Literature survey on memory level attacks in Automotive Embedded Systems

Gaurav Kalamkar, Ajey Gotkhindikar, A. R. Suryawanshi

[1]*M.E. Student Pimpri Chinchwad College of Engineering Akurdi, Maharashtra*
[2]*Subject Matter Expert,Automotive Cybersecurity,KPIT Technologies Ltd.Pune, Maharashtra*
[3]*Associate Prof. E&TC Department Pimpri Chinchwad College of Engineering, Akurdi, Maharashtra*

Abstract

Embedded systems are driving force for the automotive systems as more and more functionality and features are getting added in next generation modern cars. There are more than 70 ECUs with more than 100 million lines of code. ECUs controls most of the car's features and functions. With the development of Electric Cars and Autonomous vehicles, these ECUs are going to play an important role. Network connectivity features for instance V2V for data sharing and V2I for remote updates are integrated with the modern cars. Memories such as FLASH and RAM are also called low level memory because of their functionality at hardware level. There are many ways to read those sectors. Those attacks are broadly classified as physical, logical and side channel attacks. This paper has reviewed features and characteristics of the automotive embedded system, their vulnerabilities, threats, and general attacking strategies on memories.

Keywords—*ECU, low-level memory attack, buffer overflow, debuggers, DMA,*

I. INTRODUCTION

Embedded system is becoming an integral part of our day to day life. These systems are the driving force for technological development in many domains such as automotive, healthcare and industrial control [1]. As devices are getting more and more computationally efficiency and network enabled, security is becoming critical for all smart or intelligent systems built upon these embedded systems. Comparing to conventional IT systems, security of embedded systems are no better due to poor security design, implementation and resource constraint [2].

Today's modern vehicles are no longer just mechanical; most of the car's functionality is controlled by software [3]. As automotive embedded Systems are being network enabled, allowing for remote updates and data sharing, software corruption has become a major concern [4]. Software corruption can simplistically be considered as unauthorized instructions that are executed within the system. This can occur through behaviorally modified instruction code introduced via new software installation, updates, or application input data (such as buffer overflows). Low-level memory such as RAM, FLASH, and CACHEs play an important role in automobile embedded systems like gateways, infotainment, telematics units and other ECUs. If an attackers manages to change the firmware of the system then they can damage the entire system.

In this survey, we review attacks of the existing threats and vulnerabilities in low-level memories of embedded systems based on available information [5, 6]. There are various ways to attack embedded systems which are broadly classified under physical, logical or side channel based attacks.

Physical attacks mainly based on Reverse engineering techniques of embedded system and Eavesdropping. Logical attacks, on the other hand, can be either software based or cryptographic and Side channel attacks can be Fault injection attacks, Power analysis attack and Timing analysis attack [1, 3, 4 and 6].

II. FEATURES, VULNEARABILITIES AND THREATS OF THE AUTOMOTIVE EMBEDDED SYSTEM

A. Features:

Many advance features and vulnerabilities of automotive embedded systems has directly affected the security. We have discussed some of features and their vulnerabilities in systems. [4, 6]

- ***Limited processing power*** implies that an automotive embedded system will not be able to run application to defenses against external attacks. Hardware configuration of ECU's are very specific to their application. Internal

RAM and FLASH memory is almost in the range of few MBs. Because of this applications such as virus scanner, intrusion detection systems are very hard to deploy in the ECU.

- ***Limited power supply*** is one of the key constraints in automotive embedded systems. As newer and upgraded features are introduced in modern cars to facilitate more comfort. As power supply is limited to individual devices increased power consumption reduces system lifetime or increases maintenance frequency. Therefore very limited power supply can be dedicated for system security.
- ***Physical exposure*** is typical characteristic of systems that is unavoidable as there will be times when vehicles are deployed outside the control of the owner or operator e.g., public location, parking etc. Thus, embedded systems are inherently vulnerable to attacks that exploit physical proximity of the attacker. [7]
- ***Network connectivity*** via wireless or wired access is increasing in modern cars. Wi-Fi, 3G and 4G systems are getting deployed such systems for various applications Such as remote control, data collection, updates.

B. Threats

- ***Energy drainage (exhaustion attack):*** Limited battery power in automotive embedded systems makes them vulnerable to attacks that drain this resource. With the modern electric vehicles which runs on batteries, this leads to problem of recharging batteries again and again. Energy drainage can be achieved by increasing the computational load, reducing sleep cycles, or increasing the use of sensors or other peripherals.
- ***Physical intrusion (tampering):*** The proximity of automotive embedded systems creates vulnerabilities to attacks where physical access to the system is necessary. Examples are power analysis attacks or eavesdropping on the system bus which are basically reverse engineering techniques
- ***Network intrusion (malware attack):*** Networked embedded systems are vulnerable to the same type of remote exploits that are common for workstations and servers. Intruder can get access to system via network related services such as Bluetooth and Wi-Fi. Remote access using secure shell is the example of that.
- ***Introduction of forged information (authenticity)****:* Embedded systems are vulnerable to malicious introduction of incorrect data (either via the system's sensors or by direct write to memory). Someone with the replay attack of Bluetooth pairing key can get access to vehicle.
- ***Confusing/damaging of sensor or other peripherals****:* Similar to the introduction of malicious data, embedded systems are vulnerable to attacks that cause incorrect operation of sensors or peripherals. An examples is tampering with the calibration of a sensor. By identifying proper CAN ID's to particular sensors we can feed wrong sensors information on CAN bus which may lead to false outcome of the system which needs those data for calculation.
- ***Reprogramming of systems for other purposes (stealing):*** While many embedded systems are general-purpose processing systems, automotive systems are intended to be used for a particular use. These systems are vulnerable to unauthorized reprogramming for other uses. After getting physical access of the system this attack is very common, attacker may reprogram it to steals sensitive information in the vehicle such as CAN ID for particular data etc. or can add additional feature to help him getting access from remote location and controlling the vehicle. [7]

C. Vulnerabilities

As per Common Vulnerabilities and Exposure records, researchers have found out following vulnerabilities in embedded systems. [4, 1]

- ***Programming errors***: Many of the vulnerabilities in the selected CVE records stem from programming errors, which may lead to control flow attacks (e.g., buffer overflow problems and memory management problems such as using pointers referring to memory locations that have been freed).
- ***Network based vulnerability***: In automotive embedded systems web based vulnerability is in terms of over the updates. As these updates are coming from remote location, it is very important to make sure they are coming from valid source and the updates are not tampered.
- ***Weak access control or authentication***: Many devices such as Bluetooth uses default or weak passwords, and some devices have hard-coded passwords that provide backdoor access to those who knows the password. Few devices will have no security at all or will have open access. Such vulnerabilities make it possible for attackers to bypass access control mechanisms with minimal effort.
- ***Improper use of cryptography***: Some devices use cryptographic mechanisms for authentication purposes or for preserving the confidentiality of some sensitive information. Often, cryptographic mechanisms are not used appropriately, which leads to fatal security failures. Examples include the use of weak random number generators for generating cryptographic keys.

D. Requirement for the attacker

Following conditions are required for attacker to attack any automotive embedded systems [4].

- ***Internet facing device***: Internet facing devices such as Wi-Fi, 3G/4G has many vulnerabilities as per CVE records. Most of them are exploited by the remote attacker if the devices is connected to internet. The attacker does not necessarily need to have access privileges; the only requirement is that the attacker can potentially discover the device and send messages to it via the network.
- ***Local or remote access to the device***: This precondition requires the attacker to have some privileges that allow for logical access to the services or functions provided by the device. This logical access can be restricted to local access or it can be a remote access capability (e.g., via the Internet, via CAN). Often, the privileges required by the access are normal user privileges, and not administrator privileges.
- ***Direct physical access to the device***: Direct physical access requires the attacker to access the device physically. Attacker can get access to programing session of the device using onboard diagnostics port.
- ***Physically proximity of the attacker***: In some cases, the attacker does not need physical access. It is sufficient that the attacker can be in the physical proximity of the device. For instance, attacks on wireless devices such as Bluetooth and Wi-Fi hotspots may only require to be within the radio range of the target device.

III. LOW LEVEL MEMORY IN AUTOMOTIVE EMBEDDED SYSTEMS

An electronic control unit is a part of the vehicle which is used to control and regulate various functions such as processing the information provided by sensors and actuators.

CAN, MOST, LIN busses are deployed inside car for communication network. ECU is always connected to a network and other components through busses and gateways. Sensors convert physical quantitates such as speed, temperature, pressure to electrical signals which can be processed by ECUs. On the other hand actuators are devices which convert an electronic signal to motion [3, 8].

There are two types of memory in ECU hardware i.e. FLASH and RAM. The application is stored in FLASH memory of ECU. This application defines the behavior of that particular ECU. FLASH is reprogrammable and nonvolatile. On the other hand, RAM is a temporary storage that provides better performance. When ECU is powered on contents of FLASH memory is copied to RAM for execution.

When a system is enabled, the ECU first executes the bootloader. There are two types of bootloader is ECU, a primary bootloader (PBL) and a secondary bootloader (SBL).

PBL loads the application software stored in the flash memory. Since PBL is of very small memory size (~16k). This is programmed during manufacturing and cannot be modified for update after the unit is produced. So it is not suitable to be used for programming or updating data in the flash memory. The SBL is instead used for these purposes. The SBL is downloaded by the primary bootloader into RAM and is then activated [3, 8]. SBL can be modified with updates.

There are two execution modes of ECU. One is normal mode in which PBL will load SBL into RAM. Then after initialization, SBL loads the application. Another is programming mode where SBL will be loaded to RAM first and then FLASH can be reprogrammed.

IV. ATTACKS ON LOW-LEVEL MEMORY:

ECU stores the important data required for the functioning of automobile components. Usually memory capacity of ECU is very low and if a software is developed in an unsecure manner then system becomes vulnerable to software-based attacks such as buffer overflow.

These attacks might cause corruption of data stored in the FLASH memory. The memory of an ECU is also subjected to offline attacks. We can disassemble the ECU and take away the memory chip using an anti-static mat and a soldering iron. By using a FLASH reader we can get the complete software image. Now to make sense of the software image, we need to use a disassembler. Hence contents of the memory can be easily read out and it is not secure anymore [3].

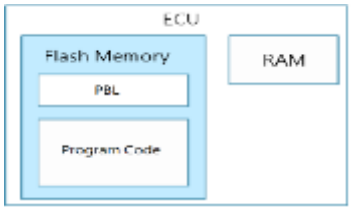

Fig. 1- ECU hardware memory map

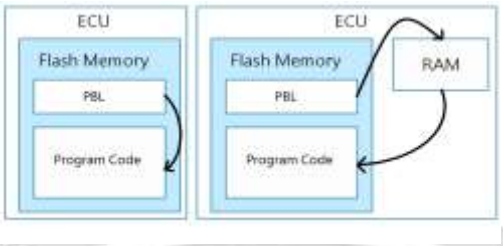

Fig. 2- ECU's execution modes

V. VARIOUS TYPES OF ATTACKS ON LOW LEVEL MEMORY

*A. **Physical:***

In this type of attack, the attacker has complete physical access to the system. For someone with manufacturing knowledge attempts to maliciously compromise the system then attack surface increases significantly. Eavesdropping is the intercepting of conversations by unauthorized receiver which is performed when sensitive information is passed via electronic media, such as e-mail or instant messaging. Micro-probing, reverse engineering and eavesdropping are the examples of physical attacks.

1) Reverse engineering:

Reverse engineering techniques has been developed to perform the opposite of a typical design flow used to build Integrated Circuits (IC). IC reverse engineering can be used for validation, debugging, patent infringement/malicious circuit modification/backdoors detection and failure analysis.

Automotive embedded systems rely heavily on non-volatile memories such as FLASH to store code and data [10]. There is a constantly growing demand for the confidentiality of the information stored in these devices for Intellectual Property (IP) protection and sensitive data including signatures and cryptographic keys. Kommerling and Kuhn [11] have shown that Mask ROM contents can be revealed using a microscope after sample preparation. Since then Mask ROMs have not been considered to be secure unless encrypted.

Several publications refers to Scanning Probe Microscopy (SPM) techniques being used to highlight differences between `0' and `1' in Flash EEPROM. For instance, the use of a current applied on a conductive tip allows us to see some interaction whenever electron charges are present within memory cells.

Voltage Contrast imaging is one of the first use of Scanning Electron Microscope. This technique was based on a setup where no external bias is applied to the sample while setup parameters permit to obtain various information on the sample.

In Scanning electron microscope technique as suggested by Franck in [12], an automated acquisition routine can be launched. Scanning area and magnification need to be first defined. It permits to collect a large set of images over a full memory without the presence of an operator. At last, image processing enables us to align all acquisitions together, to enhance the image contrast and to extract `0s' and `1s' in an automated way. It also allows us correlating the extracted data with the one load into the samples.

2) Micro-probing:

Functional analysis involves system monitoring during functional operation. Probes can be connected on the system wherever needed. Micro-probing is used to monitor on-chip signals [13] **.** Test cases are developed, and stimulus created for operating the system in its functional modes. Signal generators, logic analyzers, and oscilloscopes are used to simulate/run the system and collect the results. In automotive systems, we will connect system to CAN port and power on the system and observe the CAN packets. Also we can send CAN packets to observe the behaviors of the particular ECU.

3) Eavesdropping

In automotive devices data between two ECUs transferred via CAN channel. Each ECU transmits the message in CAN packets which is received by every ECU. When two entities send or receive sensitive information using public networks or communications channels accessible to potential attackers, they should ideally provide security functions such as *data confidentiality*, *data integrity*, and *peer authentication*. [14]. As CAN uses broadcast type communication so it is not possible for end to end communication. Therefore these security functions are hard to implement.

In automotive systems, if attackers manages to identify CAN bus then they can eavesdrop on the bus to read and identify CAN IDs. With the proper datasets of CAN IDs attacker then can steals data such as cryptographic keys or can perform attacks such as DOS or replay attack.

4) Using Debuggers:

Debuggers can be used to get access to the embedded system. Serial ports such as SPI, JTAG and UART can actually be used to read out the memory. Jeong Wook [15] explained how to get access to NAND FLASH memory in the system and read it using FTDI FT2232H. For that we have to first identify the port. Then we can connect the system to debug software and get the access to memory.

Kurt Rosenfeld [16] has enlisted possible attacks on JTAG such as sniffing secret data, obtaining test vectors and modifying authentication part. They have given all the details required to get access to memory. UrJTAG and OpenOCD can read the memory from the FLASH. For security reasons many manufactures disables JTAG port after testing.

Neil Jones [17] has explained the procedure to install a rootkit in the firmware using serial ports. In this method, we interrupt initial booting sequence of the system and get access to its bootloader using UART. After getting this access intruder can get access to and send data over CAN port to attack the system.

B. ***Logical:***

Logical attacks are basically software based attacks which are used to basically for gaining the unauthorized access of system or to do security testing. Based on these applications, logical attacks are broadly classified under cryptographic attacks and code injection attacks.

Cryptographic attacks are used to perform security attacks, such as breaking into a system. These are used exploit weakness in cryptographic algorithms. A short list of common crypto and protocol vulnerabilities is given in [14].

Most of those attacks results in tampering with the application. These includes changing instructions with intention of gaining control over a program execution flow. Attacks that are involved in violating software integrity are called code injection attacks.

Increase in the size of codes increases the number of malicious attacks. Some of the attacks include stack-based buffer overflows, heap-based buffer overflows, exploitation of double-free vulnerability, integer errors, and the exploitation of format string vulnerabilities.

1) Code injection attack

A buffer overflow is a vulnerability, a weakness which may allow a threat to exploit the software program. When the contents of a buffer are overflowed, the overflow can overwrite a portion of a processor's memory. The information stored at this memory location could possibly be lost forever. Including list of instructions that tells the program where to go and what to do next. The program will not be able to pick up where it left off or finish its tasks [18].

A buffer overflow attack or exploit is a threat that exploits buffer overflow vulnerabilities. Intruder may carefully craft the script to take advantage of this vulnerability. Script contains instructions which are provided to the program so that it loses its way during the buffer overflow. From there on the program is instructed to do whatever the attacker has intended to cause harm to the system,which can be very dangerous. This not only allows an attacker's dangerous script to be executed, but it also prevents the original program from completing its task.

Buffer overflows may be found locally or remotely [19]. Most attacks usually target a specific application or operating system, which suggests that an understanding of the differences in these is important. Most buffer overflow vulnerabilities are found in C and C++ because of careless use of programming primitives by the developer.

Exploits can lead to very damaging effects no matter whom or what the target may be. Buffer overflow exploits can cause many effects, including an application crashing, a computer crashing, a denial of service, and triggering of running code [18]. Many attackers use exploits to damage software or to gain or expose private information [20].

a) Stack based buffer overflow

A stack is a block of memory which contains temporary data. Stack pointer (SP) register points to the top of the stack. The bottom of the stack is at a fixed address. Its size is dynamically adjusted by the kernel at run time. Stack frame

pushed on the stack while calling subroutine program and popped while returning. A stack frame contains the parameters to a function, its local variables, and return address.

In addition to the stack pointer, it is often useful to have a frame pointer (FP) which points to a fixed location within a frame. Many compilers use a second register, FP, for referencing both local variables and parameters because their distances from FP do not change with *PUSH*es and *POP*s. The first thing a procedure must do when called is saving the previous FP. Then it copies SP into FP to create the new FP, and advances SP to reserve space for the local variables [9, 21, 22]

As a C or C++ programmer it is natural to assume that, if a function is invoked at a particular call site and runs to completion without throwing an exception, then that function will return to the instruction immediately following that same, particular call site.

Unfortunately, this may not be the case in the presence of software bugs. For example, if the function contains a local array, or buffer, and writes into that buffer are not correctly guarded, then the return address on the stack may be overwritten and corrupted. In particular, this may happen if data of whose is larger than buffer size is copied to the buffer, which causes buffer overflow.

Furthermore, if an attacker has control over the data used by the function, then the attacker may be able to trigger such corruption and change the function return address to an arbitrary value. In this case, when the function returns, the attacker can direct execution to the code of their choice and gain full control over the behavior of the system. This attack is called "return address clobbering".

Indeed, until about a decade ago, this attack was seen by many as the only significant low-level attack on software compiled from C and C++, and "stack-based buffer overflow" was widely considered a synonym for such attacks. More recently, this attack has not been as prominent, in part because other methods of attack have been widely publicized, but also in part because the underlying vulnerabilities that enable return-address clobbering are slowly being eliminated.

b) Heap based buffer overflow

The heap is memory region where we can allocate memory dynamically with help of *malloc* and *free* at runtime. This region is maintained as linked list. Each memory block carries its own data, such as its size and pointers to other chunks. *malloc()* provides the lowest memory address of the memory region. Variables holding memory addresses are of pointer type, hence the address returned by *malloc*() is for future use stored in such a pointer. This can be very useful but on the other hand has been a constant source of various problems with C programs, in particular if pointers are not properly initialized.

Software written in C and C++ often combines data buffers and pointers into the same data structures, or objects, with programmers making a natural assumption that the data values do not affect the pointer values. In particular, the pointers may be corrupted as a result of an overflow of the data buffer-regardless whether the data structures or objects reside on stack, or in heap memory. [21, 23, 18]

c) Shell code injection

Shell-Code is usually written in assembly and assembled into OPCODES. Shell-Code got a name because of its initial purpose was to spawn a shell. These days shell-code is much more than that, and what it can do is only limited by a hacker's creativity. Shell code cannot contain any wrong OPCODE which will terminate shell code execution. For example, whenever a payload is interpreted as a string, the *NUL* byte is a 'bad' character, as this is a string terminator, and will stop the Shell-code mid-write (there is only one place this shell-code is allowed a *NUL* byte, at the end). Shell-code is usually subject to a size restriction, based on the size of the buffer available.

When high level languages code are compiled, they are generally compiled into binaries. Binaries, being based on a number system, can be translated into another number sequence. The hexadecimal number system is used in OPCODE assembly instruction. An assembly instruction *MOV* is like an instruction label, which represents an OPCODE, for example, 0xeb represents a *JMP* assembly instruction, frequently found in shell-code. It is necessary to use these OPCODEs as shell-code because it has to sit in the middle of already compiled program, so that the CPU will execute it. It is common practice to write shell-code in assembly and use an assembler such as NASM http://www.nasm.us/ which converts assembly instructions to OPCODEs and does some low level memory management, such as creation of stack frames. [24]

d) Return to libc attack

When any function is called, the stack contains the return address first, and then the parameters as expected by the function. As we have discussed in previous section, the ultimate target of buffer overflow attack is to run shell code to obtain a shell. Typically such shell code relies on calling functions like system () or execve () on UNIX systems or WinExec () on Windows systems. These functions basically take as parameter a program name and/or path and run it on behalf of the calling binary.

So an alternative to the direct stack smashing approach is to just fake the call to the desired function directly. Execution of such called function has valid code address and is a legal operation and so it can't be prevented by a non-executable stack.
To accomplish this, it sufficient to overwrite the saved return address with the address of the targeted library function, insert an arbitrary dummy value for the return address of the function call just afterwards and adding the parameters for the library function [21].

e) Return oriented programming

In shell code injection attack, attacker drives the execution flow towards the malicious code. To prevent this, the operating system may implement a countermeasure called data execution prevention (DEP) and automatic space layout randomization (ASLR). DEP consists of flag to the memory regions. If it is set to 0, this flag marks the corresponding memory address range as non-executable. As stack memory is never meant to execute code, the OS can set the flag corresponding to the stack area to 0 right during the memory allocation. Due to this, even if attackers succeed at injecting code into the stack, they won't be able to execute the payload because the OS would refuse fetch the corresponding instructions from the memory area that holds them.
Return-oriented programming allows an attacker to exploit memory errors in a program without injecting new code into the program's address space. In a return-oriented attack, the attacker arranges for short sequences of instructions in the target program to be executed, one sequence after another.
ROP takes its name from the *RETN* assembly instruction, it is responsible for driving the process control out of the attacked process flow. *RETN* uses the saved instruction pointer from the stack and upgrades the registers esp and eip. More precisely, it adds 4 bytes to esp and puts the popped value back to eip. If attackers can exploit a buffer overflow on the target system, they can place a value of choice in the memory cell at ebp + 4 for 32 bit systems, which means that, as soon as the execution flow encounters *RETN*, the instruction pointer is loaded and the execution continues from the pointed memory cell. For this property, each instruction sequence ending in the instruction *RETN* is called a "*gadget"*.
The first gadget in the chain is invoked by placing its address on the stack (ebp + 4) with a buffer-overflow attack; after its execution, the control flow gets back to the attackers because every gadget ends with an *RETN*. By modifying the next return address on the stack, attackers can redirect the control flow to another gadget until they want to end the code execution. Because gadget chaining is essentially a sequence of jumps to pieces of existing code, being able to exploit it for malicious purposes might seem impossible. However, the ROP technique uses code misalignment to forge new instructions from code already loaded in memory.
The most important result of "gadget-based programming" for a malicious attacker is that it becomes quite easy to bypass DEP. A chain of gadgets resides in executable memory areas, as opposed to classic injected code that resided in the stack, so DEP countermeasures can't deny its execution. Although attackers could in principle write any program by assembling complex return chains, ROP has so far been used to make existing buffer-overflow attacks work even in the presence of OS countermeasures. This goal is easily reached by calling system APIs that disable DEP, thus making code on the stack executable again. [25]
Once DEP policy is set off then attackers can execute their own codes. ROP is possible because Linux and android system has large set of libraries which are loaded into memory.
The different types of ROP are:

- Jump Oriented Programming- Uses the *jump* instruction instead of the *ret*
- String Oriented Programming-SOP uses a format string bug to get the control flow.
- Blind Return Oriented Programming-BROP deals with the ROP and "timing attack"
- Signal Return Oriented Programming-Uses the *SIGRETURN* Linux signal to load values from the stack to the registers

2) Cryptographic attack

a) Brute forcing

When exploiting a vulnerability such as a buffer overflow or a format string vulnerability it often fails because the last problem was not taken care of: to get all combinations properly. For simple vulnerabilities, we can reliably guess the correct combination, or just brute force it, by trying them one after another. But as soon as we need multiple offsets this problem increases enormously, it turns out to be impossible to brute force.
With format strings, this problem only appears if we are exploiting a daemon or any program which will offer you only one try. As soon as you have multiple tries or you can see the response of the format string it is possible, although not trivial to find all the necessary offsets.
This is possible because we already have limited control over the target process before we completely take it over: our format string already directs the remote process what to do, enabling us to peek in the memory or test certain behaviors. [26]

b) Dictionary attack

A dictionary attack is a more sophisticated form of the brute force password attack, where thousands, if not millions, of randomly generated passwords are attempted in order to break password security. In the dictionary attack, the attacker starts with lists of probable passwords, removing some of the random elements of the brute force attacks.

*C. **Side channel attacks:***

It is not possible for attacker to attack in security mechanism using cryptographic algorithms every time. Security attacks on system targets weaknesses in the implementation and deployment of the cryptographic algorithms. These weaknesses can allow attackers to completely bypass, or significantly weaken, the theoretical strength of security solutions.

Side channel attacks are known for the ease with which they can be implemented, and for their effectiveness in stealing secret information from the device without leaving a trace [27]. Attackers observe side channels such as power usage, processing time and electromagnetic (EM) emissions while the chip is processing secure transactions.

The attacker feeds different input values into the system, while recording the side channels during the execution of a cryptographic algorithm (e.g., encryption using a secret key). These recorded external calculations are then correlated with the internal computations. Side channel attacks can be performed successfully at either the sender or the receiver to identify the secret keys used for encryption and/or decryption.

Power dissipation/consumption of a chip is the most exploited property to determine secret keys using side channel attacks. Devices like Smart Cards, PDAs and Mobile Phones have microprocessor chips built inside, performing secure transactions using secret keys. In automotive systems such as wireless door locking/unlocking systems uses Bluetooth devices, attacker can sniff this transaction and can use it for replay attacks.

1) Fault injection attacks

Fault injection attacks are one kind of side channel attacks that affect embedded systems by injecting a fault into the system. There are several kinds of Fault injection attacks that can affect the security of the embedded systems [14]. The kind of side channel attacks that can be induced into embedded systems are dependent on the hardware used and the applications software implemented on such hardware [28]. Hardware security devices depend on more than correct software. If the hardware ever fails to make correct computations, security can be jeopardized.

2) Timing analysis attack

In this attack, attacker makes prediction about secret values and use statistical methods to test the prediction [14]**.** Attacker observes a set of inputs and notes approximate total time required for a target device to process each. Next, attacker measures correlation between the measured times and the estimated time for the first step assuming the first step mixes in a zero bit. A second correlation is computed using estimates with a bit value of one. The estimates using the bit value actually used by the target device should show the strongest correlation to the observed times.

Countermeasures such as adding random delays increases the number of measurements required, but does not prevent the attack. Obviously, making all computations take exactly the same amount of time that would eliminate the attack, but few programs running on modern processors operate in exactly constant time.

3) Power analysis attack:

There are other ways to leak devices information apart from timing such as current drawn by hardware is related to computation it is performing. Power consumption is the integrated circuit increases if more transition occurs. There are two main categories of power analysis attacks, namely, simple power analysis (SPA) and differential power analysis (DPA).

SPA attacks rely on the observation that in some systems, the power dissipated for particular cryptographic computation is equivalent to cryptographic key used.

To calculate secret keys for complex and noisy power consumption DPA is used. For a typical attack, an attacker thousands of samples of the target device's power consumption. These samples are collected using high-speed analog-to-digital converters such as Digital oscilloscopes.

After the data collection, the attacker makes a guess about the key. DPA allows attackers to pull extremely low power signals from extremely noisy data, without even knowing the design of the target system. Countermeasures that reduce the quality of the measurements only increase the number of samples the attacker needs to collect, and do not prevent the attack.

Countermeasures to these attacks requires effective tamper resistance, particularly when used for securing payments, audiovisual content, and other high-risk data, remain expensive and challenging. [14]

4) DMA attack:

Memory access is an important technique in the area of forensic analysis. Offline analysis of the memory allows an investigators to see the state of the operating system. Unlike trying to find a malicious activity on a live systems, investigators can see the data without the operating system.

The main idea of hardware-oriented approaches is to bypass the operating system using a physical device. The device creates a memory access through a single channel, which is independent of the operating system. Very often DMA (Direct Memory Access) feature is used for direct access. This allows us to obtain a memory image without launching another process or having to use potentially contested features of the local system. A concept of Special-purpose PCI device can be used either for forensic purpose.

The firewire attack enabled bypassing of Windows authorization by reading the user password stored in memory. Thus, for security reasons, firewire port is usually disabled in many computers. Another potential path for DMA enabled memory access seems to be the network card. [29]

VI. CONCLUSION

After studying all these techniques of low level memory attack on automotive embedded system, we can conclude most of the techniques are not actually possible or economically costly. For reverse engineering techniques we have to remove the chip from the ECU but getting hardware access is not possible in real life situation. Side channel attacks can be useful to cryptographic attacks which can help to break the passwords for efficiently and accurately. But as cryptographic algorithm complexity increases, datasets also increases and time required for successful attack also increases. On the other hand logical attacks such as buffer overflows and ROP can be useful as they are software based and economical for automotive embedded systems.

REFERENCES

[1] Sri Parameswaran, Tilman Wolf, "Embedded systems security—an overview:," in *springer*, 2008.

[2] I. McLoughlin, "Secure embedded systems the threat of reverse engineering," in *IEEE*, 2008.

[3] Vijaya Prathap, Abhishaks Rachumalu, "Penetration Testing of Vehicle ECUs," CHALMERS UNIVERSITY OF TECHNOLOGY, Gothenburg, Sweden, 2013.

[4] Dorottya Papp, Zhendong Ma, Levente Buttyan, "Embedded Systems Security: Threats, Vulnerabilities, and Attack Taxonomy," in *IEEE PST*, 2015.

[5] N. Zlatanov, "Computer Memory Applications and Management," researchgate, 2016.

[6] Vijeet H. Meshram, Ashish B. Sasankar, "Security in Embedded Systems: Vulnerabilities, Pigeonholing of Attacks and Countermeasures," in *NCRTCSIT*, 2016.

[7] Dr. Charlie Miller, Chris Valasek, "Remote Exploitation of an Unaltered Passenger Vehicle," August 10, 2015.

[8] V. Bordyk, "Analysis of software and hardware configuration management for pre-production vehicles," Chalmers University of Technology, Göteborg, Sweden, 2012.

[9] U. Erlingsson, "Low-level Software Security:Attacks and Defenses," *Springer*, no. Handbook of Information and Communication Security, pp. 633-658, 2010.

[10] U. Drepper, "What Every Programmer Should Know About Memory," Red Hat, Inc., 2007.

[11] Oliver Kömmerling, Markus G. Kuhn, "Design Principles for tamper resistant smartcard processors," 1999.

[12] Franck Courbon, Sergei Skorobogatov, Christopher Woods, "Reverse engineering Flash EEPROM memories using Scanning Electron Microscopy".

[13] Randy Torrance, Dick James, "The State of the Art in IC Reverse Engineering," *springer*, vol. 5747, pp. 363-381, 2009.

[14] Paul Kocher, Ruby Lee,Gary McGraw, Anand Raghunathan, Srivaths Ravi, "Security as a New Dimension in Embedded System Design," in *ACM*, San Diego, California, USA, 2004.

[15] J. W. oh, "Reverse engineering FLASH memory for fun and profit," HP.

[16] Kurt Rosenfeld, Ramesh Karri, "Attacks and Defenses for JTAG," in *IEEE*, 2010.

[17] N. Jones, "Exploiting Embedded Devices," SANS Institute, 2012.

[18] Ashley Hall , Huiming Yu, "Tutorial on Buffer Overflow Attacks," North Carolina A&T State University.

[19] Xin Zhang, HaiYan Liu, "Design and Implementation of Remote Buffer Overflow and Implanted Backdoor," in *Atlantis Press*, 2012.

[20] S. Gupta, "Buffer Overflow Attack," *IOSRJCE*, vol. 1, no. 1, pp. 10-23, May-June 2012.

[21] J. Nelißen, "Buffer Overflows for Dummies," SANS Institute, 2002.

[22] A. One, "Smashing The Stack For Fun And Profit," *Phrack Magazine*, vol. 7, no. 49, 1996.

[23] Kyung-suk Lhee, Steve J. Chapin, "Buffer Overflow and Format String Overflow Vulnerabilities," Syracuse University, 2002.

[24] J. Hulse, "Buffer Overflows: Anatomy of an Exploit," 2012.

[25] Marco Prandini and Marco Ramilli, "Return-Oriented Programming," *IEEE Security & Privacy*, vol. 10, no. 6, pp. 84 - 87, Nov.-Dec. 2012.

[26] scut / team teso, "Exploiting Format String Vulnerabilities," Stanford, 2001.

[27] YongBin Zhou, DengGuo Feng, "Side-Channel Attacks: Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing," IACR Cryptology ePrint Archive, 2005.

[28] Dr. Sastry .J.K.R ,Sasi Bhanu. J, SubbaRao. K, "Attacking Embedded Systems through Fault Injection," in *IEEE*, 2011.

[29] Štefan Balogh, Miroslav Mydlo, "New Possibilities for Memory Acquisition by Enabling DMA Using Network Card," in *IEEE*, 2013.