ECUInjector: An Automotive ECU Security Analysis Tool

James Todd, Aidan Grant, Tim Muller and Xavier Carpent
School of Computer Science, University of Nottingham, UK

james.todd@nottingham.ac.uk
aidangrant27@gmail.com
tim.muller@nottingham.ac.uk
xavier.carpent@nottingham.ac.uk

Abstract: Electronic Control Units (ECUs) are responsible for a significant number of systems in modern day vehicles. If these systems are compromised, the resulting attacks will result in road accidents. Therefore, increased adoption of these systems has significantly widened vehicle attack surfaces. As a result, it is important to devise methods of identifying and investigating vulnerabilities within automotive control systems, allowing them to be patched before they are exploited by an attacker. To this end, we present *ECUInjector*, an open-source application designed to assist in the discovery of vulnerabilities within diagnostic interfaces and ECU communications. The software is designed in a modular way allowing it to be universal across ECU vendors and provides the ability to communicate via vehicle diagnostics buses and build vulnerability detection tools that can assist in locating potential weaknesses. We also provide some of our applications of using the software to communicate with and analyse various ECUs.

Keywords: Automotive security, Reverse-engineering, Embedded system security

1. Introduction

Electronic Control Units (ECUs) are responsible for a significant number of systems in modern day vehicles, particularly in the areas of powertrain, safety and comfort. Vehicles now contain hundreds of these units (Dijk, 2017), replacing many components that were once purely mechanical. Whilst this has been positive move in terms of efficiency and customisability, the increased adoption of such systems has significantly widened vehicle attack surfaces. As with other computer-controlled systems (e.g., networking hardware and consumer devices), vulnerabilities may exist within ECUs that leave them susceptible to attack, increasing the likelihood of road accidents (Onuma, Terashima and Kiyohara, 2017). On top of this, the constant development of these systems, increases the likelihood of vulnerabilities being introduced (Shin *et al.*, 2011).

Interconnectivity between ECU systems, as well as external diagnostic interfaces are a major target for attackers. Interconnected buses provide a means for one ECU to performed attacks towards other ECUs. On top of this, ECU diagnostic interfaces provide advanced functions (such as firmware modification), which can make it trivial for attackers to manipulate systems. It is therefore important to explore methods that vulnerabilities within automotive control systems can be identified and reported, before they can be exploited by malicious parties. This enables patches to be released that can fix these vulnerabilities, protecting road users.

With this in mind, this paper presents *ECUInjector*, an open-source application designed to assist in the discovery of vulnerabilities within diagnostic interfaces and inter-ECU communication systems. The application provides the ability to establish serial communications with various automotive control systems. On a basic level, commands can be sent and received to connected ECUs, whilst on a more complex level, advanced reverse-engineering scripts and tools can be developed to help locate vulnerabilities and weaknesses. Due to the number of standardised communications protocols that exist, *ECUInjector* has been designed using a modular approach, enabling protocols to be added as required. The software is also multi-platform, enabling its use across Windows and Mac systems.

2. Background

Modern vehicles contain numerous electronic control systems that are relied upon to monitor and control functions including engine management, braking, passenger safety and entertainment. Various legislative standards (US Environmental Protection Agency, 1990; European Union, 2002) have been introduced over the past 30 years requiring the use of these systems (with a particular focus towards On Board Diagnostics (OBD) (Gardetto, Bagian and Lindner, 2005)) from both a safety and environmental standpoint. Increased demands for driver assistance and comfort features have also led to an increase in the adoption of ECUs, resulting in new vehicles containing more than one hundred individual units (Prabhakaran, Thirumoorthi and Sri Dhivya Krishnan, 2024).

2.1 Inter-ECU Communications

ECUs within a vehicle are interconnected to enable the exchange of data, allowing sensor data from one system to be consumed by another. Road speed, for example, is a value often required by multiple ECUs, however only one ECU will have the physical sensor connections to determine the value. This information will therefore be shared onto the bus for other ECUs to make use of.

Technologies such as CAN (ISO, 1993), FlexRay (ISO, 2017) and LIN (ISO, 2016) are all examples of protocols that are utilised to implement ECU communications buses. These specifications provide guidance for the physical (wiring specifications) and data link (packet structures) layers of the buses, however the implementation of the application logic remains ECU specific. Modern vehicles often utilise multiple independent buses for different systems, with gateway ECUs providing a way to pass data between them (Robert Bosch GmbH, 2007).

2.2 Automotive Diagnostics

On-board Diagnostics (OBD) (European Union, 1998) provides a way to program and diagnose the ECUs within a vehicle and has been a legal requirement since the early 2000s (European Union, 2002). Diagnostics is performed via a dedicated OBD (J1962 (SAE, 2015)) port, which connects a diagnostics device into the vehicle's communications buses, allowing it to perform individual operations on each ECU. Dedicated protocols are defined for diagnostics communications, with some manufacturers implementing their own standards before the introduction of legislated protocols such as KWP2000 (ISO, 2000), UDS (ISO, 1998) and later DoCAN (ISO, 2004), which is utilised in vehicles today.

Diagnostic standards were adopted as part of legislation to enforce access to emissions related data, with protocols such as KWP2000 and DoCAN providing dedicated functions to ensure their compliance. Manufacturers often extend these protocols to support their own privileged operations, such as factory test procedures, dealership programming operations, and firmware updating mechanisms. Due to the advanced nature of these operations, access to these privileged operations is a security threat, and their presence therefore widens the attack surface of the vehicle. As a result, these advanced functions are often protected by security mechanisms designed to restrict access to those unaware of the relevant unlocking procedure (Van den Herrewegen and Garcia, 2018). Many early systems utilised basic "seed-to-key" algorithms, which could easily be circumvented, thus allowing malicious parties to gain access and leverage privileged operations to perform attacks (Pełechaty and Konieczny, 2024). Modern systems are adopting more secure methods using technologies such as certificates.

3. Related Work

The following section details some existing ECU communications and security analysis tools, as well as some existing work being carried out into automotive security research and the methods utilised in each case.

3.1 Existing Software

Software that interfaces with ECUs is widely used outside of security contexts, with a prevalent area being vehicle modification and independent repair. Remapping software such as MEMS Tools (Revill, 2026) and BimmerEditor (Bim-Tun, 2020) are primarily designed for tuning specific ECUs and utilise diagnostic interfaces to read and modify firmware. These tools typically provide graphical user interfaces that visualise engine calibration data as 2D or 3D tables, enabling users to adjust parameters with minimal effort; some tools additionally include advanced features such as live remapping and debugging. To utilise these tools, an inexpensive OBD-to-USB dongle is required. As such, with little understanding of the underlying diagnostic protocols, advanced modifications can be made due to their low barrier to entry; combined with their generally low-cost, these tools are attractive to hobbyists and independent tuning shops. However, due to being targeted at remapping specific ECUs, these tools tend not to be universal and provide limited access to the diagnostic interface beyond what is required for their dedicated functions.

General purpose libraries for ECU communications are also available but tend to focus on either a single or few diagnostics protocols exclusively. Python libraries, such as "python-udsoncan" (Pier-Yves Lessard, 2026) and "KWP2000-CAN" (Elias Kotlyar, 2026) provide low-level communications support for ECUs and expose defined diagnostics services as a Python API, which can be utilised across many platforms. These libraries reduce the time needed to develop scripts and tools to interact with ECUs and offer a large amount of flexibility as the open-source nature of these libraries enables modification as required. However, these libraries do not stand on their own in terms of them being stand-alone applications, so provide no GUI or pre-existing reverse-engineering functionality, and therefore require development expertise and time to utilise them in this way.

Commercial automotive software solutions, such as Vector CANoe (Vector, 2026), offer the ability to simulate and test ECUs and their diagnostic interfaces. CANoe, enables the testing of the diagnostic interface through an automated extension and using a range of protocols including UDS and KWP2000. Other tools such ETAS INCA (ETAS, 2026) and dSpace ControlDesk (dSpace, 2026) include comparable ECU and diagnostic simulation interface capabilities. These software suites are aimed at automotive ECU manufacturers who need to verify the implementation of their systems, and as such require costly commercial licences. This software is designed for validating implementations, usually focusing on accurate reporting of diagnostic trouble codes and API commands and are not aimed at security analysis or vulnerability identification. Also, due to the software being aimed at developing new systems, they often lack support for older communications standards, such as ISO9141 (K-Line).

3.2 ECU Security Research

The security mechanisms of UDS (and by extension KWP2000) are frequently targeted during security analysis of diagnostic interfaces. Pelechaty et al. (Pełechaty and Konieczny, 2024) explore the shortcomings of various randomly selected ECU diagnostic implementations using a custom test harness based on an off-the-shelf microcontroller. Through black-box penetration testing, vulnerabilities are found in the implementation of the "SecurityAccess" API method, which is based on a manufacturer specific seed-to-key algorithm, with unsuitable key lengths utilised that are susceptible to brute-force attacks. The same algorithms are also identified across multiple ECUs. In another study, Lauser and Krauß (Lauser and Krauß, 2023) use formal analysis techniques to identify vulnerabilities in security functions of both KWP2000 and UDS, which both utilise seed-to-key as well as more sophisticated cryptographic certificate-based protocols. They find variants of the included authentication protocols that are potentially vulnerable and confirm weaknesses in the KWP2000 security service attributed to weak standards and poor implementation. In a later study, Lauser et al. (Lauser, Munoz Molto and Krauß, 2024) use their previous findings to simulate an attack on an ECU implementing a weak variant of the authentication service. Using CANoe (Vector, 2026) to simulate a vulnerable ECU, they successfully execute a man-in-the-middle attack using a python script to intercept communication between the simulated ECU and an external testing tool.

These studies consistently show how vulnerabilities in diagnostic interfaces can be exploited using inexpensive hardware and software. Relying upon the manufacturer to implement the security mechanisms is pointed to as failures of protocols such as UDS. Despite these investigations exploiting the same diagnostic interface, the lack of standard tools used makes the work less repeatable. Studies often use custom test benches that are hard to replicate and tightly coupled to specific hardware setups, custom scripts or expensive software, limiting reproducibility and comparison of techniques. This lack of a common framework for analysing ECUs may also hinder the systematic analysis of non-security-related services. With evidence showing vulnerabilities occurring more commonly from implementation rather than protocol design alone, physical testing is a requirement.

4. ECUInjector

ECUInjector is an ECU reverse-engineering and vulnerability analysis tool, which allows researchers to communicate with automotive systems and work with the APIs and protocols to identify vulnerabilities. As demonstrated in Section 3, various software tools already exist that are extremely capable at communicating with ECUs, however they tend to either focus on one model or range (so their capabilities are often not transferrable) or not support vulnerability analysis. *ECUInjector* is maintained as an open-source project (available at: https://github.com/UoN-CybSec/ECUInjector).

4.1 Design Considerations

Having discovered that existing tools are either too specific to work on the majority of ECUs or lacking in necessary vulnerability analysis features, we designed the following requirements for *ECUInjector*:

- **Universal:** *ECUInjector* is compatible with many different ECU systems, protocols and implementations. This ensures that ECU vulnerability analysis can be carried out easily, without the need to build extensive software solutions, thus removing a barrier to entry (in terms of expertise and development time).
- **Diagnostic Protocol Support:** Enabling common protocols (KWP2000, CAN, KWP1281) to be defined and reused across projects, further reduces barriers to entry. Whilst many aspects of ECU design are specific to each ECU or manufacturer, many of the communications protocols that they utilise are

universal. Having a way to define a protocol and use this across projects enables a large part of the communications logic to be reused across ECUs that support the same protocols.

- **Custom Commands**: Whilst many diagnostic protocols contain a significant number of predefined operations, they also contain scope for ECU manufacturers implement own commands into the API, which are of particular interest to vulnerability analysis efforts. The tool therefore allows commands to be added on top of an existing protocol implementation to support custom operations. These are also particularly helpful when a manufacturer has not followed the protocol exactly, allowing small changes to be made to predefined command logic (discussed in Section 4.2.2).
- **Tool Creation:** On top of sending and receiving commands, the software supports the ability to create custom command scripts that can be used to build reverse-engineering tools against the diagnostic capabilities of the ECU (discussed in Section 4.2.2).
- **Cross Platform:** Use of C++ as the primary language for the application, as well as use of the standard library where required, enables *ECUInjector* to work across both Windows and Mac platforms, allowing researchers to use any system.
- **Graphical User Interface**: A GUI interface is utilised to present the necessary options to the user. The GUI contains a console which displays messages sent to and received from the ECU, as well as a command pallet and buttons to manage the active serial connection.
- **Project Based:** At any one time, users of the software may be working on multiple different reverse-engineering projects. It may also be necessary to share these projects between researchers, enabling collaboration. *ECUInjector* is therefore project based, with each ECU's commands, scripts and configuration stored individually.

4.2 Software Implementation

The system is designed around the concept of sending *commands* to ECU systems. These commands can either exist directly (in the form of hexadecimal data) or as scripts (which perform specific calculations and generate the relevant hexadecimal data accordingly). *Projects* are utilised to group a set of commands for a specific ECU. The below sections discuss these concepts in more detail.

4.2.1 Projects

Projects store relevant elements (commands, scripts and configuration) in a single organised location. Each ECU being analysed is given its own project, allowing for the centralisation of relevant commands and scripts. This is especially helpful If working across multiple different ECUs (an instance of the tool can be launched for each project). Collaboration is also enabled between researchers through sharing project files. Projects store the following elements:

- **Commands** created within the project, with their name (which is displayed in the UI), type (Raw or Script), value (for Raw scripts), and whether they should be triggered periodically.
- **Scripts** in the form of LUA source files, which are called when the associated command is triggered.
- **Globals** also in the form of LUA source files, which provide a way of defining centralised logic that can be used by the script commands (ECU commands, protocol implementations etc.).
- **Project Manifest File** storing the projects name (displayed in the GUI title bar) and intended for future expansion.

4.2.2 Commands

A primary feature of *ECUInjector* is its ability to exchange commands with ECUs. As discussed in Section 2, ECUs utilise many different protocols for communications (both in terms of diagnostics as well as inter-ECU buses). This remained in mind when designing the different types of commands, ensuring that extensibility was maintained where possible:

- **Direct commands** enable hex values to be entered straight into the command prompt, which are then delivered directly to the ECU, thus circumventing the parsing and interpretation elements of the software. ECU responses are monitored and, if received, will be displayed in the console. These commands are best used when performing initial experiments with an ECU's protocols, before creating more permanent commands once identified.

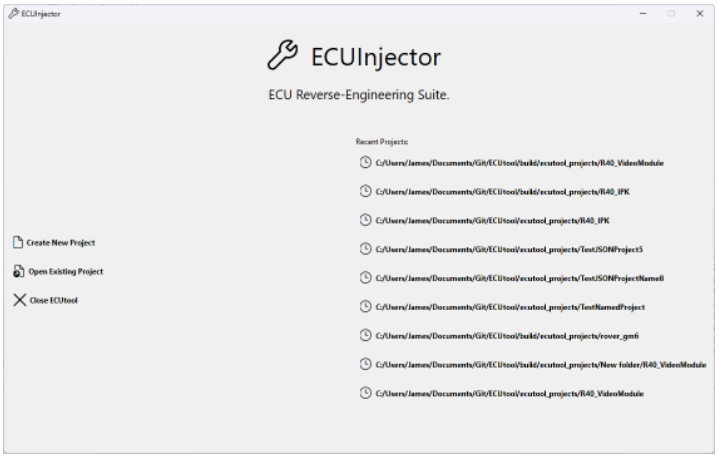

Figure 1: *ECUInjector* Start Window

- **Raw commands** work in the same way as direct commands, however, can be saved into a project for reuse. These commands also circumvent the parsing and interpretation elements of the software, and are particularly useful when manufacturers have utilised commands that do not entirely fit with existing protocols, or for simple experimentation with commands that have few static arguments. An added feature of static commands is their ability to be sent periodically, based upon a time interval, which allows them to be used as "keep-alive" signals, maintaining a diagnostic session with an ECU (after initialisation).
- **Script commands** have a LUA script file associated them which has access to an API to send and receive commands and write to the console. This gives them the ability to perform more complex operations that may require multiple command exchanges with calculations being performed in-between. These commands can be sent directly, or accompanied by arguments in the command prompt, enabling data to be fed into the underlying LUA script. The underlying LUA script has enabled implementation of "seed-to-key" authentication, firmware extraction, and building reverse-engineering tools.
- **Protocol Implementations** provide the ability to extend the software to support established communications protocols (such as KWP2000, CAN etc.), where packet structures, arguments and checksums are defined in C++ logic. Script commands can make use of this logic via the API within their associated LUA scripts.

4.3 Software Overview

The following section describes the process that a user takes to perform analysis on an ECU using *ECUInjector*.

4.3.1 Start screen

The start screen (Figure 2) provides users with the ability to create a new project (allowing them to define a folder to store the project's files), open an existing project, or close the tool. A list of recent projects is also displayed, allowing users to quickly access projects that they have been recently working on. Once a project has either been created or selected, the Diagnostics Screen is launched.

4.3.2 Diagnostics screen

The diagnostics screen is where ECU communication and analysis is carried out and provides the following functionality (see Figure 3):

- **Toolbar** (1)**:** Provides project operations (open, save, save as) and connection management.
- **Command Palette** (2)**:** Lists the commands that are part of the current project, with options to run, edit and delete each command and hide associated console output.
- **Console** (3)**:** Displays commands sent (from the software to the ECU), commands received (from the ECU into the software), and script output (which is defined within each of the LUA scripts associated with Script commands).
- **Command Prompt** (4)**:** Provides a way to manually enter hex commands to be sent directly to the ECU, and supply arguments to commands.
- **Status Bar** (5)**:** Displays serial port connection information (protocol, COM port, baud rate etc.).

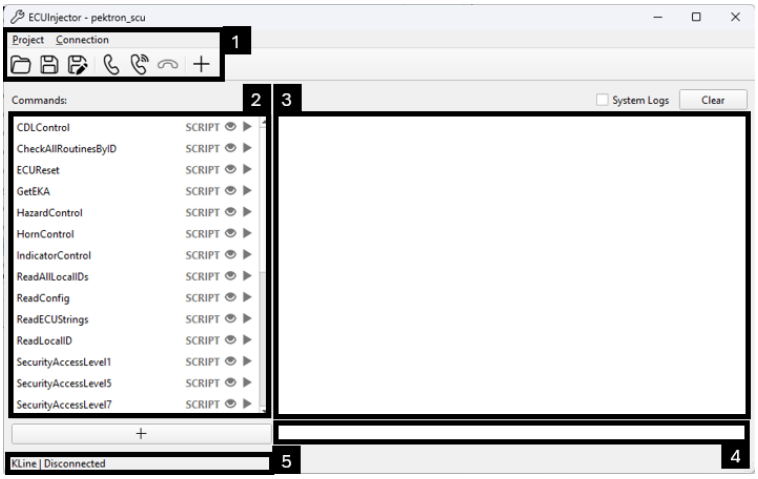

Figure 2: ECUInjector Diagnostics Screen

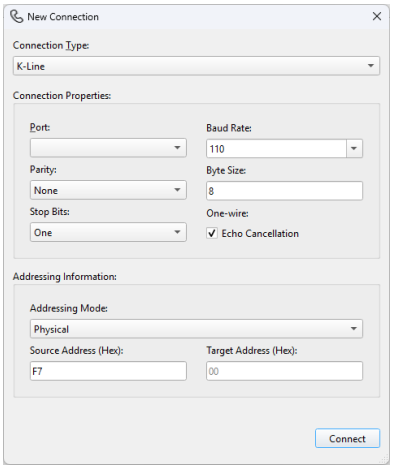

Figure 3: ECUInjector Connection Settings

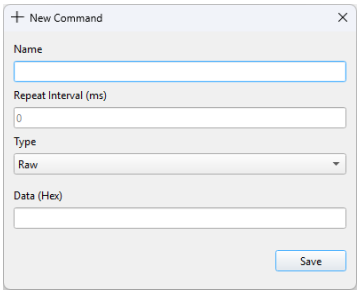

Figure 4: *ECUInjector* New Command Dialog

To begin communications with an ECU, a connection must be opened with a serial port. The "New Connection" (telephone) button, in the menu bar, launches the connection dialog (see Figure 4), where serial communication settings can be defined, including:

- **Connection Type:** The protocol defined within C++ logic that the ECU utilises. Options in this list represent protocol implementations {see Section 4.2.2) that have been defined within the software. Currently only K-Line (ISO9141 (ISO, 1994)) communications have been defined.
- **Connection Properties:** The serial port, baud rate, parity, byte size and stop bits. The "echo cancellation" option is a necessary option for certain automotive wiring standards where packets are transmitted and received on the same line, resulting in echo cancellation being required.

Once these details have been defined, the connection can be opened using the "Open Connection" button (transmitting telephone) in the menu bar and can be closed, when required, using the "Close Session" (on hook telephone) button.

Commands can be defined using the add (plus) button in the toolbar, which opens the "New Command" dialog (see Figure 5), where the command's name, repeat interval (if required), type (Raw or Script), and data (for Raw commands) is defined. Once the form is saved, the command will appear in the command pallet on the Diagnostics Screen. Script commands have associated LUA scripts, which can be accessed by right clicking the command in the pallet, and selecting "Edit script". When these commands are launched, they execute the code defined within the LUA script (which can either be done by clicking on them in the commands pallet, or by typing their name into the command prompt).

Figure 5: *ECUInjector*'s Communications Components

4.4 Software Walkthrough

The tool is extremely capable when it comes to performing reverse-engineering and vulnerability analysis upon an ECU's diagnostics protocol, with the following steps usually being followed:

4.4.1 Establishing a connection

Before any form of analysis can take place, a connection must be made with the ECU. *ECUInjector* expects a serial connection to the diagnostics bus or port, so that it can exchange information between itself and the diagnostics protocol. Most diagnostic buses run at 12V, and so specialist interfaces are required to implement a serial connection between the machine running *ECUInjector* and the diagnostic bus. USB OBD port adapters are commonly used to perform this task.

With a physical connection made, the software can be configured with the relevant communication details (see section 4.3.2). This includes the COM port (that the physical interface is connected to) and baud rate (which depends on the protocol being utilised on the communication bus). If the bus utilises ISO 9141-2 K-Line, *ECUInjector* can build packets and calculating checksums through a pre-existing protocol implementation, and therefore requires extra details such as the destination ECU's bus ID.

4.4.2 Initialisation procedures

ECUs often require "wake-up" initialisation procedures which inform the ECU that a diagnostic session is beginning and to expect further communications. The exact details of these procedures are specific to each protocol specification and can be implemented in the software either through protocol implementations (as is already the case for ISO 9141-2), or through script commands. Either way, once these commands have been defined, they can be triggered, causing a session to be initiated on the target ECU.

4.4.3 Maintaining the connection

Many ECUs require a continuous connection to maintain a diagnostics session, requiring either a command or dedicated "keep-alive" signal be sent periodically. *ECUInjector* supports sending periodic commands by specifying an interval in the command settings. The "Tester Present" command can be specified, either as a Raw or Script command (if required). Once triggered, the command will be periodically sent by the software until stopped. If a request is made to send another command in the meantime, this command will pause until those packets have been dispatched and will then resume automatically.

4.4.4 Gaining security access

Privileged commands within an ECU are often protected by security algorithms, with many ECUs relying upon "seed-to-key" algorithms. These mechanisms require multiple packets to be sent between the diagnostics system and ECU, with calculations being required at certain points during the transaction. To implement this logic within *ECUInjector* (if the seed-to-key algorithm is known), a dedicated Script command is used that can send the relevant commands and perform the necessary calculations.

4.4.5 Calling basic procedures

Once a session has been established with the ECU and security access has been gained, basic procedures can be called. In many cases, these can be implemented as Script commands that send the relevant requests to ECU and then display the result in the console.

4.4.6 Running advanced operations

More advanced commands, including firmware extraction and memory manipulation (which are often available as services that the ECU provides in a privileged mode) require more complex operations with both the ECU and *ECUInjector* itself. Firmware extraction and modification is one of the most complex operations that can be performed on an ECU, however this can easily be implemented in a Script command within *ECUInjector*. There are two main ways that ECU firmware extraction can take place, either the ECU delivers the entire firmware image in a succession of packets, or requests are made over the diagnostic port for each memory location, with

the ECU responding with its contents in each case. Either way, a basic script can be written to intercept the packets and make up the firmware image. LUA scripts within *ECUInjector* have access to APIs that enable them to write to files, meaning that the script can also be made to prepare a binary file containing the extracted firmware.

4.4.7 Creating reverse-engineering tools

Whilst many different commands can be run with the software, finding out exactly how they are implemented will still require some reverse-engineering. On top of this, it is also often useful to attempt to locate hidden commands that may have been implemented by the manufacturer. To this end, Script commands can also be utilised to write reverse-engineering tools. Below are some examples of scripts that have been written to reverse-engineer specific aspects of an ECU, and should demonstrate the capabilities of the system further:

- **Defined Service Locator:** Standards such as KWP2000 define multiple "services" (operations) that can be called via an ECU's diagnostics interface, however in many cases manufacturers will not have implemented them all. As well as this, if a procedure is not recognised an ECU will often return a specific error code. With this in mind, we developed a script that incrementally sends arbitrary data packets with each of the specified KWP2000 service IDs to the ECU and monitored the response. This enabled us to determine which services had been implemented (and therefore targetable) and had not.
- **Hidden Service Locator:** Automotive communications standards often include room for manufacturers to implement their own operations. Leveraging the logic from the previous script, we built a script that sends arbitrary command data to all service IDs outside the defined scope, and monitored the responses returned. This helped to identify extra services that were implemented by the manufacturer.
- **Hidden Routine Locator:** Diagnostic procedures can also be triggered through many ECU communications protocols, with each one being specified by an ID. We wrote a script that systematically triggered each possible procedure value, allowing a researcher to observe the ECU's I/O lines whilst doing so. This enabled us to review favourable routines to target.
- **Seed-to-Key Brute Force:** Whilst Seed-to-Key algorithms should implement a timeout feature after multiple incorrect attempts (preventing brute force attacks from being possible), it is often not the case. We developed a script to brute force the Seed-to-Key algorithm of a particular ECU and were able to successfully gain entry to privileged functions hidden behind the most privileged security mode.

4.5 Software Development

To ensure the software is portable, cross platform and extensible enough to support the requirements, the software is built in C++. Where possible, the C++ standard library has been utilised to ensure that the code remains portable. This extends to the GUI, with Qt being selected as the most suitable library due to its multi-platform support. Whilst other portable GUI libraries exist, Qt provided a quick way of developing the GUI with minimum barriers to entry, freeing up development time for other core features. Figure 1 shows the technologies used to implement the various components of *ECUInjector.*

5. Applications

ECUInjector has been able to successfully interface with multiple ECUs, with various reverse-engineering scripts being utilised to perform specific operations throughout the reverse-engineering process. Some of these ECUs utilise the KWP2000 standard, which *ECUInjector* is already quite capable of supporting (with helper functions having already been established), meaning that implementation was minimal. Other ECUs utilised different protocols, resulting in a slightly more complex implementation. In both cases, no changes were required to the underlying codebase of the application, and communications and scripts were quickly developed, thus saving on development time. The design of the software has allowed it to be easily adaptable to new ECU protocols, and this has been experienced during these implementations. Table 1 shows the variety of ECUs that have already been communicated with using *ECUInjector*, along with any advanced operations that were performed (in many cases using custom-built reverse-engineering tools within the software (see Section 4.4.7)).

With the specific case of the Pektron SCU, the software aided in detecting a vulnerability in an underlying diagnostic procedure, which lead to the wiping of important information from the ECU and a complete lockout from the system, effectively rendering the ECU, and vehicle, unusable. Use of both our seed-to key brute forcing

script and our hidden routine locating script led to the discovery of the operation that triggers this behaviour. More information can be found in another paper by the authors (Todd *et al.*, 2026).

Table 1: ECUs tested with *ECUInjector*

Vehicle	ECU	Type	Protocol	Advanced Operations
Rover	MEMS3	Engine Management	KWP2000	S2K Algorithm Implemented.
Pektron	SCU	Body Controller	KWP2000	S2K Brute-Forced, Vulnerability found (Todd *et al.*, 2026).
Bosch	ABS 5.3	ABS	KWP1281	N/A
BMW	GM6	Body Controller	BMW DS2	Firmware extracted.
BMW	Video Module	Entertainment	BMW DS2	Firmware extracted, modified and written back.
Rover 75	IPK3	Instrument Cluster	BMW DS2	N/A

6. Conclusions

ECUInjector has proven to be a capable software suite for communications with automotive Electronic Control Units and serves as a suitable way to perform vulnerability analysis. The software's part in discovering a vulnerability in Pektron SCU systems, along with further operations performed on other systems since, has been down to its ability to universally communicate with ECUs and provide ways to write advanced scripts that allow analysis to be performed as required. Finding these vulnerabilities before they are exploited by malicious parties increases the safety of road users, with this specifically being what the software is designed to aid with.

In terms of limitations, a prerequisite of *ECUInjector* is the prior knowledge of the ECU's protocols and internal workings. Furthermore, it is not always possible to circumvent protection mechanisms (this is down to the specific security mechanisms implemented within the ECU), though it is possible when these mechanisms are known or weak. Also, there is a certain level of adaptability to get the system to work with a new ECU (scripts may require changes and are not always directly transferable from one ECU to another). Finally, *ECUInjector* is not designed as a plug-and-play vulnerability detection tool -- work is still required to locate and investigate vulnerabilities. Rather, it aims at facilitating the tasks and reducing manual effort during vulnerability discovery.

ECUInjector provides a means to easily interface with an ECU and perform communications as required. Features should help researchers to author the necessary tools and scripts to perform analysis of protocol implementations and locate vulnerabilities.

Ethics and AI declarations: Ethical clearance was not required for this research. AI tools were not used in the writing of this paper.

References

Bim-Tun (2020) *BimmerEditor Software*. Available at: https://www.bmweditor.com/ (Accessed: February 17, 2026).

Dijk, L. van (2017) *Future Vehicle Networks and ECUs - Architecture and Technology considerations*.

dSpace (2026) "ControlDesk." Available at: https://www.dspace.com/en/ltd/home/products/sw/experimentandvisualization/controldesk.cfm (Accessed: February 17, 2026).

Elias Kotlyar (2026) "EliasTuning/KWP2000-CAN: KWP2000 Library for python." Available at: https://github.com/EliasTuning/KWP2000-CAN (Accessed: February 17, 2026).

ETAS (2026) "INCA." Available at: https://www.etas.com/ww/en/products-services/data-acquisition-processing-tools/software-products/inca-software-products/ (Accessed: February 17, 2026).

European Union (1998) *Directive - 98/69 - EN - EUR-Lex*. Available at: https://eur-lex.europa.eu/eli/dir/1998/69/oj/eng (Accessed: February 17, 2026).

European Union (2002) *Directive - 2002/51 - EN - EUR-Lex*. Available at: https://eur-lex.europa.eu/legal-content/EN/ALL/?uri=CELEX%3A32002L0051 (Accessed: February 17, 2026).

Gardetto, E., Bagian, T. and Lindner, J. (2005) "High-mileage study of on-board diagnostic emissions," *Journal of the Air & Waste Management Association (1995)*, 55(10), pp. 1480–1486. Available at: https://doi.org/10.1080/10473289.2005.10464745.

Van den Herrewegen, J. and Garcia, F.D. (2018) "Beneath the bonnet: A breakdown of diagnostic security," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11098 LNCS, pp. 305–324. Available at: https://doi.org/10.1007/978-3-319-99073-6_15.

ISO (1993) *ISO 11898:1993 - Road vehicles — Interchange of digital information — Controller area network (CAN) for high-speed communication*. Available at: https://www.iso.org/standard/20380.html (Accessed: February 17, 2026).
ISO (1994) *ISO 9141-2:1994(en), Road vehicles — Diagnostic systems*. Available at: https://www.iso.org/obp/ui/en/#iso:std:iso:9141:-2:ed-1:v1:en (Accessed: February 18, 2026).
ISO (1998) *ISO 14229:1998 - Road vehicles — Diagnostic systems — Diagnostic services specification*. Available at: https://www.iso.org/standard/23918.html (Accessed: February 17, 2026).
ISO (2000) *ISO 14230-4:2000 - Road vehicles — Diagnostic systems — Keyword Protocol 2000*. Available at: https://www.iso.org/standard/28826.html (Accessed: February 17, 2026).
ISO (2004) *ISO 15765-2:2004 - Road vehicles — Diagnostics on Controller Area Networks (CAN)*. Available at: https://www.iso.org/standard/33616.html (Accessed: February 17, 2026).
ISO (2016) *ISO 17987-2:2016 - Road vehicles — Local Interconnect Network (LIN)*. Available at: https://www.iso.org/standard/61223.html (Accessed: February 17, 2026).
ISO (2017) *ISO 17458-2:2013 - Road vehicles — FlexRay communications system*. Available at: https://www.iso.org/standard/59806.html (Accessed: February 17, 2026).
Lauser, T. and Krauß, C. (2023) "Formal Security Analysis of Vehicle Diagnostic Protocols," *ACM International Conference Proceeding Series* [Preprint]. Available at: https://doi.org/10.1145/3600160.3600184.
Lauser, T., Munoz Molto, G. and Krauß, C. (2024) "(Un)authenticated Diagnostic Services: A Practical Evaluation of Vulnerabilities in the UDS Authentication Service," *CSCS 2024 - Proceedings of the 2024 Cyber Security in CarS Workshop, Co-Located with: CCS 2024*, pp. 50–60. Available at: https://doi.org/10.1145/3689936.3694695.
Onuma, Y., Terashima, Y. and Kiyohara, R. (2017) "ECU software updating in future vehicle networks," *Proceedings - 31st IEEE International Conference on Advanced Information Networking and Applications Workshops, WAINA 2017*, pp. 35–40. Available at: https://doi.org/10.1109/WAINA.2017.45.
Pełechaty, P. and Konieczny, Ł. (2024) "Analysis of security vulnerabilities in vehicle on-board diagnostic systems," *Diagnostyka*, Vol. 25, No. 3(3). Available at: https://doi.org/10.29354/diag/192162.
Pier-Yves Lessard (2026) "python-udsoncan." Available at: https://github.com/pylessard/python-udsoncan (Accessed: February 17, 2026).
Prabhakaran, A., Thirumoorthi, P. and Sri Dhivya Krishnan, K. (2024) "Design and development of an intelligent zone based master electronic control unit for power optimization in electric vehicles," *Scientific Reports*, 14(1), p. 20142. Available at: https://doi.org/10.1038/s41598-024-70580-7.
Revill, A. (2026) *MEMS Tools*. Available at: https://andrewrevill.co.uk/MEMSToolsIndex.htm (Accessed: February 17, 2026).
Robert Bosch GmbH (2007) *Bosch Professional Automotive Information*. 5th ed. Available at: https://doi.org/10.1007/978-3-658-01784-2.
SAE (2015) *J1962 Diagnostic Connector*. 400 Commonwealth Drive, Warrendale, PA, United States: SAE International. Available at: https://doi.org/10.4271/J1962_201509.
Shin, Y. *et al.* (2011) "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Transactions on Software Engineering*, 37(6), pp. 772–787. Available at: https://doi.org/10.1109/TSE.2010.81.
Todd, J. *et al.* (2026) "Blank Pages Tell All: Reverse Engineering NEC 78K/0 Automotive Firmware," in *The 41st ACM/SIGAPP Symposium on Applied Computing (SAC '26)*. Thessaloniki, Greece: ACM. Available at: https://doi.org/10.1145/3748522.3779871.
US Environmental Protection Agency (1990) *Clean Air Act Title I - Air Pollution Prevention and Control, Parts A through D | US EPA*. Available at: https://www.epa.gov/clean-air-act-overview/clean-air-act-title-i-air-pollution-prevention-and-control-parts-through-d (Accessed: February 17, 2026).
Vector (2026) "CANoe." Available at: https://www.vector.com/gb/en/products/products-a-z/software/canoediva/# (Accessed: February 17, 2026).