Security and Privacy for In-Vehicle Networks

Hendrik Schweppe, Yves Roudier

EURECOM
Sophia-Antipolis, France
{schweppe, roudier}@eurecom.fr

***Abstract*—Mobile devices such as smartphones have gained more and more attention from security researchers and malware authors, the latter frequently attacking those platforms and stealing personal information. Vehicle on-board networks, in particular infotainment systems, are increasingly connected with such mobile devices and the internet and will soon make it possible to load and install third party applications. This makes them susceptible to new attacks similar to those which plague mobile phones and personal computers. The breach of privacy is equally sensitive in the vehicular domain. Even worse, broken security is a serious threat to car safety. In this paper, we show how traditional automotive communication systems can be instrumented with taint tracking tools in a security framework that allows to dynamically monitor data flows within and between control units to achieve elevated security and privacy.**

On-Board Systems; In-Vehicle Network Security; Privacy; Data Flow Tracking; Distributed Tainting; Binary Instrumentation

I. Introduction

Despite the fact that vehicles still serve their primary purpose, transportation, we have seen a shift of this paradigm: entertainment platforms and commodity functions have conquered a large portion of the vehicle and its electronics, demanding for greater processing power. At the same time, highly sophisticated and electronically triggered safety functions have been installed. A typical on-board architecture of vehicles, that we describe in more detail in the following section, is based on paradigms from the embedded world and thus allows little room for additional security features by means of processing power and bandwidth. Recently, in-vehicle networks with larger bandwidth, such as Ethernet have been introduced into the architecture [1].

In the last years, such a paradigm shift could also be observed for mobile phones: While phones were originally purely communication devices that allowed the user to place calls, they have evolved into personal computing devices, which are always connected and store a multitude of personal information. The attention of malfaiteurs has since lead to increased attacks on these platforms in order to steal data.

Data Flow Tracking is a technique that can be used for various purposes. It is used in a static manner to validate program behavior (static source code analysis) or at runtime to analyze malware or to debug software. It has recently gained attention in the field of information flow control, i.e., controlling which piece of information may be processed and stored at which places. Examples include a modified runtime environment of the Android Dalvik/Java interpreter [2], which follows the path of data through the execution and can, if the data is misused, trigger an alert or inhibit the information leak.

Our contributions are:

- We present the first of its kind architecture that introduces data flow tracking into the automotive on-board network.
- Our approach successfully hardens vehicle security in two important aspects:
  1) Privacy—data can only be used for its intended purpose in applications, i.e., the *information flow* is monitored and controlled.
  2) Security—data used in the execution of code, e.g., the return address on the stack used by a RET instruction is monitored for the absence of tags, that would reveal an integrity compromise.
- We implemented a working prototype that integrates a tainting-engine with an automotive security framework.
- We show the feasibility to transport tags among network nodes by extending the communication payload.
- We show that acceptable overhead may be attained in the near future, using optimized binary optimization for taint processing.

This paper is organized as follows: we first give a brief overview about the in-vehicle architecture of our use case. In Section III, we introduce the methodology and tools that were used for data flow tracking and describe the model in detail. We present our implementation and performance measurements in Section IV, then refer to related work in Section V and finally conclude the paper with a summary and future work.

II. In-Car Architecture

Vehicle networks are a compound of Electronic Control Units (ECUs) and on-board buses, which form a network around specific domains (e.g., powertrain and cabin control).

Our scenario is centered around the so-called “head unit”, the central hub for *infotainment*, i.e., information and entertainment. Information that is available to this device includes privacy relevant data: while the interest of in seat-position and the approximate driver’s weight (occupancy-sensors) is arguable, the head unit will also know where the vehicle is moved (GPS) and capture what is said inside the car (microphones). The head unit is connected to a number of other controllers via the internal vehicle buses. In contrast with the aforementioned privacy problem which does not endanger car satefy, active attacks, influencing the vehicle behavior via

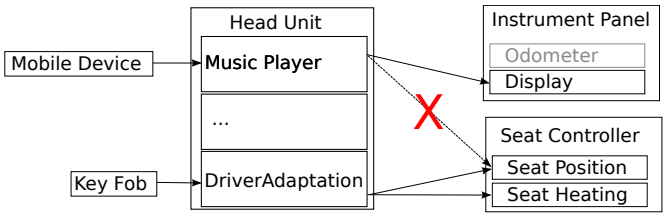

Fig. 2. Multiple applications are located at the vehicle's head-unit. They have access to different services via the network. Access control must be detailed at the level of applications, as network-based rules would not suffice to detect and prevent malicious messages, e.g., an attempt from a compromised music player to control the seat position.

fraudulent messages on the on-board network are possible, as impressively shown in [3] and [4].

Head Unit APIs: MeeGo IVI (In Vehicle Infotainment) is one branch of the MeeGo Linux project, that is used as implementation base by the Genivi alliance [5], a consortium of manufacturers and suppliers, who promote the open development of an infotainment unit. The anticipated MeeGo-IVI API [6] depicted in Fig. 1 illustrates the possibilities of interaction with vehicle service and how powerful the head unit of a vehicle can be.

Scenario: Our use case features two applications: the first one is a music player, which reads user-supplied files and displays information about the currently playing title on the instrument panel's display. The second application is a commodity function. The user's car key is identified (each key fob carries a unique identifier) and the vehicle's parameters are adjusted according to his or her preferences (driver profile saved in a file). We use the seat controls as an example of personalization. Remote access to vehicle functionality are features common to modern upper class vehicles, e.g., sending POI (Point Of Interest) coordinates to the head unit or controlling the music player from a mobile phone, as in our use case.

III. Data Flow Tracking

On-Board applications are designed in a very distributed and communication-centric fashion. Applications are built as small function blocks following a component based model that emphasizes the connection of inputs and outputs to basic or composite software modules. Those modules are finally mapped to different ECUs, thereby generating specific communication patterns. Automotive middleware such as AUTOSAR is generally used to support such architectures.

In this distributed software and hardware system, it is important to follow data not only within, but also between individual ECUs. For example, a successful attack via the tire-pressure monitoring system (shown in [7]), in which the sensors transmit their data wirelessly and unauthenticated, show that these data can compromise the display of the speedometer. Data like tire pressure sensor readings are also routinely broadcasted to the in-vehicle network and distributed to further ECUs. These data are, for example, then used as inputs for plausibility checks within the electronic stability control module or to trigger the "limp home" functionality (in case of a supposedly flat tire) within other units. If the original data has been maliciously crafted, an ECU using these data could be subject to an exploit. A possible overflow in a device like the electronic stability control can result in the brake system effectively being compromised, as it directly and individually controls brake actuators at all wheels.

Our approach makes use of the information flow (communication and internal calls) that is hedged by the middleware and the applications. This allows us to easily track in and outgoing information. Incoming data are tainted with tags we introduce. We verify the presence and absence of tags for outgoing data according to the ruleset introduced in the following section. The tainting engine that instruments all applications and libraries at runtime takes care of the propagation among the ECU's memory and registers during execution. As we want to track different kinds of sources, we use distinct taint tags, often called "colored taint".

A. Binary Instrumentation for Taint Tracking

In order to introduce taint tags into running applications, we make use of a technique that is called "binary instrumentation". This technique injects custom code to binaries at run-time, i.e., one can instrument machine instructions and system calls in order to follow the flow of data between registers or memory regions, as well as to take precautions if data is used in a questionable manner. One of the major advantages of binary instrumentation is that the source code of analyzed programs does not need to be available. This enables monitoring programs of unknown pedigree for compliance with given rules. We make use of Intel's Pin [8]. Pin is a generic tool for dynamic binary instrumentation, that provides the base for the dynamic taint tag analysis. It is available not only for x86 architectures (Windows, Linux, Mac), but also for ARM. We have conducted our experiments on IA32 Linux.

There exist a number of prototype implementations for host-based dynamic data flow tracking based on Pin [9]–[11]. We analyzed these prototype implementations for feasibility to integrate with our security framework developed in the scope of the EVITA EU project. We use this framework to trace data through execution on the platform and to propagate taint tags together with network streams within the car.

B. Data Flows: Access Control

Data can be exchanged in a multitude of ways:

- By accessing shared memory,
- Via filesystem access,
- Through inter-process communication,
- By using network frames.

To limit possible data and information leaks from functions, we first have to define which are the subjects in our model, i.e., at which granularity are data flows described. Granularities are (from fine to coarse): Instructions, Functions and System calls, Local programs, ECUs on a network.

While we cannot limit the data flow between processor instructions (e.g., moving between registers and memory), we

Group	Function	Parameter	Operation
Personalization	Driver Seat Position	Level per each part (recline seatback, slide, cushion height, headrest, back cushion, side cushion)	Get / Set
	Dashboard Illumination	% of Illumination (0 : Darkest, 100 : Brightest)	Get / Set
Driving Safety	Vehicle Top Speed Limit	km/h or mph (0 means no limitation)	Get / Set
	Door Lock Status	Lock/Unlock per each door (driver, passenger, rear left, rear right)	Get / Noti / Set

Fig. 1. Excerpt from the MeeGo IVI API draft from [6]. These functions are accessible from applications on a head unit running MeeGo. This unit is the centrally interconnected device of the vehicle, exchanging information with consumer devices (mobile phone) and internet services. Applications may interact with entertainment (music player), infotainment (navigation), and also core vehicle components, such as door locks.

can monitor the information flow at instruction level. This allows us to track data with a fine granularity and to enforce control on data on the stack, e.g., to verify that the return address of a function call has not originated from anywhere else than the function call itself. Therefore, we can eliminate a large amount of potential attacks that are based on overwriting the stack pointer, e.g., including buffer-overflows [12], format-string [13] and return-oriented-programming (ROP) [14] exploit techniques. If we would not have the ability to trace individual machine instructions, a function-based propagation would very quickly result in a state called over-tainting, as all outputs of a function need to be marked according to all input tags of the function. In addition, we can't directly prevent exploit techniques on the stack without instruction level information flow tracking.

C. Taint Based Security Policy

It is not adequate to build access control rules for individual processor instructions. Likewise, access control at the network level is too coarse (see Fig. 2 for an example). The adequate granularity to describe access control based on taints is tightly linked with the communication structure of programs and functions: we want to be able to control which program may communicate what *kind of data* with which partner. This approach is naturally modeled by a graph. In contrast to a standard graph with vertices v and edges e, we also use the kind of data as edge labels, to distinguish whether a certain data exchange (the edge of a graph) between programs (the vertices) is allowed or not. The labels of edges model the tags allowed for communication and consist of a list of taint tags t.

We define the set V to contain all vertices v of a system, E to contain all edges e, T to contain all tags t. We say that the function $\mathcal{T} : VAR \rightarrow \mathcal{P}(T)$ assigns to each variable $var \in VAR$ the set of tags belonging to it.

A positive ruleset (whitelist) of such communication contains the emitting program (source), the receiving program (destination) and a set of allowed tags T. Thus, a rule r is constructed as $r = (e, T)$, where $e \in V \times V$.

All vertices at the borders of the graph, i.e., those which have either no incoming or no outgoing edge, serve a special purpose. They are the *source* or *sink* of information. Fig. 3 shows an example of a tag propagation tree for the driver adaptation comfort function. This example consists of four rules: two in order to allow individual data flows from i) an RFID reader that receives the key fob's identifier and ii) a file on disk (both are local), and two rules to allow data streams from the driver adaptation application to two different

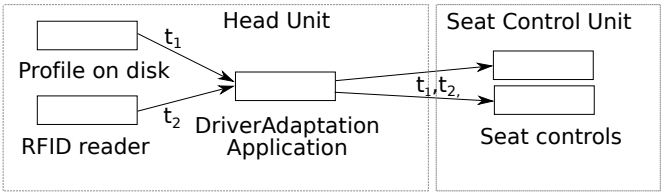

Fig. 3. Propagation tree for DriverAdaptationApplication. Data is tainted at source nodes (t_1,t_2). This corresponds to an I/O operation at the Head Unit. The application issues commands to the seat control unit via the network. The combined tags are conveyed over network communication and evaluated at the receiving side before the actual operation is triggered. They are also propagated throughout operation, to ensure maliciously crafted data do not change the application control flow.

services for seat controls, both allowed to carry data from corresponding input sources (t_1 and t_2). The latter two rules relate to network data, while the first two are local to the ECU.

a) Baseline rules: These rules have to be defined by the manufacturer a priori, e.g., as expert-knowledge along with the functions that are deployed. For the majority of vehicle functions, this does not pose any problem, as communication channels are known already at the design-phase of the on-board network and usually exist as structured data (e.g., the K-matrix for CAN buses). In the upcoming development paradigm associated with the AUTOSAR operating system, individual data exchange between programs and functions are modeled. Within AUTOSAR, the so-called *Virtual Function Bus* abstracts from physical networks: communication is modeled in UML as direct connections between programs and functions, that are later grouped into composite functions and finally mapped onto network or local communication. This facilitates the generation of the proposed rules, as all internal communication of control units is already known and described explicitly.

b) Application specific rules: For third party applications to be loaded onto the head unit, a ruleset needs to be supplied along with the application. This ruleset must be evaluated against the system's policies, i.e., the baseline rules. In addition one can imagine that the user may grant certain additional capabilities. This is similar to, but more detailed than Android's security manifest, for example. While the granularity for Android security rules is very coarse (e.g., requires network access or not), our rules include the kind of data that was used to generate specific network requests, for example. In the example of Fig. 3 one sees that the control request carries tags related to i) the key fob identifier (read via an RFID reader) and ii) the user profile (read from disk). This

allows to effectively limit access to the vehicle functionality and leakage of personal data.

D. Example of Tag Propagation

The propagation of tags happens at every intermediate processing step. For the purpose of controlling the origin of (possibly untrusted) data, we use the unification of tags as common propagation-logic.

An example for the unification of tags is given in the following listings:

Listing 1. Code Example of Taint Tag Propagation

```
data x,y;
data z;

x = read(sensor-input);
y = read(file-input);

z = concat (sensor-input, file-input);

write(output,z);
```

For tag marks t_1 and t_2 that represent sensor-input and file-input, this results in the following tags at the end of the execution: $\mathcal{T}(x) = t_1$, $\mathcal{T}(y) = t_2$, and $\mathcal{T}(z) = t_1, t_2$.

E. Network Marshalling

In order to follow data throughout the execution over multiple processing stations, our framework automatically adds the according tags for data contained in network messages and signs the message with a previously exchanged, application-specific symmetric session key. We use a low-cost symmetric message authentication code (MAC) to secure the payload. This MAC, in combination with a key distribution protocol, assures robustness against man-in-the-middle attacks. We have shown the feasibility of using application specific usage-controlled symmetric session keys in our prototype in [15]. An optional timestamp protects the payload against replay attacks.

The set of tags that is added to the payload corresponds to the propagation scheme used for all possible segments of the payload. We use a union of all tags of all n segments, i.e.,

$$T_{msg} = \bigcup_{i=1}^{n} \mathcal{T}(s_i).$$

F. Enforcement

The rule based policy is enforced at two levels. First, the immediate enforcement is done at certain instructions (RET, JMP, etc.) and system calls (execve, system, etc.), where we require that no tag is present at the according arguments (e.g., a return address). This enforcement is part of the tag engine and introduced via binary instrumentation. Secondly, whenever middleware functions are called, e.g., to send data over a remote communication channel, input data is analyzed for the *presence* and *absence* of tags corresponding to the according rule (e.g., t_1, t_2 and no other tag must be present when the DriverAdaptationApplication sends requests, cf. Fig. 3).

IV. Implementation

Applications in our prototype are implemented using a C-style middleware, that establishes communication links to other entities (UDP/TCP/CAN-TP) and secures the payload accordingly. This is where taint tracking was introduced. The middleware can access vehicle-wide security services, such as single-sign-on or key distribution over an ASN.1 RPC interface, that is offered at master nodes implemented in C++. Details on this architecture can be found in the deliverables D3.2 [16] and D3.3 [17] of the EVITA project [18].

Whenever a program reads from I/O channels that are configured as *taint sources*, the corresponding data are marked. In contrast to approaches which primarily focus on avoiding run-time attacks (using a single binary tag per byte), we use distinct tags in order to distinguish between different sources.

The dytan taint analysis engine is attached as a Pin tool, i.e., the individual programs run instrumented. Additionally to tagging I/O data at our middleware functions, dytan possesses of an XML configuration file. The configuration lists network streams and files that should be regarded as additional taint sources. Generic system calls such as *read/write* are instrumented accordingly. Dytan allows a large number of distinct tags to be followed in shadow memory. They are held in an STL map. While this kind of implementation offers great flexibility, we have seen that it performed rather poorly, compared to other engines. However, it has some distinct features (the virtually unlimited number of tags) that were advantageous to show the interest of our approach. We compare it to the performance oriented engine "libdft" in the next subsection.

We have currently limited the number of tags to 32, in order to place them into a four byte field to ensure a lower overhead for network propagation of the taints.

We used a standard Ubuntu Linux (v2.6.38) on an i5 processor. We compiled all programs in 32bit mode and without MMX and SSE instructions, as the tainting engine does not support these registers nor 64 bit operations.

Performance

Instrumenting binaries with additional code incurs runtime penalties, as for each original instruction, additional instructions are injected, thereby slowing down the overall execution time. As demonstrated in [10] with the libdft implementation, an optimization of tag-propagation instructions towards modern processors, which feature multi-stage pipelines and jump-predictions, can result in much higher performance, compared to heavier implementations. Unfortunately, the original libdft implementation only allows to track binary taints. While this is sufficient in order to detect previously unknown exploits that trigger buffer overflows and similar, it does not offer a tag-space sufficient for system-wide information flow monitoring. However, in order to show the impressive performance of their system, we also conducted experiments using libdft and included them in our results. An alpha of libdft version supporting 8 colored taints showed exactly the same performance.

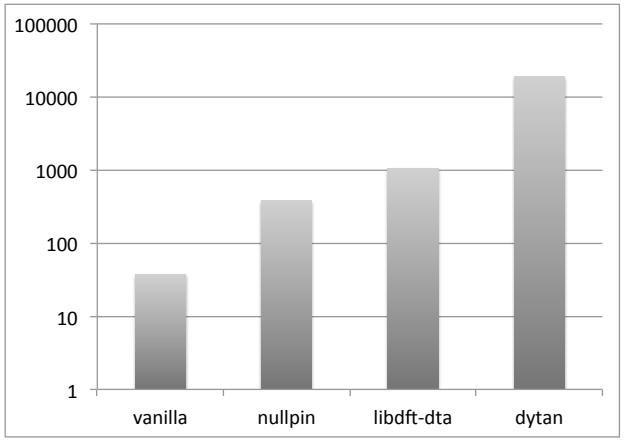

Fig. 4. Runtime of instrumented code in microseconds on logarithmical scale. Decoding of the first 100 frames of an MP3 file in i586 instructions. Unchanged code is 'vanilla' runtime, nullpin the instrumentation overhead, libdft-dta and dytan the corresponding frameworks.

We measured the runtime of a modified version of mpg123, which we integrated with our framework in order to send the MP3 ID3 tags to a remote display (see the example in Fig. 2). The runtime includes extracting the ID3 information, the process of marshalling information with taint tags and generating a MAC, as well as decoding the first 100 frames of the MP3 file, corresponding to 2 seconds of music.

	vanilla	nullpin	libdft-dta	dytan
runtime in ms	37	377	1045	18 925
factor (v)	1	10.2	28.2	511.5
factor (i)		1	2.8	50.2

Vanilla execution is the non-instrumented, plain execution of the code. The column "nullpin" relates to code instrumented with Pin, but without executing any hooks (part of libdft's sample tools). Libdft-dta and dytan are the actual tag-propagation engines. We show their overhead compared to vanilla execution (v) and instrumented execution (i).

Discussion: The performance results show that binary instrumentation results in the most significant part of the execution overhead (factor 10). A performance-oriented implementation of the tag engine itself, such as libdft, only incurs an overhead of 2.8. If the instrumentation itself can be rendered more efficient, the overhead of tracking data throughout execution is tolerable. While a small number of tags will not be able to construct a full-fledged information-flow system, it can still serve the purpose to limit data usage from certain sources (e.g., user input from media, internet or diagnosis), especially to limit leakage of private information. Other approaches to secure applications on the head unit, such as virtualization, incur a significant overhead as well.

V. Related Work

A. Exploit Prevention Techniques

Many techniques to defend against exploits on personal computers have been proposed, developed and deployed in normal installations during the last years. A non-executable stack (i.e., marking memory regions as non-executable) protects against the injection of shellcode [19]. Address-Space-Layout-Randomization (ASLR) introduced an additional burden, that protects against the return-to-libc exploiting technique, that was again defeated by return-oriented programming (ROP) [14]. A new technique of aligning code during compilation protects against ROP [20]. These techniques are slowly beginning to be integrated into today's CPU architecture, operating system memory management, and compilers. Such protections become prevalent in the PC environment, but they are only slowly taken up in the embedded world, e.g., ASLR was only introduced with iOS 4.3 and with Android 4.0. Similarly, it was just introduced with the latest version of QNX, a popular operating system for head units. However, as vehicles typically have a lifespan of over ten years, newly introduced exploit techniques likely apply to most vehicles on the streets.

Moving towards dynamic protection methods, runtime data flow tracking using input data tainting has been used. In contrast to dytan [9], the libdft framework [10] is specifically designed to run at low overhead and to provide a generic, yet flexible toolkit for taint analysis. The sample libdft-dta tool shows the applicability in an exploit-prevention context.

TaintExchange [21] was the first to demonstrate the feasibility of distributed taints using libdft. This system instruments all system calls used to exchange data between processes and to communicate over network sockets. In comparison, we are introducing the notion of a ruleset, which is used to describe acceptable behaviors of software modules.

B. Information Flow Control

The term information flow control originates at policies for large organizations with different clearance levels, i.e., some people can read more documents than others. In his work [22], Cheng introduced such clearance labels already at the middleware layer: he proposed a self-contained .NET based system that propagated those labels among remote execution and data exchange. Another interesting approach to limit sensitive data from leaving local applications was taken by [23]. For every process, they run a copy (a shadow process), that is only different in the detail, that private data are exchanged for random data. They compare the output of the programs (e.g., of write calls) and find, whether the result of the original and shadow process are the same (no private data used) or differ (private data was used in calculating the results). They claim that their approach is up to thirty times faster than taint tracking. While this is an interesting approach to avoid data leakage in enterprises, it offers no protection against platform attacks, which is one of our goals. Dynamic information flow analysis of programs in order to analyze malware, has also been done with the help of emulated environments, e.g., Valgrind [24] or QEMU [25]. While these approaches are well suited to analyze given malware programs, it can hardly be applied to our application domain. There also exists a tool based on Pin binary instrumentation for Windows,

that allows the user to trace whether certain input leaves a program via the network [11].

C. Current On-Board Security

In the last two years, the automotive on-board network has come to the attention of security researchers. We have seen that a large effect (i.e., the complete control of a vehicle) can be achieved with relatively common techniques locally [3] and remotely [4]. Vehicles have been criticized for broadcasting private data and not performing input validation [7]. Upcoming Car2X communication demands for higher security levels and introduces hardware security [26].

VI. Conclusions and Future Work

We can successfully mitigate two types of attacks on automotive on-board networks: those that break the system with classic exploit techniques (e.g., buffer overflows, etc.) and those which try to access data and services outside of their foreseen application data flow. With our approach we make sure that private information, e.g., data recorded from microphones, can only be used as anticipated.

While we have certainly shown that the methodology can be implemented and is a good contribution towards vehicular on-board security, our performance results show that the information flow tracking techniques are not ideally suited for an embedded deployment, but that certain optimizations make them seem feasible. A major trade-off can be observed between the use of binary instrumentation for information tracking (many tags needed) and for defeating attacks (only binary tag needed). If only few sources need to be tracked, libdft shows that one-byte taints, allowing for eight distinct tags, can achieve the same performance as binary taints. If we limit the use of dynamic information flow analysis to few sources (e.g., the most critical interfaces of vehicles: diagnosis, consumer electronics, internet, C2X and external sensors), an adequate performance can be maintained. As an example, a remotely triggered emergency brake could demand, in addition to existing cryptographic security, the presence of tags from Car2X and from radar sensors (used to double-check remote messages), and the absence of all other tags, e.g. those introduced at consumer-controlled interfaces. This results in a minimum of three tag sources.

Our solution is not targeting today's cars, as it involves updating the ECU's program code for both, information flow analysis and communication protocols. Although it can not be directly applied to today's vehicles, it may constitute a valuable line of defense for future vehicles.

Acknowledgements

We would like to thank James Clause for letting us use the dytan code and Vasileios Kemerlis for giving us access to an alpha version of libdft. We would also like to thank Benjamin Weyl, Alexandre Bouard, and Jonas Zaddach for their fruitful discussions and contributions.

References

[1] R. Bruckmeier, "Ethernet for Automotive Applications," in *Freescale Technology Forum*, Orlando, 2010.

[2] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, L. Jung, P. McDaniel et al., "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *9th OSDI, USENIX*, 2010.

[3] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway et al., "Experimental security analysis of a modern automobile," in *31st Security and Privacy*, IEEE, 2010.

[4] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage et al., "Comprehensive experimental analyses of automotive attack surfaces," in *20th USENIX conference on Security*, 2011.

[5] Linux Foundation, "Meego software platform chosen by the GENIVI alliance," online: http://tinyurl.com/mgoGNVI, July 2010.

[6] R. Streif, S. Bolek et al., "MeeGo: In-vehicle api." [Online]. Available: http://wiki.meego.com/index.php?title=In-vehicle/Roadmap/API&oldid=44637&printable=yes

[7] I. Rouf, R. Miller, H. Mustafa, T. Taylor, S. Oh, W. Xu et al., "Security and privacy vulnerabilities of in-car wireless networks: a tire pressure monitoring system case study," in *19th USENIX Security*, 2010.

[8] C. Luk, R. Cohn, R. Muth, H. Patil, and A. Klauser, "Pin: building customized program analysis tools with dynamic instrumentation," *ACM SIGPLAN*, 2005.

[9] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *ISSTA*, 2007, pp. 196–206.

[10] V. P. Kermelis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: Practical Dynamic Data Flow Tracking for Commodity Systems," in *VEE'12, SIGPLAN/SIGOPS*, 2012.

[11] D. Zhu, J. Jung, D. Song, and T. Kohno, "TaintEraser: protecting sensitive data leaks using application-level taint tracking," *ACM SIGOPS Operating Systems Review*, 2011.

[12] E. Levy (Aleph One), "Smashing the stack for fun and profit," *The Phrack Magazine*, vol. 49, no. 14, 1996.

[13] scut, "Exploiting format string vulnerabilities," TESO Security Group, Tech. Rep., 2001.

[14] H. Shacham, M. Page, B. Pfaff, and E. Goh, "On the effectiveness of address-space randomization," in *11th CCS*, 2004.

[15] H. Schweppe, Y. Roudier, B. Weyl, L. Apvrille, and D. Scheuermann, "Car2X communication: securing the last meter - A cost-effective approach for ensuring trust in Car2X applications using in-vehicle symmetric cryptography" in *WIVEC* 2011, September 2011.

[16] B. Weyl, M. Wolf, F. Zweers, T. Gendrullis, M. S. Idrees, Y. Roudier et al. "Secure on-board architecture specification," EVITA Project, Tech. Rep. D3.2, 2010.

[17] H. Schweppe, S. Idrees, Y. Roudier, B. Weyl, R. El Khayari, O. Henniger et al., "Secure on-board protocols specification," EVITA Project, Tech. Rep. D3.3, 2010.

[18] "The EVITA project," http://evita-project.org/, August 2008.

[19] A. Peslyak (Solar Designer), "Non-executable stack patch," Linux Kernel Patch, 1997.

[20] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-free: Defeating return-oriented programming through gadget-less binaries," in *26th ACSAC*, 12 2010.

[21] A. Zavou, G. Portokalidis, and A. Keromytis, "Taint-exchange: a generic system for cross-process and cross-host taint tracking," *6th ACS*, 2011.

[22] W. W.-Y. Cheng, "Information Flow for Secure Distributed Applications," PhD Thesis, MIT, 2009.

[23] J. Croft and M. Caesar, "Towards Practical Avoidance of Information Leakage in Enterprise Networks," in *6th USENIX HotSec*, 2011.

[24] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *NDSS*, 2005.

[25] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *14th CCS*, 2007, pp. 116–127.

[26] P. Papadimitratos, L. Buttyan, T. Holczer, E. Schoch, J. Freudiger, M. Raya et al. "Secure vehicular communication systems: design and architecture," *IEEE Communications Magazine*, vol. 46, no. 11, pp. 100–109, 2008.