KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Robust Authentication for Automotive Control Networks through Covert Bandwidth

Stien Vanderhallen

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdoptie Veilige software

Promotoren:

Prof. dr. ir. F. Piessens
Dr. J.T. Mühlberg

Assessoren:

Ir. J. Van Bulck
Dr. B. Lagaisse

Begeleiders:

Dr. J.T. Mühlberg
Ir. J. Van Bulck

Academiejaar 2019 – 2020

Preface

Before uncovering the perks of covert communication, some words of gratefulness are due. This master's thesis could never have come to be what is now, were it not for a special set of people dedicating their time and focus to standing by me along the way. First of all, my daily advisors Jan Tobias Mühlberg and Jo Van Bulck deserve special thanks, for immersing me in their unique approach to structuring chaos, and Socratic style of guidance respectively. They consistently provided me with all the support, knowledge and motivation I needed, even when life was reduced to its least humane components during the last few months. I would also like to thank my promotor Frank Piessens and my co-promotor Jan Tobias Mühlberg, for allowing me to work on a subject that initially did not have any clear boundaries, or a well-defined direction. On that note, I am deeply thankful for the numerous open source communities that have enabled this work, and many others, to be of substantially more interest than they could ever have been without. My sincere gratitude goes to the reader as well, for taking their time to explore this work, and more so its subject, hopefully enjoying it at least as much as I did.

On a less technically involved note, yet not a less important one, special thanks go to my beautiful family and friends, for incessantly offering me the greatest advice, and for understanding when I sometimes, in a flare of stubbornness, blatantly ignored it. Looking back on the highs and lows in working on this thesis, and by extension the past five years of studying, this group of companions made that journey even more exciting than its destination.

Stien Vanderhallen

Contents

Abstract

Recent advancements in the automotive industry have extended vehicles with elaborate systems for entertainment and navigation, that are connected to wide-range networks like the internet. With the onboard embedded devices that enable those novelties therefore becoming remotely accessible, internal vehicle networks are exposed with them, and by extension all components they connect. Controller Area Networks (CAN) are in widespread use for governing communication within vehicle boundaries, and inherently offer little resistance against the security threats that come with remote, possibly malicious, parties gaining access to them. Considering the safety-critical application domain of CAN, remote attackers could however cause great harm when unrestricted in their actions.

Several existing security mechanisms are effective in hardening CAN nodes, and embedded devices in general, against such increased malicious activity. For instance, cryptographic protocols have been put in place to establish the authenticity of CAN traffic. However, such communication-specific security measures come at the cost of bandwidth, computational resources, and real-time guarantees. Their repercussions on performance and timeliness go against the very nature of automotive applications, thus implying their own effects on vehicle safety.

Taking those considerations into account, recent research proposes to implement CAN communication security measures using *covert channels*, which in essence are collateral transmission forms that do not increase network load, to enable strong security guarantees without harming safety as much as existing solutions. This master's thesis further explores that approach, in three main areas. First, an overview of covert, and covert-like, bandwidth sources applicable to CAN is constructed, which extends the set of covert channels considered in existing work. Second, a timing based covert channel is selected from that overview, and assessed in different practical implementations, each designed to yield stronger guarantees of performance and reliability than its precedent. Third, an extension to an existing security framework for CAN applications is proposed, which hardens its message authentication mechanism against packet loss through that same timing channel. The security implications of that approach are analysed, and found not to harm that framework's original security guarantees.

This master's thesis investigates and assesses covert bandwidth opportunities in CAN, and migrates their use to the practical context of enhancing an existing security framework's approach to CAN message authentication.

Samenvatting

Recente ontwikkelingen in de autoindustrie hebben voertuigen uitgebreid met vooruitstrevende systemen voor entertainment en navigatie, die verbonden zijn met wijd uitgestrekte netwerken zoals het internet. Omdat zo de ingebouwde voertuigonderdelen die instaan voor die functionaliteit, bereikbaar worden vanop lange afstand, stellen die ook de interne netwerken van hun voertuig bloot, en daarmee alle componenten die daaraan verbonden zijn. Regelnetwerken worden gebruikt om de communicatie binnen een voertuig te accommoderen, en bieden intrinsiek weinig bescherming tegen de veiligheidsdreigingen geïmpliceerd door dergelijke, mogelijks kwaadwillige, toegang vanop afstand. Gegeven de kritieke functies van voertuigen, kan een aanvaller echter enorme schade aanrichten wanneer die vrij spel krijgt.

Verschilllende veiligheidsmechanismen zijn reeds ontwikkeld, die effectief zijn in het beschermen van regelnetwerk-geconnecteerde componenten, en ingebouwde computersystemen in het algemeen. Zo zijn reeds cryptografische protocols in werking gesteld om de authenticatie van regelnetwerk-berichten mogelijk te maken. Dergelijke communicatie-gerelateerde oplossingen kosten echter bandbreedte, rekenkracht en realtime garanties, wat opnieuw de veilige werking van voertuigen bedreigt.

Recent onderzoek stelt vanuit die overwegingen voor om *verborgen kanalen*, wat in essentie collaterale transmissievormen zijn die de netwerkbelasting niet verhogen, te gebruiken in de implementatie van veiligheidsmechanismen, om zo hun druk op netwerkperformantie te verlichten zonder afbreuk te doen aan de bescherming die ze bieden. Deze masterthesis gaat verder op dat elan, voornamelijk in de volgende drie richtingen. Ten eerste wordt een overzicht gemaakt van bronnen voor verborgen bandbreedte in regelnetwerken, als uitbreiding op de verborgen kanalen overwogen in bestaand onderzoek. Ten tweede wordt daaruit een kanaal geselecteerd, dat gebaseerd is op de timing van netwerkpakketten, en dat vervolgens wordt geëvalueerd in verschillende implementaties, elk ontworpen om meer bandbreedte en betrouwbaarheid te bieden dan hun voorganger. Ten derde wordt een extensie voor een bestaand veiligheidsmechanisme voorgesteld, die diens berichtauthenticatie robuuster maakt tegen pakketverlies door middel van datzelfde timing kanaal. De veiligheidsimplicaties van die extensie worden uitvoerig onderzocht, en bevonden geen afbreuk te doen aan de originele veiligheidsgaranties van diens context.

Deze masterthesis concretiseert en evalueert bronnen van verborgen bandbreedte in regelnetwerken, en migreert het gebruik ervan naar de praktische context van een bestaand veiligheidsmechanisme, om diens berichtauthenticatie te versterken.

List of Figures

List of Tables

List of Abbreviations

CAN	Controller Area Network
ECU	Electrical Control Unit
DLC	Data Length Code
CRC	Cyclic Redundancy Code
DoS	Denial of Service
MAC	Message Authentication Code
TEE	Trusted Execution Environment
PM	Protected Module
TCB	Trusted Computing Base
PSA	Protected Storage Area
NG	Nonce Generator
IAT	Inter-Arrival Time
LSB	Least Significant Bit
ITT	Inter-Transmission Time
ISR	Interrupt Service Routine
LCM	Least Common Multiple

Chapter 1

Introduction

This introductory chapter motivates and summarizes the work done in this master's thesis, through respectively discussing its problem statement, research hypothesis and main contributions. Finally, an outline of the remainder of this text is given.

1.1 Problem Statement

Modern day vehicles are equipped with an unprecedented amount of embedded computing devices, that regulate and enhance the experience of both drivers and passengers. Controller Area Networks (CAN) serve to connect such in-vehicle components and are widely used so by the automotive industry, as well as in building automation, factory control and agriculture. Given those application domains, CAN often plays a major role in safety-critical, real-time sensitive functions, such as emergency braking and parking assistance.

Therefore, CAN network security in the form of, e.g., access control and message authentication, was sacrificed for performance and timeliness. That is a sensible approach in the assumption that CAN networks are inaccessible beyond, e.g., vehicle boundaries in an automotive context. However, the rise of onboard systems for entertainment and navigation broke that assumption, by connecting a subset of embedded components not only to CAN, but also wider-reaching networks like the internet. As such, a remotely accessible vector for penetrating CAN networks was introduced, which resulted in a wide range of known practical attacks against them, and by extension the devices they connect [12, 35, 6]. Those findings show that given the safety-critical functions performed by embedded components, major harm could follow from their CAN communication being tampered with.

Cryptographic protocols for message authentication [36, 34, 55], and trusted execution environments that protect the software using those protocols [31, 32, 51], have been proposed and implemented to harden CAN nodes against such malicious activity. Despite their strong security guarantees however, existing message authentication measures double the amount of CAN bus congestion, and consequently weaken real-time application properties, which again endangers vehicle safety.

A difficult balance thus exists in CAN applications, between employing resource-demanding security measures on the one hand, and resorting to attacker-sensitive performance enhancements on the other hand. Both are motivated by the safety demands of this automotive context, yet incompatible by definition. TACAN [58] therefore proposes to alleviate the resource demands of message authentication measures, by using covert channels in their implementation. Those non-conventional transmission forms exploit behavioural properties of regular, bandwidth-consuming traffic for their information carrier, and consequently incur no extra network load. As such, they enable strong security guarantees, without compromising vehicle safety.

1.2 Research Hypothesis

As introduced by TACAN [58], covert channels in CAN provide with supplementary bandwidth, that can be used to mitigate the safety issues incurred by existing message authentication mechanisms, without compromising their security guarantees. VulCAN [51] is a specific example of a security design that could benefit from incorporating covert communication into its approach to message authentication. Although it exhibits great efficiency in its implementation, some challenges related to VulCAN performance and availability remain, for example in its nonce synchronisation mechanism. Covert channels could mitigate such issues, e.g., by enabling explicit nonce transmission without harming VulCAN performance.

From those considerations emerges the research hypothesis of this work, which states that covert bandwidth is a useful resource in enhancing existing measures for message authentication in CAN. In investigating that hypothesis, this master's thesis explores what covert channels can exist in CAN, how those behave in relation to regular, non-covert communication, what amount of bandwidth and reliability they offer, and how they can enhance VulCAN's approach to nonce synchronisation.

1.3 Contributions

The main objectives of this thesis are to model opportunities for covert transmission in CAN, assess their feasibility in practice, and apply them for defensive purposes in existing security frameworks. The main contributions made are listed below.

— A breadth-first exploration of covert, and covert-like, bandwidth sources applicable to CAN is executed, and systematically summarised in a matrix structure. The latter allows for direct comparison of the identified transmission methods, based on a property set constructed to illustrate practical usability.

— An in-depth investigation of timing based covert communication in CAN is conducted. That analysis is supported by the practical implementation, and quantitative evaluation, of a message inter-arrival time channel on MSP430 hardware, which for the purpose of timing measurement accuracy is extended with an interrupt mechanism dedicated to CAN message arrival. This implementation is publicly available at `https://github.com/Stienvdh/Sancus-IAT`.

— An extension to the existing VulCAN [51] design is proposed, that equips its vatiCAN backend with an availability-enhancing nonce synchronisation strategy built on timing based transmission. The security implications of that approach are discussed. Furthermore, a practical implementation, and evaluation, of this VulCAN extension is done, and made publicly available for upstreaming in VulCAN at https://github.com/Stienvdh/vulcan/tree/iat-nonce.

1.4 Outline

Below, an outline of the remainder of this thesis text is given.

Chapter 2 provides with the background information relevant for this work. It first introduces the CAN protocol itself, its vulnerabilities, and a set of known practical attacks. Then, it elaborates on the concept of trusted execution, as well as Sancus and VulCAN. Finally, it discusses side-channel attacks, and side channels themselves, as well as illustrates those with concrete examples.

Chapter 3 gives an overview of covert channels applicable to CAN. It first discusses each channel's mechanisms and characteristics. Next, that information is structured in a matrix, that allows for direct comparison of its constituting channels.

Chapter 4 elaborates on a covert channel relying on packet inter-arrival timings, by means of a proof-of-concept implementation in Java. This chapter first introduces the channel's core design elements, and then assesses its performance and reliability.

Chapter 5 migrates the packet inter-arrival time channel introduced in the previous chapter, to low-level technology. It addresses the design choices made in that implementation, which is then assessed and compared against its Java counterpart. In an effort to improve channel performance even further, this chapter introduces and assesses an alternative interrupt driven implementation. Finally, it lists the sources of noise this packet inter-arrival time channel is generally subject to.

Chapter 6 describes a specific application context for the low-level packet inter-arrival time channel of the previous chapter. Motivated by security- and performance-related challenges in VulCAN's approach to nonce synchronisation, as well as the benefits gained from covert communication, it proposes an extension to VulCAN's vatiCAN backend that leverages the aforementioned timing channel for nonce synchronisation. This chapter concludes with an extensive security analysis of that proposed VulCAN extension.

Chapter 7 serves as a conclusion to this work, by repeating its main contributions, acknowledging its limitations, and proposing areas for future research.

Chapter 2

Background and Related Work

This chapter positions the remainder of this work in its context of previous research on CAN security threats, trusted execution and side-channel attacks. From the combination of those fields emerges the purpose of this master's thesis, which is an attempt to improve CAN application security through repurposing side channels from attack vectors to software-controlled, supplementary bandwidth, presuming this approach pressures the strong constraints on timeliness and performance imposed by the automotive context of CAN less than existing trusted execution designs.

2.1 CAN Security Threats

This section presents some core subjects in previous research on CAN security, and the lack thereof. It first discusses the CAN protocol, to introduce the mechanisms that lead to its subsequently described vulnerabilities and practical attack scenarios.

2.1.1 CAN Protocol Design

Application context. The Controller Area Network (CAN) protocol [45] is used by the automotive industry for intra-vehicle communication. It serves to connect, e.g., the electrical components regulating a vehicle's brakes to its parking sensors, or its dashboard velocity meter to its wheels. Such parties connected to a CAN bus are referred to as Electrical Control Units, or ECUs. Figure 2.1 shows a simple CAN network topology, with several ECUs connected to the same CAN bus.

CAN bus. To fit this context of mostly embedded computing devices taking part in CAN communication, CAN is built upon a broadcast medium without an addressing mechanism, thus relieving from any overhead related to connection management and/or network joining and leaving. Moreover, ECUs are not synchronized in their access to a CAN bus, meaning all connected ECUs can both transmit on and read from their CAN bus at any time. Preventing from unstable bus behaviour due to this design, a 0-bit is deemed dominant, which means its transmission overwrites a (recessive) 1-bit. The bus itself is clocked at a configurable frequency.

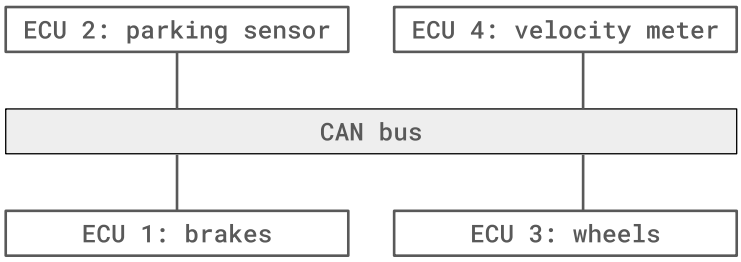

Figure 2.1: Graphical representation of an example CAN network topology connecting different components within a vehicle

start of frame	**(extended) ID**	data length code	payload data	CRC	ACK	end of frame

Table 2.1: CAN data frame fields

CAN data frame design. A CAN data frame consists of different fields, a simplified overview of which is given in Table 2.1. First of all, an *ID field* is leveraged for associating messages to their corresponding application and/or sender in the absence of an addressing mechanism. It has a length of 11 bits in original CAN, and is extended to 29 bits in CAN 2.0 [45]. Next, a 4-bit *Data Length Code* (DLC) announces the size in bytes of the *payload data* carried by the message at hand, which is of a variable length up to 8 bytes. A *Cyclic Redundancy Code (CRC)* [4] furthermore serves in detecting erroneous transmission, by forming a 15-bit checksum calculated over the transmitted message. An *acknowledgement* (ACK) bit at the end of a frame is to be set by any node on its CAN bus, to confirm transmission.

Bus arbitration. As to assert some degree of collision avoidance, CAN nodes can only initiate transmission after detecting 3 consecutive free CAN bus cycles. Moreover, they sense their bus for broadcasting the expected bit values while transmitting, and stop transmission whenever a discrepancy is detected. Evidently, the latter can only occur when one of two concurrently transmitting nodes sends a dominant (0) bit, while the other sends a recessive (1) bit.

Consequently, the ID field serves the purpose of denoting not only its message's origin, but also its priority. Indeed, the lower a frame's ID, the higher chance it has of retaining bus occupation until transmission of its data payload.

2.1.2 CAN Network Attacks

Modern vehicles go beyond mere internal networking, in that they are remotely accessible through internet-connected systems for entertainment and navigation. Their internal CAN networks, as well as the in-vehicle components they connect, thus are now exposed to remote attackers, against which little security measures were taken in CAN protocol design. Below, a CAN network attacker profile and some practical attack scenarios are discussed.

Attacker Profile

A wide range of CAN network attacker capabilities has been described to be enabled by both the nature of CAN communication, and its remote accessibility [12, 18, 56]. Their execution requires gaining control over the transmission logic of an ECU connected to the targeted CAN bus, either through physical attachment of an in-vehicle component [35], or a remote code injection [6] that leverages external vehicle accessibility. Below, the main capabilities enabled by such ECU control are listed.

— **Eavesdropping**: Since CAN is built upon a broadcast medium without any access control mechanism, ECUs are free to record and inspect all traffic on the CAN bus they are connected to. CAN moreover provides no confidentiality mechanism in its design, which means malicious nodes receive the same plaintext data as the benign nodes they are eavesdropping on.

— **Message insertion**: Due to CAN not providing with any bus access control and/or transmitter authentication, a malicious ECU can inject a message with any ID and data payload on a CAN bus, and have it be successfully received by other nodes on that network, as long as it conforms to the specification of a CAN frame format, and its CRC field carries a suitable checksum.

— **Message deletion**: As elaborated on before in Section 2.1.1, any 1-bit on a CAN bus can be overwritten by a 0-bit, and CAN nodes stop message transmission whenever they detect such overwriting. Thus, appropriately timed 0-bit transmission by a malicious ECU can cause deletion of any CAN message.

Denial of Service Attacks

The majority of existing CAN attack scenarios leverages the message deletion capability introduced before for launching a Denial-of-Service (DoS) attack that, e.g., suppresses CAN packets governing brake actions [35, 12]. In this automotive context, such paralysing of a vehicle could obviously have major safety-related consequences, as is the case in blocking brake messages, or in deleting packets that warn for component malfunctioning as another example. Given the lack of an authentication, or access control, mechanism in CAN, there is no limit to the set of applications that are vulnerable to such denial of service.

Different approaches to denial-of-service attacks have already been described [12], ranging from total bus congestion to suppress all CAN traffic, to more intricate, selective attacks that aim to circumvent detection. Due to their eavesdropping capability mentioned before, malicious nodes are able to monitor their CAN bus while launching a DoS attack, and thus can choose to target only a selected set of vehicle operations, based on their associated CAN message ID. Moreover, as message transmission is halted at any 1-bit overwrite, even just one meticulously timed 0-bit transmission suffices to suppress a complete CAN message [35]. Malicious nodes can thus cause great harm while transmitting only little extra traffic, which benefits them in face of an intrusion detection system [28], that in contrast can easily detect equally harmful bus flooding.

Detecting such malicious bit, or message, injection can for example be done by first modelling the timing characteristics of benign traffic on a specific CAN bus, and using those results in assessing the trustworthiness of subsequently monitored traffic [28]. When some node is detected to behave maliciously by such a scheme, these known techniques for launching a DoS attack can be leveraged for security instead of harm, by forcing that node into silence.

2.1.3 CAN Application Attacks

Remote accessibility of internal vehicle networks inevitably serves as an attack vector for remote code injection targeted at the applications running on the ECUs connected to them. That threat has been demonstrated by several practical attacks that hijack, e.g., in-vehicle applications for braking and door locking [27, 6, 18, 15, 26]. In this automotive context, the repercussions of application logic modification can be severe, taking the compromise of speed regulation in parking assistance software as an example. This class of attacks extends the attacker capabilities listed in Section 2.1.2 with higher-level malicious activity, i.e., not only in CAN traffic itself, but also in its management and processing. The application attacker profile considered here thus has the added capability of arbitrary code execution on the nodes attached to its targeted CAN network, rendering it substantially more powerful.

A particularly dangerous application attack scenario lies in the compromise of a gateway ECU that is installed to connect two network segments of different security privileges [18, 26], such as one governing vehicle speed and another accommodating an entertainment system. Due to the latter being remotely accessible by definition, that gateway is exposed to remote code injection, thus yielding an application attacker access to the safety-critical segment as well. Note that when not restricted to long-distance remote vehicle access, such an attack can be as easy to execute as connecting a malware-infected smartphone to the targeted car's Bluetooth [6].

2.2 Trusted Execution

In the face of, amongst many others, the CAN application security threats discussed in Section 2.1, several measures have been proposed that aim to weaken network attackers in their capabilities, as well as harden software modules against compromise. This section elaborates on a subset of those efforts, both on a theoretical and practical level. First, some security concepts are introduced, which are then illustrated in a discussion of concrete implementations.

2.2.1 Software Attestation

In software attestation, verification of the logic executed by a component is targeted. More specifically, its software is to be proven to equal some pre-defined, trusted version, as to ensure no malicious tampering with it has taken place. Concretely, attestation can be done through a *prover* node calculating a hash value over its loaded code, and a *verifier* component comparing that value against a pre-calculated

hash of the trusted code expected to be loaded. That comparison thus concludes attestation, and therefore should be done by a trusted party to be deemed secure. Moreover, no component should be able to produce the desired attestation hash value without having loaded the correct software, which means secure attestation requires a trusted component calculating tentative hash values.

In order to provide with those prerequisites, multiple architectures have been developed that enable software attestation at the hardware level [23], such as Sancus [31, 32] and Intel SGX [24]. Most of those extend their attestation capabilities with *software isolation*, which imposes access control on the code and data belonging to attested software modules. As such, it prohibits untrusted parties from tampering with such modules, in order for initial attestation results to retain their truth value. Moreover, to prevent return-oriented programming attacks, isolated software can only be executed starting from pre-defined entry points.

This traditional prover-to-verifier attestation scheme can be deemed unsuitable in certain application scenarios, e.g., due to performance considerations, which is why alternative approaches have been proposed. One such example is a mutual attestation mechanism [47] in which the prover and verifier parties take on both roles, thus relieving from the need for a single-purpose verifying component. Another advancement on "classic" attestation leverages the concept of aggregation, and extends the one-to-one relation between a prover and a verifier to a many-to-one alternative. SANA [1] is an example protocol enabling such aggregated software attestation, and introduces a notion of *aggregators* that combine attestation messages of either provers or other aggregators into one message to be assessed by a single verifier. As such, the software of multiple provers, like a swarm of IoT devices, can be attested in a single verifier interaction.

2.2.2 Message Authentication

As opposed to attestation, which (when isolation is enabled) is performed once at software initialization and involves hashing over a full module, message authentication is performed at every packet transmission, and takes the message at stake as a parameter in its procedure. More concretely, in order to authenticate a message, its sender typically calculates a Message Authentication Code (MAC) over (1) the message itself, (2) some secret key and (3) a nonce value providing freshness (cf. Section 2.2.5), and transmits that MAC along with the message at hand. Receiver parties can then verify the identity of a packet's origin, through validating such MAC values to be calculated over an appropriate message, secret and nonce.

Evidently, message authentication provides security guarantees different from software attestation. The latter is applied for secure bootstrapping, whereas message authentication aims for runtime security. Consequently, both are required in ensuring authentic execution of applications [33]. Message authentication without software attestation provides with authenticated traffic, that could be generated by malicious logic, and software attestation without message authentication yields zero security guarantees on runtime communication. Indeed, although the code executed by some component passes attestation, a message's origin cannot be verified in the latter case.

Several authentication protocols have been developed specifically for CAN. Some examples are vatiCAN [34], LiBrA-CAN [13], CANAuth [55] and LeiA [36]. The main common factor amongst those protocols is their aim for backward compatibility, i.e., CAN nodes not supporting their form of authentication should not break when other components on their CAN bus enable authenticated communication. AUTOSAR formulates standardized guidelines on such backward compatible in-vehicle message authentication [2], which are complied to by vatiCAN [34] and LeiA [36].

2.2.3 Trusted Execution Environments

As mentioned before in Section 2.2.1, some architectures provide their loaded software modules with guarantees for attestation and isolation at the hardware level. Most of those instantiate the concept of a Trusted Execution Environment (TEE) [42, 23, 52], which is defined to support the shielding of security-critical software components in *Protected Modules* (PMs), while leveraging some *Trusted Computing Base* (TCB) at any architectural level (e.g., Sancus in hardware [32], Fides at the hypervisor level [43], Salus in the operating system kernel [41]).

Protected modules. A protected module, or *enclave*, is divided into a code- and data-section. Both reside in the same address space as unprotected memory, yet are subject to a different, more restrictive access control policy that separates protected memory from unprotected memory. That access control mechanism is often defined by a program-counter based model [42, 44], which means access rights to a memory location are granted depending on both that location itself, and the program counter value at the moment access to it is attempted. Proper access control rules for shielding enclave memory [42] are listed in Table 2.2. Note that these rules deem all memory outside of the considered PM unprotected, which includes other PMs.

In essence, these rules enforce a protected code section to be read-only and executable by untrusted code, starting from only its pre-defined entry points. A protected data section, on the other hand, can be accessed for both reading and writing by only its corresponding code section.

From - To	**PMx**			**unprotected**
	entry points	*code*	*data*	
PMx	RX	RX	RW	RWX
unprotected	X	/	/	RWX

Table 2.2: Access rights granted to a memory location *To* when the program counter value is set to *From* in a trusted execution environment. RWX = Read/Write/Execution access

Trusted computing base. The security guarantees offered by a TEE assume correct functionality of a carefully designed, TEE-specific set of hardware and/or software components, which constitutes its TCB. A TEE's TCB serves to disable misbehaving components outside of it from harming TEE security guarantees, and therefore is desired to be as small as possible. Indeed, shrinking its TCB lowers the amount of critical components in a TEE, and as such enlarges resistance of its security guarantees against more powerful attackers. Furthermore, an architecture's TCB is preferred to reside in hardware rather than software from a security perspective, as hardware components generally are less vulnerable to compromise. A software-based TCB however allows for larger portability across platforms, which leaves optimal TCB design to be context-specific.

2.2.4 Sancus

Sancus [31, 32] is an open-source [30] TEE implemented at the hardware level, and enforces program counter-based access control through extending its nodes with a *Protected Storage Area* (PSA) holding metadata (i.e., PM code- and data-section locations) and cryptographic information belonging to the PMs loaded in its memory. Its design assumes a Sancus-enabled node N built by an infrastructure provider IP, onto which a software provider SP deploys one or more protected software modules SM_i. Based on those entities, each PM is assigned a software module key K_{N,SP,SM_i}, which is stored in the corresponding node's PSA.

Key management. Creation of $K_{N,SP,SM}$ for a software module SM starts from a node master key K_N that is first randomly generated, then securely stored by IP and finally loaded into the PSA of N. A software provider key $K_{N,SP}$ then is derived from K_N and a public SP identifier, upon which IP securely shares $K_{N,SP}$ with SP. Finally, $K_{N,SP,SM}$ is derived from $K_{N,SP}$ and a SM identifier, which is a hash of both its code section and the layout formed by the memory locations of its code- and data-section. Both key derivations involved in this construction are done through a hardware-implemented function, as to ensure they are unaffected by malicious software. $K_{N,SP,SM}$, as mentioned before, is stored in PSA, and therefore is inaccessible from software. Special instructions are however included in Sancus, that allow for indirect use of $K_{N,SP,SM}$ in a way that guarantees it to only be used during execution of the trusted code section belonging to its corresponding SM.

Remote attestation. Sancus moreover provides with strong remote attestation guarantees [10], in that the integrity of a protected module SM deployed by SP can be attested by SP with high assurance. Due to both SM and SP having access to $K_{N,SP,SM}$, both can compute a hash over respectively SM's loaded and expected contents. Authenticated transmission of the former resulting hash value to SP then yields the ability of remote, i.e., not on N itself, verification of SM content integrity. Due to $K_{N,SP,SM}$ only being available from its corresponding SM code section, SP has high confidence in correct attestation responses (1) having been calculated by the attested SM and (2) that SM not having been tampered with.

2.2.5 VulCAN

VulCAN [51] aims to enable message authentication, and software attestation/isolation in the specific context of CAN applications. This section discusses key principles in its design, and as a prelude to the application context investigated further in this text, focuses on its approach to nonce synchronisation.

VulCAN Design

VulCAN [51, 49] migrates the use of Sancus enclaves (cf. Section 2.2.4) to an automotive context, to enable authenticated CAN communication (cf. Section 2.2.2), as well as software attestation and software isolation (cf. Section 2.2.1) guarantees. It explicitly excludes operating system, networking and I/O interaction software from its TCB, thus allowing for both CAN traffic and unprotected ECU application logic to be taken over by an attacker while retaining its security guarantees.

Message authentication in VulCAN is done by accompanying each application message transmission with a subsequent authentication frame, that serves to carry a 64-bit MAC value calculated over the former. Two message authentication protocols, namely vatiCAN [34] and LeiA [36], due to their AUTOSAR-compliance [51, 2] have been selected to govern MAC calculation and MAC transmission details, each relying on Sancus' cryptographic primitives to minimally enlarge VulCAN's TCB size beyond the TCB it inherits from Sancus. More concretely, the software implementing those authentication protocols is required in VulCAN's TCB, and leveraging the already-there Sancus design for secure storage of the key material used in MAC-related computations lowers its size. Outside of application-specific software that is critical in CAN communication, VulCAN adds no other components to that TCB. To ensure trusted execution of the software-based part of its TCB, VulCAN can moreover be configured for it to be isolated in a Sancus enclave.

Nonce Synchronisation in VulCAN

Below, the general purpose of and difficulties encountered in nonce synchronisation first are discussed. A subsequent elaboration on how the concept is handled in both VulCAN backends, and what their respective weaknesses in that area are, renders those difficulties more concrete.

Nonce synchronisation. In the context of message authentication as implemented in VulCAN, nonces serve the purpose of providing with message freshness. More specifically, a counter value, or *nonce*, is associated with each application message, and used as a parameter in both the calculation and validation of that message's MAC. As opposed to the cryptographic key used as another parameter in that MAC calculation, nonces can safely be disclosed to any party. Indeed, obtaining the nonce associated with some payload does not suffice for an attacker to construct a valid corresponding MAC, as it is generally assumed not to have access to that secret key.

By using a unique nonce for each message, any two valid MAC values are calculated using a different nonce, which means those values themselves most likely are not

equal, depending on the MAC algorithm strength. As such, valid MAC values are to some extent guaranteed to correspond to *fresh* messages, i.e., messages that have not been authenticated and processed as such before. Therefore, attackers who in themselves are not capable of producing valid MAC values, cannot successfully resort to replaying authenticated traffic on a CAN bus and trick receivers into accepting it.

This mechanism poses a non-trivial challenge in nonce synchronisation between a sending (MAC calculation) and receiving (MAC validation) node, i.e., in rendering that receiver, when it validates a MAC, aware of the nonce used by the sender when calculating that MAC. Both VulCAN backends currently use 32-bit nonces, which places a large load on a CAN bus, should nonces be transmitted along with their messages. Partially reverting to implicit nonce synchronisation thus is preferable, e.g., by having both the sender and receiver store a local nonce, which they increment on successful transmission, respectively arrival, of an authenticated message. Such an approach however introduces its own complications when message loss occurs, as a receiving node's local nonce then could get behind on the sending node's.

Moreover, an extra challenge lies in guaranteeing nonce uniqueness, and therefore message freshness, while limited to a finite nonce space. Different strategies have been implemented with respect to these considerations, two of which are applicable to VulCAN and therefore discussed below, each with their own implications for bandwidth use and security guarantees.

LeiA backend. VulCAN's LeiA [36] backend leverages the extended CAN ID field of both application messages and authentication messages for transmission of a 16-bit counter that constitutes the lower half of the nonce associated with them, and that is incremented by one on every authentication message transmission. To account for the upper 16 nonce bits associated with a message, LeiA uses locally stored *epoch values* that are both incremented and explicitly transmitted on their CAN bus on either ECU reset, 16-bit counter overflow, or MAC verification failure.

As such, LeiA has the advantage of relieving from the bus load of full nonce transmission, while retaining robustness against message loss by explicitly transmitting partial nonce values. However, LeiA's use of the extended CAN ID field breaks backward compatibility with applications whose functionality relies on it. Indeed, those can no longer use a 29-bit message ID space, due to part of it being reserved by LeiA. Moreover, VulCAN assessment [51] shows how this scheme places considerable pressure on the performance of legacy applications not using extended IDs, due to its implication of extra CAN bus interactions in transmission.

vatiCAN backend. In contrast to LeiA, vatiCAN does not modify regular CAN traffic for the purpose of nonce synchronisation. Instead, sending and receiving nodes keep a local nonce value, which is incremented on successful authentication frame transmission, respectively arrival. As mentioned before, such a scheme however breaks on message loss. Figure 2.2 illustrates how vatiCAN authentication indeed fails after a packet is lost, due to the local receiver nonce n_R differing from the local sender nonce n_S in subsequent communication.

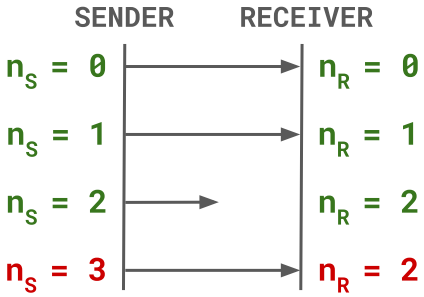

Figure 2.2: Illustration of sender and receiver nonces n_S and n_R going out of sync due to message loss in the vatiCAN backend of VulCAN. After message loss, authentication fails on subsequent communication because of the receiver expecting a lower nonce to be used than actually used by the sender in MAC computation.

To prevent from all communication after message loss being discarded because of failed authentication, vatiCAN provides with a Nonce Generator (NG) component, which periodically broadcasts randomly generated values that serve to resynchronise the local nonces of all participating parties. That approach however renders vatiCAN susceptible to advanced replay attacks, due to such randomization causing nonce reuse, as shown by VulCAN [51]. The main existing effort to eliminate this threat, is an increase of vatiCAN nonce size to 64 bits [7], which does not suffice as a suitable mitigation. Acknowledging that vulnerability, VulCAN excludes NG from its vatiCAN backend, thus trading off message loss recovery for replay attack resistance.

2.3 Side-Channel Attacks

General principles. Despite numerous efforts to establish security guarantees in embedded systems, as discussed in Section 2.2, recent work has unveiled the severity of the previously neglected category of side-channel attacks. The side channels used as their attack vector do not reside at the architectural level itself, as enclaved execution generally is successful in shielding trusted software from its untrusted environment, yet follow from the state changes caused at the microarchitectural level [17, 22, 50, 40] or the application level [39, 21, 20], which are observable by parties outside of protected modules. When effective as an attack vector, side channels typically leak private information on the execution and data of such trusted modules.

Some practical examples of side-channel attacks exploit microarchitectural leaks following from speculative execution [17, 22, 50] or cache access [57]. In Sancus specifically, the Nemesis [54] attack exposes interrupt latency timings as leaking private information on ongoing enclave execution, from which enclaved secrets like passwords and PIN codes can subsequently be deduced. More prevalent side effects can also carry such private information, taking the timing of packets in network transmission as an example of particular interest in the rest of this work. Below, some existing attack scenarios leveraging such timing channels are elaborated on, which illustrates the amount, as well as type, of information they can carry.

Timing attacks. Due to real-time considerations, most network protocols aim to induce as little latency in transmission as possible. Without any software-level countermeasures, timing properties of applications leveraging such networks thus propagate to the network traffic itself. When malicious parties can moreover eavesdrop on that traffic in real time, such timing information is disclosed with it [20]. Applications that leak sensitive data through their network packet timings, are therefore susceptible to an eclectic category of timing based attacks.

Lipp et al. [21] and KeyDrown [39] describe specific timing attack instances, that exploit keystroke interrupt timings to deduce the corresponding keys pressed on a keyboard. Those can subsequently leak sensitive information like inserted passwords and PIN codes. Such leaks are unharmful when only accessible by trusted parties, but that assumption is often too strong in the face of an ever more powerful attacker.

Detection of such attacks poses itself as a difficult task, since eavesdropping generally leaves no trace. Prevention efforts like KeyDrown [39] therefore offer a more effective mitigation strategy. KeyDrown proposes the injection of dummy keystrokes to "drown" real keystrokes in, as to obfuscate the information contained in their timing. When translating that concept to the insertion of dummy traffic in CAN, that approach however imposes an obvious bandwidth constraint, which most automotive networks cannot accommodate, leaving this approach inapplicable to CAN. Another, less bandwidth-demanding, prevention strategy entails the introduction of randomized latency in keystroke message transmission [38], which might break real-time dependent functionality of some applications as a drawback.

From those considerations emerges an *anonymity trilemma* as formalized in existing research [8, 20]. Indeed, a three way trade-off exists between anonymity, latency and consumed bandwidth, due to all approaches in anonymizing message timing traces requiring either extra bus congestion, or deviation from regular frame transmission latency. In original CAN design, anonymity was sacrificed to account for the strong constraints on performance and timeliness as imposed by its application context, which led to the considerably large set of CAN attacker capabilities listed in Section 2.1. Recent work [29] on CAN based system design explicitly states how that imbalance can no longer be justified due to modern remote attacker capabilities [27, 6], and proposes new design principles in search of a more appropriate balance between security, performance and real-time guarantees.

Practical example: VulCAN buffer comparison. VulCAN [51, 49], as described in Section 2.2.5, is designed to authenticate messages at receiver side by comparing a self-calculated MAC against a received MAC, which is currently implemented approximately[1,2] as follows:

```
authentication_failed = (mac_me.quad != mac_recv.quad)
```

[1] https://github.com/sancus-pma/vulcan/blob/master/can-auth/vatican.c
[2] https://github.com/sancus-pma/vulcan/blob/master/can-auth/leia.c

Although it is correct, this approach can possibly introduce a side channel. Indeed, this MAC comparison, in its compilation to machine language, could be transformed to a `for`-loop that subsequently checks 8 MAC byte pairs to be equal, and exits on a differing pair. That is a valid transformation, because once any 8 corresponding bits do not match, there is no need to check the remainder of the MAC values involved. However, the time spent on executing such a loop reveals which byte pair caused it to exit prematurely, should authentication fail. Measuring that timing thus yields a malicious party partial information on `mac_me`, which means a sequence of well-crafted `mac_recv` payloads could possibly leak `mac_me` entirely, which in its turn could lead to malicious traffic injecting `mac_me` and therefore unrightfully passing authentication. This attack scenario lies within an attacker's capabilities, as it can leverage its message insertion ability for `mac_recv` payload injection, and its eavesdropping capability for timing MAC comparisons.

This side channel was found to currently not exist in VulCAN, yet should be prevented in the light of forward compiler independence. As a minor contribution of this thesis, an implementation that extends VulCAN with the ability to perform guaranteed constant-time buffer comparison, and uses it in MAC comparison, is made publicly available for upstreaming in VulCAN at https://github.com/Stienvdh/vulcan/tree/ct-time-compare.

2.4 Conclusion

Existing work on CAN networks has revealed its substantial vulnerability to various attacks, mainly due to a lack of mechanisms for access control and message authentication. These security flaws were traded off for performance and real-time guarantees, as is reasonable in CAN's application domain of connecting electrical components that reside in a network presumably reaching no further than vehicle boundaries. However, recent developments in the automotive industry expose those internal networks to parties far beyond those boundaries, through incorporating internet-connected GPS devices and entertainment systems. Such remote accessibility enables attackers outside of a targeted vehicle, to penetrate into its unprotected CAN networks and thus cause great harm to its safety, e.g., by suppressing brake actions.

From those evolutions, amongst others, resulted a need for security measures that harden CAN traffic itself, and the applications governing it, against this new, more powerful attacker profile. A subset of those measures was discussed in this chapter, mostly focusing on software isolation and message authentication guarantees, as well as how those concepts are implemented in Sancus and VulCAN. Despite those successful efforts, a relatively new class of side-channel attacks reveals some remaining security threats, and at the same time the amount of information that can be carried in non-conventional transmission channels.

Chapter 3

Identifying Hidden Bandwidth Channels in CAN

To create a structured overview of the sources for hidden bandwidth in CAN communication, this chapter categorizes and compares a subset of them. First, the concepts of covert channels and hidden bandwidth channels are specified. The subsequent discussion on their instantiations in CAN is guided by the evaluation of a set of communication channel properties relevant for this work, which is elaborated on before the introduction of the hidden bandwidth sources themselves. Finally, a synthesis is presented in a matrix structure, that allows for direct comparison of its constituting hidden bandwidth channels.

3.1 Covert vs. Hidden Bandwidth

In order to provide with a framework for the terminology used in the rest of this text to be interpreted in, this section explicitly states what hereafter is meant by transmission being covert and/or hidden bandwidth, and how those categories relate to each other. Note that these definitions do not aim for an unconditional, or generally applicable formalisation of their concepts. Instead, they are constructed for the specific context of CAN networks, and their illustrative examples with them. Nevertheless, their core elements can easily be translated to other types of communication.

3.1.1 Covert Channels

This work defines a covert channel as *data transmission that (1) for its payload carrier leverages some behavioural property of an underlying form of non-covert communication (here, CAN traffic) and (2) in exploiting that carrier does not require write access to non-covert traffic data objects (here, CAN frames).* Below, an elaboration on that definition is given, covering its aspects that are deemed most interesting in the context of this work.

Timing/storage covert channels. Covert transmission is typically divided into two subclasses. *Timing channels* use packet timings as their covert payload carrier, whereas *storage channels* use memory locations, or data objects, that all participating nodes have access to. The definition above does not explicitly mention those two categories, yet allows for similar subclasses in covert communication.

Manipulation of a CAN packet's timing can be done regardless of its contents, which adequately qualifies packet timing as a covert payload carrier. Therefore, all timing channels applicable to CAN, such as the packet Inter-Arrival Time (IAT) channel proposed by TACAN [58], are considered covert here. In contrast, not all CAN storage channels introduced in previous work meet these covertness requirements. For example, TACAN proposes the overwriting of CAN data field Least Significant Bits (LSBs) with "covert" payloads as a source of extra bandwidth, which violates the second part of this covertness definition.

Position in existing work. Most literature on covert communication presents the concept as hiding data transmission from external parties, by "covering" it in underlying non-covert, or *overt* [59], traffic. That notion is explicitly included in the first part of the definition used here. Beyond that common ground however, similar to the disagreement between TACAN [58] and this work on LSB manipulation qualifying as a covert channel, previous research does not exhibit consensus on a more detailed definition of covertness, in that certain channels have been considered covert by some authors, and non-covert by others.

LSB manipulation is a good example of such an in-between case. A subset of existing research allows for covert channels to overwrite underlying traffic frames as long as it goes unnoticed at the application level [3, 59, 48, 58], and therefore categorises LSB manipulation as covert transmission. The definition above however prohibits covert channels from packet, and thus LSB, manipulation regardless of the application logic at hand, which follows a different group of authors [16, 19], amongst which Lampson [19] in the original definition of covert channels.

Application dependence. A remark should be made on the application dependence of this covertness definition. Indeed, the behavioural property of underlying transmission that is to be exploited for covert communication, typically emerges from the nature of the application governing it. Therefore, some covert channels lose their payload carrier when built upon a different application, which disables non-zero covert bandwidth transmission. Such a situation however does not lead to violation of this covertness definition, as there is no CAN frame write access needed in exploiting some covert payload carrier, when there simply is none to be exploited.

As such, whether a channel qualifies as covert does not depend on its underlying application, yet its ability to expose non-zero covert bandwidth does. An example of such dependence can be found in TACAN's packet inter-arrival time channel [58], which was already classified as covert earlier in this section, and requires its underlying application to employ periodic communication. By definition, it yields zero bandwidth transmission when built upon aperiodic non-covert communication.

Security implications. Most "covert" channels in existing literature are presented in a context for them to be leveraged for malicious purposes, more specifically in leaking confidential information out of security policy sight. Lampson in his original definition of covert channels [19], and the US Department of Defense in its later, alternative formulation [48], both explicitly require the sending party in a covert channel to have a privilege level different from the receiving party, as to enable security policy bypass through covert communication.

Following the arguments made for TACAN [58], this work poses no requirements on the security privileges of the parties participating in covert and/or hidden (cf. Section 3.1.2) communication. Indeed, covert transmission here is to be leveraged for defensive purposes, in an effort to increase the amount of bandwidth available for securing CAN communication. It is therefore intended to enforce security policies, rather than bypass them, which typically requires the possibility of participating parties to be of equal privilege levels.

3.1.2 Hidden Bandwidth Channels

CAN networks generally allow for only their bus to serve as a common area of storage for ECUs attempting covert communication. Only few, or even zero, storage covert channels can thus be identified in this specific context, as the data placed on that bus entirely consists of CAN frames, which are not allowed to be manipulated in covert communication as defined in Section 3.1.1. Therefore, a notion of *hidden bandwidth* is introduced here. It serves to classify channels that are of interest in the context of this work, yet do not qualify as covert according to Section 3.1.1.

More concretely, a hidden bandwidth channel hereafter is defined as *data transmission that (1) leverages some information carrier contained in an underlying form of non-hidden bandwidth (here, CAN traffic) and (2) in exploiting that carrier does not require write access to underlying traffic data objects (here, CAN frames) that are required read access to by the application governing them.* Note that the ability of a hidden bandwidth channel to expose non-zero bandwidth, similar to the remark made on the covert channel definition in Section 3.1.1, depends on its underlying application. Here, that dependence is more explicit than in that covertness definition.

This is a substantially weaker definition than was formulated for covertness, leaving all covert bandwidth to qualify as hidden, but not vice versa. Indeed, covert transmission allows no CAN frame manipulation, whereas hidden bandwidth communication can overwrite its unread parts. The inverse statement of all hidden bandwidth qualifying as covert does however not hold, as proven by the counterexample of the previously introduced LSB manipulation channel, which uses the least significant bits in CAN frame data fields for its payload carrier and therefore is not considered covert transmission here. However, when built upon an application that discards a number of least significant bits, e.g., because of low accuracy requirements, this approach can be considered hidden bandwidth transmission as long as it affects no more bits than are discarded in application logic.

Channel property	Description
Stack level	The level from which a channel can be controlled (hardware/software)
Real-time compliance	Whether or not enabling a channel affects real-time properties of underlying transmission
Application dependence	Dependence on the nature of the application a channel is built on for it to expose non-zero hidden/covert bandwidth
Bandwidth parameters	Channel parameters constituting a channel's bandwidth formula
Bandwidth formula	Expresses the maximum bandwidth of a channel in bit/s as a function of its bandwidth parameters
Covertness	Denotes whether a channel is considered covert according to Section 3.1.1

Table 3.1: The properties taken into account for comparing hidden bandwidth channels in CAN

3.2 Channel Properties

As to enable a structured comparison of the transmission forms presented in this chapter, a set of core channel characteristics has been identified. Those constitute the columns of the matrix presented in Section 3.5, the rows of which are elaborated on in Section 3.3. It should be noted that this set is not exhaustive, i.e., it does not cover all properties that are of interest in discussing communication channels. For example, the sensitivity of a channel to noise on the CAN bus it is deployed on, influences the value of that channel to be used in a given context, yet is not included in this selection. Table 3.1 lists and defines the properties that were considered.

Taking into account the application dependence remarks made in the covert/hidden bandwidth definitions of Section 3.1, application dependence is explicitly included in this property set. It denotes whether the channel at hand poses restrictions on the applications it can be applied to in exposing non-zero hidden, or covert, bandwidth.

Real-time compliance could be considered an aspect included in application dependence, as real-time deadlines are imposed by that same application, yet is proposed as a separate channel property. Such real-time aspects are not considered in the covert/hidden bandwidth definitions of Section 3.1, on which the application dependence definition used in this chapter is based. Therefore, they are deemed to belong to a separate channel property here.

The characteristics deemed most important in the context of this work are software-controllability and covertness, as the channels satisfying both are of most interest for implementation purposes. The software-controllability of a channel allows for it to be portable across CAN-nodes and -buses, and its covertness closely fits the research hypothesis of this work, as well as generally lowers the application dependency of the amount of covert bandwidth available.

3.3 CAN Hidden Bandwidth Channels

This section elaborates on the hidden bandwidth channels listed in the matrix structure of Table 3.2. The channel identifiers used in that matrix, precede a description of each channel here. For every case, an elaboration on the core mechanisms of the corresponding channel is given, and some remarks regarding the channel properties listed in Table 3.1 if deemed necessary. This overview is structured by the CAN frame design presented in Table 2.1.

3.3.1 CAN Frame ID Field

ID1: Dedicating ID bits. In some application scenarios, the amount of distinct ID's needed to function properly is smaller than possibly representable using all 11, or 29 in CAN 2.0, ID field bits. Such applications can be configured to dedicate some amount of bits in the (extended) ID field of its CAN frames to hidden data. As a complete ID field must be sent over the CAN bus at hand at every message transmission, no more or less bits are to be sent when enabling this channel than when not. That means no influence on real-time deadlines is to be considered. Note that this channel excludes the use of extended ID bits for hidden data transmission in applications originally using 11-bit IDs, due to backward compatibility issues.

ID2: Manipulating arbitration collision frequency. Existing attacks against CAN [12, 35] describe how the arbitration mechanism in CAN can be leveraged to launch selective DoS attacks against CAN nodes. That technique can be repurposed as a covert channel, in that nodes can cause selective arbitration collisions in order to transmit covert data to their targeted node. As such, the frequency of collisions could serve as an encoding of some covert payload. An error mechanism included in the CAN protocol can take care of the underlying traffic not suffering any packet loss through enabling retransmission [45], but such extra network traffic might cause real-time compliance to deteriorate, and only a CAN transceiver-specific amount of retransmissions, and thus collisions, is allowed per message. Moreover, the governing of arbitration collisions is not generally possible from CAN driver software, restricting the applicability of this channel in this work to discussion purposes.

3.3.2 CAN Frame Data Field

D1: Dedicating LSBs. As described by TACAN [58], some applications are resilient to lowered accuracy in the data transmitted on their CAN bus. Some amount of LSBs can in that case be used for transmission of a hidden payload, as long as no more bits than allowed to be altered by the underlying application are repurposed as hidden data. This approach however does not classify as a true covert channel, since it modifies CAN traffic, which goes against the covertness definition of Section 3.1.1. As only manipulation of bits is done in this hidden bandwidth channel, without changing the amount of bits transmitted per CAN frame, nor its timing, this approach does not affect real-time properties of its underlying traffic.

D2: Manipulating packet size. Another approach to leveraging the CAN frame data field is using the size of the packets transmitted as a carrier of hidden information, by varying the length of their data field, and the value of their data length code field accordingly. The application such channel is built on must tolerate the associated variations in accuracy, i.e., it must not require a maximum amount of bits in the data field of its CAN frames at all times, similar to the application requirement of **D1**. In such case, the bandwidth of this channel is determined by the amount of bits that can safely be dropped from, or added to, each packet's data field.

Since this channel affects the amount of bits sent over its CAN bus, its influence on real-time compliance has to be considered. However, it allows for both lowering and enlarging the amount of bits sent, which means it does not necessarily, if at all, impact that compliance negatively.

D3: Data field padding. Whenever an application is known to use a predictable (in extremis, fixed) data field length in its communication, the data fields in its packets can be padded to their maximum of 8 bytes with hidden data. Naturally, this channel only yields non-zero bandwidth when that predictable data field length is lower than 8 bytes. Moreover, it increases the amount of non-hidden traffic generated, which has the potential of breaking its real-time compliance.

3.3.3 CAN Frame Error Detection Field

E1: Dedicating the CRC field. As CAN provides with automatic retransmission of frames when those carry an invalid CRC field, that field can be overwritten by hidden data on initial transmission, for a receiver to be monitored on the CAN bus before that transmission is deemed to be erroneous and retried. As the CRC field however is not controllable from software, this channel is not of interest for implementation purposes. Moreover, retransmission yields extra network traffic, which affects the real-time properties of underlying communication.

E2: Pass/fail of error detection check. In a less invasive manner than proposed by **E1**, it is possible to embed hidden information in the passing or failing of a packet's CRC check. More specifically, a sender can purposely manipulate its CRC field for it to yield an error when checked at receiver side. A receiver monitoring this channel can then decode traces of its CRC check outcomes, without needing the corresponding CRC fields themselves, to the intended hidden payload. The already provided capability of CAN nodes to handle CRC checks thus is leveraged for hidden bandwidth communication in this channel. However, as mentioned before in discussing **E1**, the CRC field is not controllable from software, which makes this channel of less interest here. Although it is application independent, this channel moreover incurs extra network traffic due to message retransmission, and therefore influences real-time compliance of its underlying non-hidden communication.

3.3.4 Timing Channels

Independent of the protocol over which non-covert communication happens, the timing of messages is a useful source of covert bandwidth. When provided with a sufficiently small clock granularity in software, and no exuberant load on the considered CAN bus, a reasonable amount of information can be embedded in the transmission- and corresponding arrival-times of messages. As this channel type is software-controllable, as well as truly covert according to the definition of Section 3.1.1, its instantiations are of great value for implementation purposes.

T1: Packet reordering. Not only the timing of a packet, but also its ordering relative to other packets can be an encoding of covert data. More specifically, when an application is in control of packet transmissions with different ID fields, it can adjust the order of its packets to a sequence from which hidden information can be decoded. That practice obviously impacts real-time behaviour and thus cannot be considered generally applicable, even less because of its prerequisite of the underlying application controlling multiple packet IDs.

The bandwidth formula listed for this channel in the matrix overview of Table 3.2 presumes covert data to be encoded in the ID field of each packet relative to the ID of the message immediately preceding it. Variations on this approach could take into account longer message sequences, and therefore result in other bandwidth formulas.

T2: Manipulating packet inter-arrival times. Even when not in control of multiple CAN message IDs, information can be encoded in the timing of packets, more specifically in their inter-arrival times. Assuming an application makes use of periodic communication with some fixed message inter-transmission time T, deviations from that value T can encode some covert payload, as proposed by TACAN [58]. Making a receiving party monitor packet inter-arrival times then enables it to decode those to the corresponding covert data. This approach obviously endangers real-time deadlines, and furthermore is only useful in applications using periodic communication.

T3: Combining inter-arrival times and reordering. As the ordering of payloads is a property orthogonal to the timings between them, **T1** and **T2** can be combined into one hybrid timing channel. More specifically, information can be encoded at sender side in the ordering of the messages on the one hand, and in their inter-transmission times on the other hand. A receiver can monitor both from software to then decode them to the corresponding covert data. Note that this approach also combines the underlying application prerequisites of both channels, which are governance of multiple message IDs and periodic communication in this case. Should an application be in control of only one message ID, this channel is reduced to the basic inter-arrival time channel discussed before as **T2**.

3.3.5 Hybrid Channels

As already illustrated by the **T3** channel discussed in Section 3.3.4, several of the channels described in this section can be combined into one hybrid channel. Two conditions should however be fulfilled before committing to such a combination:

— **Orthogonality**: The channels combined must be pairwise orthogonal, i.e., the use of one cannot interfere with the mechanisms enabling any of the others. To illustrate, the use of N ID bits to transmit hidden data (**ID1**) can seamlessly be combined with dedication of the M least significant bits in the CAN data field (**D1**). In contrast, overwriting those M least significant bits with hidden data serves no purpose if some bits in the data field are dropped/added for hidden communication via packet size (**D2**).

— **Prerequisite compatibility**: The applications for which a hybrid channel is suitable, must be in the common subset of the applications appropriate for the channels constituting that hybrid channel. Their application prerequisites thus should be compatible, i.e., they should not contradict each other.

Such combination of prerequisites was already illustrated in the hybrid channel of Section 3.3.4 (**T3**), where combining **T1** and **T2** restricted appropriate applications to those that have both control over multiple message ID's, and rely on periodic communication.

3.4 Constructing Channel Bandwidth Formulas

This section elaborates on what constitutes the bandwidth formulas listed in Table 3.2. First, a high-level abstraction applicable to all channels presented in Section 3.3 is made, which is followed by concrete instantiations. Hidden bandwidth channels are hereafter referred to as *parasite channels*, that are built on top of *host channels*.

In each case of the parasite channels proposed, functionality depends on some host communication channel providing with a message stream to be leveraged by that parasite channel. Naturally, the bandwidth provided by those hosting channels affects the parasite channel's bandwidth. More specifically, the packet rate of that hosting channel equals the amount of messages to every second be of use to the hidden bandwidth channel built on top of it. As such, when expressing the amount of bits transmitted by a hidden bandwidth channel per hosting message as b_{par} in *bits/message*, multiplication with the hosting channel's message rate m_{host} in *messages/sec* results in the bandwidth of that parasite channel (bw_{par} in *bits/sec*):

$$bw_{par} = b_{par} m_{host} \tag{3.1}$$

Explicit b_{par}. Some of the parasite channels presented in Section 3.3, are explicitly defined in terms of some amount of bits N transmitted per hosting message. Consequently, b_{par} in Equation (3.1) can simply be substituted by N when instantiating it for such a specific parasite channel. The channels **ID1**, **D1**, **D3** and **E1** fall under this category for bandwidth formula construction.

Deriving b_{par} from variation in behaviour. The rest of the parasite channels introduced does not transmit actual bits, yet makes the hosting channel behave differently depending on the information to be transmitted through hidden bandwidth. When expressing the amount of distinct behavioural states the parasite channel can cause the hosting channel to exhibit on each message transmission as s, calculating its binary logarithm yields the amount of hidden bits that can be encoded in one behavioural state, and thus can be transmitted when manipulating the transmission of one host channel message. That calculation is shown in Equation (3.2). The concrete definition of such behavioural states, depends on the specific parasite channel that is considered, which is elaborated on below.

$$b_{par} = \lfloor \log_2(s) \rfloor \tag{3.2}$$

In **ID2**, the amount of collisions caused on the CAN bus at hand defines a distinct behavioural state. As that parameter c can be varied across different instances of the **ID2** channel, it is a variable its bandwidth formula. On the contrary, in **E2** only 2 distinct states are possible, i.e., the CRC check either passing or failing, which means no dependency other than on m_{host} is exhibited by its bandwidth formula.

Concerning **T1**, the distinct behavioural states defining the covert payload transmitted, correspond to different orderings of the packets under control of a sending party. Suppose i depicts the amount of distinct message IDs controlled by that transmitter. Then at most i different message IDs can precede any CAN packet, yielding as many distinct behavioural states that can be constituted by the parasite channel per host channel message.

T2 presents some more complexity in the identification of its distinct behavioural states. By definition, different amounts of deviation from the regular time interval T between messages, constitute different behavioural states. Both the sender and the receiver are limited by the clock granularity of their software implementing parasite channel logic when respectively creating and identifying distinct behavioural states in this channel. Moreover, the clock frequency of their CAN bus limits the granularity of message timings as well. That is why the total amount of time $2T$ possible to be deviated from the host's message interval T is adjusted to a variable a_{eff} by dividing it by the Least Common Multiple (LCM) of the software clock granularity available and the involved CAN bus' speed. a_{eff} thus depicts the amount of software-distinguishable offsets on that CAN bus, of which according to Equation (3.2) the binary logarithm is to be taken to yield b_{par} in Equation (3.1).

As **T3** is an orthogonal composition of $T1$ and $T2$, construction of its bandwidth formula is simply the sum of theirs, which have already been elaborated on.

3.5 Channel Matrix

This section offers a synthesis of the previous three. The properties introduced in Section 3.2 are evaluated for each channel presented in Section 3.3, the results of which are brought together in the matrix structure of Table 3.2. The *Channel ID* column refers to the identifiers introduced in Section 3.3. The other columns refer to the properties listed in Table 3.1, and the rows correspond to the different hidden bandwidth channels proposed here. The bandwidth formulas listed in this matrix are constructed as described in Section 3.4

The structure of this overview allows for a direct comparison of different hidden bandwidth sources in CAN, and serves as a guide in constructing useful statements about those channels and their mutual relationships. To illustrate, the matrix reveals how only timing channels are of considerable interest for this work, as they form the only category that is both software-controllable and considered truly covert according to the definition of Section 3.1.1.

3.6 Conclusion

This chapter centered around hidden bandwidth sources in CAN, presenting several instances and evaluating them with respect to a set of communication channel properties, such as their bandwidth. It was stated how both software-controllability, and covertness of these channels are of most interest in the context of this work, from an implementation perspective and the research hypothesis considered respectively. Judging from the set of channels introduced, it is however to be concluded only few of them satisfy both properties. That does not render this discussion without value, as most of the channels proposed are software-controllable and thus still useful in extending the appropriate applications to exploit hidden bandwidth. Moreover, these considerations motivate the focus on timing channels in the remainder of this work.

Ch. ID	**Stack level**	**Real-time compliance**	**Application dependence**	**Bandwidth parameters**	**Bandwidth formula** ***(m=host channel message rate)***	**Covertness**
ID1	SW	Yes	Yes	hidden bits inserted N	Nm	No
ID2	HW	No	No	collisions allowed/message c	$\lfloor \log_2(c) \rfloor m$	Yes
D1	SW	Yes	Yes	hidden bits inserted N	Nm	No
D2	SW	No	Yes	max. # bits inserted N_i, max. # bits dropped N_d	$\lfloor \log_2(N_i + N_d) \rfloor m$	No
D3	SW	No	Yes	amount of bits padded N	Nm	No
E1	HW	No	No		$15m$	No
E2	HW	No	No		m	No
T1	SW	No	Yes	amount of ID's controlled i	$\lfloor \log_2(i) \rfloor m$	Yes
T2	SW	No	Yes	application message period T, SW clock granularity g, CAN bus clock frequency f	$\lfloor \log_2(a_{eff}) \rfloor m$ with $a_{eff} = \lfloor 2T/LCM(g, 1/f) \rfloor$	Yes
T3	SW	No	Yes	cf. **T1** and **T2**	$(\lfloor \log_2(i) \rfloor + \lfloor \log_2(a_{eff}) \rfloor)m$ with $a_{eff} = \lfloor 2T/LCM(g, 1/f) \rfloor$	Yes

Table 3.2: Matrix of the CAN channels described in Section 3.3, structured on the properties listed in Table 3.1, with **ID1**: Dedicating ID bits - **ID2**: Manipulating arbitration collision frequency - **D1**: Dedicating least significant bits - **D2**: Manipulating packet size - **D3**: Data field padding - **E1**: Dedicating the CRC field - **E2**: Pass/fail of error detection check - **T1**: Packet reordering - **T2**: Manipulating packet inter-arrival times - **T3**: Combination of **T1** and **T2**

Chapter 4

Java Implementation of an Inter-Arrival Time Channel

To illustrate the potential of CAN packet timings as a means for covert communication, this chapter presents a transmission channel based on modification of packet inter-arrival times, and its practical implementation in Java, which is publicly available at https://github.com/Stienvdh/Java-IAT. It provides with basic error detection and is instantiated to transmit some fixed payload from one sender to one receiver. Its core concepts are hereafter discussed first, followed by a performance assessment.

4.1 Channel Design

This Inter-Arrival Time (IAT) channel, similar to the IAT based covert channel proposed in TACAN [58], relies on a sender encoding covert data to packet Inter-Transmission Times (ITT) and a receiver monitoring and decoding message inter-arrival times. Those parties are presumed to employ periodic CAN communication, such that information can be carried in deviation from their fixed message period. Here, this IAT channel is used for transmission of some hard-coded covert message, extended with extra data for error detection purposes. However, its design allows for it to be generally applicable as a covert communication channel for CAN networks. This section gives an overview of its constituting mechanisms, and how those are relevant for the channel's performance.

4.1.1 Channel Robustness

Silence bits. Analogous to TACAN's IAT channel [58], the concept of *silence bits* is used in this IAT channel. Those precede and succeed every covert message to denote its beginning and ending respectively. To illustrate, suppose s_{start} silence bits are sent at the start of a covert message and s_{end} at its end. The amount of covert bits n_{total} to be transmitted in communicating one covert message of length n_m then can be calculated as follows, with n_e the amount of bits used for error detection:

$$n_{total} = s_{start} + n_m + n_e + s_{end} \tag{4.1}$$

This equation illustrates how enlarging the amount of silence bits used to delimit covert messages, lowers the useful bandwidth of this IAT channel. Indeed, more non-covert traffic has to be dedicated to cover an IAT message when more silence bits are required for delimiting it. However, silence bits serve another purpose of being a reference level for packet inter-arrival times, as illustrated in Section 4.1.2. Their encoding corresponds to an inter-arrival time equal to the underlying channel's message period, which allows a receiving party to adjust to small runtime effects on CAN message timings. From that perspective, increasing s_{start} and s_{end} offers this IAT channel higher reliability and robustness to noise.

Running average. As packet inter-arrival times are affected by the unpredictable noise properties of the CAN bus they are measured on, this IAT channel should account for such variations. As proposed by TACAN [58], it provides with a running-average mechanism for that purpose. A parameter L is introduced, which specifies the size of the window over which a running average of IAT values is maintained and sampled by the receiver. That approach means the sender must adjust its inter-transmission times accordingly, i.e., repeat them L consecutive times.

Similar to the remark made on the amount of silence bits used, increasing L lowers the useful bandwidth of this IAT channel, yet contributes to its reliability. When deploying this channel on a heavily loaded CAN bus, a higher value for L thus is appropriate, since more variations on IAT values are to be expected. In contrast, a clear bus justifies lowering L, due to fewer noise effects to be accounted for.

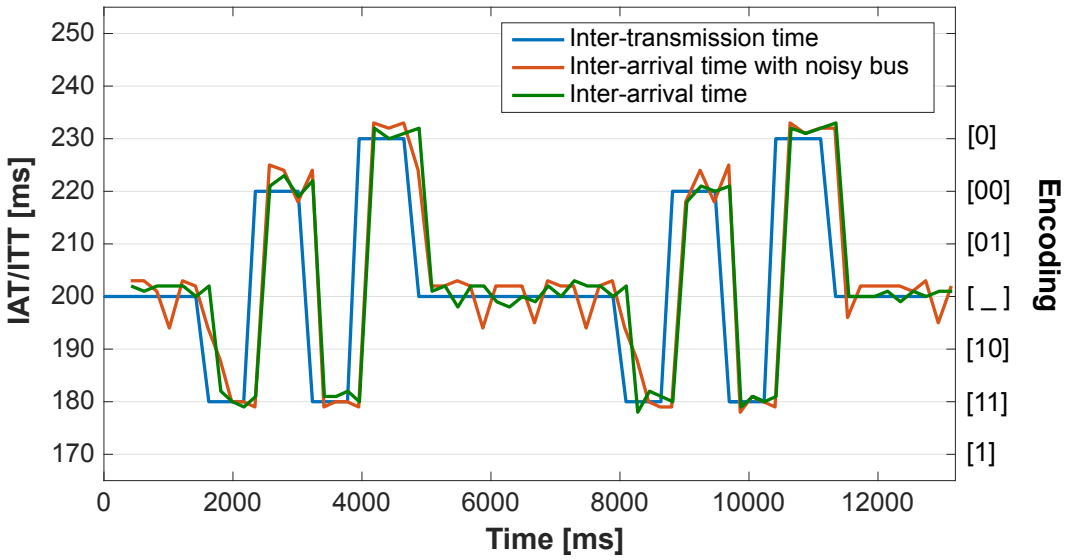

Figure 4.1: Inter-transmission and -arrival time trace for two consecutive transmissions of a message `11001` over a Java IAT channel implementation using $n_{start} = n_{end} = 2$ silence bits, 2 error detection bits, the encoding of Equation (4.2) and the example parameter values of Table 4.1, on both a clear and noisy CAN bus

4.1.2 Data Encoding and Decoding

Encoding. The encoding of (silence) bits b_{send} in this IAT channel is done by making a sender modify L consecutive inter-transmission times t, following for example Equation (4.2), with T the message period of underlying CAN traffic and δ a deviation parameter that is IAT channel specific.

$$t = \begin{cases} T + 3\delta & if\ b_{send} = 0 \\ T + 2\delta & if\ b_{send} = 00 \\ T + \delta & if\ b_{send} = 01 \\ T - \delta & if\ b_{send} = 10 \\ T - 2\delta & if\ b_{send} = 11 \\ T - 3\delta & if\ b_{send} = 1 \\ T & otherwise \end{cases} \tag{4.2}$$

In the implementation of this channel, the encoding as described in Equation (4.2) is provided, as well as an encoding that transmits just one covert bit per ITT. Evidently, care should be taken in selecting an appropriate encoding, as deviating from an application-defined value for T in inter-transmission times means real-time deadlines might be missed by that application. Also, for some application-defined maximum deviation from T, a trade-off should be made. Using a small δ and thus enabling many levels in encoding, creates a larger theoretical bandwidth of this IAT channel. In contrast, a larger δ, and thus encoding less covert bits per ITT value, makes this channel more robust against variations on IAT values as introduced by the CAN bus it is deployed on. More generally, the fact that inter-arrival times are not exactly equal to their corresponding inter-transmission times because of CAN bus properties, should be taken into account when designing an appropriate encoding.

Decoding. IAT value decoding to a covert bit sequence $b_{receive}$ is done at receiver side, by sampling its running average of IAT values on receiving every L'th message, with L the size of its running average window (cf. Section 4.1.1). That sample IAT value t_{sample} is converted to (silence) bit(s) $b_{receive}$ by calculating the ITT value used for encoding closest to it, and looking up its corresponding encoded bits. This is where CAN bus noise might cause decoded bits not to correspond to those encoded by the sender, as it can affect an IAT value such that it is closer to a different encoding ITT value than actually used by that sender.

Figure 4.1 shows a trace of ITT- and IAT values measured in two consecutive transmissions of a message `11001`, using two start- and end-silence bits, and two error detection bits (cf. Section 4.1.3). The IAT encoding used corresponds to Equation (4.2), with $T = 200ms$ and $\delta = 10ms$. IAT values are shown for both a noisy (50% bus load) and clear CAN bus configuration. The figure uses _ to denote silence bits. Please note the differences between ITT- and corresponding IAT-values, which illustrate how even without noise on a CAN bus, the measures taken into account here for channel robustness are in no way redundant.

4.1.3 Error Detection: Cyclic Redundancy Code

A Cyclic Redundancy Code (CRC) is a rather simple error-detection code based on polynomial division [4]. More specifically, along with a message, its remainder after division by some pre-defined polynomial of degree N is sent. A receiver can then detect bit transmission errors by executing the same division and checking its remainder to be equal to the transmitted CRC. When N equals 1, this CRC algorithm is reduced to adding a parity bit to messages.

In the implementation of this IAT channel, an error detection interface is provided, that can be instantiated by an error detection mechanism possibly different from CRC, which offers modularity. A CRC implementation of that interface is provided, and its parameter N, as well as the polynomial used for division, can easily be modified. Care should however be taken in ensuring that polynomial is of degree N.

4.1.4 Application: Pre-defined Payload Communication

Similar to how error detection is done in this channel's implementation, the covert messages to be exchanged over this IAT channel are interacted with over an interface. For illustrative purposes, a protocol that simply repeats a hard-coded payload is provided. The modularity of this design however allows for any type of a more sophisticated communication protocol to be used. The combination of this implementation's design for data encoding/decoding, as well as its error detection mechanism, thus constitutes a general, covert CAN communication channel prototype for applications with periodic message exchange.

4.2 Implementation Technology

In interacting with CAN, this implementation relies on USBtin [11]. This USB-to-CAN interface can, amongst other options, be used from Java software via a library[1], or through a graphical user interface[2]. This Java implementation depends on the former for sending and receiving CAN messages, but the IAT channel-specific operations of adjusting inter-transmission times and monitoring inter-arrival times do not require any of its functionality. That reveals how the implementation discussed here is reasonably flexible in using different CAN-interfaces, because its core functionality does not depend on their details.

Concerning CAN hardware, the minimal requirement for deploying this implementation is a set of two connected USBtin nodes; three if a noisy bus is to be simulated. One takes on the role of a sender, the other becomes a receiver. An optional third instance can be used to generate random traffic on their CAN bus, in order to introduce noise into this IAT channel. The effects of such noise are further discussed in Section 4.3.

[1] https://github.com/EmbedME/USBtinLib
[2] https://github.com/EmbedME/USBtinViewer

4.3 Channel Assessment

This section first qualitatively discusses the parameters affecting this IAT channel's bandwidth and reliability, and then presents a quantitative evaluation of its Java implementation, which is available at https://github.com/Stienvdh/Java-IAT.

4.3.1 Performance Parameters

As this IAT channel relies on modifying inter-arrival times of packets periodically generated by some application, its performance as well as other properties are strongly influenced by that application. For example, an application that sends messages over its CAN network every 10ms could allow for an IAT channel of higher bandwidth than an application sending messages every 100ms, as IAT values are generated more often. Secondly, this IAT channel by definition has several configurable variables, such as the δ offset already introduced in Equation (4.2). Finally, the properties of the CAN bus on which this IAT channel is deployed have a great influence on the timings of CAN messages transmitted over it, and thus on the packet inter-arrival times constituting this channel. The noisier a bus, the less reliable this channel becomes, as IAT values deviate under influence of noise frames occupying their CAN bus during transmission.

In Table 4.1, the parameters taken into consideration for an assessment of this IAT channel's implementation are listed. An example value for each parameter is given for illustrative purposes, as well as a qualitative description of the relation between each parameter and this IAT channel's bandwidth and reliability. Please note that these qualitative judgements only hold when all non-corresponding parameters are kept constant, while the corresponding parameter's value is raised, and that this IAT channel has interesting properties other than its bandwidth and reliability, such as its interaction with real-time deadlines.

Parameter	**Example value**	**Influence on IAT channel bandwidth/reliability if larger**
Application dependent		
Message period	200ms	Negative/Positive
Clock granularity in software	1ms	Negative/Negative
IAT channel dependent		
δ offset	10ms	Neutral/Positive
Running average window	4	Negative/Positive
Maximum amount of δ offsets	3	Positive/Negative
CAN bus dependent		
Bus speed	10kbaud	Positive/Positive
Bus load	50%	Neutral/Negative

Table 4.1: Parameters influencing IAT channel performance in CAN

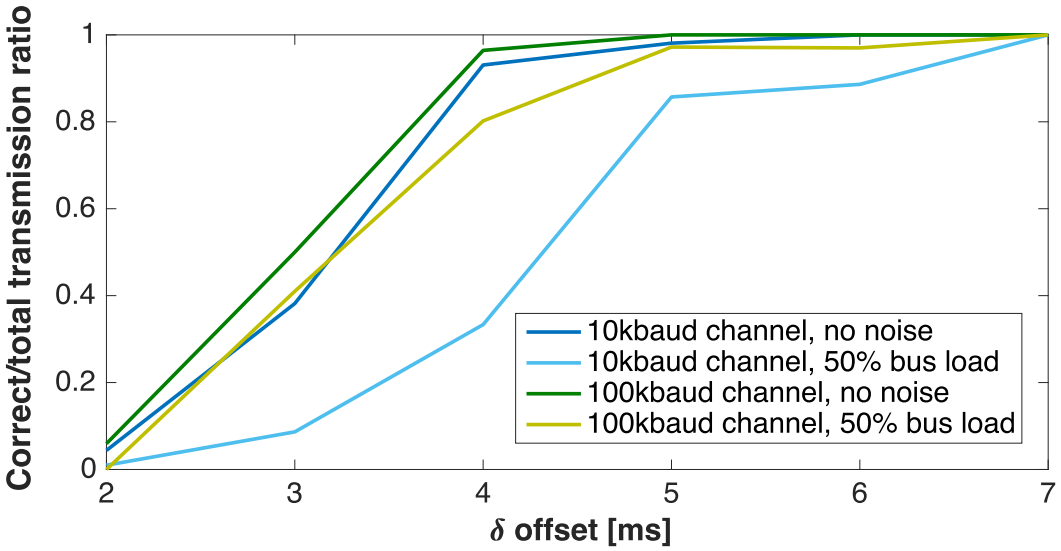

Figure 4.2: Ratio of correctly transmitted messages versus total amount of messages transmitted over a Java IAT channel implementation, when varying δ for different bus speeds and bus loads and using the example parameter values of Table 4.1

4.3.2 Channel Reliability

Until now, the effects of varying the performance parameters of Section 4.3.1 were only discussed qualitatively. Figure 4.2 quantifies those statements by showing the correctly transmitted fraction of the total amount of covert messages sent over this IAT channel, with different values for some of those performance parameters, namely bus speed, bus load and δ offset. These results were obtained by making a sending USBtin instance transmit the covert message `11001` in the same context as illustrated in Figure 4.1, 1000 subsequent times over this IAT channel implementation. A receiving USBtin instance monitors the amount of messages received that pass error detection (Section 4.1.3) as well as the hard-coded communication protocol (Section 4.1.4) correctly, i.e., the received message is checked to match the one originally transmitted. A third USBtin-instance is used for generating a 50% bus load of random traffic.

Figure 4.2 confirms the qualitative statements made about the influence of bus speed, bus load and δ offset on the reliability of this IAT channel. More specifically, a higher bus speed, a lower bus load and a higher δ offset all contribute to a higher channel reliability. Furthermore, some more specific observations can be made. For instance, the bus speed of a clear bus does not affect IAT channel reliability significantly. For both a 10kbaud bus speed and a 100kbaud bus speed, the channel's reliability drops below a reasonable 90% when δ falls below 4ms. The effect of bus noise is more obvious, and additionally dependent on bus speed. More specifically, channel reliability drops below 90% at $\delta = 6ms$ on a noisy 10kbaud bus, and at $\delta = 4ms$ on a noisy 100kbaud bus.

4.4 Conclusion

This chapter presented a Java implementation of a covert inter-arrival time channel that was proposed by TACAN [58]. Relying on a sender adjusting its packet inter-transmission times, and a receiver monitoring inter-arrival times, it provides a periodic CAN communication channel with a way to exploit extra bandwidth. This implementation is designed to be modular concerning error detection, the covert communication protocol used and data encoding/decoding. Therefore, it can serve as a generally applicable covert transmission channel, when built upon a non-covert message stream that is periodic. The reliability of the implemented channel was quantitatively evaluated to be negatively influenced by, amongst others, the load on the CAN bus it is deployed on, lowering the speed of that bus, and lowering the amount of variation in packet inter-arrival times used to encode covert data.

Chapter 5

Low-Level Implementation of an Inter-Arrival Time Channel

As to improve IAT channel performance compared to the results achieved using Java, this chapter migrates its logic to low-level technology. It first introduces the key mechanisms of this alternative implementation, and motivates the most important design choices made. As such, a framework is constructed in which to interpret the channel performance results that are presented throughout.

5.1 Channel Design

Using an approach similar to the Java implementation presented in Chapter 4, this low-level IAT channel transmits some covert payload by adjusting the originally equal-sized time intervals between non-covert, periodic CAN messages. At sender side, packet inter-transmission times are modified accordingly, and receiver-side logic monitors message inter-arrival times before decoding them to their corresponding covert payload. This section discusses the core concepts constituting this channel and its implementation in more detail.

5.1.1 Data Encoding and Decoding

The encoding and decoding between a covert payload and packet inter-arrival times used here assumes an underlying non-covert communication channel that transmits a CAN message at the end of every time interval of length T. This IAT channel itself defines an offset δ, which denotes the amount of deviation from T in packet inter-arrival times is used for encoding covert data. This approach is very similar to the IAT channel proposed by TACAN [58].

Encoding. The sender-side encoding of a covert bit b_{send} to an appropriate inter-transmission time t_{ITT} here is done using a maximum deviation from T of one δ offset, and the encoding depicted in Equation (5.1). Note that, similar to the encoding used in the Java prototype of Chapter 4, a maximum deviation of multiple δ offsets could

be used as well. Such an approach allows for more than one covert bit to be encoded in every inter-transmission time, as a trade-off for larger real-time effects.

The actual establishment of an inter-transmission time at sender side is done by setting a timer interrupt to fire after t_{ITT}, in whose Interrupt Service Routine (ISR) the next CAN packet of this channel's underlying application is sent.

$$t_{ITT} = \begin{cases} T + \delta & if\ b_{send} = 1 \\ T - \delta & if\ b_{send} = 0 \\ T & otherwise \end{cases} \tag{5.1}$$

Decoding. The receiver-side decoding of a registered packet inter-arrival time t_{IAT} to a corresponding covert payload $b_{receive}$, here is implemented to follow the decoding of Equation (5.2), which matches the encoding of Equation (5.1).

The registering of IAT values in this implementation is done through blocking execution until a CAN message arrives, stop a timer when it does, then register that timer's value, and finally restart it from zero.

$$b_{receive} = \begin{cases} 1 & if\ \ t_{IAT} > T + \delta/2 \\ 0 & if\ \ t_{IAT} < T - \delta/2 \\ \_ & otherwise \end{cases} \tag{5.2}$$

5.1.2 Limited IAT Accuracy Due to Receiver-Side CAN Buffer Polling

By design, the CAN driver[1] software used in this implementation uses a polling mechanism in message receiving. More specifically, when instructed to receive a message by IAT channel receiver-side logic using `can_recv`, it actively checks CAN controller receive buffers for incoming packets, and continuously repeats that check, or *polls* those buffers, until it detects incoming traffic, upon which `can_recv` returns and the receiver involved can register the corresponding packet inter-arrival time.

Consequently, all messages arriving at a CAN controller during the time interval between two subsequent buffer pollings appear to the IAT channel receiver to have arrived at the same time. Indeed, as this CAN driver software only processes received frames on buffer pollings, arrival time registration can only be done at those same moments. In that context, any registered arrival time is not precise, but representing some more accurate timestamp within one polling interval. As such, this implementation is limited by an **IAT measurement granularity of one polling interval**, i.e., arrival times cannot be measured at cycle-accurate precision.

On the MSP430 microcontrollers [46] used for this implementation, timing experiments demonstrate a polling interval length of 343 cycles, which at a 20MHz clocking frequency corresponds to 17.5μs.

Naturally, this receiver-side IAT granularity has its repercussions on IAT channel reliability. In essence, a receiver is not able to accurately monitor packet inter-arrival

[1] https://github.com/sancus-pma/vulcan/blob/master/drivers/mcp2515.c

times that are not a multiple of the time interval between its receive buffer pollings. Therefore, senders must produce only packet inter-transmission times that are a multiple of that interval, as to not undermine the theoretical possibility - that is, when without any CAN bus noise or other components affecting IAT channel performance - of perfect IAT channel reliability. Such sender-side modifications are thus required, and implemented here, to accommodate the aforementioned receiver-side IAT measurement granularity, while maximizing channel reliability.

5.1.3 Accounting for Computational Latency

Since the timer mechanisms used in this implementation offer cycle-accuracy in their measurements and interrupts, it is sensible to exploit that precision in handling ITT/IAT values for those not to be affected by the computational latency incurred while producing them. This subsection discusses how this IAT channel implementation accounts for such latency in the logic of both senders and receivers. Please note how the modifications made here are dependent on this particular implementation of this particular IAT channel, which causes the reasoning behind them to be of higher value than their technical details.

ITT correction (sender side). The main source of computational latency in the timely transmission of CAN messages is timer management, which pertains to the termination of the timer set when sending a previous CAN message, the calculation of a desired ITT value, and the initialisation of a timer that will interrupt after that ITT interval, upon which that sequence repeats (see also Section 5.1.1). Those computations delay the start of the timer that is to interrupt when a next message is to be sent, which means that timer should be set for an adequately smaller interval than the packet inter-transmission time intended to be established.

Table 5.1 illustrates this mechanism by listing the computation steps taken by an IAT channel sender in transmitting two subsequent CAN messages. It indicates how the ITT perceived on the CAN bus involved consists not only of the timer interval set by that sender, but also some timer management computations. As that perceived ITT needs to be accurate for maximal IAT channel reliability, that timer interval should be adjusted accordingly.

Cycle-accurate timing experiments on MSP430 microcontrollers depict a 20-cycle latency induced by these timer management computations, which corresponds to $1\mu s$ at a 20MHz clocking frequency. Consequently, this implementation subtracts 20 cycles from the desired ITT value before setting a timer that interrupts after the resulting time interval. As such, this implementation is limited to ITT values higher than 20 cycles, but since only ITT values that are a multiple of 343 cycles are used (see Section 5.1.2), that is no hard limitation.

Send CAN message	Idle	Timer interrupt	Stop timer	Calculate ITT	Start timer	Send CAN message	...
Timer interval		**Timer management latency**				...	
ITT perceived on CAN bus						...	

Table 5.1: Sender-side computations done in a low-level implementation of an inter-arrival time channel in between two CAN message transmissions, with indication of the resulting inter-transmission time perceived on the CAN bus at hand.

IAT decoding	Blocking CAN receive	Stop timer	Store timing	Start timer	IAT decoding	...
Timer result		**Timer management latency**			...	
IAT perceived on CAN bus					...	

Table 5.2: Receiver-side computations done in a low-level implementation of an inter-arrival time channel in between two CAN message arrivals, with indication of the inter-arrival time perceived on the CAN bus at hand.

IAT correction (receiver side). Similar to the way timer management introduces computational latency in the ITT values perceived on a CAN bus, IAT values at receiver side are affected as well. More specifically, the stopping of the timer started when receiving a previous CAN message, the reading and storing of its value, and its restarting cause part of each physically perceived packet inter-arrival time not to be accounted for by its corresponding timer result.

Table 5.2 illustrates the different computations done by an IAT channel receiver between two CAN message arrivals, and how those constitute both a timer result and timer management latency. As the combination of those two factors results in the actual packet inter-arrival time, timer management latency is to be added to the timer result before IAT value decoding.

On MSP430 hardware, receiver-side timer management latency was measured to be 126 cycles, which corresponds to 6.3μs at a clocking frequency of 20MHz.

5.2 Implementation Technology

The implementation presented in this chapter was done using two Sancus-enabled MSP430-microcontrollers [46], both connected over a SPI-interface to their own MCP2515 CAN controller [25] for CAN communication. One implements receiver-side functionality, the other takes on the role of a sender. Both connect to the same CAN bus via their respective CAN controllers. When introducing CAN bus noise for the performance measurements done in Section 5.3, the same USBtin hardware as used for the Java IAT channel prototype presented in Chapter 4 is leveraged for this setup, and its software for generating noisy background traffic likewise.

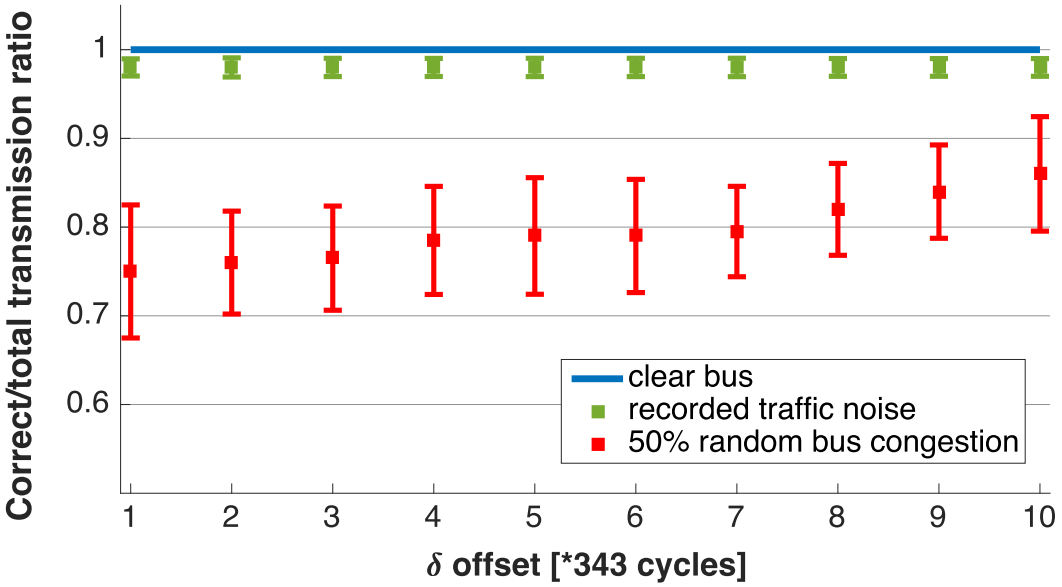

Figure 5.1: Fraction of transmitted payloads correctly received for varying values of δ in a low-level IAT channel implementation on MSP430 microcontrollers, in three different bus configurations (clear bus, pre-recorded background traffic, randomised 50% bus load)

5.3 Channel Assessment

Experimental setup. In order to assess the reliability of the IAT channel constituted by the low-level implementation presented in this chapter, a sender is configured to adjust its periodic communication (period = 15.000 cycles) to transmit the same covert payload over this IAT channel 100 times. An IAT-measuring receiver then decodes the resulting inter-arrival times and for this experiment is extended to compare its results to the expected covert payload, which produces a measure for the reliability of the IAT channel presented here. This scenario is repeated 10.000 times, gathering as many reliability measurement samples. Data encoding and IAT decoding is done as depicted in respectively Equation (5.1) and Equation (5.2).

That full transmission sequence is executed while varying the value for δ as used in Equation (5.1) and Equation (5.2) from 1 to 10 polling intervals, as prescribed by Section 5.1.2. Each configuration is measured in the context of three different CAN bus conditions; a clear bus without any other traffic than the IAT channel itself, a bus with 50% random noise congestion and a bus simulating CAN traffic as recorded on a real-life vehicle. Those different sources of noise are introduced by attaching a USBtin-node to the CAN bus used by the IAT-sender and -receiver, as mentioned in Section 5.2. In all scenarios, the CAN bus used works at a baud rate of 50kHz.

Figure 5.1 shows the measurement results as obtained through this experimental setup. Please note the δ offset being expressed in multiples of 343 cycles, which corresponds to one polling interval (cf. Section 5.1.2).

Results in clear bus conditions. As depicted in Figure 5.1, this low-level prototype performs with 100% reliability in clear bus conditions on all δ values that are a non-zero multiple of one polling interval, which corresponds to a minimal offset of 17.5μs at a clocking frequency of 20MHz. In comparison, such reliability is only attained at a δ of at least 6ms in a Java prototype of this channel (cf. Section 4.3).

From these results, and taking into account the decoding scheme used (cf. Equation (5.2)), can be concluded that the used CAN bus' transmission timings are deterministic up to at most half a polling interval, i.e., the timing intervals needed for transmitting the same message twice over that same CAN bus differ at most half the length of a polling interval.

Results with 50% random bus congestion. Figure 5.1 shows how reasonably lower channel reliability is attained on a CAN bus with randomized background noise, compared to clear bus conditions. An average reliability of at least 75% was measured when using δ offsets that are a non-zero multiple of one polling interval. Naturally, enlarging the δ offset used benefits IAT channel performance, resulting in an average reliability of 85% when using a δ offset of 10 polling intervals.

A similar reliability in the Java implementation of this IAT channel in noisy bus conditions was obtained at a δ offset of about 4-5ms (cf. Figure 4.2), compared to this implementation's 17μs offset. Whether these results correspond to a reasonable degree of channel reliability, evidently depends on application context, but the performance advantage over the Java prototype is obvious, and moreover comparable to the relative improvement in clear bus conditions that was discussed before.

Results with real-life background traffic. As is apparent from Figure 5.1, results in CAN bus conditions of pre-recorded real-life background noise, are roughly equivalent to the reliability measured in clear bus conditions. On average, only 1% of covert payloads transmitted are received incorrectly in this bus configuration.

These results motivate the application of this implementation in real-life scenarios in terms of its reliability, as 99% of covert information being able to be transmitted successfully is a reasonable degree of message loss in many applications. However, this channel's inherent consequences for real-time behaviour of its underlying communication channel encourage an effort to lower the minimal δ offset needed for such IAT channel reliability even further, which is made in Section 5.4.

5.4 Interrupt Driven Implementation

A major limitation of the low-level IAT channel implementation discussed so far, stems from its polling based approach to CAN message receiving, that causes the need for a software-defined inter-arrival time value granularity of one polling interval (cf. Section 5.1.2). This section explores an alternative, interrupt driven implementation that aims for cycle-accuracy in IAT values, thus enabling an equal or higher IAT channel bandwidth at lower variation in packet timings. This implementation is publicly available at `https://github.com/Stienvdh/Sancus-IAT`.

5.4.1 Channel Design

This interrupt driven alternative does not differ from its polling based counterpart in its core mechanisms. Indeed, encoding and decoding of covert data is done through the same process of a sender adjusting its inter-transmission times and a receiver monitoring and decoding packet inter-arrival times, following the schemes in Equation (5.1) and Equation (5.2). This approach however lifts the constraint of inter-transmission and -arrival times needing to be a multiple of one polling interval, through redesigning the way inter-arrival times are monitored at receiver side.

Interrupt Driven IAT Value Retrieval

Figure 5.2 gives a graphical representation of this new approach to collecting IAT values. It shows how at every CAN message arrival, an interrupt is fired at the receiver, upon which a software-defined interrupt service routine registers the time elapsed since the arrival of the previous message, and stores it in a circular buffer that is accessible to the receiver's software for decoding and/or further processing. IAT registration is thus initiated at the exact moment of message arrival, instead of at the end of a some subsequent polling interval, which obviously yields great improvement in accuracy.

This interrupt driven approach has the added advantage of making IAT value registration fully transparent to the channel's receiver-side logic. Indeed, its implementation involves the mere addition of a suitable ISR and a circular buffer to the receiver's software in order to support its dependence on IAT values for covert communication. In contrast, the polling based implementation discussed before involved emulation of similar registration logic in the receiver's IAT channel logic.

Beyond these mechanisms, this implementation and its previously described alternative do not differ in the processing of CAN messages and their inter-arrival times, subsequent to the latter having been registered. With the logic constituting this alternative IAT channel, which entails for example IAT encoding and decoding, thus not differing from its polling based counterpart, the measures described in Section 5.1.3 to account for computational latency at the software level can safely be migrated to this implementation. Receiver-side adjustments however now are applied in an interrupt service routine, rather than in the main application execution path.

Message ID Masking and Filtering

Problem: IAT value confusion. Since this implementation was designed to register message inter-arrival times whenever *any* packet arrives at receiver side, and CAN networks are based on broadcast communication, the complication could arise of messages, that do not belong to this IAT channel's host application, causing arrival interrupts and therefore unintended IAT value registration. A mechanism thus needs to be put in place to ensure only inter-arrival timings of messages of this IAT channel's underlying application are measured and stored in the circular buffer supplying those IAT values to a receiver's application logic.

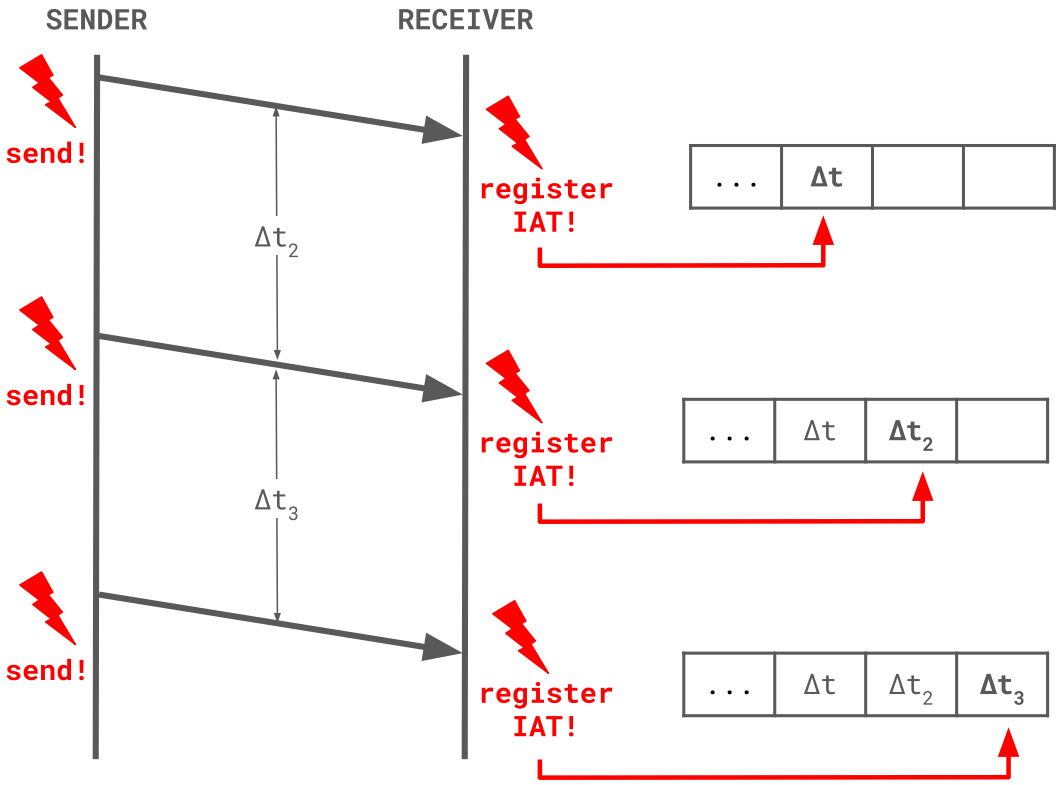

Figure 5.2: Graphical representation of an interrupt driven IAT channel. Interrupts at receiver side enable IAT value registering in a software-accessible buffer, and timer interrupts at sender side enable proper inter-transmission times.

Solution 1: extending ISR logic. A naive approach to preventing such IAT value confusion, is the checking of a CAN packet's ID to belong to the underlying application before registering its timing. However, CAN packet inspection typically involves a considerable amount of computational latency, which is less than desirable to occur in ISR logic. IAT registration, and therefore a potential message ID check, in this approach however is done in an ISR, which discourages the use of this first solution, although it is effective in mitigating IAT value confusion.

Solution 2a: ID masking and filtering. The MCP2515 CAN controllers used for this implementation (cf. Section 5.4.2) offer the possibility of controlling which packets on their CAN bus are able to enter their receive buffers, and thus cause an interrupt to fire, based on their message ID. More concretely, an incoming message's ID is first filtered, then masked before it is allowed to enter such a buffer. These operations are implemented at the hardware level, which means they incur little latency in IAT registration. Moreover, this mechanism allows for a considerably small ISR to be executed on CAN message arrival, as elaborated on in Appendix A. Indeed, the ISR used in this implementation counts only 5 lines of C code.

ID bit	Filter bit	Mask bit	Result
-	-	0	**0**
1	1	1	**0**
0	1	1	**1**
1	0	1	**1**
0	0	1	**0**

Table 5.3: Possible bit configurations in masking and filtering of CAN message IDs. Dashes denote both 0- and 1-bits (*don't care* bits).

Applying a *filter* to a message ID entails performing a bitwise XOR operation between the two, setting all the bits of the resulting bit sequence on whose corresponding positions the filter- and ID-bits are not equal, and zeroing all other bits. When a *mask* is then applied, a bitwise AND is executed of the filtered result and that mask, resulting in an all-zero bit sequence if and only if all the original ID-bits in the corresponding positions are either unmasked, or masked and equal to their corresponding filter bits.

Table 5.3 summarizes these operations, and their results for individual bits. All message ID bits of an incoming CAN packet must result in a 0-bit for the message at hand to be loaded into the receive buffer of the corresponding CAN controller, and therefore cause an interrupt in this implementation.

Solution 2b: multiple receive buffers. The seemingly limiting approach of restricting only the host application's messages to be accessible from the receiver's software, can be mitigated due to MCP2515 providing two separate receive buffers. Each can be configured independently concerning its mask and filters, as well as whether an interrupt fires when a message enters them. By thus enabling interrupts, masks and filters on only one of those buffers, this implementation can still function correctly while allowing other messages than accepted after masking/filtering to be available to the receiver's software, after being loaded from the second buffer.

As this implementation serves as a mere prototype for an IAT channel, there is no need for messages other than the underlying application's being loaded into CAN controller receive buffers. Hence both mask the full message ID, and filter it to match the host application messages' ID. Moreover, an interrupt is fired when a packet enters either buffer, as it is thus guaranteed to belong to the communication of the application upon which this IAT channel is built.

5.4.2 Implementation Technology

This interrupt driven channel leverages the same hardware (MSP430 microcontrollers [46] + MCP2515 CAN-controllers [25], connected through a SPI-interface) as its polling based alternative presented before. To however enable this interrupt driven approach, some modifications have been done to that hardware, to leverage the interrupt functionality provided in the MCP2515 CAN controllers. More specifically,

a line is added between the dedicated interrupt pin of each CAN controller, and a general purpose digital I/O pin on their respective MSP430 microcontrollers. The following configurations are moreover done to actually enable interrupt handling on CAN message arrival as prescribed by this channel's design:

- **Enabling CAN controller interrupts**: Setting the interrupt enable (IE) bits dedicated to the two receiving buffers on a CAN controller, causes the controller's interrupt pin to be driven low whenever a CAN message enters either buffer. All other CAN controller interrupt sources (e.g., transmission errors) are disabled, as only that event is of interest to this implementation.

  Moreover, the CAN controllers that were used only provide one dedicated interrupt pin, which leaves the functionality of identifying the CAN controller interrupt source to MSP430 software, should multiple types of events have the ability to drive the interrupt pin low. As to relieve MSP430 interrupt handlers from this extra logic, only CAN message arrival interrupts are enabled.

- **Enabling MSP430 I/O interrupts**: Through the extra attached line between each microcontroller and its CAN controller, a digital I/O pin on the microcontroller will be driven low whenever the CAN controller's interrupt pin is, which means whenever a CAN message arrives. Setting the appropriate (i.e., dedicated to the I/O pin attached to the CAN controller) IE bit on the microcontrollers allows for such interrupts to be propagated to, and subsequently handled by, their software.

- **Setting CAN controller masks and filters**: Due to CAN being a broadcast-based protocol, more messages than a microcontroller is actually interested in could arrive at its CAN controller, causing more interrupts to be fired than necessary, which in turn results in IAT value confusion. This issue, as well as its proposed solution of configuring receive buffer masks and filters, have already been introduced in Section 5.4.1.

5.4.3 Channel Assessment

Experimental setup. The scenario used to assess this interrupt driven channel is similar to the one used for its polling based counterpart (cf. Section 5.3). Some covert payload is transmitted 100 consecutive times over this IAT channel, in 10.000 repetitions. This sequence is executed using δ offset values varying from 150 to 350 MSP430 cycles, and in four different CAN bus configurations. Three of those (clear bus, 50% random bus congestion, pre-recorded traffic) correspond to the configurations discussed in Section 5.3, and a fourth of 75% random bus congestion is added in this specific assessment. That final bus configuration was added due to this interrupt driven channel exhibiting near-perfect performance in all other bus conditions, and is thus meant to demonstrate the limits of this approach.

Figure 5.3 shows the results obtained from measuring channel reliability in these configurations, by indicating their average fraction of messages being transmitted correctly, and the standard deviation on that number in the conducted experiment.

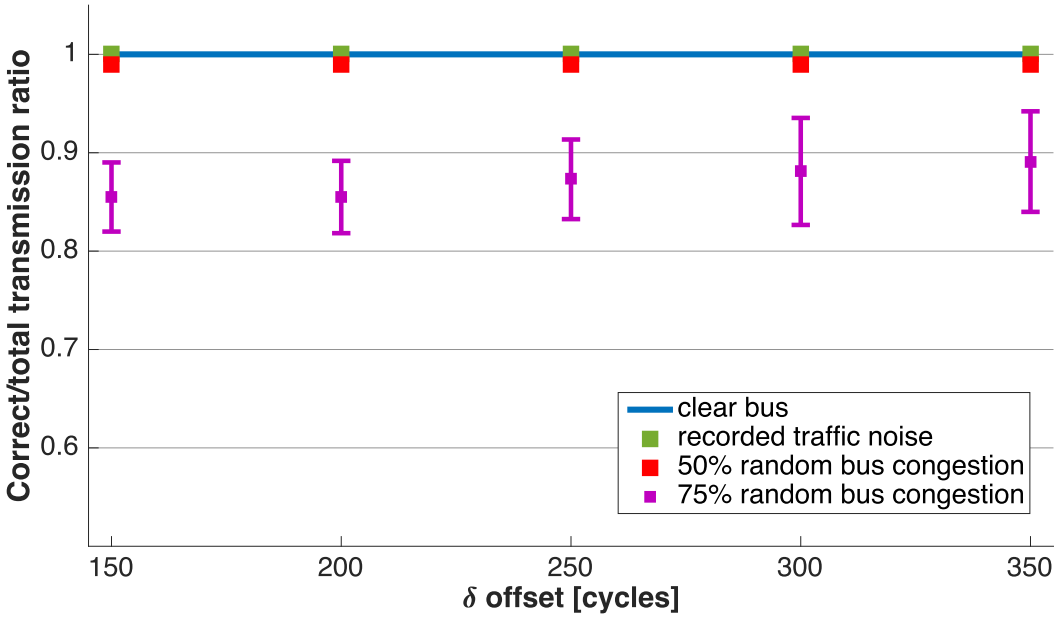

Figure 5.3: Fraction of covert payloads transmitted correctly for varying values of δ on an interrupt driven IAT channel implementation on MSP430 hardware, in 4 bus configurations (clear bus, 50% and 75% random bus congestion, pre-recorded background traffic)

Results on clear bus, 50% bus congestion, pre-recorded traffic. In all three bus configurations for which this implementation's polling based counterpart is assessed in Section 5.4.3, this channel matches and/or improves the polling based channel's performance to 100% reliability, using δ offsets lower than one polling interval. In comparison, the polling based channel offers 75% reliability in a 50% random bus congestion setting and a δ offset of one polling interval.

These results are enabled by interrupt driven IAT value registering (Section 5.4.1) on the one hand, which allows for IAT value accuracy below 343 cycles. On the other hand, low-level masking and filtering of incoming messages (Section 5.4.1) partially relieves from noise sensitivity through not loading noise frames into CAN controller receive buffers, instead of doing so and then discarding them, as was the case in the polling based implementation.

Results in 75% random bus congestion. When raising bus congestion to 75%, channel reliability drops to an average of 85 to 90 percent, when varying δ from 150 to 350 MSP430 cycles. Software-level noise sources for this IAT channel have been eliminated as much as deemed possible, so this performance degradation mainly stems from bus occupancy by noise traffic, which disables timely arrival of the non-covert traffic constituting this IAT channel. Other factors causing limitations to this channel's performance, are discussed in the next section.

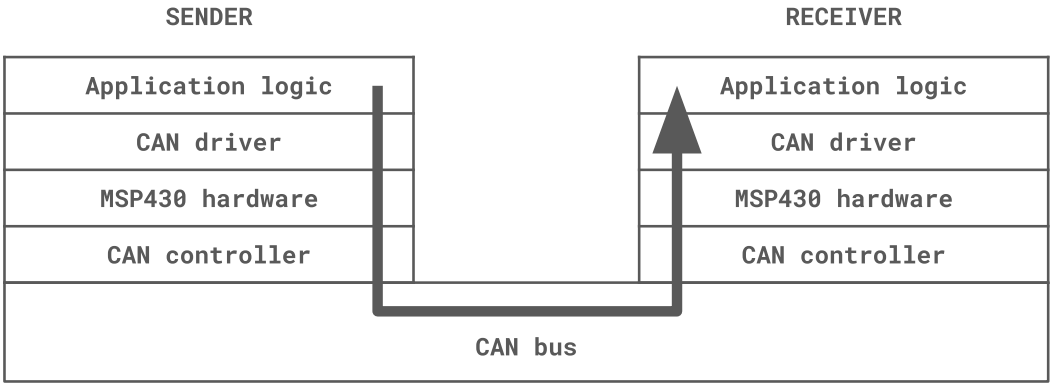

Figure 5.4: Graphical representation of the software/hardware stacks a CAN message traverses from sender to receiver in a low-level implementation of an inter-arrival time channel.

5.5 Sources of IAT Channel Noise

Throughout the three implementations (Chapter 4, Section 5.1, Section 5.4) of the IAT channel presented in this work, careful consideration of the sources of noise and channel unreliability has been done and proper mitigations were introduced where deemed possible and effective. This section categorizes those noise sources, and offers a selection of measures to prevent, detect and/or correct erroneous IAT channel performance. Figure 5.4 illustrates the stack traversed by CAN packets that constitute this IAT channel when implemented on MSP430 hardware, and thus guides this discussion.

5.5.1 Application Logic

This subsection discusses channel deficits originating at the software level, i.e., in the logic governing calculation and manipulation of inter-transmission times at sender side, as well as monitoring and decoding of inter-arrival times at receiver side. Naturally, this level is the most interesting in terms of noise mitigation, as it is the most convenient to be adjusted.

Programming language properties. The properties of the programming language used for implementing application logic obviously affect its timing properties. This is one of the main sources of unreliability in the Java implementation, as Java exhibits great non-determinism in its timing characteristics. Since the low-level implementations, in contrast, can rely on cycle-accurate timings and timer interrupts, that non-determinism is eliminated, and this noise source with it.

Computational latency. Execution of the logic constituting proper ITT/IAT values in itself influences their accuracy. For instance, when encoding a covert payload to an inter-transmission timing, part of it has already passed when calculation

finishes. Similar computational latency is introduced at receiver side, in managing (stopping/reading/starting) the timer that measures IAT values.

As the logic implemented for this channel does not include any randomized and/or non-deterministic operations, the computational latency introduced at both ends (sender and receiver) can be deemed deterministic and thus effectively accounted for in the application logic itself. Section 5.1.3 elaborates on this approach for the polling based implementation of this IAT channel, which is analogously applied to the interrupt based implementation of Section 5.4.

In the Java implementation, no measures are taken to mitigate computational latency effects. Although its logic consists of only deterministic operations, Java's inherent timing non-determinism, which was introduced earlier in this subsection, renders measuring and accounting for computational latency highly unreliable.

5.5.2 CAN Driver

CAN driver software handles all interactions between application logic and actual CAN communication through a CAN controller. Its properties obviously affect both the ITT and IAT values eventually measured on the CAN bus at hand, as all incoming and outgoing messages pass through it. Lower flexibility than at the application logic level however is available in driver software, as its purpose is to be ported across multiple applications, which might be affected negatively by changing its timing characteristics. In the low-level implementations presented in this chapter, a MCP2515 CAN controller driver[2] is used, and the Java implementation of Chapter 4 leverages a USBtin library[3] for this purpose.

Message propagation mechanism: transmission. All driver software used in these implementations exhibits determinism in its transmission timings, as messages are transmitted using only deterministic operations, and at the exact time the appropriate driver interface method is called. As such, inter-transmission timings are not affected by CAN driver software, since it introduces the same latency in transmitting each underlying application message.

Message propagation mechanism: receiving. The USBtin library used in the Java implementation is based on intercepting serial port interrupts for propagating incoming CAN messages to application logic, meaning no software-level non-determinism is introduced in receiving CAN messages. Registered inter-arrival times in that Java prototype thus are not affected by the USBtin CAN driver logic.

The MCP2515 driver however does exhibit non-determinism in its incoming CAN message timings. Due to its polling based approach introduced in Section 5.1.2, the latency introduced in propagating a packet is non-deterministic in the sense that it depends on that packet's timing within one polling interval, which is unpredictable.

[2] https://github.com/sancus-pma/vulcan/blob/master/drivers/mcp2515.c
[3] https://github.com/EmbedME/USBtinLib

As mentioned before, modifying CAN driver software is an infeasible approach, which is why this source of noise is mitigated at the application level. Initially, ITT and IAT value granularity artificially is adjusted to have the same granularity of one polling interval (cf. Section 5.1.2), which overcomes this non-determinism. The interrupt driven implementation of Section 5.4 relieves from that artificial construct by registering arrival times in an interrupt driven fashion (cf. Section 5.4.1), which retains only the deterministic latency introduced at the driver level.

5.5.3 MSP430 Hardware

The MSP430 microcontrollers used for the two low-level IAT channel implementations affect ITT/IAT values in that at sender side, its timer peripheral is used to interrupt when a message is to be transmitted, and at receiver side, that timer peripheral serves to measure inter-arrival times. In the interrupt driven implementation variant of Section 5.4, the used microcontroller's interrupt mechanism moreover influences the registering of inter-arrival time values (cf. Section 5.4.1).

Timer accuracy. As stated in MSP430 specifications [46], cycle accurate timings are available on MSP430 hardware through its timer peripheral. Timing interrupts, as well as timing measurements, thus do not introduce any noise on ITT, nor IAT values, as both are obtained using that peripheral in the low level implementations.

Interrupt handling mechanism. The propagation of interrupts from their source to application logic evidently incurs latency in message transmission, as well as in IAT value registering in the interrupt driven implementation. MSP430 hardware however ensures that this latency is deterministic [46], and thus accountable for at the application logic level.

5.5.4 CAN Controller

The MCP2515 CAN controllers used in the two low-level implementations, influence ITT/IAT values through propagation of CAN messages in both the incoming and outgoing direction. In the interrupt driven approach, these controllers moreover serve to propagate an interrupt on an incoming message to their MSP430 microcontroller.

Message and interrupt propagation. Similar to MSP430 hardware, all message and interrupt propagation of MCP2515 CAN controllers introduce deterministic, and thus relatively unharmful, latency [25].

SPI interface clocking frequency. The SPI interface connecting each CAN controller to its microcontroller, is clocked at 10MHz in the low-level implementations of this IAT channel. The latter however has a clocking frequency of 20MHz, which means ITT/IAT values have a granularity of at least 2 (MSP430) cycles. Concretely, two messages transmitted/arriving in the same two-cycle period, appear to the microcontroller as indistinguishable in terms of their timing.

MCP2515 clocking frequency. Each CAN controller itself is clocked at yet another frequency of 16MHz. Consequently, ITT/IAT values in software should artificially be constructed to be multiples of 4, in order not for them to be affected by clock frequency differences between a MSP430 microcontroller, its SPI interface and its MCP2515 controller.

5.5.5 CAN Bus

Obviously, the CAN bus over which all messages constituting this IAT channel are transmitted has major effects on their ITT and IAT values. It is the only stack level in Figure 5.4 where actors not constituting this IAT channel can affect its functioning, e.g., through occupying the bus when it is needed by this IAT channel. That externally controlled aspect renders CAN bus noise to be interesting for channel assessment purposes, as it is out of IAT channel implementation control.

Clocking frequency. Similar to the clocking frequency discrepancies introduced before, a CAN bus itself has a (configurable) clocking frequency of, e.g., 500kHz, which differs from MSP430 hardware. A software-level ITT/IAT value granularity of 40 (MSP430) cycles in that case needs to be established for this channel to accommodate all clocking frequency differences mentioned throughout this section (MSP430 microcontroller, MCP2515 CAN controller, SPI interface, CAN bus).

Background traffic. Throughout this work, IAT channel noise introduced by CAN background traffic, and its influence on channel performance, has been discussed extensively (see Section 4.3, Section 5.3, Section 5.4.3). As this noise stems from external actors, there is no IAT channel modification available to overcome it. Application logic however can be adjusted to partially mitigate the consequences of background traffic, e.g., by extending covert payloads with error detection and/or error correction bits, as included in the Java prototype and discussed in Section 4.1.1.

5.6 Conclusion

This chapter presented a migration of the Java implementation of an IAT channel (cf. Chapter 4) to a low-level variant, deployed on MSP430 microcontrollers interacting with MCP2515 CAN controllers. In its high-level mechanisms, this implementation is very similar to the Java implementation. However, as the technology used here provides more accurate timing mechanisms, those are leveraged to enable a lower granularity in ITT/IAT values, and thus a higher IAT channel reliability. The main software-level limitation of this implementation is the polling mechanism it uses in CAN message receiving, i.e., its continuous buffer polling prohibits cycle-accuracy in ITT/IAT values. In spite of that constraint, the resulting channel outperforms its Java counterpart on both a clear and noisy CAN bus configuration significantly.

In an effort to further improve performance results, an interrupt driven approach to covert IAT communication in CAN was discussed as well. In that implementation, whenever a message arrives at receiver side, an interrupt is fired in whose interrupt service routine that message's timing is registered, thus supplying near cycle-accurate timings to receiver software. Moreover, filtering and masking of message IDs is done, as to disable noise packets causing such interrupts and disturbing recorded timings. This enhanced channel thus offers great accuracy in registering the inter-arrival times that encode covert payloads, which enables more covert bandwidth at lower timing variations than used in previously presented implementations. An assessment of its performance confirms 100% reliability of this implementation in clear bus conditions, as well as with 50% random bus congestion, or with pre-recorded background traffic.

Outside those mitigated by these low-level implementations, several IAT channel noise sources originate from the different levels in the hardware/software stack traversed by CAN messages. An overview of those sources is discussed, revealing CAN bus background traffic as the main source of IAT channel noise.

Chapter 6

Nonce Synchronisation in VulCAN through Authentication Frame Timing

Given the promising results of interrupt driven IAT-exploitation on openMSP430-hardware, that mechanism can sensibly be applied to a more specific, practical context. This chapter motivates and discusses the timing between application- and corresponding authentication-traffic in VulCAN as a concrete source of such covert bandwidth and proposes its use for nonce synchronisation purposes, in an effort to alleviate performance- and security-related issues in current VulCAN design. From that discussion emerges the proof that covert transmission, despite its drawbacks, can considerably benefit existing security solutions, even when exposing only little supplementary bandwidth.

6.1 Problem Statement

The implementation of message freshness guarantees has proven to be a non-trivial challenge in authenticated communication [51, 37]. Most solutions are based on associating some freshness value, or *nonce*, to each message, whose uniqueness is a prerequisite in passing authentication. Although effective, such nonce mechanisms imply a difficult trade-off between bandwidth consumption and security guarantees. Indeed, the explicit transmission of a nonce for each message demands an increased traffic throughput, which encourages lowering nonce size. However, a larger nonce space allows for stronger guarantees of nonce uniqueness, and thus freshness, as there simply are more unique values available.

Many authentication protocols [36, 34] therefore resort to transmitting only partial nonce values explicitly, leaving their remainder implicitly stored and appropriately updated by the participants in authenticated communication. That approach benefits both bandwidth use and security properties, yet breaks when those implicit nonce segments come to be desynchronised amongst participants due to message loss, spoofing or component reset.

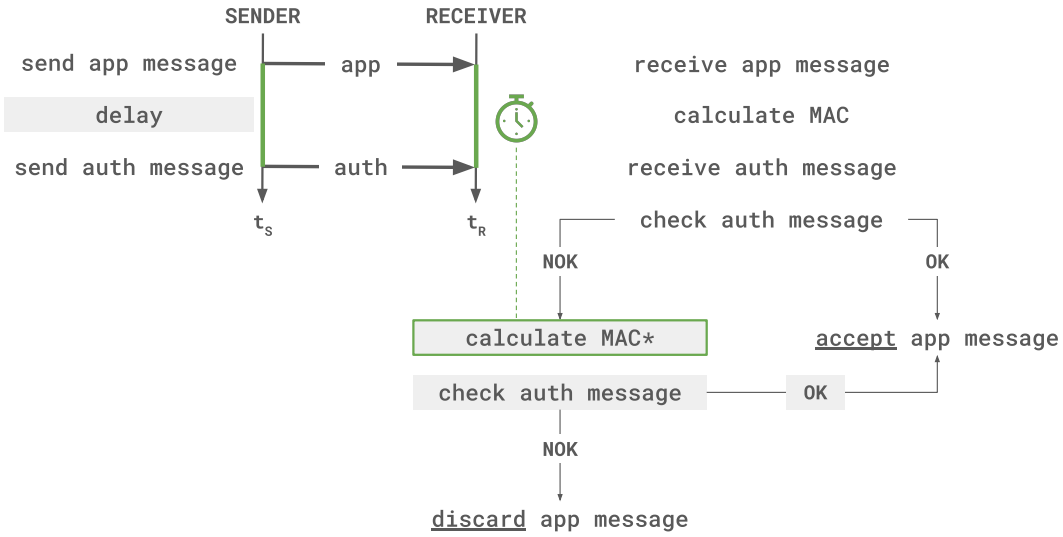

Figure 6.1: High-level overview of IAT based nonce synchronisation in VulCAN's vatiCAN backend, with differences from original VulCAN design indicated in grey

In VulCAN specifically, two alternative authentication protocols, namely LeiA [36] and vatiCAN [34], are implemented, both of which employ such partially implicit nonce values. Current VulCAN design has no measures put in place to recover from implicit nonces going out of sync, that do not involve a system-wide nonce broadcast, which in itself requires extra bandwidth use, as well as the discarding of valid traffic until such a broadcast occurs. vatiCAN is particularly fragile in that context, as it presumes fully implicit nonce values and therefore has zero resistance to nonce desynchronisation in the event of message loss. LeiA, in contrast, renders only the upper half of nonce values implicit, thus accommodating some degree of nonce desynchronisation. Section 2.2.5 discusses the VulCAN-specific challenges stemming from these nonce synchronisation mechanisms in more detail.

6.2 Overview of the Approach

In essence, this chapter proposes an extension to VulCAN's vatiCAN backend that, through exploiting covert bandwidth, aims to strengthen its nonce mechanism's ability to recover from message loss. This section gives a high-level overview of how exactly this approach works, by focusing on how it differs from current VulCAN design. Figure 6.1 serves as an illustrative guide to this scheme, and indicates where original VulCAN ends, and this extension begins. Several steps have been left out in both this textual and graphical overview to avoid cluttering, but those will be discussed in more detail throughout the rest of this chapter.

In original VulCAN design, transmission of a message from a sender to a receiver over a vatiCAN-authenticated connection progresses as follows. First, the sender transmits the message at hand, or *application message*. Then, it calculates a Message

Authentication Code (MAC) using its local nonce. Finally, it transmits a second message carrying that MAC, referred to as the *authentication message*. Upon receiving the application message, the receiver calculates a MAC for it using its local nonce, which it compares against the MAC carried by the authentication message it subsequently receives. If the two MACs are equal, authentication passes and the application message is accepted by the receiver, whereas it is discarded otherwise.

In this VulCAN extension, the sender introduces a delay before sending the authentication message, which is an encoding of the N least significant bits of its local nonce value. The receiver proceeds as before, yet monitors the message pair's inter-arrival time. Should authentication fail using its local nonce, it does not immediately discard the application message at hand. Instead, it replaces the N least significant bits of its local nonce with the bits it decodes from the obtained inter-arrival time, and then calculates a different MAC using that new nonce, with which it retries authentication.

As such, this approach migrates the IAT channel discussed in Chapter 4 and Chapter 5 to the practical context of VulCAN. More specifically, the interrupt driven implementation of Section 5.4 is used here, as it was assessed to offer the highest reliability. Whereas the latter carries its covert payload in timings between application messages, this instance exploits their individual timing relative to their authentication frame. Apart from that consideration, they employ the same mechanisms for covert communication, which in this approach are leveraged for the specific purpose of nonce synchronisation in VulCAN's vatiCAN backend.

6.3 Motivation and Drawbacks

This section motivates the nature of the approach presented in Section 6.2, more specifically in its use of authentication frame timing for covert transmission on the one hand, and in the application of that covert bandwidth to nonce synchronisation in VulCAN's vatiCAN backend on the other hand. That distinction is explicitly made, to emphasize how either could benefit VulCAN design independent of the other, when moving beyond the context of the specific approach discussed here.

6.3.1 Covert Communication through Authentication Frame Timing

Transparency. The VulCAN library has full control over both the sending and receiving of application messages, as well as authentication messages. As such, all timing-related operations needed for exploiting this covert bandwidth can be embedded in the VulCAN library itself, without affecting its user interface.

It should however be noted that real-time aspects of VulCAN use are affected when exploiting authentication frame timings. More concretely, as the VulCAN library at receiver side propagates application messages to the receiver's higher-level logic only after they have been authenticated, delaying authentication frames consequently delays application frame processing. This exploitation thus cannot be deemed fully transparent, due to these timing considerations.

Backward compatibility. As the original VulCAN library does not rely on any authentication frame timing management in sending and receiving CAN traffic, the covert transmission described here is a mere extension of its existing functionality, and no compatibility with vulcanized nodes is broken when enabling it. Moreover, it does not introduce any additional CAN bus load in comparison to original VulCAN, yet only an extra computational load at sender side due to payload-to-timing encoding, as well as at receiver side for the inverse computation and inter-arrival time monitoring.

However, applications that in their functionality rely on CAN packet timings, depending on their sensitivity to deviation in those timings, might break when their vulcanized CAN nodes transition to using this extension, due to its real-time consequences already described before in the transparency argument. This approach thus offers no absolute backward compatibility.

Application independence. Whereas the IAT channels presented before in Chapter 4 and Chapter 5 require their underlying application to use periodic communication, no such constraints are imposed by the IAT channel used in this approach. As covert information is transmitted through timings not between subsequent application messages, but between each single application message and its corresponding authentication frame, the IAT channel used here does not depend on inter-application-message timings, which relieves from the communication periodicity prerequisite posed by the channels discussed in previous chapters.

Similar to the aforementioned arguments, nuance has to be brought into this claim of application independence. Indeed, application properties can limit the bandwidth of this IAT channel, as authentication frames cannot be delayed beyond subsequent application messages without losing those. As such, high-traffic VulCAN applications offer low, in extremis zero, covert bandwidth.

6.3.2 Application to Nonce Synchronisation in VulCAN's vatiCAN Backend

2^N-length message loss burst recovery. The explicit, although covert, transmission of N least significant local nonce bits, allows for a gap between local nonce values of a sender and a receiver as large as 2^N to be bridged successfully on authentication message arrival. For security purposes, as elaborated on in Section 6.5, only receiver nonces getting behind on sender nonces can be recovered from, which in vatiCAN corresponds to recovery from message loss.

Indeed, vatiCAN guarantees monotonically increasing local nonce values, that are updated on successful message transmission (arrival) at sender (receiver) side, which means a higher local nonce at a sender implies some transmissions were left without a corresponding arrival. As vatiCAN moreover increments local nonce values by one on each successful transmission/arrival, a nonce gap of 2^N corresponds to as many subsequent messages getting lost. In comparison, current VulCAN design sacrifices message loss recovery completely for replay attack resistance in its vatiCAN backend, as discussed in Section 2.2.5, which is partially mitigated by this approach.

6.4 Design and Implementation

The proposed approach to governing nonce synchronisation through authentication frame timing manipulation in VulCAN's vatiCAN backend, is discussed in this section, as an elaboration on the high-level overview given in Section 6.2. Both sender-side and receiver-side logic are covered, each supported by a graphical representation. This design was inspired by a counter-based nonce synchronisation mechanism proposed for AUTOSAR [37], mainly in its receiver-side processing of the partial nonces transmitted covertly. Some considerations on the configuration of this design are furthermore given, as to illustrate the motivation for and drawbacks to this approach in a more practical manner than used before in Section 6.3. Finally, a practical implementation of this design is discussed.

6.4.1 Sender Side

In original VulCAN design, a sending node on each authenticated message transmission first sends the message at hand, then calculates its corresponding MAC value and without delay transmits that value as the payload of a second message.

In this alternative nonce synchronisation mechanism, that sender encodes the N least significant bits of the nonce it used for MAC computation, to an inter-transmission time it then establishes through delaying transmission of that authentication payload. In this design, that delay is caused by executing a busy-waiting loop of appropriate length. ITT encoding furthermore is done through multiplying the number formed by the N nonce bits to be transmitted, by a timing interval δ. Figure 6.2 graphically represents the logic executed at sender side, with the steps differing from original VulCAN design indicated in grey.

6.4.2 Receiver Side

Receiving nodes on an authenticated communication channel in the original VulCAN design first receive an application message, then calculate an expected MAC value using their local nonce value, and then receive the corresponding authentication frame. If the latter carries a payload equal to that expected MAC, authentication succeeds and the application message is proceeded to the receiver's higher-level logic. Otherwise, that application message is discarded.

The first extension to this design resides mostly in CAN driver software, and accommodates the collection of IAT values. First, a circular *IAT buffer* is added to the receiver's software, which holds inter-arrival timings of both application messages and authentication messages, and which is accessible to CAN driver software as well as the VulCAN library. Registration of IAT values is embedded in CAN driver software, using an interrupt mechanism similar to the IAT channel implementation of Section 5.4. On each CAN message arrival, an interrupt is fired, whose ISR entails measuring its timing relative to the preceding arrival and storing that value in the IAT buffer. That ISR is included in Appendix A, which shows it counts only 5 lines of C code. Moreover, an *IAT index* is added to the receiver's software, which is

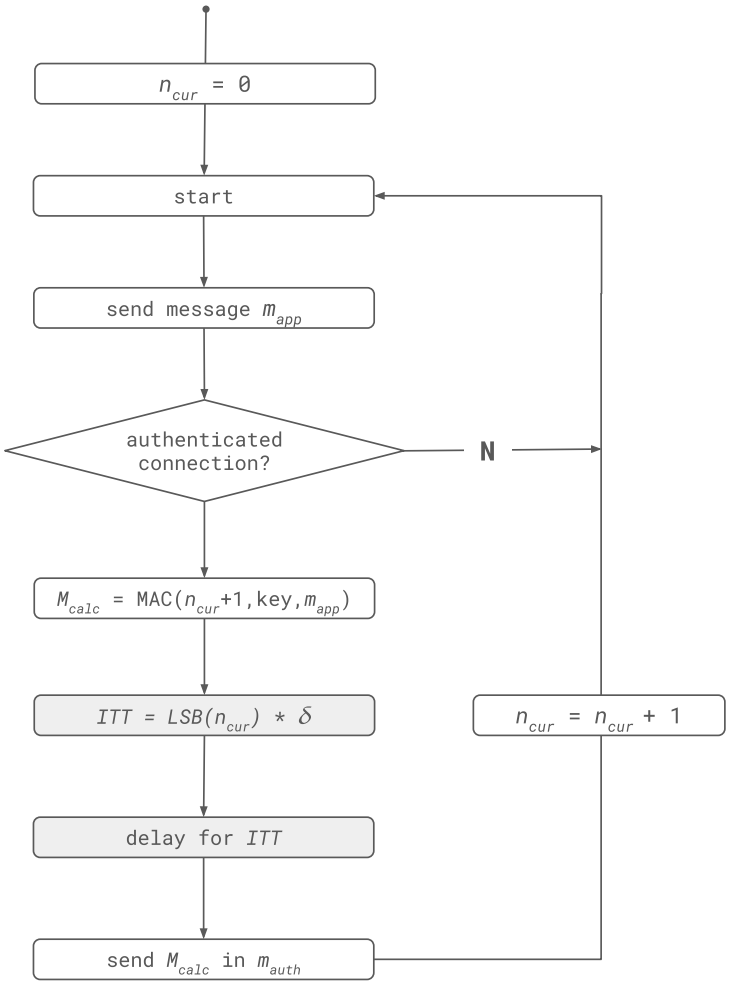

Figure 6.2: Sender-side message transmission flowchart when leveraging authentication frame timings for nonce synchronisation in VulCAN's vatiCAN backend. Differences to the original VulCAN design are indicated in grey. N denotes the amount of nonce bits transmitted covertly, δ the IAT value granularity used.

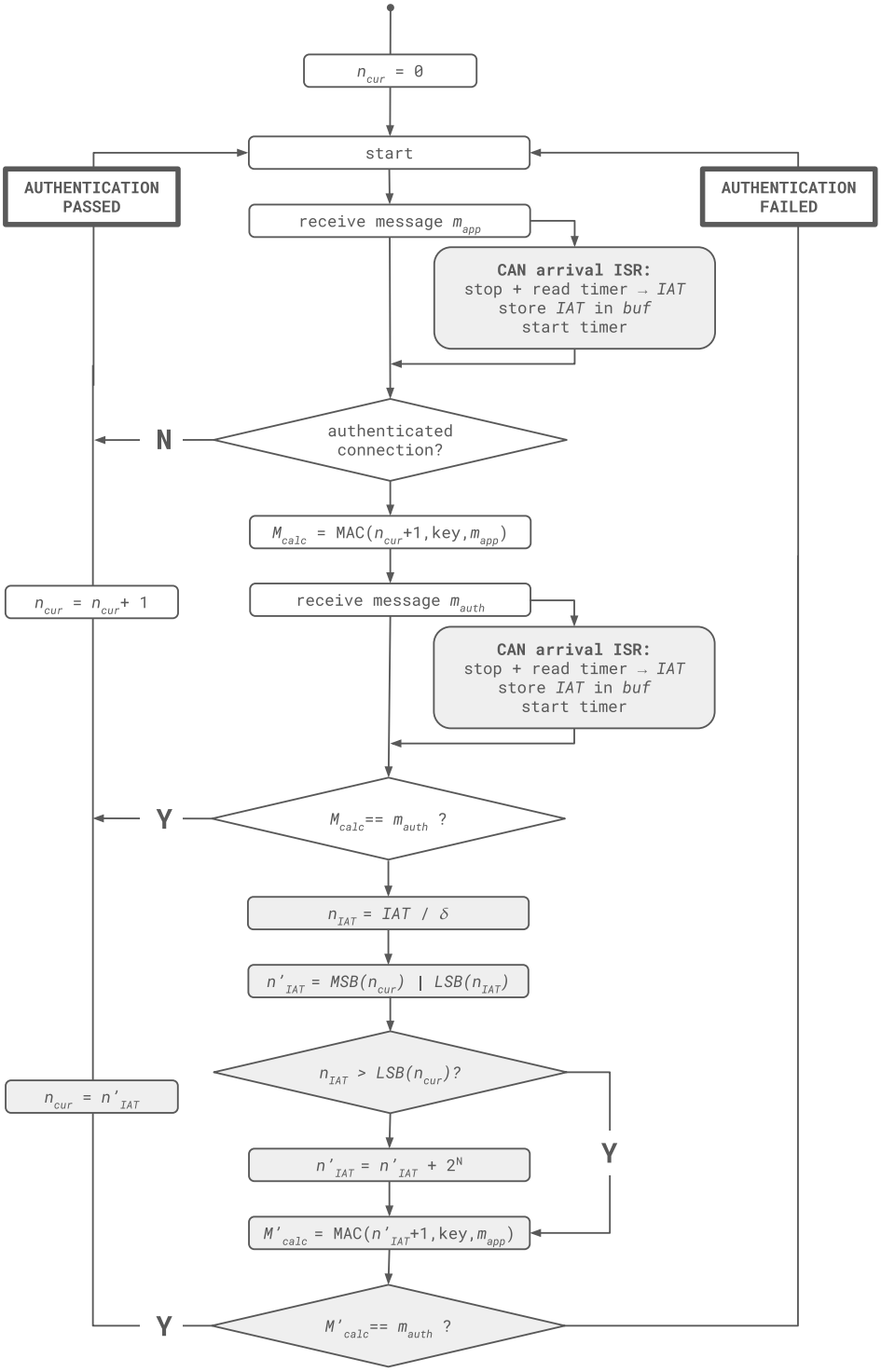

Figure 6.3: Receiver-side message receiving flowchart when leveraging authentication frame timings for nonce synchronisation in VulCAN's vatiCAN backend. Differences to the original VulCAN design are indicated in grey. N denotes the amount of nonce bits transmitted covertly, δ the IAT value granularity used. *buf* refers to a buffer holding IAT values.

updated by this extended CAN driver on message arrival, to denote the position of the most recently registered value in the IAT buffer. That index, like the IAT buffer, is accessible by both the VulCAN library and this extended CAN driver.

Second, the vatiCAN backend of the VulCAN library itself is extended with additional operations on message authentication failure. In such an event, a new tentative nonce first is constructed based on the most recent authentication frame's timing as stored in the IAT buffer, and located in memory by adding the IAT index to the start address of that IAT buffer. In construction of that new nonce, N nonce bits N_{dec} first are obtained by dividing that corresponding IAT value by δ (cf. Section 6.4.1), and then compared to the N least significant bits N_{rec} of the receiver's local nonce value. If N_{dec} exceeds N_{rec}, the receiver's local nonce least significant bits are replaced by N_{dec}. Otherwise, the upper $32 - N$ bit part of that local nonce value is incremented by 1 before that replacement is done.

Based on that new nonce, a second MAC value is calculated at the receiver, which is then compared to the received MAC value. In case those are equal, authentication succeeds on this second attempt, and the application message at hand is propagated to the receiver's application logic. Should authentication however fail again, the receiver's nonce is set to its original local value, on which the first authentication failure occurred, and the application message at stake is discarded.

Figure 6.3 illustrates these receiver-side operations, indicating additions to the original VulCAN implementation in grey. The execution flow branching shown in that overview, refers to the interrupt driven approach to IAT value registering as implemented in Section 5.4, i.e., that registration is done in the execution of an interrupt service routine that is embedded in CAN driver software.

6.4.3 Design Parameter Configuration

Section 6.3 introduced both the motivation for and drawbacks to this approach. The trade-offs that consequently exist in configuring the design parameters left unquantified up to this point, are described in this section.

Number of nonce bits transmitted covertly (N). The amount N of least significant nonce bits that are covertly transmitted through authentication frame timings, requires careful deliberation concerning the underlying application's properties, and targeted robustness against message loss.

A higher value for N means more bits are encoded in one authentication frame delay, which means larger delays can occur. Indeed, encoding more bits requires more variety in ITT/IAT values, and given a fixed timing interval δ between encoding possibilities, that means larger delays need to be established (cf. Section 6.4.1). As discussed in Section 6.3, such delays could break backward compatibility for a subset of VulCAN applications. However, message loss bursts as long as 2^N can be overcome by this proposed scheme, as explained in Section 6.3. In that perspective, a higher value for N thus is desirable, which introduces a trade-off in the determination of N between backward compatibility, and message loss robustness.

IAT encoding granularity (δ). The value for δ denotes the granularity of ITT/IAT values, i.e., the timing interval distinguishing different nonce bit encodings. Another trade-off presents itself in defining this value, namely between channel reliability and backward compatibility.

A higher δ means IAT values can be affected by (attacker) bus noise more extensively before causing a receiver to decode them to unintended nonce bits, which means this mechanism becomes more reliable than when using a lower value for δ. That statement is confirmed by the assessments of previous IAT channel implementations (Section 4.3, Section 5.3, Section 5.4.3). However, such higher δ values inherently cause larger authentication frame delays, meaning the same backward compatibility issues as mentioned in Section 6.3 emerge. Consequently, the choice of δ is a non-trivial consideration in configuring this design.

6.4.4 Implementation

An implementation of this nonce synchronisation strategy was done on Sancus-enabled MSP430 [46] hardware leveraging MCP2515 [25] CAN controllers. The vatiCAN backend of the existing VulCAN library[1] was extended accordingly for nonce synchronisation, and its MCP2515 driver[2] for CAN arrival interrupt handling, as well as IAT buffer management. Together, these additions put into practice the execution flow that was presented and discussed in this section. Appendix B discusses the scenario in which this implementation was assessed, as well as the results that were obtained correspondingly.

This implementation is made publicly available for merging upstream in VulCAN at https://github.com/Stienvdh/vulcan/tree/iat-nonce.

6.5 Security Considerations

As nonce mechanisms have been put in place for security purposes, this nonce synchronisation approach is carefully designed not to harm security guarantees, nor enlarge VulCAN's attack surface through extending it with this scheme. In that perspective, this section discusses the system model and attacker profile taken into account, as well as the security measures put in place to prevent the latter from being more effective than in original VulCAN design.

6.5.1 System Model

Figure 6.4 shows the system model considered in discussing the security properties of this nonce synchronisation approach. It consists of a sender and a receiver that are both vulcanized ECUs and that communicate over an authenticated connection, which is built upon a CAN bus that is also accessible by a third, rogue ECU. As this nonce synchronisation scheme is formulated as an extension to VulCAN, both the

[1] https://github.com/sancus-pma/vulcan/blob/master/can-auth/vatican.c
[2] https://github.com/sancus-pma/vulcan/blob/master/drivers/mcp2515.c

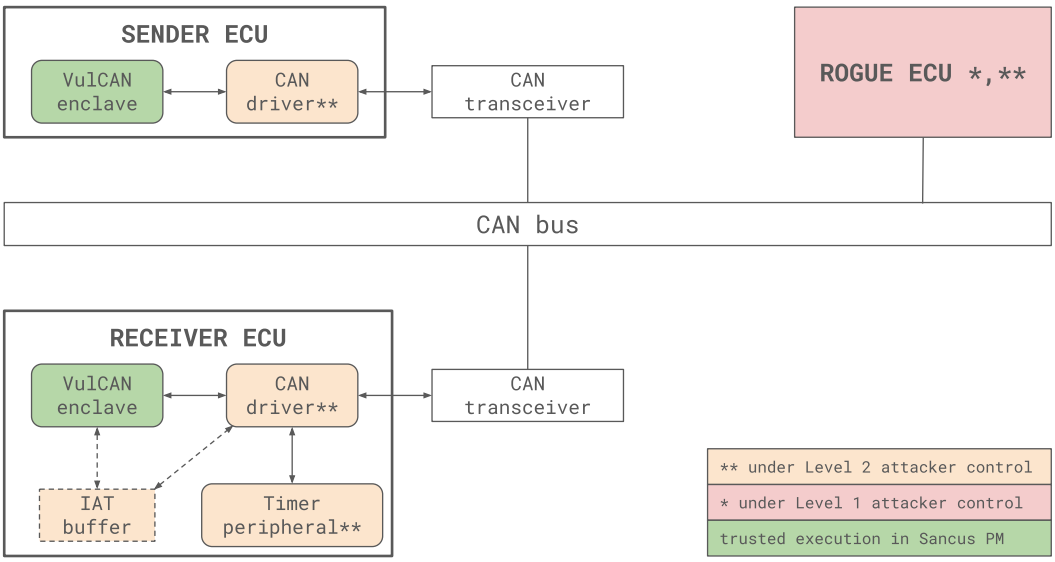

Figure 6.4: System model considered in exploiting authentication frame timing for nonce synchronisation in VulCAN

sender and the receiver are assumed to govern authenticated CAN communication through an appropriately extended VulCAN library that is executed in a trusted Sancus enclave. Their respective CAN driver software and CAN transceiver hardware lie outside that trusted environment, and are therefore deemed under attacker control. The receiver ECU is furthermore considered to provide with an untrusted timer peripheral, which serves to measure authentication frame timings, and an IAT buffer in which those are stored.

6.5.2 Attacker Profile - Level 1: Network Attacker

As demonstrated by previously described CAN network attacks [12, 35] and elaborated on in Section 2.1.2, CAN networks inherently offer little security. By extension, a timing channel built upon CAN communication, such as the one proposed here, is vulnerable to multiple security threats. This Level 1 attacker model considers a malicious party that, either via remote code injection [27] or physical vehicle access, can execute arbitrary code on an ECU that is connected to the CAN bus over which VulCAN-based authenticated communication is done, yet does not participate in it. Its position is represented and indicated in the system model illustration of Figure 6.4 accordingly, in the form of a rogue ECU.

This Level 1 attacker can successfully affect the timing channel used in this approach as listed below. Note that these actions pertain to IAT transmission itself, rather than its underlying CAN communication. Nevertheless, both are evidently interdependent, which leads to most IAT channel attacker capabilities following from the CAN network attacker capabilities listed in Section 2.1.2.

— **A1: Eavesdropping.** As CAN employs broadcast communication without enforcing bus access control, this attacker can eavesdrop on, as well as record, packet inter-arrival times and thus all IAT channel payloads. As nonce values however can be disclosed without affecting security guarantees (cf. Section 2.2.5), this attacker capability does not cause any harm and thus will not be focused on hereafter. This disclosing of all CAN traffic does however provide the attacker with the scheduling and/or CAN network architecture knowledge that could be a prerequisite for other attacker capabilities [12].

— **A2: Message manipulation.** In multiple ways, an attacker is capable of delaying the CAN traffic constituting IAT values. For example, injection of high-priority dummy frames, or well-timed 0-bit transmissions, can cause legitimate CAN transmission to be delayed due to the nature of its arbitration mechanism (cf. Section 2.1.1), or targeted nodes can temporarily be forced into bus-off mode [12, 35]. Consequently, all IAT values are vulnerable to modification. Note how attackers can both increase and lower IAT values, through respectively delaying authentication and application frames, which means an attacker can cause arbitrary IAT values to be constituted.

— **A3: Message deletion.** In order to delete IAT payloads from this covert communication channel, the corresponding CAN packets are to be deleted, through for example CAN bus flooding [12], or a selective DoS attack (cf. Section 2.1.2). There is no way for attackers to delete IAT payloads from this timing channel without suppressing the corresponding CAN frames, as their presence inevitably can be detected, and their timing thus registered.

— **A4: Message insertion.** Analogous to **A3**, an IAT value can be inserted into this IAT channel through inserting two corresponding CAN frames on the targeted CAN bus. As a Level 1 attacker has free access to that bus, she can place any two messages with any inter-arrival time on it.

  Note that those inserted messages need to have the proper ID's in order for their inter-arrival time to be registered by a vulcanized receiving node, since the message ID masking and filtering as described in Section 5.4.1 is included in this design as well. More specifically, VulCAN dictates the use of a message ID equal to the application message's incremented by one, for its corresponding authentication frame. For IAT value insertion to work, the second inserted message's ID thus has to be equal to the first message's ID incremented by one, and the latter moreover has to be equal to the ID used by the targeted VulCAN communication channel.

6.5.3 Attacker Profile - Level 2: Software Attacker

Going beyond the Level 1 profile, VulCAN [51] assumes a more powerful attacker that is capable of arbitrary code execution in all software modules that are not Sancus-protected. Consequently, it controls both non-vulcanized nodes like the Level 1 attacker does, and non-enclaved modules on vulcanized nodes. The system model

considered here, like VulCAN design, explicitly excludes CAN driver software from its trusted computing base, which means driver operations are to be considered untrusted, and under this Level 2 attacker's control. With that driver logic propagating all CAN traffic to and from VulCAN software, overtaking it yields attacking parties an ideal man-in-the-middle position, as illustrated in Figure 6.4.

That same system model illustrates how a Level 2 attacker moreover controls the timer peripheral measuring authentication frame timings at receiver side, which causes its measurements to be deemed untrusted. Moreover, its IAT buffer is under Level 2 attacker control as well, rendering its contents untrusted. Finally, Section 6.4.2 discusses how CAN driver software is in charge of updating a, thus untrusted, index into that IAT buffer, to locate the most recent IAT value. These considerations give rise to the attacker capabilities listed below. Note that a Level 2 attacker also has **A1**-**A4** abilities, as it is strictly more powerful than a Level 1 attacker.

— **A5: Stealthy message manipulation.** As opposed to the IAT message manipulation capability **A2** discussed in the Level 1 profile, a Level 2 attacker is capable of altering IAT values without invading the CAN bus involved. At receiver side, it can take over either CAN driver software, IAT buffer contents, or the timing peripheral and thus cause arbitrary values to be registered and used as alleged authentication frame timings.

— **A6: IAT buffer location manipulation.** CAN driver software is deemed untrusted in the system model illustrated in Figure 6.4, and requires write access to the IAT buffer. Therefore, that buffer is placed in unprotected memory, and thus under control of a Level 2 attacker. With this extended VulCAN library reading IAT values from that buffer, that situation leads to untrusted memory locations being dereferenced by trusted software. With trusted modules being granted access to both private and public memory locations, a Level 2 attacker could, without breaking VulCAN functionality, position this IAT buffer partially or fully inside enclave memory. IAT value dependent and attacker-observable properties of VulCAN execution, e.g., the passing/failing of authentication, can subsequently leak protected data, as that data in this scenario is used as alleged IAT values.

— **A7: IAT index manipulation.** An index into the receiver's IAT buffer, that denotes the position of the most recently collected IAT value, is updated by CAN driver software, and therefore under Level 2 attacker control, as well as used for IAT value retrieval in trusted VulCAN code. Therefore, this attacker can attempt to make that trusted code dereference an arbitrary, possibly protected, memory location, by supplying a corresponding index. Moreover, similar to the **A6** capability, this attacker could position that IAT index at a protected location in memory, and deduce private information from the trusted VulCAN library dereferencing that location.

6.5.4 Known Nonce Mechanism Threats

Since the IAT channel described here is used for nonce synchronisation, it must be designed carefully such that the attacker capabilities discussed before cannot be leveraged for malicious activity targeted at that nonce mechanism, and thus the freshness guarantees it aims to establish. The following attack scenarios describe how nonce mechanisms can in general serve as an attack vector, and how those threats present themselves in the original VulCAN implementation.

Replay attacks. Through recording CAN traffic, and later replaying it on the same bus, applications that reuse nonce values can be tricked into accepting that malicious, replayed traffic as authentic. As such, the intended freshness guarantees are broken, as messages thus can get accepted and processed multiple times. Such a scenario is what is generally referred to as a replay attack.

VulCAN design acknowledges how the vatiCAN protocol for message authentication is particularly vulnerable to an advanced replay attack [51], and mitigates that threat by explicitly disabling nonce reuse in its vatiCAN backend at the cost of message loss recovery, as explained in more detail by Section 2.2.5.

Artificial nonce incrementation. Through incrementing the nonce a receiver believes to be used by a sender, without it having received a validly authenticated message to rightfully cause such an incrementation, subsequent sender messages fail authentication at that receiver, as their MAC is calculated using a different nonce than expected by that receiver. When implemented carefully, this kind of attack could moreover lead to nonce reuse at the receiver, and thus a replay attack.

Through the use of monotonically increasing nonces in both VulCAN backends, as well as a secret session key in authentication frame computation, VulCAN attackers [51] can only launch such an incrementation attack in the form of a replay attack, which has been described above. Indeed, as they are assumed not to be in possession of the session key used, because it resides in Sancus-protected memory, they are unable to construct a valid authentication payload for a message and nonce value other than already transmitted/used by the underlying VulCAN application. However, constructing a valid authentication payload is the only possible approach to causing nonce incrementation in VulCAN, as nonce management is done in enclaved execution.

6.5.5 Security Measures

This proposed nonce synchronisation approach was designed not to harm VulCAN security properties, i.e., not to enlarge its vulnerability to both replay attacks and artificial nonce incrementation, as discussed by Section 6.5.4, in the face of both Level 1 and Level 2 attackers. All security measures listed below are executed in a protected Sancus module by design, as they form an extension of the VulCAN enclaves depicted in the system model of Figure 6.4.

M1: ITT (IAT) encoding (decoding). The encoding and decoding between nonce bits and ITT/IAT values in the mechanism described here is a simple multiplication with/division by a parameter δ. This approach, in a less powerful attacker profile, could have been designed to provide additional security guarantees, but as attackers can eavesdrop on, and arbitrarily manipulate, all IAT values (cf. Section 6.5.4), this design is justified to be as algorithmically simple as possible.

M2: Monotonically increasing nonces. This scheme entails that whenever authentication fails using a receiver's local nonce value, a second try is done using a different nonce that is constructed from decoding the corresponding authentication frame timing (cf. Section 6.4.2). That construction is explicitly designed to only yield nonces higher than the original local value, i.e., when the receiver changes its local nonce value because any authentication try succeeds, it is guaranteed to become strictly higher than its original local nonce value.

M3: Delayed nonce commit on authentication failure. Should an authentication retry using an IAT based nonce fail, the local nonce of the receiver involved is reset to its original value, which was used for the first authentication attempt. Consequently, local nonce value incrementation only happens when the corresponding authentication frame is calculated using either the receiver's original, or IAT-constructed nonce value.

The combination of **M2** and **M3** requires an attacker targeting this nonce mechanism to possess a valid pair of application and authentication messages, that is moreover calculated using a nonce strictly higher than that of the receiving party in the targeted communication channel. Without such a pair, its modification, deletion and insertion capabilities on this IAT channel are insufficient to launch either a replay attack, or a nonce incrementation attack based on this VulCAN extension.

A valid message pair is available only when the attacker involved is either capable of constructing its own valid authentication frame for an application message, or can replay a recorded pair. On vulcanized nodes, MAC calculation is done in a protected Sancus module, to which an attacker (of either Level 1 or Level 2) has no access, rendering proper authentication frame computation impossible. The second approach of replaying recorded traffic has been disabled in VulCAN [51], and since any opportunity for a replay attack directed at this nonce synchronisation approach coincides with an opportunity for launching a replay attack against original VulCAN communication, this scheme does not introduce a replay attack vulnerability.

Consequently, **M1**, **M2** and **M3** together offer resilience of this nonce synchronisation scheme to a Level 1 attacker (cf. Section 6.5.2). Indeed, none of its capabilities render it capable of generating effective, valid message authentication pairs. A Level 2 attacker (cf. Section 6.5.3) does not have that ability either, but its control over an IAT buffer and IAT index read by trusted vatiCAN code leads to the additional need for security measures **M4** and **M5**.

M4: IAT buffer validation. Attacker capability **A6** describes how the system model of Figure 6.4 leads to trusted code dereferencing Level 2 attacker-controlled memory locations, which is a known attack vector for leaking enclaved memory [53]. Validation of that buffer is therefore done at VulCAN initialisation. More specifically, it is checked to completely lie outside enclaved memory. As a minor contribution of this work, the currently insufficient functionality in Sancus for verifying a buffer to lie completely outside of protected memory is extended to follow good practices [53]. This code is made publicly available for merging upstream in Sancus at https://github.com/Stienvdh/sancus-compiler/tree/buffer-check.

M5: IAT buffer index validation. Attacker capability **A7** poses the same threat of trusted code dereferencing untrusted memory locations as described in **M4**, which in this design is mitigated by validating the IAT index to have a value between 0 and the length of the IAT buffer at the time it is dereferenced by trusted VulCAN code. Moreover, that index itself is validated to lie outside of protected memory.

In conclusion, these security measured do not disable a malicious party from executing its capabilities **A1**-**A7**, yet mitigate their possibly harmful consequences. Indeed, taking **A7** for an example, a Level 2 attacker is still capable of changing the IAT index to an arbitrary value, yet it cannot leverage that ability for trusted memory leaking due to **M5**. Similarly, IAT values can still be manipulated due to both **A2** and **A5**, but a subsequent effective replay attack is disabled by the combination of security measures **M2** and **M3**.

6.5.6 Novel VulCAN Attack in Extended vatiCAN Backend: Message Suppression

Despite the security measures taken in this design, it enables a novel attack against VulCAN's vatiCAN backend involving message suppression. This subsection presents a concrete attack scenario, and proposes an effective mitigation.

Attack scenario. The very motivation of this approach enables a message suppression attack scenario previously ineffective in VulCAN's vatiCAN backend. It is not directly related to freshness guarantees, yet can be executed by both a Level 1 and Level 2 attacker when enabling this proposed nonce synchronisation strategy. Concretely, due to this approach allowing for recovery from message loss, deliberate message deletion is intrinsically recovered from as well.

More specifically, an attacker is capable of selectively suppressing as much as 2^N subsequent CAN application messages at a time, while all other messages are processed normally at receiver side. This attack can harm vehicle safety, e.g., when messages warning for malfunctioning components are suppressed, and the receiver party involved unknowingly continues regular execution while it would initiate some reactive operation sequence when receiving those suppressed warnings. In contrast, the use of the vatiCAN backend in original VulCAN design implies discarding of all CAN traffic subsequent to such message suppression (cf. Section 6.1).

Note how in VulCAN's LeiA backend, which is unaffected by this approach, this message suppression attack is enabled as well, due to its explicit transmission of the lower half of nonce values offering the same message loss resistance that forms this attack vector in this extended vatiCAN backend.

Mitigation. Since a receiver party cannot distinguish between local nonce desynchronisation caused by either message loss or message suppression, it cannot detect this attack scenario to be executed. Moreover, prevention of the multitude of denial-of-service attacks against CAN [12, 18, 35], which enable deliberate message suppression, is far beyond reach in current CAN design. Therefore, it is up to the application using this extended VulCAN library to harden itself against message suppression, if necessary in its context. Concretely, such an application could include some logical sequence number in its CAN packets, as to be able to detect missing packets in the message stream arriving through the VulCAN library at receiver side.

6.5.7 Interrupting Trusted Execution

This nonce synchronisation approach, in pursuit of maximal authentication frame timing accuracy, is built upon the interrupt driven approach to inter-arrival time registering that was discussed and assessed in Section 5.4. More concretely, Figure 6.3 shows how at receiver side, an interrupt is fired on CAN message arrival, in whose interrupt service routine its timing relative to the most recently arrived frame is registered. In the system model illustrated in Figure 6.4, this corresponds to the receiver's CAN controller interrupting execution on the receiver ECU, and transferring control to its CAN driver module.

As that receiver ECU runs both trusted and untrusted modules, this mechanism should be able to interrupt both without harming security guarantees. The ISR concerned is embedded in untrusted CAN driver software, which means interrupting untrusted execution does not entail any security complications. In contrast, protected enclave interruption involves exiting and re-entering a trusted context, which does affect security guarantees when implemented improperly.

Interrupt based protected module attacks. Enclave entry and exit in itself has been thoroughly researched in Sancus [31, 32], as well as other TEEs [23], which is leveraged in supporting interruptible enclaves [9]. However, the Nemesis attack [54] has proven interrupt latency of Sancus- and Intel SGX-protected modules to leak information on enclaved instruction sequences, and by extension enclaved data like passwords and PIN codes. In concurrent research, SGXlinger [14] shows interrupt latency in Intel SGX to leak more coarse-grained information on enclaved execution. Therefore, interruption of the receiver ECU in the system model assumed here is to be deemed vulnerable to Nemesis-like side-channel attacks, when none of the following mitigations are enabled in its trusted execution environment.

Mitigation 1: non-interruptible enclaves. Given these known interruptible enclave vulnerabilities, a naive yet effective countermeasure lies in prohibiting interrupts during enclave execution, as is the case in original Sancus design [31]. As argued by Busi et al. [5], that approach however endangers availability guarantees by risking infinite enclave execution, and limits functionality of enclaved applications.

In the specific context of this nonce synchronisation mechanism, its correct functionality is indeed broken when enclave interrupts are disabled, even in absence of malicious actors. Indeed, when interrupts are disabled during trusted execution, authentication frame timing registration could be deferred until after enclave exit, leading to possibly severe loss of accuracy. Indeed, message arrivals at any time during enclave execution are then registered as having occurred on enclave exit, regardless of their exact timing. Depending on the protected module software properties, such loss of accuracy could render this nonce mechanism useless, i.e., without meaningful added value to current VulCAN design in terms of resilience against message loss.

Mitigation 2: eliminating side channels. As interruptible enclave attacks mostly leverage side channels stemming from interrupt latency measurements, their mitigation when retaining enclave interruptibility must explicitly eliminate such information leakage. Busi et al. [5] propose a provably secure interruptible enclave design, relying on full abstraction in proving its security guarantees. In short, their approach extends existing interruptible enclave design with interrupt latency padding such that an attacker cannot derive any information from latency measurements.

Authentication frame timing effects. As depicted in Figure 6.3, an interrupt is fired on both application and authentication frame arrival at a receiver ECU in this design, upon which their timing is registered in the handling of that interrupt. The overhead incurred in transferring control to an interrupt service routine depends on whether trusted or untrusted execution was interrupted, since the former context requires exiting a trusted enclave, whereas the latter does not. However, as *inter*-arrival timings are measured in the concerned ISR, values stored in the receiver's IAT buffer will not be affected by that variability as long as the application and authentication message arrivals either both interrupt untrusted, or trusted execution, except for possible Nemesis-exploitable variations in the latter case. Otherwise, their inter-arrival time will be registered to be lower or higher, when respectively only the application frame, or its corresponding authentication frame interrupt an enclave. These considerations are assessed and confirmed in Appendix B.

Apart from IAT effects, the nature of the execution interrupted at receiver side moreover influences the amount of computational latency incurred by enabling the VulCAN extension proposed here, which as repeatedly stated in Section 6.3 could break backward compatibility with real-time sensitive VulCAN applications.

6.6 Conclusion

This chapter discussed how authentication frame timings in VulCAN's vatiCAN backend can be exploited for nonce synchronisation purposes. The motivation for this proposed approach lies in both the nonce-related challenges concerning performance and security in current VulCAN design, and the benefits of backward compatibility and transparency gained from covert communication. The proposed scheme encodes the N least significant bits of the nonce value used in authentication frame computation, in the timing of that frame. Whenever authentication fails at receiver side, that timing is decoded to construct an alternative nonce value, with which authentication is retried. As such, up to 2^N subsequent message losses can be recovered from when IAT values are established correctly, compared to zero in the vatiCAN backend of original VulCAN design.

As this approach involves nonces, which are typically installed for security purposes, the security implications of this approach have been thoroughly investigated. Two attacker profiles were defined, one of which is more powerful and equivalent to the attacker model considered in original VulCAN design. Next, suitable security measures were installed to offer resilience against both attacker profiles, form whose discussion was concluded that this extension does not enlarge VulCAN's attack surface, beside the introduction of a message suppression attack vector which the alternative LeiA backend of VulCAN was already vulnerable to. Finally, as this approach is built on an interrupt driven timing channel, the effects on security and latency of interrupting trusted execution were considered for different approaches to enclave interruption, and assessed for this specific VulCAN extension.

Chapter 7

Conclusion

The problem statement made at the beginning of this work, addressed how in-vehicle networks like CAN nowadays are faced with a powerful remote attacker, and how existing security measures, although they are effective, go against the strong performance requirements of this safety-critical automotive context. In a less resource-intensive approach to securing CAN communication, TACAN [58] proposes to repurpose side channels, which typically serve as an attack vector [54, 17, 22], for defence instead of harm, by using the supplementary bandwidth they offer for accommodating security measures without increasing network load.

In this master's thesis, such covert communication was found to offer considerable bandwidth and reliability, as well as substantial benefits when incorporated into existing security implementations, all while exhibiting a large degree of backward compatibility and transparency.

This concluding chapter first repeats the contributions made in that perspective, and thereafter couples their remaining limitations to possible areas for future work.

7.1 Contributions

Below, the main contributions of this master's thesis are listed.

— An analysis of covert bandwidth sources in CAN was executed, and its findings were synthesized in a matrix structure. Chapter 3 discussed the methodology used for ensuring a structured approach to that exploration, and its results.

— A quantitative evaluation of timing based covert communication in CAN was conducted, and enabled by several practical implementations of a packet inter-arrival time channel. Chapter 4 considered a proof-of-concept implementation in Java, whose performance and reliability was improved on by a lower-level prototype discussed in Chapter 5. Section 5.4 moreover showed how extending that prototype to fire an interrupt on CAN arrival yields higher timing accuracy, and therefore further enhances this channel's performance and reliability. This interrupt driven prototype was made publicly available at `https://github.com/Stienvdh/Sancus-IAT`.

— An extension to the existing VulCAN [51] design for CAN message authentication was proposed in Chapter 6. It partially resolves the availability issues faced in VulCAN's current vatiCAN [34] backend by extending its nonce synchronisation scheme, and ensures backward compatibility and transparency by leveraging covert transmission as presented in the interrupt driven IAT channel prototype of Section 5.4. The security implications of this approach, and the effects of interrupting trusted Sancus enclaves [31] on latency and security, were evaluated in Section 6.5. That discussion showed how this extension's careful design does not harm VulCAN's original security guarantees, beside the introduction of a message suppression attack vector in its vatiCAN backend, which already existed in its alternative LeiA [36] backend.

A practical implementation of this extension was made publicly available for merging upstream in VulCAN at https://github.com/Stienvdh/vulcan/tree/iat-nonce.

7.2 Limitations and Future Work

This section acknowledges the limitations of this master's thesis and its contributions, as well as leverages those for proposing possible future directions in investigating this work's research hypothesis.

Limitation to CAN. Several implicitly fixed variables exist throughout the work conducted in this master's thesis. For instance, the use of covert bandwidth is investigated in the specific context of CAN, whereas other interesting opportunities could exist in alternative network protocols as well. One such protocol is CAN+ [60], which forms an extension to CAN, and is based on exploiting the higher clock frequency of most embedded devices relative to their CAN bus' speed, for extra CAN transmission during their overclocked cycles. This context could allow for more intricate covert channels, for example based on the timing properties of transmission within such overclocked time intervals. That approach could allow for a timing channel similar to the one presented in Chapter 5, without the prerequisite of periodicity in its underlying communication, as timing information of this specific kind can be contained within the transmission of each individual message.

IAT channel prototype technology dependence. The prototype for timing based covert communication presented in Chapter 5, and even more so its interrupt driven enhancement discussed in Section 5.4, is bound to MSP430 hardware for its application logic, and dependent on MCP2515 controllers for regulating CAN communication. By extension, the results obtained in assessment of this prototype are specific for those technologies. On that same note, several corrections for computational latency have been included in this prototype design, as elaborated on in Section 5.1.3, which evidently are technology-dependent as well. Should this timing channel prototype thus be ported to different hardware in future endeavours, a reconsideration of its performance, reliability and configuration is due.

IAT channel design flaws. The inter-arrival time channel that was discussed and evaluated in Chapter 4 and Chapter 5, has a rather subtle flaw in its design. In essence, this IAT channel manipulates the time interval between each message and its predecessor, for encoding a payload in the deviation of such intervals from a pre-defined fixed value. Although this approach is effective in exposing covert bandwidth, it could have considerable long-term repercussions on message latency. Indeed, the delay (expedition) of a message relative to the one sent before, implies an absolute, equal-sized delay (expedition) of all subsequent traffic, as to retain the latter's presumed inter-arrival timing properties. Such long-term latency is compensated for when the decreasing and increasing of relative timings occur at the same rate, which reduces the severity of this consideration. However, the usability of this IAT channel could be improved by eliminating this drawback, for example through only encoding a covert payload in the relative timing of every other message, and using the next to neutralise its effects on long-term message latency, or by changing the approach to encoding covert payloads in this timing channel altogether.

VulCAN extension technology dependence. Analogous to the limitations of the IAT channel prototype discussed before, the implementation of the VulCAN extension proposed in Chapter 6, as well as its results shown in Appendix B, are highly technology-specific. For instance, its design presumes an interrupt mechanism for CAN arrival, which might not be available on alternative hardware. As a second example, the considerations on interrupting trusted execution that are made in Section 6.5.7, and quantified in Appendix B, are expressed specifically for Sancus enclaves, and might substantially differ when regarding other TEEs[23].

VulCAN extension applicability limitations. The VulCAN extension proposed in Chapter 6 has a rather limited scope, as it is built on the VulCAN-specific approach to message authentication of pairing each message with a subsequent authenticating frame, controlling the transmission and arrival of both, and having a user interface transparent to authentication traffic. The proposed approach therefore is not generally portable to other message authentication frameworks. Moreover, the applicability of this extension is restricted to VulCAN's vatiCAN backend, leaving the issues faced in its LeiA counterpart unaccounted for. The proposed design however registers message timings independent of their use, which might allow for them to benefit the LeiA backend as well, should a suitable application context arise.

Although timing registration is done regardless of its use, an application context for the covert bandwidth this extension is built on beyond VulCAN library boundaries is not enabled by the current approach, as the sender-side logic regulating authentication frame timing communication is currently embedded in VulCAN code itself. In that light, future work could extend VulCAN's user interface to allow for the specification of an authentication frame timing parameter in transmission, thus enabling generic use of the covert bandwidth exposed by this VulCAN extension.

Applications beyond VulCAN. This work discussed a practical application of covert transmission in VulCAN's approach to message authentication. However, the covert channels discussed in Chapter 3 were formulated for CAN communication in general, which means they could benefit application domains beyond VulCAN as well. They moreover are not restricted to be leveraged for message authentication, or other security mechanisms, as they offer supplementary bandwidth usable for any purpose. As such, they could even be implemented as an extension to CAN communication itself when embedded in CAN driver software, instead of incorporated in existing applications and designs. They could therefore enable a backward compatible, transparent mitigation for any existing bandwidth scarcities in CAN applications.

7.3 Concluding Thoughts

When reflecting on the results of this master's thesis, and its significance in a wider context than the problem statement it was motivated by, several final conclusions emerge. First, it was shown how side channels, which are collateral transmission forms used by attackers in extracting trusted information from protected memory, are not a mere attack vector to be eliminated from secure implementations. Instead, they can reinforce such implementations, when their bandwidth is exploited for accommodating security measures rather than leaking data. The opportunities for such covert transmission were made concrete for CAN, and their evaluation showed great potential for them to benefit practical applications in the automotive industry, industrial control, agriculture, or any other context in which CAN is used.

Second, that presumed practical feasibility of covert communication was instantiated by its use in a rather specific context. More concretely, it was leveraged for increasing the robustness of an existing message authentication scheme, without harming its strong security guarantees. That effort was successful in enabling such robustness, while incurring no extra bandwidth use, which traditional message authentication strategies do not allow. Although the scope of this approach is limited, its results confirm the possibility for similar endeavours in exploiting covert bandwidth to enhance a wide range of practical application domains.

The results of this master's thesis thus encourage the widespread use of covert channels in resolving numerous challenges faced by existing and/or future security frameworks, or even any type of benign application logic.

Appendices

Appendix A

Minimal Interrupt Service Routine for Inter-Arrival Time Registration

This appendix includes the interrupt service routine that was developed for registering packet inter-arrival timings in an interrupt driven manner, as used in the IAT channel implementation of Section 5.4 and the VulCAN extension proposed in Chapter 6. It is designed specifically for a MSP430 microcontroller using a MCP2515 controller for its CAN communication, yet illustrates its main hardware-independent operations. The source code was explicitly included below, to prove how the measures taken to render it as minimal as possible (cf. Section 5.4.1), effectively resulted in only 5 lines of executed C code.

```
1  void ican_recv_callback(void)
2  {
3          // Adjust buffer index
4          can_iat_index = (can_iat_index+1)%CAN_IAT_BUFFER_SIZE;
5
6          // Measure + store IAT
7          TSC_TIMER_END(iat_timer);
8          can_iat_timings[can_iat_index] = iat_timer_get_interval();
9          TSC_TIMER_START(iat_timer);
10
11         // Clear interrupt flag on MSP430
12         P1IFG = P1IFG & 0xfc;
13 }
```

Listing 1: Minimal interrupt service routine for inter-arrival time registration on MSP430 hardware with a MCP2515 CAN controller

Appendix B

Timing Based VulCAN Nonce Synchronisation Assessment

Experimental scenario To assess the performance and reliability of the nonce synchronisation approach proposed in Chapter 6, a sequence of 10.000 identical application messages is transmitted from a sending to a receiving vulcanized node, over an authenticated connection using an extended VulCAN library. In this setting, 2 least significant nonce bits (N) are encoded in each authentication frame timing, using an interval of 1200 cycles (δ) between encoding values. As to quantify the effects of Sancus enclave interruption on IAT values, the receiver node is equipped with a simple PM whose only entry point function `wait_for_receive` first polls for CAN message arrival, and then returns. Those additions enable forced enclave interruption, to simulate a real-life context in which that occurs spontaneously.

This scenario is repeated in the following four receiver configurations, yielding their respective results:

— `wait_for_receive` is never called (Figure B.1, Figure B.2 and Figure B.3)

— `wait_for_receive` is only called before application frame receiving (Figure B.1)

— `wait_for_receive` is only called before authentication frame receiving (Figure B.2)

— `wait_for_receive` is called before both application frame and authentication frame receiving (Figure B.3)

Results From Figure B.1 follows that IAT values are on average 30 cycles lower when obtained while interrupting a Sancus enclave on application frame arrival, compared to when only interrupting untrusted execution. In contrast, Figure B.2 shows how they increase by the same average amount when interrupting trusted execution on authentication frame arrival. As therefore expected, Figure B.3 shows how IAT values are roughly equivalent when interrupting an either untrusted or trusted context on both application and authentication frame arrival.

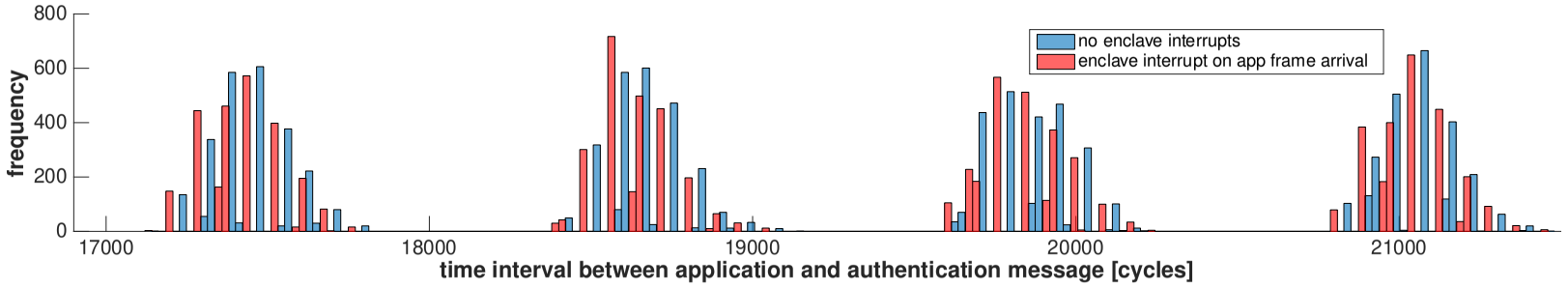

Figure B.1: Nonce-carrying IAT values in VulCAN when interrupting a Sancus enclave on application frame arrival

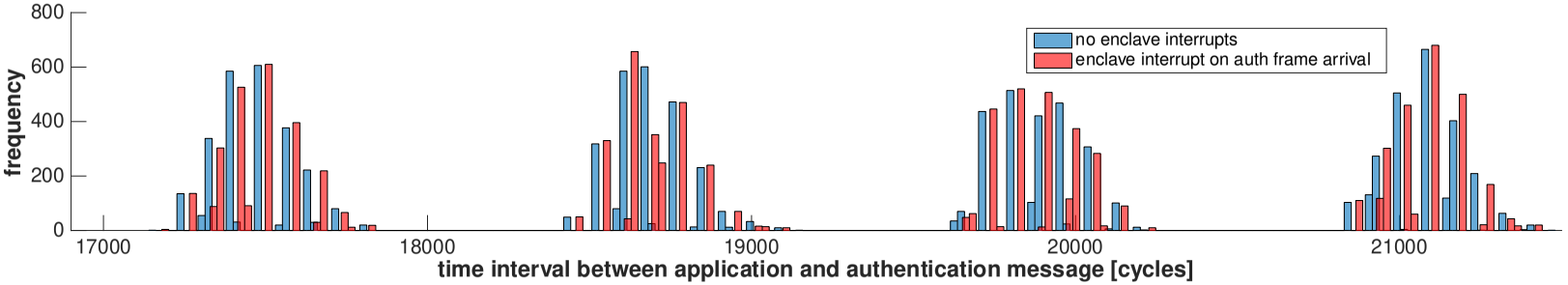

Figure B.2: Nonce-carrying IAT values in VulCAN when interrupting a Sancus enclave on authentication frame arrival

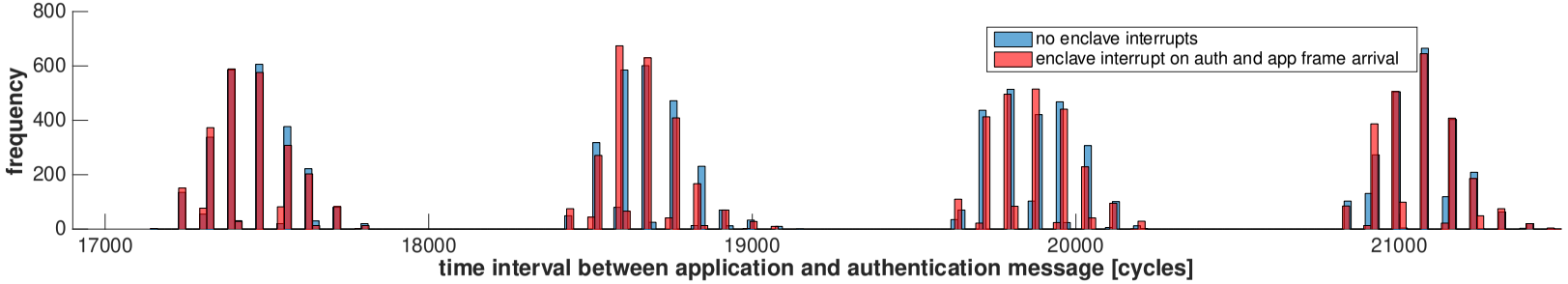

Figure B.3: Nonce-carrying IAT values in VulCAN when interrupting a Sancus enclave on every frame arrival

Bibliography

[1] M. Ambrosin, M. Conti, A. Ibrahim, G. Neven, A. R. Sadeghi, and M. Schunter. SANA: Secure and Scalable Aggregate Network Attestation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 731–742. ACM, 2016.

[2] AUTOSAR Classic Platform Specification 4.3. Specification of secure onboard communication. https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_SecureOnboardCommunication.pdf, 2016.

[3] V. Berk, A. Giani, G. Cybenko, and N. Hanover. Detection of covert channel encoding in network packet delays. *Dartmouth College Technical Report TR2005-536*, 2005.

[4] C. Borrelli. IEEE 802.3 cyclic redundancy check. *Application Notes on Virtex Series and Virtex-II Family, XAPP209 (v1.0)*, 2001.

[5] M. Busi, J. Noorman, J. Van Bulck, L. Galletta, P. Degano, J. T. Mühlberg, and F. Piessens. Provably secure isolation for interruptible enclaved execution on small microprocessors. In *Computer Security Foundations Symposium (CSF)*. IEEE, 2020.

[6] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno, et al. Comprehensive experimental analyses of automotive attack surfaces. In *Proceedings of the 20th USENIX Conference on Security*, volume 4, pages 447–462. USENIX Association, 2011.

[7] CISPA Helmholtzzentrum i.G. vatiCAN documentation. http://www.automotive-security.net/vatican/doc/, 2018.

[8] D. Das, S. Meiser, E. Mohammadi, and A. Kate. Anonymity trilemma: Strong anonymity, low bandwidth overhead, low latency - choose two. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 108–126. IEEE, 2018.

[9] R. De Clercq, F. Piessens, D. Schellekens, and I. Verbauwhede. Secure interrupts on low-end microcontrollers. In *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, pages 147–152. IEEE, 2014.

[10] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito. SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS '12)*, pages 1–15. Internet Society, 2012.

[11] T. Fischl. USBtin - USB to CAN interface. https://www.fischl.de/usbtin/, 2016.

[12] S. Fröschle and A. Stühring. Analyzing the capabilities of the CAN attacker. In *Computer Security – ESORICS 2017*, pages 464–482. Springer, 2017.

[13] B. Groza, S. Murvay, A. van Herrewege, and I. Verbauwhede. LiBrA-CAN: A Lightweight Broadcast Authentication Protocol for Controller Area Networks. In *International Conference on Cryptology and Network Security*, pages 185–200. Springer, 2012.

[14] W. He, W. Zhang, S. Das, and Y. Liu. SGXlinger: A new side-channel attack vector based on interrupt latency against enclave execution. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 108–114. IEEE, 2018.

[15] T. Hoppe, S. Kiltz, and J. Dittmann. Security threats to automotive CAN networks – practical examples and selected short-term countermeasures. In *International Conference on Computer Safety, Reliability, and Security*, pages 235–248. Springer, 2008.

[16] R. A. Kemmerer. A practical approach to identifying storage and timing channels: Twenty years later. In *Proceedings of 18th Annual Computer Security Applications Conference*, pages 109–118. IEEE, 2002.

[17] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

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

[19] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[20] P. Leu, I. Puddu, A. Ranganathan, and S. Čapkun. I send, therefore I leak: Information leakage in low-power wide area networks. In *Proceedings of the 11th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 23–33. ACM, 2018.

[21] M. Lipp, D. Gruss, M. Schwarz, D. Bidner, C. Maurice, and S. Mangard. Practical keystroke timing attacks in sandboxed JavaScript. In *Computer Security – ESORICS 2017*, pages 191–209. Springer, 2017.

[22] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security '18)*, pages 973–990. USENIX Association, 2018.

[23] P. Maene, J. Götzfried, R. De Clercq, T. Müller, F. Freiling, and I. Verbauwhede. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Transactions on Computers*, 67(3):361–374, 2018.

[24] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 10:1–10:1. ACM, 2013.

[25] Microchip Technology Inc. MCP2515: Stand-alone CAN controller with SPI interface. http://ww1.microchip.com/downloads/en/DeviceDoc/MCP2515-Stand-Alone-CAN-Controller-with-SPI-20001801J.pdf, 2019.

[26] C. Miller and C. Valasek. A survey of remote automotive attack surfaces. *Black Hat USA*, 2014.

[27] C. Miller and C. Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015.

[28] M. R. Moore, R. A. Bridges, F. L. Combs, M. S. Starr, and S. J. Prowell. Modeling inter-signal arrival times for accurate detection of CAN bus signal injection attacks: a data-driven approach to in-vehicle intrusion detection. In *Proceedings of the 12th Annual Conference on Cyber and Information Security Research*, pages 1–4. ACM, 2017.

[29] H. Mun, K. Han, and D. H. Lee. Ensuring safety and security in CAN-based automotive embedded systems: A combination of design optimization and secure communication. *IEEE Transactions on Vehicular Technology*, 2020.

[30] J. Noorman. Sancus: Lightweight and open-source trusted computing for the IoT (source code). https://distrinet.cs.kuleuven.be/software/sancus/, 2017.

[31] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd USENIX Security Symposium (USENIX Security '13)*, pages 479–498. USENIX Association, 2013.

[32] J. Noorman, J. V. Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling. Sancus 2.0: A low-cost security architecture for IoT devices. *ACM Transactions on Privacy and Security (TOPS)*, 20(3):1–33, 2017.

[33] J. Noorman, J. T. Mühlberg, and F. Piessens. Authentic execution of distributed event-driven applications with a small TCB. In *Security and Trust Management*, pages 55–71. Springer, 2017.

[34] S. Nürnberger and C. Rossow. – vatiCAN – vetted, authenticated CAN bus. In *Cryptographic Hardware and Embedded Systems – CHES 2016*. Springer, 2016.

[35] A. Palanca, E. Evenchick, F. Maggi, and S. Zanero. A stealth, selective, link-layer denial-of-service attack against automotive networks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 185–206. Springer, 2017.

[36] A. I. Radu and F. D. Garcia. LeiA: A Lightweight Authentication Protocol for CAN. In *Computer Security – ESORICS 2016*, pages 283–300. Springer, 2016.

[37] T. Rosenstatter, C. Sandberg, and T. Olovsson. Extending AUTOSAR's counter-based solution for freshness of authenticated messages in vehicles. In *2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 1–109. IEEE, 2019.

[38] T. Ruffing, P. Moreno-Sanchez, and A. Kate. P2P mixing and unlinkable bitcoin transactions. In *Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSS '17)*. Internet Society, 2017.

[39] M. Schwarz, M. Lipp, D. Gruss, S. Weiser, C. Maurice, R. Spreitzer, and S. Mangard. KeyDrown: Eliminating software-based keystroke timing side-channel attacks. In *Proceedings of the 25th Annual Symposium on Network and Distributed System Security (NDSS '18)*. Internet Society, 2018.

[40] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 753–768. ACM, 2019.

[41] R. Strackx, P. Agten, N. Avonds, and F. Piessens. Salus: Kernel support for secure process compartments. *EAI Endorsed Transactions on Security and Safety*, 15(3):e1–e1, 2015.

[42] R. Strackx, J. Noorman, I. Verbauwhede, B. Preneel, and F. Piessens. Protected software module architectures. In *ISSE 2013 Securing Electronic Business Processes*, pages 241–251. Springer, 2013.

[43] R. Strackx and F. Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 2–13. ACM, 2012.

[44] R. Strackx, F. Piessens, and B. Preneel. Efficient isolation of trusted subsystems in embedded systems. In *International Conference on Security and Privacy in Communication Systems*, pages 344–361. Springer, 2010.

[45] Texas Instruments. Introduction to the Controller Area Network (CAN). *Application Report SLOA101*, pages 1–17, 2002.

[46] Texas Instruments. MSP430x1xx family: User's guide. http://www.ti.com/lit/ug/slau049f/slau049f.pdf, 2006.

[47] F. Turan and I. Verbauwhede. Propagating trusted execution through mutual attestation. In *Proceedings of the 4th Workshop on System Software for Trusted Execution*. ACM, 2019.

[48] US Department of Defense. Trusted computer system evaluation criteria. *DoD 5200.28-STD*, 1986.

[49] J. Van Bulck. VulCAN: Efficient component authentication and software isolation for automotive control networks (source code). https://distrinet.cs.kuleuven.be/software/vulcan/, 2017.

[50] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security '18)*, pages 991–1008. USENIX Association, 2018.

[51] J. Van Bulck, J. T. Mühlberg, and F. Piessens. VulCAN: Efficient component authentication and software isolation for automotive control networks. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 225–237. ACM, 2017.

[52] J. Van Bulck, J. Noorman, J. T. Mühlberg, and F. Piessens. Towards availability and real-time guarantees for protected module architectures. In *Companion Proceedings of the 15th International Conference on Modularity*, pages 146–151. ACM, 2016.

[53] J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, and F. Piessens. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1741–1758. ACM, 2019.

[54] J. Van Bulck, F. Piessens, and R. Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 178–195. ACM, 2018.

[55] A. Van Herrewege, D. Singelee, and I. Verbauwhede. CANAuth - a simple, backward compatible broadcast authentication protocol for CAN bus. In *ECRYPT Workshop on Lightweight Cryptography*, 2011.

[56] M. Wolf, A. Weimerskirch, and C. Paar. Security in automotive bus systems. In *Workshop on Embedded Security in Cars*. Bochum, 2004.

[57] Y. Yarom and K. Falkner. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security '14)*, pages 719–732. USENIX Association, 2014.

[58] X. Ying, G. Bernieri, M. Conti, and R. Poovendran. TACAN: Transmitter authentication through covert channels in controller area networks. In *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*, pages 23–34. ACM, 2019.

[59] S. Zander, G. Armitage, and P. Branch. A survey of covert channels and countermeasures in computer network protocols. *IEEE Communications Surveys and Tutorials*, 9(3):44–57, 2007.

[60] T. Ziermann, S. Wildermann, and J. Teich. CAN+: A new backward-compatible Controller Area Network (CAN) protocol with up to 16x higher data rates. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1088–1093. European Design and Automation Association, 2009.

Fiche masterproef

Student: Stien Vanderhallen

Titel: Robust Authentication for Automotive Control Networks through Covert Bandwidth

Nederlandse titel: Robuuste Authenticatie voor Regelnetwerken in Voertuigen via Verborgen Bandbreedte

UDC: 621.3

Korte inhoud:

Online presence has become a necessity in ever more aspects of modern day life, including transportation. In-vehicle networks, such as CAN, and the devices they connect, therefore are no longer out of reach for remote attackers. Suitable security measures exist for hardening embedded devices against such increased malicious activity, like cryptographic protocols for authenticating CAN traffic. However, those jeopardize the very nature of vehicular operations, as they incur significant amounts of extra bandwidth use and network latency, whereas this automotive context inherently requires strong guarantees on performance and timeliness.

A resource-neutral approach to securing CAN communication thus is due. Recent work derived an approach to that task from existing research on side-channel attacks, and repurposes the information carried by side channels from the leaking of sensitive data, to the covert transmission of security-enhancing payloads.

In an aim to further explore the feasibility of those efforts, this master's thesis investigates and assesses concrete opportunities for covert transmission in CAN, and aims to extend and/or improve existing security measures for embedded devices with the use of covert communication. The following three main contributions to this domain are made. First, a systematic overview of covert, and covert-like, bandwidth sources in CAN is constructed. Second, an extensive analysis and quantitative evaluation of timing based covert communication in CAN is executed, and supported by practical implementations. Third, this work proposes an extension to an existing security framework, which leverages covert bandwidth for enabling stronger availability guarantees of its message authentication scheme.

As such, this master's thesis makes the means that are available for covert communication in CAN concrete, evaluates their characteristics, and shows the substantial benefits they can offer when incorporated into existing security mechanisms.

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdoptie Veilige software

Promotoren: Prof. dr. ir. F. Piessens
Dr. J.T. Mühlberg

Assessoren: Ir. J. Van Bulck
Dr. B. Lagaisse

Begeleiders: Dr. J.T. Mühlberg
Ir. J. Van Bulck