Uptane: Securing Software Updates for Automobiles[*]

Trishank Karthik Kuppusamy
NYU Tandon School of
Engineering
trishank@nyu.edu

Akan Brown
NYU Tandon School of
Engineering
akan.brown@nyu.edu

Sebastien Awwad
NYU Tandon School of
Engineering
sebastien.awwad@nyu.edu

Damon McCoy
NYU Tandon School of
Engineering
mccoy@nyu.edu

Russ Bielawski
University of Michigan
jbielaws@umich.edu

Cameron Mott
Southwest Research Institute
cameron.mott@swri.org

Sam Lauzon
University of Michigan
slauzon@umich.edu

André Weimerskirch[†]
University of Michigan,
Lear Corporation
aweimerskirch@lear.com

Justin Cappos
NYU Tandon School of
Engineering
jcappos@nyu.edu

ABSTRACT

Software update systems for automobiles can deliver significant benefits, but, if not implemented carefully, they could potentially incur serious security vulnerabilities. Previous solutions for securing software updates consider standard attacks and deploy widely understood security mechanisms, such as digital signatures for the software updates, and hardware security modules (HSM) to sign software updates. However, no existing solution considers more advanced security objectives, such as resilience against a repository compromise, or freeze attacks to the vehicle's update mechanism, or a compromise at a supplier's site. Solutions developed for the PC world do not generalize to automobiles for two reasons: first, they do not solve problems that are unique to the automotive industry (e.g., that there are many different types of computers to be updated on a vehicle), and second, they do not address security attacks that can cause a vehicle to fail (e.g. a man-in-the-middle attack without compromising any signing key) or that can cause a vehicle to become unsafe. In this paper, we present Uptane, the first software update framework for automobiles that counters a comprehensive array of security attacks, and is resilient to partial compromises. Uptane adds strategic features to the state-of-the-art software update framework, TUF, in order to address automotive-specific vulnerabilities and limitations. Uptane is flexible and easy to adopt, and its design details were developed together with the main automotive industry stakeholders in the USA.

1 Introduction

The vulnerabilities of software update systems pose a major security risk to modern computers [6, 10, 23]. Software updaters are often inadequately protected, leading to substantial damage. For example, in 2014, South Korean banks and media companies were attacked causing 756 million dollars in damages [7]. On the flip side of the coin, software repositories that host software updates have also been compromised. Software repositories run by major organizations such as Adobe, Apache, Debian, Fedora, FreeBSD, Gentoo, GitHub, GNU Savannah, Linux, Microsoft, npm, Opera, PHP, RedHat, RubyGems, SourceForge, and WordPress repositories have all been compromised at least once [3–5,13–19,21,22,24,

26,27,29–31,33–37,39]. In one such case, a compromised Source-Forge mirror distributed a malicious version of phpMyAdmin, a popular database administration software [33]. The modified version allowed attackers to gain system access and remotely execute PHP code on servers that installed the software.

Automobiles introduced software updates more than a decade ago, and today many electronic components in a vehicle can be updated by an automotive technician with a proper tool. However, available update mechanisms played a crucial role in a variety of published automotive related hacks, such as the recent study by Miller & Valasek [28] as well as Thuen's study of an insurance dongle [38]. It is easy to recognize that software updates over-the-air (OTA) for vehicles will become a standard operation, since it enables new business models, and the seamless addition of new features, as well as fixing safety and security flaws. The cost of a vulnerable update mechanism is endlessly higher than in the PC world since a vulnerability can potentially impact the vehicle's performance.

Even when skilled developers attempt to address flaws in software update systems, it is common for them to make fatal security errors [10]. As a result, an attacker can cause a huge amount of damage, as seen from the examples cited earlier. While the first compromise-resilient solutions [25] are currently being deployed in practice, these existing solutions do not apply well to automobiles because of differences in the operational model (e.g. that there are many different types of computers to be updated on a vehicle). Previous solutions fail to address security attacks that can cause a vehicle to fail even if attackers can perform only man-in-the-middle-attacks, and have not compromised any signing key.

This work presents the design of Uptane, a novel and practical software update framework for automobiles. To the best of our knowledge, Uptane is the first software update framework for automobiles that addresses a comprehensive and broad threat model (Section 5). Our work enhances the security of previous update systems by adding and validating new types of signed metadata to improve resilience to attacks. Since different automobile manufacturers and tier-1 suppliers have their own development and deployment infrastructure, Uptane does not prescribe a rigid, one-size-fits-all solution. Instead, we provide a flexible framework that enables different parties to configure the provided security benefits to their needs and environment.

[†]Work done while at the University of Michigan.

To build Uptane, we have devised new design features that retain strong security guarantees even in the face of compromises of parts of the system. Uptane includes new features beyond the standard mechanisms to secure vehicular software updates with little-to-no detriment to usability for automobile manufacturer. These features are: using *additional storage* to recover from attacks where the software on an ECU has been overwritten with incorrect data; *broadcasting metadata* to prevent attacks where different ECUs are shown different versions of metadata at the same time; using a *vehicle version manifest*, or information signed by every ECU about what it has installed, to detect attacks where ECUs have installed versions of software that may not work together; and using a *time server* to limit attacks where ECUs are indefinitely held back from the latest updates.

Our contributions are as follows:

- We present a comprehensive and broad threat model for software updates on automobiles (Section 3).

- We discuss previous solutions for building compromise-resilient software update systems, with a focus on why they do not adequately address the threat model for automobiles (Section 4).

- We present Uptane, a practical software update framework for automobiles that adds new design features to solve problems associated with previous solutions.

Maybe most important, we regularly present our design to the major automotive industry stakeholders in the USA to ensure that the design is reasonable and ready for deployment. More information about our Uptane industry effort can be found at [1, 2].

2 Background

Uptane adapts proven strategies used in securing updates for software repositories to meet the unique specifications of computing devices on vehicles. In order to understand these adapted strategies, it is important to understand why updating software on automobiles is a special challenge, and why the proposed strategies have proven so effective in other applications.

2.1 Automobiles

A modern vehicle consists of mechanical parts (e.g., engine and brakes) that are controlled via software on microcomputers called *Electronic Control Units* (ECUs). These units are responsible for executing specific functions, from tightening a seat belt during an accident to adjusting a passenger side mirror, and are distributed throughout the vehicle. A number of ECUs, such as the telematics unit, also have the ability to transmit and receive information from the outside world. Different network segments can be connected through gateway ECUs, such as the body control module (which may connect, for example, lights to switches).

ECUs may reside on different network segments, or *buses*, within the same vehicle. Figure 1 illustrates a hypothetical example where the telematics unit, engine, and diagnostics ECUs reside on the high-speed (and more expensive) Controller Area Network (CAN) bus, and the side windows, door locks, and power windows ECUs reside on the low-speed (and less expensive) Local Interconnect (LIN) bus. Other bus types include MOST, FlexRay, Automotive Ethernet, and CAN-FD. The LIN bus has a single master, up to 16 slaves, and allows speeds up to 20Kbps. The CAN bus has no master (any ECU can transmit at any time, although messages are prioritized), accommodates up to 2048 ECUs (with 11-bit CAN IDs, although an extension allows for 29-bit CAN IDs), and allows speeds up to 1Mbps.

An *original equipment manufacturer* (OEM), such as Ford or General Motors, chooses the ECUs that reside on a vehicle model,
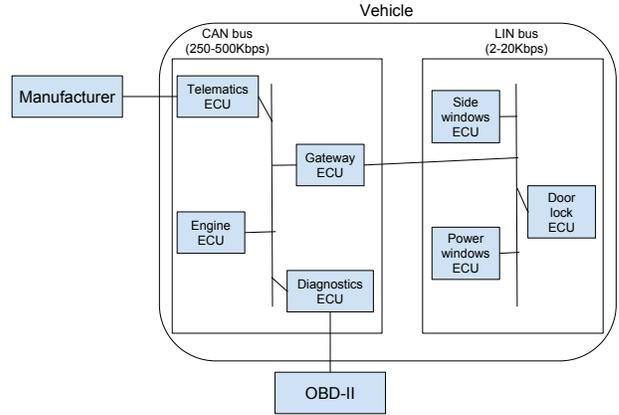

Figure 1: How ECUs reside on different network segments within a vehicle.
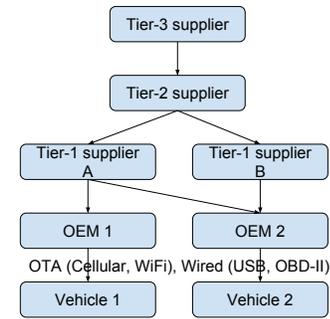

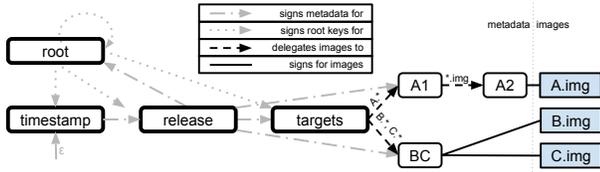**Figure 2:** How software is distributed from suppliers to OEMs to vehicles.

usually produced by automotive *suppliers*. Tier-1 suppliers, which directly supply OEMs, may outsource the development of sub-components of ECUs (e.g., the baseband module) to tier-2 suppliers, which in turn may then outsource to tier-3 suppliers. The software for an ECU is maintained by a supplier, and delivered to the OEM to be distributed to vehicles. Thus, the source code for software on ECUs may not be available to OEMs. Figure 2 illustrates how software is distributed in this supply chain.

To facilitate software updates produced both by the OEM and suppliers, the OEM maintains a software repository that hosts software update files. Dealerships download the latest ECU software from this repository to then flash it to ECUs via the OBD2 port, an OEM may push software updates to all applicable vehicles, or a vehicle may pull the latest known software updates. Some gateway ECUs, or telematics and infotainment ECUs, have a wireless connection to the Internet. Updates are disseminated to vehicles via this wireless connection, or through physical distribution channels, such as dealerships, customer-inserted USB sticks, or OBD-II ports, which are usually reserved for diagnostics.

We assume that all software on the repository is organized by *images*, where an image contains all of files needed to run an application on an ECU. There is exactly one image per ECU.

2.2 The Update Framework (TUF)

The Update Framework (TUF) [25, 32] is a security system designed to protect users of software repositories, such as Microsoft Windows Update, Ubuntu, the Python Package Index (PyPI), Ruby-Gems, or Docker Hub, from a number of security attacks [9–12]. The technology integrates a layer of signed *metadata*, such as cryp-

Figure 3: The four basic (root, timestamp, release, and targets) roles used in TUF.

Role	Purpose
root	Serves as the certificate authority for the repository. Distributes and revokes the public keys used to verify the root, timestamp, release, and targets role metadata.
timestamp	Indicates whether there is any new metadata or image on the repository.
release	Indicates which images have been released at the same time by the repository.
targets	Indicates metadata such as the cryptographic hashes and file sizes of images. May delegate this responsibility to other, custom-made roles.

Figure 4: The four basic roles that TUF uses to add signed metadata to the repository.

tographic hashes and file sizes, to the software repository. By verifying these metadata, e.g., by checking the hash of a downloaded software update against the signed hash of the same update in the metadata, security attacks can be detected and prevented before installation. TUF does not aim to prevent a compromise before it happens; rather, it aims to limit the impact of a compromise when it happens.

TUF is designed around four key principles. The first is separation of duties. Different pieces of metadata are signed by the repository *administrators* using different *roles* in order to distribute responsibilities and increase compromise resilience [25]. Figure 3 illustrates the four basic roles that must exist on a TUF-secured repository: the root, timestamp, release, and targets roles. The root role serves as the certificate authority: it distributes and revokes the *public keys* used to verify metadata produced by each of these four roles (including itself). The timestamp role indicates whether there are any new metadata or images on the repository. The release role indicates which images have been released by the repository at the same time. The targets role provides metadata, such as hashes and file sizes of images, and may *delegate* the responsibility of signing metadata about images to other, custom-made roles. For example, in Figure 3, the targets role has delegated all images that match the filename pattern "A.*" to the A1 role, and all images that match the filename patterns "B.*" and "C.*" to the BC role. In turn, the A1 role delegates a subset of its images (in this case, only the "A.pkg") to the A2 role. A delegation binds the public keys used by a delegatee to a subset of the images these keys are trusted to sign. This means that the targets role would distribute and revoke the public keys for the A1 and BC roles, whereas the A1 role would do the same for the A2 role.

The second design principle is the optional use of a threshold number of signatures. The metadata file for a role can be configured so it must be signed using a minimum number of t out of n keys. This significantly increases the number of keys that must be compromised to launch an attack.

The third principle is an explicit or implicit process to revoke keys. The public keys used to verify a metadata file can be implicitly revoked by including an expiration date in the metadata file (so that the keys would be considered expired after that date), or they can be explicitly revoked by signing new metadata that replaces the old keys with new keys.

The fourth principle is using offline keys, or private keys kept physically disconnected from the Internet, to sign the most sensitive roles. This is done so that, even if attackers compromise the repository, they are unable to sign new and malicious versions of sensitive metadata.

By combining these four design principles, the impact of a key compromise can be distributed and minimized such that users are insulated from many security attacks — even if the repository is compromised.

Figure 4 summarizes the responsibilities of each of the four basic roles. An ECU would install images with a software updater that downloads and verifies all the metadata signed by these roles. The verification process starts with the software updater downloading and verifying all of these metadata from the repository. An image should be installed only if it matches the signed metadata.

TUF is a flexible security system that can be adapted to manage even complex arrangements concerning who is trusted to sign images. With this adaptability, TUF can be used to sign software on automotive repositories. However, it will require a few modifications in order to meet the special challenges of securing updates on ECUs.

3 Threat model

In this section, we define a threat model, or a look at many ways attackers can affect the security of software updates delivered between repositories and vehicles. First, we describe the higher-level goals that motivate attackers to target automotive ECUs. (Section 3.1). Second, we describe capabilities that attackers need to acquire to achieve these goals (Section 3.2). Third, given these capabilities, we describe several security attacks that attackers can perform to achieve their goals (Section 3.3). Finally, we state the security goals we believe are achievable with Uptane (Section 3.4), and a few that are out of scope at this stage (Section 3.5).

3.1 Attacker goals

We surmise that attackers target vehicle ECUs to achieve one or more of the following four goals, listed in increasing order of impact:

- Read updates: Attackers aim to learn the contents of software updates in order to reverse-engineer ECU firmware and, in doing so, to steal intellectual property from the vehicle.

- Deny updates: Attackers want to prevent vehicles from fixing software problems.

- Deny functionality: Attackers try to stop ECUs from functioning correctly, thus causing the vehicle or a component to fail or behave abnormally, either temporarily or permanently.

- Control: Attackers want to modify vehicle performance.

3.2 Attacker capabilities

To meet the broad goals stated above, an attacker is capable of the following actions:

- Intercept and change network communications (i.e., perform man-in-the-middle attacks). These actions can be accomplished from either:

  - Outside the vehicle: for example, the attacker could control a cellular network used to distribute updates, or

  - Inside the vehicle: for example, the attacker could control communications over a gateway ECU, normal ECU, OBD-II port, USB, etc. Using this, attackers could spoof messages as having originated from any source.

Icon	Security attack
	Eavesdrop attack
	Drop-request attack
	Freeze attack
	Partial bundle installation attack
	Rollback attack
	Endless data attack
	Mixed-bundles attack
	Mix-and-match attack
	Arbitrary software attack

Figure 5: The security attacks that ECUs should handle.

- Compromise ECUs in a vehicle.

- Compromise cryptographic keys used to sign software updates or the servers that store those keys.

Furthermore, we assume that attackers have access to software updates previously released by the repository. They may do this by pretending to be a vehicle, and periodically requesting new updates from the repository.

3.3 Security attacks

To achieve the goals listed above, attackers can use the following security attacks (listed in Figure 5).

3.3.1 Read updates

First, attackers are interested in the contents of software updates, perhaps in order to reverse-engineer ECU firmware. In doing so, they can steal intellectual property from the vehicle. One way to read this data is to stage an *eavesdrop attack*, where attackers can read unencrypted updates sent from the repository to the vehicles.

3.3.2 Deny updates

Second, attackers want to prevent vehicles from fixing software problems by denying access to updates. They may do so with one of the following attacks:

- *Drop-request attack*: blocks network traffic outside or inside the vehicle to prevent an ECU from receiving any updates.

- *Slow retrieval attack*: causes an ECU to receive an application update so slowly that a known security vulnerability can be exploited by an attacker in the meantime. Note, that both the strawman for Uptane and its final design are capable of handling this type of attack for all ECUs.

- *Freeze attack*: indefinitely sends an ECU the last known update, even if there may be newer updates on the repository.

- *Partial bundle installation attack*: causes some ECUs to not install the latest updates. Attackers can do this by dropping traffic to these ECUs. Sometimes this attack may happen accidentally, such as if updates are interrupted due to running out of power.

3.3.3 Deny functionality

Third, attackers would like to cause vehicles to fail to function in one of the following ways:

- *Rollback attack*: causes an ECU to install outdated software with known vulnerabilities.

- *Endless data attack*: The simplest way for attackers to induce failure is to execute an *endless data attack*. This strategy causes an ECU to crash by sending it an indefinite amount of data, thus making it run out of storage.

- *Mixed-bundles attack*: A *mixed-bundles attack* causes failure of ECUs to interoperate, thus preventing the vehicle from working correctly. Attackers do this by causing ECUs to install incompatible versions of software updates that must not be installed at the same time. Specifically, attackers cannot create new bundles of images, but they can show different bundles to different ECUs at the same time. For example, ECU-1 is given an update from bundle 1, whereas ECU-2 is given an update from bundle 2.

- *Mix-and-match attack*: A *mix-and-match attack* also causes ECUs to fail to interoperate. If attackers have compromised repository keys, they can use these keys to release an arbitrary combination of new versions of images. For example, both ECU-1 and ECU-2 install updates from bundle 3. However, attackers have abused repository keys to sign this bundle, which points to incompatible versions of software updates. Note that a mix-and-match attack is worse than a partial bundle installation or mixed-bundles attack, because attackers can arbitrarily combine updates.

3.3.4 Control

Fourth, and most severe of all, attackers can cause an ECU to install software of the attacker's choosing. This means that the attacker can arbitrarily modify the vehicle's performance. They may do so with an *arbitrary software attack*, where attackers overwrite the software on an ECU with malicious software.

3.4 Defender goals

Inasmuch as possible, we aim to achieve compromise resilience [25] to all attacks. Ideally this should also occur when some portions of the OEM have been compromised. We aim to set the bar for such a compromise high enough that an attack would be unlikely to affect all vehicles, and/or an attacker would have to perform an unfeasible amount of work to do so.

A defender also wants to be able to recover from attacks in the most secure way possible. So a compromised key should be able to be securely and swiftly revoked, and vehicles not yet compromised should not be at risk. Ideally, a compromised ECU should be able to be restored to correct operation with minimal negative consequences.

3.5 Non-goals

The following types of attacks are considered outside the scope of this paper:

- Physical attacks, such as mechanics manually tampering with ECUs after pulling them out of vehicles.

- Compromise of the build system, version control system, packaging process, etc. of a supplier or an OEM. Since other work exists to solve this problem (e.g., git signing [20], TPMs, Toto), we do not attempt to duplicate those techniques here.

- Using *remote exploits* to compromise an ECU using a programming error (e.g., buffer or heap overflow, ROP, use-after-free, etc.).

- Random failures. Since the automotive industry has handled these failures well, we focus only on attacks by an intelligent adversary in this paper.
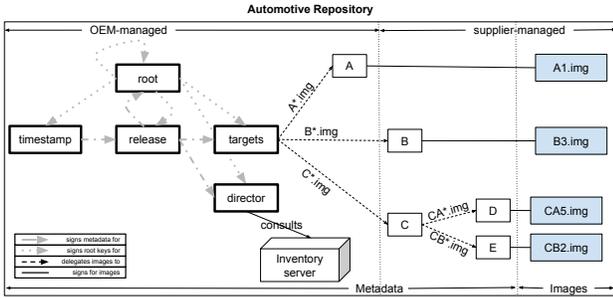
Figure 6: TUF roles customized and configured for automotive repositories. Note the addition of the director role, and the delegation of ECU images from the OEM to its suppliers.

Figure 7: The relationship between ECUs on a vehicle. Details are explained in the text.
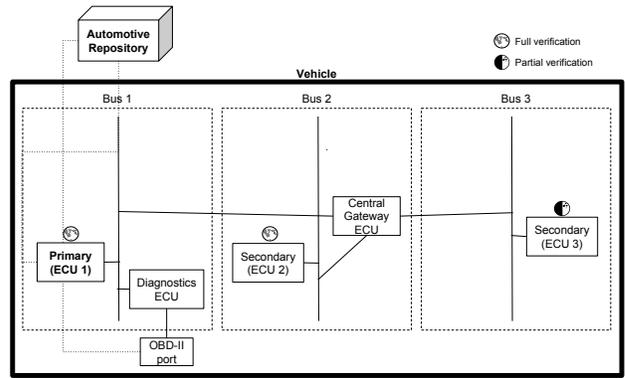
4 Strawman design

In this section, we present a strawman design utilizing a straightforward application of TUF, and reveal why it would not adequately address our threat model (Section 3). First, we discuss how an OEM could customize TUF for vehicle updates by adding new roles (Section 4.1). Second, we explain how an ECU verifies metadata in this design (Section 4.2). Third, we describe the workflow for downloading and verifying software updates (Section 4.3) on a vehicle. Fourth, we discuss how an OEM may control the director's ability to rollback updates (Section 4.4). Finally, we show the limitations of this strawman design (Section 4.5), setting the stage for the new design features for Uptane presented in Section 5.

4.1 Customizing TUF for automotive repositories

Though designed for software on traditional repositories, TUF roles can be used to sign software on automotive repositories as well. However, to do so, we need to add a new role and use the pre-existing targets role to delegate images.

The first change required is the addition of the *director* role to the repository (see Figure 6). Adding this role gives the OEM more complete control over the choice of images downloaded and installed by vehicles. The director role functions much like a traditional package manager, except that it operates on the server side instead of the client. It can be used to instantly blacklist faulty versions of software, or to customize software for different vehicles of the same type when vehicle owners may have paid more for extra features. When a vehicle contacts the repository for updates, it must identify itself to the director using its *vehicle identification number* (VIN). Given the VIN, the director would consult the inventory server to learn which ECUs are installed on this vehicle. Then, the director signs *director metadata*, or fresh instructions for the vehicle about which images should be downloaded and installed by its ECUs. These instructions bind the unique serial number of every ECU to the filename and hash of the image it must install. Since these instructions are always customized for every vehicle, and freshly signed by the director role, a man-in-the-middle attacker cannot copy these instructions and replay them to other vehicles. The director also performs dependency resolution [8] on behalf of vehicles. This means that if one ECU image depends on another, then the director would include both of these images in its instructions.

Second, the OEM may use the pre-existing targets role to delegate the signing of these images to its suppliers. These delegations are flexible, and allow either the OEM or its suppliers to sign images, in keeping with the decentralized structure of the auto supply chain. We use four examples, illustrated in Figure 6, to show how flexible these delegations can be. In the first example, the tar-
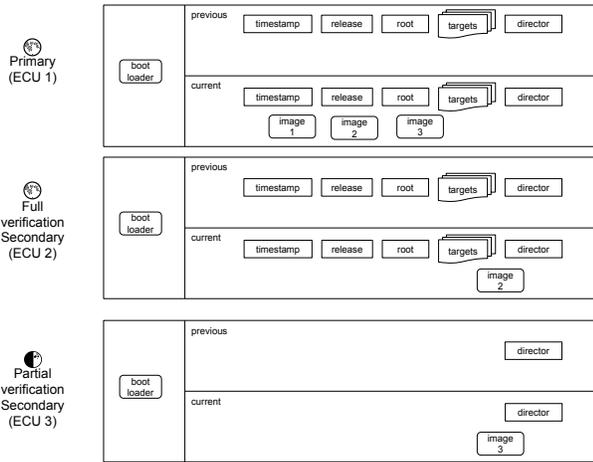
gets role delegates to role A all images produced by supplier A. Although supplier A builds these images and uploads them to the repository, the supplier has opted not to sign these images. Instead, the OEM itself controls role A, and signs these images on behalf of the supplier. In the second example, the targets role delegates to role B all images produced by a supplier B. In this case, the supplier has opted to sign the images produced by supplier. In the third example, the targets role delegates to role C all images produced by a supplier C. Instead of signing images, this role delegates one subset of images to the role D, and another subset to role E. These roles could correspond to software developers who are currently employed by supplier C, and who are responsible for developing, building, and testing ECU images. At any time, the role C can change the developers responsible for signing images, without requiring the OEM to update its delegations.

Even with the changes described above, it is important to remember that TUF was designed with the assumption its users would be working with powerful computing devices, such as smart-phones, tablets, laptops, desktops, or servers. For these types of devices, downloading and verifying all of this TUF metadata is not a challenge. However, a vehicle consists of many different computers (ECUs), each of which may verify metadata to a different extent from the others. Securing this diverse group of computing units means conducting metadata verification in different ways.

4.2 Verifying TUF metadata on vehicles

To design an effective system for verifying metadata on car ECUs, one first needs to understand that ECUs are highly heterogeneous in terms of computing power, memory storage, and security capabilities. We classify every ECU on a vehicle either as *primary* or as *secondary*. For the sake of simplicity, we assume that there is only one primary ECU per vehicle. The primary downloads, verifies, and distributes metadata, as well as images, to all secondaries. The primary verifies all metadata on behalf of all secondaries in order to protect them from security attacks. A secondary depends upon the primary to be given its metadata and image, but verifies them before installing the image. We assume that the primary can communicate with all secondaries (possibly indirectly through a central gateway as in Figure 7). As illustrated in Figure 8, every ECU verifies an image downloaded from the repository in one of the following two ways.

In the first, more secure method, a *full verification* ECU (e.g., the primary, and ECU 2) verifies and caches the timestamp, release, root, and targets metadata. It obtains these files from the primary, which has already verified them. Despite this, the ECU checks the files again, to protect against a compromised primary or a man-

Figure 8: The metadata and images verified and cached by each type of ECU.

Figure 9: Security attacks that affect the primary, organized by attacker capabilities. DR, TS, RS, SP, TR, and RT denote the director, timestamp, release, supplier, targets, and root keys, respectively. Red keys are easier to compromise than blue keys. Thin-bordered keys are easier to compromise than thick-bordered keys. The * symbol denotes that an attack is limited to ECUs signed by the given roles.

in-the-middle on the car's network. To check the metadata, these ECUs need storage space to keep a previous and current copy. In order to prevent a variety of security attacks, the ECU verifies all of these metadata following the TUF specification. It also verifies that the hash of a downloaded image matches the hash of the image in both the director and targets metadata. Thus, an attacker must compromise both roles to cause this ECU to install a malicious image.

In the second, less secure method, a *partial verification* ECU (e.g., ECU 3) verifies and caches only the director metadata. Once again, this information is provided by the primary. The ECU that performs partial verification requires enough storage space to keep a previous and current copy of the director metadata. During verification, the ECU verifies that the hash of a downloaded image matches the hash of the image in the signed and verified director metadata.

Every ECU performs its software updates using a *bootloader*. The bootloader is a small program that runs every time an ECU is rebooted to enable the process of downloading the latest metadata and image from the primary, and verifying the image before transferring execution to the image. The OEM supplies the bootloader, as well as the latest metadata and images, onto ECUs when the vehicle is initially manufactured.

4.3 Software update workflow

In this section we put together all the elements described above to show how the proposed strawman design would work to securely update software on a vehicle.

As a vehicle is running, the primary ECU downloads the latest relevant software updates. First, the primary downloads and verifies the latest timestamp, release, root, and targets metadata. Second, the primary downloads and verifies the director metadata, which contains information (e.g., filenames and hashes) about images that all ECUs should install next. Third, based on that information, the primary downloads and verifies images for all ECUs. It verifies that the targets and director metadata match each other, and that all images match these metadata. Fourth, the primary sends the appropriate metadata to every secondary. At this point, the vehicle is ready to apply the ECU updates at the next convenient opportunity (e.g., upon the next restart).

After the vehicle is restarted, we assume that all ECUs reboot, run their bootloader, and update to the latest software. First, the pri-

mary sends the latest downloaded image to every secondary, which overwrites its previous image. (In order to prevent slow retrieval attacks, we assume that every ECU uses a cycle counter.) Second, every bootloader verifies its latest downloaded metadata. Third, every bootloader verifies that the latest downloaded image matches the latest downloaded metadata. Then, each bootloader overwrites its previous with the latest downloaded metadata. Finally, each bootloader transfers execution to the latest downloaded image.

4.4 Controlling the director's ability to rollback updates

Using the director role to control updates in an online manner comes with benefits and drawbacks. A primary benefit is that the OEM does not need to use offline keys (which are cumbersome in actual practice) in order to immediately blacklist images that are listed in the targets metadata. However, a drawback is that this opens the ECU to rollback attacks when the director is compromised, and obsolete versions of images are still listed in the targets metadata. Unless noted otherwise, we assume that the director cannot instruct an ECU to replace its current image on disk with an older version number. The OEM may do this by setting a default flag in a configuration file. We will discuss in a separate document how an OEM may override this behavior in deployment.

4.5 Security analysis

The strawman design falls short in preventing many attacks on ECUs, even with its enhanced security features. In particular, an attacker can cause a vehicle to fail even without compromising any keys. The figures in this subsection enumerate security attacks to which ECUs with different types of metadata verification are vulnerable, given various attacker capabilities, such as having compromised a set of keys used to sign updates. In each figure, we assume that attackers (1) have compromised some set of keys used to sign metadata, and (2) can respond to requests either outside or inside the vehicle. Only fixed combinations of keys are shown in these figures, because any other combination would reduce to one of the documented cases.

Figure 9 lists the security attacks to which the primary is vulnerable. All rows, except for the last, are listed in increasing order of difficulty for the attacker to compromise.

Even if attackers have not compromised any key (or if they have compromised only the director keys), the primary is vulnerable to an eavesdrop attack if the network connection between the primary and the repository is not encrypted (e.g., using TLS). The primary is also vulnerable to a drop-request attack, because attackers can drop requests to the repository. Furthermore, the primary is vulnerable to a freeze attack because, unlike desktop and server machines, ECUs do not typically have reliable real-time clocks. Thus, the pri-

Attacker capabilities	Type of secondary		
	Full verification		Partial verification
MitM			
MitM + DR			
MitM + TS RS DR			
MitM + TS RS DR SP			
MitM + TS RS DR TR			
MitM + RT			

Figure 10: Security attacks that affect secondaries if attackers have *not* compromised the primary. The + symbol denotes that an attack requires attackers to be able to respond to requests inside the vehicle.

Attacker capabilities	Type of secondary		
	Full verification		Partial verification
MitM			
MitM + DR			
MitM + TS RS DR			
MitM + TS RS DR SP			
MitM + TS RS DR TR			
MitM + RT			

Figure 11: Security attacks that affect secondaries if attackers *have* compromised the primary.

mary may not be able to tell whether a metadata file has expired. Finally, the primary is vulnerable to a partial bundle installation attack, because man-in-the-middle attackers inside the vehicle can drop updates to some (but not all) ECUs.

If attackers have compromised at least the director keys, then the primary is also vulnerable to a mix-and-match attack, because attackers can use the director to arbitrarily combine new versions of images, and thus cause the vehicle to install incompatible versions of images. (If attackers have also compromised the timestamp and release keys, then they can arbitrarily combine new versions of targets metadata files.)

If attackers have compromised a supplier's keys (as well as the timestamp, release, and director keys), then the primary is also vulnerable to rollback and arbitrary software attacks if the supplier signs its images (denoted by the asterisk symbol).

If attackers have compromised the root keys, then the primary is vulnerable to indefinitely long freeze attacks, because attackers can renew signatures on any metadata file or postpone its expiration.

Figure 10 documents the security attacks to which secondaries are vulnerable if attackers have *not* compromised the primary. There are two differences from the previous figure.

First, all secondaries are always vulnerable to endless data attacks, because attackers who can respond to requests within the vehicle can tamper with images distributed between the primary and secondaries. Since the primary overwrites the secondary's image, all secondaries are vulnerable.

Second, all secondaries are also always vulnerable to mixed-bundle attacks attacks because attackers who can respond to requests inside the vehicle (denoted by the + symbol) can show different versions of metadata to different secondaries at the same time.

Figure 11 documents the security attacks to which secondaries are vulnerable if attackers *have* compromised the primary. The most notable difference from the previous table is that even if attackers have compromised only the director keys, then all partial

verification secondaries are vulnerable to rollback and arbitrary software attacks. This is because these secondaries depend upon the primary (now compromised, unbeknownst to them) to prevent these attacks by comparing the director to the targets metadata.

To conclude this section, we observe that the strawman design, which represents the straightforward application of existing technology, will not prevent vulnerabilities even if the attacker has not compromised any keys. All an attacker has to do is tamper with requests inside the vehicle, which is significantly easier than compromising keys. Furthermore, there is no secure way to recover from many types of successful attacks. In the next section, we present several design features that Uptane offers OEMs to significantly improve software update security beyond the strawman design.

5 Uptane: Design

Uptane is a new software update framework designed specifically to meet the security needs of programs running on automotive ECUs. Though its core processes are adapted from TUF, Uptane adds several distinct features that reflect the demands of the automotive environment. We begin this section by describing the design goals (Section 5.1) and overview (Section 5.2) of Uptane. Then, we describe the four design features (Section 5.3) unique to Uptane that target the security weaknesses identified in the strawman design. An OEM is free to choose and apply some and not all of these features, depending on its security concerns. However, as we will analyze in Section 5.4, we strongly recommend using all of these features in order to provide as much compromise resilience [25] as possible.

5.1 Design goals

Our primary goal in developing Uptane was to create a software update mechanism for vehicles capable of retaining the strongest level of security possible (e.g., compromise resilience [25]), even if said vehicle should be attacked by an intelligent and determined adversary. To the extent that is practical, an in-depth defense should be employed to require an attacker to compromise many different systems (which are protected in diverse ways) to successfully launch an attack.

We aim to ensure, as much as possible, the authenticity and integrity of software updates, regardless of the source of the download. While we do not guarantee the availability of software updates (e.g., by preventing drop-request attacks), our mechanism must not reduce this quality in existing security systems deployed by OEMs. Another critical goal in formulating our design is that the software update mechanism must be flexible and lean to be considered for deployment by OEMs. This means that the architecture must be flexible enough to support different use models and deployment scenarios. Deployment concerns, such as broad applicability to a variety of architectures, were also taken into account.

5.2 Design overview

Uptane uses a variety of design features to provide a security level for software updates that goes beyond what is used today in the automotive industry. A repository includes the hash of an updated firmware image in a metadata file, and signs that metadata file, which is equivalent to signing the image in the first place in terms of security. A variety of digitally signed metadata files, detailed in the previous section and this one, provides protection against attacks described in Section 3.3. The update might be provided in encrypted format and/or as a differential (delta) update to the vehicle. The repository bundles all images and metadata files, and may use a protected channel such as TLS to connect with a primary in the vehicle. The primary then unpacks the bundle and distributes the update files to the proper secondaries, which then

Boot-loader	Previous metadata	Latest downloaded metadata	**Previous image**	Latest downloaded image (possibly a delta)

Figure 12: An ECU should use *additional storage* to be able to recover from endless data attacks.
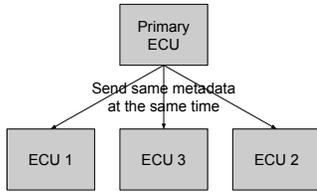
Figure 13: ECUs should be networked such that the primary is forced to *broadcast* metadata to all secondaries.

verify the image before executing the update. Uptane provides a security framework that can be applied to existing update protocols such as OMA-DM, and existing data exchange protocols such as UDS and OTX, as long as those protocols support the exchange of a data block that is provided by the framework.

5.3 Design features

Uptane uses four design features to solve specific security weaknesses identified in the strawman design. These features are: leverage *additional storage* to recover from endless data attacks (Section 5.3.1); *broadcast metadata* to prevent mixed-bundles attacks (Section 5.3.2); utilize a *vehicle version manifest* to detect partial bundle installation attacks (Section 5.3.3); and use a *time server* to limit freeze attacks (Section 5.3.4).

#### 5.3.1 Using additional storage to recover from endless data attacks

Man-in-the-middle attackers can execute endless data attacks on ECUs that have only enough space to keep one image (Section 4.5). Thus, if these attackers send an ECU random data instead of an actual image, then it is unable to boot to a working image, even though the bootloader can verify that that the random data does not match the latest downloaded metadata.

In order to solve this problem, an ECU can use *additional storage*, where it has enough storage to maintain not only a previous image, but also the latest downloaded image (or at least a delta from the previous to the latest downloaded image), as illustrated in Figure 12. This allows an ECU to be updated without having a previous known good image overwritten. Thus, if attackers have tampered with the image during transmission, then it may still boot to a functioning image.

There are many ways of implementing additional storage. The simplest way is to use A/B storage, which requires space to keep two images. Another more cost-efficient way is to require only enough additional space to keep a delta of blocks that have changed from the previous to the latest image. If this delta matches the latest targets and/or director metadata, then the ECU would proceed to update the previous image using the delta. Otherwise, it would discard the delta, and continue to use the previous, working image. This method does not require additional space if the ECU already has enough free space to store this delta.

5.3.2 Broadcasting metadata to prevent mixed-bundles attacks

Attackers who control the primary, or can respond to requests within the vehicle, can execute mixed-bundles attacks because they can show different versions of metadata to different ECUs at the
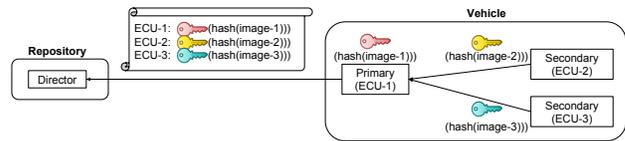
Figure 14: The primary reports the *vehicle version manifest*, or information signed by every ECU about what it has currently installed, whenever it requests updates from the director.
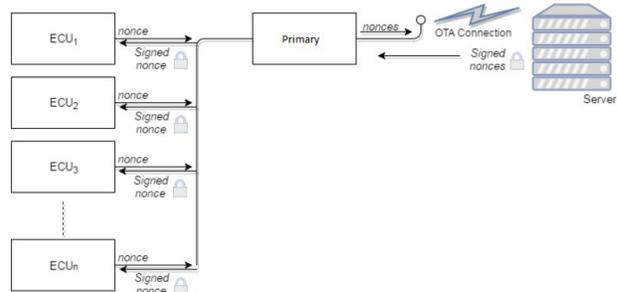
Figure 15: ECUs learn about the latest known time from an external *time server*.

same time (Section 4.5). In order to prevent this problem, the primary should *broadcast* the latest downloaded metadata to secondaries (see Figure 13). Ideally this is implemented by the network (e.g., Ethernet broadcast or CAN bus) such that any metadata sent to one secondary by a primary will be seen by all other secondaries at the same time. Note that, like the rest of Uptane, this feature does not require ECUs to authenticate communications sent to each other. Note that if secondaries are located on different network segments, and an attacker can interrupt and suppress metadata broadcasts on a subset of network segments, then a mixed-bundles attack is possible. However, this attack is less severe in impact compared to more significant threats stemming from other potential attacks, since the adversary has already gained access to the internal vehicle communication network.

5.3.3 Using a vehicle version manifest to detect partial bundle installation attacks

Even if there are no attacks on ECUs, sometimes a scenario might accidentally occur that is similar to the result of a partial bundle installation attack because only some ECUs successfully installed the last set of updates sent by the director. Regardless of how these attacks happened, an OEM can detect these attacks using a *vehicle version manifest*, or information signed by every ECU about what it has currently installed (as illustrated in Figure 14). After verifying and installing an update, every ECU would use its symmetric key, provisioned by the OEM during vehicle manufacture, to attest to information, such as the hash, about what it has installed, and send it to the primary. Note that an ECU would sign and send its manifest only once per ignition cycle. The primary would collect these signatures to build the vehicle version manifest, and send it to the director whenever it requests updates. If the director detects a mismatch between the last updates sent to the vehicle compared to what it has actually installed, then the OEM may be alerted about this for further follow-up.

5.3.4 Using a time server to limit freeze attacks

Attackers can execute freeze attacks on ECUs because unlike desktop and server machines, ECUs do not typically have reliable real-time clocks. Thus, the primary may not be able to tell whether a metadata file has expired (Section 4.5).

Figure 16: Security attacks that affect the primary, organized by attacker capabilities. DR, TS, RS, SP, TR, and RT denote the director, timestamp, release, supplier, targets, and root keys respectively. Red keys are easier to compromise than blue keys. Thin-bordered keys are easier to compromise than thick-bordered keys. The * symbol denotes that an attack is limited to ECUs signed by the given roles. The # symbol denotes than an attack is limited till the earliest expiration timestamp. The @ symbol denotes that an attack can be detected, if not prevented.

Figure 17: Security attacks that affect secondaries if attackers have *not* compromised the primary.

Figure 18: Security attacks that affect secondaries if attackers *have* compromised the primary.

In order to solve this problem, every ECU should periodically (e.g., once between every two ignition cycles) update its current time using an external *time server*. Figure 15 illustrates how this may happen in the background while the vehicle is running. First, every ECU sends a nonce to the primary which, in turn, passes the nonces along to the time server. Then, the time server individually signs for each nonce the current time and nonce, and sends the response to the primary. The primary verifies this response (ensuring that the signatures are valid, and that the time is the same for every nonce), and sends the current time to secondaries.

On reboot of an ECU, while it is running its bootloader, it verifies the signature of the time server using its public key, updates its time, and chooses a new nonce to be sent later to the time server. (Note that an ECU would verify this signature only once per ignition cycle.) Whenever the ECU verifies the latest downloaded metadata, it would check whether the metadata has expired using this latest downloaded time.

Note that if every secondary also includes its latest known time in building the vehicle version manifest, then the director can detect whether the primary is indefinitely replaying the same vehicle version manifest.

5.4 Security analysis

Finally, we discuss why these design features provide improved security over the strawman design (Section 4.5). The figures in this subsection enumerate security attacks that ECUs with different properties are vulnerable to, given various attacker capabilities (such as having compromised a set of keys used to sign updates). Only fixed combinations of keys are shown in these figures, because any other combination would reduce to one of the documented cases. The fixed combination of keys are listed in increasing order of difficulty to compromise. Note that in the following analysis, we assume that all of the aforementioned design features are used.

Figure 16 lists the security attacks to which the primary is vulnerable. All rows, except for the last, are listed in increasing order of difficulty for the attacker to compromise. There are two important differences from Figure 9.

First, although partial bundle installation attacks cannot be prevented, these attacks can be detected by the director (when it has not been compromised) using the vehicle version manifest (marked by the green @ symbol). Hence, when the director can detect these attacks, their impact is temporary, because the director can help vehicles to recover from them.

Second, attackers can execute freeze attacks only after having compromised the timestamp, release, and director keys. These attacks are limited to the earliest expiration timestamp of the root, the director, or a targets metadata file (marked by the green # symbol).

Figure 17 documents the security attacks to which secondaries are vulnerable if attackers have *not* compromised the primary. There are two important differences from Figure 10 and the previous figure. First, since the primary must broadcast metadata to secondaries, attackers cannot execute mixed-bundles attacks. Second, the secondary is not vulnerable to an endless data attack, unless it has been remotely exploited. This is because even if the bootloader finds that the latest downloaded metadata and image do not match each other, then it can restore to the previous working image on additional storage.

Figure 18 documents the security attacks to which secondaries are vulnerable if attackers *have* compromised the primary. The important difference from Figure 11 and the previous figure is that when attackers have compromised the director keys, then attackers are able to execute rollback and arbitrary software attacks on all partial verification secondaries on all vehicles. In contrast, these attacks can be executed on only some full verification secondaries on some vehicles when attackers have compromised at least the right supplier's keys.

6 Conclusions

Software updates in the automotive world are increasingly important, but make vehicles vulnerable at the same time. In this paper, we presented the secure framework Uptane for automotive software updates over-the-air, which is based on the established TUF standard. Uptane is resilient against partial compromise, and Uptane covers far more security considerations than other automotive software update mechanisms we are aware of. For instance, we have shown that vehicles that use Uptane are protected from critical security attacks, in the most likely case where attackers are able to perform man-in-the-middle attacks, but have not compromised any signing key.

To the best of our knowledge, Uptane is also the first software update framework for automobiles that addresses a comprehensive and broad threat model. The design features discussed in this paper require an OEM to make only relatively modest software changes for its ECUs. Uptane also offers OEMs two different security levels that can be tailored to the safety-sensitivity of the ECU. We recommend that safety-critical ECUs implement our full verification model, and non-safety-critical ECUs deploy our partial verification model to save resources without significant loss of security. ECUs that can perform neither full nor partial verification should not be remotely updated; instead, they could be updated using strictly physical means that include a proper proof of physical presence.

Uptane is regularly discussed with the major automotive industry stakeholders in the USA to ensure its ability to be deployed [1]. We strive to make Uptane an industry-wide accepted specification for secure software updates. For more information, a detailed design, specification, and the latest developments on Uptane, please visit our website [2].

7 Acknowledgments

8 References

[1] Uptane discussion forum. https://uptane.umtri.umich.edu/forum/, 2016.

[2] Uptane web site. https://uptane.github.io/, 2016.

[3] Apache Infrastructure Team. apache.org incident report for 8/28/2009. https://blogs.apache.org/infra/entry/apache_org_downtime_report, 2009.

[4] Apache Infrastructure Team. apache.org incident report for 04/09/2010. https://blogs.apache.org/infra/entry/apache_org_04_09_2010, 2010.

[5] B. Arkin. Adobe to Revoke Code Signing Certificate. https://blogs.adobe.com/conversations/2012/09/adobe-to-revoke-code-signing-certificate.html, 2012.

[6] A. Bellissimo, J. Burgess, and K. Fu. Secure software updates: disappointments and new challenges. *Proceedings of USENIX Hot Topics in Security (HotSec)*, 2006.

[7] Brewster, Tom. When governments attack—online. http://www.bbc.com/capital/story/20140414-when-governments-attack, 2014.

[8] D. Burrows. Modelling and resolving software dependencies. https://people.debian.org/~dburrows/model.pdf, 2005.

[9] J. Cappos, S. Baker, J. Plichta, D. Nyugen, J. Hardies, M. Borgard, J. Johnston, and J. H. Hartman. Stork: package management for distributed VM environments. In *The 21st Large Installation System Administration Conference, LISA'07*, 2007.

[10] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman. A look in the mirror: Attacks on package managers. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 565–574. ACM, 2008.

[11] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman. Package management security. *University of Arizona Technical Report*, pages 08–02, 2008.

[12] J. Capppos. *Stork: Secure Package Management for VM Environments*. Dissertation, University of Arizona, 2008.

[13] J. Corbet. An attempt to backdoor the kernel. http://lwn.net/Articles/57135/, 2003.

[14] J. Corbet. The cracking of kernel.org. http://www.linuxfoundation.org/news-media/blogs/browse/2011/08/cracking-kernelorg, 2011.

[15] Debian. Debian Investigation Report after Server Compromises. https://www.debian.org/News/2003/20031202, 2003.

[16] Debian. Security breach on the Debian wiki 2012-07-25. https://wiki.debian.org/DebianWiki/SecurityIncident2012, 2012.

[17] P. W. Frields. Infrastructure report, 2008-08-22 UTC 1200. https://www.redhat.com/archives/fedora-announce-list/2008-August/msg00012.html, 2008.

[18] Gentoo Linux. rsync.gentoo.org: rotation server compromised. https://security.gentoo.org/glsa/200312-01, 2003.

[19] GitHub, Inc. Public Key Security Vulnerability and Mitigation. https://github.com/blog/1068-public-key-security-vulnerability-and-mitigation, 2012.

[20] GitHub Inc. Signing commits using GPG. https://help.github.com/articles/signing-commits-using-gpg/, 2016.

[21] GNU Savannah. Compromise2010. https://savannah.gnu.org/maintenance/Compromise2010/, 2010.

[22] D. Goodin. Attackers sign malware using crypto certificate stolen from Opera Software. http://arstechnica.com/security/2013/06/attackers-sign-malware-using-crypto-certificate-stolen-from-\opera-software/, 2013.

[23] J. Knockel and J. R. Crandall. Protecting Free and Open Communications on the Internet Against Man-in-the-Middle Attacks on Third-Party Software: We're FOCI'd. In *Presented as part of the 2nd USENIX Workshop on Free and Open Communications on the Internet*, Berkeley, CA, 2012. USENIX.

[24] B. M. Kuhn. News: IMPORTANT: Information Regarding Savannah Restoration for All Users. https://savannah.gnu.org/forum/forum.php?forum_id=2752, 2003.

[25] T. K. Kuppusamy, S. Torres-Arias, V. Diaz, and J. Cappos. Diplomat: Using delegations to protect community repositories. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 567–581, Santa Clara, CA, Mar. 2016. USENIX Association.

[26] H. Magnusson. The PHP project and Code Review. http://bjori.blogspot.com/2010/12/php-project-and-code-review.html, 2010.

[27] Microsoft, Inc. Flame malware collision attack explained. http://blogs.technet.com/b/srd/archive/2012/06/06/more-information-about-the-digital-certificates-used-to-sign-\the-flame-malware.aspx, 2012.

[28] V. Miller. Remote exploitation of an unaltered passenger vehicle. http://illmatics.com/Remote%20Car%20Hacking.pdf, 2015.

[29] M. Mullenweg. Passwords Reset. https://wordpress.org/news/2011/06/passwords-reset/, 2011.

[30] Red Hat, Inc. Infrastructure report, 2008-08-22 UTC 1200. https://rhn.redhat.com/errata/RHSA-2008-0855.html, 2008.

[31] RubyGems.org. Data Verification. http://blog.rubygems.org/2013/01/31/data-verification.html, 2013.

[32] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine. Survivable key compromise in software update systems. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 61–72. ACM, 2010.

[33] Slashdot Media. phpMyAdmin corrupted copy on Korean mirror server. https://sourceforge.net/blog/phpmyadmin-back-door/, 2012.

[34] J. K. Smith. Security incident on Fedora infrastructure on 23 Jan 2011. https://lists.fedoraproject.org/pipermail/announce/ 2011-January/002911.html, 2011.

[35] The FreeBSD Project. FreeBSD.org intrusion announced November 17th 2012. http://www.freebsd.org/news/2012-compromise.html, 2012.

[36] The PHP Group. php.net security notice. http://www.php.net/archive/2011.php#id2011-03-19-1, 2011.

[37] The PHP Group. A further update on php.net. http://php.net/archive/2013.php#id2013-10-24-2, 2013.

[38] C. Thuen. Remote control automobiles, 2015.

[39] L. Voss. Newly Paranoid Maintainers. http://blog.npmjs.org/ post/80277229932/newly-paranoid-maintainers, 2014.