Atualização pelo Ar (OTA) de Dispositivos Embarcados Veiculares com Computação Confiável Fim-a-Fim

Guilherme A. Thomaz[1], **Matteo Sammarco**[2] **e Miguel Elias M. Campista**[1]

[1]Grupo de Teleinformática e Automação (GTA)
Universidade Federal do Rio de Janeiro (UFRJ)
[2]Pesquisador independente

{guiaraujo,miguel}@gta.ufrj.br, matteosam@hotmail.it

Resumo. *Os veículos conectados possuem softwares que precisam ser atualizados para corrigir vulnerabilidades e adicionar novas funcionalidades. As atualizações Over-the-Air (OTA) evitam que o proprietário leve o veículo até uma assistência técnica, mas abrem brechas para ataques que modificam os executáveis e colocam vidas em risco. Este trabalho propõe uma arquitetura de OTA que combina as duas tecnologias de segurança por hardware mais adotadas no mercado: o Intel SGX no servidor e o ARM TrustZone no cliente. O trabalho se diferencia por propor o uso de computação confiável fim-a-fim para proteger a arquitetura de atacantes capazes de controlar todo o sistema operacional, tanto no servidor quanto no veículo. A implementação utiliza o CACIC-DevKit no servidor e um dispositivo embarcado veicular com o sistema seguro OP-TEE. Os experimentos revelam que o impacto do uso de TEE é de apenas 2% do tempo total para transferência de um bloco de 1KB de software.*

Abstract. *Connected vehicles have software that needs to be updated to correct vulnerabilities and add new features. Over-the-air (OTA) updates prevent the owner from bringing the vehicle to a service center but enable attacks that modify the executables, putting lives at risk. This paper proposes an OTA architecture that combines the two most widely adopted hardware security technologies: Intel SGX on the server and ARM TrustZone on the client. This work stands out by proposing end-to-end trusted computing to protect the architecture from attackers capable of controlling the entire operating system, both in the server and vehicle. The implementation uses the CACIC-DevKit on the server and a vehicular-embedded device with the OP-TEE secure system. The experiments reveal that the impact of TEE is only 2% of the total time for transferring a 1KB software block.*

1. Introdução

Os veículos modernos possuem dezenas de Unidades de Controle Eletrônicas (*Electronic Controller Units* – ECUs) responsáveis por funcionalidades de segurança e até mesmo de entretenimento [Kornaros et al. 2020]. Entretanto, alguns trabalhos demonstram ataques cibernéticos capazes de desligar o freio ou o motor, por exemplo, levando ao *recall* de milhões de veículos por montadoras [Koscher et al. 2010]. Ademais, a

Este trabalho foi realizado com recursos do CNPq, CAPES, FAPERJ, FAPESP (2015/24485-9, 2014/50937-1) e Fundação de Desenvolvimento da Pesquisa - Fundep - Rota 2030.

recente popularização dos veículos elétricos e autônomos aumentou a quantidade de *software* embarcado, sugerindo que os gastos das montadoras para corrigir erros em *softwares* aumentará. Nesse contexto, sistemas de atualizações pelo ar (*Over-the-air* – OTA) aproveitam que os veículos modernos são conectados à Internet para difundir novas versões dos *softwares* e evitar que os veículos sejam levados até a assistência técnica. Além de reduzir os custos das montadoras com *recalls*, as atualizações OTA permitem a adição de novas funcionalidades nos veículos ao longo do tempo, aumentando a satisfação do cliente [Henriques 2022, Mukherjee et al. 2021]. Entretanto, atacantes já exploraram as atualizações OTA para introduzir *malwares*, causando prejuízos estimados em centenas de milhões de dólares [Hern 2013]. Em um contexto veicular o cenário se agrava, pois vidas são colocadas em risco.

Uma resposta para aumentar a segurança de atualizações OTA veiculares é o uso de Ambientes de Execução Confiáveis (*Trusted Execution Environments* – TEE) na ECU. O TEE utiliza recursos de *hardware* para proteger as atualizações OTA até mesmo no caso de um atacante que controle todo o *firmware* [Henriques 2022, Mukherjee et al. 2021, Kornaros et al. 2020]. Essas propostas assumem que o servidor em nuvem que envia os *softwares* é confiável. Essa premissa vai na contramão dos paradigmas de segurança mais recentes, como o de confiança zero (*Zero Trust*), no qual os ataques podem vir de usuários de dentro da infraestrutura em nuvem que abusam de seus privilégios [Sarkar et al. 2022]. Um caminho natural para mitigar esse problema em sistemas OTA veiculares é integrar soluções de TEE na ECU com TEE em nuvem.

Este trabalho propõe uma arquitetura que utiliza TEE no servidor da montadora e na ECU para proteger as atualizações OTA veiculares. A principal contribuição do trabalho em relação às outras propostas na literatura é a integração do Intel SGX (*Software Guard Extensions*) em nuvem com o ARM TrustZone na ECU para proteger a transferência de *softwares* no ambiente veicular. A proposta de computação confiável fim-a-fim se diferencia de outros trabalhos por prover a transferência segura do *sopftware* até mesmo na presença de atacantes privilegiados em todos os componentes da arquitetura. A implementação utiliza as ferramentas OP-TEE, CACIC-DevKit e SQLite3 e a avaliação de desempenho revela que a latência introduzida pelo uso do TEE representa apenas 2% do tempo total de transferência de *software* de 1KB, confirmando que os mecanismos de segurança propostos não representam obstáculos para a operação da proposta.

Este trabalho está estruturado da seguinte forma. A Seção 3 descreve um sistema de atualizações OTA veicular típico e define os requisitos de segurança e o modelo de atacante. A Seção 4 apresenta os ambientes de execução confiáveis, com foco no SGX e no TrustZone. A Seção 5 descreve os componentes da arquitetura proposta e suas interfaces. A Seção 6 descreve o *hardware* e os *softwares* utilizados para implementação da arquitetura, bem como os desafios encontrados. A Seção 7 apresenta os resultados de avaliação de desempenho. Por fim, a Seção 8 apresenta os trabalhos relacionados, enquanto a Seção 9 conclui o trabalho e aponta direções futuras.

2. Trabalhos relacionados

O arcabouço de atualizações OTA mais adotado na indústria é o Uptane, proposto por Karthik *et al.* [Karthik et al. 2016]. O artigo do Uptane conta com a participação de empresas automotivas importantes e estende as especificações do TUF para resolver

problemas de segurança específicos de sistemas veiculares. No Uptane, os roubos de chaves nos servidores podem ser mitigados por revogação, mas não são eliminados, pois a implementação não é resistente a atacantes com alto nível de privilégio. No lado do cliente, a implementação assume que o atacante não consegue controlar arbitrariamente o *firmware* das ECUs primárias e secundárias. Assim, outros trabalhos na literatura expandem esse arcabouço ou propõem arquiteturas alternativas.

Henriques executa a verificação do *software* baixado na ECU primária em um enclave SGX [Henriques 2022]. O serviço de verificação é implementado no veículo usando o Uptane, chamadas RPC (*Remote Procedure Call*) e a plataforma Gramine-SGX para desenvolvimento de enclaves [Tsai et al. 2017]. Os resultados da avaliação de desempenho mostram que a sobrecarga introduzida pelo uso dessas ferramentas em relação ao Uptane padrão é de 91%. A proposta é muito complexa para dispositivos com poucos recursos computacionais. Atualmente, o uso de SGX no veículo não é economicamente viável, pois a tecnologia está disponível apenas em processadores de servidores com alto custo e alto consumo energético.

Mukherjee *et al.* adaptam o Uptane para atualizar os *softwares* de veículos elétricos com bateria no mundo seguro do TrustZone [Mukherjee et al. 2021]. Os metadados e a imagem são transferidos para uma aplicação confiável do OP-TEE rodando em um Raspberry Pi 3, por meio da interface UART e do protocolo UDS (*Unified Diagnosis Service*). O trabalho utiliza a ferramenta SAW (*Software Analysis Workbench*) para verificar vulnerabilidades no código usando modelos semânticos, mas não apresenta resultados de desempenho [Carter et al. 2013]. Além disso, o modelo de atacante assume que os servidores de data e hora, repositório de imagens e diretor são seguros.

Kornaros *et al.* usam o TrustZone para proteger aplicações críticas de uma motocicleta, como o controlador de tração, o comando de aceleração (*drive-by-wire*) e o sistema de atualização OTA da interface gráfica do painel [Kornaros et al. 2020]. O artigo também estende a segurança do protocolo CAN e implementa em uma FPGA (*Field Programmable Gate Array*) um *firewall on-chip* para proteção contra ataques físicos à memória. Os autores reproduzem o fluxo completo de atualização de *software* em uma moto na presença de um atacante, utilizando microcontroladores com o TrustZone-M. O gerenciamento do *software* no servidor e a transferência para o veículo estão fora do escopo. Assim, todas as medidas de desempenho e funcionalidades de segurança se limitam aos CIs do veículo e à rede local.

Diferente dos trabalhos anteriores, este trabalho propõe uma arquitetura que utiliza TEE para prover segurança fim-a-fim na atualização de *softwares* de veículos por OTA. O sistema é compatível com as características desejáveis dos sistemas comerciais de OTA veicular. Ao mesmo tempo, ele é resiliente a atacantes privilegiados nos servidores e no veículo. A avaliação de desempenho revela que a sobrecarga adicionada pelos procedimentos propostos é baixa.

3. Segurança de Atualizações OTA em Veículos Conectados

Atualmente, os veículos possuem dezenas de ECUs que se conectam por redes em barramento, como a CAN (*Controller Area Network*). Um tipo especial é a ECU primária, também chamada de nó de conectividade, que funciona como um *gateway* para comunicação das ECUs secundárias com dispositivos sem fio através de tecnologias como

o IEEE 802.11p, 802.15.4, LTE e 5G NR. Uma das formas de atualizar os *softwares* das ECUs é por meio de serviços prestados nas concessionárias, que consistem em carregar a versão mais recente do sistema em memória *flash* usando uma interface física como a OBD-II (*On-Board Diagnosis*). Outra forma é através de atualizações OTA, nas quais o *software* é enviado para a ECU primária pela interface sem fio e redistribuído para as ECUs secundárias [Karthik et al. 2016]. Os sistemas de atualização OTA de veículos conectados devem garantir:

1. confidencialidade do *software*: o fornecedor deseja manter o sigilo do código binário para evitar engenharia reversa;
2. integridade do *software*: um ataque que injete um *software* malicioso pode levar até mesmo a fatalidades;
3. autenticidade das ECUs: as ECUs utilizadas nos diferentes modelos podem ser muito diferentes entre si, de modo que é necessário que apenas a ECU correta receba o seu *software* compatível;
4. prazo de validade do *software*: um atacante pode tentar forçar uma atualização de uma versão mais antiga com vulnerabilidades de segurança conhecidas;
5. disponibilidade do processo: um atacante pode negar a atualização OTA para manter o *software* desatualizado.

Os sistemas de atualização de *software* como o Windows Update e o Python Package Index (PyPI) utilizam o padrão TUF (*The Update Framework*). Entretanto, o trabalho desenvolvido por Karthik *et al.* mostra que esse padrão não é flexível o suficiente para lidar com a diversidade dos *softwares* de ECUs gerenciados por fabricantes de veículos conectados [Karthik et al. 2016]. Os autores destacam que o projeto de um sistema de atualização OTA deve levar em conta o fluxo completo do processo de desenvolvimento e distribuição dos *softwares*. Tipicamente, o fabricante original de equipamentos (*Original Equipment Manufacturer* – OEM) é responsável por distribuir os *softwares* para as ECUs por meio de repositórios, enquanto que os fornecedores (*suppliers*) apenas assinam e entregam os *softwares* para os fabricantes.

Este trabalho considera um modelo de atacante mais forte do que o utilizado em outros trabalhos da literatura, capaz de controlar os sistemas operacionais e os *firmwares*, tanto das ECUs quanto do repositório de *softwares* em nuvem. Assim, o atacante pode ler e modificar qualquer pacote de rede e arquivo, bem como todos os dados na memória principal e nos registradores da CPU, tanto no veículo quanto no servidor. Para tornar a arquitetura resiliente a esse modelo de atacante, este trabalho propõe uma solução de computação confiável fim-a-fim, utilizando tecnologias discutidas na próxima seção. Ataques de canal lateral e ataques à disponibilidade, como negação de serviço (*Denial of Service* – DoS), estão fora do escopo do artigo, podendo ser mitigados com soluções ortogonais às discutidas [De Souza et al. 2022, Oleksenko et al. 2018].

4. Ambientes de Execução Confiáveis

A Computação Confiável (*Trusted Computing*) é um conjunto de tecnologias que garantem segurança por recursos de *hardware*. A primeira solução proposta pelo Grupo de Computação Confiável (*Trusted Computing Group* – TCG) foi o Módulo de plataforma Confiável (*Trusted Plattform Module* – TPM): um microcontrolador que armazena chaves e executa programas simples de criptografia em um ambiente isolado. Uma

evolução dessa ideia são os Ambientes de Execução Confiáveis (*Trusted Execution Environments* – TEE), que isolam todas as etapas do armazenamento e processamento de dados sensíveis. A vantagem do uso de TEE em conjunto com os mecanismos de criptografia e autenticação tradicionais é a proteção contra atacantes privilegiados, que controlam o super-usuário, o sistema operacional ou o hipervisor. A principal implementação de TEE da arquitetura ARM é o TrustZone e da arquitetura x86, da Intel, é o SGX (*Software Guard Extensions*).

4.1. Intel SGX

O Intel SGX estende o conjunto de instruções x86 para que uma aplicação instancie regiões protegidas na memória principal, denominadas enclaves [Costan and Devadas 2016]. As instruções que inicializam o enclave na memória calculam uma função resumo (*hash*) do seu conteúdo em nível de microarquitetura, de modo que nenhum *software* pode mudar o resultado dessa medição. Um mecanismo de atestação do enclave assina a medida criptográfica com uma chave exclusiva da plataforma para provar a autenticidade do código para uma entidade remota. Essa funcionalidade assume que a Intel oferece um serviço seguro para provisionar as chaves privadas para os enclaves e verificar a autenticidade das assinaturas [Scarlata et al. 2018].

Os enclaves são implementados utilizando um Kit de Desenvolvimento de *Software* (*Software Development Kit* – SDK) em C/C++. Os enclaves são compilados como bibliotecas dinâmicas, que contêm rotinas chamadas por aplicações interessadas em realizar um processamento sensível. O acesso à faixa de endereços de memória reservados para o enclave envolve verificações de segurança em nível de microarquitetura de modo a garantir a confidencialidade e a integridade dos códigos e dados, mesmo quando o sistema operacional é malicioso. Os dados sensíveis processados por um enclave podem ser encriptados com uma chave de selagem acessível apenas pelo enclave antes de serem escritos em disco [Anati et al. 2013].

4.2. ARM TrustZone

A arquitetura ARM é comum em microcontroladores e *Systems on Chip* (SoC), utilizados em sistemas embarcados e *smartphones*. O ARM TrustZone separa todo o *hardware* e *software* em um mundo seguro (*Trusted Execution Environment* – TEE) e um mundo normal (*Rich Execution Environment* – REE) [Ngabonziza et al. 2016]. A distinção entre os modos seguro e inseguro é feita pelo bit 33 do registrador de configuração de segurança (*Secure Configuration Register* – SCR), que só pode ser modificado por um *software* monitor. O mundo seguro contém sua própria faixa de endereços de memória, seu próprio mapeamento de dispositivos de entrada e saída e seu próprio *kernel*, de modo que acessos do mundo inseguro são negados.

O TrustZone não define por padrão um sistema de armazenamento seguro e atestação. A arquitetura ARM é utilizada por diversos fabricantes e a implementação e a disponibilização desses mecanismos varia de dispositivo para dispositivo. Para garantir uma raiz de confiança no *hardware*, o dispositivo pode oferecer uma chave OTP (*One-Time Programmable*) acessível no mundo seguro. Ademais, para evitar que um atacante comprometa o sistema operacional seguro, o dispositivo deve prover um mecanismo de inicialização segura, ou seja, o inicializador (*bootloader*) deve calcular um *hash* do *kernel* do TEE e comparar com o valor esperado gravado em uma memória persistente segura.

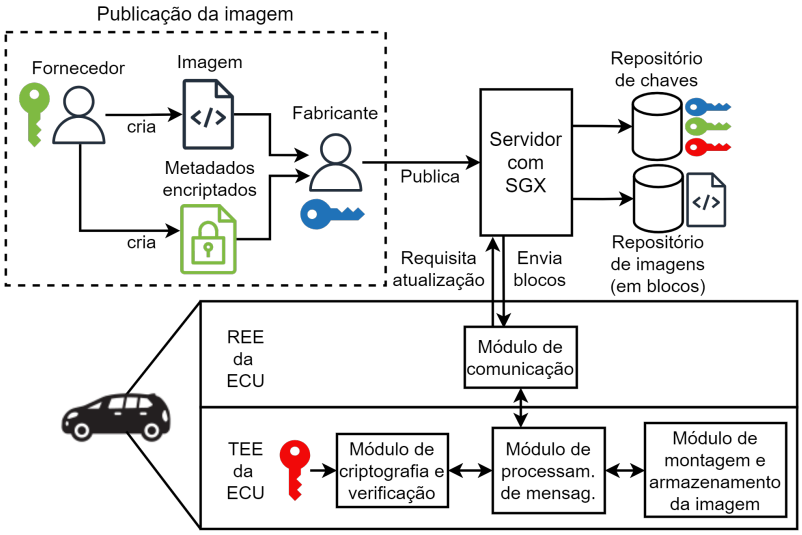

Figura 1. Arquitetura proposta. O servidor com SGX verifica se o fabricante publicou a imagem do fornecedor sem comprometê-la e deve disponibilizá-la para o TEE das ECUs.

5. Proposta para Atualizações OTA com Computação Confiável Fim-a-Fim

A arquitetura proposta, apresentada na Figura 1, é compatível com os papéis descritos na Seção 3, facilitando a sua adoção em sistemas OTA automotivos comerciais. Os componentes e procedimentos serão discutidos ao longo desta seção.

Os fornecedores, o fabricante e as ECUs dos veículos compatíveis com o esquema de OTA proposto são os clientes, que se comunicam com um servidor compatível com o Intel SGX. Cada cliente é identificado por um número hexadecimal público (`ID`), e possui uma chave de comunicação (`CC`) simétrica secreta utilizada para encriptar mensagens enviadas para o servidor. As `CC`s do fornecedor e do fabricante são compartilhadas com o servidor caso ele consiga provar para esses clientes que é seguro por meio da atestação, apresentada na Seção 4. Para garantir a autenticidade das mensagens, o modelo de atacante assume que o fabricante e o fornecedor devem guardar a `CC` de forma segura, sem que um atacante consiga roubá-la e fabricar mensagens falsas. Ademais, o fabricante grava a `CC` da ECU em uma memória persistente do TEE da ECU com TrustZone. O servidor encripta as `CC`s com sua chave de selagem antes de escrevê-las no disco e utiliza o `ID` correspondente para localizar a chave, evitando que um atacante com alto nível de privilégio no servidor roube as chaves dos clientes.

Os *softwares* são disponibilizados no formato de imagens armazenadas em um repositório controlado pelo servidor, como em trabalhos anteriores [Karthik et al. 2016, Mukherjee et al. 2021]. O repositório consiste em um banco de dados que armazena as imagens em pedaços encriptados com uma chave de armazenamento `CA`, denominados blocos. A `CA` é selada em disco de modo que apenas o enclave do servidor consegue decriptar os blocos. No *download* e *upload* de uma imagem a transmissão é feita bloco a bloco. Para evitar corrupção e perda nas transmissões, cada bloco é enviado utilizando TCP, consequentemente criando um canal confiável na camada de transporte. Cada *software* possui um código identificador do *software* (CIS) e cada versão possui um número de versão (NV). Além disso, há uma lista de `ID`s de clientes que podem baixar o *software*,

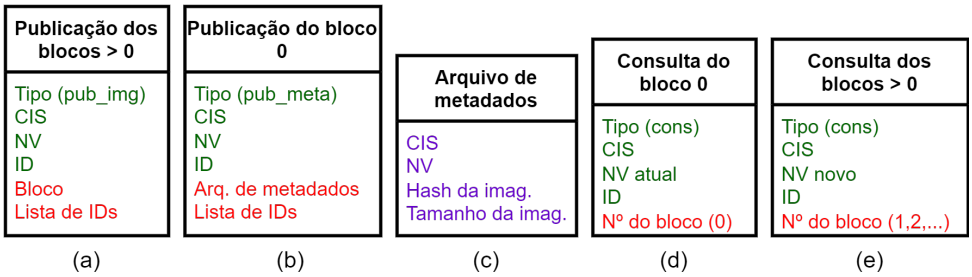

Figura 2. Campos das mensagens definidas na arquitetura. A cor verde indica dados em claro, vermelho indica dados encriptados pelo fabricante e roxo indica dados encriptados pelo fornecedor. O *nonce* e o *hash* são omitidos.

encriptada junto com os blocos.

Antes de serem encriptados, mensagens e dados sensíveis são concatenados com um número aleatório único (*nonce*), e o dado encriptado acompanha uma cópia desse número em claro. Na decriptação, o sistema deve verificar: i) se o *nonce* decriptado coincide com a cópia em claro, e ii) se o *nonce* não foi utilizado previamente, evitando assim ataques de reprodução (*replay*). Além disso, a encriptação deve ser acompanhada de um *hash* dos dados concatenado com o *nonce*. O *hash* possibilita que o sistema verifique a integridade da mensagem, comparando o *hash* do dado decriptado com o *hash* em claro.

5.1. Publicação de uma imagem

Após desenvolver um novo *software* ou uma nova versão de um *software*, o fornecedor o envia para o fabricante, responsável por gerenciar o repositório. O envio da imagem do fornecedor para o fabricante pode ser feito por qualquer canal de comunicação seguro externo à arquitetura. O fabricante pode realizar testes estáticos e dinâmicos na imagem recebida do fornecedor para verificar possíveis problemas. Vale notar que tanto o envio da imagem quanto os testes estão fora do escopo deste trabalho. O objetivo dos protocolos propostos pela arquitetura para *upload* é garantir que o *software* só seja disponibilizado se fabricante e fornecedor agirem de acordo com o protocolo. Assim, se o fornecedor sabotar o *software* do fabricante ou se o fabricante modificar o *software* de forma indevida, o servidor irá detectar a ação maliciosa e as ECUs não conseguirão fazer o *download*.

Para cada bloco da imagem, o fabricante monta uma mensagem de publicação de bloco, cujos campos são apresentados na Figura 2a. O tipo da mensagem (publicação de imagem), o `CIS`, o `NV` e o `ID` do fabricante são enviados em claro. O conteúdo do bloco em si, o `ID` do fabricante e a lista de `IDs` de ECUs que podem baixá-lo são encriptados com a `CC` do fabricante. Após enviar a mensagem de publicação para o servidor, apenas o enclave é capaz de decriptá-la com a `CC` do fabricante. Por fim, o enclave encripta com a `CA` o bloco recebido junto com o `CIS`, o `NV` e a lista de `IDs` antes de publicá-lo no banco de dados de blocos de imagens. O `CIS` e o `NV` podem ser usados para busca de blocos no banco de dados.

O fornecedor também cria um arquivo de metadados que inclui a função resumo (*hash*) da imagem, o `CIS`, o `NV` e o tamanho do arquivo, conforme a Figura 2c [Karthik et al. 2016]. Esse arquivo é encriptado com a `CC` do fornecedor e en-

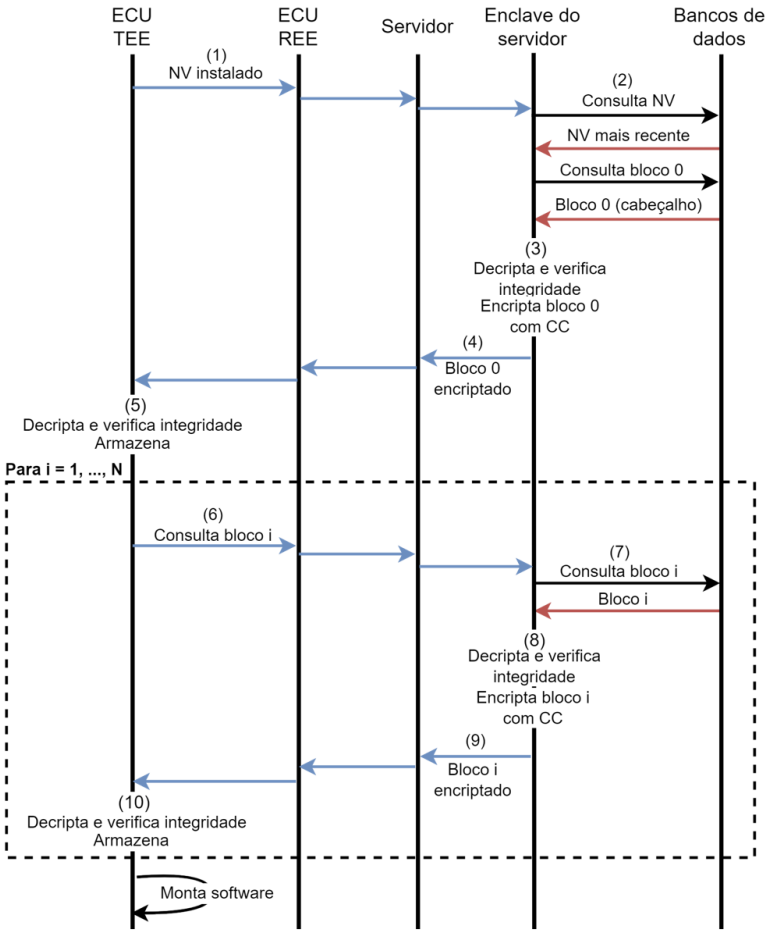

Figura 3. Protocolo de transferência OTA de um *software* com computação confiável fim-a-fim no caso de um *download* bem sucedido.

tregue ao fabricante junto com a imagem, de modo que o fabricante não consiga decriptá-lo. Após publicar todos os blocos, o fabricante publica os metadados encriptados pelo fornecedor usando a mensagem apresentada na Figura 2b. Essa etapa é importante, pois o servidor irá verificar se os blocos publicados previamente estão de acordo com os metadados do fornecedor, ou se o fabricante comprometeu a imagem. Para isso, o enclave do servidor decripta os metadados utilizando a `CC` do fornecedor, busca os blocos do *software* publicados previamente usando o `CIS` e o `NV`, monta a imagem completa na memória e verifica se o *hash* e o tamanho coincidem com o especificado nos metadados. Em caso afirmativo, os metadados encriptados são armazenados no banco de dados seguro como um bloco especial da imagem, chamado de bloco 0 ou bloco de metadados. Caso contrário, o servidor não publica o bloco 0 e, sem esse bloco, as ECUs não conseguirão prosseguir com o *download* dos outros blocos.

5.2. Transferência OTA para a ECU

A principal contribuição do artigo está no protocolo para atualização OTA com computação confiável fim-a-fim, ilustrado na Figura 3. A numeração na figura é referenciada entre parênteses ao longo do texto para facilitar a explicação. O TEE da ECU envia uma mensagem inicial de consulta apresentada na Figura 2d (1). Após recuperar a `CC` selada em disco e decriptar a mensagem, o enclave do servidor consulta o banco de dados

para verificar se existe uma imagem com mesmo `CIS`, e `NV` maior e, se houver, consulta o bloco 0 (2). Caso não haja uma versão mais recente, o servidor retorna uma mensagem para que a ECU finalize o procedimento. Em seguida, o enclave decripta o bloco 0, verifica se o `NV` coincide com o esperado e o envia para o TEE da ECU (3 e 4). Apenas após decriptar e efetuar as verificações de segurança no bloco 0, (5) o TEE da ECU pode prosseguir com o *download* dos blocos propriamente ditos.

O TEE da ECU prossegue com o envio de mensagens de consulta para os blocos seguintes (1, 2, ..., N, onde N é o número de blocos da imagem), ilustradas na Figura 2d (6). Para cada mensagem, o enclave do servidor consulta o bloco (7), o decripta e o retorna para o TEE da ECU na forma encriptada (8 e 9). O TEE da ECU decripta e armazena cada um dos N blocos da imagem (10) para, ao final, reconstruir a imagem completa. O procedimento de instalação da aplicação ou *firmware* na ECU independe da arquitetura.

Vale relembrar que cada bloco, incluindo o bloco 0, é encriptado junto com uma lista de `IDs` de ECUs que podem baixá-los, especificada pelo fabricante na mensagem de publicação. Assim, antes de encriptar um bloco e enviá-lo para a ECU, o enclave verifica se o `ID` da ECU está na lista de permissões de acesso. Assim, o fabricante, conhecendo os `IDs` das ECUs, controla quais componentes nos seus veículos podem acessar cada *software* de forma agnóstica ao fornecedor.

O enclave pode obter um bloco inválido do banco de dados caso a verificação de integridade com *hash* ou a verificação do `NV` mais recente falhem. Nesse caso, o servidor encaminha uma mensagem de bloco inválido para o cliente, pois o banco de dados pode ter sido comprometido por um atacante. A estratégia usada para recuperar uma versão válida no banco de dados foge do escopo. As mesmas verificações de segurança do lado da ECU podem revelar a presença de um *man-in-the-middle* interceptando e falsificando mensagens. Nesse caso, a ECU pode repetir a requisição algumas vezes antes de desistir e abortar a operação.

6. Implementação do Sistema Proposto

A implementação da arquitetura permite e transferência segura dos blocos de *software* de um servidor com o Intel SGX para uma ECU com o TrustZone.

6.1. Implementação do servidor com o CACIC-DevKit

O servidor foi implementado utilizando o CACIC-DevKit, apresentado em [Thomaz et al. 2023a], que permite o desenvolvimento de sistemas de processamento e controle de acesso a dados publicados em nuvem utilizando uma arquitetura com enclaves denominada CACIC (Controle de Aceso Confiável a Dados da Internet das Coisas em Nuvem utilizando Enclaves). A API (*Application Programming Interface*) do CACIC-DevKit fornece funções em C++ flexíveis quanto à estrutura do banco de dados e aos tipos de dados. Assim, a ECU primária pode utilizar uma aplicação programada com as funções de cliente do CACIC-DevKit para enviar as mensagens de consulta a blocos ao servidor. A ferramenta chama funções de um enclave do SGX para realizar as operações criptográficas e verificações de segurança e controle de acesso necessárias para a publicação e consulta de forma transparente ao desenvolvedor. Assim, a escolha dessa ferramenta se justifica por abstrair as funções do Intel SGX SDK, simplificando a programação sem introduzir sobrecarga de desempenho significativa [Thomaz et al. 2023b]. A troca de mensagens

no CACIC-DevKit utiliza uma implementação em C++ do protocolo HTTPS denominada `cpp-httplib` [1]. O banco de dados para armazenamento das imagens do lado do servidor é implementado com o SQLite3, sendo os atributos `CIS` e `NV` utilizados para buscas dos blocos utilizando a Linguagem de Consultas Estruturada (*Structured Query Language* – SQL). O *hardware* do servidor é um computador com CPU Intel i9-10900 2.80 GHz, 20 *threads* e 32GB de RAM e o sistema operacional é um Ubuntu 20.04 LTS. A implementação do servidor encontra-se disponível em `https://github.com/GTA-UFRJ/CACIC-DevKIt/tree/ota-vehicle-extension`.

6.2. Implementação do cliente com o OP-TEE

Para implementação da aplicação na ECU, foi utilizado um Raspberry Pi 3 Model B com SoC Broadcom BCM2837, CPU com 4 núcleos Cortex-A53 1.2GHz 64 bits e 1GB de RAM. O REE executa um sistema operacional de tempo real (*Real-Time Operating System* – RTOS) baseado no *kernel* Linux RT_PREEMPT. O sistema operacional no TEE é um Linux leve denominado OP-TEE [Göttel et al. 2019]. A escolha do sistema se justifica por ele ser adequado para dispositivos embarcados com poucos recursos computacionais e por ser adotado em outros trabalhos [Mukherjee et al. 2021]. O *bootloader* da placa inicializa os sistemas operacionais gravados em um cartão microSD.

O REE não possui *drivers* para vídeo HDMI nem USB apesar da disponibilidade dos *hardwares* na placa. Assim, o REE transmite o terminal através da interface UART (*Universal Asynchronous Receiver/Transmitter*), disponível nos pinos de GPIO (*General Purpose Input/Output*) da placa. Um computador pessoal executa a aplicação `picocom` para se comunicar com a UART por meio de um adaptador USB/UART e, assim, acessar o terminal do REE. O sistema também não possui aplicações para acesso remoto por SSH (*Secure Shell*), compilação e depuração em C e gerenciamento e instalação de pacotes de *software*. Os códigos necessários para a aplicação cliente são compilados com o *kernel*. O TEE é mais simples ainda, consistindo em um sistema operacional sem linha de comando executado concorrentemente ao REE. Esse sistema recebe chamadas do REE por soquetes de rede utilizando uma API que segue a especificação industrial GlobalPlatform API para interação com dispositivos seguros, como TEEs [Suzaki et al. 2020]. O TEE é responsável por executar Aplicações Confiáveis (*Trusted Applications* – TAs) dentro do TrustZone e retornar os resultados para a aplicação no REE que invocou a chamada.

A aplicação cliente é dividida em dois módulos. O primeiro módulo troca de mensagens com o servidor, e é executado no REE, onde se encontram os *drivers* de *Ethernet*. O segundo módulo é responsável pelas operações de montagem e encriptação das requisições e decriptação e armazenamento das respostas, e é executado como uma TA no TEE, por se tratar de operações sensíveis para a segurança. Entretanto, a API de cliente do CACIC-DevKit não foi programada dessa forma, pois assume a ausência de um atacante com alto nível de privilégio do lado do cliente. Essa decisão de projeto exigiu o desenvolvimento de novas funções também para o cliente CACIC-DevKit a fim de torná-lo compatível com o OP-TEE.

O esforço de portabilidade envolveu superar três obstáculos principais. O primeiro foi a programação da TA em C, que utiliza as bibliotecas do OP-TEE ao invés das tipicamente disponíveis no Linux. O segundo foi o desenvolvimento da aplicação do

[1] `https://github.com/yhirose/cpp-httplib`

lado do REE, pois a biblioteca `cpp-httplib` não é compilada nesse sistema. Logo, um módulo para montagem, envio e recepção das mensagens HTTP a partir das interfaces `read` e `write` teve que ser implementado. O terceiro desafio surgiu pelo fato do CACIC utilizar o algoritmo de criptografia autenticada AES-GCM (*Advanced Encryption Standard - Galois/Counter Mode*). Como esse algoritmo não é disponibilizado no OP-TEE, foi necessário mudar o código do CACIC-DevKit para utilizar o AES-CTR (*Counter Mode*). Diferente do GCM, o modo CTR do AES não gera um Código de Autenticação de Mensagem (*Message Authentication Code* – MAC). Para atender aos requisitos da Seção 3, foi necessário implementar o mecanismo de verificação de integridade, conforme descrito na Seção 5. Assim, um SHA256 (*Secure Hash* com 256 bits) da mensagem em claro junto com o *nonce* é computado e transmitido junto com a mensagem. A implementação do cliente encontra-se disponível em `https://github.com/GTA-UFRJ/Secure-E2E-OTA`.

7. Avaliação de Desempenho

O primeiro experimento avalia a sobrecarga introduzida por cada etapa do processamento no tempo de transferência de um bloco de 1KB (1024 *bytes*). O objetivo é avaliar o impacto dos mecanismos de segurança com TEE propostos no tempo de *download* utilizando a implementação descrita na Seção 6. Para medir os tempos de execução no servidor, utilizou-se uma ferramenta em C++ desenvolvida pelo autor, baseada no uso de temporizadores de alta precisão, conforme apresentado em [Thomaz et al. 2023b]. O cliente utiliza ferramentas de medida de tempo da biblioteca padrão C no REE e da biblioteca do OP-TEE. A Tabela 1 apresenta o tempo médio e a incerteza, com intervalo de confiança de 95%, obtidos após repetir o experimento 500 vezes.

A preparação da sessão entre REE e TEE é realizada apenas na inicialização da aplicação de *download*, antes da requisição ao bloco 0. Essa etapa cria um armazenamento persistente para os blocos no TEE. A inicialização do enclave é realizada apenas na inicialização do servidor, pois o CACIC-DevKit utiliza múltiplas *threads* dentro do mesmo enclave para processar as requisições de forma concorrente. A finalização da sessão na ECU é realizada apenas após a montagem do *software*, com o recebimento do último bloco. O tempo de transferência de um bloco de 1KB, descontando esses procedimentos de preparação de sessão, inicialização e finalização, é igual a 59,3ms.

A sobrecarga introduzida pelo TEE se deve: i) à necessidade de transferir os

Tabela 1. Sobrecarga introduzida por cada etapa na transferência OTA de um bloco do servidor para a ECU.

Local	Procedimento		Tempo médio (ms)	Incerteza (ms)
ECU	Preparação da sessão entre REE e TEE		181	2,00
Servidor	Inicialização do enclave		6,18	0,06
ECU	Construção da requisição	Dentro do TEE	0,39	0,04
		Sobrecarga da API com TA	0,53	0.04
ECU	Envio da mensagem de consulta (no REE)		1,27	0,08
Servidor	Consulta ao bloco e envio		4,68	0,01
ECU	Decriptação e armazenamento do bloco recebido	Decriptação dentro do TEE	0,78	0,04
		Armazenamento persistente no TEE	50,92	0,40
		Sobrecarga da API com TA	0,70	0,57
ECU	Finalização da sessão entre REE e TEE		15,17	0,01

dados do REE para o TEE por meio da GlobalPlattform API, e ii) à troca de contexto usando o modo monitor, descrito na Subseção 4.2. A primeira chamada ao TEE ocorre na construção da mensagem de consulta, pois é necessário encriptar os campos privados com a `CC`, conforme descrito na Seção 5. A sobrecarga é igual a 0,53ms, enquanto que a operação de construção da mensagem em si dura apenas 0,39ms. A segunda chamada ao TEE ocorre na decriptação e armazenamento do bloco recebido. Nesse caso, a sobrecarga para chamada do TEE dura 0,70ms, a decriptação dura 0,78ms e o armazenamento dura 50,92ms. Conclui-se que a sobrecarga total do TEE é de 1,23ms, que representa apenas 2% dos 59,3ms do tempo de transferência.

O segundo experimento mede o tempo necessário para a ECU fazer o *download* de 1KB utilizando blocos de diferentes tamanhos. O objetivo é avaliar o impacto da fragmentação em blocos no desempenho. A Figura 4 confirma que o uso de blocos maiores reduz o tempo total de *download*. O tempo necessário para transferir cada bloco é 142,5ms para blocos de 256B, 200ms para 512B e 290ms para 1024B, confirmando que blocos maiores demoram mais tempo para serem transmitidos, decriptados e armazenados. O tempo de inicialização e finalização é fixo em 196ms e, considerando que *softwares* utilizados em ECUs tipicamente possuem alguns *megabytes*, este tempo se torna pequeno comparado com o tempo de transferência dos blocos. As sobrecargas introduzidas pela consulta no servidor, em azul, e pelo TEE na ECU, em vermelho, são desprezíveis. A transmissão de um bloco pequeno pode ser vantajosa quando há um atacante que compromete a integridade de uma transmissão, pois o servidor reenvia apenas um pedaço pequeno do *software* corrompido para a ECU. Vale lembrar que as perdas e corrupções de pacotes são resolvidas pelo TCP. O RTT (*Round-Trip Time*) médio da conexão entre cliente e servidor no momento do experimento, estimado utilizando o utilitário `ping`, é de 0,609ms. Tendo em vista a ausência de implementações semelhantes, esses experimentos representam um avanço na literatura ao demonstrar a viabilidade da computação confiável fim-a-fim para OTA.

8. Trabalhos relacionados

O arcabouço de atualizações OTA mais adotado na indústria é o Uptane, proposto por Karthik *et al.* [Karthik et al. 2016]. O artigo do Uptane conta com a participação

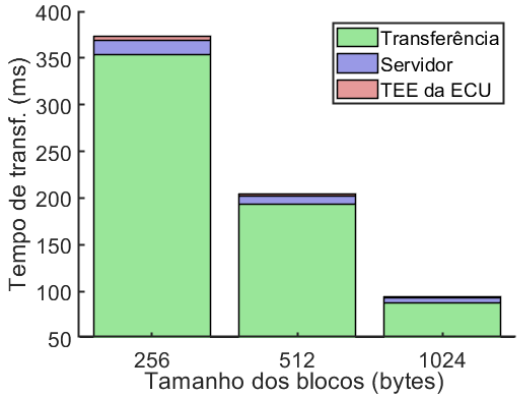

Figura 4. O tempo para transferir 1KB do servidor para a ECU cai com o aumento do tamanho do bloco. As sobrecargas introduzidas pelo TEE e pelo servidor são desprezíveis.

de empresas automotivas importantes e estende as especificações do TUF para resolver problemas de segurança específicos de sistemas veiculares. No Uptane, os roubos de chaves nos servidores podem ser mitigados por revogação, mas não são eliminados, pois a implementação não é resistente a atacantes com alto nível de privilégio. No lado do cliente, a implementação assume que o atacante não consegue controlar arbitrariamente o *firmware* das ECUs primárias e secundárias. Assim, outros trabalhos na literatura expandem esse arcabouço ou propõem arquiteturas alternativas.

Henriques executa a verificação do *software* baixado na ECU primária em um enclave SGX [Henriques 2022]. O serviço de verificação é implementado no veículo usando o Uptane, chamadas RPC (*Remote Procedure Call*) e a plataforma Gramine-SGX para desenvolvimento de enclaves [Tsai et al. 2017]. Os resultados da avaliação de desempenho mostram que a sobrecarga introduzida pelo uso dessas ferramentas em relação ao Uptane padrão é de 91%. A proposta é muito complexa para dispositivos com poucos recursos computacionais. Atualmente, o uso de SGX no veículo não é economicamente viável, pois a tecnologia está disponível apenas em processadores de servidores com alto custo e alto consumo energético.

Mukherjee *et al.* adaptam o Uptane para atualizar os *softwares* de veículos elétricos com bateria no mundo seguro do TrustZone [Mukherjee et al. 2021]. Os metadados e a imagem são transferidos para uma aplicação confiável do OP-TEE rodando em um Raspberry Pi 3, por meio da interface UART e do protocolo UDS (*Unified Diagnosis Service*). O trabalho utiliza a ferramenta SAW (*Software Analysis Workbench*) para verificar vulnerabilidades no código usando modelos semânticos, mas não apresenta resultados de desempenho [Carter et al. 2013]. Além disso, o modelo de atacante assume que os servidores de data e hora, repositório de imagens e diretor são seguros.

Kornaros *et al.* usam o TrustZone para proteger aplicações críticas de uma motocicleta, como o controlador de tração, o comando de aceleração (*drive-by-wire*) e o sistema de atualização OTA da interface gráfica do painel [Kornaros et al. 2020]. O artigo também estende a segurança do protocolo CAN e implementa em uma FPGA (*Field Programmable Gate Array*) um *firewall on-chip* para proteção contra ataques físicos à memória. Os autores reproduzem o fluxo completo de atualização de *software* em uma moto na presença de um atacante, utilizando microcontroladores com o TrustZone-M. O gerenciamento do *software* no servidor e a transferência para o veículo estão fora do escopo. Assim, todas as medidas de desempenho e funcionalidades de segurança se limitam aos CIs do veículo e à rede local.

Diferente dos trabalhos anteriores, este trabalho propõe uma arquitetura que utiliza TEE para prover segurança fim-a-fim na atualização de *softwares* de veículos por OTA. O sistema é compatível com as características desejáveis dos sistemas comerciais de OTA veicular. Ao mesmo tempo, ele é resiliente a atacantes privilegiados nos servidores e no veículo. A avaliação de desempenho revela que a sobrecarga adicionada pelos procedimentos propostos é baixa.

9. Conclusões

Este trabalho propôs uma arquitetura que combina computação confiável na nuvem e em ECUs para atualização OTA de *softwares* veiculares. A implementação utilizando um servidor com Intel SGX e um dispositivo embarcado com TrustZone é mais

segura que as encontradas em outros trabalhos, pois protege contra atacantes que controlam os sistemas operacionais no cliente e no servidor. A avaliação desempenho também se diferencia, pois revela a sobrecarga introduzida por cada etapa do processamento, demonstrando que a latência introduzida é pouco significativa. Como trabalhos futuros é prevista a implementação de um módulo para atualizar o *firmware* da ECU primária, bem como transferir *softwares* recebidos pela interface sem fio para ECUs secundárias na rede veicular local. Outras direções de pesquisa que colaboram para o aprimoramento do sistema proposto são: i) a comparação de diferentes implementações de TrustZone em microcontroladores e SoCs e ii) a avaliação de diferentes ferramentas de desenvolvimento com SGX no servidor.

Agradecimentos

O presente trabalho foi realizado com o apoio do CNPq, da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES), código de financiamento 001, da FAPERJ (E-26/211.144/2019), da FAPESP (2015/24494-8) e da Fundação de Desenvolvimento da Pesquisa - Fundep - Rota 2030.

Referências

Anati, I., Gueron, S., Johnson, S., and Scarlata, V. (2013). Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13. ACM New York, NY, USA.

Carter, K., Foltzer, A., Hendrix, J., Huffman, B., and Tomb, A. (2013). SAW: the software analysis workbench. In *Proceedings of the 2013 ACM SIGAda annual conference on High integrity language technology*, pages 15–18.

Costan, V. and Devadas, S. (2016). Intel SGX explained. *Cryptology ePrint Archive*.

De Souza, L. A. C., Camilo, G. F., M. Campista, M. E., M. K. Costa, L. H., and M. B. Duarte, O. C. (2022). Enhancing Automatic Attack Detection through Spectral Decomposition of Network Flows. In *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*, pages 2074–2079.

Göttel, C., Felber, P., and Schiavoni, V. (2019). Developing secure services for IoT with OP-TEE: a first look at performance and usability. In *Distributed Applications and Interoperable Systems: 19th IFIP WG 6.1 International Conference, DAIS 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17–21, 2019, Proceedings 19*, pages 170–178.

Henriques, A. C. P. (2022). Secure Over-the-Air Vehicle Updates using Trusted Execution Environments (TEE). Master's thesis, Universidade do Porto.

Hern, A. (2013). North Korean 'Cyberwarfare' Said to Have Cost South Korea £500m. *The Guardian*, 16.

Karthik, T., Brown, A., Awwad, S., McCoy, D., Bielawski, R., Mott, C., Lauzon, S., Weimerskirch, A., and Cappos, J. (2016). Uptane: Securing software updates for automobiles. In *International Conference on Embedded Security in Car*, pages 1–11.

Kornaros, G., Tomoutzoglou, O., Mbakoyiannis, D., Karadimitriou, N., Coppola, M., Montanari, E., Deligiannis, I., and Gherardi, G. (2020). Towards holistic secure networking in connected vehicles through securing CAN-bus communication and firmware-over-the-air updating. *Journal of Systems Architecture*, 109:101761.

Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., et al. (2010). Experimental security analysis of a modern automobile. In *2010 IEEE symposium on security and privacy*, pages 447–462. IEEE.

Mukherjee, A., Gerdes, R., and Chantem, T. (2021). Trusted Verification of Over-the-Air (OTA) Secure Software Updates on COTS Embedded Systems. In *Workshop on Automotive and Autonomous Vehicle Security (AutoSec)*, volume 2021, page 25.

Ngabonziza, B., Martin, D., Bailey, A., Cho, H., and Martin, S. (2016). Trustzone explained: Architectural features and use cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pages 445–451. IEEE.

Oleksenko, O., Trach, B., Krahn, R., Silberstein, M., and Fetzer, C. (2018). Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *2018 Usenix Annual Technical Conference (USENIX ATC 18)*, pages 227–240.

Sarkar, S., Choudhary, G., Shandilya, S. K., Hussain, A., and Kim, H. (2022). Security of zero trust networks in cloud computing: A comparative review. *Sustainability*, 14(18):11213.

Scarlata, V., Johnson, S., Beaney, J., and Zmijewski, P. (2018). Supporting third party attestation for Intel SGX with Intel data center attestation primitives. *White paper*.

Suzaki, K., Nakajima, K., Oi, T., and Tsukamoto, A. (2020). Library implementation and performance analysis of GlobalPlatform TEE Internal API for Intel SGX and RISC-V Keystone. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1200–1208. IEEE.

Thomaz, G. A., Guerra, M. B., Sammarco, M., and Campista, M. E. M. (2023a). CACIC-DevKit: Construção de Sistemas IoT com Políticas de Acesso Customizáveis e Segurança por Hardware. In *Anais Estendidos do XLI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, pages 1–8. SBC.

Thomaz, G. A., Guerra, M. B., Sammarco, M., Detyniecki, M., and Campista, M. E. M. (2023b). Tamper-proof access control for IoT clouds using enclaves. *Ad Hoc Networks*, 147:103191.

Tsai, C.-C., Porter, D. E., and Vij, M. (2017). Graphene-SGX: A Practical Library {OS} for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658.