VulCAN: Efficient Component Authentication and Software Isolation for Automotive Control Networks

Jo Van Bulck
imec-DistriNet, KU Leuven
jo.vanbulck@cs.kuleuven.be

Jan Tobias Mühlberg
imec-DistriNet, KU Leuven
jantobias.muehlberg@cs.kuleuven.be

Frank Piessens
imec-DistriNet, KU Leuven
frank.piessens@cs.kuleuven.be

ABSTRACT

Vehicular communication networks have been subject to a growing number of attacks that put the safety of passengers at risk. This resulted in millions of vehicles being recalled and lawsuits against car manufacturers. While recent standardization efforts address security, no practical solutions are implemented in current cars.

This paper presents VulCAN, a generic design for efficient vehicle message authentication, plus software component attestation and isolation using lightweight trusted computing technology. Specifically, we advance the state-of-the-art by not only protecting against network attackers, but also against substantially stronger adversaries capable of arbitrary code execution on participating electronic control units. We demonstrate the feasibility and practicality of VulCAN by implementing and evaluating two previously proposed, industry standard-compliant message authentication protocols on top of Sancus, an open-source embedded protected module architecture. Our results are promising, showing that strong, hardware-enforced security guarantees can be met with a minimal trusted computing base without violating real-time deadlines under benign conditions.

CCS CONCEPTS

• **Security and privacy → Embedded systems security**; **Distributed systems security**; *Key management*; *Security protocols*; *Domain-specific security and privacy architectures*;

KEYWORDS

Automotive security, CAN, Protected module, Trusted computing

ACM Reference Format:
Jo Van Bulck, Jan Tobias Mühlberg, and Frank Piessens. 2017. VulCAN: Efficient Component Authentication and Software Isolation for Automotive Control Networks. In *Proceedings of 33th Annual Computer Security Applications Conference (ACSAC'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3134600.3134623

1 INTRODUCTION

Today's road vehicles are controlled by a growing number of interconnected Electronic Control Units (ECUs) that jointly operate the car's behavior and safety-critical functionality. Abstractly speaking, these ECUs exchange messages and form an extensive distributed system that interprets sensor readings and reacts to important events by triggering the relevant actuators, e.g., brakes, airbags, or steering gear. In practice, however, these software interactions are susceptible to adversarial manipulations, as automotive communication standards, including the widely used Controller Area Network (CAN), typically provide no message authenticity guarantees at all. This lack of security becomes untenable with the advent of networked infotainment systems that expose safety-critical components to remote attackers, threatening the safety of passengers and other road users alike. The past years have indeed seen a steady stream of automotive attack demonstrations [8, 10, 19, 22, 27, 28, 50], whose risks have been acknowledged by emerging industry standards [1, 40] that encompass authentication and software security.

Recent research [34, 38] studies CAN authentication protocols compliant to these standards. Importantly, however, these protocols are not deployed in existing vehicles, and experimental implementations lack the performance needed to satisfy stringent automotive real-time constraints. Moreover, existing proposals only address software security and key confidentiality with respect to adversaries that do *not* control code execution on targeted ECUs. Yet, it has been shown repeatedly that attackers are capable of influencing code execution or even running their own code on individual ECUs to compromise key material or messages before or after authentication [22, 28]. As such, we consider threat models that assume only network communication can be interfered with rather naive.

This paper introduces VulCAN,[1] an efficient approach to implement secure distributed automotive control software on lightweight trusted computing platforms. Our approach is distinguished from previous work by relying on trusted hardware and a minimal software Trusted Computing Base (TCB), suitable for thorough validation and formal verification. Specifically, we rely on hardware-enforced memory protection primitives to *isolate* critical software components on participating ECUs, and to harden them against code-abuse attacks such as return-oriented programming [9]. We furthermore show how to securely interface such "vulcanized" software components with peripheral I/O devices, and dynamically *attest* their integrity across the untrusted vehicle network. Finally, we leverage the hardware-level cryptographic primitives commonly found in trusted computing architectures to *efficiently* implement message authentication for the CAN bus, which is an essential condition to preserve real-time deadlines when not under attack.

At the hardware level, our VulCAN prototype is based on Sancus [33], an open-source protected module architecture that extends an embedded openMSP430 processor. At the application level,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACSAC'17, December 2017, San Juan, Puerto Rico, USA

© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.

ACM ISBN 978-1-4503-5345-8/17/12…$15.00
https://doi.org/10.1145/3134600.3134623

[1] Vulcan, the ancient Roman deity of fire and smithery, was entrusted with the forgery of weapons and armor for gods and heroes.

we implement and evaluate two recently published AUTOSAR-compliant CAN authentication protocols [34, 38], which we compare in terms of performance overhead, TCB size, and offered security. We provide a detailed description of our FPGA-based setup, and explain how to securely shield an off-the-shelf Seat instrument cluster via a CAN security gateway. To encourage future research on the evaluation of security architectures and protocols for automotive control networks, we make our hardware designs, a simulator, and the deployed software stack publicly available at https://distrinet.cs.kuleuven.be/software/vulcan. In summary, we make the following contributions:

- We explore the use of embedded trusted computing technology in automotive control networks to improve on software security, and to reduce message authentication overhead.
- To our knowledge, we present the first comparison of CAN authentication protocols in terms of security guarantees, performance results and TCB size. As part of our analysis, we discovered a novel and practical replay attack against vatiCAN's [34] frequent nonce renewal scheme.
- We discuss practical implications of our approach with respect to vehicle maintenance and key management, and introduce a security gateway approach to transparently shield unmodified legacy devices on the CAN bus.
- We present an extended application scenario that demonstrates our security guarantees, and may serve as a basis for future automotive security research in general.

2 BACKGROUND

2.1 Controller Area Network Authentication

The CAN bus is the most commonly used broadcast communication medium in modern automobiles. From an application's perspective, a CAN message consists of an 11-bit ID, followed by an optional 18-bit extended identifier, and up to 8 bytes of data payload (see Fig. 1). Dedicated transceiver hardware chips implement a protocol for message acknowledgement and bus arbitration for sending/receiving data frames. Specifically, CAN requires a fixed data transmission rate, and allows recessive bits (one) to be overwritten by dominant bits (zero) during transmission. Message acknowledgement can thus simply be implemented by overwriting the ACK bit at the end of the data frame in real-time. Likewise, to solve bus arbitration, CAN transceivers are required to continue listening on the bus while sending the message ID at the beginning of the data frame, and to back off when one of their ID bits has been overwritten. This scheme ensures that messages with lower IDs effectively have higher priorities. Finally, each CAN frame features a 16-bit Cyclic Redundancy Check (CRC) field to detect transmission errors.

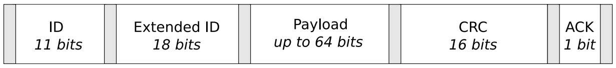

Figure 1: Extended data frame standardized by CAN 2.0B.

CAN was originally developed in 1983, when remote attackers were of no concern, and does not provide any form of message authentication. Any ECU connected to the network can spoof messages with arbitrary sender ID and payload, which forms the basis of many attacks [10, 22, 27, 28]. As a response, the AUTOSAR [1] standardization body published industry guidelines for backwards-compatible message authentication in vehicular networks. To the best of our knowledge, only two AUTOSAR-compliant CAN authentication protocols have been proposed to date: LeiA [38] and vatiCAN [34], where only the latter was actually implemented and evaluated in terms of performance. Both protocols require a symmetric pre-shared cryptographic key at sender and receiver ECUs to compute a Message Authentication Code (MAC) over the message ID, payload, and a monotonically incrementing nonce to protect against replay attacks. To deal with CAN payload length limitations and to ensure backwards compatibility, 64-bit MACs are sent as a separate message with a different ID, following the actual message to be authenticated.

2.2 Embedded Protected Module Architectures

Security Primitives. The VulCAN approach provides a notion of authenticated CAN components, which entails integrity and authenticity of critical messages, both while in transit over the untrusted vehicular network, as well as during processing by isolated software components on the participating ECUs. For this, VulCAN builds on the trusted computing features provided by Protected Module Architectures (PMAs) [24, 42], a new brand of security architectures that support the secure and isolated execution of critical software modules with a minimal TCB. Hardware implementations of PMAs for higher-end systems, notably Intel's Software Guard Extensions (SGX) [25] and ARM's TrustZone [3], as well as several lightweight prototypes for embedded application domains [6, 11, 21, 31] have been presented. These proposals have in common that they provide Protected Modules (PMs) with strong confidentiality and integrity guarantees regarding their internal state, without having to trust *any* other software executing on the platform, including traditionally privileged operating system code.

Modern PMAs offer a number of security primitives to (*i*) isolate PM software protection domains, (*ii*) attest the integrity of an isolated PM, and (*iii*) facilitate key management for secure communication. In the following, we elaborate on these features as provided by the Sancus architecture, but it should be noted that the VulCAN solution can in principle be implemented on any platform that provides the above security primitives.

Sancus. Sancus [31, 33] is a fully open-source[2] embedded PMA and hardware-only TCB that extends the memory access logic and instruction set of a low-cost, low-power openMSP430 [13] microcontroller. Sancus supports multiple mutually distrusting software components that each consist of two contiguous memory sections in a shared single-address-space. A hardware-level program counter-based access control mechanism [43] enforces that a PM's private *data section* can only be accessed by its corresponding *code section*, which can only be entered through a single *entry point*. Sancus' generic memory isolation primitive can furthermore be used to provide secure driver PMs with exclusive ownership over Memory-Mapped I/O (MMIO) peripheral devices that are accessed through

[2] https://distrinet.cs.kuleuven.be/software/sancus/

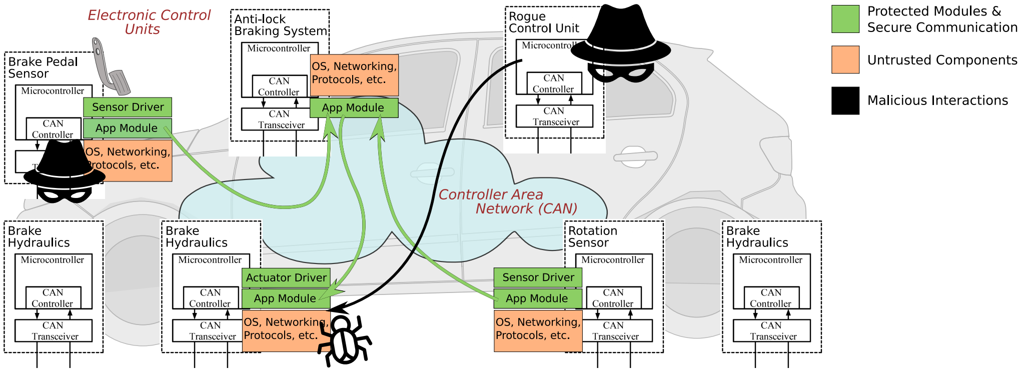

Figure 2: An example CAN network scenario to illustrate basic attacks and the security guarantees offered by our approach.

the address space. Since Sancus modules only feature a single contiguous private data section, however, secure I/O on Sancus platforms requires the use of small driver modules entirely written in assembly code, using only registers for data storage [32].

Sancus also provides hardware-level authenticated encryption, key derivation, and key storage functionality by extending the CPU with a cryptographic core. Specifically, to implement secure communication and local/remote attestation, Sancus employs a three level key hierarchy. At the root of this hierarchy, every Sancus-enabled CPU contains a unique hardware-level *node master key* only known by the infrastructure provider that owns the embedded computing node. Infrastructure providers furthermore assign unique identifiers to independent software vendors who are to install a PM on a particular node. These vendor identifiers are used in the second level of the key hierarchy to derive *vendor keys* from the node master key. Finally, *module keys* are derived from a vendor key using the *module identity*, which is composed of the contents of a PM’s code section and the load addresses of both its sections. The security of Sancus’ attestation guarantees is based on the unforgeability of these module identities. Upon enabling PM isolation, after the module has been loaded by untrusted system software, a Sancus-enabled processor computes the corresponding module key, and stores it in a hardware-level protected storage area for exclusive use by the PM. The benign software vendor in possession of the vendor key can compute the module key independently. The *use* of a certain module key (e.g., by creating a MAC over a fresh challenge) thus suffices to attest that a PM with a specific code section has been loaded on a specific device, with isolation enabled and without having been tampered with. The symmetric module key can now be used to establish a confidentiality and integrity-preserving communication channel between a remote software vendor and its newly deployed PM. For this, PMs can benefit from Sancus’ efficient cryptographic hardware primitives by means of dedicated processor instructions for authenticated encryption/decryption using either the module key, or an in-memory key provided by software. Likewise, Sancus supports *secure linking* of two PMs residing on the same computing platform by means of novel instructions for caller/callee authentication.

Sancus finally comes with a dedicated C compiler that automates the process of PM creation, and hides low-level concerns such as secure linking, private call stack switching, and multiplexing user-defined entry functions through the single physical entry point.

3 VULCANIZED CAN COMPONENTS

In this section, we present our approach to secure, authenticated execution of distributed automotive applications that run on multiple ECUs and communicate via an untrusted vehicular network. Our approach compartmentalizes such an application into a small group of trustworthy *authenticated software components*, i.e., isolated PMs. In contrast to previous work, VulCAN supports multiple mutually distrusting components on a single ECU, such that – apart from classical denial-of-service attacks – code executing in one component can never be adversely impacted by any other software executing on the same or another processor in the CAN network. We depict an exemplary CAN network in Fig. 2, where an authenticated *trusted path* is set up between a software component that senses a braking pedal and wheel rotations, over an Anti-lock Braking System (ABS) component, all the way to a brake hydraulic actuator component. In short, VulCAN guarantees that any braking action by the actuator component can be traced back to the authenticated software components that originally sensed the driver’s braking pedal and the wheel speed.

In the following, we first introduce our attacker model, and precisely formulate the requirements for practical and trustworthy vehicle component security. Finally, we elaborate on how to achieve the desired security guarantees.

3.1 Attacker Model

The adversary’s goal is to impersonate a protected component connected to the CAN network. More specifically, she wants to trick a receiver component into accepting a CAN message with chosen ID and payload, as if it originated from a valid sender component. We protect against attackers with two important capabilities, namely arbitrary message manipulation, and arbitrary code execution.

Arbitrary Message Manipulation. Analogous to previous work [15, 18, 34, 38, 46], we consider an attacker without physical access, but who has successfully gained remote access to the car’s internal network. She might for instance infiltrate an ECU over one of the numerous wireless interfaces in a modern automobile [8, 10]. This gives her the ability to (*i*) broadcast her own CAN messages with arbitrary ID and payload, (*ii*) observe and record all traffic on the CAN bus, and (*iii*) intentionally destroy or modify packets as they pass by. Due to the inherent broadcast nature of the shared CAN bus, denial-of-service attacks are ultimately out-of-scope.

Arbitrary Code Execution. Existing CAN authentication proposals (cf. previous paragraph) as well as vehicular hardware security modules [49] explicitly assume an adversary who has *not* compromised the ECU for which she intends to fake packets. It has been repeatedly shown, however, that automotive software and update mechanisms are vulnerable to a wide range of attacks [10, 22] that allows an adversary to execute her own code on individual ECUs. We therefore model an attacker with arbitrary code execution on *every* ECU in the network. This means she can compromise *all* software (including privileged operating system support software), except for the trusted authenticated software components that are explicitly protected by Sancus.

3.2 Problem Statement

We precisely state the requirements and challenges for trustworthy component authentication on an unmodified CAN bus. We distinguish between *protocol* and *system requirements.* The first are reflected in the design of the CAN authentication protocol, whereas the latter concern the actual ECUs that implement and participate in the authenticated communication. Closely following previous research [34, 38], we identified the following protocol requirements:

P1: Message Authentication. A receiver component should get a strong guarantee that a message with specified ID and payload was indeed sent by a trusted sender component.

P2: Lightweight Cryptography. The use of public key cryptography is ruled out, for typical ECUs are severely constrained in computational power and storage space.

P3: Replay Attack Resistance. Packet loss should be anticipated, but the authentication scheme should be immune to replay attacks, even when a large amount of traffic was captured.

P4: Backwards Compatibility. The authentication scheme must be compatible with existing off-the-shelf CAN transceiver chips. Legacy unmodified applications without authenticated communication should continue to function.

While the above requirements are mostly met by at least two recent CAN authentication protocols [34, 38], we argue that a *practical* and *trustworthy* solution should furthermore offer the following system-level guarantees, which are not achieved by state-of-the-art authentication schemes:

S1: Real-Time Compliance. While denial-of-service attacks are explicitly out-of-scope, the CAN bus is widely used to initiate safety-critical functionality such as brakes and steering, necessitating a fast authentication scheme that preserves stringent real-time deadlines when not under attack.

S2: Component Isolation. The integrity of message processing and authentication algorithms, and the confidentiality of key material should be protected against an attacker with arbitrary untrusted code execution on participating ECUs.

S3: Component Attestation. When starting her car, and while driving, the motorist should get a strong guarantee that critical software components have been loaded on specific ECUs with isolation enabled (S2), and without having been tampered with.

S4: Dynamic Key Update. The system should support secure key provisioning at runtime, and allow broken ECUs to be replaced by a distrusted automobile repair shop. Extracting keys in one ECU should not compromise message authentication (P1) for uninvolved components.

S5: Secure Legacy ECU Integration. Automotive suppliers cannot be expected to adopt hardware/software changes immediately on all ECUs that require authentication. It should be possible to transparently *shield* unmodified legacy ECUs as a transition measure.

3.3 Authenticated CAN Bus

Before explaining how we achieve the strong system-level software security guarantees (S1–S5), we first elaborate on our protocol requirements (P1–P4). We carefully selected the latter such that they can be fulfilled by any AUTOSAR [1] compliant authentication protocol. The VulCAN approach is inspired by, and remains largely compatible with, two such proposals: vatiCAN [34] and LeiA [38].

Message Authentication (P1–P4). As explained in the background section, CAN follows a publish-subscribe broadcast model where prioritized 11-bit IDs (optionally enlarged with an 18-bit extended identifier) are associated with a data payload of up to 64-bit. However, CAN does not reserve bandwidth for authentication metadata. Our requirement for backwards compatibility (P4) with existing CAN transceivers rules out previous research [15, 46] that leverages the extra bandwidth provided by the CAN+ extensions (cf. Section 7). Following the approach of vatiCAN and LeiA, we therefore decouple the authentication metadata from the actual message to be authenticated. More specifically, we first broadcast the authenticated message in plain text, and afterwards construct and transmit authentication data on a different CAN identifier, to which only authentication-aware receiver components subscribe. This scheme ensures compatibility with unmodified legacy applications that do not rely on message authentication (P4).

To satisfy requirements (P1, P2), we associate a symmetric 128-bit cryptographic key with each authenticated CAN identifier. Like vatiCAN, but in contrast to LeiA, our design allows multiple IDs to share the same key to reduce memory consumption costs, if desired (P2). Valid sender/receiver components use the key to construct a 64-bit Message Authentication Code (MAC) over both ID and payload (P1), including a monotonically increasing counter to protect against replay attacks (P3). More specifically, to authenticate a message with identifier i, payload p, and counter c_i, we compute:

$$m = MAC(key_i,\ (i \mid p \mid c_i)) \tag{1}$$

Note that, when key_i is truly unique per identifier (as in the LeiA specification), i should not be included in the above message, since it is implicitly authenticated through the key. After calculating the MAC, the sender transmits it as the payload of a separate CAN message with ID $i + 1$. This scheme practically avoids priority inversion problems, for CAN identifiers are also used as priority indicators during bus arbitration. In case, however, that the CAN identifier $i + 1$ is already in use by the legacy application (P4), a different application-specific authentication ID has to be selected.

Nonce Initialization (P3, P4). Requirement (P3) demands that replaying a previously authenticated message should result in that message being discarded by the receiver. To this end, our authentication scheme Eq. (1) includes a monotonically increasing counter or nonce c_i as a source of freshness in the MAC computation for a message with identifier i. Identical nonce values should never be reused under the same key and associated data, however, which provides us with two important challenges: (*i*) nonce initialization on platform reset or c_i counter overflow, and (*ii*) nonce resynchronization on packet loss.

A common way to address the first challenge, nonce initialization, is the use of short-term *session keys* [38, 46]. At system boot time, or whenever the session counter c_i overflows, both parties generate a fresh session key based on the long-term pre-shared secret and a larger *epoch counter*. With this scheme, individual session counters c_i can safely start from zero. Session keys thus reduce the burden of secure nonce management for multiple sessions to secure persistent storage of a single long-term key and epoch counter per connection per ECU. Our final key provisioning scheme (S4) further relaxes this requirement to secure persistent storage capacity on a *single* trusted ECU, by leveraging trusted computing features as discussed in more detail in Section 3.4.

Nonce Resynchronization (P3, P4). Receiver ECUs have to deal with packet loss as they may miss a message or authentication frame, for instance when in sleep mode or under heavy network load. Due to the periodic broadcast nature of typical CAN applications, legacy receivers can easily recover from such failures. On an authenticated CAN bus, subsequent MAC comparisons will fail if sender and receiver nonces get out-of-sync. Observe that nonce values c_i are not confidential, however, and that receivers can accept MACs generated with any nonce that is strictly higher than the previously authenticated nonce value. As such, packet loss could be anticipated by including the plain text sender nonce as part of the authentication payload, but our 64-bit MACs (P1) leave no free space in a standard CAN frame. Depending on the application, some unused parts of the message payload might be used to encode (the least significant bits of) the nonce, as also suggested by AUTOSAR [1]. As a more general solution, VulCAN leaves nonce resynchronization choices to the underlying protocol being used.

Our vulcanized LeiA [38] implementation encodes relatively small 16-bit nonces in the extended CAN identifier field, and reestablishes session keys on nonce counter overflow. This allows receiver components to easily recover after missing one or more authenticated messages. Furthermore, to be able to recover from longer-term network failures, a genuine receiver can send a dedicated error frame that prompts a sender to broadcast its current epoch counter value. Note, however, that LeiA might partially break legacy applications (P4) that rely on the availability of extended CAN identifiers. For example, the vehicle instrument cluster discussed in Section 6 utilizes at least one extended CAN identifier. Our experimental evaluation (Section 5) also reveals that LeiA's use of extended identifiers also comes with a performance penalty.

An alternative approach to dealing with packet loss, is provided by the vatiCAN [34] protocol that uses larger 32-bit nonces, but relies on a trusted global Nonce Generator (NG) component to periodically reset nonces in the entire network. Specifically, every few milliseconds, NG broadcasts a randomly chosen value to be used by all participating ECUs as the new initial value for all counters c_i. NG is assumed to use a modified CAN transceiver, equipped with hypothetical (P4) hardware-assisted spoofing prevention, such that an attacker cannot inject arbitrary nonce renewal requests. We show in Appendix A, however, that vatiCAN's frequent random nonce renewal design is vulnerable to advanced replay attacks based on the (generalized) birthday problem from probability theory. Depending on the targeted application, our attacks allow an adversary to start replaying authenticated messages after recording as little as 30 minutes of broadcast CAN traffic. Our vulcanized vatiCAN implementation therefore omits global nonce generation, and properly uses session keys, as discussed above, to ensure the same c_i value is *never* reused under the same cryptographic key.

Efficient MAC Computation (S1). Recall that the payload of an authenticated CAN message is sent first, before the actual authentication frame containing the MAC. The vatiCAN [34] authors leverage this property to compute MACs *in parallel* at both the sender and receiver ECUs. The sender first broadcasts message m to be authenticated, and subsequently computes and sends the corresponding MAC_s. As soon as m has arrived, the receiver starts computing MAC_r, which is afterwards compared to MAC_s to authenticate m. While such parallel MAC computation cuts authentication overheads by half, current software-only solutions still require several milliseconds to compute a single 64-bit MAC [34].

Our Sancus-based VulCAN implementation on the other hand, computes Eq. (1) using Sancus' [31, 33] hardware-level authenticated encryption primitive. We configured our Sancus cores with 128 bits of security, resulting in 128-bit message authentication tags. To overcome CAN payload length limitations, we truncate the resulting MAC by discarding the eight least significant bytes. This approach adheres to the relevant AUTOSAR specification that recommends to "always use a key length of at least 128 bit", but also states that "a MAC truncation can be beneficial. [...] In general, MAC sizes of 64 bit and above are considered to provide sufficient protection" [1]. An important advantage of using hardware-level cryptography is that MAC computation times decrease with an order of magnitude, as further explored in the evaluation section. We consider such acceleration crucial to meet stringent real-time automotive safety requirements in benign conditions (S1).

3.4 Component Isolation and Authentication

In this section, we elaborate on our system-level requirements (S2–S5) that transform a conventional entangled vehicle network into a minimal set of small and robust application components that mutually authenticate and trust each other.

Software Isolation and Key Storage (S2). The security of any authentication scheme critically relies on maintaining key confidentiality. In this respect, previous automotive security proposals [15, 34, 38, 46] invariably assume uncompromised sender/receiver ECUs, neglecting advanced attacker capabilities. Our strengthened attacker model, on the other hand, considers adversaries with arbitrary ECU code execution such that any cryptographic key material that is not explicitly protected against untrusted system software can be trivially extracted. We therefore leverage PMA protection

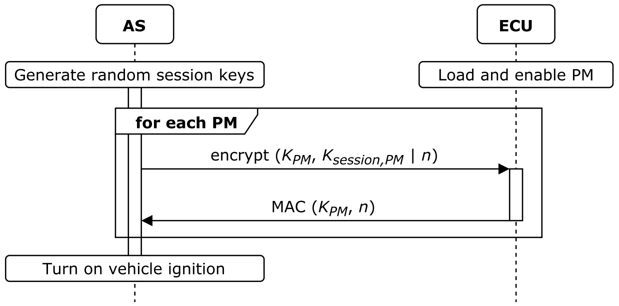

Figure 3: Load-time attestation and session key provisioning protocol between trusted Attestation Server (AS) and individual ECUs hosting PM software components.

to create isolated "safe harbors" for constructing and processing authenticated CAN messages.

VulCAN reduces the TCB by explicitly distrusting management software, including the CAN driver. Instead, authenticated CAN messages are constructed and verified *within* a Sancus PM, such that connection keys never leave the private data section. Critical application software that reacts upon authenticated messages is furthermore included in the PM protection domain, so as to assure its untampered execution (S2). Moreover, the protected application logic can integrate dedicated driver PMs to securely sense or actuate I/O devices, as explained in Section 2.2. As such, a distributed *trusted path* can be set up from a brake pedal sensor all the way to a brake actuator component, without having to trust *any* of the unprotected support software that processed the message on sender/receiver ECUs. The unprotected software components involved in relaying the message may be strictly necessary from a functionality and availability perspective, yet, they are not involved in implementing the security properties of the application. Thus, our solution effectively guarantees that any braking action at the receiver PM can be traced back to the sender PM that originally sensed the driver's brake pedal. As such, vulcanized CAN components significantly reduce the TCB, making them more suitable for safety and security certification, and well within reach of formal software verification techniques [37].

Software Attestation and Key Provisioning (S3, S4). The above guarantees are only maintained *after* PM authenticated components have been correctly isolated and provisioned with symmetric cryptographic keys. Since PMs are not persisted across reboots, an attacker initially has full control over every ECU in the network when the car is turned on. This confronts us with three closely related challenges: how to (*i*) attest the integrity of critical distributed software components, (*ii*) establish session keys over the untrusted CAN bus, and (*iii*) replace broken ECUs in an untrustworthy automobile repair shop.

To overcome these challenges, the VulCAN design includes a trusted Attestation Server (AS) that engages in a remote attestation protocol with individual PMs. Specifically, AS possesses the Sancus module-specific 128-bit key K_{PM} of every authenticated software component in the network, such that it can get assurance that a particular PM has been isolated on a specific ECU without having been tampered with (cf. Section 2.2). When starting the vehicle, untrusted system software on the participating ECUs is in charge of loading PM components, and the secure PMA hardware computes the corresponding module-specific keys based on the PM's identity. A simple challenge-response attestation protocol to prove possession of the expected K_{PM} thus suffices for AS to establish that PM has been properly loaded. To properly protect against replay attacks (P3), however, AS should make sure to always include a monotonically incrementing attestation nonce n in the challenge.

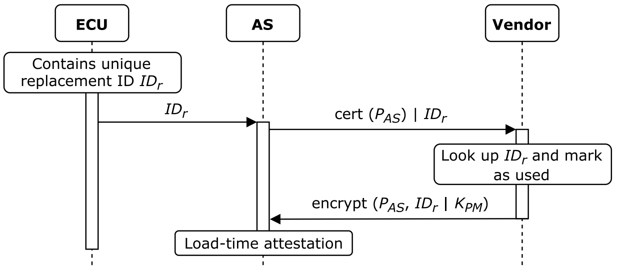
ECU
AS
Vendor
Contains unique replacement ID ID_r
ID_r
cert (P_{AS}) | ID_r
Look up ID_r and mark as used
encrypt (P_{AS}, ID_r | K_{PM})
Load-time attestation

Figure 4: Protocol for integrating a new ECU into an existing control network. The in-vehicle Attestation Server (AS) is shipped with a certificate of its public key.

Regarding the second challenge, session key distribution, we leverage the observation that distributed load-time attestation already necessitates communication between AS and every PM. We thus let AS generate random 128-bit session keys for all connections whenever starting the car, and modify the above attestation protocol to include dynamic key provisioning (S4), as illustrated in Fig. 3. AS first broadcasts an encrypted challenge for every PM, containing the newly generated session key as well as the global nonce value n. Note that 128-bit session keys wont fit within a single CAN message, and the load-time challenge therefore needs to be spread across multiple successive CAN frames. We encrypt the challenge with K_{PM} to ensure that session keys are only provisioned to properly isolated and unmodified PM. The attestation process is finally completed when a PM responds with a MAC over n, using K_{PM}. We propose to connect the vehicle ignition switch directly to AS, such that the car refuses to start when load-time attestation fails (S3). This way, the driver is given a strong assurance that any critical event (e.g., braking) while driving can only be caused by a legitimate sender component (e.g., braking pedal).

To address the final requirement, ECU replacement, we equip AS with asymmetric cryptography. Note that this does not violate our requirement for lightweight ECUs (P2), as every car only needs a *single* authentication server component that would typically be deployed on one of the higher-end vehicle processors. Specifically, we assume AS is shipped with a certificate of its public key, signed with the private key of the legitimate car vendor. Figure 4 outlines the protocol. When a new ECU joins an existing control network, it initially broadcasts a unique *replacement identifier* ID_r. Upon receiving ID_r, AS sends its public key certificate and the new replacement identifier to the remote car vendor over the untrusted automobile repair shop's network connection. The car vendor acts as the trusted *infrastructure provider* in the Sancus key hierarchy (Section 2.2), and thus possesses the node master keys of all ECUs it produced. Upon receiving ID_r and a valid AS certificate, the vendor computes the Sancus module-specific key K_{PM} for any authenticated software components to be executed on that ECU, and uses AS's public key to send them securely back to the car. AS can now proceed with the load-time attestation protocol for the newly joined ECU over the local CAN bus, as depicted in Fig. 3.

To prevent an attacker from acquiring the module key K_{PM} herself, the remote car vendor should *mark* used part identifiers ID_r, and never hand out K_{PM} to a different vehicle attestation server. Please also note that AS stores all module keys for subsequent attestations, such that public key cryptography and a network connection are only rarely needed, when upgrading the car. In this respect, AS can reset its module-specific attestation nonce n after every PM software update, since K_{PM} depends on the code section.

Shielding Legacy ECUs (S5). VulCAN requires hardware and software changes for critical ECUs, whereas part suppliers are typically restrained to quickly implement such changes in the heterogeneous automotive landscape. In this regard, previous software-only solutions [34, 38] remain backwards compatible with, but cannot provide any security guarantees for ECUs that do not adopt the required software changes. We therefore present a transition mechanism that can be used to securely *shield* unmodified legacy ECUs from an untrusted CAN bus.

Intuitively, we position a Sancus-enabled gateway ECU in front of the legacy ECU, and perform the necessary authentication transparently on behalf of the shielded legacy component(s). This basic idea is similar to the already existing CAN gateways that separate high-speed from low-speed traffic in modern automobiles. It has been repeatedly shown [10, 22], however, that existing such CAN gateways can easily be bridged by exploiting or reprogramming their software. Our approach on the other hand relies on PMAs, as outlined above, to establish a minimal TCB on the gateway ECU. More specifically, the security gateway participates in message authentication and software attestation protocols on the untrusted CAN network as usual, but makes sure to only forward successfully authenticated messages over the *trusted private CAN bus*. To prevent an attacker in control of untrusted software on the gateway ECU from forwarding traffic on the private CAN bus herself, we deploy a secure I/O driver PM that takes exclusive ownership over the CAN transceiver device. This of course implies that the protected CAN driver code becomes part of the TCB, which we consider acceptable given that the CAN driver needs to be trusted on the gateway ECU only, and can be considerably simplified when forwarding packets.

4 SECURITY ANALYSIS

Protocol Requirements (P1, P3). We first evaluate the protocol-level message authentication and replay protection requirements for a traditional adversary that controls the network, but not the software on sender/receiver ECUs. In this case, the security argument reduces to the security of the underlying MAC scheme. According to Eq. (1), an adversary could collect a legacy CAN identifier and plain text payload, plus its corresponding MAC and non-secret nonce c_i to try and brute-force all possible 128-bit session keys. Finding the correct session key would require 2^{127} MAC evaluations on average, which is considered infeasible even for extremely motivated adversaries. Alternatively, the attacker could try to correctly guess the smaller 64-bit MAC output. Note, however, that the birthday paradox for finding an *arbitrary* collision does not apply here, for the adversary's goal is to forge a message with chosen ID, nonce, and payload. As such, the probability to correctly guess the MAC output is 2^{-63}. To verify her guess, however, the attacker would have to interact online with the valid receiver. The available time frame for such an attack is reasonably limited, for nonce counters are incremented after every legitimate message and session keys are refreshed on counter overflow. The same property finally ensures proper replay attack resilience.

Note that the security of the LeiA [38] CAN authentication protocol has been formally proven, under the MAC unforgeability assumption. We present a practical replay attack against the vatiCAN [34] protocol, however, in Appendix A. Being based on the birthday paradox, the attack abuses the increased probability of nonce reuse due to vatiCAN's frequent random nonce renewal scheme. As discussed above, our vulcanized vatiCAN implementation therefore omits the global nonce generator.

System Requirements (S2–S5). We first define the in-vehicle TCB[3] for authenticated CAN components more precisely. As with any execution platform, the processor is ultimately trusted, which renders the security argument for VulCAN's system requirements inevitably specific to the implementation platform (in our case the minimalistic secure Sancus [31, 33] hardware). We furthermore trust the relevant authenticated software components themselves; in the case of braking for instance, only those PMs that participate in the creation and processing of that event are considered trusted. We expect the size of these modules to be within reach of formal verification [37] and secure compilation [36] techniques. Finally, we trust the global vehicle attestation server introduced above. That is, we assume secure, tamper-proof storage capacity and code execution on the AS node. Importantly, unlike previous CAN security proposals [34, 38, 46], our solution necessitates tamper-resistant persistent storage on the AS node *only*. We thus relax the requirements for lightweight ECUs, and move the burden of long-term cryptographic key and nonce storage to a single platform that can be more easily maintained. Specifically, we advise the use of widely available higher-end trusted execution technology such as ARM TrustZone [3] or Intel SGX [25] on the processor hosting AS.

The security of distributed software attestation (S3) follows from the correctness assumption of AS and the Sancus hardware. For Sancus' hardware-level key derivation scheme ensures that the expected K_{PM} is only available to a correctly isolated PM on a specific ECU, and AS includes a persistent counter n in the attestation challenge to establish a fresh guarantee every time the driver starts her car. Likewise, the security of isolated execution and key secrecy (S2) reduces to the correctness of the Sancus hardware. Please note that our adversary model considers physical attackers that exploit hardware side-channels to extract cryptographic key material out-of-scope. We do assume that all ECUs are shipped with independent node master keys, however, such that extracting keys from one ECU does not affect uninvolved components (S4). Our Sancus-based implementation is furthermore immune to

[3] Note that the development environments of the car vendor and part suppliers are isolated from the in-vehicle network, and thus not within reach of the attacker.

remotely exploitable software side-channels, since keys are never directly processed by software, and Sancus uses a constant-time hardware implementation of SpongeWrap [4] with spongent [5] as the underlying hash function.

The security of our session key provisioning scheme (S4) follows from the security of the attestation protocol, on which it piggybacks. Likewise, the security of the ECU replacement protocol between the vehicle attestation server and the remote car vendor entirely depends on the correctness of AS. As long as the adversary does not get hold of AS's private key, she never gets to know K_{PM} by observing network traffic. An attacker that actively interacts with the remote car vendor does not learn K_{PM} neither, since the vendor is required to keep track of assigned replacement IDs on a first-come, first-served basis that ensures keys are never retransmitted to a different attestation server. The only leverage left for an adversary in this scheme, is to launch a denial-of-service attack by claiming all replacement IDs. Such availability issues are out of the scope of our work, but can easily be mitigated at the application level. The car vendor could for instance engage in a complementary protocol with the automobile repair shop to establish the legitimacy of the replacement part to be installed, before actually assigning ID_r and handing out K_{PM} to the vehicle attestation server AS.

Finally, the security of shielding unmodified legacy ECUs (S5) with a Sancus-enabled gateway follows from the protocol-level security guarantees (P1, P3), plus authenticated component isolation (S2) and attestation (S3). Specifically, since the CAN driver itself executes in a PM in this case, the gateway can guarantee that only legitimate events (i.e., properly authenticated messages on the public bus) trigger CAN forwarding on the private bus.

5 EXPERIMENTAL EVALUATION

We fully implemented the vatiCAN [34] and LeiA [38] specifications, but leave the implementation of the attestation server as future work. We can only directly relate our approach to the previously reported software-only vatiCAN evaluation, however, for our work presents the first actual implementation of LeiA. In this section, we evaluate our vulcanized vatiCAN/LeiA implementations in terms of runtime overhead, memory footprint, and TCB size. Note that we do not evaluate bus congestion due to added CAN frames, as this does not yield interesting results beyond what is reported by vatiCAN, i.e., a modest 3% increase.

All experiments were conducted on a testbench featuring six Xilinx Spartan-6 FPGAs, each synthesized with a Sancus-enabled OpenMSP430 core [13, 33] running at 20 MHz. With the given clock speed, 1 CPU cycle corresponds to 50 ns, and 10,000 cycles correspond to 0.5 ms. Each Sancus core was configured to provide 128 bits of security. Analogous to previous research [34], we interfaced our ECUs over SPI with widely used off-the-shelf Microchip MCP2515 CAN transceiver chips on a common bus speed of 500 kBit/s. All source code was compiled with the Sancus C compiler based on LLVM/Clang v4.0.0 with optimizations set for size (`-Os`).

5.1 TCB Size and Memory Footprint

We implemented vulcanized vatiCAN/LeiA as a small C library that leverages Sancus' hardware-level authenticated encryption primitives, and that can optionally be included in a protected Sancus PM. Importantly, when the library is included in a PM, the remainder of the software stack remains explicitly *untrusted* regarding MAC computation integrity and confidentiality of key material. That is, the TCB encompasses *only* the vatiCAN/LeiA library and any application-specific protected message processing code. In this regard, we measured 212 lines of trusted source code for LeiA, and only 147 lines for vatiCAN, using the `sloccount` utility [47] and excluding debug code. Regarding the unprotected software stack, even our elementary CAN driver alone already requires 322 lines of code. In comparison, we measured over 670 lines of code for the popular CAN bus shield for Arduino devices [41], and established embedded operating systems such as Contiki or FreeRTOS exceed several tens of thousands of lines of code. As such, vulcanized CAN authentication modules significantly reduce the TCB, making them more manageable in security and safety validation efforts.

Table 1: Overhead to send an (authenticated) CAN message with/without Sancus encryption and software protection.

Scenario	Cycles	Time	Overhead
Legacy (standard ID)	8,135	0.41 ms	–
Legacy (extended ID)	9,620	0.48 ms	18%
vatiCAN (extrapolated†)	58,948	2.95 ms	625%
Sancus+vatiCAN (unprotected)	15,570	0.78 ms	91%
Sancus+vatiCAN (protected)	16,036	0.80 ms	97%
Sancus+LeiA (unprotected)	18,770	0.94 ms	131%
Sancus+LeiA (protected)	19,211	0.96 ms	136%

† Inferred from the observed Sancus+vatiCAN timings by replacing the hardware based MAC computation cycles with the reported Keccak SHA-3 computation cycles.

The memory footprint of the unprotected CAN driver measures 2,482 bytes. For our vulcanized vatiCAN/LeiA libraries on the other hand, the total binary sizes measure respectively 790/1,818 bytes when compiled as an unprotected application, and 906/1,948 bytes when compiled as part of a protected module. The slightly increased PM binary size is due to compiler-generated code stubs inserted on every call to and from the unprotected CAN driver. vatiCAN furthermore requires 22 bytes of metadata for each authenticated connection (a 16-bit ID, 128-bit symmetric key, and 32-bit nonce), whereas LeiA requires 44 bytes (16-bit ID, 64-bit epoch counter, 16-bit nonce, 128-bit long-term/session keys). Note that the 128-bit long-term keys can be shared across multiple connections, depending on the required security guarantees.

5.2 Performance Evaluation

To yield a fair overhead comparison, it should be noted that vatiCAN was evaluated on a 16 MHz ATmega 8-bit ECU. We expect software performance to be within the same order of magnitude, however, on our 20 MHz MSP430 16-bit microcontrollers.

MAC Computation. For each secure message, vatiCAN computes a 64-bit MAC over 112 bits of associated data (ID, payload, nonce). This operation reportedly requires about 47,600 clock cycles or 2.95 ms, using an optimized Keccak SHA-3 hash function in the software-only vatiCAN implementation [34]. Our hardware-assisted solution on the other hand is over 11 times faster, requiring only 4,222 clock cycles or 0.21 ms to compute a 128-bit MAC. Our vulcanized LeiA

Table 2: Round-trip (ping-pong) time intervals.

Scenario	Cycles	Time	Overhead
Legacy	20,250	1.01 ms	–
vatiCAN (extrapolated†)	121,992	6.10 ms	502%
Sancus+vatiCAN unprotected	35,236	1.76 ms	74%
Sancus+vatiCAN protected	36,375	1.82 ms	80%
Sancus+LeiA unprotected	42,929	2.15 ms	112%
Sancus+LeiA protected	43,624	2.18 ms	115%

implementation, on the other hand, uses only 16-bit nonces, which reduces associated data to 96 bits and MAC computation time down to 4,049 cycles (0.20 ms).

Cryptographic acceleration is crucial to fulfil real-world automotive safety requirements (e.g., automated collision avoidance) in benign conditions. At a highway speed of 100 km/h for instance, vatiCAN's software-based MAC computation corresponds to a travelling distance of 8 cm, whereas VulCAN cuts this distance down to a mere 0.6 cm. Naturally, authentication overhead becomes even more important when a message has to be processed by multiple interacting components, such as anti-lock braking, electronic brake force distribution, and electronic stability control systems.

Message Transmission. We furthermore investigated the *overall* overhead to send an authenticated message.[4] The first row of Table 1 lists the baseline, i.e., the time required by our CAN driver to send a plain unauthenticated message with an 11-bit standard ID (about 0.41 ms). Recall, however, that the limited CAN payload length requires us to send MACs in a separate authentication message, following the actual message to be authenticated. Due to this inherent CAN limitation, VulCAN must send two messages for every legacy message to be authenticated. In this respect, our baseline measurement in Table 1 includes the time needed by the transceiver chip to initiate transmission and signal acknowledgement from the receiver device. Our driver also includes an option, however, to return immediately after loading the relevant device registers, and without waiting for the transmission to complete successfully. This option is used in our vulcanized vatiCAN/LeiA implementations when sending the legacy message that is to be authenticated. By unblocking immediately after submitting the legacy payload for transmission, VulCAN can start the MAC computation early on, and only afterwards wait for acknowledgement of the legacy and authentication messages. This optimization technique makes that the overall time to send one authenticated message can still be slightly less than the time to send two unauthenticated legacy messages.

As an important consequence of Sancus' hardware-level cryptographic primitives, our vulcanized vatiCAN/LeiA implementations compute MACs in only *half* the time required to send a message, whereas MAC computation in software-only vatiCAN easily outweighs the message transmission time (with almost a factor 6). Comparing the third and fourth rows of Table 1 indeed reveals that the total relative overhead for sending an authenticated message with Sancus+vatiCAN decreases the relative overhead with a staggering 534%, as compared to the extrapolated software-only vatiCAN case. Regarding LeiA, the second row of Table 1 reveals that sending a legacy CAN message with an extended 29-bit identifier is slightly more expensive (about 18%), as extra device registers must be loaded. This explains why the overall overhead for Sancus+LeiA is slightly larger than that of vulcanized vatiCAN.

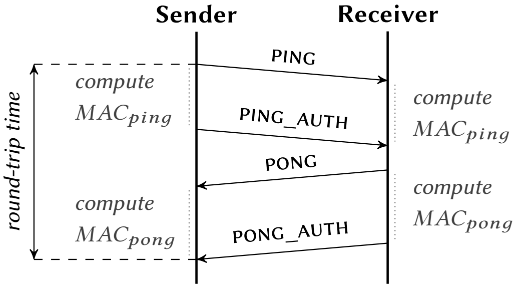

Figure 5: Round-trip time experiment timing overview.

In the last experiment, we not only use Sancus' hardware-level encryption, but also leverage its software isolation primitive to maintain key secrecy on a partially compromised ECU. More specifically, we construct CAN messages and their corresponding MACs in a PM, such that keys are never exposed to untrusted system software. As compared to the unprotected case, Sancus+vatiCAN/LeiA now also has to transparently copy message memory buffers to and from the unprotected CAN driver. Previous work [30, 44] has demonstrated that such protection domain switches come with a limited performance penalty, which is indeed reflected with a modest 5 to 6% execution time overhead in Table 1.

Round-Trip Time. To assess authentication delays in a sender/receiver scenario, we performed a round-trip time experiment. One ECU first broadcasts an 8-byte CAN frame and upon successful reception, the receiver ECU immediately broadcasts a reply message. For a legacy, unauthenticated CAN network, we measured a ping-pong time interval of 1.01 ms to exchange two messages in total. On an authenticated CAN network, four messages have to be exchanged, and sender and receiver each have to compute two MACs, as shown in Fig. 5.

Table 2 lists the results of the round-trip time experiment. Since twice the amount of messages have to be exchanged (to carry the authentication codes), the performance overhead to be expected is again around 100%. As compared to Table 1, however, the relative overhead for vulcanized vatiCAN/LeiA decreased even further, down to respectively 74% and 112%. This reflects the fact that messages now also have to be received and processed by the receiver, making the contribution of MAC computation less important in the overall timing. Likewise, the induced Sancus software protection overhead amounts only 3 to 6%, and can be expected to drop even further if the protected modules spend more time processing the message. The extrapolated measurements for software-only vatiCAN on the other hand, demonstrate once more that software-level MAC calculation remains dominant, increasing the relative overhead with over 400%.

We acknowledge that the above overheads may be prohibitive for some existing real-world applications. Evaluating this is beyond

[4] We focus on message sending here, since authenticated message reception delays inherently rely on the behavior of the sender. We assess overall send/receive overhead in a round-trip time macro benchmark.

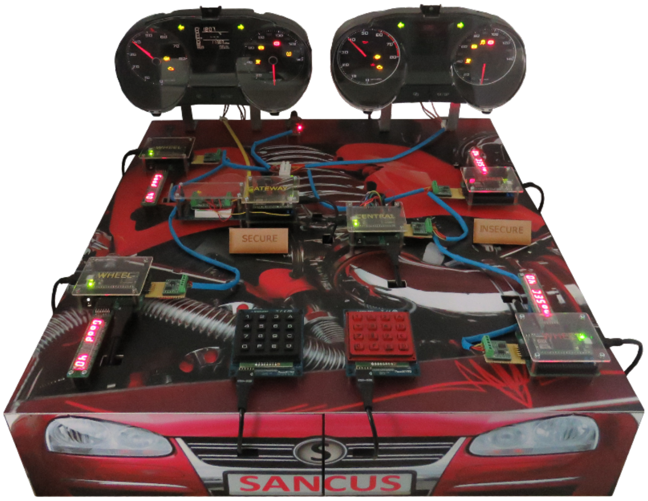

Figure 6: Hardware-in-the-loop application scenario with original instrument clusters and Sancus-enabled ECUs.

the scope of this paper. However, in comparison with state-of-the-art software-only solutions, our results show that VulCAN can target a substantially larger class of safety-critical applications where real-time deadlines must be satisfied. Ultimately, the choice of authentication protocol, communication technology, and trusted computing platform remains application-specific.

6 AN EXTENDED APPLICATION SCENARIO

To explore the feasibility of our approach in actual application scenarios, we constructed a moderate-sized automotive test bench. Figure 6 shows a photograph of our setup, which consists of two off-the-shelf automobile instrument clusters from the 2014 Seat Ibiza model, and six Sancus-enabled ECUs that were configured as described in Section 5. We furthermore attached a CAN-to-USB convertor to be able to monitor traffic and inject CAN messages from an ordinary laptop computer.

Application Overview. Figure 7 provides a simplified schematic of the demo scenario. It involves a central ECU that hosts a protected Sancus software module PM_{cs} to securely process commands from the legitimate motorist. To this end, we attach a keypad device that abstracts the motorist's interaction with the vehicle (i.e., steering wheel and braking/accelerator pedals). To gain exclusive access to the keypad peripheral, PM_{cs} securely links to a local PM_{key} MMIO driver assembly module, as explained in Section 2.2. For demonstration purposes, we also attached a second unprotected keypad, which is colored red and sensed by attacker code to represent an adversary with CAN message injection capabilities and arbitrary unprotected code execution on the central ECU.

The setup further features four wheel ECUs that each simulate an actuator (brake), abstracted by a peripheral LED display that displays the current wheel RPM. Each wheel ECU hosts a protected software module PM_{rpm} that listens to authenticated CAN messages from the central system to update individual wheel RPM on the local LED display via a secure MMIO driver module PM_{led}.

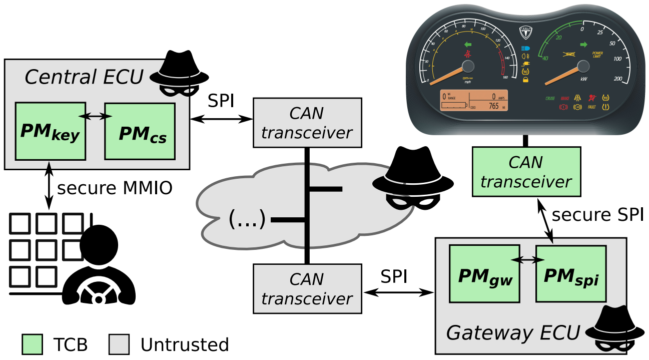

Figure 7: Schematic of the demo scenario depicted in Fig. 6.

The two vehicle instrument clusters finally represent unmodified legacy devices, where one is directly connected to the untrusted CAN bus, and the other is shielded by means of a Sancus-enabled gateway. The gateway ECU hosts a PM_{gw} component that listens to selected instrument cluster IDs on the global CAN bus, making sure that only authenticated messages are forwarded. To prevent untrusted software on the gateway ECU from injecting messages directly on the CAN bus that connects to the shielded dashboard, we include the CAN driver code in the PM_{gw} protection domain, and rely on a small PM_{spi} assembly module to secure the interaction with the MMIO-based SPI master peripheral that communicates with the CAN controller

Discussion. An important benefit of our vulcanized vatiCAN/LeiA software libraries, is that CAN authentication remains largely transparent to the distributed application components. At the application level, we programmed PM_{cs} to accept keypad commands for speeding up/down and steering left/right. In response to these events, authenticated CAN messages are created that respectively update the instrument's cluster RPM needle and turning light indicators. The distributed wheel ECUs respond to authenticated messages to update their individual wheel RPM, where the central system applies slightly more traction to the outer wheels while turning, as commonly performed in automotive traction control systems. The attacker keypad offers the same functionality, and indeed triggers responses on the *unprotected* instrument cluster. However, critical PM components safely reject attacker-generated messages. In particular, PM_{gw} does not forward the spoofed CAN message to the secure dashboard, but instead creates a message that triggers a warning light indicator to notify the motorist of the ongoing attack.

Our solution encompasses untampered CAN authentication and application-level message processing, as well as secure I/O. We thus guarantee that any critical actuator event (i.e., wheel LED display change or secure dashboard update) can only be caused by a chain of processing events that can ultimately be traced back to the authenticated component PM_{key} that originally sensed the keypad input, provided by the motorist herself. Importantly, our approach maintains this guarantee even when an attacker compromised the network and the unprotected software stack on participating ECUs.

7 RELATED WORK

Authentication on the CAN. Besides vatiCAN [34] and LeiA [38], a range of protocols and approaches for implementing authentication in automotive bus systems have been proposed, however these do not comply with the AUTOSAR [1] industry recommendations.

CANAuth [46] and LiBrA-CAN [15] propose authentication protocols for the CAN+ [51] specification. These schemes are backwards compatible so that authentication-aware ECUs can co-exist with legacy ECUs. Upgraded hardware, modified CAN transceivers and more powerful ECUs, are required to implement the protocols and cryptographic algorithms, however. For example LiBrA-CAN was evaluated on high-end ECUs and laptop CPUs, showing cryptographic performance comparable to our approach. VulCAN provides similar authenticity while being AUTOSAR-compliant, and relying on much lighter ECUs and off-the-shelf CAN hardware.

MaCAN [7, 16] does not require specifically adapted CAN transceivers but divides 8-byte CAN data into a 4-byte payload and a 4-byte MAC. Furthermore, the protocol requires network-wide trusted key- and time servers. The secrecy of credentials in MaCAN has been proven in [7] under a passive attacker model. LCAP [17] requires only 2 bytes of a CAN message to carry authentication information, but uses a large number of addresses (IDs) and messages to achieve node synchronization. CaCAN [23] introduces a central monitor node, which authenticates other ECUs and destroys unauthorized frames in real-time using custom CAN hardware.

All the above approaches require some form of specialized hardware to efficiently implement cryptography and to allow for specific operation on the CAN. The challenges of key distribution, software integrity, and key confidentiality in the presence of active ECU attackers, remain unaddressed. The VulCAN solution, on the other hand, aims for strictly stronger security guarantees and implements those by means of lightweight trusted computing technology.

A recent, comprehensive study [12] on the abilities of CAN network-level attackers shows that eavesdropping, message modification and ECU impersonation are not the only attack vectors in automotive control networks. From an availability perspective, selected target nodes could for instance be muted completely. Such attack vectors cannot be addressed by means of cryptography alone, but require physical isolation of safety-critical control networks, plus a notion of software security that prevents unauthorized software from abusing existing CAN hardware (cf. Section 6).

Trusted Execution Technology in Cars. The idea of relying on trusted computing components to develop secure automotive control systems was first coined in 2008 [35]. With the goal of attesting vehicular computing infrastructures, and providing additional security capabilities such as accelerated cryptography and key management, few implementations that leverage trusted execution technology [20] have been published. One such approach explored by the EVITA project, uses the Hardware Security Module (HSM) [49], a secure co-processor for accelerated ciphers, key storage, random number generation, and a secure clock. HSM runs at several hundred MHz and consequently exhibits significantly better cryptographic performance than Sancus, yet, it also relies on a substantially larger and more expensive hardware TCB. With Vector's MICROSAR [14], operating system support and CPU infrastructure for an HSM-like co-processor is commercially available.

Lightweight PMA-based trusted computing architectures such as Sancus provide strictly stronger security guarantees. Most importantly, PMAs are capable of not only attesting software components at boot time, but also isolating them at runtime to maintain security guarantees over the entire lifetime of the program. Our choice for Sancus is motivated by a recent exhaustive PMA overview [24] that indicates Sancus is the only embedded architecture with an open-source hardware design and software tool chain.

Related PMA Use Cases. Sancus has been previously applied in a secure smart metering infrastructure [29]. The security guarantees that Sancus provides in this distributed application scenario have been generalized and partly formalized [32], and are similar to our approach. Yet, real-time requirements are relaxed in comparison with automotive applications. An approach to securely share system resources on Sancus has been proposed [44], and subsequent research [45] has provided insight into implementing secure PM interruptability and scheduling to ensure real-time responsiveness on Sancus platforms. We expect these developments to further improve our approach in terms of availability guarantees. Secure interruption of hardware-enforced embedded PMs was first explored in TrustLite [21]. Subsequent work developed TyTAN [6], a minimal software-based root of trust for dynamic attestation of interruptible tasks. We expect TyTAN to be extensible to offer guarantees similar to our approach, but with the inherent performance and TCB implications of software-based cryptography.

Finally, Sancus has been used to develop a trust assessment system for IoT applications [30], which allows a PM to securely inspect and attest a host operating system on a lightweight computing node. In an automotive context, this technology could be used to integrate third-party hardware or software components into a vehicle while allowing the manufacturer to securely monitor their behavior.

8 CONCLUSIONS AND FUTURE WORK

This paper introduced VulCAN, a trusted computing design for message authentication plus software component attestation and isolation in vehicular communication networks. We contributed enhanced security requirements for CAN, and evaluated our hardware-assisted solution as compared to previous software-only authentication schemes. To the best of our knowledge, we are the first to present such a comparison, and to consider attackers capable of arbitrary (unprotected) code execution on participating ECUs. Our results show that relatively inexpensive microcontrollers equipped with lightweight embedded cryptography and software component isolation enable strong security guarantees, while maintaining real-time deadlines for safety-critical applications in benign conditions.

In future work, we will investigate real-time responsiveness and availability guarantees on partially compromised ECUs [45]. We furthermore plan to implement a secure vehicular attestation server [39], and investigate application scenarios in the context of V2X communications [48]. More generally, we see compelling use cases for our approach to authentic execution [32] for distributed embedded control systems in IoT domains, or the Industry 4.0.

Acknowledgments. This research is partially funded by the Research Fund KU Leuven. Jo Van Bulck is supported by a doctoral grant of the Research Foundation – Flanders (FWO).

REFERENCES

[1] AUTOSAR Specification 4.3. 2016. Specification of module secure onboard communication. https://www.autosar.org/standards/classic-platform/release-43/software-architecture/safety-and-security/. (2016).

[2] Morton Abramson and WOJ Moser. 1970. More birthday surprises. *The American Mathematical Monthly* 77, 8 (1970), 856–858.

[3] Tiago Alves and Don Felton. 2004. TrustZone: Integrated hardware and software security. *ARM white paper* 3, 4 (2004), 18–24.

[4] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. 2011. Duplexing the sponge: Single-pass authenticated encryption and other applications. In *Selected Areas in Cryptography*. Springer, 320–337.

[5] Andrey Bogdanov, Miroslav Knezevic, Gregor Leander, Deniz Toz, Kerem Varici, and Ingrid Verbauwhede. 2012. SPONGENT: The design space of lightweight cryptographic hashing. *IEEE Trans. Comput.* 99.

[6] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. 2015. TyTAN: Tiny trust anchor for tiny devices. In *Design Automation Conference (DAC '15)*. IEEE, 1–6.

[7] Alessandro Bruni, Michal Sojka, Flemming Nielson, and Hanne Riis Nielson. 2014. *Formal security analysis of the MaCAN protocol*. Springer International Publishing, Cham, 241–255.

[8] Madeline Cheah, Siraj A. Shaikh, Olivier Haas, and Alastair Ruddle. 2017. Towards a systematic security evaluation of the automotive Bluetooth interface. *Vehicular Communications* 9 (2017), 8–18.

[9] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10)*. ACM, New York, NY, USA, 559–572.

[10] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. 2011. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*. San Francisco.

[11] Karim Eldefrawy, Aurélien Francillon, Daniele Perito, and Gene Tsudik. 2012. SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. In *19th Annual Network and Distributed System Security Symposium (NDSS '12)*.

[12] Sibylle Fröschle and Alexander Stühring. 2017. Analyzing the capabilities of the CAN attacker. In *ESORICS '17 (LNCS)*, Vol. 10492. Springer, Heidelberg, 464–482.

[13] Olivier Girard. 2009. openMSP430 – a synthesizable 16bit microcontroller core written in Verilog. https://opencores.org/project,openmsp430. (2009).

[14] Vector Informatik GmbH. 2017. MICROSAR – AUTOSAR basic software and RTE. https://vector.com/vi_microsar_en.html. (2017).

[15] Bogdan Groza, Stefan Murvay, Anthony Van Herrewege, and Ingrid Verbauwhede. 2012. Libra-CAN: a lightweight broadcast authentication protocol for controller area networks. In *International Conference on Cryptology and Network Security*. Springer, 185–200.

[16] Oliver Hartkopp, C. Reuber, and R. Schilling. 2012. MaCAN: Message authenticated CAN. In *Escar Conference, Berlin, Germany*.

[17] Ahmed Hazem and Hossam A.H. Fahmy. 2012. LCAP: A lightweight CAN authentication protocol for securing in-vehicle networks. In *10th escar Embedded Security in Cars Conference, Berlin, Germany*, Vol. 6.

[18] O. Henniger, L. Apvrille, A. Fuchs, Y. Roudier, A. Ruddle, and B. Weyl. 2009. Security requirements for automotive on-board networks. In *9th International Conference on Intelligent Transport Systems Telecommunications, (ITST)*. 641–646.

[19] Tobias Hoppe, Stefan Kiltz, and Jana Dittmann. 2008. Security threats to automotive CAN networks – practical examples and selected short-term countermeasures. In *Computer Safety, Reliability, and Security (SAFECOMP '08)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 235–248.

[20] GlobalPlatform Inc. 2011. The trusted execution environment: Delivering enhanced security at a lower cost to the mobile market. https://www.globalplatform.org/documents/GlobalPlatform_TEE_White_Paper_Feb2011.pdf. (2011).

[21] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. 2014. TrustLite: A security architecture for tiny embedded devices. In *EuroSys '14*. ACM, 14 pages.

[22] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. 2010. Experimental security analysis of a modern automobile. In *Security and Privacy, 2010 IEEE Symposium on*. IEEE, 447–462.

[23] R Kurachi, Y Matsubara, H Takada, N Adachi, Y Miyashita, and S Horihata. 2014. CaCAN – centralized authentication system in CAN. In *14th Int. Conf. on Embedded Security in Cars (ESCAR '14)*.

[24] P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. Freiling, and I. Verbauwhede. 2017. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Trans. Comput.* 99 (2017).

[25] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *HASP '13*. ACM, 8 pages.

[26] Earl H McKinney. 1966. Generalized birthday problem. *The American Mathematical Monthly* 73, 4 (1966), 385–387.

[27] Charlie Miller and Chris Valasek. 2014. A survey of remote automotive attack surfaces. *Black Hat USA* (2014).

[28] Charlie Miller and Chris Valasek. 2015. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA* (2015).

[29] Jan Tobias Mühlberg, Sara Cleemput, Mustafa A. Mustafa, Jo Van Bulck, Bart Preneel, and Frank Piessens. 2016. An implementation of a high assurance smart meter using protected module architectures. In *WISTP '16 (LNCS)*, Vol. 9895. Springer, Heidelberg, 53–69.

[30] Jan Tobias Mühlberg, Job Noorman, and Frank Piessens. 2015. Lightweight and flexible trust assessment modules for the Internet of Things. In *ESORICS '15 (LNCS)*, Vol. 9326. Springer, 503–520.

[31] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. 2013. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd USENIX Security symposium*. USENIX Association, 479–494.

[32] Job Noorman, Jan Tobias Mühlberg, and Frank Piessens. 2017. Authentic execution of distributed event-driven applications with a small TCB. In *STM '17 (LNCS)*, Vol. 10547. Springer, Heidelberg, 55–71.

[33] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. 2017. Sancus 2.0: A low-cost security architecture for IoT Devices. *ACM Transactions on Privacy and Security (TOPS)* 20 (2017), 7:1–7:33. Issue 3.

[34] Stefan Nürnberger and Christian Rossow. 2016. – vatiCAN – Vetted, authenticated CAN bus. In *Cryptographic Hardware and Embedded Systems – CHES '16: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 106–124.

[35] H. Oguma, A. Yoshioka, M. Nishikawa, R. Shigetomi, A. Otsuka, and H. Imai. 2008. New attestation based security architecture for in-vehicle communication. In *IEEE GLOBECOM '08 - 2008 IEEE Global Telecommunications Conference*. 1–6.

[36] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. 2015. Secure compilation to protected module architectures. *ACM Trans. Program. Lang. Syst.* 37, 2 (2015), 6:1–6:50.

[37] Pieter Philippaerts, Jan Tobias Mühlberg, Willem Penninckx, Jan Smans, Bart Jacobs, and Frank Piessens. 2014. Software verification with VeriFast: Industrial case studies. *Science of Computer Programming (SCP)* 82 (2014), 77–97.

[38] Andreea-Ina Radu and Flavio D. Garcia. 2016. LeiA: A lightweight authentication protocol for CAN. In *Computer Security – ESORICS '16: 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part II*. Springer International Publishing, Cham, 283–300.

[39] Vincent Raes and Vincent Naessens. 2017. Development of an embedded platform for secure CPS services. In *CyberICPS '17 (LNCS)*. Springer, Heidelberg. In press.

[40] SAE International. 2016. J3061: Cybersecurity guidebook for cyber-physical vehicle systems. (2016). http://standards.sae.org/j3061_201601/.

[41] Seeed Studio. 2017. CAN BUS shield driver for Arduino/Seeeduino. https://github.com/Seeed-Studio/CAN_BUS_Shield. (2017).

[42] Raoul Strackx, Job Noorman, Ingrid Verbauwhede, Bart Preneel, and Frank Piessens. 2013. Protected software module architectures. In *Securing Electronic Business Processes*. Springer, 241–251.

[43] Raoul Strackx, Frank Piessens, and Bart Preneel. 2010. Efficient isolation of trusted subsystems in embedded systems. In *Security and Privacy in Communication Networks*. Springer, 344–361.

[44] Jo Van Bulck, Job Noorman, Jan Tobias Mühlberg, and Frank Piessens. 2015. Secure resource sharing for embedded protected module architectures. In *WISTP '15 (LNCS)*, Vol. 9311. Springer, 71–87.

[45] Jo Van Bulck, Job Noorman, Jan Tobias Mühlberg, and Frank Piessens. 2016. Towards availability and real-time guarantees for protected module architectures. In *MASS '16, MODULARITY Companion 2016*. ACM, New York, 146–151.

[46] Anthony Van Herrewege, Dave Singelee, and Ingrid Verbauwhede. 2011. CANAuth – a simple, backward compatible broadcast authentication protocol for CAN bus. In *ECRYPT Workshop on Lightweight Cryptography*, Vol. 2011.

[47] David A Wheeler. 2004. SLOCCount. https://www.dwheeler.com/sloccount/. (2004).

[48] Jorden Whitefield, Liqun Chen, Frank Kargl, Andrew Paverd, Steve Schneider, Helen Treharne, and Stephan Wesemeyer. 2017. Formal analysis of V2X revocation protocols. In *STM '17 (LNCS)*, Vol. 10547. Springer, Heidelberg, 147–163.

[49] Marko Wolf and Timo Gendrullis. 2012. Design, implementation, and evaluation of a vehicular hardware security module. In *Information Security and Cryptology (ICISC '11) (LNCS)*, Vol. 7259. Springer, Berlin, 302–318.

[50] Marko Wolf, André Weimerskirch, and Christof Paar. 2004. Security in automotive bus systems. In *Workshop on Embedded Security in Cars*.

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

A NONCE GENERATOR BIRTHDAY ATTACK

In this appendix, we outline a replay attack against vatiCAN's [34] global Nonce Generator (NG) scheme. The vatiCAN protocol prevents trivial replay attacks by including a monotonically increasing 32-bit nonce value c_i in the MAC for each message with identifier i. Receiver ECUs are expected to increment their shadow nonce counter whenever a valid authenticated message was sent. To accommodate for packet loss, however, vatiCAN relies on a trusted NG component that periodically broadcasts a randomly chosen global value g to be used by all participating ECUs as the new initial value for all counters c_i. Naturally, the NG broadcast frequency should be sufficiently high, as it represents the worst-case time interval in which an ECU may discard valid authenticated messages. vatiCAN broadcasts a nonce renewal message every 50 ms. NG is furthermore assumed to use a modified CAN transceiver, equipped with hardware-assisted spoofing prevention, such that an attacker cannot inject arbitrary nonce renewal requests.

We show, however, that vatiCAN's frequent random nonce renewal approach is vulnerable to more advanced replay attacks that rely on a moderate amount of previously recorded CAN broadcast traffic. More specifically, we are interested in the probability that n 32-bit global nonce values randomly chosen by NG contain at least one duplicate. In case such a nonce repetition occurs, an attacker can successfully replay previously authenticated CAN messages for the next 50 ms. Provided the nonce repetition probability is sufficiently high, the adversary could for instance engage safety-critical functionality such as brakes or steering systems, after collecting only a reasonable amount of traffic.

The above problem is an instance of the well-known (generalized) "birthday problem" [26] that asks for the probability p that a collision occurs in n randomly chosen samples out of a set of d possibilities. The approximated probability is given by:

$$p(n, d) \approx 1 - e^{\frac{-n(n-1)}{2d}} \tag{2}$$

After filling in vatiCAN's nonce size $d = 2^{32}$, Eq. (2) exhibits a 90% nonce reuse probability after only $n = 135,000$ NG nonce renewals. Likewise, a 99% probability is reached within $n = 199,000$. In other words, at a nonce renewal interval of 50 ms, an adversary is sure to expect a global nonce value g that was previously broadcast by NG *within* 2 to 3 hours. Since all vatiCAN components reset their internal nonce counters $c_i = g$ after receiving a valid NG message, the attacker can now successfully replay previously recorded CAN traffic for the next 50 ms (until the next NG broadcast). We also note that, in case the attacker controls custom CAN transceiver hardware, she might furthermore tear down future NG nonce renewal messages (by destroying the CRC checksum on the fly) so as to extend her 50 ms replay attack window.

While the generalized birthday attack outlined above is already quite practical, it can be significantly improved when also considering that NG should not necessarily produce two *exact* same nonce values. Indeed, when NG broadcasts a value $h = g + k$ that is within a distance k to a previously used global nonce g, the attacker may simply skip the first k messages when replaying traffic from the g epoch. Likewise, should NG broadcast a value $h = g - k$, the adversary may simply wait for k benign authenticated messages before starting to replay traffic from the g epoch. Intuitively speaking, the probability for a k-near nonce collision increases dramatically when taking into account this novel insight. In fact, this modified attack scenario is an instance of a "near birthday problem" [2], which is approximated by the following equation:

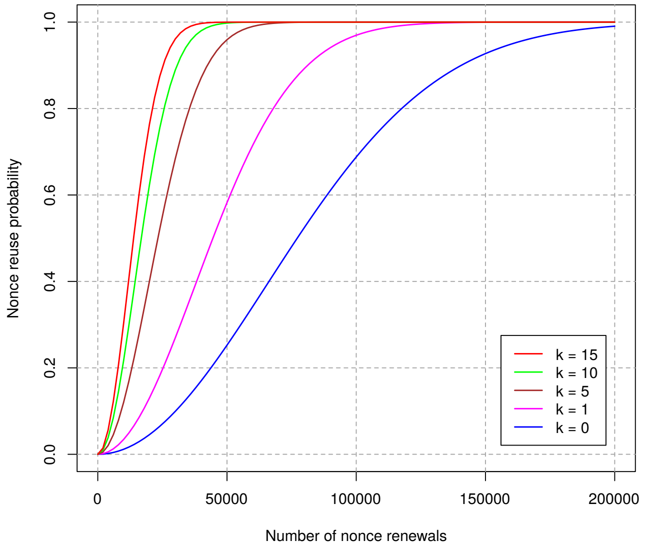

Figure 8: Nonce reuse probabilities in function of NG broadcasts, as provided by Eq. (3) for 32-bit nonces, and for various k-values (where $k = 0$ corresponds to Eq. (2)).

$$p(n, k, d) \approx 1 - \frac{(d - nk - 1)!}{d^{n-1}(d - n(k+1))!} \tag{3}$$

Figure 8 visualizes the nonce reuse probability distributions for various values of k. The distributions distinctly approach the 100% nonce collision probability more rapidly as the k parameter increases. Note that $k = 0$ corresponds the conventional birthday problem Eq. (2) that clearly achieves nonce reuse certainty within $n = 200,000$ (i.e., 167 minutes at a 50 ms NG nonce renewal interval). When $k = 1$, however a 99% nonce reuse probability is already reached within $n = 115,000$ (96 minutes). Likewise, for $k = 5/10/15$, a 99% certainty is reached within respectively $n = 60,000$ (50 minutes), $n = 44,000$ (37 minutes), and $n = 36,000$ (30 minutes). This means that, depending on the application under attack, as little as 30 minutes of recorded CAN traffic might suffice to successfully replay a safety-critical authenticated message, effectively defeating vatiCAN authentication.

It should be clear from the above explanation that randomization is inherently *insufficient* to protect against advanced replay attacks. Instead, the only way to properly prevent these attacks is to never reuse the same nonce value under the same key. As explained above, Sancus+vatiCAN therefore (re-)establishes fresh symmetric session keys on platform boot and/or nonce counter overflow.