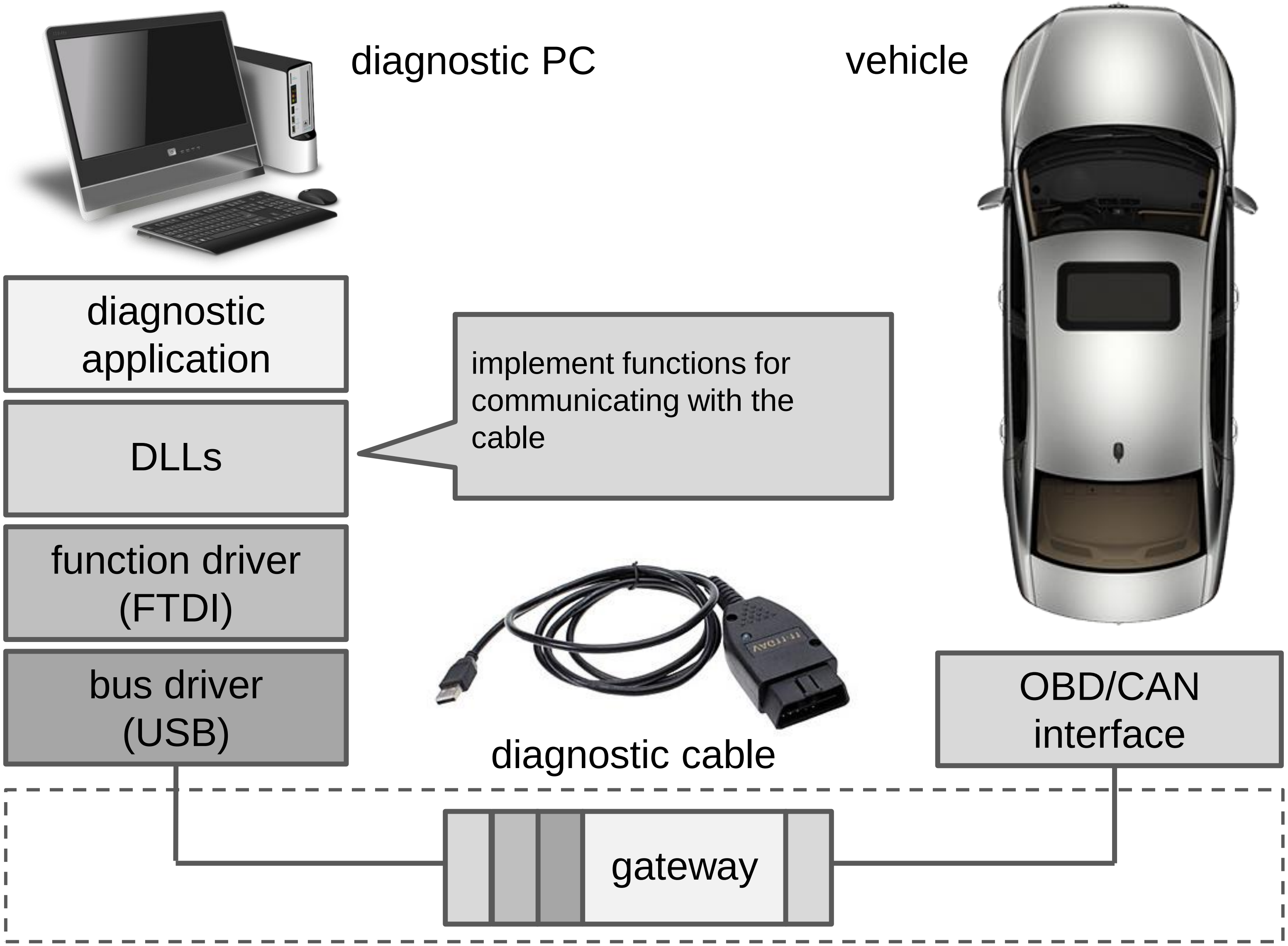
implements diagnostic functions (reading and setting parameters)

diagnostic cable

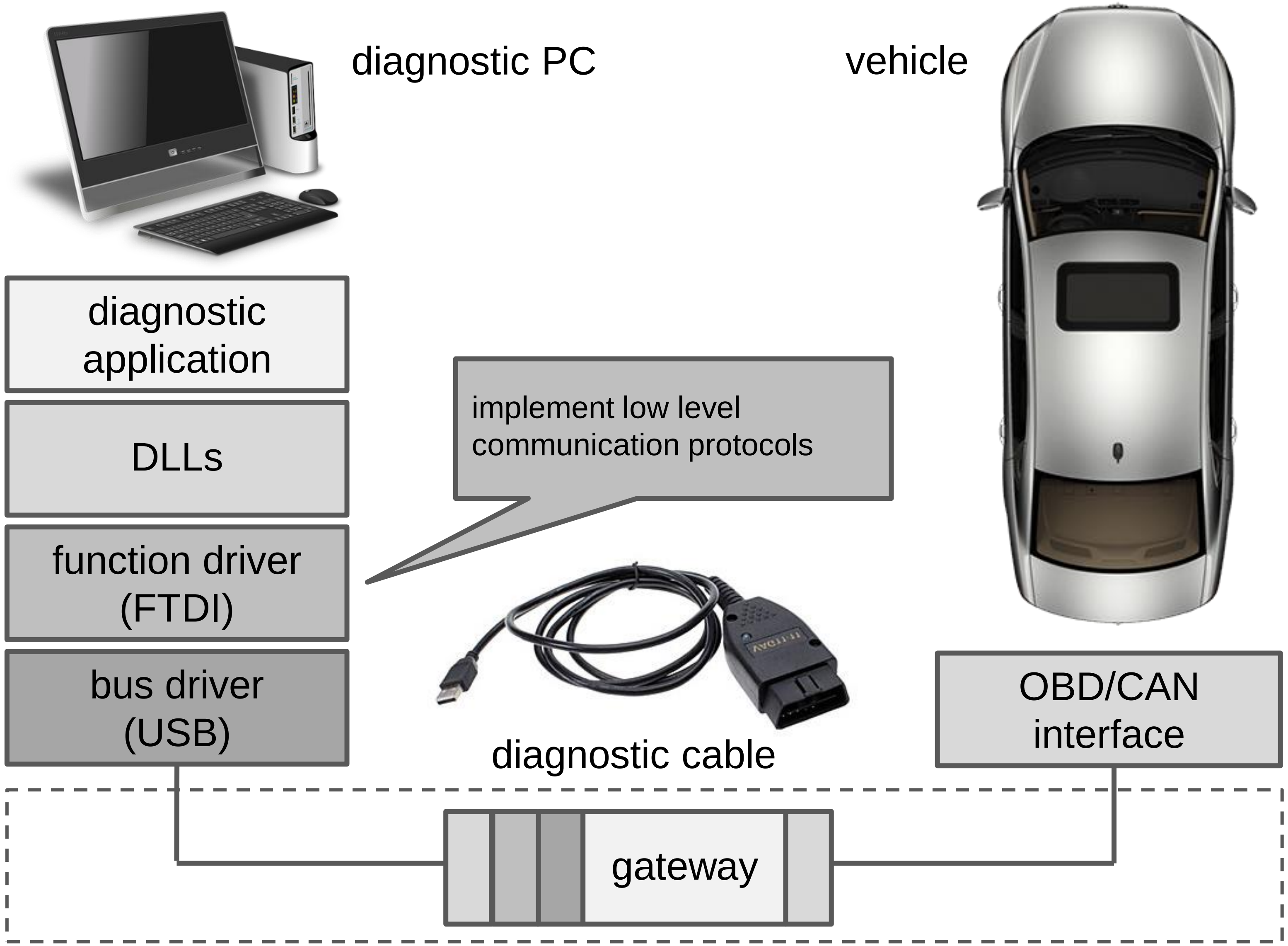
OBD/CAN interface

gateway

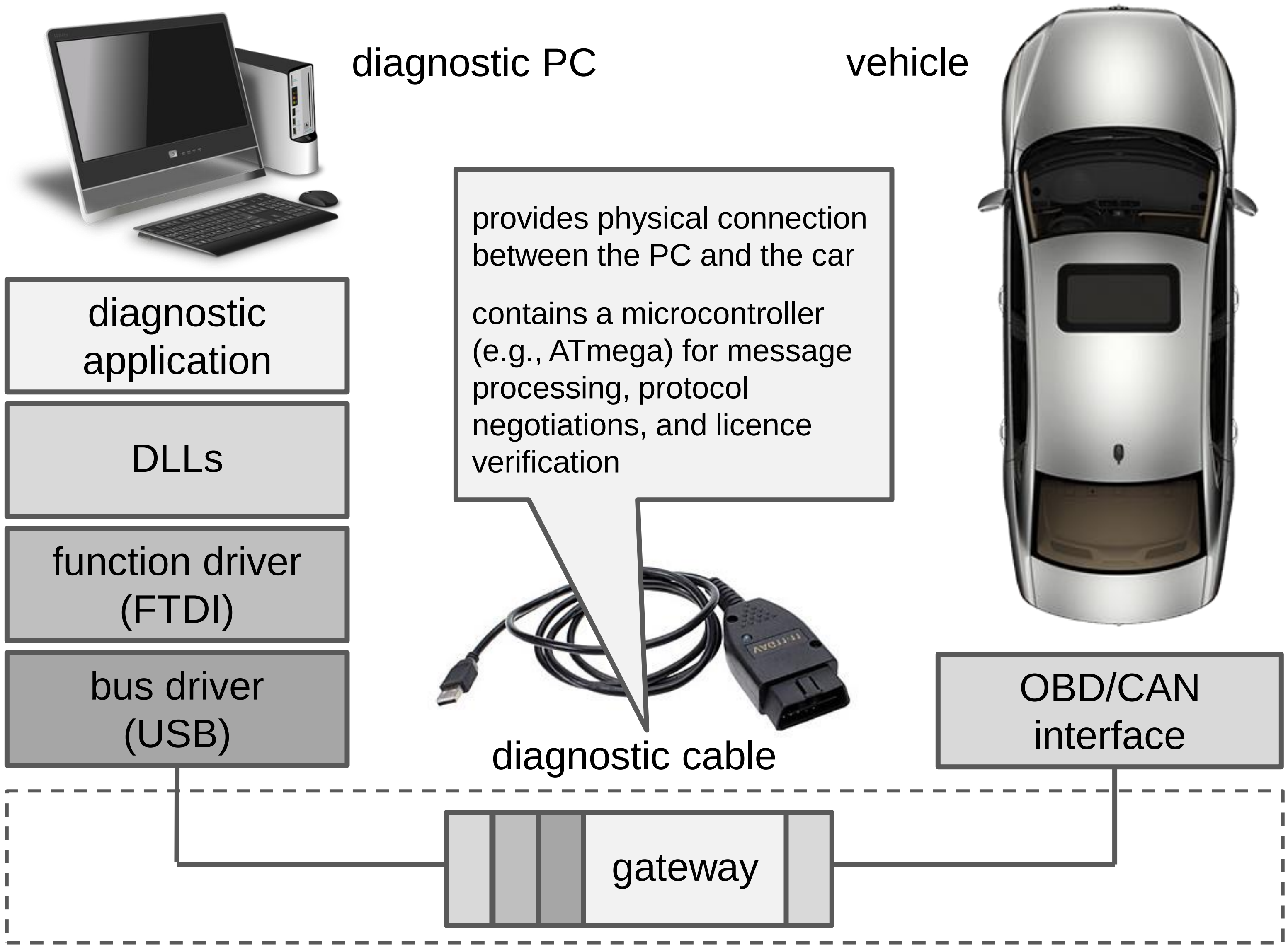
System model



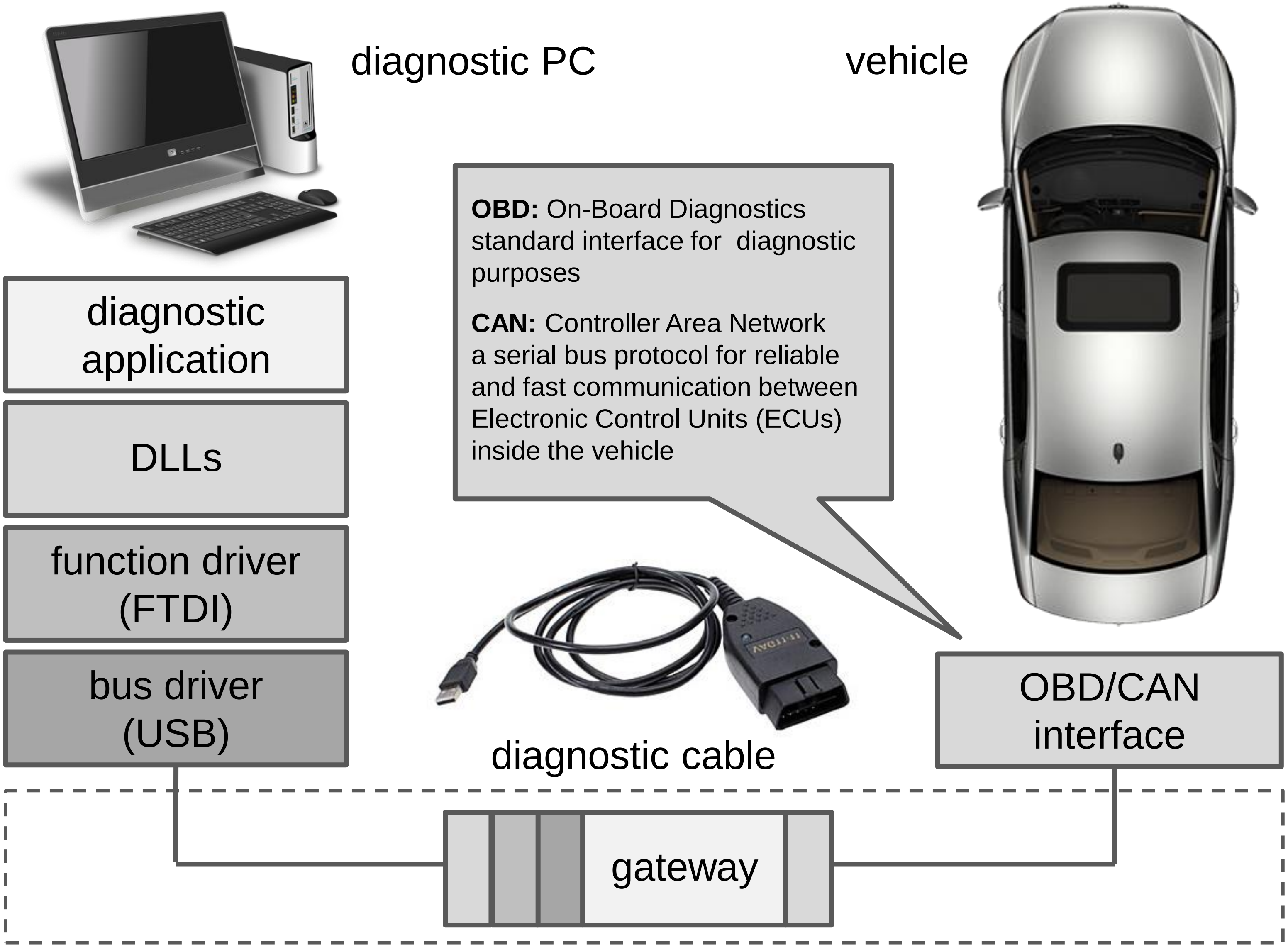
System model



System model



System model



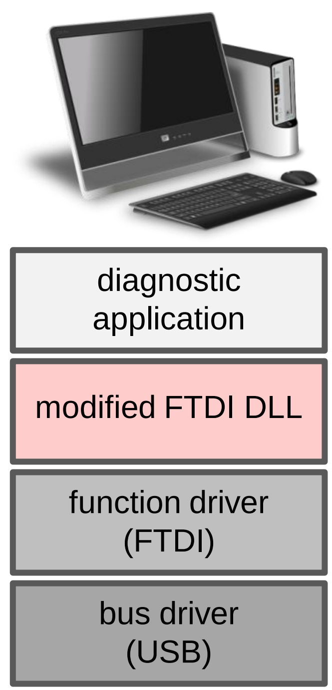
Protection mechanisms

- signed DLLs
  - all the DLLs loaded by the diagnostic software are digitally signed
  - however, signatures on DLLs are not checked (or perhaps checked ”silently”)
- program obfuscation
  - the executable (PE) of the diagnostic software is obfuscated with some ”commonly used” methods to prevent static analysis
  - however, the program de-obfuscates itself in memory when launched
  - so, we could access the de-obfuscated binary by attaching a debugger to the running program when its window was displayed on the screen
- license verification
  - the running application reads specific memory blocks from the microcontroller in the cable that contains the license, and also from the FTDI chip's EEPROM
  - it also performs some challenge-response type authentication during cable initialization
  - the cable we bought on-line (for a few tens of dollars) was verified successfully

Implementing a Man-in-the-Middle attack

- our goal was to implement a man-in-the-middle component between the diagnostic application and the vehicle, which can
  - eavesdrop communications (can help reverse engineering the protocol)
  - play back recorded messages to the car or to the diagnostic application
  - inject fake messages in the car

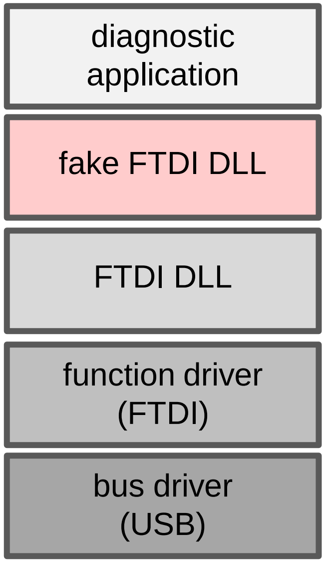
- one option was to modify the FTDI DLL (binary) loaded by the application
  - no strong verification of loaded 3rd party DLLs



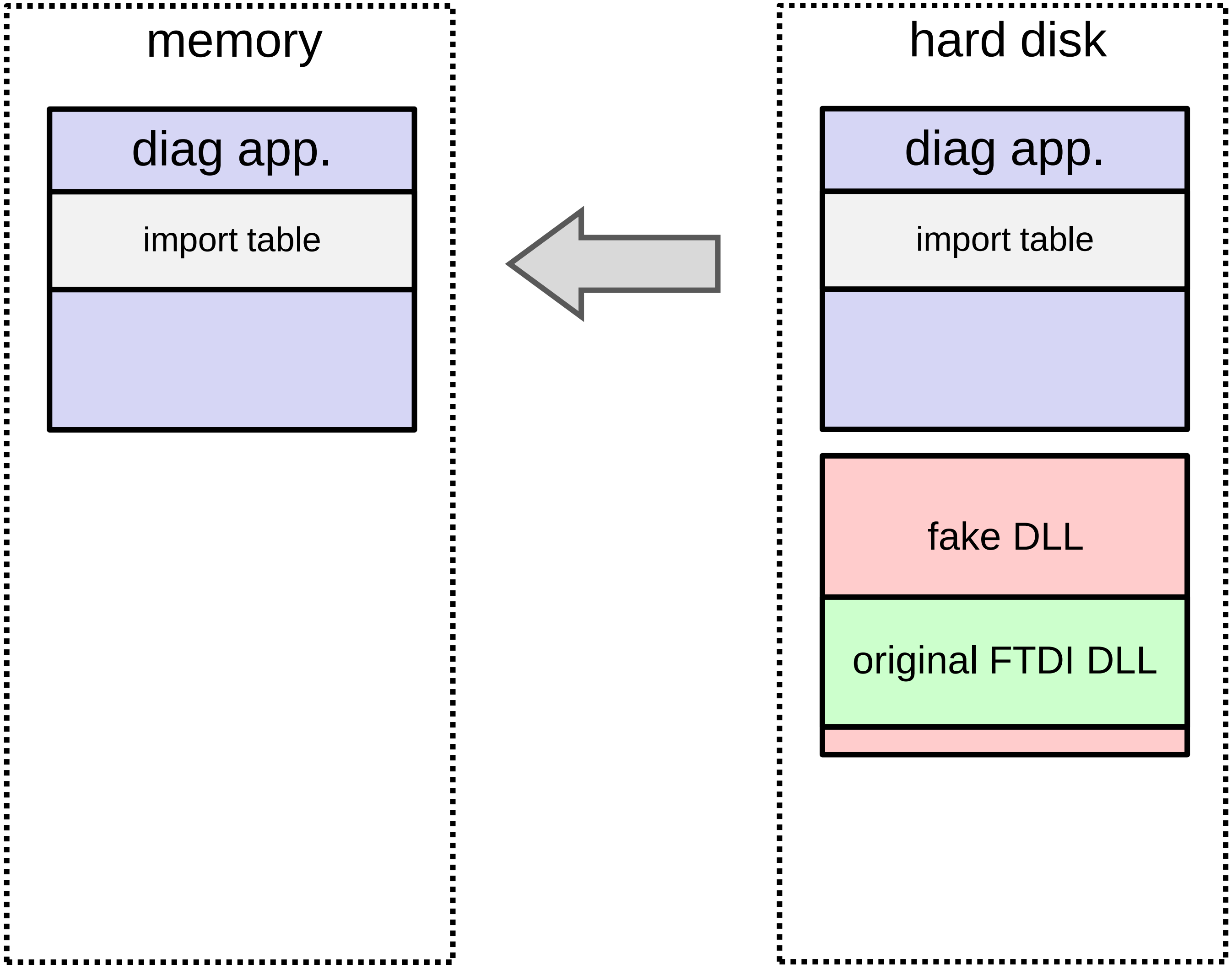
Implementing a Man-in-the-Middle attack

- our goal was to implement a man-in-the-middle component between the diagnostic application and the vehicle, which can
  - eavesdrop communications (can help reverse engineering the protocol)
  - play back recorded messages to the car or to the diagnostic application
  - inject fake messages in the car

- one option was to modify the FTDI DLL (binary) loaded by the application
  - no strong verification of loaded 3rd party DLLs
- but it seemed even easier to create our own fake FTDI DLL that tampers with the messages and then redirects calls to the original FTDI DLL



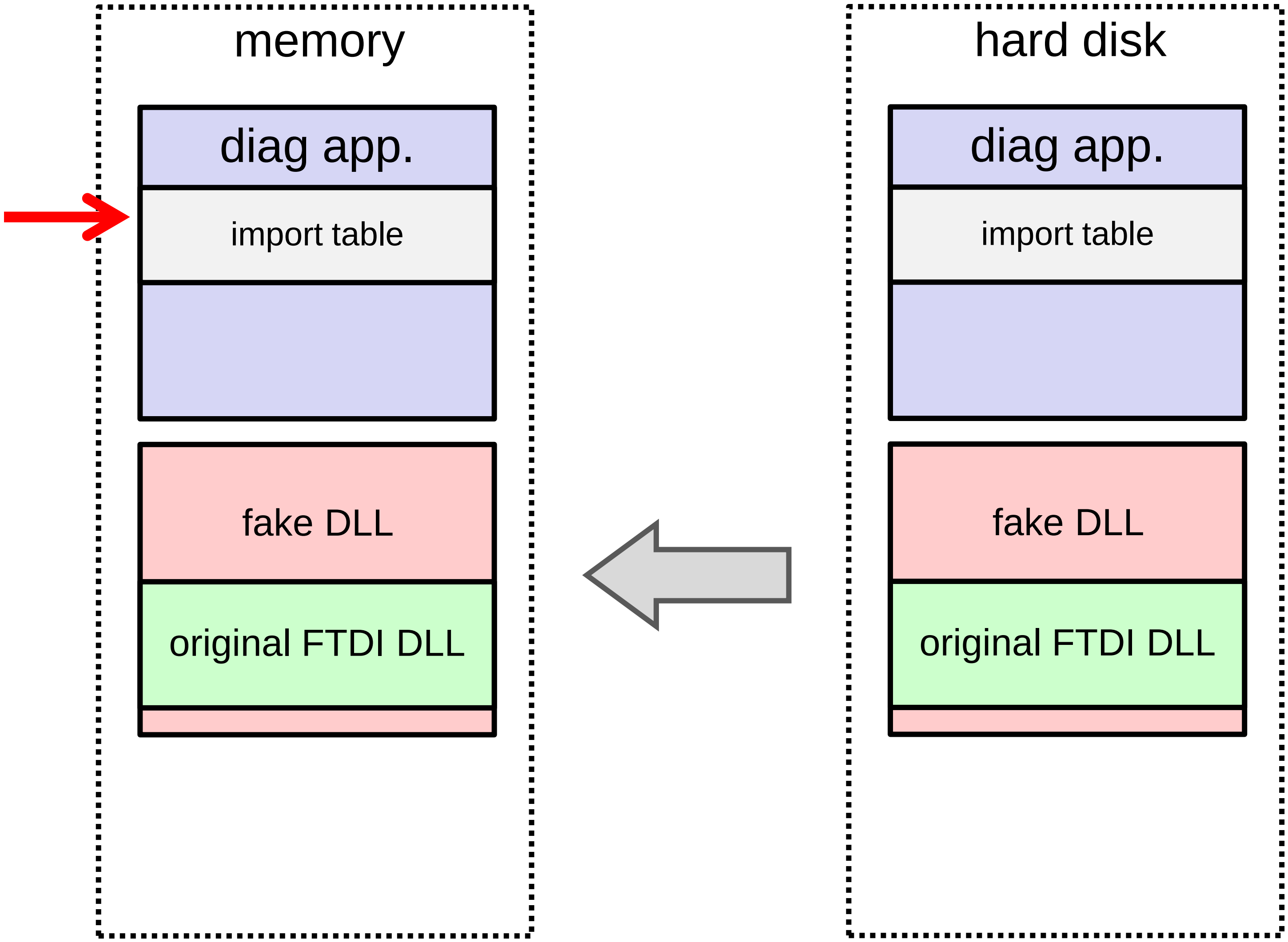
DLL replacement



fake DLL with original name

contains original DLL in a resource section

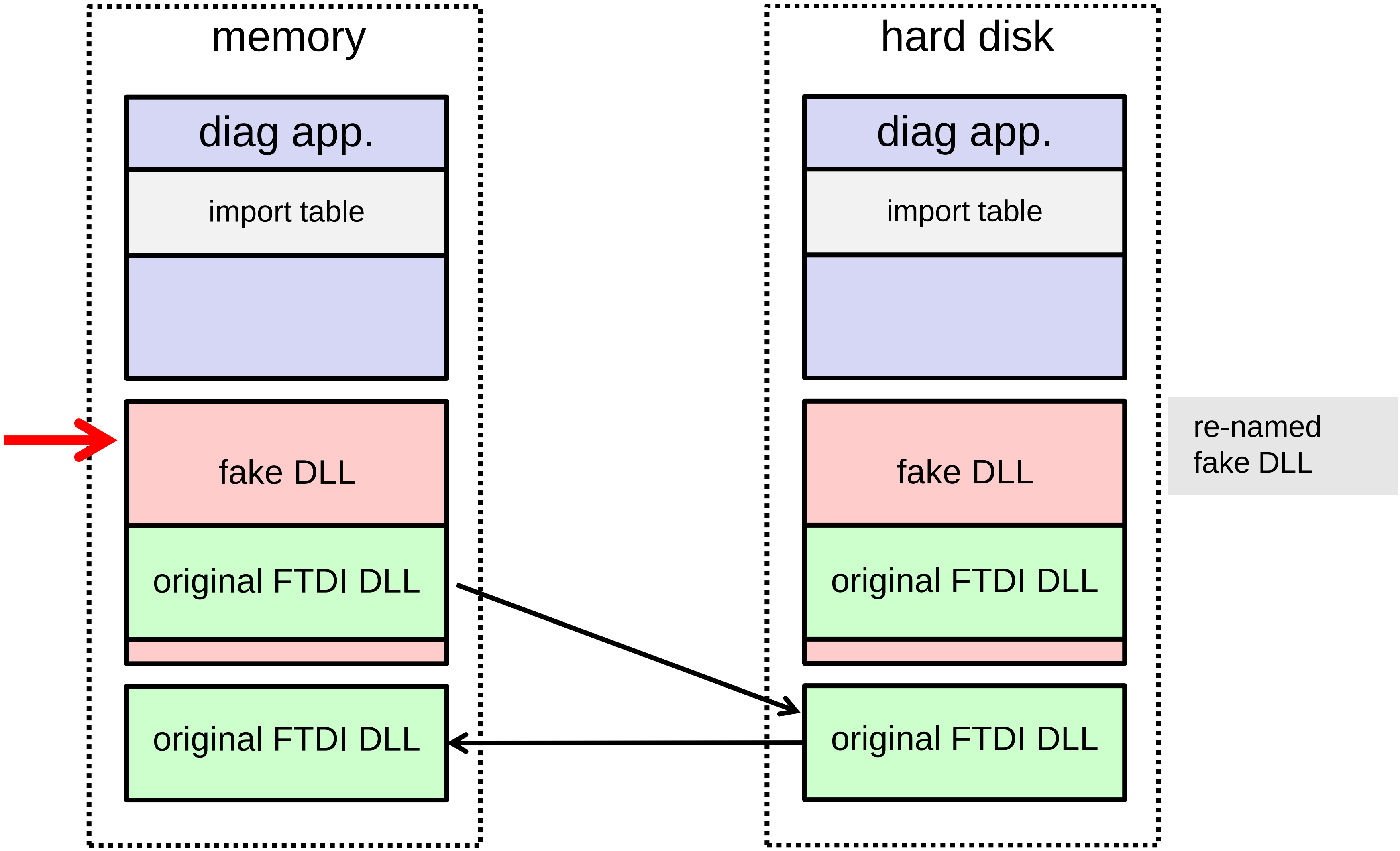
DLL replacement



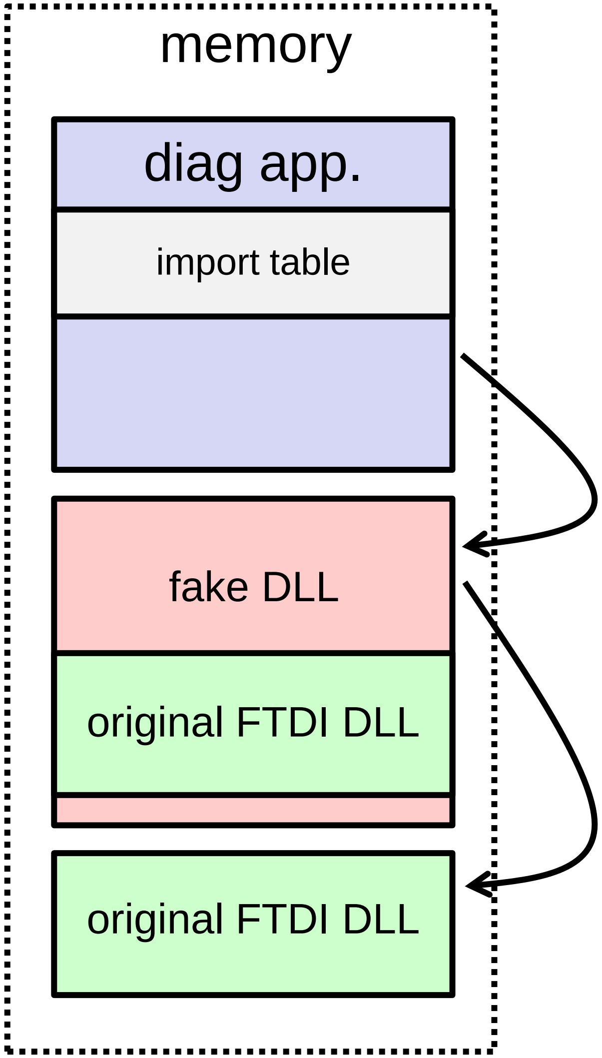
fake DLL with original name

contains original DLL in a resource section

DLL replacement



DLL replacement

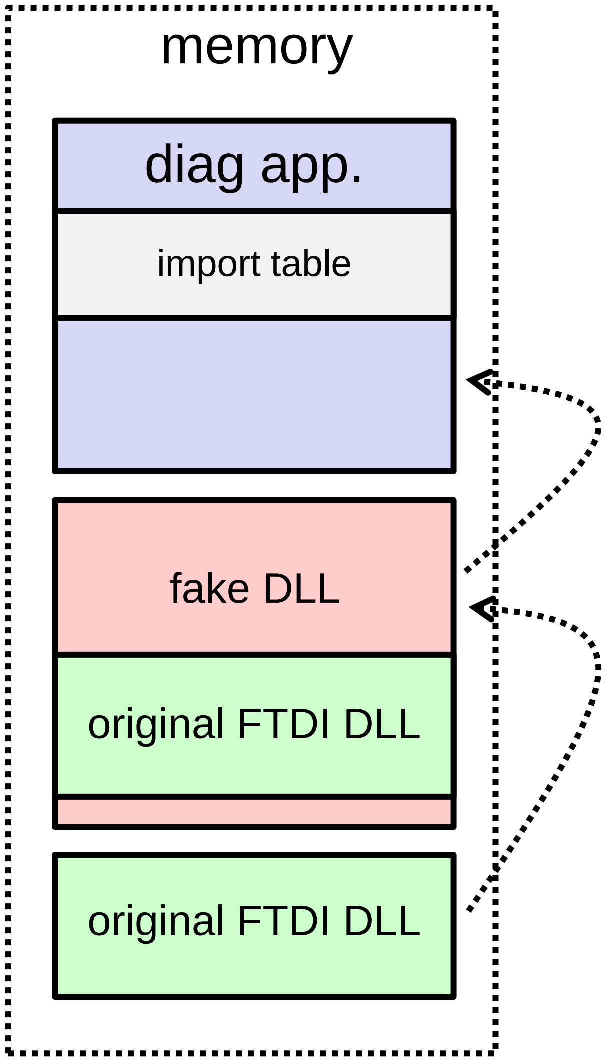


when application calls the read or the write function

our code performs MitM

and redirects the call to the original function

DLL replacement



when the call returns,
we have control again

Could have been made harder...

- verification of digital signature and CRC should be performed before loading any external components (DLLs, data files, …)
- after loading, integrity of external components in memory should be checked regularly
- these checks shouldn't be triggered by just some condition, but one should rather integrate them into many of the calculations
- common or close to common protocol implementations should be avoided, and unusual and proprietary solutions should also be used
- one should use proper cryptography in an appropriate way
  - XORing with a static mask is not proper crypto

Protocol reverse engineering

- to reverse engineer the protocol we used the following:
  - our fake FTDI DLL to capture data between the diagnostic application and the cable
    - we used modified versions of the original FTDI DLL exports
      - modified FT_Read to capture incoming messages
      - modified FT_Write to capture outgoing messages
  - OllyDbg 2.01 to reverse the diagnostic application
  - CheatEngine 6.0 for memory scanning
  - HxD 1.7.7.0 hex editor to view/edit captured data
  - and some handmade tools for filtering captured data

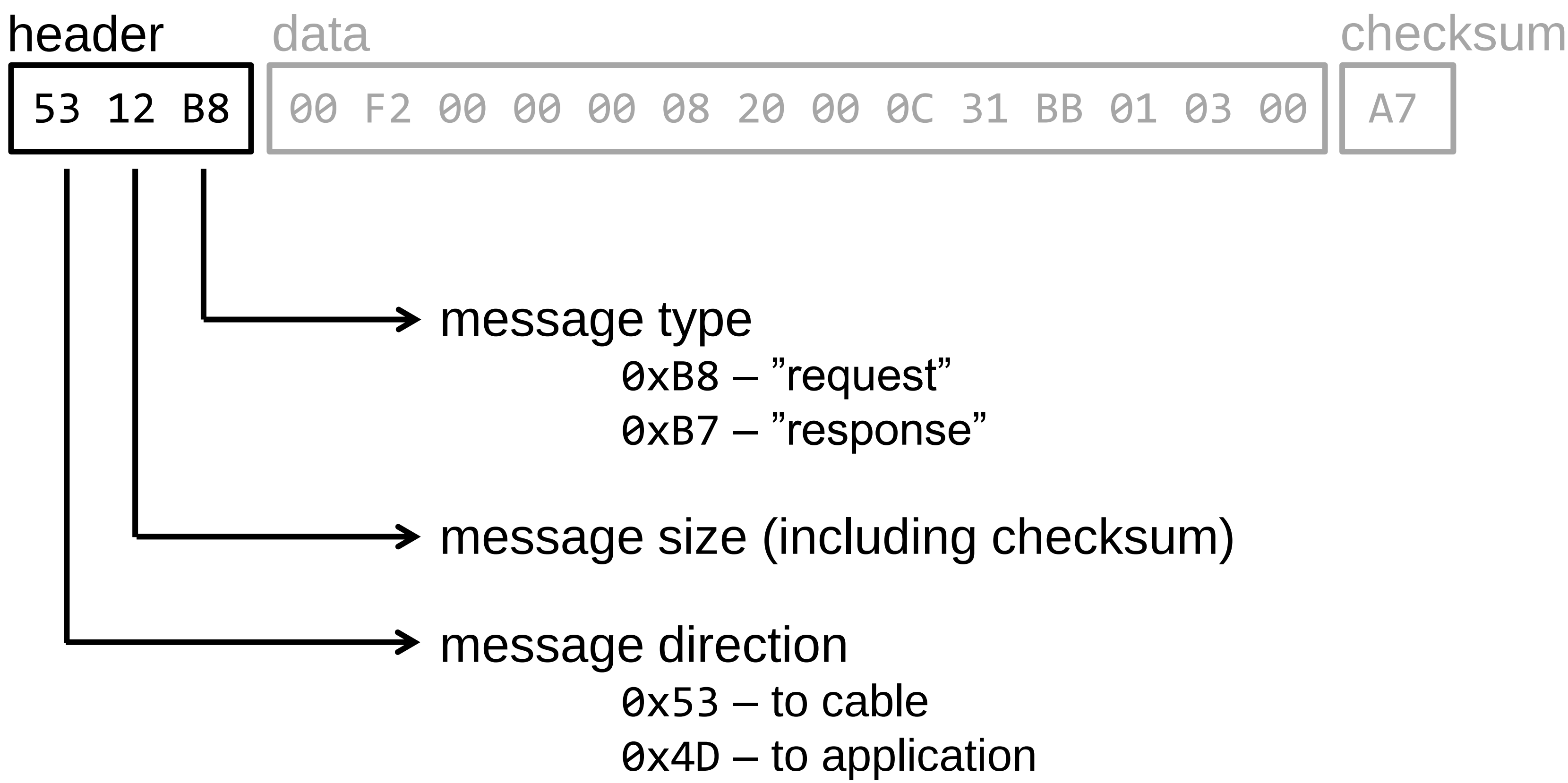
Protocol reverse engineering

- ordinary messages

header	data	checksum
53 12 B8	00 F2 00 00 00 08 20 00 0C 31 BB 01 03 00	A7

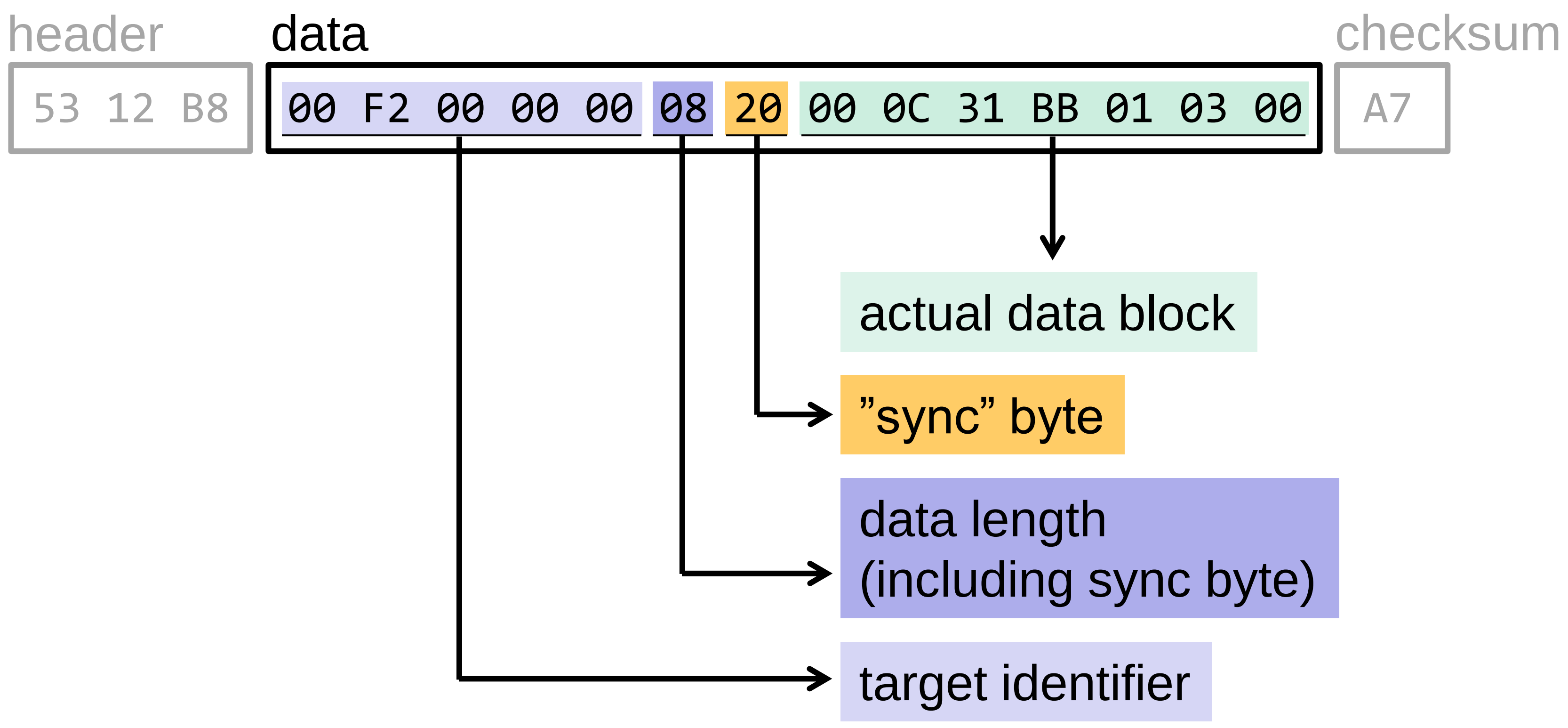
Protocol reverse engineering

- ordinary messages



Protocol reverse engineering

- ordinary messages



Protocol reverse engineering

- ordinary messages

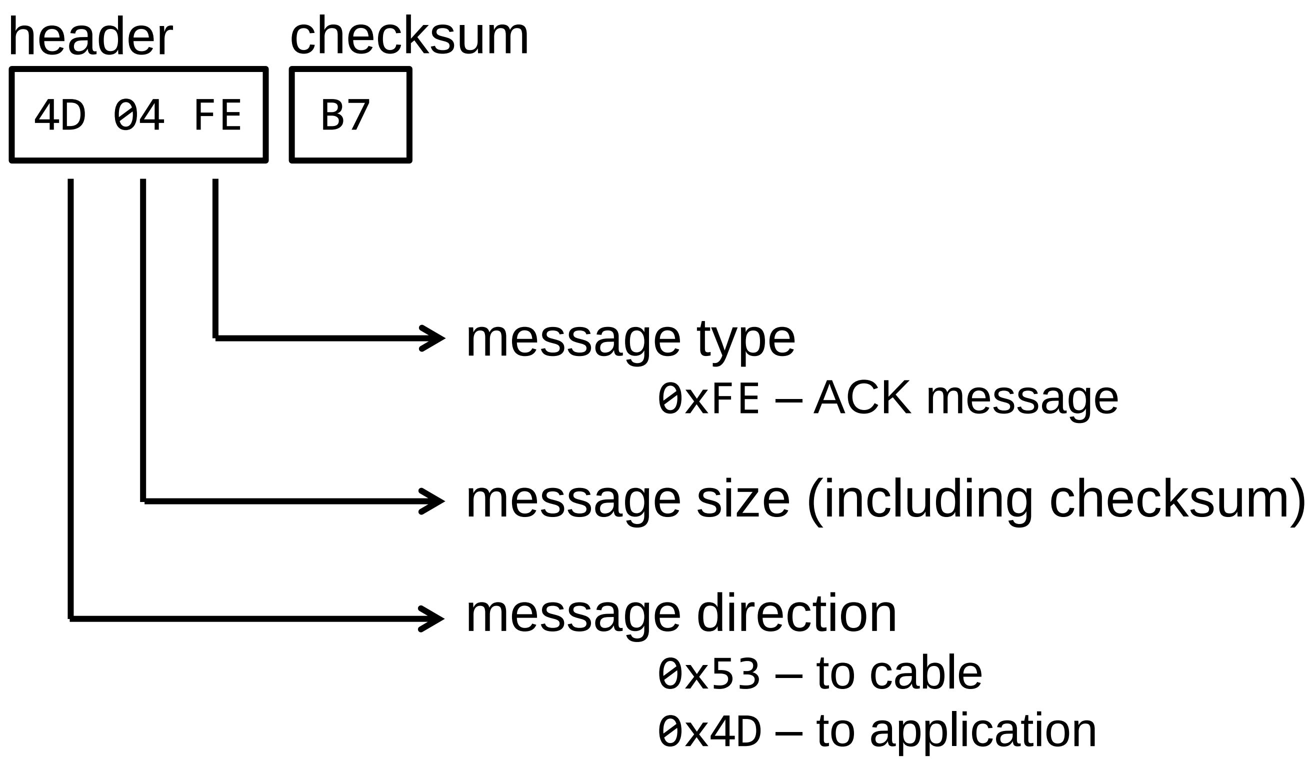
header	data	checksum
53 12 B8	00 F2 00 00 00 08 20 00 0C 31 BB 01 03 00	A7

checksum byte computed as the XOR of all bytes of the message

e.g., `0x53 + 0x12 + 0xB8 + ... + 0x00 = 0xA7`

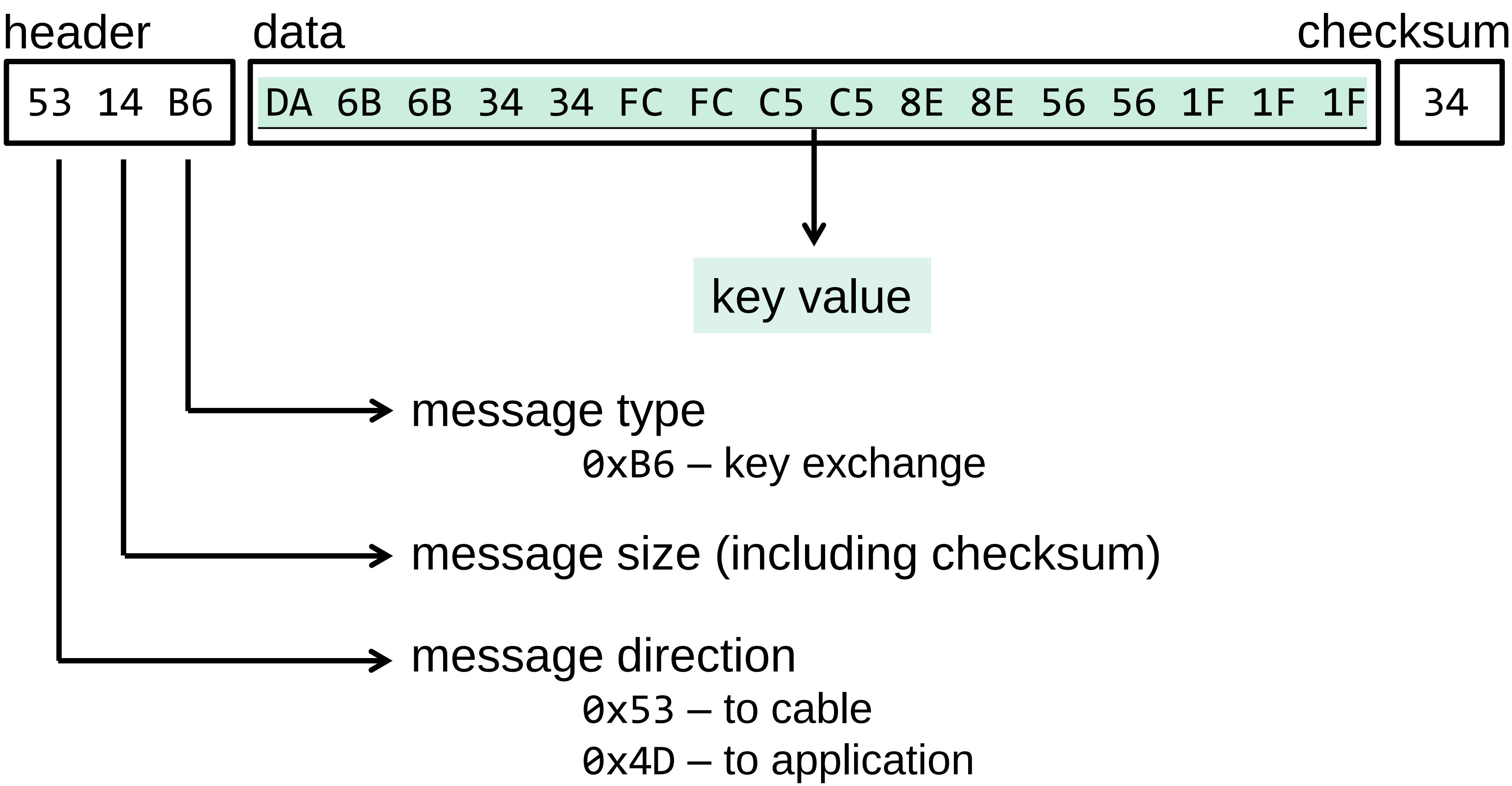
Protocol reverse engineering

- ACK message



Protocol reverse engineering

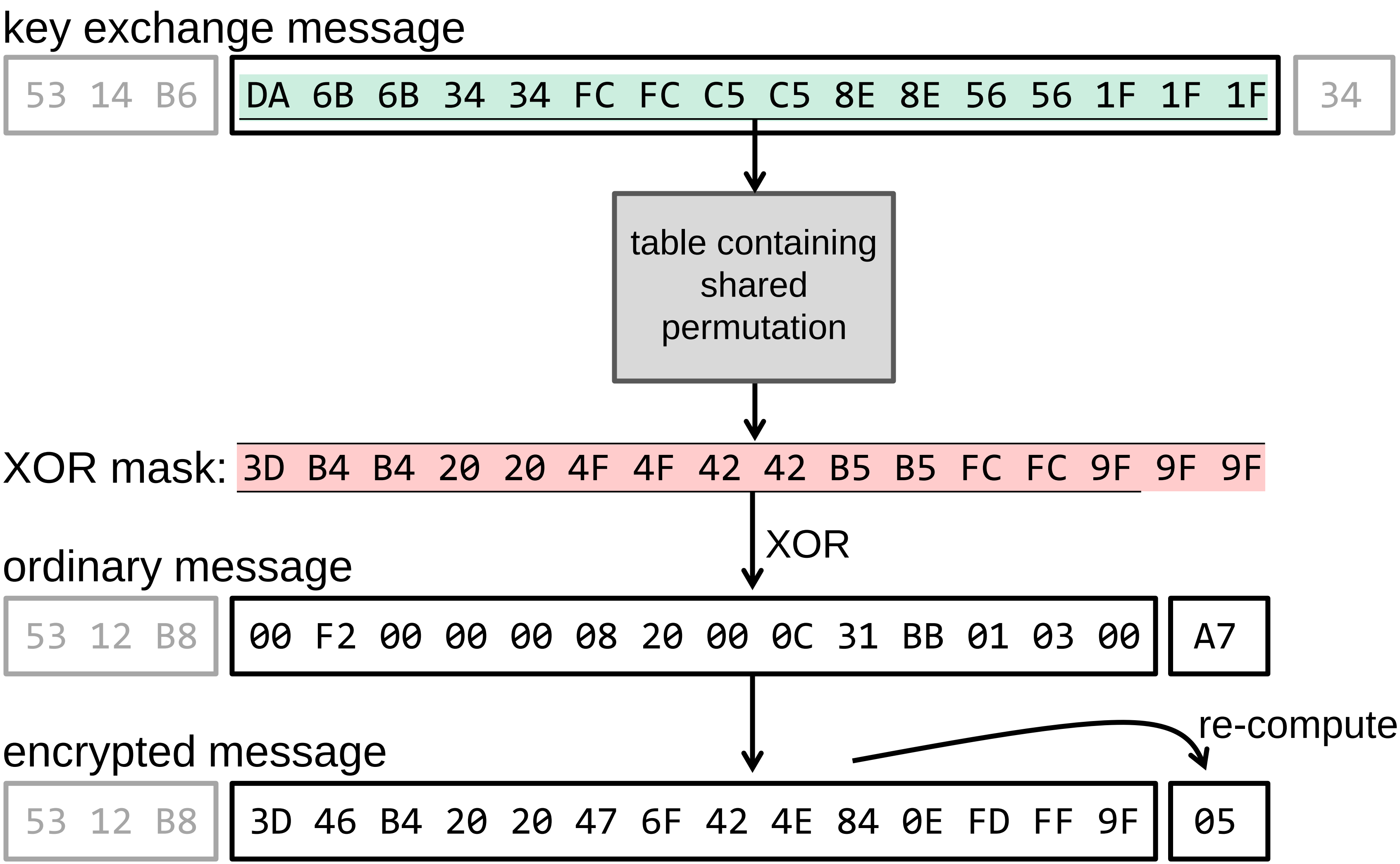
- key exchange message



Protocol reverse engineering

- encryption mechanism
  - the application and the cable share a random permutation of all byte values (`0x00 – 0xFF`) arranged in a table
  - the key value received by the cable in the key exchange message is interpreted as a set of indices into this table
  - the values selected by these indices from the table form a XOR mask
  - messages are encrypted by XORing them with this XOR mask
    - only message content is encrypted, headers remain clear
    - after encryption, checksum is re-computed

Encryption illustrated



Encryption illustrated

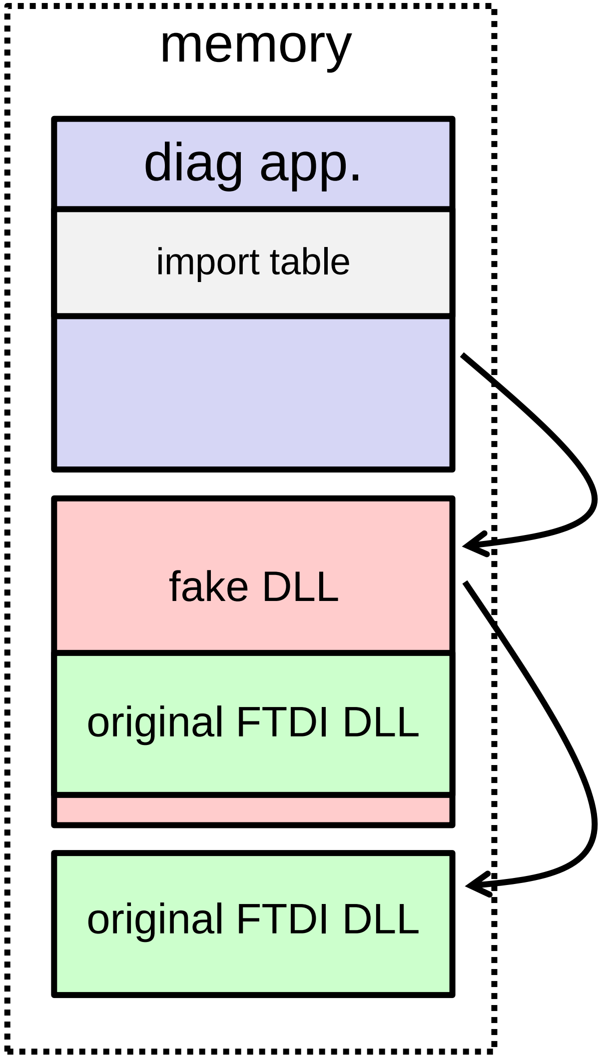
key value: DA 6B 6B 34 34 FC FC C5 C5 8E 8E 56 56 1F 1F 1F

table with permutation

Offset	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x00	00	EA	28	3A	50	1D	B7	58	DB	11	60	0E	E2	C1	41	D8
0x10	80	2C	70	3F	B0	B3	D7	58	BE	C2	42	8D	FB	B2	B6	9F
0x20	B5	EE	05	EC	66	55	9F	EE	2C	E0	AB	05	79	85	56	76
0x30	0C	5B	6B	B3	20	FE	25	DD	E3	35	41	87	29	D2	56	BD
0x40	62	DA	69	44	C9	E6	BC	24	DF	C9	E8	64	18	73	2B	15
0x50	D4	16	03	92	90	87	FC	05	5E	E6	C6	[illegible]	92	81	8A	5F
0x60	BF	F6	7E	CB	E2	98	B8	01	DC	14	40	B4	26	55	68	BD
0x70	C0	A4	60	62	6A	14	05	D9	17	1D	FA	08	9F	87	FA	8F
0x80	B4	89	6C	08	17	33	38	8D	0B	09	DA	7B	0B	F1	B5	76
0x90	B8	4F	A9	AE	15	6E	E7	60	F6	22	04	9F	B6	AB	4D	54
0xA0	29	DD	5B	84	D1	7E	E7	D1	55	F0	DF	44	2E	0F	B8	49
0xB0	A4	5D	07	FB	F9	5D	4B	A2	E4	3C	0D	7A	41	B6	2C	B7
0xC0	06	38	72	C5	79	42	6A	D4	A0	10	76	94	F9	79	1C	3D
0xD0	6C	17	A1	D3	7E	A8	D9	A8	C7	B4	3D	22	A6	71	3E	BE
0xE0	33	E3	D9	55	75	47	6C	9F	D6	B2	C8	F5	D3	F6	87	5B
0xF0	F8	C5	A0	BD	0C	19	38	79	89	D2	BB	1E	4F	A1	2C	73

XOR mask: 3D B4 B4 20 20 4F 4F 42 42 B5 B5 FC FC 9F 9F 9F

Logging messages sent to the car

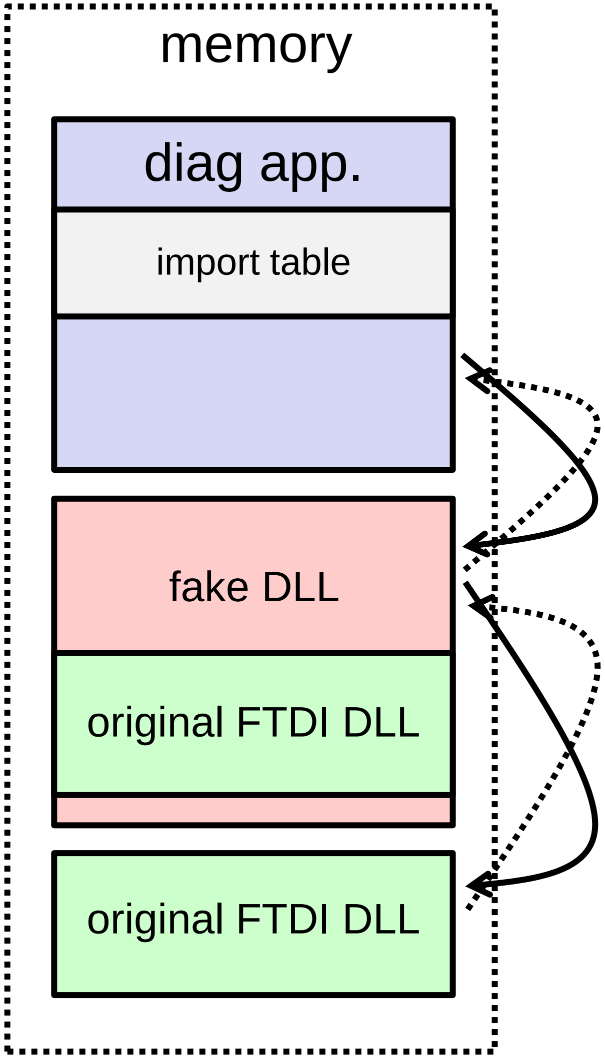


(1) when application calls the FT_write function

(2) we save message content

(3) and pass on the control to the original FTDI DLL

Logging messages received from the car



(4) and return to the application

(1) when application calls the FT_read function

(3) when the call returns, we save the content of the response

(2) we pass on the control to the original FTDI DLL

Logging entire sessions

- before beginning any kind of diagnostic operation the cable needs to be initialized (or ”tested”)
- during initialization, the software examines capabilities of the cable (speed, limits, license) and the car (if cable is connected to a car)
  - initialization results are stored in temporary files (d1.bin, d2.bin, d3.bin, ...)
- usually, these tests are also run before any larger operation-block (e.g., before entering Airbag Control Module)
- the diagnostic software also checks license data stored in the cable
  - the cable needs to be connected to the CAN bus
  - one can bypass this by connecting DC 12V to the OBD2 connector pin-16 (+) and pin-4 (-, ground)

Logging entire sessions

- logs are simple binary files that contain messages sent between the application and the cable
- example:

request size

response size

request

response

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	05	00	00	00	07	00	00	00	A5	53	04	02	55	4D	07	02
00000010	01	54	44	59	0D	00	00	00	0C	00	00	00	53	0D	09	01
00000020	CC	10	BA	8C	16	DF	A8	39	E4	4D	0C	09	28	87	CD	6D
00000030	A9	CC	34	6C	7A	04	00	00	00	14	00	00	00	53	04	04
00000040	53	4D	14	04	52	4F	53	53	54	45	43	48	00	00	00	A8
00000050	B0	E7	92	24	13	04	00	00	00	06	00	00	00	53	04	82
00000060	D5	4D	06	82	00	00	C9	04	00	00	00	05	00	00	00	53
00000070	04	0D	5A	4D	05	0D	02	47	04	00	00	00	04	00	00	00
00000080	53	04	B0	E7	4D	04	FE	B7	04	00	00	00	04	00	00	00
00000090	53	04	B1	E6	4D	04	FE	B7	07	00	00	00	04	00	00	00
000000A0	53	07	B2	00	B8	05	5B	4D	04	FE	B7	09	00	00	00	04
000000B0	00	00	00	53	09	B3	00	FF	E0	FF	00	09	4D	04	FE	B7
000000C0	09	00	00	00	04	00	00	00	53	09	B3	01	FF	E0	00	00

Logging entire sessions

- we made logs of:
  - port test
    - Port Status: OK, Interface: Found!, K1:OK, K2:OK, CAN:OK
    - operation succeeded
  - auto-scan (full scan)
    - Session Init, Scan of all controllers (ECUs), Session Close
    - operation succeeded
  - Airbag Control Module
    - Session Init, Enable/Disable front passenger airbag, Session Close
    - operation succeeded
  - ABS Brakes Control Module
    - Session Init, Enable/Disable ABS booster, Session Close
    - operation failed (maybe wasn't supported)

Replaying sessions

- replay files are similar to logs, but they contain only messages to be sent to the cable (FT_Write)
- example:

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	A5	53	04	02	55	53	0D	09	01	0C	9E	7F	9E	93	5C	25
00000010	ED	22	53	04	04	53	53	04	82	D5	53	04	0D	5A	53	04
00000020	B0	E7	53	04	B1	E6	53	07	B2	00	B8	05	5B	53	09	B3
00000030	00	FF	E0	FF	00	09	53	09	B3	01	FF	E0	00	00	F7	53
00000040	09	B4	00	3F	E0	00	00	31	53	09	B4	01	43	E0	00	00
00000050	4C	53	09	B4	02	60	00	00	00	8C	53	09	B4	03	00	00
00000060	00	00	ED	53	09	B4	04	00	00	00	00	EA	53	09	B4	05
00000070	EF	40	00	00	44	53	06	B5	00	64	84	53	06	B5	01	60
00000080	81	53	14	B6	FA	C3	54	1D	E6	AE	3F	08	D1	99	62	2B
00000090	BC	85	4D	16	9D	53	11	B8	BB	85	90	B2	6C	BF	A2	1B
000000A0	17	32	7E	06	40	91	53	0B	B8	BB	A0	50	B2	6C	B9	15
000000B0	D9	53	04	A0	F7	53	04	16	41	53	06	0B	00	00	5E	53
000000C0	06	0B	01	00	5F	53	06	0B	02	00	5C	53	06	0B	03	00

Replaying sessions

- our replay tool is a separate process that uses the original FTDI DLL for writing to the cable
- cable initialization requires initializing the FTDI device correctly (i.e., setting baud rate, timeouts, …)
  - following functions are called with appropriate parameters:
    - FT_SetLatencyTimer(device, 2);
    - FT_SetTimeouts(device, 1, 100);
    - FT_SetDataCharacteristics(device, 8, 0, 0);
    - FT_SetBaudRate(device, 115200);
- then we can write with function FT_Write
  - after writing out a message we wait around 300 ms before writing the next message
- don't forget to write `0xA5` (one-byte message) at the beginning
  - sort of session initialization

Switching off the airbag

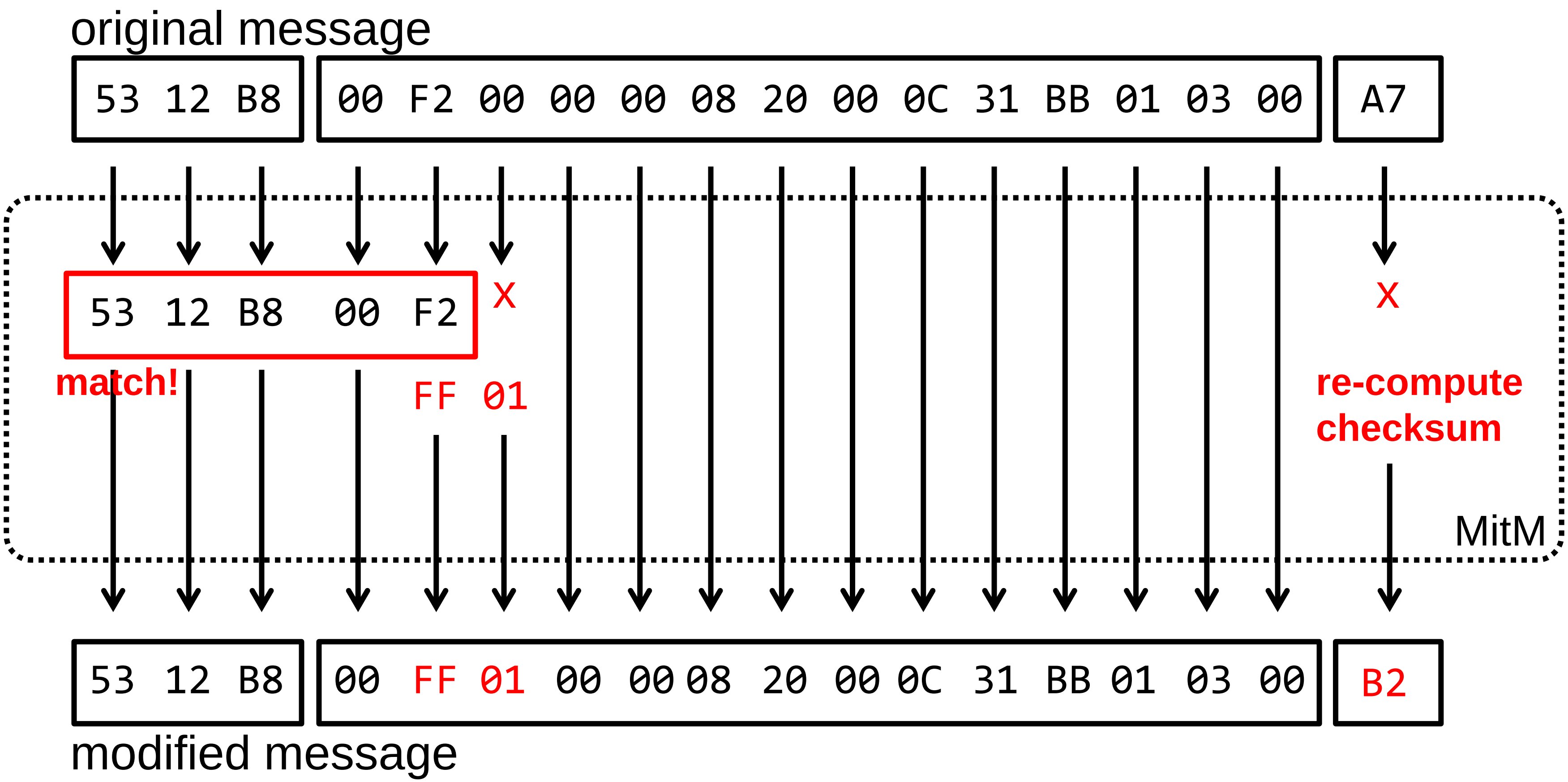
- we could easily replay a previously recorded messages that switched the passenger airbag off
  - easy means that there's no need to wait for any response, change encoding, ...
  - we just sent a previously recorded messages to the Airbag Control Module
    - Session Init, Disable front passenger airbag, Session Close
    - operation succeeded
- as our replay tool is a separate application, the replay message is invisible to the diagnostic application!

Modification of messages on-the-fly

- application sends messages to the cable in a byte-after-byte manner
- on-the-fly modification of messages requires
  - matching some pre-specified sample in the byte sequence
  - and replacing follow-up bytes with a pre-specified pattern
- easily done by the Man-in-the-Middle capability we have in our fake FTDI DLL

On-the-fly modification illustrated

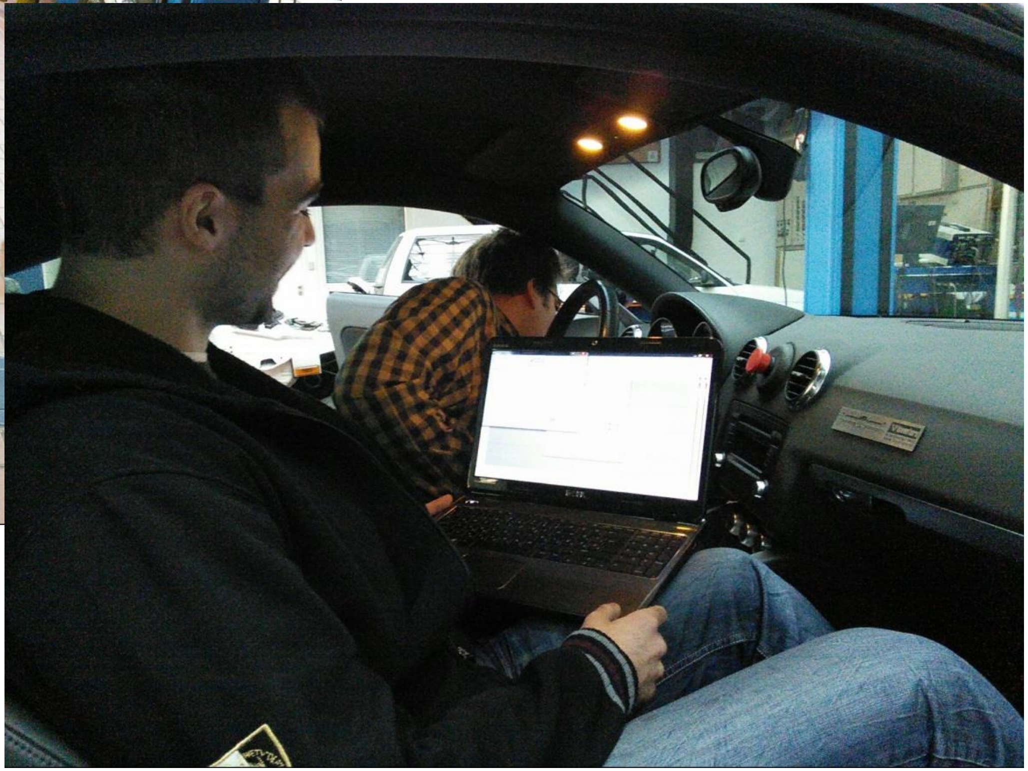
sample to match: `53 12 B8 00 F2`
replacement pattern: `FF 01`



Experiments

MŰEGYETEM

GÉPJÁRMŰVEK ÉS JÁRMŰGYÁRTÁS TANSZÉK



experiments were carried out during spring 2015

Example – Logging a full scan

- diagnostic application exports scan result as a simple log file

Log-TRUZZZ8J471000320-18530km-11514mi.txt

```
Wednesday,27,May,2015,14:44:25:37412
[redacted]
Data version: 20111111



--------------------------------------------------------------------------------
--------------------------------------------------------------------------------


Chassis Type: 1K0
Scan: 01 02 03 08 09 15 16 17 19 25 42 44 46 47 52 56

VIN: TRUZZZ8J471000320   Mileage: 18530km/11514miles

00-Steering Angle Sensor -- Status: OK 0000
01-Engine -- Status: Malfunction 0010
02-Auto Trans -- Status: Malfunction 0010
03-ABS Brakes -- Status: Malfunction 0010
08-Auto HVAC -- Status: Malfunction 0010
09-Cent. Elect. -- Status: Malfunction 0010
15-Airbags -- Status: Malfunction 0010
16-Steering wheel -- Status: Malfunction 0010
17-Instruments -- Status: Malfunction 0010
19-CAN Gateway -- Status: Malfunction 0010
25-Immobilizer -- Status: Malfunction 0010
```

Example – Logging a full scan

Log-TRUZZZ8J471000320-18530km-11514mi.txt

```
Chassis Type: 1K0
Scan: 01 02 03 08 09 15 16 17 19 25 42 44 46 47 52 56

VIN: TRUZZZ8J471000320   Mileage: 18530km/11514miles
```

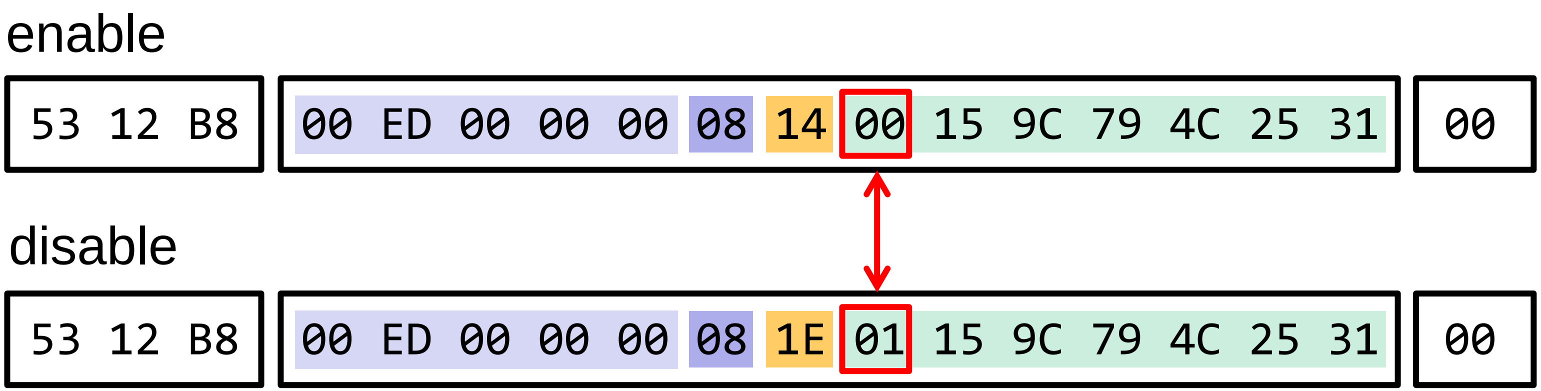
```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

000005F0  8A FF 4A FF 00 0F 00 00 00 99 00 00 00 53 0F B8   Š˙J˙.....™...S.¸
00000600  00 65 C0 00 00 05 10 00 02 1A 9B 00 4D 04 FE 00   .eŔ.......›.M.ţ.
00000610  4D 0B B7 01 60 00 CE E7 01 B1 00 4D 12 B7 01 60   M.·.`.Îç.±.M.·.`
00000620  00 4C 78 08 20 00 30 5A 9B 31 4B 30 00 4D 12 B7   .Lx. .0Z›1K0.M.·
00000630  01 60 00 CE E7 08 21 39 30 37 35 33 30 4A 00 4D   .`.Îç.!907530J.M
00000640  12 B7 01 60 00 4C 78 08 22 20 20 30 31 36 30 10   .·.`.Lx."  0160.
```

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00001840  00 0F 00 00 00 45 00 00 00 53 0F B8 00 E8 00 00   .....E...S.¸.č..
00001850  00 05 15 00 02 1A 90 00 4D 04 FE 00 4D 0B B7 01   ........M.ţ.M.·.
00001860  60 00 4C 78 01 B6 00 4D 12 B7 01 60 00 CE E7 08   `.Lx.¶.M.·.`.Îç.
00001870  21 00 13 5A 90 54 52 55 00 4D 12 B7 01 60 00 4C   !..Z.TRU.M.·.`.L
00001880  78 08 22 5A 5A 5A 38 4A 34 37 00 4D 12 B7 01 60   x."ZZZ8J47.M.·.`
00001890  00 CE E7 08 13 31 30 30 30 33 32 30 00 0B 00 00   .Îç..1000320....
```

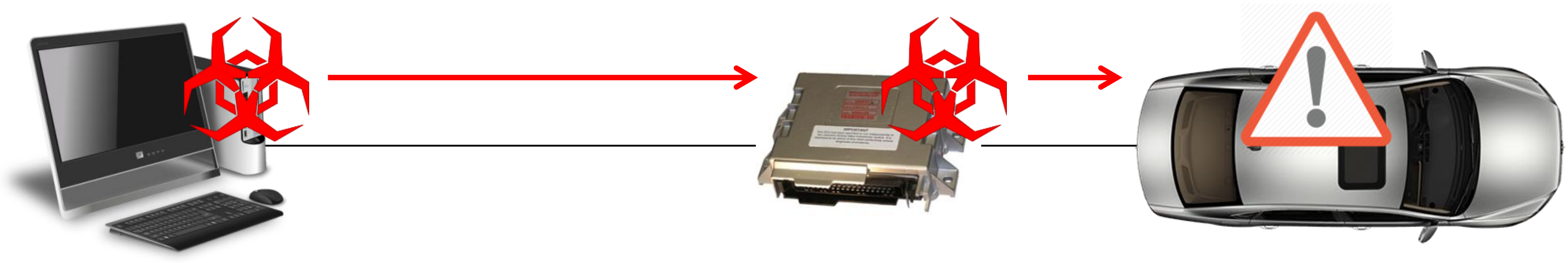
Replaying airbag enable/disable messages



- we used our separate replay tool to replay back previously recorded passenger airbag enable and disable messages
- after every replay we deleted all temporary config files and started the diagnostic application to check the results
- all replays were successful

Conclusions

- cyber attacks on modern vehicles is a plausible threat
- lot of research on remote attacks, but ...
- a Stuxnet-style attack may have a higher risk



- we demonstrated *in practice* that such an attack is easy to implement against cars by minimal modification of a diagnostic application (could be done by a malware)
- our proof-of-concept implementation allows for Man-in-the-Middle attacks between the application and the car
- for illustration purposes, we switched off the passenger airbag stealthily with a replay attack

Outlook

the Internet-of-Things:
billions of network enabled
embedded devices

Outlook

the Internet-of-Things:
billions of network enabled
embedded devices

what can we *really* do about this?

Laboratory of Cryptography and System Security (CrySyS Lab)
Budapest University of Technology and Economics
www.crysys.hu

contact:
Levente Buttyán, PhD
Associate Professor, Head of the Lab
buttyan@crysys.hu