Stealthy Tracking of Autonomous Vehicles with Cache Side Channels

Mulong Luo
Cornell University
ml2558@cornell.edu

Andrew C. Myers
Cornell University
andru@cs.cornell.edu

G. Edward Suh
Cornell University
suh@ece.cornell.edu

Abstract

Autonomous vehicles are becoming increasingly popular, but their reliance on computer systems to sense and operate in the physical world introduces new security risks. In this paper, we show that the location privacy of an autonomous vehicle may be compromised by software side-channel attacks if localization software shares a hardware platform with an attack program. In particular, we demonstrate that a cache side-channel attack can be used to infer the route or the location of a vehicle that runs the adaptive Monte-Carlo localization (AMCL) algorithm. The main contributions of the paper are as follows. First, we show that adaptive behaviors of perception and control algorithms may introduce new side-channel vulnerabilities that reveal the physical properties of a vehicle or its environment. Second, we introduce statistical learning models that infer the AMCL algorithm's state from cache access patterns and predict the route or the location of a vehicle from the trace of the AMCL state. Third, we implement and demonstrate the attack on a realistic software stack using real-world sensor data recorded on city roads. Our findings suggest that autonomous driving software needs strong timing-channel protection for location privacy.

1 Introduction

Recent years have seen significant efforts to develop autonomous vehicles. Autonomous unmanned aerial vehicles (UAVs) have already been used in some cases for commercial parcel delivery [21]. Today's passenger vehicles include many advanced driver assistance features, and future vehicles are expected to have even more autonomous driving capabilities. For example, Tesla vehicles include the Autopilot [14] system, which enables autonomous cruise on freeways. Uber [15] and Waymo [18] are testing commercial taxicab services using fully autonomous vehicles. While autonomous vehicles can enable many exciting applications, they also introduce new security risks by allowing a computing system to sense and control the physical system.

In this paper, we show that the location privacy of an autonomous vehicle may be compromised by software side-channel attacks when the vehicle's driving software and the attack software share a hardware platform. In particular, we demonstrate that a cache side-channel attack can be used to infer the route/location of a vehicle that uses the adaptive Monte-Carlo localization (AMCL) algorithm [35] for localization. Previous studies on traditional computer systems have demonstrated many cache side-channel attacks for inferring confidential information, so it is not surprising to find cache side channels in the computing platforms of autonomous vehicles. What is novel and interesting about our attack is that the cache side channel can be used to infer a victim vehicle's physical state, exploiting the correlation between the physical state of the vehicle and the cache access patterns of the vehicle's control software. Moreover, our experimental results show that this information leak is sufficient to identify the vehicle's route from a set of routes in the known environment, and even the location of a vehicle if an attacker knows the vehicle's initial location.

In autonomous vehicles, perception and control algorithms are often adaptive in order to improve their efficiency and accuracy. The adaptive algorithms perform more computation when there is more uncertainty in the environment or an event that affects the vehicle's state, such as a new obstacle showing up or the vehicle making a turn; conversely, they perform less computation when there is no significant change. These adaptive behaviors are natural and important for efficiency. However, they also create strong correlation between the algorithm's memory access patterns and a vehicle's physical movement and environment. For example, we found that the amount of data accessed by the AMCL algorithm, commonly used for localization, reveals when the algorithm's uncertainty on the vehicle's location changes. This correlation allows our cache side-channel attack to infer when a vehicle is turning.

While the observation that the AMCL algorithm's cache behavior is strongly correlated to a vehicle's physical state is interesting by itself, we found that cache side-channel attacks on an autonomous vehicle's control software introduce new challenges that do not exist in traditional cache side-channel attacks. Unlike cryptograhic keys in memory, the physical state of a vehicle changes continuously as the vehicle moves. Work on inferring AES keys via cache side channels has aggregated results from multiple measurements [55]. However, it is difficult to measure the fast-changing physical state of a vehicle multiple times using a cache side channel. Moreover, physical environments are inherently noisy. As a result, cache timing measurements are affected not only by noise in the computing system but also by physical noise.

In this paper, we address these challenges and demonstrate an end-to-end cache side-channel attack on the location privacy of an autonomous vehicle. Specifically, we demonstrate that an unprivileged user-space program, without access to sensor inputs or protected state of control software, can predict the route or the location of an autonomous vehicle using a prime-and-probe cache timing channel attack on the control software. Our attacks differ from many previous cache side channel attack in that we use timing measurements over a period of time when a vehicle is moving. We introduce a statistical learning model based on random forests to predict the route or the location of a vehicle from cache timing measurements while dealing with noise. The experimental results based on both a simulated robot and recorded data from a real-world vehicle show that this attack can fairly accurately predict the vehicle's route or location.

Our results show that the location privacy of an autonomous vehicle can be compromised when its perception and control software share hardware resources with less trusted software. Without new processor designs that provide strong isolation guarantees regarding timing channels, our findings suggest that separate platforms should be used for autonomous driving software and the rest of the system.

The following summarizes the main contributions of the paper:

- We show that the adaptive behaviors of perception and control algorithms may introduce a new security vulnerability that reveals the physical properties of a vehicle or its environment through side channels.
- We introduce statistical-learning models that predict the AMCL algorithm's state from its cache access patterns, and infer the route or the location of a vehicle from the trace of the predicted AMCL state.
- We implement and demonstrate the attack on a realistic software stack using both simulated environments and real-world sensor data recorded from a vehicle.

The rest of paper is organized as follows. Section 2 discusses the threat model. Section 3 discusses the background on autonomous vehicles and cache side channels. Section 4 describes the attack implementation. Section 5 describes our testbeds and evaluates the attack's effectiveness. Section 6 discusses the implications of the attack, and Section 7 reviews related work. Finally, we conclude the paper in Section 8.

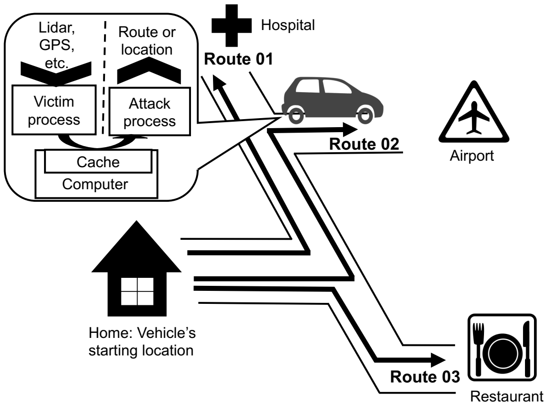


Figure 1: The threat model. The attack software runs on the same processor with the autonomous-driving software, and learns the route of the vehicle through cache side channels.

2 Threat Model

The goal of the attacker is to infer the location information of a vehicle based on cache side channels. In particular, the attacker predicts the route that an autonomous vehicle takes from a set of known routes.

Figure 1 illustrates the threat model discussed in this paper. While the figure shows a passenger vehicle as an example, we note that the proposed attack method and principle may be applied to other autonomous vehicles such as delivery robots or drones. We assume that the attacker is an entity that can deploy a software module on the vehicle. We refer to the software module as "attack software" or "attack process". In this paper, we use process, program, and software interchangeably. The victim is an autonomous vehicle (the "victim vehicle") whose route information needs to be protected. Localization software on the victim vehicle (the "victim software" or "victim process") has direct access to sensors and to its location-related information, and is the target of our cache-side channel attack. The attacker has no physical access to the victim vehicle, and performs its attack only through the attack software. We assume that the attack software cannot circumvent the access controls of the operating system and has no direct access to the location information.

Assumptions on the attacker. We assume that the attacker knows details of the victim vehicle including the software and hardware configuration of its computing platform as well as the mechanical system. We also assume that an attacker has detailed knowledge of the environment in which the victim vehicle operates and knows a set of routes that the victim may take. For example, the attacker should have the map of the victim's environment, and may use another vehicle to collect detailed sensor measurements of the area in order to train its prediction models. The aim of the attack is to infer the victim vehicle's route or location in a known environment, rather than to track the victim vehicle in an unknown environment.

To make cache side-channel attacks possible, we assume that attack software can run on the same processor where victim software runs. This co-location may be achieved by compromising less safety-critical software components that are already on the victim or via untrusted applications that is allowed to be installed. The attack software is also assumed to be able to send the vehicle's location information to a remote attacker once it acquires the information. On the other hand, we assume that the operating system securely prevents the attack software from directly reading sensors or the location.

Assumptions on the victim. We consider an autonomous vehicle that is controlled by an onboard computer. We assume that the autonomous-driving software uses an adaptive algorithm, such as adaptive Monte-Carlo localization (AMCL) [35] for localization or Faster R-CNN [59] for object detection, whose compute requirements change depending on the vehicle's movements or environments. Our attack exploits the fact that memory access patterns of these adaptive algorithms are affected by the victim vehicle's movements.

Assumptions on the environment. We assume that the environment has unique characteristics that enable identification of the vehicle's position and route. Analogously, humans can localize themselves in a known city using visual details such as buildings or signage. Our work exploits variability in possible vehicle paths to guess the route of the vehicle from the turns it takes.

Out-of-scope attacks. We do not consider any physical attacks on a vehicle. As we assume that the attack software does not have permission to access sensor data, we do not consider any attacks that rely on direct access to the physical measurements of an environment [48, 49] (e.g., inferring locations based on local temperature, light intensity, etc.). Besides, we do not consider traditional attacks that exploit software vulnerabilities to compromise an operating system or the driving software itself. We assume that the driving software is not malicious or compromised, and do not consider covert-channel attacks where the driving software intentionally leaks the vehicle location.

3 Background

3.1 Autonomous Vehicle Architecture

Autonomous vehicles perform tasks in the physical world without human intervention. As shown in Figure 2, an autonomous vehicle comprises three main hardware subsystems: sensors/information collectors, an onboard computer, and actuators/command executors. Sensors are used to collect information from the physical world. The collected data are then processed by the onboard computer, which generates actuation commands. The actuation commands are executed by the actuators, which usually have observable and intentional effects on the physical world, such as turning the steering wheel of the vehicle. Both sensors and actuators are connected to the onboard computer using a bus protocol such as USB, PCIe, GPIO, or CAN bus [31].

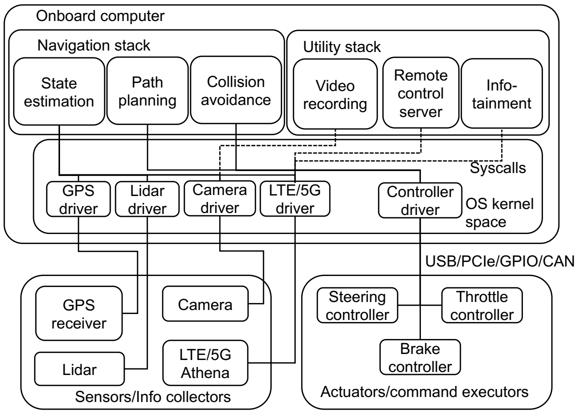


Figure 2: General hardware and software architecture of an autonomous vehicle.

The navigation software stack hosted on the onboard computer reads preprocessed sensor data from device drivers and writes commands to the controller driver. There are two major tasks performed by the navigation software:

- **Perception/estimation**. This is the process of converting the sensor data (e.g., timestamps returned by a GPS receiver) into the most likely physical state (e.g., location on the earth). This is needed for two reasons. First, sensor data contain noise from measurements. Thus, an estimation algorithm is needed to remove the noise and get a statistically sound state. Second, the actual physical state (e.g., location of a vehicle on a map) cannot be directly measured from sensors (e.g., LiDAR signal, which is a vector of distances to obstacles in its scanning directions). An estimation algorithm (e.g., adaptive Monte-Carlo localization [34]) infers the most probable location based on the LiDAR data.

- **Control/decision**. This is the process of determining a sequence of control commands that optimize a certain objective function (expected arrival time, distance to travel, etc.) given the estimated state. For example, given an estimation of the current location and the final destination on a map, the controller should determine a trajectory to the destination and issue a sequence of acceleration, stop, and steering commands so that the vehicle follows the planned path.

As shown in Figure 2, the state estimation module in the navigation stack needs to read data from sensors such as GPS, LiDAR, camera, and LTE/5G to make correct state estimations. Estimated state, such as the vehicle location, is used by the path planning module, which makes decisions on which trajectory to take and sends commands to the controller. There is also a collision avoidance module, which can override the commands to the controller when there is a safety issue.

There is also a utility software stack, which performs vehicle-specific tasks that are not critical to safety. For example, a passenger vehicle may have an infotainment system providing a music streaming service, while an autonomous video-recording drone may have software to control a high-resolution camera. Because the utility stack is not safety-critical, it should not have unnecessary access to sensors or actuators. For example, a music streaming app may require access to the LTE/5G network to download music, but should not be able to access or record GPS data. This can be enforced by OS-level access-control mechanisms.

3.2 Adaptive Monte-Carlo Localization

Localization is a task that determines the locations of objects on a given map based on sensor inputs. It is needed by many advanced driving assistance systems and required by autonomous vehicles. Adpative Monte-Carlo Localization (AMCL) is a special case of general MCL [34], and was used by multiple teams [26, 43, 50] in the DARPA Grand challenge [25]. Many recent research autonomous driving projects [27, 40, 60, 64] have also used AMCL. For example, the CaRINA intelligent robotic car [32] uses AMCL for its LiDAR-based location [40].

Algorithm 1 shows the pseudocode for general Monte-Carlo localization. Given a map M_0 of a certain area and a probability distribution $P : M_0 \mapsto \mathbb{R}$ over the map M_0, at time t, N particles (i.e., hypothetical locations of the vehicle) are randomly generated based on the distribution. For each particle L_i, the sensor measurement S_t is combined with the particle to infer the position of the obstacles on the map. For example, in a 1-D case, if the distance sensor detects an obstacle 10 m from the hypothetical location of the vehicle and the hypothetical location is 20 m from the starting location, it is inferred that that obstacle is 30 m (10 m + 20 m) from the starting location. Inferred obstacles are plotted on a new empty map M_i, which is then compared with the given map M_0 to calculate the fidelity p_i of the particle L_i, based on the assumed distribution of measurement errors. For example, the fidelity p_i will be high if the inferred map M_i closely matches the given map M_0, and low if the two maps differ significantly. Finally, k-means clustering [36] is used to determine the most probable geometrical clustering center $L_{est,t}$ of these particles $\{L_i\}$, weighted by $\{p_i\}$ at time t. Also, the probability distribution $P : M_0 \mapsto \mathbb{R}$ is updated for the next measurement S_{t+1}.

The number of particles N in Algorithm 1 is not necessarily fixed. When the distribution $P : M_0 \mapsto \mathbb{R}$ converges, a small N is enough for accurate estimation. When the distribution $P : M_0 \mapsto \mathbb{R}$ spreads across the map M_0, the parameter N may need to be increased. In AMCL, N changes with time t; we denote it by N_t. The exact value of N_t at time t is determined by the Kullback–Leibler distance (KLD) [34] between the estimated distribution $P : M_0 \mapsto \mathbb{R}$ and the underlying ground-truth distribution $P_0 : M_0 \mapsto \mathbb{R}$:

$$N_t = \frac{k-1}{2\varepsilon}\{1 - \frac{2}{9(k-1)} + \sqrt{\frac{2}{9(k-1)}} z_{1-\delta}\}^3 \qquad (1)$$

Here, $z_{1-\delta}$ is the upper $(1-\delta)$ quantile of standard normal distribution, ε is the upper bound of the KLD, and k is the number of bins occupied during sampling at time t (e.g., if the map is partitioned into 1,024 bins and only 300 bins are occupied, in this case, $k = 300$). Theoretically, N_t could be any positive integer. Practically, there is a maximum limit N_{max} and a minimum limit N_{min} to ensure real-time performance and k-means clustering accuracy, respectively. In our experiments, we found that the AMCL implementation uses either the maximum or the minimum number of particles in most cases.

```
Input: Map $M_0$, a probability distribution over the whole
       map $P : M_0 \mapsto \mathbb{R}$ , sensor measurement time
       series $S_1, S_2, ...S_t$, number of particles $N$, number
       of clusters $K$, transient odometry $d_1, d_2, ...d_t$.
Result: Estimated states $L_{est,1}, L_{est,2}, ..., L_{est,t}$ on map
foreach sensor measurement $S_t$ at time $t$ do
    Randomly generate $N$ particles (i.e., hypothetical
      locations) $\{L_i\}$ on the map based on distribution
      $P : M_0 \mapsto \mathbb{R}$;
    foreach particle $L_i$ ($1 \leq i \leq N$) do
        Overlay measurement $S_t$ on the particles $L_i$;
        Generate the extrapolated map $M_i$ based on the
          measurement $S_t$ and location $L_i$;
        Compare the extrapolated map $M_i$ and the given
          map $M_0$, calculate the fidelity $p_i$;
    end
    Determine the most probable cluster center
      $L_{est,t} = kmeans(K; L_1, \ldots, L_N; p_1, \ldots, p_N)$;
    Update the probability distribution $P : M_0 \mapsto \mathbb{R}$ based
      on particles $L_1, \ldots, L_N$, corresponding fidelity
      $p_1, \ldots, p_N$ as well as transient velocity $d_t$;
end
```

Algorithm 1: General Monte-Carlo localization.

3.3 Cache Side Channel

In modern computing systems, off-chip memory (e.g., DRAM) accesses are much slower than on-chip memory accesses served by a cache. Also, a cache is usually shared among multiple programs. For example, a last-level cache (LLC) in a multi-core processor is used by multiple processing cores concurrently. L1 and L2 caches may be dedicated to a specific core, but are still time-shared among programs that run on the core.

The shared cache implies that one program's memory accesses can affect whether another program can find its data in the cache, or needs to access off-chip memory. As a result, one program can infer another program's memory accesses by measuring its own memory access latency. When a victim program accesses its data from memory, it can evict the cached data of other programs in order to bring its own data into cache. An attack program can infer whether the victim program had a cache miss or not, and which memory address was accessed, by measuring the latency of its memory access, which reveals whether the data was found in the cache or not. This measured latency leaks the victim program's memory-access pattern to the attack program. There are many existing cache side-channel attack techniques, including prime+probe [45, 54], evict+time [55], flush+reload [44, 71], prime+abort [28], flush+flush [37], etc. In this work, we use the prime+probe attack, but we expect that our attack can also be implemented using other types of cache side-channel attacks.

4 The Proposed Attack

4.1 Vulnerability in AMCL

An autonomous vehicle running AMCL is vulnerable to a cache side-channel attack that aims to infer its kinematics. This is because the memory access pattern of AMCL depends on the number of particles N_t at each time t, which has strong correlation with the real-time vehicle kinematics.

First, the number of particles N_t affects the memory access pattern of AMCL, which can be inferred through a cache side-channel attack. The following steps summarize how the memory accesses in AMCL for an iteration at time t are determined, based on Algorithm 1 and a reference implementation in ROS [1].

1. Calculate the number of particles N_t using Equation (1);
2. Create N_t particle objects in a fixed-size buffer[1];
3. For each particle, access the memory locations of the particle object and perform necessary computation.

If N_t increases, more memory locations will be accessed. The memory accesses can be observed by another program through a cache side channel.

Second, the number of particles N_t has a strong correlation with the vehicle kinematics at time t. It is obvious from Equation (1) that N_t increases with k, which represents the number of bins occupied by particles. The value of k depends on the level of uncertainty in the estimation. As shown in a previous study [35], when the observed environment is unstable (e.g., due to signal loss), N_t increases to compensate for the increased estimation uncertainty. Our observation is that N_t increases when the vehicle is turning as shown in Figure 3.

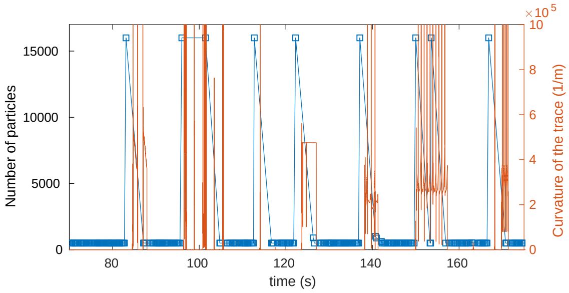


Figure 3: An example showing the correlation between the number of particles in AMCL and the vehicle trajectory curvature (high curvature indicates the vehicle is turning). Obtained from a Jackal robot simulation.

Third, the route or the position of a vehicle can be inferred from kinematic information. In theory, if the curvature $\kappa(t)$ of the vehicle's trajectory as a function of time t is obtained using the side channel, we can obtain the route that a vehicle is taking by matching the curvature of the trajectory with the candidate routes on the map. In addition, if we know the initial location of the vehicle, we can predict the location of the vehicle by enumerating routes that connect the initial location and the candidate locations on the map.

In practice, instead of using curvature, whose precise value is hard to directly infer, we use the information on the number of particles to predict the route of the vehicle.

4.2 Attack Overview

Based on the vulnerability described in Section 4.1, it is possible to implement a cache side-channel attack that infers the route or the location of an autonomous vehicle running AMCL. We implement our attack using the following steps.

1. **Prime+Probe**: Collect the cache probing time for each cache set over fixed time intervals, forming a sequence of *cache-timing vectors* in which each vector represents the probing times for cache sets at a specific time interval.
2. **Particle Predictor**: Use a binary classification model to predict the number of particles for each time interval based on the cache-timing vectors for each interval.
3. **Route Predictor**: Use a random forest model to predict the route or the position of a vehicle based on the trace of the number of particles.

Figure 4 shows the overall flow of the attack. We describe each step in more detail in the next three subsections.

[1] Original ROS AMCL implementation dynamically allocates and frees memory space for N_t particles in each iteration rather than using a fixed-size buffer. Instead, we use a statically-allocated buffer to avoid unnecessary overhead for dynamic memory allocation. While not included in the paper, we also tested our attack with the dynamic memory allocation, and confirmed that the attack works for both static and dynamic allocation.

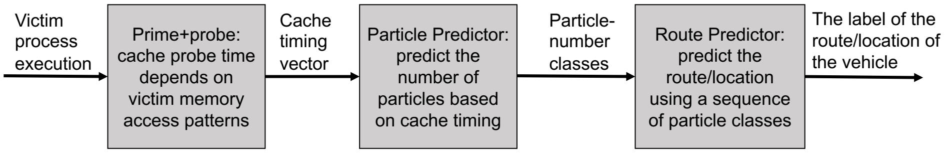

Victim process execution
Prime+probe: cache probe time depends on victim memory access patterns
Cache timing vector
Particle Predictor: predict the number of particles based on cache timing
Particle-number classes
Route Predictor: predict the route/location using a sequence of particle classes
The label of the route/location of the vehicle

Figure 4: The overall flow of the attack.

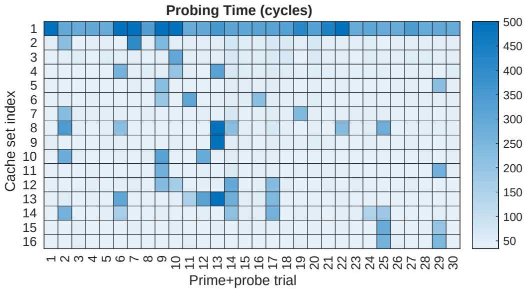


Figure 5: Cache side-channel measurements of 16 cache sets from the L1D cache of Intel i5-3317u.

4.3 Acquiring Victim Cache Access Pattern

In this work, we use a prime+probe attack to infer the memory accesses of victim software. First, the attack program fills the cache with its own data by sequentially accessing a set of memory addresses. Then, the victim accesses the cache. After that, the attack program probes the same memory addresses and records the latency of each access. If a specific memory address is evicted by the victim program, the probe time will be longer. Thus, the memory access pattern of the victim program can be inferred.

The result of the prime+probe attack is a sequence $\{\mathbf{T}_t\}$ in which each element $\mathbf{T}_t$ at time t is a K-dimension vector $(\tau_t^1, \tau_t^2, ..., \tau_t^K)$ where K is the number of cache sets. For example, in Figure 5, each column is a 16-D vector representing the probing time of 16 sets in the L1 data (L1D) cache. The result is from an Intel i5-3317u dual-core processor whose L1D cache of one core has 64 sets total. For brevity, we show only 16 sets out of 64.

Many cache side-channel attacks exist. For example, the evict+time attack [55] has been used to extract cryptographic keys on a system when many measurements can be made using the same key. The flush+reload attack [44] has been used when shared memory locations, such as a shared library, can be accessed by both attacker and victim software. We use the prime+probe attack because it can effectively infer the victim's memory accesses even without multiple measurements and without a shared library between the attacker and the victim.

4.4 Particle Predictor

In practice, we found that the AMCL algorithm usually uses either the maximum or the minimum number of particles. Given this observation, we formulate the prediction of the number of particles as a binary classification problem.

The input of the model is the vectors from the prime+probe cache attack $\{\mathbf{T}_t\}$. We take a time window of size $2T+1$ of $\mathbf{T}_t$, i.e., $(\mathbf{T}_{t-T}, ..., \mathbf{T}_t, ...\mathbf{T}_{t+T})$ as the input of the model, and the output particle-number class N_t is in one of the two classes, i.e., $N_t \in \{L, H\}$, where L and H denote "Low" and "High", respectively. Formally, the classification task is defined as follows:

- **Given**: t_{end} tuples of $(\mathbb{T}_t, N_t)$ $(t \in \{1, 2, ..., t_{end}\})$, where $\mathbb{T}_t$ is a $(2T+1) \cdot K$-dimension vector $\mathbb{T}_t = (\tau_{t-T}^1, ..., \tau_{t-T}^K, ..., \tau_{t+T}^1, ..., \tau_{t+T}^K)$ for each t, and $N_t \in \{L, H\}$ for each t.
- **Find**: a model $f : \mathbb{R}^{(2T+1)\cdot K} \mapsto \{L, H\}$ such that the classification score $\sum_{t=1}^{t_{end}} d(f(\mathbb{T}), N_t)$ is maximized, where $d : \{L, H\} \times \{L, H\} \mapsto \mathbb{R}$ is defined as follows:

$$d(N_1, N_2) = \begin{cases} 1, & \text{if } N_1 = N_2. \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

We observe that the two classes are unbalanced, i.e., the number of samples in the "High" class is much smaller than the number of samples in the "Low" class. This is because when a vehicle is moving on a map with predefined roads, for most of the time, it is moving straight and the trajectory curvature is small. Due to the correlation between the number of particles and the curvature, as mentioned in Section 4.1, more samples in the "Low" particles-number class are seen. Traditional binary classifiers such as SVM [36] do not perform well on such unbalanced datasets. To address the problem, we use RUSBoost [62], a classification algorithm designed to alleviate class imbalance in the dataset. RUSBoost combines both random undersampling (RUS) and boosting to improve classification accuracy.

Figure 6 shows an example of the prediction of the number of particles in AMCL (max/min number of particles 16,000/500) using RUSBoost on the cache timing channel information collected from the L1D cache of an Intel processor. The model correctly predicted the timing of events where there exists a spike in the number of particles. To evaluate prediction quality, we use Dynamic Time Warping (DTW) [61], a popular metric for measuring similarity of two temporal sequences. DTW allows us to compare two sequences even when the exact locations of spikes are slightly off. The DTW distance between the predicted and the ground truth is 539,407

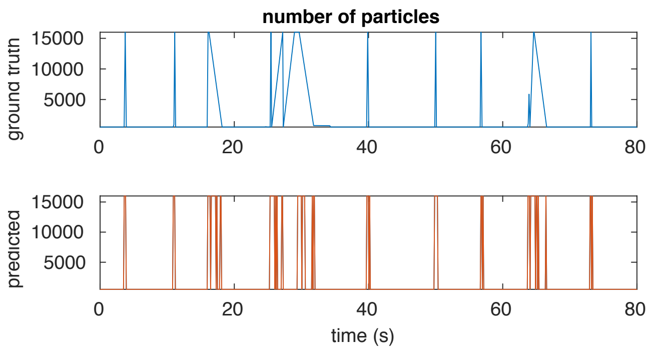


Figure 6: Ground truth and predicted number of particles using RUSBoost on AMCL running on Intel i5-3317u.

Method	Train	2-fold	5-fold
RUSBoost	536,013	514,656	510,006
SVM	150,890	543,716	547,580

Table 1: Comparison of the average DTW distance between RUSBoost and SVM.

particles. Considering that one false prediction point incurs a distance of 16,000−500=15,500, the DTW distance implies $539,407 \div 15,500 \approx 35$ false prediction points in a single trace containing about 1,000 data points.

We compare SVM and RUSBoost prediction results in Table 1. In the table, we list average DTW distance of training, 2-fold, and 5-fold validation[2]. We use 100 traces for each experiment. The results show that even though SVM has low training DTW distance, RUSBoost has lower 2-fold and 5-fold validation distance, indicating RUSBoost model performs better and overfits less for this modeling task.

4.5 Route Predictor

Given a sequence of the particle classes $(N_1, N_2, N_3, ..., N_t, ..., N_{t_{end}})$ we need a model that predicts the route or the location of the vehicle. There are two related tasks:

1. **Route prediction**: Given a set of known routes, find the route that a vehicle takes.

2. **Location prediction**: Given the starting location of a vehicle and a set of possible final locations on a known map, determine the final location of the vehicle.

The task of predicting the final location can be considered a specific form of route prediction, in which the set of known routes contains all routes on the map that connect the starting location and possible final locations. In that sense, both the route prediction and location prediction tasks can be formulated in a unified way.

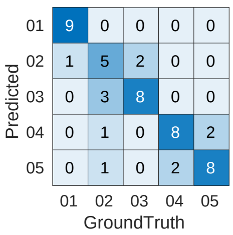


Figure 7: kNN classification results.

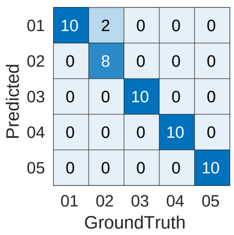


Figure 8: RF-50 classification results.

Different routes may not necessarily have the same length t_{end}, and for the same route, t_{end} may vary based on the speed of the vehicle. To handle the variations in the trace length, we pad each sequence $\mathbf{N} = (N_1, N_2, N_3, ..., N_t, ..., N_{t_{end}})$ into a sequence $(N_1, N_2, N_3, ..., N_t, ..., N_{t_{end}}, ..., N_{t_{t_{max}}})$ with length t_{max} by assigning a new element $P \in \{L, P, H\}$ (for padding) to all N_t for $t_{end} < t \leq t_{max}$. After that, we can formulate the prediction as a standard classification problem:

- **Given:** M tuples $(\mathbf{N}_i, \mathbf{l}_i)$ in which $1 \leq i \leq M$ and $\mathbf{N}_i \in \{L, P, H\}^{t_{max}}$ is a vector of maximum length t_{max} and $\mathbf{l}_i \in \{\mathbf{l_1}, \mathbf{l_2}, ..., \mathbf{l_n}\}$ is the label representing a route or a location.

- **Find:** $g : \{L, P, H\}^{t_{max}} \mapsto \{\mathbf{l_1}, \mathbf{l_2}, ..., \mathbf{l_n}\}$ such that $\sum_{i=1}^{M} c(g(\mathbf{N_i}), \mathbf{l}_i)$ is maximized. Here the cost function is defined as follows:

$$c(\mathbf{l}_1, \mathbf{l}_2) = \begin{cases} 1, & \text{if } \mathbf{l}_1 = \mathbf{l}_2. \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

4.5.1 Predicting Route

We can identify a route by comparing the sequence of particle-number classes ("Low" or "High") along the route. In this case, the label $\mathbf{l}_i$ represents a distinctive route i.

We can use a classification algorithm, e.g., k-nearest neighbor (kNN) or random forest (RF) [36] to classify different routes. For example, Figure 7 and Figure 8 show an example of classification results using kNN and RF with 50 trees (RF-50) for five distinct routes in Maze 1 in Figure 14. This experiment uses a Jackal robot described in Section 5.1. For each sequence of particle-number classes, we use all other sequences as the training set and find the route label for the sequence. The overall accuracy is 76% and 96%, respectively. Given its higher accuracy, we use the random forest (RF) as the route-prediction model.

4.5.2 Predicting Location

If an attacker knows the initial location of a vehicle, our route prediction approach can be used to predict the final location

[2] For evaluating a machine-learning model on a dataset, N-fold validation divides the dataset into N sets. For each test, it uses all but one set to train the model while holding out the one set for validation.

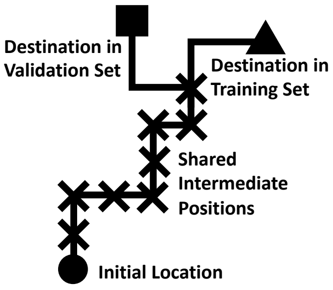

Destination in Validation Set
Destination in Training Set
Shared Intermediate Positions
Initial Location

Figure 9: Though the destination of a run in the validation set might not appear in the training set, the intermediate locations along the path are shared.

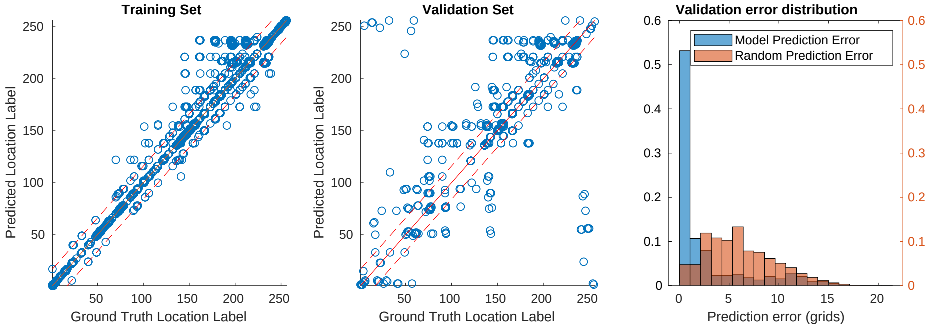


Figure 10: Training, validation accuracy, and validation-error distribution of location prediction for a dataset of 3,633 samples. For this experiment, the measured (ground-truth) sequence of the particle-number classes is used as an input.

of the vehicle from a particle-number class sequence. In this case, the label $\mathbf{l}_i$ represents the final location. For example, we can partition a map into $Q_x \times Q_y$ grid cells and assign each cell (q_x, q_y), where $1 \leq q_x \leq Q_x$ and $1 \leq q_y \leq Q_y$, a unique integer label $\mathbf{l_i} = (q_y - 1) \cdot Q_x + q_x$.

Usually, if an autonomous vehicle starts from a fixed starting location and takes the shortest path to each destination, the paths will form a *shortest-path tree* [53] on a given road network graph. We also use the RF model for this modeling task because in addition to its general pattern-matching capability, it also captures the tree structure of the shortest-path tree.

In practice, the total number of possible destinations ($Q_x \times Q_y$) can be quite large, and collecting sufficient training (and validation) data from multiple runs to all possible destinations can be difficult. Instead, in our experiments, we model an attacker who collects data for a subset of possible destinations; we randomly select a subset of destinations for the training runs and the validation runs separately, and include intermediate locations to create a larger training and validation sets. For each run with a randomly-chosen destination, the intermediate points along the path as well as the final destination are used as *target locations* for samples in the training and validation sets. The runs in the training set and the validation set do not necessarily share the same destination. The model will not be able to predict the target locations in validation samples that never appear in the training samples. However, as Figure 9 shows, the intermediate positions along paths with different destinations may overlap, and the model will be able to correctly predict the samples that use these intermediate positions as their target locations even though the final destinations of the runs are different. [3]

Figure 10 shows an example of the training and validation accuracy of an RF-50 model, which predicts a location label based on a sequence of the particle-number classes. The maze is partitioned into a 16-by-16 grid. The experiment is performed using a dataset in which we collected 3,633 samples based on 100 simulation runs in Maze 1 shown in Figure 14, where the starting location of the vehicle is in the center of the maze. We use the samples collected from 80 runs for training and the remaining 20 runs for validation. For the destinations of runs in the validation set, only 4 destinations out of the 20 destinations appear in the training set, however, after adding multiple samples using the intermediate locations also as target locations, 131 out of the total 135 target locations in the validation set are covered by the training samples.

We calculate the distance between the predicted location and the actual location, and show the distribution in Figure 10. Over 75% of the predictions fall within 3 cells of the actual target location, indicating the RF model can effectively capture the relation between locations and sequences of the particle-number classes.

5 Evaluation

5.1 Evaluation Setup

5.1.1 Evaluation Testbed

We evaluate the attack using two different setups. First, we use a simulated Jackal robot running in a world created by the Gazebo simulator for a controlled evaluation environment. We perform both route and location prediction using the simulated environment. Second, we use the real-world data collected on a Nissan LEAF driving around Oxford, UK to evaluate the attack in a more realistic environment. Because the Oxford dataset only includes a limited set of routes in the city, we only evaluate route prediction using the data.

Gazebo: As shown in Figure 11, our testbed hardware has two computers connected via Ethernet ports. The client has a dual-core Intel i5-3317u processor, and the host runs a quad-

[3] See Appendix A for a more detailed discussion on how destinations of simulation runs in the training and validation sets affect the prediction accuracy.

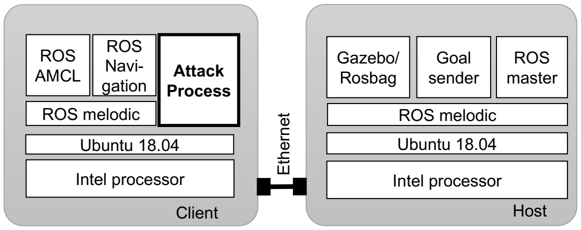
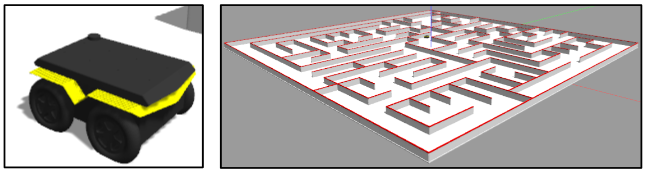


Figure 11: The testbed setup for the evaluation.



(a) A Jackal UGV; (b) a maze.

Figure 12: 3D physics-based simulation in Gazebo.

core Intel i5-3470 processor with 8GB of memory and Nvidia GT710 for graphic rendering. Both of them run Ubuntu 18.04 [16] and support ROS Melodic [8] for interaction with the physical world.

To create a simulated world, we use Gazebo [3], a ROS-compatible physics-based simulator. Figure 12 shows examples of a simulated vehicle and a maze in Gazebo. To efficiently create complex mazes for our experiments, we use an open-source Gazebo plugin [7] that generates maze models such as the one in Figure 12(b) based on a text description.

We run the entire software stack (including Ubuntu, ROS, AMCL and other control software) of a Clearpath Jackal Unmanned Ground Vehicle (UGV) [5] on the client. The Jackal UGV, shown in Figure 12(a), is a configurable and extensible platform commonly used for autonomous vehicle studies. In the simulations, we attach SICK [12] LMS1xx series LiDAR to the Jackal UGV as the sensor for 2D localization. We use the ROS implementation of AMCL [1] for LiDAR-based localization.

Oxford: For the real-world experiment, we use the Oxford RobotCar dataset [46], which is collected on a Nissan LEAF along a 10 km route around central Oxford, UK, from May 2014 to December 2015. We converted all the data to rosbag [11] format in order to replay it in the lab environment, and we run AMCL on a platform with an Intel Xeon E3-1270 four-core processor with 16GB memory, which is similar to the configuration used by the Apollo autonomous driving platform [2, 9].

For each trace in the dataset, the LiDAR scan data is provided by SICK LD-MRS LiDAR attached in front of the vehicle. Odometry information is recorded by a NovaTel SPAN-CPT GNSS/INS receiver [13]. The original RobotCar dataset uses CSV files and we preprocess them by converting the LiDAR and odometry data as well as the corresponding timestamps into a single rosbag file for evaluation. To provide a reference map for AMCL, we use the 3-D pointcloud recorded by the SICK LMS-151 LiDAR on the vehicle. We project all the points in the pointcloud of heights between 0.5m-2m (that can be captured by LD-MRS LiDAR) onto a 2-D plane, which forms the 2-D map used for AMCL.

The RobotCar dataset contains multiple traces along one route. We divide the route into seven segments, and perform route prediction using the seven segments as different routes.

5.1.2 Prime+Probe Attack Configurations

We describe the implementation details of the prime+probe attack on the client computer. The cache configurations of the processors used are listed in Table 2. We perform attacks using the L1D cache and the LLC for both platforms. The L1D attack explores an idealized scenario while the LLC attack explores less restrictive and more realistic scenario. We adopt higher sampling rate, smaller steps, and assign attack and victim processes as real-time processes in the L1D attack.

Platform	CPU	L1D		LLC	
		Sets	Size	Sets	Size
Gazebo	i5-3317u	64	32K	4096	3M
Oxford	E3-1270	64	32K	8192	8M

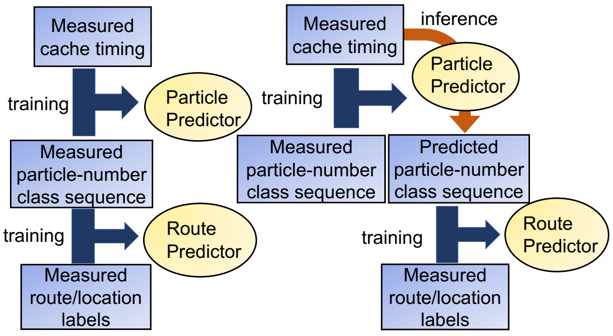
Table 2: Processor cache configurations used in experiments.

L1D attack: We assign the attack and victim processes on the same core by assigning them the same CPU affinity value. We set both attack and victim processes as real-time processes with the victim process at higher priority. In Linux, a real-time process cannot be preempted by a userspace non-real-time process. Thus, the L1D state left by the victim process will not be destroyed before probing. In addition, the higher priority of the victim process guarantees that the victim process will not be preempted by the attack process unless it yields control. For the L1D attack, we probe every set in the cache, and the entire cache is probed every 100 ms.

LLC attack: The attack and victim processes may run on different cores for the LLC attack. We use the MASTIK tookit [6], which implements the algorithm in [45] that finds the eviction sets on a physically-addressed LLC, to perform the prime+probe attack. We probe only one cache set for each consecutive 64 cache sets, which reduces the CPU utilization of the attacker and the amount of data generated. The entire cache is probed every 300 ms instead of 100 ms. Despite the reduced cache probing rate, our results show that it is still possible to predict the number of particles with high accuracy.

5.1.3 Training Procedure

Here, we describe the procedure that we use to train the particle predictor and the route predictor in our evaluation. Given the measured cache timing, the particle-number class sequences (i.e., sequence of “High” and “Low” classes), and


Measured cache timing
training
Particle Predictor
Measured particle-number class sequence
training
Route Predictor
Measured route/location labels
Measured cache timing
inference
Particle Predictor
training
Measured particle-number class sequence
Predicted particle-number class sequence
training
Route Predictor
Measured route/location labels

(a) Sequential training (b) Cascaded training

Figure 13: Procedures for training the two models.

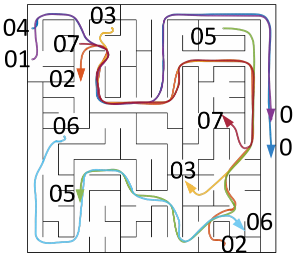


Figure 14: Maze 1 with 7 random routes.

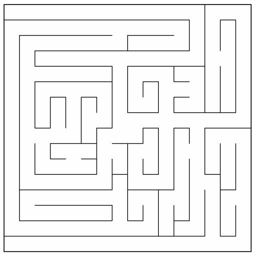


Figure 15: Maze 2.

labels for the routes or the locations, there are two possible procedures for training the two models: (1) sequential training and (2) cascaded training. As Figure 13(a) shows, in sequential training, we train the particle predictor using the measured cache timing and the measured particle-number class sequences, and then train the route predictor using the measured particle-number class sequences and the measured route/location labels.

However, errors may accumulate in the particle predictor and the route predictor, harming end-to-end prediction accuracy. We choose the cascaded training procedure as depicted in Figure 13(b). First, the particle predictor is trained the same way. Then, we use the predicted particle-number class sequences, rather than the measured particle-number class sequences, together with the measured route/location labels, to train the route predictor. Finally, the trained particle predictor and the route predictor are used for the end-to-end attack evaluation.

5.1.4 Maps for Evaluation

Gazebo: We use two mazes shown in Figure 14 and Figure 15, which are both partitioned into 16-by-16 grids. The topology of a simple maze ensures that any grid is reachable and there is only one possible path. Compared to Maze 2, Maze 1 contains more branches and less straight lanes.

Oxford: the map used in the Oxford dataset is shown in Figure 16. We select 7 routes labeled from “01” to “07” .

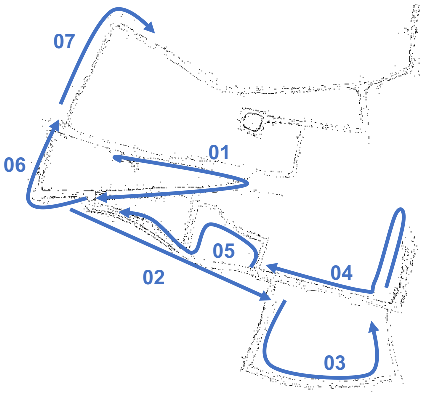


Figure 16: Map for the Oxford RobotCar dataset.

Model	Train	2-fold	5-fold	10-fold
RF-1	33.6%	45.7%	26.4%	32.9%
RF-10	66.4%	69.3%	72.9%	70.7%
RF-20	75.0%	77.9%	80.0%	76.4%
RF-50	86.4%	82.3%	87.1%	86.4%
RF-100	86.4%	82.1%	88.6%	88.6%
RF-200	90.0%	88.6%	88.6%	90.0%

Table 3: RF route-prediction accuracy with the varying number of trees, for the 7 routes in Maze 1.

Model	Train	2-fold	5-fold	10-fold
RF-1	72.2%	76.2%	70.6%	68.3%
RF-10	74.6%	73.0%	73.8%	76.2%
RF-20	75.4%	75.4%	77.0%	77.8%
RF-50	75.4%	74.6%	78.6%	79.4%
RF-100	75.4%	75.4%	77.0%	79.4%
RF-200	77.0%	73.0%	77.8%	80.2%

Table 4: RF route-prediction accuracy with the varying number of trees, for the 7 routes in Oxford.

5.2 Impact of Random Forest Size

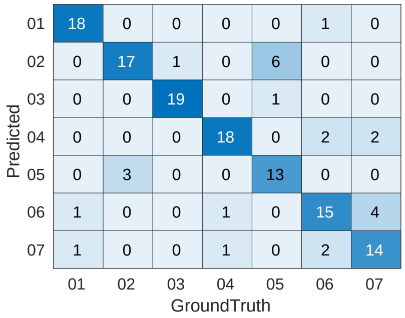
We examine the impact of the size of the random forest model on the route and location prediction accuracy. We use the ground-truth particle-number classes rather than predicted particle-number classes in this study, in order to exclude the effects of particle predictor.

5.2.1 RF Size for Route Prediction

We compare the route prediction accuracy of the RFs with a different number of trees. Table 3 and Table 4 show the result for Maze 1 and Oxford, respectively. The general trend is that the accuracy increases with the number of trees but the added benefit decreases with the number of trees.

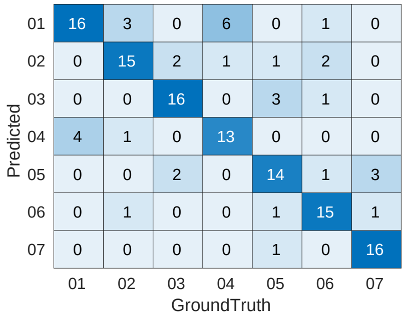
Model	Train	2-fold	5-fold
RF-1	82.1%	53.3%	62.1%
RF-10	86.6%	69.6%	72.8%
RF-20	87.5%	64.6%	73.4%
RF-50	88.9%	65.7%	74.0%
RF-100	88.1%	66.9%	74.9%
RF-200	87.5%	67.2%	74.7%

Table 5: The percentage of predictions that are within 3 grids from the true target location.



	01	02	03	04	05	06	07
01	18	0	0	0	0	1	0
02	0	17	1	0	6	0	0
03	0	0	19	0	1	0	0
04	0	0	0	18	0	2	2
05	0	3	0	0	13	0	0
06	1	0	0	1	0	15	4
07	1	0	0	1	0	2	14

Figure 17: L1D route prediction results for Gazebo.



	01	02	03	04	05	06	07
01	16	3	0	6	0	1	0
02	0	15	2	1	1	2	0
03	0	0	16	0	3	1	0
04	4	1	0	13	0	0	0
05	0	0	2	0	14	1	3
06	0	1	0	0	1	15	1
07	0	0	0	0	1	0	16

Figure 18: LLC route prediction results for Gazebo.

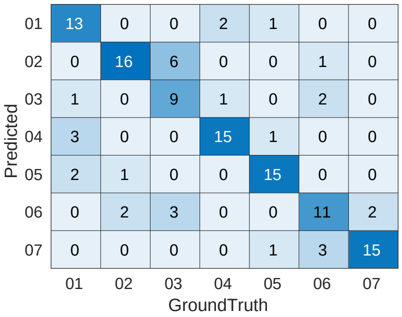
5.2.2 RF Size for Location Prediction

We compare the prediction accuracy of the random forest (RF) with a different number of trees for the training, 2-fold validation, and 5-fold validation. Table 5 shows the result. Silimar to the route prediction, the accuracy increases with the RF size but the added benefit decreases.

5.3 End-to-end Evaluation Results

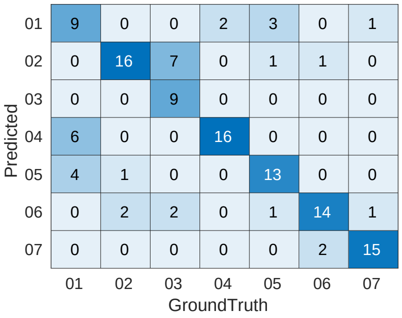
5.3.1 Route Prediction

We use the RF-100 model for the route prediction task and we use 10-fold validation for evaluating the prediction accuracy. **Gazebo:** We randomly generate seven routes on Maze 1, as shown in Figure 14, and collect 20 instances for each route. Figure 17 and Figure 18 show the classification results. The overall route prediction accuracy is 81.4% and 75% for the L1D and LLC attacks, respectively.



	01	02	03	04	05	06	07
01	13	0	0	2	1	0	0
02	0	16	6	0	0	1	0
03	1	0	9	1	0	2	0
04	3	0	0	15	1	0	0
05	2	1	0	0	15	0	0
06	0	2	3	0	0	11	2
07	0	0	0	0	1	3	15

Figure 19: L1D route prediction results for the Oxford dataset.



	01	02	03	04	05	06	07
01	9	0	0	2	3	0	1
02	0	16	7	0	1	1	0
03	0	0	9	0	0	0	0
04	6	0	0	16	0	0	0
05	4	1	0	0	13	0	0
06	0	2	2	0	1	14	1
07	0	0	0	0	0	2	15

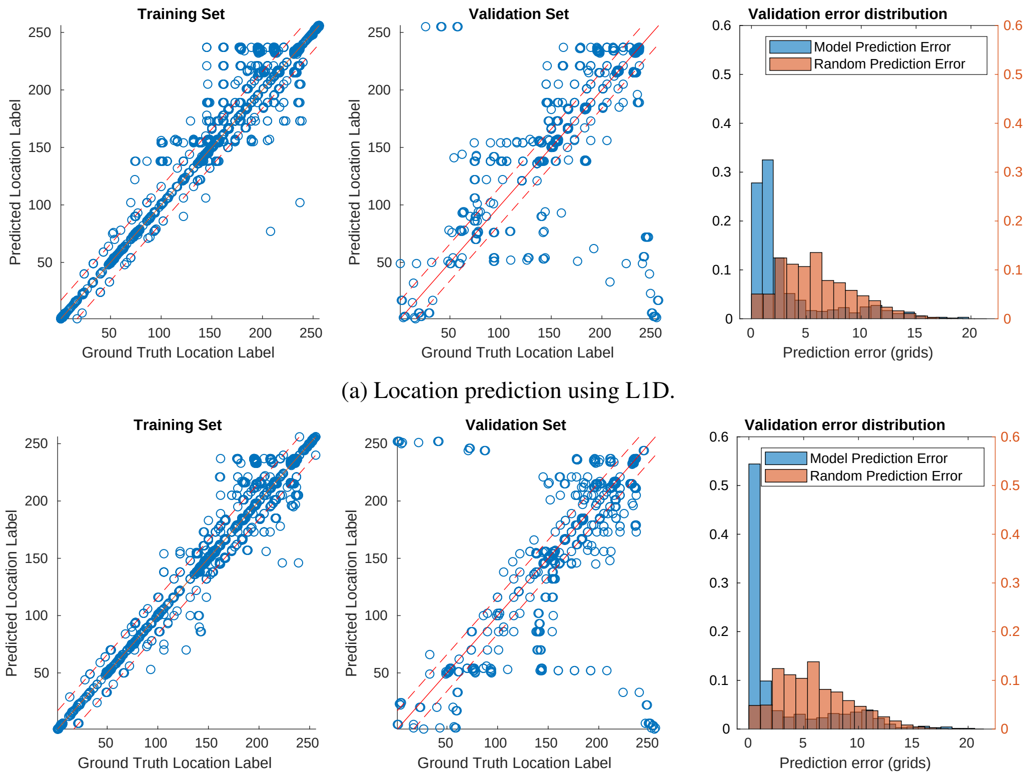
Figure 20: LLC route prediction results for the Oxford dataset.

Oxford: We use 126 sequences collected on the seven routes in the Oxford dataset for the route prediction. Figure 19 and Figure 20 show the confusion matrices of the prediction based on the L1D side channel and the LLC side channel, respectively. The route prediction accuracy is 74.6% and 73.0% for the L1D and LLC attacks, respectively.

5.3.2 Location Prediction with Gazebo

We use the RF-50 model for the location prediction task. We evaluate location prediction using the method described in Section 4.5.2. For each maze, we randomly select 100 grid centers as destinations. For a simulation run for each destination, we record the source-to-grid trajectory and the corresponding cache timing measurements and generate multiple training or validation samples by using the final destination as well as intermediate grid points on the trajectory as target locations. We then put all these generated trajectories and corresponding cache timing vectors in the dataset. Samples generated from the first 80 runs are used for training and the rest are used for validation. For Maze 1 and Maze 2, there are 3,633 and 2,048 samples in the dataset, respectively.

Figure 21 shows the training and validation accuracy of the models trained using the L1D and LLC attacks on Maze 1. For the location prediction, a wrong prediction label does not necessarily mean the prediction is far from the actual location. Thus, we also calculate the Euclidean distance as a validation error. For the L1D attack, the average validation error is 2.87 grid cells and 74.6% of the predictions fall within 3 cells. For the LLC attack, the average validation error is



(a) Location prediction using L1D.

(b) Location prediction using LLC.

Figure 21: Training, validation accuracy, and validation error distributions of end-to-end location prediction with Maze 1.

3.17 cells and 70.1% of the predictions fall within 3 cells. For random guesses, the average error is 6.01 cells and 20.2% of the predictions fall within 3 cells.

For Maze 2, the average validation error is 2.58 grid cells and 75.2% of the predictions fall within 3 cells for the L1D attack, and the average validation error is 3.61 cells and 68.7% of the predictions fall within 3 cells for the LLC attack. The average error is 7.67 cells and 13.2% of the prediction fall within 3 cells for random guesses.

5.3.3 L1D Cache vs. LLC Attacks

We summarize the prediction accuracy of the L1D cache and LLC side-channel attacks for both mazes and RobotCar experiments in Table 6. As mentioned in Section 5.1.2, the sampling periods are 100 ms and 300 ms, respectively. The table also shows the results of L1D attacks with a sampling period of 300 ms, matching that of the LLC attack.

The results show that both L1D and LLC attacks can predict a route or a location. For the L1D attacks, the prediction accuracy is similar for both sampling periods. The accuracy is slightly higher for the L1D attack than for the LLC attack. However, the L1D attack is more difficult to perform as it requires the attack and victim processes to both run on the same core.

Task-Period	Route		Location			
Map	Maze 1	Oxford	Maze 1		Maze 2	
Metric	Accuracy		error	3-grid	error	3-grid
L1D-100ms	81.4%	74.6%	2.87	74.6%	2.58	75.2%
L1D-300ms	80.0%	73.0%	3.03	73.5%	2.47	78.8%
LLC-300ms	75.0%	73.0%	3.17	70.1%	3.61	68.7%
Random-N.A.	14.3%	14.3%	6.01	20.2%	7.67	13.2%

Table 6: Comparison of prediction accuracy of the L1D attack with different sampling periods, the LLC attack, and random guess.

6 Discussion

6.1 Processor Architecture

We study and demonstrate the proposed side-channel attack on autonomous vehicles using an x86 platform. The x86 architecture is widely used in autonomous vehicle development including multiple teams during the DARPA Grand Challenge [22,23,25,51,52,67] as well as more recent commercial developments by Baidu [2], Waymo [19,20], and Uber [4]. While we did not investigate the proposed attacks on other architectures such as ARM, we believe that the attack can be generalized to other architectures given that cache timing-channel attacks have been demonstrated in many different platforms.

6.2 Generality of the Vulnerability

We rely on the adaptive behavior of AMCL to perform our attack. In general, we believe that the high-level observation that an adaptive algorithm can leak information about a vehicle's physical state can be generalized to other cyber-physical system (CPS) software whose memory access pattern depends on private physical state. Obviously, not all control/localization algorithms have such a vulnerability. For example, the data access pattern of a Kalman filter or a PID control algorithm is largely independent of input values, and does not leak physical state. However, we believe that the adaptive behaviors will become increasingly common in autonomous system software for two reasons:

1. To ensure safety and improve estimation accuracy, most autonomous vehicles have two or more sources of sensor inputs that are fused for better estimation. A simple Kalman filter-based estimation method does not work well in this scenario. Adaptive particle filter-based estimation is more suitable for the state estimation of a non-Gaussian distribution in a high-dimensional space.

2. In addition to estimation, many perception algorithms, such as object detection [59] and recognition [58], are also adaptive and have input-dependent memory access patterns. The proposed cache side-channel attack may be extended to exploit such perception algorithms to infer private physical information.

We note that if multiple software components with adaptive memory access patterns run on the same machine simultaneously, their memory accesses may interfere with each other, exhibiting more complex patterns. In that case, the machine learning model for prediction will need to either deal with interference as noise or be trained with the combined memory access patterns.

6.3 Limitations of the Attack Model

We provide a proof-of-concept end-to-end attack on inferring the route/location of an autonomous vehicle. To be successful, the proposed attack needs a victim autonomous vehicle to satisfy a few key assumptions:

- The autonomous vehicle uses a control software module with adaptive computing behavior (e.g., AMCL) where memory access patterns depend on the vehicle's physical state;

- The attacker can control a software module on the vehicle (e.g., via installing a third-party software module or compromising an existing module);

- The software module controlled by the attacker shares a cache with the victim control software module.

Given these assumptions, an attacker can deploy an attack program on the victim's computer system and spy on the control software module through a cache side channel. We believe that these assumptions are reasonable for future autonomous vehicles.

First, as mentioned in Section 6.2, software modules with adaptive computing behavior (including AMCL) have been widely used in research/industry prototypes. For efficiency, it makes an intuitive sense to dynamically adjust the amount of computation based on uncertainty or environments at runtime.

Second, connected vehicles with an Internet connection and an integrated infotainment system demand an open software architecture that exposes a wider attack surface to remote attackers. For example, it is likely that an infotainment system will allow third-party applications to be downloaded on the vehicle's computer system. Studies on connected vehicles also show that a vehicle's onboard computers contain software vulnerabilities similar to traditional computers and may be compromised by remote exploits.

Third, most vehicles are cost-sensitive, and there will be pressure to lower hardware costs by having multiple software components share hardware resources. According to an industry report [10], the automobile electronic cost will increase from 35% to 50% of the total car cost from 2020 to 2030. In fact, some companies are already adopting shared hardware in their products. For example, Visteon's SmartCore [17] runs both non-safety-critical infotainment system and safety-critical advanced driving-assistance systems on the same processor.

On the other hand, the proposed attack can be prevented by breaking one of the three key assumptions. For example, for safety, future autonomous vehicle platforms may use two different hardware platforms for safety-critical control tasks and network-connected infotainment functions.

6.4 Difficult-to-Predict Routes

We rely on the number of particles in AMCL for route/location prediction. Several real-world scenarios may exhibit less distinguishable characteristics in the traces of the number of particles, reducing the prediction accuracy.

- Identifying different routes on long highways: highways are designed for smooth traffic and generally the number of particles remain at minimum between entry and exit.

- Identifying different routes in a grid road network (e.g., downtown area): since our model does not explicitly distinguish left and right turns, the prediction might be pointing to a mirrored route/location.

However, in many scenarios, a vehicle will go through suburban, downtown roads and highways, and a route through a combinations of these roads exhibits a distinctive trace that can be distinguished from other routes.

7 Related Work

Side-Channel Attacks for Physical Properties In this paper, we use the cache side-channel attack to infer physical properties such as a vehicle's route or location. In addition to the cache side channel, there are other side channels that can be used to learn physical properties. For example, Michalevsky et al. observe that cellular signal strength, which is directly viewable in the smartphone software without privilege, is location-dependent [49]. By recording the time series of the signal strength, they are able to track the location of the smartphone. Similarly, Han et al. use the accelerometers on smartphones as a data source for location inference [39].

In addition to inferring physical location information, side channels can also be used to identify vehicle drivers. For example, Enev et al. [30] show that the driver of an automobile can be inferred by looking at the brake pedal and other types of information on the CAN bus while the vehicle is moving.

These attacks assume that an attacker has direct access to information on the physical world or behaviors such as the signal strength/accelerometer. To prevent such attacks, the accesses can be blocked by the OS. On the other hand, the attack on this paper exploits microarchitecture-level side channels and show that a program's memory access patterns can also leak information on the physical world.

Non-Crypto Cache Side Channel Our side channel attack is a non-cryptographic attack. Previous studies have also used the cache side channel for other types of non-cryptographic attacks. For example, Yan et al. use the cache side-channel attack to extract the hyperparameters of a neural network [70]. Shusterman et al. propose the cache occupancy channel, which records the number of evictions for each memory address during a fixed time period, to identify the website on a browser [63]. These attacks target relatively static information that does not change during the attack. There are also attacks on more dynamic assets. For example, Gruss et al. show that keystrokes can be inferred in real time using a cache side-channel attack [38]. Brasser et al. use cache access patterns to reveal a DNA sequence streamed into an SGX enclave for analysis at run time [24]. In this attack, the information can be inferred from a transient cache profile without considering the history. In this paper, we expand the scope of the non-crypto cache side-channel attacks by showing that a vehicle's route/location can also be learned from memory access patterns. In order to infer the route/location from memory access patterns that change quickly, our attack considers a history of cache profiles using machine-learning models.

Side-Channel Attack Protection We leverage cache side channels to extract the physical information of the vehicle. There are many proposals for defending against cache side-channel attacks. They can be classified into two categories, namely isolation and randomization. We discuss some of the representative papers here.

Isolation includes spatial isolation (partition) or temporal isolation (scheduling). For partition, DAWG [42] adopts way-partitioning to prevent side channel leakage. NoMo [29] provides dynamic cache reservation to active threads to prevent cache side-channel attacks. STEALTHMEM [41] partitions the LLC into a non-secure region and a secure region using page coloring. Temporal isolation leverages the observation that the cache side-channel attacks need coordinated timing between attack and victim programs in order to observe the cache state. The scheduler can enforce a certain scheduling policy to prevent side channel leakage [33, 65, 68].

For randomization, Wang et al. proposed the random permutation cache (RPcache) to prevent cache side-channel leakage [69]. More recently, Qureshi et al. proposed encrypted-address and remapping to prevent cache attack [56, 57]. These approaches randomize the memory address. Additionally, we can also randomize the clock that an attacker needs to use to obtain cache timing measurements. A randomized clock can prevent an attack program from getting precise timing and inferring correct state of the cache [47, 66].

Many protection mechanisms have been developed, but microarchitectural side-channel protection is not widely adopted in today's computing systems. For strong protection, many of these techniques also require hardware changes, preventing adoption by existing systems. Our study shows a new threat for autonomous vehicles, motivating stronger side-channel protection in future processor designs.

8 Conclusion

In this paper, we show that the cache side-channel attack can be used to stealthily infer routes and locations of autonomous vehicles. Our results show that the location privacy of an autonomous vehicle can be compromised when its perception and control software share hardware resources with less trusted software. Without a new processor design whose isolation guarantee includes time channels, our findings suggest that separate hardware should be used for trusted autonomous driving software and the rest of the system.

Acknowledgments

We thank our shepherd Yossi Oren and the anonymous reviewers for their helpful feedbacks on this paper. We thank Jacopo Banfi, Mark Campbell, Mohamed Ismail, Alex Ivanov, and Yizhou Zhang for the insightful discussions. This work was funded in part by NSF grant CNS-1513797 and ECCS-1932501, NASA grant NNX16AB09G and the Jacobs Fellowship of Cornell University.

References

[1] amcl - ROS Wiki. https:/wiki.ros.org/amcl.

[2] Apollo. http://apollo.auto.

[3] Gazebo. http://gazebosim.org.

[4] Inside a Self-driving Uber. https://www.infoq.com/presentations/uber-self-driving-software.

[5] Jackal UGV. https://www.clearpathrobotics.com/jackal-small-unmanned-ground-vehicle.

[6] Mastik: A Micro-Architectural Side-Channel Toolkit. https://cs.adelaide.edu.au/~yval/Mastik.

[7] Maze Generator for Gazebo. https://github.com/PeterMitrano/gzmaze.

[8] Melodic - ROS. http://wiki.ros.org/melodic.

[9] Nuvo-6108GC series. https://www.neousys-tech.com/en/product/application/edge-ai-gpu-computing/nuvo-6108gc-gpu-computing.

[10] PwC Semiconductor Report. https://www.pwc.de/de/automobilindustrie/assets/semiconductor_survey_interactive.pdf.

[11] rosbag - ROS Wiki. http://wiki.ros.org/rosbag.

[12] SICK USA. https://www.sick.com.

[13] SPAN-CPT Single Enclosure GNSS/INS Receiver. https://www.novatel.com/products/span-gnss-inertial-systems/span-combined-systems/span-cpt.

[14] Tesla Autopilot. https://www.tesla.com/autopilot.

[15] Uber Advanced Technologies Group. https://www.uber.com/info/atg.

[16] Ubuntu 18.04.01 LTS (Bionic Beaver). http://releases.ubuntu.com/18.04.

[17] Visteon SmartCore. https://www.visteon.com/products/domain-controller.

[18] Waymo. https://www.waymo.com.

[19] Waymo and Intel Collaborate on Self-Driving Car Technology. https://newsroom.intel.com/editorials/waymo-intel-announce-collaboration-driverless-car-technology.

[20] Waymo's Autonomous Fleet Has Intel Inside. https://www.electronicdesign.com/automotive/waymo-s-autonomous-fleet-has-intel-inside.

[21] Zipline. https://www.flyzipline.com.

[22] A. Bacha, C. Bauman, R. Faruque, M. Fleming, C. Terwelp, C. Reinholtz, D. Hong, A. Wicks, T. Alberi, D. Anderson, et al. Odin: Team victortango's entry in the DARPA urban challenge. *Journal of Field Robotics*, 25(8):467–492, 2008.

[23] J. Bohren, T. Foote, J. Keller, A. Kushleyev, D. Lee, A. Stewart, P. Vernaza, J. Derenick, J. Spletzer, and B. Satterfield. Little ben: The ben franklin racing team's entry in the 2007 DARPA urban challenge. *Journal of Field Robotics*, 25(9):598–614, 2008.

[24] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[25] M. Buehler, K. Iagnemma, and S. Singh. *The DARPA Urban Challenge: Autonomous Vehicles in City Traffic*, volume 56. Springer, 2009.

[26] L. B. Cremean, T. B. Foote, J. H. Gillula, G. H. Hines, D. Kogan, K. L. Kriechbaum, J. C. Lamb, J. Leibs, L. Lindzey, C. E. Rasmussen, et al. Alice: An information-rich autonomous vehicle for high-speed desert navigation. *Journal of Field Robotics*, 23(9):777–810, 2006.

[27] L. de Paula Veronese, E. de Aguiar, R. C. Nascimento, J. Guivant, F. A. A. Cheein, A. F. De Souza, and T. Oliveira-Santos. Re-emission and satellite aerial maps applied to vehicle localization on urban environments. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4285–4290, 2015.

[28] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen. Prime+ abort: A timer-free high-precision L3 cache attack using Intel TSX. In *26th USENIX Security Symposium*, pages 51–67, 2017.

[29] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):35, 2012.

[30] M. Enev, A. Takakuwa, K. Koscher, and T. Kohno. Automobile driver fingerprinting. *Proceedings on Privacy Enhancing Technologies*, 2016(1):34–50, 2016.

[31] M. Farsi, K. Ratcliff, and M. Barbosa. An overview of controller area network. *Computing & Control Engineering Journal*, 10(3):113–120, 1999.

[32] L. C. Fernandes, J. R. Souza, G. Pessin, P. Y. Shinzato, D. Sales, C. Mendes, M. Prado, R. Klaser, A. C. Magalhaes, A. Hata, et al. Carina intelligent robotic car: architectural design and applications. *Journal of Systems Architecture*, 60(4):372–392, 2014.

[33] A. Ferraiuolo, Y. Wang, D. Zhang, A. C. Myers, and G. E. Suh. Lattice priority scheduling: Low-overhead timing-channel protection for a shared memory controller. In *22nd IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 382–393, 2016.

[34] D. Fox. KLD-sampling: Adaptive particle filters. In *Advances in Neural Information Processing Systems (NIPS)*, pages 713–720, 2002.

[35] D. Fox. Adapting the sample size in particle filters through KLD-sampling. *The International Journal of Robotics Research*, 22(12):985–1003, 2003.

[36] J. Friedman, T. Hastie, and R. Tibshirani. *The Elements of Statistical Learning*, volume 1. Springer Series in Statistics New York, NY, USA, 2001.

[37] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+ Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[38] D. Gruss, R. Spreitzer, and S. Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium*, pages 897–912, 2015.

[39] J. Han, E. Owusu, L. T. Nguyen, A. Perrig, and J. Zhang. Accomplice: Location inference using accelerometers on smartphones. In *Fourth International Conference on Communication Systems and Networks (COMSNETS)*, pages 1–9, 2012.

[40] A. Y. Hata and D. F. Wolf. Feature detection for vehicle localization in urban environments using a multilayer lidar. *IEEE Transactions on Intelligent Transportation Systems*, 17(2):420–429, 2015.

[41] T. Kim, M. Peinado, and G. Mainar-Ruiz. STEALTH-MEM: System-level protection against cache-based side channel attacks in the cloud. In *21st USENIX Security symposium*, pages 189–204, 2012.

[42] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer. DAWG: A defense against cache timing attacks in speculative execution processors. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 974–987, 2018.

[43] J. Leonard, J. How, S. Teller, M. Berger, S. Campbell, G. Fiore, L. Fletcher, E. Frazzoli, A. Huang, S. Karaman, et al. A perception-driven autonomous urban vehicle. *Journal of Field Robotics*, 25(10):727–774, 2008.

[44] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. ARMageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium*, pages 549–564, 2016.

[45] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy (SP)*, pages 605–622, 2015.

[46] W. Maddern, G. Pascoe, C. Linegar, and P. Newman. 1 year, 1000 km: The Oxford RobotCar dataset. *The International Journal of Robotics Research*, 36(1):3–15, 2017.

[47] R. Martin, J. Demme, and S. Sethumadhavan. TimeWarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. *ACM SIGARCH Computer Architecture News*, 40(3):118–129, 2012.

[48] Y. Michalevsky, D. Boneh, and G. Nakibly. Gyrophone: Recognizing speech from gyroscope signals. In *23rd USENIX Security Symposium*, pages 1053–1067, 2014.

[49] Y. Michalevsky, A. Schulman, G. A. Veerapandian, D. Boneh, and G. Nakibly. Powerspy: Location tracking using mobile device power analysis. In *24th USENIX Security Symposium*, pages 785–800, 2015.

[50] I. Miller and M. Campbell. Particle filtering for map-aided localization in sparse GPS environments. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1834–1841, 2008.

[51] I. Miller, M. Campbell, D. Huttenlocher, F.-R. Kline, A. Nathan, S. Lupashin, J. Catlin, B. Schimpf, P. Moran, N. Zych, et al. Team Cornell's Skynet: Robust perception and planning in an urban environment. *Journal of Field Robotics*, 25(8):493–527, 2008.

[52] M. Montemerlo, J. Becker, S. Bhat, H. Dahlkamp, D. Dolgov, S. Ettinger, D. Haehnel, T. Hilden, G. Hoffmann, B. Huhnke, et al. Junior: The Stanford entry in the urban challenge. *Journal of Field Robotics*, 25(9):569–597, 2008.

[53] P. Narváez, K.-Y. Siu, and H.-Y. Tzeng. New dynamic algorithms for shortest path tree computation. *IEEE/ACM Transactions On Networking*, 8(6):734–746, 2000.

[54] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1406–1418, 2015.

[55] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006.

[56] M. K. Qureshi. CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 775–787, 2018.

[57] M. K. Qureshi. New attacks and defense for encrypted-address cache. In *46th IEEE/ACM International Symposium on Computer Architecture (ISCA)*, pages 360–371, 2019.

[58] J. Redmon and A. Farhadi. YOLO9000: better, faster, stronger. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7263–7271, 2017.

[59] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-CNN: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems (NIPS)*, pages 91–99, 2015.

[60] J. Rohde, I. Jatzkowski, H. Mielenz, and J. M. Zöllner. Vehicle pose estimation in cluttered urban environments using multilayer adaptive monte carlo localization. In *19th International Conference on Information Fusion (FUSION)*, pages 1774–1779, 2016.

[61] H. Sakoe, S. Chiba, A. Waibel, and K. Lee. Dynamic programming algorithm optimization for spoken word recognition. *Readings in Speech Recognition*, 159:224, 1990.

[62] C. Seiffert, T. M. Khoshgoftaar, J. Van Hulse, and A. Napolitano. RUSBoost: A hybrid approach to alleviating class imbalance. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 40(1):185–197, 2010.

[63] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom. Robust website fingerprinting through the cache occupancy channel. In *28th USENIX Security Symposium*, pages 639–656, 2019.

[64] R. Spangenberg, D. Goehring, and R. Rojas. Pole-based localization for autonomous vehicles in urban scenarios. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2161–2166, 2016.

[65] R. Sprabery, K. Evchenko, A. Raj, R. B. Bobba, S. Mohan, and R. Campbell. Scheduling, isolation, and cache allocation: A side-channel defense. In *IEEE International Conference on Cloud Engineering (IC2E)*, pages 34–40, 2018.

[66] D. Trilla, C. Hernandez, J. Abella, and F. J. Cazorla. Cache side-channel attacks and time-predictability in high-performance critical real-time systems. In *55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018.

[67] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer, et al. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics*, 25(8):425–466, 2008.

[68] V. Varadarajan, T. Ristenpart, and M. M. Swift. Scheduler-based defenses against cross-VM side-channels. In *23rd USENIX Security Symposium*, pages 687–702, 2014.

[69] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 494–505, 2007.

[70] M. Yan, C. Fletcher, and J. Torrellas. Cache telepathy: leveraging shared resource attacks to learn DNN architectures. In *29th USENIX Security Symposium*, 2020.

[71] Y. Yarom and K. Falkner. FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium*, pages 22–25, 2014.

A Impact of Destination Selection on Location Prediction

The proposed classification algorithm cannot predict a location that is not in the training set. Our location prediction experiments are performed using randomly-selected destinations where the training set and the validation set contain different sets of destinations. Thus, we generate multiple training/validation samples using the intermediate locations along each path. The intermediate locations help creating more samples in both sets that share the same location label even when the destinations of the entire paths are different. For example, a simulation run with a length L to one destination has $L-1$ intermediate locations, and generates L samples with L different target locations to predict. Intuitively, if the simulation destinations in the training set and the validation set are spatially close, there will be more intermediate locations that are common between the two sets, which will lead to more validation samples whose target locations exist in the training set.

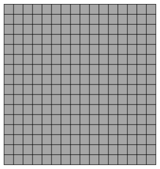


Figure 22: Both the training and validation sets contain the identical set of the destinations on the map (grey).

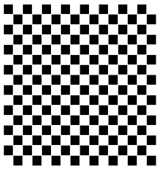


Figure 23: Training set destinations (black) and validation set destinations (white) are interleaved.

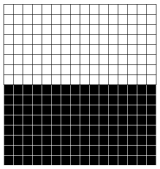


Figure 24: Training set destinations (black) and validation set destinations (white) are separated.

A.1 Destination Selection Strategy

Here, we study different strategies for selecting destinations of simulation runs for the training set and the validation set and their impacts on prediction accuracy.

Identical Destinations In this strategy, the training set and the validation set have an identical set of simulation destinations (Figure 22). We select all 256 locations in Maze 1. For each destination, we use two simulation runs, one for the training set and one for the validation set, for the total 512 runs.

Interleaved Destinations In this strategy, the training set and the validation set have interleaved destinations, forming a chessboard pattern (Figure 23). There is no overlap between the training and validation sets. We select the "black" destinations for training and the "white" destinations for validation. For each destination, we have two runs for the total 512 runs. The interleaved strategy leads to mutually exclusive destinations in the training and the validation sets, but for each destination in the validation sets, there is a destination in the training set is just one grid away.

Separated Destinations In this strategy, the training set and the validation set are spatially separated. (Figure 24). We use the bottom part of Maze 1 for the training set and the top part of Maze 1 for the validation set. For each destination, we have two runs for the total 512 runs. In this strategy, the destinations in the training set and the validation set are not only mutually exclusive, but also spatially far part in the opposite directions.

In Table 7, we compare the number of overlapped target locations between the training set and the validation set for different destination-selection strategies. The table shows the results for the three strategies discussed above as well as the random-selection scheme described in Section 4.5.2. Note that the target locations in the table include the intermediate locations in each simulation run. The table shows the total number of unique target locations in the validation set as well as the number of target locations that also appear in at least one sample in the training set. The samples indicate the individual samples in the validation set that are used to obtain the prediction accuracy; multiple samples may have the same target location. For the identical-destination strategy, 100% of the target locations in the validation sets are covered by the training set. For the interleaved strategy, 88.8% of the target locations and 99.4% of the validation samples are covered by the training set. However, in the separated strategy, only 47.2% of the target locations are covered by the training set. The uncovered target locations have location labels not found in the training set, thus, they will lead to the same number of prediction errors. As a consequence, the prediction accuracy for the separated destination will be lower.

Strategy		Identical	Interleaved	Separated	Random
Target locations	Total	256	224	178	135
	In training	256	199	84	131
	Percentage	100 %	88.8 %	47.2 %	97.0 %
Samples	Total	8,627	8,650	10,954	726
	In training	8,627	8,601	9,408	720
	Percentage	100 %	99.4 %	85.9 %	99.2 %

Table 7: The number and the percentages of the target locations and samples in the validation set under different destination selection strategies.

Strategy	Identical	Interleaved	Separated	Random
3-grid accuracy	75.9%	77.3%	50.7%	74.6 %
Mean error	2.49	2.40	5.44	2.87

Table 8: Prediction results using different strategies for choosing destinations in the training and the validation sets.

A.2 Prediction Results

We compare the prediction results of the three strategies and the random destination strategy in Table 8. The prediction accuracy for the interleaved, identical, and random destinations are similar, while the accuracy for separated destinations is significantly lower. This is consistent with the low percentage of the target locations that are covered by the training set under the separated-destination strategy. The result shows that the spatial proximity of destinations in the training and validation sets, rather than the exact overlap of the destinations in the training and the validation set, is important for the prediction accuracy. The random destination strategy, which we used in Section 4 and Section 5, preserves the spatial proximity of the destinations between the training and validation sets. Thus, the prediction accuracy is similar to that of using identical and interleaved destinations strategies.