Leveraging In-Car Security by Combining Information Flow Monitoring Techniques

Alexandre Bouard*, Hendrik Schweppe*,†, Benjamin Weyl*, Claudia Eckert‡
*BMW Forschung und Technik GmbH, Munich, Germany
Email: {alexandre.bouard, hendrik.schweppe, benjamin.wey}@bmw.de
†EURECOM, Sophia-Antipolis, France
Email: schweppe@eurecom.fr
‡Technische Universität München, Munich, Germany
Email: claudia.eckert@sec.in.tum.de

***Abstract*—Modern automobiles are increasingly connected to the world and integrate always more electronic components managing simultaneously infotainment and safety functions. Much more than just a simple transportation mean, the car is now customizable and like current smartphones, it will soon allow to load and install third-party applications directly from the Internet, which raises some security issues. Until now, the car manufacturer has full control over the development process of the in-car software and in particular can perform any required security tests before the car production. The integration of untrusted pieces of code requires from now on new dynamic security mechanisms operating during the life time of the car. In this paper, we present the integration of data flow tracking tools in an automotive middleware allowing dynamic security monitoring of untrusted applications. We describe the implementation and integration of these mechanisms and provide their evaluation.**

***Index Terms*—automotive applications, security, privacy, information flow controls, data flow tracking**

I. Introduction

Today automotive applications are deployed on on-board electronic platforms before their delivery and in-car integration. They are usually designed and programmed specifically for a brand or even for a precise car model. The car manufacturer always knows the application developers, either because they directly belong to the company or are working for a subcontractor, and can therefore contractually set certain responsibilities and testing processes. This clear relationship allows to pinpoint potentially arising liability issues if the security is broken. However, this model does not necessarily protect from security weaknesses [1] and leaves very little room for a flexible deployment. In contrast to this aforementioned pre-assembly-line development, loadable and on-the-fly installable applications have revolutionized the mobile device world and are now coming into cars [2]. While being foreseen only for the infotainment domain, they bring a number of risks that can hardly be deemed. With access to Internet and to a wide range of on-board functions, these "apps" may secretly do more than they appear to do under the hood [3] and may lead to severe driver's privacy infringements or even worse to life-threatening issues. The architecture for deploying and running such "apps" needs to be secured accordingly.

Dynamic data flow tracking (DFT) has been successfully applied in various security domains: exploit detection [4], malware analysis [5] and system monitoring [6]. It allows to tag and track data of interest as they propagate within a running application. Our approach presents the combination of two information flow control (IFC) approaches in order to secure, with acceptable performance, the execution environment of original on-board applications and third-party applications (TPAs) against security exploits and privacy infringements. DFT is locally applied to monitor TPA and is integrated in a car-wide security framework enforcing a decentralized IFC model. The main contributions presented in this paper are:

- A new *authorization model* combining local DFT techniques and car-wide IFC mechanisms.
- A *prototype implementation* integrating an automotive and IP-based communication middleware, *Etch* [7] and a DFT tool, our customized version of *libdft* [8].

The rest of the paper proceeds as follows. After giving a brief overview in Section II about the on-board automotive architecture and related work, Section III introduces our model combining two different information flow control techniques. Then, Section IV presents the implementation and integration of the security tools in an automotive middleware. Section V provides the evaluation of our concepts and prototype and finally Section VI our conclusions.

II. Scope of this Work

This section summarizes background information on future automotive systems and related work about security & privacy. A threat model and relevant scenarios are provided as well.

A. Current and Future Automotive Architectures

The automotive on-board network includes up to 70 Electronic Control Units (ECUs). ECUs are organized in subnetworks around specific domains (e.g., power train, infotainment), interlinked by different communication buses, e.g., CAN, MOST. On-board applications are divided into elementary function blocks, which are distributed over several ECUs and use broadcast to exchange data. Due to plaintext on-board communication and a lack of input validation in the ECUs, cars have been shown weak against simple attacks [1].

Tomorrow, Ethernet/IP will be used as communication standard for the on-board network and will provide a larger

 ISBN: 978-1-61208-288-2

bandwidth. It will allow comply with the future application requirements for driver assistance and infotainment, which require large volume of data to be processed at real time [9]. Secondly, secure and mature protocols from the Internet will be immediately applicable. Then, the use of engineering-driven middleware will greatly simplify the communication management. It will abstract and automate the network addressing and security enforcement [10]. Finally the centralization of most of the external communication interfaces (e.g., LTE, Wi-Fi) around a multi-platform antenna (called proxy here) will give to car makers the opportunity to design a single security gateway for all Car-to-X (C2X) communications [11].

B. Threat Model and Attack Scenarios

Securing today's and tomorrow's car is challenging. Automotive applications are rarely updated and involve more and more new connected features. Their functional behavior relies on complex software, not free from any security weaknesses [1] and processing a huge amount of sensitive data. An attacker could therefore leverage defects in the logic of an application or in a weak security mechanism. The car could therefore leak private information or industrial secrets, have its integrity threatened and endanger the life of its occupants.

Our scenario is depicted in Figure 1 and considers both internal and external communication partners. We take the example of a TPA running on the Head Unit (HU), which is connected to services of the Internet, a CE device and to several original on-board applications. We mainly focus on attack scenarios trying to bypass security policies and taking advantage of the TPA in order to 1) corrupt internal resources or 2) release sensitive data to an unauthorized external entity. We consider a TPA, which respects the internal API of the car. However, it may present some weaknesses that are exploitable by an attacker.

1) **Integrity attack scenario**: the TPA forwards malicious messages from an external malicious communication partner or gets compromised. As a result the TPA may send bogus packets on the on-board network or access/modify critical resources on the HU and may dangerously disturb the car functioning.
2) **Confidentiality attack scenario**: the TPA accesses private/secret information, like the driver's home address in the navigation system. The TPA, even without the authorization to share it, may still send it to the outside, either directly over the proxy or through an intermediate step, e.g., an on-board application communicating with the outside.

This work aims at enhancing the information security and tackling the threats related to unfair entities, on which the car manufacturer has no control, while still keeping the car's requirements for high robustness and low latency in mind.

Assumptions: Next-generation ECUs will be equipped with a security middleware allowing on-board communications over strong security protocols like IPsec [10]. In addition they will soon make use of secure hardware extensions providing secure boot and secure key storage [12]. As a consequence, we assume that the middleware and the hardware platform are flawless and cannot be compromised. Finally, we trust the ECUs to establish secure communication channel with each others and to enforce the expected security mechanisms. We do not consider denial-of-service attacks in this work.

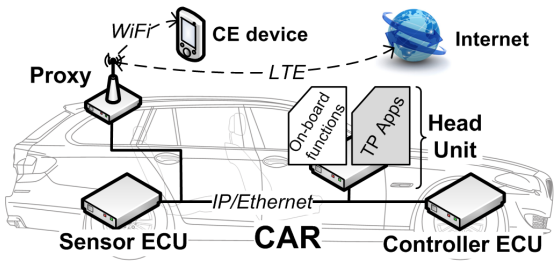

Fig. 1. Automotive scenario and considered communication channels. Solid right-angle lines represent the wired on-board network. The dashed arrows represent external communications over different wireless networks.

C. Related Work

DFT is not a new research topic, many DFT tools have already been implemented and tested for various security purposes, mostly for malicious code detection and information leakage. The main idea is to recognize data of interest according to predefined policies, to associate them with metadata called "taint tags" and to track their propagation within a running application or system. Two classes of tools can be distinguished: the single-process- and the cross-process-ones.

Single-process DFT tools [4], [8] instrument every machine instruction performed by a process. To do so, they generally make use of dynamic binary instrumentation (DBI) frameworks like Pin [13]. They usually suffer from significant decrease performance and need additional memory for taint propagation. They do not require source code modification or customized OS.

Cross-process DFT tools capture system-wide data flow and usually rely on modified runtime environments [6], emulators like QEMU [5] or hardware extensions [14]. They are usually heavyweight systems requiring an extensive maintenance. They instrument every instruction performed in the host and as a consequence impose a very significant overhead for the overall system. TaintDroid [6] alleviates the issue by regarding some libraries as "trusted", i.e., not monitored.

Solutions for **DFT in distributed systems** generally offer little reusability and require every peer to run the DFT tool, e.g., DBTaint [15], targeting data flow in data-bases, Neon [16] using a modified NFS server to track the taints of inbound/outbound packets. Taint-Exchange [17] presents a generic framework based on libdft [8] allowing exchanges of taints over the network without proposing any concrete security model or policy enforcement. Another interesting automotive approach [18] proposes a security model using a DFT tool and network taint exchanges for every application running on the on-board network. While enhancing the security, we believe that such approaches will not meet the automotive latency requirements, if every on-board application is instrumented.

 ISBN: 978-1-61208-288-2

IFC is a general term. It usually designates a type of mandatory access control, allowing an entity, e.g., a person or a process, to access a resource depending on its clearance. IFC models have been already applied to secure distributed systems e.g., at the process level in customized OSes exchanging labeled messages through the network [19] or thanks to customized switches and a central synchronization server [20]. In opposition to DFT, these approaches provide better performance. They protect the information confidentiality and integrity, but do not protect against security vulnerabilities.

Outcome: Considering our requirements for low latency, we orient our approach towards efficient single-process DFT. The TPA is monitored, while trusted applications of the HU are not monitored. Potential misbehavior of the TPA is locally contained by the DFT tool. Communications between the TPA and other on-board applications are secured thanks to the enforcement of car-wide IFC policies.

III. A Combined Approach

The middleware is a software layer, common to every on-board application, including the TPA. Our approach makes use of the middleware to link the DFT tool and its local action to the security framework enforcing car-wide IFC policies. The rest of the section explains in more detail A) how the DFT tool works, B) how DFT tools and trusted on-board applications securely communicate, and C) how the DFT monitoring is integrated with a car-wide IFC framework.

A. Tracking and Controlling the Execution

DFT tools are characterized by three main elements: the taint sources, the intra-taint propagation and the taint sinks. For this subsection, we consider the pseudocode of Figure 2.

Taint sources: Taint sources are programs or memory locations, where data enter the monitored system after the invocation of a function or of a system call. If recognized as data of interest, they are tainted and tracked. Based on our scenarios, we identify all traditional I/O channels used by the TPA as sources : inter-process communication (e.g., pipe), filesystem and network socket. For instance, we monitor the functions "receiveBuffer()" (line 1) and "readBuffer()" (line 2) and tag the buffers "x" and "y" accordingly.

Intra-taint propagation: During runtime, tainted data are tracked while being copied and altered by the application execution, like in the function "processBuffers()" (line 3), which generates some data out of two tainted buffers that is tainted as well. The taint information is stored and dynamically propagated in a shadow memory mapped to the actual process memory. The taint expressiveness can be adapted depending on the needs. Originally DFT was used to protect software vulnerabilities from being exploited and a simple binary tainting was sufficient to track untrusted data (e.g., one bit tainting a byte of memory). But considering our goal to both protect the system integrity and the information sensitivity, we require more possible taint values with regard to the input sources. In practice, to limit the execution and communication overhead, we use four values: (3) for highly sensitive data of the car manufacturer (e.g., industrial secret) (2) for user's very private data (e.g., location, routes), (1) for user's private data (e.g., username, preferences) and (0) for nonsensitive data. This scale does not reflect data integrity, because by definition TPA cannot be trusted to produce data that are safe to directly process.

	Shadow memory, taint of		
	x	y	z
0:	(0)	(0)	(0)
1: buffer **x** = receiveBuffer(from_a_ECU);	(2)	(0)	(0)
2: buffer **y** = readBuffer(from_sensitive_file);	(2)	(1)	(0)
3: buffer **z** = proccessBuffers(**x**, **y**);	(2)	(1)	(2,1)
4: write(**z**, in_output_file);	(2)	(1)	(2,1)
5: sendBuffer(**z**, to_another ECU);	(2)	(1)	(2,1)

Fig. 2. On the left, an example of code with data dependencies (in bold, the data to taint). On the right the intra-taint propagation for the buffers x, y, z along the code execution.

Taint sinks: Like sources, taint sinks are function calls and memory locations, where the presence of a taint is checked in order to enforce a policy. The policies concern decision about transmitting data to a specific function, or using the data as program control data (e.g., return address). It determines whether the data can be written to a standard output (e.g., in a file, line 4) or sent over the network (line 5).

B. Middleware-based Taint Propagation

DFT tools allow to eliminate numerous attacks related to stack pointer overwriting, like buffer-overflow or format-string exploits. Other trusted automotive applications are not instrumented and directly communicate with the untrusted TPA. Thanks to the middleware and the exchange of security metadata, the DFT tool can share the intra-taints with other on-board applications.

Extra-taint propagation: Figure 3 shows the propagation of taints between a TPA and other on-board applications on other ECUs. The system calls, related to the network socket management (lines 2, 5 in Figure 2 and bullets 3, 4 in Figure 3) are intercepted by the *Injector*. For inbound messages (bullet 3), the *Injector* checks if the trusted applications is allowed to communicate with the TPA, extracts the taint of the payload from the middleware header and taints the received data in its shadow memory. For outbound messages (bullet 4), the *Injector* checks if the TPA is allowed to communicate with the addressee and adds the taints related to the message payload in the middleware header. Both sides of the communication establish a secure communication channel.This prevents any unauthorized taint manipulation or eavesdropping during the message exchange. After the message reception, the middleware of the receiving application extracts the taint values from the payload and enforces the related security policies.

Middleware enforcement: Unlike the DFT framework, the middleware of an on-board application enforces static policies and cannot be aware of each new TPA's requirements and policies. The middleware therefore enforces a taint-based filtering involving generic rules for all TPAs. The middleware trusts the DFT framework to communicate accurate taint values. The

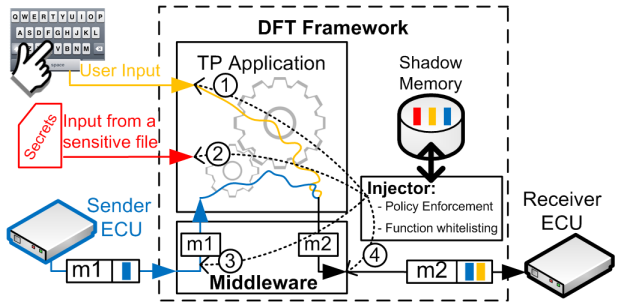

Fig. 3. Overview of the DFT framework. The solid lines show the input and output data of the TPA. The colors represent different levels of sensitivity, that are expressed by the taint values (i.e., blue/yellow/red and 1/2/3). These taints are injected using binary instrumentation (*Injector*). The *Injector* monitors the execution, especially system calls (dotted lines) and the propagation of memory and registers. $m1$ and $m2$ are tainted messages sent respectively to and from the TPA. The TPA output $m2$ shows a combination of the sources blue and yellow but not red and is therefore tainted accordingly.

taint values inform the middleware whether the data may be sensitive. The integrity of the data cannot be assessed, but TPA and middleware communicate through a dedicated channel. The middleware is therefore aware of the integrity risk. It first decides, whether it can process the payload depending on its version and its security level. and then evaluates, based on the taint values, which DFT/IFC rules of Section III-C to enforce.

Security policies: In order to control the data flows in the HU and over the network, we identify two types of policies: the first ones regulating the network exchanges and the DFT engine (middleware- & DFT-specific), the other ones locally enforced for a specific TPA (application-specific).

a) *Middleware- & DFT-specific policies*: Defined by the car manufacturer, these rules are static and defined during the design phase. They regulate the communication establishment between on board applications and specify the rules for IFC and for the interface DFT/IFC. Taint values and related source, propagation and sink tainting rules are statically set in the DFT engine by the car manufacturer

b) *Application-specific policies*: These policies only concern the TPA. When loaded on the ECU, the TPA is supplied with a customized rule set defining the associated permissions. These new rules are evaluated against the static ones and integrated in the DFT framework. Similar to the manifest provided by every Android application, the rule set provides more detailed policies and function whitelisting. For obvious security reasons, the rule set will have to be approved and signed by the car manufacturer after a testing process. In addition, other privacy-relevant policies may be specified directly by the user thanks to an on-board configuration interface.

C. Combining DFT & Car-wide IFC Enforcement

DFT provides efficient ways to control a TPA and to add/extract information from the in-band middleware protocol. However, because they monitor every single instruction of an application, they impose a significant overhead. As a consequence we chose to limit the taint information to four values. On-board applications are exchanging data which belong to different drivers or passengers and require different levels of integrity, e.g., to trigger a safety mechanism. Four taint values may lack some expressiveness. We therefore chose to couple the DFT mechanisms to a solution offering more flexibility: an automotive IFC framework.

Automotive IFC Framework: Our work about IFC in cars has been recently submitted [21] and makes use of a model inspired from Dstar [19]. We define as service, a group of on-board applications running on top of the same middleware and sharing the same security concerns in term of integrity (e.g., because they get access to the same safety mechanisms) and confidentiality (e.g., because they share data of same sensitivity). IFC is about controlling and monitoring which pieces of information are exchanged through the network between services. The goals of our framework are twofold. First it allows to control the access to on-board resources, e.g., service or file. The IFC framework makes sure that the access enquirer has a sufficient integrity level. Then it allows to prevent information leakage, caused by unintentional bugs and unfair external peers, like CE devices and online services.

Every service/user is assigned a label, i.e., a set of tags. Tags are unique values qualifying for each service/user either its integrity- or its confidentiality-concerns (i.e., the sensitivity of the information it processes). These labels form a lattice enforcing a form of mandatory access control, where information from a service can flow to a second one only if it fulfills certain conditions. Concretely, in order to send a message from A to B, the confidentiality tags of A have to be included in the label of B and the integrity tags of B in the label of A. When enforced, these conditions allow to preserve first the information confidentiality, i.e., A is sure that B is authorized to receive its message, and then the information integrity, i.e., B is sure that A provides a suitable integrity level and can therefore process the received information. Then depending on the use case, a service can "own" a tag, which means that for more flexibility it can chose to not enforce the condition linked to it. For example, the proxy owns several user confidentiality tags, i.e., reflecting the confidentiality concern of a user This allows the proxy to be able to receive information from several users of the car. A confidentiality tag of a user U in its label would constrain it to receive information only from U.

The enforcement of the IFC policies is performed in the middleware of each service each time a message is received or sent. The security logic is isolated from the application logic. Because a service is not necessarily aware of the label used by another service, the middleware of the sender adds some metadata to the message payload describing the label, on which the receiving middleware has to enforce the IFC policy. The tags in a service label are static and defined by the car manufacturer at design time. Only user tags can be generated by the proxy during runtime, e.g., when a new user get logs into the car. In this case, the proxy can grant the ownership of the new tag to some services like the HU. It allows these services to securely handle the data of the user and the communications with the TPAs (see next paragraph "IFC/DFT Interface"). Labels and tags are only for an on-board usage and we only trust on-board services to enforce

 ISBN: 978-1-61208-288-2

the IFC rules. External communication partners and TPAs do not enforce any of them. Messages to and from the outside are filtered by the proxy based on IFC rules. Messages to and from a TPA are filtered and linked to IFC rules by services authorized to communicate with it. In comparison to DFT, the IFC framework controls the data flows at a coarser level and provides better performance, a necessary requirement when dealing with time-critical mechanisms.

IFC/DFT Interface: This interface concerns the services allowed to communicate with the TPA. The applications of the concerned services are not aware of this interface. Their middleware links IFC labels to DFT taints. The TPA is not assigned any label like a service, but a user identity is linked to it. Along with the taint field, an identity field is added to the header of each messages from and to the TPA.

Every authorized service can send data to the TPA without constrain. The service middleware makes sure to taint the message header with (3), if the data are sensitive for the car manufacturer. For the driver's private data, the service taints the message as (2) or (1) depending on the information sensitivity. The middleware adds the identity field of the user whose information are private and the DFT framework makes sure that this identity is the same as the one the TPA has been assigned to, otherwise it ignores the message.

For every tainted message coming from the TPA, the middleware adapts its processing based on the received taint:

- a taint (3) forces the middleware to make sure that the data do not leave the car. The proxy will not forward such data to the outside. A service, whose applications send information out (i.e., with a user tag in their label), will not process them.
- the taint (2) and (1) impose the middleware to own the user confidentiality tag or have it in its label, in order to pass the data to its applications. The difference between (1) and (2) allows the proxy to forward data with taint (2) only to very trusted external peer, e.g., the user's CE device and (1) to less trusted ones, e.g., Facebook.
- a taint (0) does not impose any constrain, the service can freely use the data in any way, it wants.

Traditional IFC [19], [21] classifies every output of a TPA, which receives sensitive information, as necessarily containing sensitive information as well. DFT allows us to follow the exact processing of the sensitive data, so that not all outputs of the TPA need to be considered highly classified. But DFT does not consider data integrity. All output of the TPA can be potentially dangerous. Only middleware layers performing input validation should be authorized to process these outputs.

IV. IMPLEMENTATION

This Section describes the integration of a DFT tool with our IP-based automotive middleware.

Middleware: For the implementation, we chose the middleware Etch, an open-source software project under the Apache 2.0 license. Etch proposes a modular and extensible architecture providing an efficient serialization and is considered as a serious candidate for the automotive purpose [22]. We used its C-binding and extended the header for our 2 middleware versions: with a field of 8 bytes for the IFC version (4 for the integrity tags, 4 for the confidentiality ones) and with a identity field of 4 bytes and a taint field of 4 bits for the DFT version (each bit expressing the presence of a taint). Label serialization/extraction and enforcement is performed in the software logic of the middleware.

Then we developed an Etch proxy in C, similar to the one in [11]. The proxy provides two secure communication interfaces: external over SSL/TLS and internal over IPsec. Internal and external communication partners communicate over a mirror-service, making the communication decoupling completely transparent. The proxy is application-unaware. Either it extracts the label/taint field (depending on the message target) from the payload of an outbound message and enforces the required policy, or it adds a label/taint field (depending on the message source) to the inbound message. User tags and identity for DFT are based on the name provided by the client certificate of the SSL/TLS connection.

The DFT Tool: We use the DFT framework libdft [8]. libdft relies on the Intel's Pin [13] for DBI, i.e., in order to inject custom code into an unmodified binary during runtime. libdft allows to instrument machine instructions, system calls to track data flows between registers and memory locations. It can also raise a warning or stop the runtime in case of unauthorized behavior. This tool provides good performance in comparison of other frameworks [18] and a well-defined API for a customizable security enforcement.

More than just using libdft, we extended its expressiveness to the four taints mentioned in Section III-A, one byte of process memory being tagged by two bits of the shadow memory. We limited ourselves to four taints, in order to keep the size of the shadow memory reasonable and to provide an efficient taint propagation and management, locally for the DFT tool and for the other services. We extended libdft with the possibility to taint differently a user input (i.e., input from the keyboard) from a file input. The framework manages the access to files present on the HU thanks to a white-list specifying for each TPA how to taint information read from a file and how data should be tainted in order to be written in a file. The framework monitors system calls related to network inputs and outputs. It allows us to taint data of the ingress traffic depending on the received taint value. For outbound messages, the framework automatically determines the taints of the payload and injects them in the header.

Testing environment: We performed the implementation and experiments of Section V-A on computers interlinked with Gigabit Ethernet and running standard 32-bit Fedora Linux on an Intel Atom N270 (1,6 GHz) with 1GB RAM. This configuration is comparable to current unix-based HUs, which operate at 1,3 GHz [23]. Besides, we did not do extensive modifications of the Etch middleware mechanisms, which already provides suitable performances when tested on a microcontroller [22]. Therefore we believe that the addition of a simple IFC/DFT access control layer should not significantly slow down the middleware. This needs, however, to be verified

for a more rigorous validation.

V. Evaluation

In order to evaluate our system, in this Section we quantify the overhead of our implementation and discuss its security.

A. Performance Evaluation

We measure the middleware throughput (in call/sec) between a CE device and an on-board TPA, running on the HU, in order to demonstrate the overhead of our DFT/IFC framework. Benchmarks are run on three separate machines running Etch services: a CE device, a proxy and the HU. The CE device sends a simple Etch message containing an integer to the TPA. The TPA retrieves a series of integers from a file on the HU. Based on the received numbers and information read from the file, it computes an answer and sends it back. We vary the size of the returned buffer to stress the middleware and taint propagation mechanisms. The TPA plays the role of a server, providing infotainment content to the CE device (e.g. music, picture). The messages go first through the proxy, then through a trusted HU service and finally to the TPA. IFC rules are enforced between proxy and HU service. Our results, in Figure 4, present the throughput performances of this scenario for various security levels and buffer sizes. We first measured them without any security feature enabled as reference (1). We then performed the same tests when adding the communication encryption (2) (SSL on the link CE device–Proxy and IPsec for the link Proxy–HU) and the enforcement of IFC rules (3), in order to determine a lower bound overhead imposed by the security framework without DFT. We finally repeated the measurements, while adding the DFT-based monitoring of the TPA (4) and evaluated its impact.

Discussion: We realized that, when normalized, the performances with the different enabled security features are proportionally the same regardless of the buffer size. We present the normalized results in Table I. It shows that the encryption is responsible for the most significant part of the system overhead (∼43%). The impact of the IFC framework between proxy and HU service stays minor in comparison (∼4%). The DFT monitoring decreases the system performance by 22%. This penalty is mostly due to the instrumentation of the sockets and the taint propagation. With DFT enabled, the use of bigger buffer is more suitable and achieves a relatively large bandwidth (up to 1,23 Mbit/sec). The use of DFT and IFC can therefore be suitable for infotainment use cases involving a CE device and requiring moderate bandwidth. Our example makes use of a communication between proxy and HU, in order to show the impact of using IFC and DFT simultaneously. Direct communications CE device–TPA over the proxy reach a bandwidth of 2,17 Mbit/sec with DFT and encryption.

However, our evaluation is mostly focused on our middleware in a small 3-node network for a specific scenario involving a simple TP application. Tests performed with libdft for bigger applications like a web-browser [8] or a MP3-player [18] have shown more significant latency. For optimal performances, the TPAs should remain small and simple and maximize the use of "trusted", i.e., non monitored, libraries. Additional investigations in larger network producing more traffic are recommended for further validation.

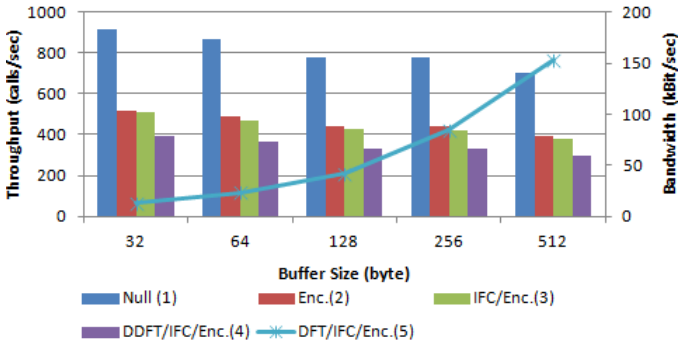

Fig. 4. Middleware throughput (1-4) and bandwidth (5) average for various buffer sizes and security features enabled (Enc.: The communications are encrypted, IFC: IFC rules are enforced, DFT: the TPA is monitored).

TABLE I
NORMALIZED PERFORMANCE OF THE SCENARIO IN SECTION V-A. FACTOR (I) AND (II) TAKE RESPECTIVELY (1) AND (3) AS REFERENCE.

	Null (1)	Enc. (2)	IFC/Enc. (3)	DFT/IFC/Enc. (4)
Factor (i)	1	0.57	0.55	0.43
Factor (ii)	-	-	1	0.78

B. Security Discussion

For this section we describe how our system would react to the attack scenarios of Section II-B. Both scenarios feature an attacker taking control of the TPA, e.g., through a buffer overflow vulnerability. The DFT framework is designed and configured to detect attacks involving the overwriting of stack pointers and can stop the application. As a result an attacker cannot compromise the integrity of the TPA.

About the integrity scenario: This scenario considers an unauthorized access to car resources that may disturb the car functionality, e.g., access to a process, file of the HU or to critical services on other ECUs. The DFT is configured to block every system call involving the access of shared memory, filesystem and inter-process communications and restricts the ones concerning the access to HU files and network sockets. The TPA is therefore constrained to write in the files that have been whitelisted by the rule set provided by the car manufacturer. In a same way, the TPA is able to communicate only with services that have been authorized and therefore never gets access to highly critical services, e.g., brake controller. The TPA cannot directly get access to a HU service, except through a socket, but it has to be authorized to do so. Besides, in case of an unsuitable rule set, the contacted middleware can still ignore the message if it considers it does not provide the necessary security mechanism, e.g., input validation. However, this system does not protect against denial-of-service attacks.

About the confidentiality scenario: This scenario mostly considers the release of sensitive information to the outside. As said earlier, the TPA is constrained to whitelisted files, its capacity to write and read are controlled as well. Every data

 ISBN: 978-1-61208-288-2

read from a file or received from another service are tainted with a value related to the sensitivity of the file or to the values present in the message header. This taint is propagated during runtime. In order to release data, the TPA either write the data into a file, whose access is whitelisted, or send them through the network to another authorized service. We do not consider information leakage through a file here and focus on the network exchanges heading out, i.e., through the proxy. The TPA may directly contact the proxy. The proxy, based on the taint of the message header, decides whether it may forward to the outside. The TPA may decide to choose an indirect way to reach the proxy: through another service, which communicates with the outside. When receiving a message from the TPA, the service middleware decides whether it can process the data or not. Having a user tags in its label generally means a high chance to forward data to the outside, so the middleware should refuse data tainted with (3). On the other hand, services with internal service tags, are not likely to have their information sent to the outside. However the tag-ownership concept may still allow such information to be sent out, therefore the decision to process (3)-tainted depends on the use-cases the ECU is involved in and is set by the car manufacturer. As for (2)- and (1)-tainted data, they can be processed only by services owning or labeled with the user tags related the user identity contained in the message header. This allows to share private information to services respecting the user's privacy. However this does not allow to maintain the difference between the 2 types of sensitivity.

Unlike OSes like Android, which controls applications with a limited set of coarse permissions, DFT allows a very fine granular security customization and enforcement. A main advantage of the DFT concerns the application flexibility. Even if the TPA receives sensitive data, the DFT framework determines and shares whether the outputs have to be considered as sensitive, or not. In addition, the coupling of IFC and DFT provides an efficient enforcement to secure internal information exchanges, while monitoring untrusted TPAs in contact with the outside world.

System limitation: We assumed in Section II-B that the integrity of the OS, the middleware and the DFT framework were ensured by a secure boot. However these mechanisms do not protect against runtime attacks, which could be significantly harmful when being performed on critical entities like the proxy or the HU. They may be detected by host-based intrusion detection tools performing scans and recognition of instruction patterns within a running platform [24]. Though these solutions might significantly degrade the system performance and should be used in a carefully selected manner.

VI. Conclusion

In this paper, we presented a security architecture, leveraging DFT engines to secure the on-board integration of automotive TPAs. Locally, the DFT framework controls and monitors the TPA against exploitation of security vulnerabilities. Regarding the network communications, the middleware-based exchange of taint information allows the DFT tool and the trusted services to preserve the data confidentiality and system integrity. Interface rules between labels and taints allow an efficient and simultaneous integration of the local DFT and the car-wide IFC. However while enhancing the car security, DFT/IFC tools have shown their limits in term of performance and can not be used for time-critical applications without further optimization but are suitable for an infotainment usage. Full virtualization solutions may be an efficient and secure alternative, that we intend to investigate.

References

[1] K. Koscher et al, "Experimental security analysis of a modern automobile", in proc. of the 31st IEEE S&P, 2010, pp. 447–462.

[2] Z. Lutz, "Renault debuts r-link", Engadget and Renault press release at leweb'11, December 2011 (retrieved 2013).

[3] E. Slivka, "Apple pulls russian sms spam app from app store", http://www.macrumors.com/2012/07/05/apple-pulls-russian-sms-spam-app-from-app-store/, 2012 (retrieved May 2013).

[4] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou and Y. Wu, "Lift: a low-overhead practical information flow tracking system for detecting security attacks", in proc. of MICRO–39, 2006, pp. 135–148.

[5] H. Yin, D. Song, M. Egele, C. Kruegel and E. Kirda, "Panorama: capturing systemwide information flow for malware detection and analysis", in proc. of the 14th CCS, 2007, pp. 116–127.

[6] W. Enck et al, "Taintdroid: an information-flow tracking system for realtime privacy", in proc. of the 9th OSDI, 2010, pp. 393–407.

[7] Etch homepage. http://incubator.apache.org/etch/ (retrieved May 2013).

[8] V. Kemerlis, G. Portokalidis, K. Jee and A. Keromytis, "libdft: practical dynamic data flow tracking for commodity systems", in proc. of the 8th ACM SIGPLAN/SIGOPS VEE, 2012, pp. 121–132.

[9] A. Maier, "Ethernet - the standard for in-car communication", in 2nd Ethernet & IP @ Automotive Technology Day, 2012.

[10] A. Bouard et al, "Driving automotive middleware towards a secure ip-based future", in proc. of the 10st ESCAR, 2012.

[11] A. Bouard, J. Schanda, D. Herrscher, C. Eckert, "Automotive proxy-based security architecture for ce device integration", in proc. of 5th MobileWare 2012, Springer, 2012, pp. 62–76.

[12] Fujitsu Semiconductor Europe, "Fujitsu announces powerful mcu with secure hardware extension (SHE) for automotive instrument clusters", In Fujitsu Press Release at www.fujitsu.com, 2012 (retrieved May 2013).

[13] Intel's Pin homepage. http://www.pintool.org/ (retrieved May 2013).

[14] M. Dalton, H. Kannan and C. Kozyrakis, "Real-world buffer overflow protection for userspace & kernelspace", in proc. of the 17th USENIX SEC, 2008, pp. 395–410.

[15] B. Davis and H. Chen, "Dbtaint: cross-application information flow tracking via databases", in proc. of the 1st USENIX WebApps, 2010.

[16] Q. Zhang et al, "Neon: system support for derived data management", in proc. of the 6th SIGPLAN/SIGOPS VEE, 2010, pp. 63–74.

[17] A. Zavou, G. Portokalidis and A. Keromyitis, "Taint-Exchange: a generic system for cross-process and cross-host taint tracking", in proc. of the 6th IWSEC, 2011, pp. 113–128.

[18] H. Schweppe and Y. Roudier, "Security and privacy for in-vehicle networks", in proc. of the 1st IEEE VCSC, 2012.

[19] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières, "Securing distributed systems with information flow control", in Proc. of the 5th USENIX NSDI, 2008, pp. 293–308.

[20] A. Ramachandran, Y. Mundada, M. Tariq and N. Feamster, "Securing enterprise networks using traffic tainting", in Special Interest Group on Data Communication, 2008.

[21] A. Bouard, B. Weyl and C. Eckert, "Practical information-flow aware middleware for in-car communication", submitted to CyCar'13, 2013.

[22] K. Weckemann, F. Satzger, L. Stolz, D. Herrscher and C. Linnhoff-Popien, "Lessons from a minimal middleware for ip-based in-car communication" in proc. of the IEEE IV'12, 2012, pp. 686–691.

[23] BMW AG. Navigation System Professional, http://www.bmw.com/com/en/insights/technology/technology_guide/articles/navigation_system.html (retrieved May 2013)

[24] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection", in proc. of NDSS symposium 2003, Internet Society, 2003.

 ISBN: 978-1-61208-288-2