ON CYBER-PHYSICAL FORENSICS, ATTACKS, AND DEFENSES

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Rohit Bhatia

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2019

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Dongyan Xu, Chair
    Department of Computer Science
Dr. Dave (Jing) Tian
    Department of Computer Science
Dr. Zeynel Berkay Celik
    Department of Computer Science
Dr. Sonia Fahmy
    Department of Computer Science

Approved by:
    Dr. Clifton W. Bingham by Monica M. Shively
        Head of the Departmental Graduate Program

To my parents.

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor, Professor Dongyan Xu, for providing me guidance and support during my time at Purdue. His tireless and constant efforts towards research have taught me the value of hard work and perseverance. He has also provided me with invaluable experience regarding identifying research problems in various domains, evaluating different solutions, and reviewing the work of other researchers. Most importantly of all, he has also taught me the importance of communication and delivering our contributions to fellow researchers.

I would also like to thank Professor Sonia Fahmy, Professor Berkay Celik, and Professor Dave Tian for serving in my final exam committee. Their invaluable comments have helped improve this dissertation. I would also like to thank Professor Xiangyu Zhang, Professor Mathias Payer, Professor Byoungyoung Lee, and Professor Berkay Celik for their insights and contributions in our research collaborations.

I owe special thanks to Dr. Vireshwar Kumar for being a supportive mentor and collaborator, and motivating me to put in my best effort. I am also thankful to fellow members of the FRIENDS lab, both former and current, including Khaled Serag, Taegyu Kim, Brendan Saltaformaggio, Yonghwi Kwon, and Chunghwan Kim among many others.

I am grateful to my old and current roommates Sunny Chugh, Avanish Mishra, Mayank Kakodkar, and Viplove Arora for being my sounding boards and for teaching me how to cook. I am also grateful for all the friendships I have cultivated at Purdue, including but not limited to my pseudo-roommate Akash Patil, Rohil Jain, Anamika Shreevastava, Vibhav Bisht, and Akash Kumar. I cherish the time we spent together and hope to spend more in the future.

Last but not the least, I owe special thanks to my parents for providing support through difficult times. I am also grateful to my sister Sneha and brother-in-law Nitin

for their constant encouragement. I am also thankful to my nephews Siddhaant and Ayansh for being a constant source of happiness in my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Bhatia, Rohit Ph.D. Student, Purdue University, December 2019. On Cyber-Physical Forensics, Attacks, and Defenses. Major Professor: Dongyan Xu.

Cyber-physical systems, through various sensors and actuators, are used to handle interactions of the cyber-world with the physical-world. Conventionally, the temporal component of the physical-world has been used only for estimating real-time deadlines and responsiveness of control-loop algorithms. However, there are various other applications where the relationship of the temporal component and the cyber-world are of interest. An example is the ability to reconstruct a sequence of past temporal activities from the current state of the cyber-world, which is of obvious interest to cyber-forensic investigators. Another example is the ability to control the temporal components in broadcast communication networks, which leads to new attack and defense capabilities. These relationships have not been explored traditionally.

To address this gap, this dissertation proposes three systems that cast light on the effect of temporal component of the physical-world on the cyber-world. First, we present Timeliner, a smartphone cyber-forensics technique that recovers past actions from a single static memory image. Following that, we present work on CAN (Controller Area Network), a broadcast communication network used in automotive applications. We show in DUET that the ability to control communication temporally allows two compromised ECUs, an attacker and an accomplice, to stealthily suppress and impersonate a victim ECU, even in the presence of a voltage-based intrusion detection system. In CANDID, we show that the ability to temporally control CAN communication opens up new defensive capabilities that make the CAN much more secure.

The evaluation results show that Timeliner is very accurate and can reveal past evidence (up to an hour) of user actions across various applications on Android devices. The results also show that DUET is highly effective at impersonating victim ECUs while evading both message-based and voltage-based intrusion detection systems, irrespective of the features and the training algorithms used. Finally, CANDID is able to provide new defensive capabilities to CAN environments with reasonable communication and computational overheads.

1 INTRODUCTION

1.1 Dissertation Statement

Systems such as those used in vehicles, aircrafts, healthcare, smart-cities, manufacturing, and many other applications often involve the computer algorithms controlling and reacting to the various components of the physical world. Such systems, called cyber-physical systems, have extensive models for interactions between the control algorithm and the physical-world through actuators and sensors. However, the analysis of temporal component of the physical-world is restricted to calculating real-time deadlines of actuator messages and responsiveness of the control-loop algorithms to sensor inputs. The temporal component of the physical-world is however of interest in various other applications. One such application is identifying a sequence of past actions for cyber-forensic investigators, which can be inferred from the cyber-world if the temporal sequence of past actions from the physical-world is captured in the cyber-world's memory. Another application is in broadcast communication networks such as Controller Area Network (CAN), where temporal control of communication can open new capabilities.

Just like forensic investigators retracing a criminal's steps in the real world, cyber-forensic investigators also seek to recreate a criminal's past actions on their digital devices. Traditional approach of timeline reconstruction involves manually recovering application-specific evidence [1–3]. This approach neither includes the vast variety of smartphone applications, nor the inter-application order of the criminal's past actions. Further, such a sequence is unlikely to be stored explicitly on a smartphone, and is disabled by various smartphone manufacturers [4]. As a result, reconstruction of a criminal's past actions remains an open challenge, which can be addressed by exploring the relationship between temporal sequence of the past actions to the smartphone memory.

Controller Area Network (CAN) bus is a broadcast communication bus used in vehicles. Researchers [5–14] have shown that the increasingly internet-connected ECUs of modern vehicles are susceptible to compromise by malicious adversaries. As these compromised ECUs have access to the CAN bus, the adversary is able to control various critical functions [8, 9, 13–15] of the vehicle. A variety of defenses have been proposed, from costly and impractical cryptographic solutions [16, 17] to the more practical intrusion detection systems [11, 18–23]. Voltage-based intrusion detection system (VIDS) in particular have been shown highly effective in detecting compromised ECUs masquerading as another, and are considered state-of-the-art. This dissertation however shows that controlling the transmission time of CAN messages introduces new capabilities, which allow (1) an attacker to evade VIDS, and also (2) enables the CAN bus to have stronger defenses.

1.2 Contributions

The work explores how temporal aspect of certain systems affects the cyber components of those system. The work particularly focuses on smartphone forensics, CAN bus attacks, and CAN bus defenses. The contributions of this dissertation can be summarized as follows:

- Timeliner [24] is a smartphone memory forensics technique for automatically reconstructing the sequence of past actions performed by a criminal. Timeliner is able to accurately recover and reorder actions taken over different applications, even if the applications were terminated.

- DUET is a stealthy and advanced ECU masquerade attack. DUET allows two ECUs, an attacker and an accomplice, to suppress and masquerade as a victim ECU, all while evading detection from voltage and message based intrusion detection systems.

- RAID is an effective defense against DUET. RAID is lightweight and can be run complementary to existing IDS.
- CANDID provides a software-only solution to allow programmatic modifications of various components of CAN messages. These programmatic modifications can be performed efficiently within the resource-constrained ECUs, and enables new defensive capabilities.

1.3 Dissertation Components

1.3.1 Timeliner

An essential forensic capability is to infer the sequence of actions performed by a suspect in the commission of a crime. Unfortunately, for cyber investigations, user activity timeline reconstruction remains an open research challenge, currently requiring manual identification of datable artifacts/logs and heuristic-based temporal inference. We propose a memory forensics capability to address this challenge. We present Timeliner, a forensics technique capable of automatically inferring the timeline of user actions on an Android device *across all apps*, from a single memory image acquired from the device. Timeliner is inspired by the observation that Android app Activity launches leave behind key self-identifying data structures. More importantly, this collection of data structures can be temporally ordered, owing to the predictable manner in which they were allocated and distributed in memory. Based on these observations, Timeliner is designed to (1) identify and recover these residual data structures, (2) infer the user-induced transitions between their corresponding Activities, and (3) reconstruct the device-wide, cross-app Activity timeline. Timeliner is designed to leverage the memory image of Android's centralized ActivityManager service. Hence, it is able to sequence Activity launches across all apps — even those which have terminated. Our evaluation shows that Timeliner can reveal substantial evidence (up to an hour) across a variety of apps on different Android platforms.

1.3.2 DUET

The controller area network (CAN) is widely adopted in modern automobiles to enable communications among in-vehicle electronic control units (ECUs). Lacking mainstream network security capabilities due to resource constraints, the CAN bus is susceptible to a variety of attacks launched by malicious compromised ECUs. A common type of attack is the ECU masquerade attack, where a compromised ECU impersonates an uncompromised victim ECU and spoofs the latter's CAN messages. A cost-effective defense against such attacks is the CAN bus voltage-based intrusion detection system (VIDS) which have been proved effective in detecting masquerade attacks that each involve a single attacker ECU. We present a novel masquerade attack strategy called DUET, which involves a duo of "attacker" and "accomplice" ECUs working together to manipulate, silence and then impersonate a victim ECU while evading state-of-the-art VIDS. DUET follows a three-stage strategy while leveraging two new attack tactics that exploit fundamental deficiencies of the CAN protocol. Our evaluation of DUET on real CAN buses (including three in two real cars) demonstrates high stealth, swiftness, and success rate of DUET. Finally, we propose a lightweight, effective defense that prevents DUET with negligible computation and reasonable communication overheads.

1.3.3 CANDID

A modern automobile is equipped with a variety of electronic control units (ECUs) which communicate over a controller area network (CAN) to enable safety-critical functions such as adaptive cruise control. Unfortunately, lack of any fundamental security mechanism in CAN makes it vulnerable to a variety of attacks including denial-of-service, replay and masquerade attacks. Some of the vital challenges in thwarting such attacks include deploying an effective defense on resource-constrained ECUs and ensuring backward-compatibility of CAN messages while satisfying their stringent latency constraints. To spur the development of innovative solutions for addressing these challenges, we propose CANDID, a software-only solution which provides novel tools for dynamically modifying messages while being compliant to the CAN protocol. CANDID can be utilized to implement a *dialect* (spoken by ECUs) which comprises of a suite of policies for mitigating security vulnerabilities. We present specific case studies to illustrate how such dialects practically defend CAN against the known attacks in the existing literature. Results obtained through extensive experiments on our testbed and two real cars validate that CANDID readily enables the deployment of dialects in existing ECUs with minimal computation and communication overhead.

1.4 Dissertation Organization

The dissertation is organized as follows:

- Chapter 1 describes the need to understand the relationship between the temporal component of the physical-world, and the cyber-world. It then presents the various components of this dissertation, their research problems, and the techniques used in our solutions.
- Chapter 2 provides the details for Timeliner, our memory forensics work.
- Chapter 3 describes DUET, our novel ECU masquerade attack. It also includes RAID, the lightweight and effective defense against DUET.
- Chapter 4 covers CANDID, our software-only solution to enact policies allowing dynamic modification of CAN messages.
- Chapter 5 describes the various research efforts that are closely related to our contributions, and provides key differentiating features.
- Chapter 6 concludes this dissertation.

2 “TIPPED OFF BY YOUR MEMORY ALLOCATOR”: DEVICE-WIDE USER ACTIVITY SEQUENCING FROM ANDROID MEMORY IMAGES

One of the critical steps in a forensic investigation is deriving a timeline of a suspect’s activities. As described in [25], this task “involves evaluating the context of a scene and the physical evidence found there in an effort to identify what occurred and in what order it occurred.” In the physical world, this is often modeled as inferring causal and temporal relations between events involving the suspect(s) and victim(s).

In cyber investigations inferring a suspect’s temporal sequence of activities on his/her mobile device remains a challenging problem. Currently, investigators must manually coalesce datable (often modifiable) forms of application-specific evidence: e.g., call/message databases [1, 2] or web browsing logs [3] saved on a mobile device’s SD-card. Since no semantic links readily exist between these evidence sources, the activity timelines reconstructed in this way are often heuristic and incomplete at best. Further, while Android captures coarse-grained information about user actions, major Android phone manufacturers routinely disable these features [4].

To illustrate this challenge, consider the variety of mobile apps utilized in the commission of even a simple espionage crime: Upon receiving a go-ahead call from a conspirator, the criminal uses his smartphone camera to take photographs of sensitive documents and forwards them via a secure messaging app to the conspirator. Being wary of his safety, the criminal immediately deletes the photographs and terminates the messaging app. Finally, the criminal opens his banking app to verify the payment from his conspirator in his account. Each of these actions alone does not suggest a pattern of espionage, but the causal relationship between the various user actions (derived from their temporal ordering) indicates the commission of the crime. Unfortunately, in order to reconstruct the temporal sequence of these actions, investigators currently have to perform three manual steps: (1) recover application-specific evidence from the confiscated device, (2) infer temporal ordering relations among the user actions, and (3) derive a global timeline to reveal the causal relationships between user actions.

In this paper, we will show how Android memory forensics — performed on a single memory image *without* temporal logs — can achieve high-accuracy user/app action sequencing. Through an in-depth analysis of mobile app activity handling within Android, we have identified a set of *application-independent* in-memory artifacts which represent state and display changes across all applications. These artifacts, aptly named *Activities*, are generated and managed by the Android subsystem (specifically the ActivityManagerService) — putting them out of the reach of any app's execution and making their recovery and interpretation generic with regard to the app/user actions they represent. Further, each *Activity launch* that an app performs is unique and leaves behind a signature, namely a collection of *residual data structures* that are indicative of a specific user action on the device.

Leveraging the power of app-generic Activities, we then turn our attention to the *automatic, device-wide temporal sequencing of app Activities*. Again, we glean clues from the Android subsystem's in-memory artifacts. By modeling the operation of the Android memory allocator, we found that these residual data structures are allocated memory locations (called "slots") in a sequentially increasing ordering. Put simply: if it is possible to recover the *spatial ordering* of these slots for a sequence of historic Activities, then we can perform inference of the specific *temporal ordering* of those Activities.

Inspired by these findings, we develop Timeliner, a memory forensics technique which automatically performs inference of an Android device user's past actions from Activities *across different apps (even those which have terminated)* found in one memory image. The development of Timeliner overcomes a number of challenges during the identification and subsequent temporal sequencing of Activities: (1) Frequent allocation and garbage collection (GC) induces fragmentation in memory, causing consecutive allocations to become non-contiguous, making identification and segregation of allocations into Activity launches difficult. (2) In a fragmented memory, identification of spatial ordering among the Activity-identifying residual data structures is also challenging. (3) Android's memory allocator utilizes thread-local buffers for small allocations, spreading some residual data structures across several memory locations, making the spatial ordering potentially ambiguous. (4) Long temporal gaps between Activity launches can lead to spatial gaps between their residual data structures, requiring Timeliner to join multiple separated spatial orderings.

To address these challenges, Timeliner works in three stages. First, Timeliner identifies and recovers all *residual data structures* (several hundred per Activity launch) from a subject Android memory image (Section 2.2.1). This step is akin to an investigator first identifying as many crime-related events as possible. Next, like an investigator finding causal relationships between the crime-related events, Timeliner uses spatial ordering of the recovered Activity launches to infer transitions between pairs of Activities (Section 2.2.2). Lastly, similar to an investigator reconstructing the expected timeline, Timeliner orders the pairwise transitions to derive the global ordering of Activities (Section 2.2.2). Note that Timeliner does not compete with forensic tools that recover evidence based on content [26–29], but instead complements them by providing contextual meaning to the evidence they recover by establishing a device-wide, inter-app temporal sequence of user actions.

We have evaluated Timeliner, using micro-benchmarks and recreations of real criminal investigations, across multiple commercially available Android phones and a wide range of applications covering messaging, voice calls, banking, email, file management, and video streaming. Our results show that Timeliner is highly accurate (our case studies recover as many as 18 prior device-wide Activities) and provides convincing evidence to investigators through the reconstructed timelines of user actions. We also show that the techniques behind Timeliner are neither limited to the Android platform (by applying Timeliner to the jemalloc allocator) nor to only user-actions (by applying Timeliner to inter-application interactions called Broadcasts).

2.1 Background

Activity is the fundamental abstraction for a user action provided by the Android framework. In particular, an Activity is described as a "single, focused thing that the user can do" in the Android developer documentation [30]. To showcase how each Activity within an app models such a "single, focused thing," Table 2.1 in Section 2.3 lists a number of Activities we encountered during our evaluation. As a user interacts with their device (e.g., clicking a button on the UI or receiving a call), the Activity corresponding to each action will be "launched".

Timeliner aims to reconstruct the temporal sequence of recent Activity launches that occurred on a subject device. To identify the launch of a specific Activity from an input memory image, Timeliner locates and recovers the unique data structures left behind from the execution of the Activity launch's logic. Then, to reconstruct the sequence of these Activity launches, Timeliner infers their temporal ordering from the data structures' spatial ordering (their allocation pattern in memory). We now discuss the enabling principles for these techniques.

2.1.1 Memory Allocator Design

Android's memory allocator is a "Run" of "Slots" allocator (named *RosAlloc*), where *slots* are individual memory locations for object allocations and a *run* is a list of slots of the same size. Like other Run of Slots allocators (e.g. phkmalloc, jemalloc), allocation is handled through a per-run bitmap. Each thread in a process manages its own thread-local runs for smaller slots, along with shared runs for larger slots. As an example, Figure 2.1 shows a few sample thread-local and shared runs of different sizes for two threads.

An allocation request is first assigned to the run whose slots best fit the requested size. A slot is chosen from this run based on a "first-available" algorithm which assigns it to the first empty slot picked from the bitmap, and subsequently the object is instantiated at this location. When the run is completely filled, a new run is used,

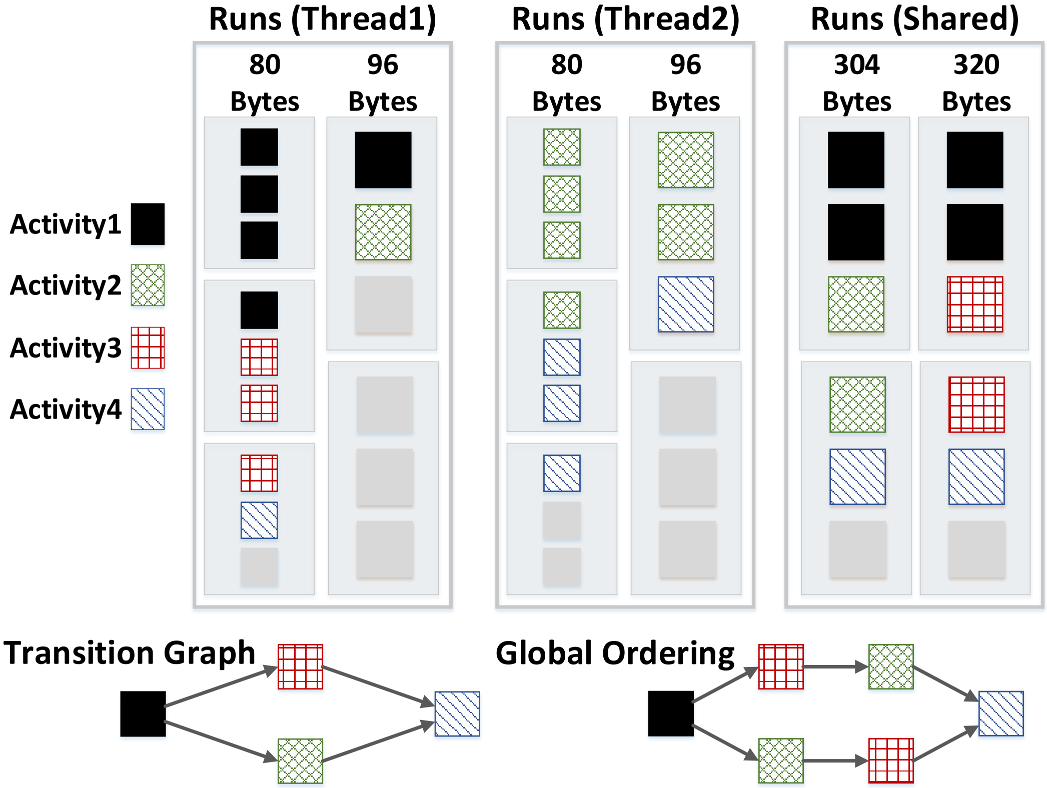


Figure 2.1.: Spatial Layout of Allocations in RosAlloc.

with a similar “first-available” algorithm choosing the run with the lowest address. These algorithms are deliberately designed to reduce fragmentation, preferring low addresses for allocations.

The implication of these “first-available” algorithms is as follows: If an allocation a (immediately) temporally precedes an allocation b of the same size, then a will be assigned a slot preceding b. Put simply, allocations that have a temporal ordering will be assigned memory locations that have a corresponding spatial ordering. This property is key to Timeliner, which attempts to solve the reverse problem: Identify the original temporal ordering of the allocations from their spatial ordering in a memory image.

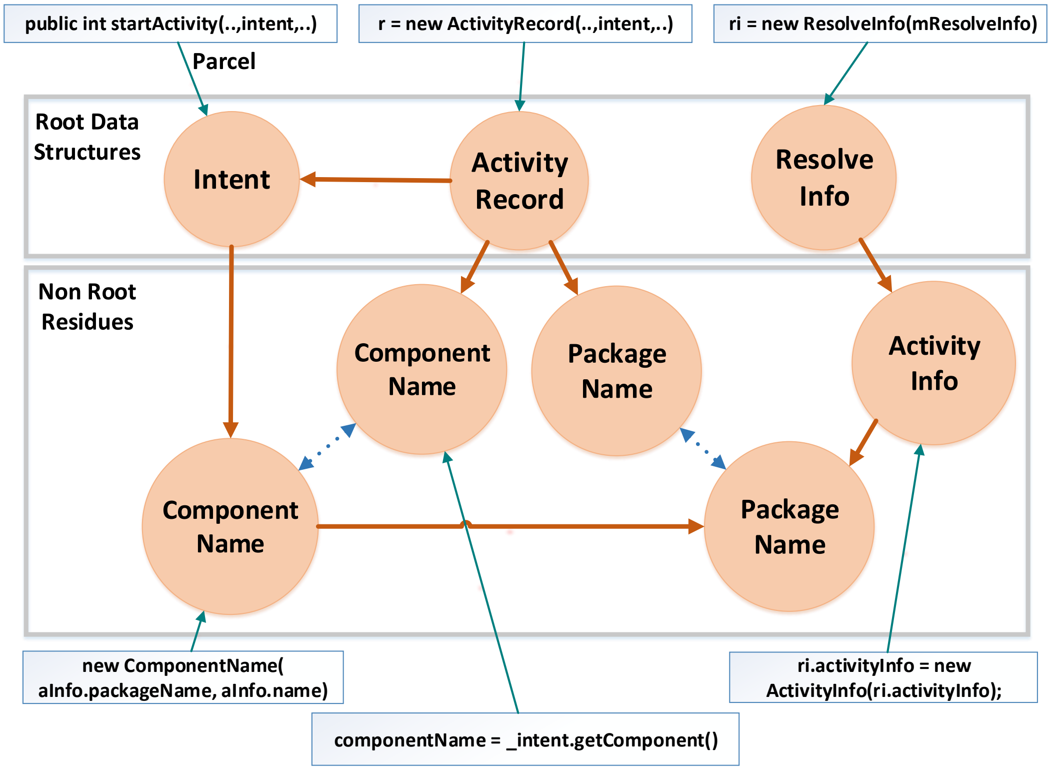

public int startActivity(..,intent,..)
r = new ActivityRecord(..,intent,..)
ri = new ResolveInfo(mResolveInfo)
Parcel
Root Data Structures
Intent
Activity Record
Resolve Info
Non Root Residues
Component Name
Package Name
Activity Info
Component Name
Package Name
new ComponentName(aInfo.packageName, aInfo.name)
componentName = _intent.getComponent()
ri.activityInfo = new ActivityInfo(ri.activityInfo);

Figure 2.2.: Example Residual Data Structures Generated During Activity Launch.

2.1.2 Identifying an Activity Launch from Allocations

As an Activity is launched, the transition from the previous Activity to the newly launched Activity is centrally handled by the *ActivityManagerService*. As such, the ActivityManagerService receives several RPCs (Remote Procedure Calls) when an Activity launch takes place. While executing the RPCs, the ActivityManagerService will allocate several key data structure “clusters” (i.e., a network of interconnected data structures), hereinafter referred to as the residual data structures of an Activity. An example of this can be seen in Figure 2.2, where a specific Activity’s Intent object is allocated as an RPC argument which links to other objects that are allocated through routine execution.

An important property of these residual data structures is that they are highly inter-connected. As the objects in residual data structures are allocated during the same Activity launch, they share a number of field values and are interconnected via pointers. This is highlighted in Figure 2.2, where the ActivityRecord, Intent, and ResolveInfo objects are required to share references (both directly and through their fields). There is another required value equivalence between the PackageName fields of ActivityRecord and ResolveInfo objects.

Another key property of these residual data structures is that they are organized as trees rooted at *application-generic* objects. This is noticeable in Figure 2.2, where the ActivityRecord, Intent, and ResolveInfo (application-generic) objects can be utilized to identify (and traverse) the entirety of the residual data structures. We call these top-most, application-generic data structures *root* data structures, and Timeliner utilizes 14 different root data structures to identify the residual data structures that are leftover from each Activity launch.

As shown in the sample size distribution in Figure 2.3, residual data structures are mostly composed of a large number of small allocations in thread-local runs, and a small number of large allocations in shared runs. Both types of allocations provide the spatial information utilized by Timeliner. While being limited to thread-local runs, the large number of allocations ensures that these allocations are spread out over several runs across various threads. Similarly, while fewer in number, the allocations in shared runs provide more robust spatial ordering. In this way, while RosAlloc's implementation includes both thread-local and shared runs, Timeliner's design is not dependent on it. We demonstrate this by extending Timeliner to another memory allocator (jemalloc) during our evaluation — which utilizes only thread-local runs.

Lastly, note that the lifespan of residual data structures is dependent on the Activity they represent. Further, the Activities belonging to the current Activity stack survive garbage collection. However, even those that survive get diminished in number. This is because many objects in the residual data structures are utilized only temporarily during Activity launch execution, making them candidates for garbage

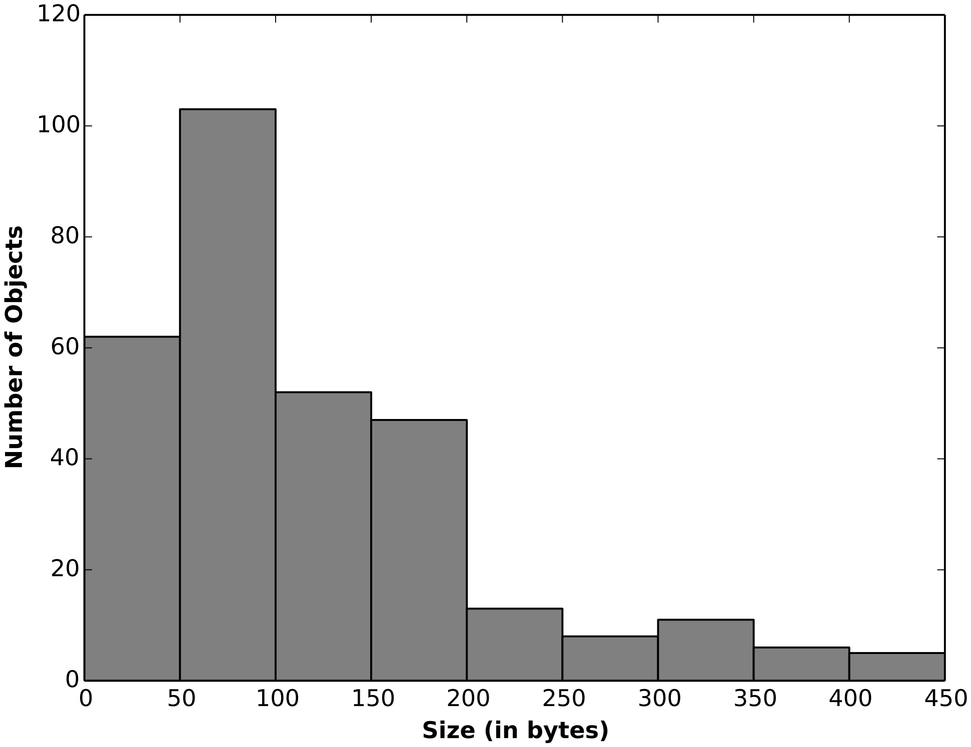


Figure 2.3.: Sample Size Distribution for Residual Data Structures.

collection. Despite their reduced numbers, these *diminished* residual data structures are still recoverable and identifiable (i.e., they reveal the original Activity's name). This allows Timeliner to identify Activities that occurred before the last garbage collection, which we call *garbage collected Activities*.

2.1.3 Inferring Temporal Ordering from Spatial Ordering

As described above, each Activity launch generates residual data structures, produced through the execution of the launch logic. Figure 2.1 provides an example of the allocations from four Activity launches, where all slots of the same color represent the residual data structures of a single Activity launch. The most important property to note from the layout of the residual data structures is that they are of varying allocation sizes and therefore occupy slots in multiple runs, both thread-local and shared.

In Section 2.1.1 we described how *within a single run* the temporal ordering of allocations results in a corresponding spatial ordering. This is the first step to inferring the temporal ordering of the residual data structures from two Activity launches — enabling Timeliner to solve for the above principle (temporal ordering leads to spatial ordering) in reverse. Timeliner applies the above principle across multiple runs and recovers the temporal ordering for a pair of Activity launches, which we refer to as the *transition* between two Activities. Timeliner models the transitions as a directed edge starting from a node for the former Activity and directed towards a node for the latter. The nodes and edges for all Activities recovered from a memory image are organized into a *transition graph*. Figure 2.1 includes an illustration of the transitions between various pairs of Activities.

A timeline of Activities that satisfies the transition graph should satisfy each edge individually, i.e. for every transition $u \to v$, from Activity u to Activity v, u should occur before v in the timeline. This ordering, known as the *topological ordering* of a graph, allows Timeliner to solve for the timeline by topologically sorting the transition graph.

Unfortunately, within individual runs, fragmentation of memory (i.e., a new allocation fills a slot before an existing allocation) can mislead Timeliner's reconstruction. This will result in an erroneous edge originating from one Activity and pointing to another. Handling these misleading (i.e., incorrect) transitions will require Timeliner to prune such edges from the transition graph. Figure 2.4 describes two examples of such pruning.

Notice also that Timeliner's transition graph can entirely miss edges which should exist. This can happen in two ways: (1) If the current set of runs becomes filled, then the spatial ordering information inferred from those runs (i.e., one transition between two Activities) is lost. As a result, the transition graph is partitioned into several connected components, the temporal orderings of which are termed *local orderings*.

Luckily, as the runs are chosen just like slots, i.e. with a "first-available" algorithm, the different local orderings can be joined later based on the spatial ordering of their runs into a single *global ordering*. (2) If two successive Activities do not share a common run, then there is no evidence of the transition between them. In this case, an ambiguity exists in their spatial ordering, and hence temporal ordering, leading to multiple possible timelines for those Activities. Hence, Timeliner needs to find all the topological orderings for the given transition graph. An example is shown in Figure 2.1, where there are two possible timelines.

2.2 Timeliner Design

Timeliner operates on only a single memory image from an Android device. From this memory image, Timeliner isolates and inspects the ActivityManagerService process's dynamic memory allocation space. Note that because Timeliner only relies on generic framework defined objects like Activities, Timeliner has no application-specific requirements in its design or implementation. Further, Timeliner's operation is entirely automated with no supervision required from an investigator — allowing it to be immediately deployable in practical investigations. In the remainder of this section, we will present the three phases of Timeliner's design.

2.2.1 Identifying Residual Data Structures

As Section 2.1.2 introduced, highly-interconnected sets of objects called residual data structures are left over from the execution of past Activity launches. Therefore, Timeliner must first recover objects and segregate them into residual data structures. This procedure is shown in Algorithm 1. As a running example, we shall recall the structures shown in Figure 2.2 throughout this section.

Timeliner first scans the input memory image to identify all objects previously allocated by the ActivityManagerService, whether still active or deallocated but waiting for garbage collection ("dead objects"). This step is accomplished with the help of the

Algorithm 1 Segregating Residual Data Structures.

```
Input: Object List $O$, RootClass List $Roots$
Output: ResidueObjectSet List $Residues$
                                        ▷ Identify and Add the Root Data Structures
  Object List $rootObjs \leftarrow \varnothing$
  for Object $o \in O$ do
      if $o.class \in Roots$ then
          $rootObjs \leftarrow rootObj \cup o$
                                        ▷ Segregate into different partial ResidueSets
  ResidueObjectSet List $PartialResidues \leftarrow \varnothing$
  for Object $rootObj \in rootObjs$ do
      ResidueObjectSet $newResSet \leftarrow rootObj$
                                        ▷ Match each root to identified partial ResidueSets
      for ResidueObjectSet $resSet \in PartialResidues$ do
          for Object $singleRes \in resSet$ do
              if $MATCH(rootObj, singleRes)$ then
                                        ▷ Merge partial residueSets for the same Activity
                  $newResSet \leftarrow newResSet \cup resSet$
                  $PartialResidues \leftarrow PartialResidues - resSet$
                  break
                                        ▷ Add the new partial residueSet back to PartialResidues
      $PartialResidues \leftarrow PartialResidues \cup newResSet$
                                        ▷ Recurse to get ResidueSets
  ResidueObjectSet List $Residues \leftarrow \varnothing$
  for ResidueObjectSet $resSet \in PartialResidues$ do
      ResideObjectSet $fullResSet \leftarrow RECURSE(resSet)$
      $Residues \leftarrow Residues \cup fullResSet$
```

runtime type information included in the managed runtime of Android (ART), which includes type information for objects and their fields. These are included in every process's memory space, so Timeliner can recover them directly. Note that Timeliner also recovers dead objects since Android's memory management is automatic and slots remain allocated until a garbage collection event. This list of recovered objects is given as the input to Algorithm 1.

Next, Timeliner parses this list of objects and identifies the root data structures. Defined in Section 2.1.2, these data structures are crucial for the identification of residual data structures (the "fingerprints" left by Activity launches). In Figure 2.2, we can see that the Intent, ActivityRecord, and ResolveInfo objects are three instances of such root data structures. An important point to note is that these root data structures are highly inter-connected, and as such, they can be used to segregate the recovered objects into distinct residual data structures.

This is exactly the approach used in Timeliner, as explained in Algorithm 1, the list of root data structures is segregated into distinct partial residual data structures. Note that these resultant residual data structures (named ResidueSets in Algorithm 1) are called partial, as they do not (yet) include the various non-root residual data structures reachable from the recovered instances of root data structures.

This segregation is affected by the "MATCH" function, which contains predefined application-generic relationships between the root data structures. In Figure 2.2, Intent and ActivityRecord are matched by their predefined ComponentName field values, and also a direct pointer from ActivityRecord to Intent. Further, ResolveInfo is linked to Intent via their predefined PackageName and ComponentName field values. Timeliner also leverages value equivalence in the PackageName fields of the ResolveInfo and ActivityRecord objects.

With these segregated partial residual data structures, Timeliner then recursively adds fields of the root objects that link non-root residual data structures from each Activity launch. This is represented in Algorithm 1 as the "RECURSE" function. When applied to the case presented in Figure 2.2, this would add any additional data structures reachable from the Intent, ActivityRecord, and ResolveInfo objects, leading to a full set of residual data structures (which represent an individual Activity launch).

After this step, Timeliner has obtained a list of Activities (whose launches create distinct residual data structures) and now needs to establish their temporal ordering. In the next section, Timeliner shall build a transition graph for these Activities.

2.2.2 Building the Transition Graph

As discussed in Section 2.1.3, two Activities are said to have a transition if they have a corresponding temporal ordering. Simply put, an Activity e has a transition to an Activity f if e is launched before f.

As noted in Section 2.1.1, the residual data structures are assigned slots spread across several runs. Further, recall from Section 2.1.3, that a single run can give misleading information due to fragmentation, and thus Timeliner utilizes multiple runs to infer transitions between two Activities, say Activity e and Activity f.

We define an Activity e as a set of pairs, where each pair consists of a run and a list of corresponding slots occupied by residual data structures of the Activity e. This can be represented as:

$$\begin{aligned} e = \{(r, s) \mid r \in \mathit{Runs} \wedge \\ s = \{i \mid r[i] \in \mathit{Residue}(e)\}\} \end{aligned} \tag{2.1}$$

where r is a Run and s is the list of indices of occupied slots for Activity e. The "Residue" function represents the residual data structures allocated during the launch of Activity e, which were identified in the previous section.

Identifying Transitions. An Activity e will have a transition to an Activity f if (1) they share runs where all allocations of e precede all allocations of f and (2) they do not share a run where the opposite ordering occurs, that is, any allocation of e succeeds any allocation of f. With these properties in mind, we define the following two functions, allPrecede and anySucceed, counting the common runs that have all allocations of e preceding allocations of f and those with any allocation of e succeeding allocations of f, respectively.

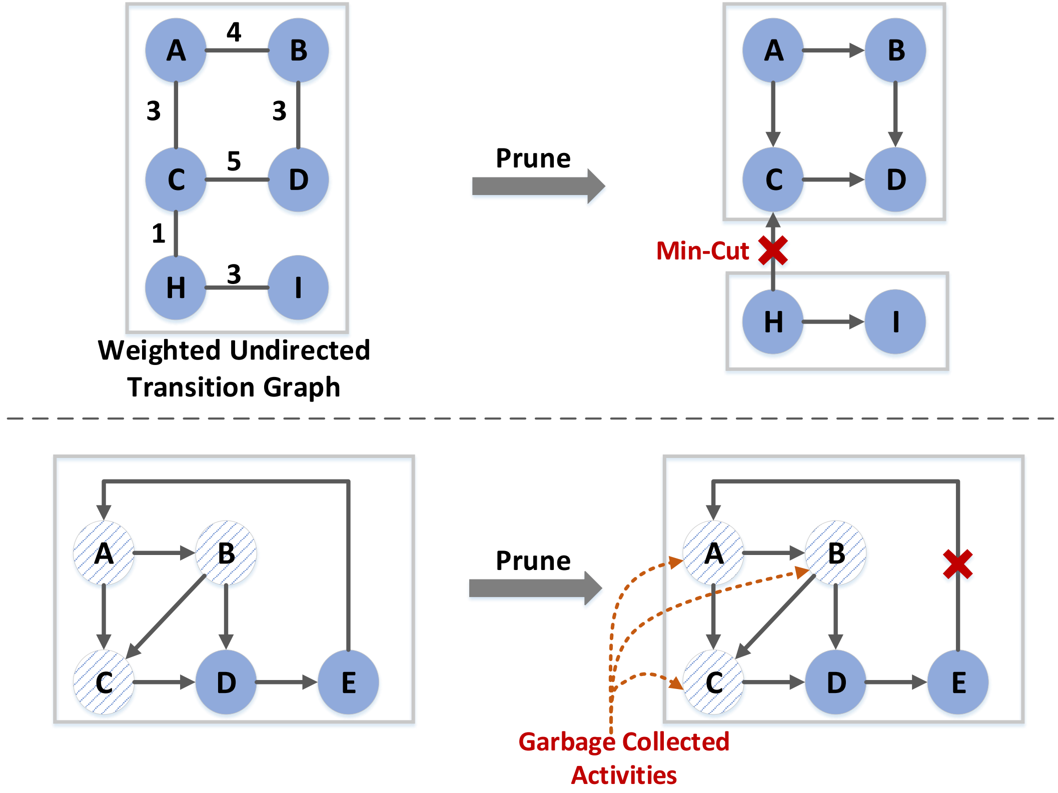

A
B
C
D
H
I
4
3
3
5
1
3
Weighted Undirected Transition Graph
Prune
A
B
C
D
Min-Cut
H
I
A
B
C
D
E
Prune
A
B
C
D
E
Garbage Collected Activities

Figure 2.4.: Prune Erroneous Edges from Transition Graph.

$$\begin{aligned} allPrecede(e,f) &= |\{r \mid (r,m) \in e \ \wedge \ (r,n) \in f \ \wedge \\ &\qquad \max(m) < \min(n)\}| \\ anySucceed(e,f) &= |\{r \mid (r,m) \in e \ \wedge \ (r,n) \in f \ \wedge \\ &\qquad \max(m) > \min(n)\}| \end{aligned} \tag{2.2}$$

where e and f are Activities, r is a common run, and m and n are lists of slots in the common run r. There exists a transition between e and f if allPrecede(e,f) is positive and anySucceed(e,f) is equal to zero. Timeliner organizes these Activities as nodes and transitions as edges in a graph, called the *transition graph*.

Assigning Weights to Transitions. The higher the number of shared runs between two Activities (while maintaining the correct ordering), the higher the confidence in the transition, as the odds of coincidentally sharing runs decreases exponentially with the number of shared runs. Hence, if a transition exists, we assign the transition from Activity e to Activity f a weight equal to the number of shared runs between the two Activities. Note that the number of common runs is equal to allPrecede(e,f) + anySucceed(e,f), however as anySucceed(e,f) is equal to zero for a transition, the weight a transition is assigned is equal to allPrecede(e,f).

Pruning the Transition Graph. As noted in Section 2.1.3, due to fragmentation, it is possible for a new allocation to fill up a slot before a pre-existing allocation. This means that an erroneous spatial ordering exists from the new allocation to the pre-existing one, and this implies a transition edge from a new Activity to a much older one. Clearly, this erroneous edge in our transition graph can lead to a wrong temporal ordering and must be pruned.

To handle such erroneous transitions, we use the observation that while it is possible for a new Activity to reuse a run and share it with an older Activity, it is improbable for them to share two runs, and unlikely to share three. Hence, we assume a maximum weight of two for such edges. Further, such an erroneous edge can be classified as one of the following two cases:

1. **Transition between two connected components:** The edge connects two connected components, which contain legitimate transitions and are highly inter-connected. In other words, such an edge serves as the *minimum cut* in the graph, and as discussed above, is limited to a maximum weight of two. An example can be seen in the top half of Figure 2.4, where there is an erroneous edge pointing from node H to node C, serving as the minimum cut between the two sub-graphs. Pruning the graph in the example leads to two connected components, one with nodes $\{A, B, C, D\}$ and the other with nodes $\{H, I\}$.

   To implement this, Timeliner utilizes a min-cut algorithm (Stoer-Wagner [31]) on the corresponding undirected version of the transition graph, running the algorithm for each connected component, as explained in Algorithm 2. Assuming that there are n Activities and hence $O(n^2)$ edges, the time complexity of the algorithm is $O(n^3)$.

2. **Transition in the same connected component:** The edge connects two nodes in the same connected component, creating a cycle. Note that this can only happen if a garbage collection event happens between the Activities covered in the transition subgraph, allowing a new Activity to re-use a run used by an older Activity.

   Timeliner utilizes this property to remove the erroneous edge, pointing from an Activity launched after the garbage collection event, to an Activity launched before. As noted in Section 2.1.2, the Activities that occurred before the last garbage collection event, called garbage collected Activities, can be identified by their diminished residual data structures. An example can be seen in Figure 2.4, with the erroneous edge pointing from node E to node A removed from the transition graph. The complexity of this algorithm is linear with respect to the number of edges.

In the next section, Timeliner utilizes this pruned transition graph and reconstructs the device-wide sequence of Activities, which we call the *timeline*.

2.2.3 Reconstructing the Global Ordering for Activities

Before we describe the procedure of finding the temporal ordering, it is important to prove the existence of one. As discussed in Section 2.1.3, Timeliner solves for the topological ordering for a given transition graph. It is a known property of directed graphs that a topological ordering exists if and only if it is acyclic. While it is obvious that a graph with a cycle cannot be topologically ordered, it can also be proven that a depth first search in an acyclic graph (where a node is processed after its children are processed) leads to a reverse topological ordering. Hence, proving the existence of a topological ordering is equivalent to proving that the directed graph

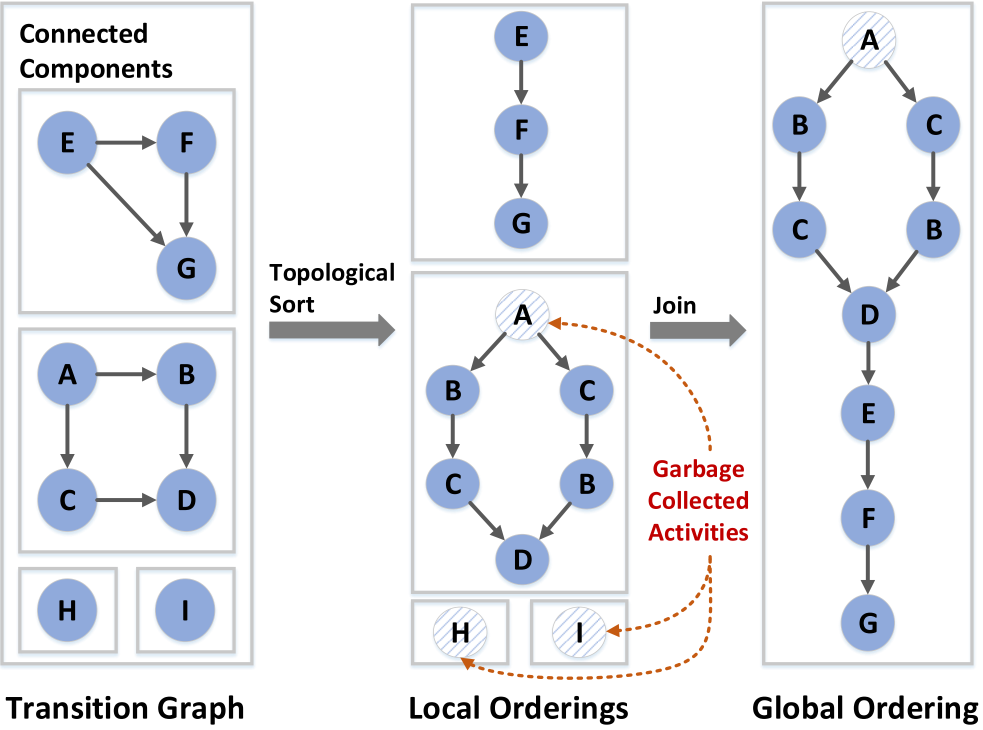


Figure 2.5.: Deriving the Global Ordering from the Transition Graph.

is acyclic. In Timeliner's transition graph, because of the "first-available" memory allocator algorithm, cycles can only be erroneous. Therefore, the acyclic property of the transition graph is guaranteed by the pruning described in Section 2.2.2.

Local Orderings. Given a list of Activities recovered in Section 2.2.1, Timeliner topologically orders the transition graph from Section 2.2.2 to establish various local orderings of Activity launches. However, as discussed in Section 2.1.3, there are possible ambiguities in the temporal orderings of Activities, causing multiple possible timelines. An example can be seen in Figure 2.5, with the local ordering of nodes $\{A, B, C, D\}$ having two possible solutions. To effectively compute all the solutions, Timeliner performs a depth-first search with backtracking. The complexity of this algorithm is O(numTimelines*n^2), assuming n Activities and numTimelines solutions of temporal orderings.

Algorithm 2 Reconstruct the Global Ordering.

Input: Graph *Transitions*

Output: Graph *Timeline*

```
                                        ▷ Remove Erroneous Edges - O($n^3$)
GraphList $Components \leftarrow Transitions_{components}$
for Graph $g \in Components$ do
    $(cutSize, cutEdges) \leftarrow$min-cut($undirected(g)$)
    if $cutSize < 3$ then
        $Transitions \leftarrow Transitions - cutEdges$
    if $HasCycle(g)$ then
        for Transition $(e \rightarrow f) \in g$ do
            if $not\ IsGarbageCollected(Residue(e))$ then
                if $IsGarbageCollected(Residue(f))$ then
                    $g \leftarrow g - (e \rightarrow f)$
                        ▷ Topologically Sort into Local Orderings - O(numTimelines*$n^2$)
GraphList $LocalOrderings \leftarrow \varnothing$
GraphList $Components \leftarrow Transitions_{components}$
for Graph $g \in Components$ do
    $LocalOrderings \leftarrow LocalOrderings \cup$ topological-sort($g$)
                                        ▷ Identify Joinable Local Orderings - O($n$)
GraphList $JoinableOrderings \leftarrow \varnothing$
for Graph $g \in LocalOrderings$ do
    Activity $a \leftarrow g.lastActivity()$
    if $not\ IsGarbageCollected(Residue(a))$ then
        $JoinableOrderings \leftarrow JoinableOrderings \cup g$
                                ▷ Join Local Orderings into Global Ordering - O($n$)
Graph $Timeline \leftarrow \varnothing$
$JoinableOrderings.lastActivity().sort()$
for Graph $g \in JoinableOrderings$ do
    $Timeline.append(g)$
```

Table 2.1.: List of Some Applications and a Few Example Activities.

Application	Activities			
WhatsApp	HomeActivity	Conversation	VoipActivity	RecordAudio
	CameraActivity	MediaGallery	ProfileActivity	VoiceMessaging
WeChat	LauncherUI	ChattingUI	AlbumPreviewUI	VideoActivity
	SelectContactUI	ContactInfoUI	GroupCardSelectUI	NearbyFriendsIntroUI
Signal	ConversationListActivity	ConversationActivity	RedPhone	NewConversationActivity
	GroupCreateActivity	ContactSelectionActivity	ShareActivity	SmsSendToActivity
Skype	HubActivity	PreCallActivity	ContactDirectorySearch	ContactProfileActivity
	ContactEditActivity	ContactDetail	ContactAddNumber	AddParticipantsActivity
Messaging	ConversationListActivity	ConversationActivity	WidgetReplyActivity	PeopleAndOptionsActivity
	ApplicationSettingsActivity	ShareIntentActivity	WidgetPickConversation	VideoShareActivity
Dialer	InCallActivity	CallLogActivity	CallDetailActivity	PeopleActivity
	QuickContactActivity	BlockedNumbersActivity	ImportVCard	CallSubjectDialog
Chase	AccountsActivity	BillPayAddStartActivity	BillPayAddVerifyActivity	BillPayHistoryActivity
	QuickPayChooseRecipient	TransferActivity	QuickDepositStartActivity	FindBranchActivity
Gmail	ConversationListActivity	ComposeActivityGmail	AccountSetupFinalActivity	GmailPreferenceActivity
Facebook	SplashScreenActivity	PickerLauncherActivity	ComposerActivity	FbMainTabActivity
File Browser	FileBrowserActivity	TaskProgressActivity	FileConverterActivity	HttpServerActivity
Netflix	HomeActivity	SearchActivity	ShowDetailsActivity	PlayerActivity

Joinable Local Orderings. As described in Section 2.1.1, Timeliner uses the property that runs are allocated via a "first-available" algorithm. This implies that, just like allocation slots, the runs for different local orderings are spatially ordered. However, this spatially-increasing ordering does not always hold true, because of garbage collection events. A garbage collection event frees up low memory runs that are used by future Activities, causing a backward jump in the spatial ordering. Hence, Timeliner only joins those local orderings whose last Activities occur after the last garbage collection event. Such *joinable* local orderings can be distinguished by identifying garbage collected Activities as discussed in Section 2.1.2. An example can be seen in Figure 2.5, where the two local orderings with nodes $\{A, B, C, D\}$ and $\{E, F, G\}$ are joinable, even though node A is a garbage collected Activity. Note that while garbage collection makes it difficult to order garbage collected Activities, it does not entirely prohibit it. For example, a local ordering that includes Activities that span a garbage collection event, from both before and after the event, is a joinable local ordering.

Global Ordering. To join multiple joinable local orderings into a single global ordering, Timeliner first identifies the Activities that were launched after the last garbage collection, as explained in Section 2.1.2. Then Timeliner joins the local orderings which end with these Activities, following the spatial ordering of the runs that hold their allocations, yielding the global ordering. For example, in Figure 2.5, we see that the local orderings with nodes $\{A, B, C, D\}$ and $\{E, F, G\}$ are joined into a global ordering. The joining of local orderings into a global ordering has a complexity linear with respect to the number of Activities.

The resultant global ordering is returned by Timeliner to the investigators as the device-wide sequence of user actions.

2.3 Timeliner Evaluation

Timeliner is implemented as a plugin for the AOSP (Android Open Source Project) and executes within an Android emulator, utilizing ART's runtime environment to identify crucial data structures for the memory allocator. Timeliner also reuses ART's various libraries to automatically parse and process the definitions of the residual data structures stored in the input memory image.

Setup. Timeliner is evaluated across 3 commercially available smartphones (Samsung Galaxy S4, LG G3 and Motorola Moto G3) using a variety of different applications. These include messaging apps such as WhatsApp, WeChat, Signal (widely renowned for security), each vendor's Messaging app, voice and video telephony apps like the vendors' Dialer apps and Skype. We also include email applications such as Gmail, the Chase Banking personal banking app, a video streaming app (Netflix), a social network app (Facebook), and various utility apps such as File Browser, Downloads, PDFReader, Camera, and Google Maps.

Table 2.1 lists a small subset of the Activities that are present in some different apps that we used in our evaluation. As Table 2.1 shows, the names of these Activities are very descriptive of the user actions they represent. For example, even within sophisticated apps like Signal, we can see Activities such as *ConversationListActivity* and *ConversationActivity* which describe viewing a list of past conversations versus clicking into a single conversation. Representing the action of making a voice/video call, we see the *VoipActivity* in WhatsApp, *VideoActivity* in WeChat, *RedPhone* (making a secure phone call) in Signal, *PreCallActivity* in Skype, and *InCallActivity* in Dialer. Even fine-grained app-specific actions can be captured, such as *ComposeActivityGmail* for composing an email, *BillPayAddStartActivity* for initiating a bill payment, and *QuickDepositStartActivity* for starting a check deposit. These vivid descriptions are due to the fact that Activities serve as intuitive abstractions for user actions.

However, the most important information for a criminal investigator is not just isolated user actions but the complex *sequencing of Activities*. For example, a *FileBrowserActivity* followed by a *TaskProgressActivity(delete)* showcases that the user deleted a file. This interplay of activities can be used to develop the timeline of a crime: For example, a user first takes a photograph with the *CameraLauncher* Activity in the camera app, followed by sharing the photo via WhatsApp's *ChooserActivity(share)* and *ImagePreview* activities. Finally, the user opens the photo via the *FileBrowserActivity* and deletes it with the *TaskProgressActivity(delete)* Activity. This semantically-meaningful series of user actions can be essential for quickly focusing a developing criminal investigation.

2.3.1 Garbage Collection

Garbage collection events can clear the allocations of Activities that are not alive (on the Activity stack) and hence do not have a reference towards them, causing limited evidence recovery. Further, garbage collection can also break the spatially-increasing ordering (due to the "first-available" algorithms) by causing a jump from high to low memory addresses as those runs become available. Hence, garbage collection events lead to (1) loss of evidence and (2) partial loss of spatial ordering for the remaining evidence in memory. However, note that while garbage collection makes it difficult to order garbage collected Activities, they can still be ordered if they are a part of some joinable local ordering.

To understand the limitations on Timeliner due to garbage collection, we begin the evaluation of Timeliner by evaluating (1) the frequency of garbage collection and (2) Timeliner's recovery after garbage collection events.

We first evaluate garbage collection frequency across different devices and under different usage conditions, while aiming to measure the time duration between different garbage collection events. To do so, we instrumented and measured the frequency of garbage collection on the *ActivityManagerService* process (the subject of Time-

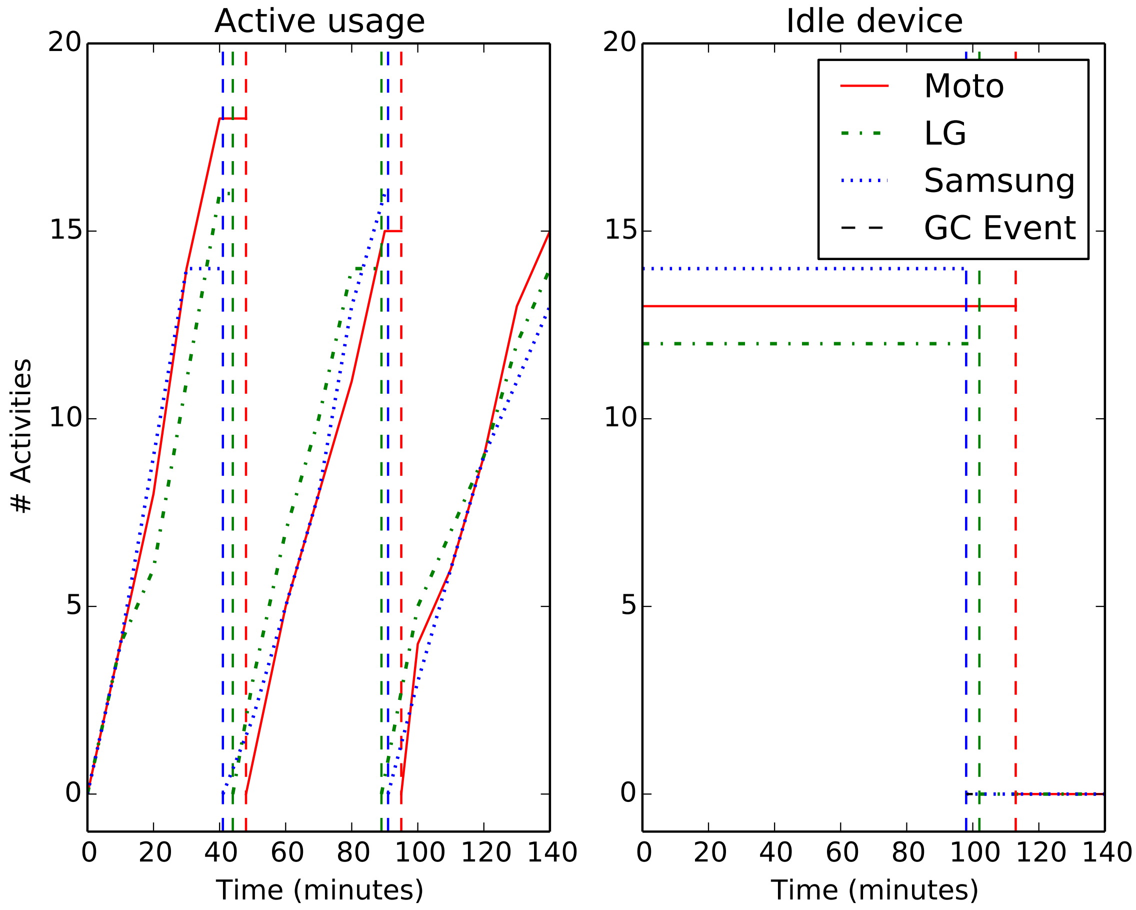


Figure 2.6.: Duration Between Garbage Collection Events Across Three Devices During Active and Idle Usage.

liner's reconstruction). Note that this is a service provided by the Android framework, which is largely unaffected by the processing done in any application. Hence, while certain applications are quite memory intensive (causing heavy workload of allocations), the frequent garbage collections are limited to their own processes.

This garbage collection profiling was carried out under two different conditions: (1) the phone was left idle and (2) the phone underwent constant user activity. For this purpose, we first installed all the applications listed in Table 2.1 on the three devices. For the idle case, we turned the device's screen off and left the phone idle. However, due to the presence of events raised by various background services (e.g., the AlarmManager service) the memory usage of the ActivityManager process slowly increases with time. For the active usage case, we repeatedly followed the sequence of Activities shown in Set A listed in Table 2.3, raising a new Activity every two to three minutes.

Figure 2.6 presents our profiling results across the three devices under each usage condition, with Activities raised since last garbage collection event shown for each device. In the active usage case, garbage collection events were triggered periodically after 41-50 minutes (that followed raising 14 to 18 Activities). Interestingly, even after repeated triggering of the garbage collection events, this period remained roughly the same. This suggests a stable heap size and memory usage pattern. In the idle case, garbage collection events were triggered after 98 to 113 minutes, again due to the slower increase in memory usage within the *ActivityManagerService* process when the device is idle.

To further confirm Timeliner's recovery under garbage collection, we profiled garbage collection events and Activity launches on the Motorola device. During this time, we captured memory snapshots every 10 minutes (as any one of those could be the one taken when investigators confiscate the device) and use Timeliner to recover the Activities in the memory image. The results are detailed in Table 2.2.

As expected, we can see that the Activities recovered by Timeliner include the set of Activities launched since the last garbage collection event. Timeliner also recovered Activities that survived garbage collection. Out of these Activities that survived the garbage collection event, a few of them also get ordered as they were part of a joinable local ordering. Note that there are two garbage collection events because of memory allocation requests (GC_FOR_ALLOC) at 44 and 82 minutes, respectively.

In a criminal investigation, a device's past usage will blend both idle and active periods, but in either case this is an ample time window to capture the details of a crime carried out on a smartphone.

Table 2.2.: Recovery by Timeliner under Garbage Collection.

Time (minutes)	Total Activities	Activities Since GC	Activities Recovered	Activities Ordered
0	1	0	1	0
10	8	7	8	7
20	11	10	11	10
30	15	14	15	14
40	17	16	17	16
GC_FOR_ALLOC at t = 44 minutes				
50	23	6	8	7
60	28	11	13	12
70	31	14	16	15
80	34	17	19	18
GC_FOR_ALLOC at t = 82 minutes				
90	41	7	10	8
100	46	12	15	13
110	50	16	19	17

Table 2.3.: Timeline Reconstruction for Micro-Benchmark Experiments across Different Devices.

Device	Experiment Set	Activity Count	Duration (Minutes)	Activities Recovered	Activities Ordered	Root Structures	Residual Structures	Local Orderings	Number of Timelines	Kendall-Tau Distance
Samsung S4	Set A	15	39	17	16	91	6526	4	1	0
	Set B	13	37	14	14	77	5584	2	1	0
	Set C	16	51	18	16	95	6218	5	1	0
	Set D	12	22	13	12	65	4881	4	1	0
	Set E	13	34	15	14	81	5079	3	1	0
	Set F	15	45	16	15	86	6049	5	1	0
LG G3	Set G	14	38	16	15	85	5427	5	1	0
	Set H	16	42	18	16	94	6395	4	1	0
	Set A	15	33	17	16	92	6241	4	1	0
	Set I	14	28	16	14	83	5280	5	1	0
	Set J	15	37	16	16	97	6429	3	1	0
	Set B	13	28	15	14	78	5227	3	1	0
Moto G3	Set G	14	35	15	14	83	5016	4	1	0
	Set D	12	28	15	13	70	4589	3	1	0
	Set C	15	57	16	15	88	5782	5	1	0
	Set I	14	39	16	14	87	5296	4	1	0
	Set A	15	41	17	16	85	5721	5	1	0
	Set H	14	48	15	14	83	5192	3	1	0

Table 2.4.: Experiment Sets for Micro-Benchmarks.

Experiment Set	Sequence of Activities (Randomly Chosen for Micro-Benchmarks)
Set A	Launcher → WhatsApp{ Conversation → HomeActivity → Conversation → VoipActivity → Conversation → CameraActivity → Conversation } → Launcher → Gmail{ ConverastionListActivity → ComposeActivityGmail } → Launcher → Downloads{ FilesActivity → ShareActivity } → Launcher
Set B	Launcher → Skype{ HubActivity → ContactDirectorySearch → ContactProfileActivity → ContactEditActivity } → Launcher → Dialer{ CallLogActivity → CallDetailActivity → PeopleActivity → InCallActivity } → Launcher → Netflix{ HomeActivity → PlayerActivity }
Set C	Launcher → WeChat{ LauncherUI → ChattingUI → VideoActivity → SelectContactUI → SingleChatInfoUI → ContactInfoUI } → Launcher → Signal{ ConversationList → Conversation → GroupCreate } → Launcher → Signal{ GroupCreate → ContactSelection } → Launcher → Gmail{ ConversationList → GmailPreference } → Launcher
Set D	Launcher → Messaging{ ConversationList → Conversation } → Launcher → Messaging{ Conversation → PeopleAndOptions } → Dialer{ InCallActivity } → Launcher → Chase{ AccountsActivity → BillPayAddStartActivity → BillPayAddVerifyActivity } → Launcher
Set E	Launcher → Netflix{ HomeActivity → SearchActivity → ShowDetailsActivity → PlayerActivity } → Launcher → WhatsApp{ HomeActivity → GroupMemberSelector → NewGroup → Conversation } → Launcher → Downloads{ FilesActivity } → Launcher
Set F	Launcher → Signal{ ConversationListActivity → ConversationActivity → DocumentsActivity → RedPhone } → Launcher → Skype{ HubActivity → PreCallActivity → HubActivity → ContactProfileActivity } → Launcher → Chase{ AccountsActivity → BillPayHistoryActivity → QuickPayTodoListActivity } → Launcher
Set G	Launcher → Gmail{ ConverastionListActivity → ComposeActivityGmail } → Launcher → Dialer{ CallLogActivity → PeopleActivity } → Launcher → Dialer{ PeopleActivity → BlockedNumbersActivity → InCallActivity } → Launcher → Downloads{ FilesActivity → UploadActivity } → Launcher
Set H	Launcher → Chase{ HomeActivity → AccountsActivity → AlertsHistoryActivity → FindBranchActivity → LocationInfoActivity → AccountsActivity } → Launcher → Dialer{ CallLogActivity → CallDetailActivity → PeopleActivity } → Launcher → WhatsApp{ HomeActivity → Conversation → VoipActivity → CameraActivity }
Set I	Launcher → Skype{ HubActivity → PreCallActivity → ContactProfileActivity } → Launcher → Signal{ ConversationListActivity → ConversationActivity → RedPhone → NewConversationActivity → ConversationActivity → DocumentsActivity } → Launcher → Gmail{ ConversationListActivity → GmailPreferenceActivity }
Set J	Launcher → WhatsApp { HomeActivity → GroupMemberSelector → NewGroup → Conversation } → Launcher → Skype{ HubActivity → ContactDirectorySearch → ContactProfile → ContactEdit } → Launcher → Netflix{ HomeActivity → SearchActivity } → Launcher → Netflix{ SearchActivity → ShowDetailsActivity → PlayerActivity }

2.3.2 Micro-Benchmarks

To evaluate Timeliner's reconstruction capability, this section presents micro-benchmark results measured during Timeliner's recovery across a variety of memory images. For this recovery, the authors interacted with the sets of Activities described in Table 2.1. The activities in the applications were launched following one of ten random sequences, the exact sequences of launches are detailed in Sets A through J in Table 2.3. We performed six experiments on each of the three devices, each experiment using a different activity sequence taken from the defined ten random sequences. Each Activity in a sequence is started and left on the screen for a varying amount of time, around two to three minutes. To mitigate the effect of garbage collection on these micro-benchmarks, we initiated a garbage collection before starting each experiment. Memory images were captured by a custom handler invoked at the next garbage collection event, implemented by instrumenting the internal garbage collection event handler.

To verify the accuracy of Timeliner's recovery, we compared the reconstructed timeline with the original list of Activities. The ground truth about Activity launches was captured by profiling the *ActivityManagerService* process. We stored the addresses of the allocated Activity-launch related data structures along with the original timeline of the activities. The allocated data structures were stored to correctly identify recovered activities from the original sequence and this timeline was used to verify Timeliner. Note that Timeliner did not need nor have access to this ground truth information and reconstructed Activity timelines completely oblivious to our external measurement.

Table 2.3 provides a summary of the micro-benchmark results from these experiments. The first column shows the device the experiment was run upon and the experiment set used, followed by *Activity Count* and *Duration* of the ground truth timeline. The fourth and fifth columns list the number of Activities recovered and the number of Activities ordered by Timeliner. Next two columns present recovery metrics: the total number of roots and residual data structures recovered. The eighth and the ninth columns show the number of local orderings recovered and the number of possible timelines in the global ordering and the last column compares the original ground truth to the recovered timeline (minus Activities not in the ground truth) via *Kendall-Tau* distance [32]. Kendall-Tau distance compares two ordered lists and calculates the number of pairwise disagreements between them. This is a good measure for a timeline as the more displaced an activity is from its correct position, the higher the Kendall-Tau distance (therefore a minimal distance value is best). Finally, the exact sequence of Activities in the 10 experiment sets is presented at the bottom of Table 2.3.

First, from Table 2.3, observe that for some cases like Set C in Samsung, Set J in LG, and Set C and Set H in Motorola, the Activity count in the experiment is less than the number of Activities in the experiment set. This is because in these cases, a garbage collection event was triggered before the sequence of Activities was finished. Hence, a smaller set of Activities is taken from the experiment set. For other cases, if the sequence was finished without a garbage collection event, one was triggered manually.

Even though garbage collection is triggered manually in some cases, the workload in Table 2.3 is quite similar to the one in Section 2.3.1. For example, the number of Activities in the original timeline varies from 12 to 16 with an average of 14.16 Activities per experiment, similar to what was observed in Section 2.3.1. Similarly, the time duration varies from 22 to 57 minutes with an average of 37.88 minutes, again similar to the observed results in Section 2.3.1.

The results of the micro-benchmarks are also quite similar to the results from Section 2.3.1. For example, just like in Table 2.2, Timeliner recovers more Activities than the ones that were raised after the last garbage collection event because some Activities will survive garbage collection. While not all Activities can be ordered because of loss of spatial and temporal information, some local orderings are joinable. This allows Timeliner to order more Activities than were in the experiment set. For the micro-benchmarks, Timeliner recovers 13 to 18 Activities with an average of 15.83 Activities per experiment. The global orderings generated by Timeliner contain 12 to 16 Activities with an average of 14.67 Activities per experiment.

Looking at the data structure metrics, we see that Timeliner recovers an average of 84.44 root data structures per experiment, which equals 5.33 roots per recovered Activity. Similarly, Timeliner recovers an average of 5607.33 residual data structures per experiment, which leads to 354.15 residual data structures per recovered Activity. These averages are roughly constant across the various experiments, implying that the residual data structures are (roughly) application-generic.

We also compare the metrics across the three devices. On average, the Samsung device yields 14.33 Activities over 38 minutes, while LG has 14.83 Activities over 34.33 minutes, and Motorola has 14.83 Activities over 41.33 minutes. We also observe that (per-Activity) the Samsung device yields an average of 369.21 residual data structures, LG has an average of 357.13 residual data structures, and Motorola an average of 336.12 residual data structures, which follow the earlier observations that there are roughly similar number of residual data structures per activity even across devices. The similarity of these results gives us confidence that vendor-customizations rarely affect both low-level primitives of memory allocation and application-generic residual data structures.

Finally, we compare the metrics that pertain to ordering, namely local orderings, number of possible timelines in the global ordering and the Kendall-Tau distance. As we can see, we get a few local orderings, varying from 2 to 5 for different experiments. From the other two metrics, it is visible that Timeliner is highly successful in ordering the Activities. Not only is Timeliner highly precise, with 1 unique timeline in the global ordering of every experiment, it is also highly accurate with the Kendall-Tau distance for all the experiments being equal to zero. In other words, Timeliner is able to perform perfect recovery of the Activity timeline.

The accuracy of Timeliner, while surprising, is intuitive as there are no spatial (and hence temporal) ambiguities because of the following two properties: (1) application-generic residual data structures contain a large number of objects spread across shared and thread-local runs, ensuring unambiguous spatial ordering, and (2) a complete global ordering of the Activities after the last garbage collection event is ensured by their local orderings being joinable (because of the "first-available" algorithms).

Next, we show that Timeliner's design is generic and applicable across various Android versions and even other memory allocators.

2.3.3 Design Generality

While Timeliner is implemented within a specific Android platform, Timeliner's *design and operation* is generic, and Timeliner is immediately applicable across the newest and most widely used Android versions. The devices in this evaluation all use different Android platforms: the Samsung running Android 5.0, LG running Android 5.1, and Android 6.0 running on the Motorola. These versions comprise 61.5% of the current Android devices and represent a wide variety of available Android smartphones [33].

Timeliner's generic design is due to a robust set of root data structures used to identify the residual data structures. The same set of root data structures is highly efficient because the Activity launch logic is similar across various Android versions. Further, Timeliner can also be applied to memory allocators other than RosAlloc, as many other memory allocators also perform "first-available" allocations.

Extension to jemalloc. jemalloc is a memory allocator widely used across products such as Firefox, Cassandra, Redis, among many others [34]. Our investigation reveals that jemalloc (without thread caching) also utilizes a similar design to RosAlloc, with a "first-available" algorithm for memory allocation. In particular, we discuss the design and extension of Timeliner to mozjemalloc [35], a modified implementation of jemalloc used across various Mozilla products such as Firefox and Thunderbird. The following discussion is based on the mozjemalloc version bundled as the default memory allocator in Firefox 55.0.

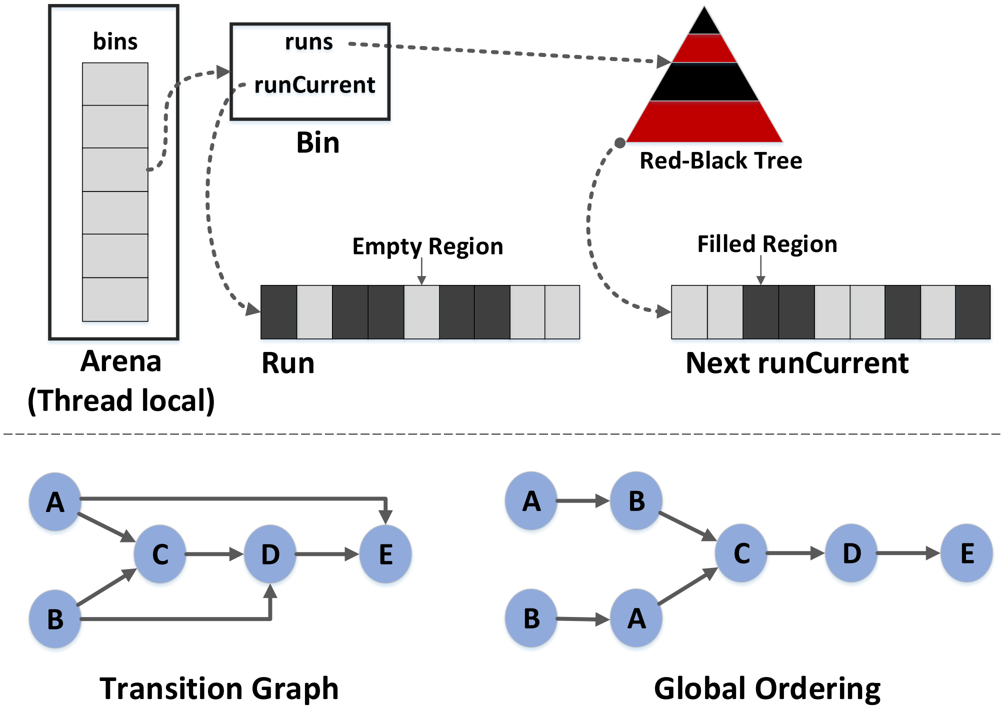


Figure 2.7.: mozjemalloc Design and Simulated Results.

The design of mozjemalloc is shown in Figure 2.7, with allocations stored in *Regions* (slots in RosAlloc) and regions of same size organized in *Runs*. Runs with allocations of the same size are placed in bins, and bins are placed in a (thread-local) *Arena*. Each bin has a current run to allocate from, with the rest of the runs (that have free Regions) organized in a red-black tree. The algorithms utilize a bitmap for "first-available" Region allocation. New runs are also allocated in accordance to a "first-available" algorithm, utilizing a red-black tree.

To evaluate Timeliner's recovery on mozjemalloc, we simulate Activity launches by following the allocation size distribution from Figure 2.3, spread out evenly across two threads in mozjemalloc. We simulated five Activity launches on mozjemalloc, which was initialized with five threads, and inferred their transitions with Timeliner—applying the same spatial-temporal principle from before. Timeliner constructed and topologically sorted a transition graph for the Activities in order to reconstruct the global ordering shown in Figure 2.7. As the figure illustrates, there was an ambiguity in the order of Activities, with the two possible timelines shown.

While Timeliner is designed to recover user actions (Activities), it is not limited to them. We extended Timeliner to recover and order app actions (*BroadcastReceiver* callbacks on system events) carried out through Intents and BroadcastRecord objects. These interactions are lightweight and do not produce the plethora of data structures created by Activities. As such, we end up with a small number of residual data structures which are also limited to allocations in the smaller-sized thread-local runs. This implies that for two Broadcasts, there can be a spatial ambiguity similar to what was observed in mozjemalloc. Note that as an Activity generates residual data structures spread across various threads, making it very likely that an Activity and a Broadcast share several thread-local runs, it enables Timeliner to infer spatial ordering between an Activity and a Broadcast. The next section demonstrates this in our case study of a spyware attack investigation.

2.3.4 Case Study: Spyware Attack Investigation

Being the most widely used smartphone platform, Android has increasingly been targeted by various sophisticated spyware attacks [36, 37]. Spyware has recently been employed by nation states targeting journalists and activists [38] and even by abusive spouses to monitor their families [39]. Modern spyware are extremely stealthy and sometimes do not even require physical access for installation, relying on drive-by downloads and vulnerabilities. These spyware track the victim's calls, texts, app usage, and smartphone features such as keyboard inputs, location, microphone, and camera. In this case study, we examine the capability of Timeliner to recover the actions of TheOneSpy [40], a commercially available spyware application.

Unknown to the victim ("John"), his smartphone has been infected with TheOneSpy. While on his way to a confidential meeting, John receives a text reminder for

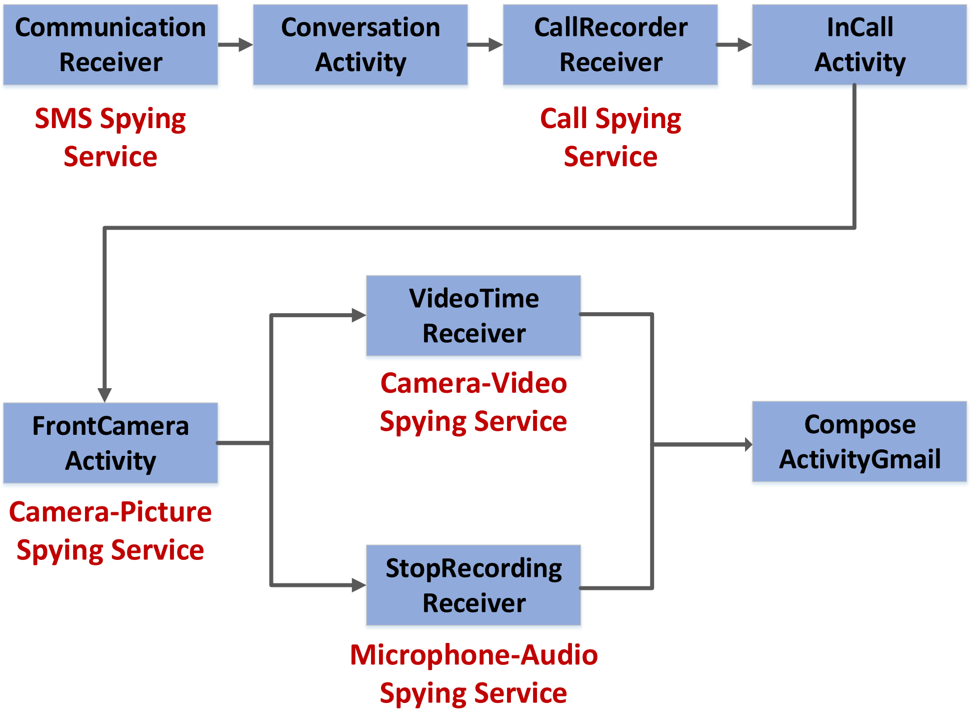


Figure 2.8.: Recovered Transition Graph for Spyware Attack.

the meeting and an incoming call for the meeting location. During the meeting, John receives an email for which he initiates a response. However, he notices that the keyboard has been changed from the default (Android Keyboard) to a custom one, which is visually differentiable. A quick investigation in the smartphone's settings reveals that the custom keyboard is the spyware's keylogger, and the spyware has access permissions for the microphone and camera. To confirm the spyware's activity, a memory image is taken and analyzed with Timeliner.

For this case study, we consider only those Activities and Broadcasts that are relevant to the spyware application. Timeliner recovers 9 Activities/Broadcasts with 26 roots and 1638 residual data structures, with one occurring before the last garbage collection. Two local orderings are combined to form a single global ordering, with two possible timelines. Note that the Kendall-Tau distance of the two timelines are 0 and 1 — as the ground truth is one of the possible timelines.

The deduced transition graph is shown in Figure 2.8. The two possible timelines are shown in Table 2.5 starting with a Broadcast receiver *CommunicationReceiver* on the spyware and the user opening a text with *ConversationActivity*. Following these is another Broadcast receiver *CallRecorderReceiver* and the user answering the call with *InCallActivity*. Next, there are a few spyware Activities/Broadcasts: *FrontCameraActivity*, *VideoTimeReceiver*, and *StopRecordingReceiver*. Finally, there is a *ComposeActivityGmail* Activity as the user replies to the email. While the names are quite verbose, a quick look at the spyware bytecode confirms that *CommunicationReceiver* and *CallRecorderReceiver* are used for spying on incoming texts and calls, respectively. *FrontCameraActivity*, *VideoTimeReceiver*, and *StopRecordingReceiver* are used for remotely recording pictures, videos, and audio, respectively. The confirmation of the spying activity makes it highly likely that the secrecy of the meeting had been compromised.

Finally, note that while there is a spatial and hence a temporal ambiguity between the two Broadcasts, Timeliner still establishes a sufficient evidence of the meeting being compromised — as the remote video recording and audio recording is contained between the phone call and the received email. Both timelines are shown in Table 2.5.

Next, we show the application of Timeliner in a few crime scenarios where the culprit is a human (instead of spyware).

2.3.5 Case Study: Military Espionage

Timeliner is particularly useful in investigating misuse of mobile devices in secured environments such as a Sensitive Compartmented Information Facility (SCIF) where personal mobile devices are not allowed and commonly, a locker is provided outside the SCIF where mobile phones can be secured, or they are left in the employee's car. Our case study is motivated by real espionage cases, such as the prosecution of Air Force

Table 2.5.: Results for Case Studies on Motorola G3.

Case Study	Activity Count	Duration (Minutes)	Activities Recovered	Activities Ordered	Root Structures	Residual Structures	Local Orderings	Number of Timelines	Kendall-Tau Distance
Spyware Attack	8	27	9	8	26	1638	3	2	{0,1}
Military Espionage	18	36	19	18	112	7151	5	1	0
Distracted Driving	17	16	18	17	92	6259	3	1	0
Kidnapping Investigation	18	41	19	18	101	6879	5	1	0

Table 2.6.: Timelines Recovered in Case Studies.

Case Study	Recovered Timelines
Spyware Attack	Spyware{ CommunicationReceiver } → Messaging{ Conversation } → Spyware{ CallRecorderReceiver } → Dialer{ InCallActivity } → Spyware{ FrontCameraActivity → VideoTimeReceiver → StopRecordingReceiver } → Gmail{ ComposeActivityGmail }
	Spyware{ CommunicationReceiver } → Messaging{ Conversation } → Spyware{ CallRecorderReceiver } → Dialer{ InCallActivity } → Spyware{ FrontCameraActivity → StopRecordingReceiver → VideoTimeReceiver } → Gmail{ ComposeActivityGmail }
Military Espionage	Signal{ RedPhone → ConversationActivity } → Launcher → CameraActivity → ChooserActivity(share) → WhatsApp{ ContactPicker → Conversation → ImagePreview → Conversation } → Launcher → FileBrowser{ FileBrowserActivity → TaskProgressActivity(delete) → FileBrowserActivity } → Launcher → Messaging{ Conversation } → Launcher → Chase{ HomeActivity → AccountsActivity }
Distracted Driving	Maps{ MapsActivity } → RecentsActivity → Netflix{ HomeActivity → SearchActivity → MovieDetailsActivity → PlayerActivity } → Launcher → Dialer{ DialContactsActivity → InCallActivity → DialContactsActivity } → Launcher → Netflix{ PlayerActivity → MovieDetailsActivity → SearchActivity → HomeActivity } → Launcher → RecentsActivity
Kidnapping Investigation	CameraActivity → Launcher → Skype{ HubActivity → PreCallActivity → HubActivity } → Launcher → Messaging{ ConversationListActivity → ConversationActivity } → Launcher → Skype{ HubActivity → PreCallActivity → HubActivity } → Launcher → Maps{ MapsActivity } → Launcher → Facebook{ PickerLauncherActivity → ComposerActivity → FbMainTabActivity }

Intelligence Officer Brian Regan [41] or that of Gillette employee Steven Davis [42] who were prosecuted for stealing classified national documents and corporate secrets respectively.

Our perpetrator Skip was hired as a defense contractor, working on a classified project for a federal agency, with routine access to sensitive documents. One day, after attempting to access classified information unrelated to his job, he checked out of the SCIF and walked around the parking lot of the facility. Alerted to the recent unauthorized attempts to access classified information, security personnel followed Skip into the parking lot, where they determined that he was carrying a mobile phone.

The security personnel use Timeliner to determine the timeline of Skip's recent actions. As it turns out, Skip received a secure call (*RedPhone*) just before he checked into the SCIF. After entering the SCIF, he used the Camera app (*CameraLauncher*) in his phone to take some photographs. These photographs were then sent over WhatsApp (*ChooserActivity(share)*) after he exited the SCIF and then summarily deleted (*TaskProgressActivity(delete)*). For selling the classified information, he received a deposit in his bank account, resulting in a text message (*ConversationActivity*), which he verified by opening the Chase Banking app (*AccountsActivity*). This timeline was deemed incriminating and Skip was then arrested and charged. As Table 2.5 shows, Timeliner recovers 19 Activities, 112 roots, and 7151 residual data structures for this timeline.

This case study demonstrates how important a timeline of user-actions is to an investigation. A traditional content recovery alone would be extremely limited as the photographs of the classified documents were deleted by Skip, severely limiting an investigation relying on content. Timeliner, on the other hand, provides conclusive proof of photographs being taken, shared, and then deleted.

We acknowledge that federal authorities are currently more likely to have the expertise and resources to react quickly enough to use Timeliner to retrieve actionable evidence before the detrimental effects of garbage collection occur, as in Skip's case above. However, with proper resources and training, Timeliner is also usable in a variety of scenarios by local and state authorities. The next case studies explore two such very important potential uses.

2.3.6 Case Study: Distracted Driving

In this case study we consider the problem of distracted driving. Specifically, using a smartphone while driving, which accounts for roughly 18% [43] of all injury-inducing automotive crashes. This situation is becoming so severe, that akin to a breathalyzer test, the state of New York is considering a *Textalyzer* law [44]. This law shall allow a police officer to conduct on-the-spot forensic analysis of a smartphone to determine if a driver was distracted while driving. Traditional techniques focus only on app-specific events, limited mostly to text/call/email/browsing logs [45]. On the other hand, Timeliner's app-agnostic capabilities work *without* temporal logs and provide much stronger proof of a driver's suspected distraction while driving.

We base this case on an accident involving a Tesla vehicle [46] where the driver was determined to be watching a movie after putting the car into the *AutoPilot* mode, which would clearly be classified as distracted driving. In our case study, the driver called roadside assistance after an accident. The police arrive a few minutes later and notice the "recent apps" screen on the driver's smartphone. Suspecting termination of an application they image the smartphone memory.

Table 2.5 shows that Timeliner recovered all 17 Activities that the driver used during the course of this case study via recovering three local orderings with 92 root and 6259 residual data structures. In the reconstructed timeline, investigators can see that the driver was first running the *MapsActivity* in Google Maps. At some point, the driver started Netflix with *HomeActivity*, followed by *SearchActivity* and *MovieDetailsActivity*, and finally playing a video with the *PlayerActivity*. Then the driver goes to the Dialer app and places a phone call, which was identified as the call to roadside assistance using call logs. After the call, the driver restarts the Netflix app, but backs out of all activities and finally terminates the app from the *RecentsActivity*.

The timeline confirms that the Netflix video was playing before the call to roadside assistance (and hence before the accident) was placed, but was terminated afterwards to hide the incriminating actions. Timeliner's ability to generate a timeline to precisely capture user actions across several applications — even in the face of deliberate app termination to hide evidence — is essential in this case. Timeliner is able to reconstruct the evidence solely from the phone's memory and termination of applications by the user does not hinder the recovery of actionable evidence.

2.3.7 Case Study: Kidnapping Investigation

Mobile phone investigations have aided in the apprehension of numerous criminals, and being a memory forensics technique, Timeliner can be used to quickly focus an investigation. We base this case study loosely on the real kidnapping/murder investigation detailed in [47]. Described as a "kidnapping gone wrong", the victim was bound using duct tape, and unfortunately she died from asphyxiation before a fake "rescue" the kidnapper had planned could take place.

In our case study, the kidnapper ("Kyle") and an accomplice ("Fred") force the victim ("Sally") into a pickup truck. A passerby (who identified Sally) quickly informs the police and the police identify Kyle as matching the description of the kidnapper. Kyle, located by the police at his residence, claimed he did not leave his house and showed his recent social media uploads (a photo at home) as proof.

A field-investigation of his phone, with the aid of Timeliner, reveals his actions in the recent past (shown in Table 2.5). Timeliner recovers 19 Activities with 101 roots and 6879 residual data structures. Joining three of the local orderings, Timeliner is able to precisely and accurately recover the timeline. The timeline shows that Kyle took the "alibi" photo with *CameraLauncher*, but did not post it immediately. Instead, he used the Skype app to call a person (*PreCallActivity*), then message Sally over text (*ConversationListActivity* and *ConversationActivity*), followed by another call via Skype. Then he used Google Maps (*MapsActivity*) to navigate and then finally posts the "alibi" photo to Facebook (*PickerLauncherActivity* and *ComposerActivity*). The police identify the Skype call recipient as Fred via Skype logs and obtain clearance to deploy Stingrays ("IMSI catcher" devices) against Fred's number. This allows them to rapidly find both Sally and Fred in a nearby wooded area.

This case study demonstrates how Timeliner complements the traditional content recovery forensics. While the accomplice is identified by Skype logs on the smartphone, and the message recipient is identified as Sally with messaging logs, Timeliner provides the incriminating evidence of fake "alibi" photograph being taken before the Skype calls, which raises suspicion and provides enough proof to deploy Stingrays.

2.4 Discussion

As Timeliner relies on the observation that a temporal ordering in allocations produces a spatial ordering, any attempt by a device owner or an app to hide their actions must attack either the recovery of residual data structures or change the allocator's deterministic behavior.

However, as Timeliner focuses on only the ActivityManagerService process, separate from the apps, erasing evidence by modifying records or running garbage collection is not possible for even the most technically advanced criminals or privacy sensitive app developers. The only way a device owner may affect the ActivityManagerService is to flash a custom "Timeliner-aware" Android runtime system onto their device — which is both technically difficult (modifying Android's internals) and risky (may "brick" the device).

One way an app developer can avoid Timeliner's recovery is by not utilizing Activities, and building their own functionality to emulate Android's Activity stack (e.g., by intercepting back keypresses). However, this is a prohibitively cumbersome process for most app developers and has only been implemented in a few apps including certain web browsers and gaming apps due to strict performance requirements. Further, this also prevents an apps' interaction with other apps, for example – File Browser can directly share a file with the Gmail app (using an Intent for the ComposeActivity) but not with the gmail website opened on the Chrome browser app.

2.5 Summary

Targeting the problem of re-sequencing an Android device user's past actions, we present Timeliner, a memory forensics technique that reconstructs a timeline of *Activities* across all apps (including those which have terminated) that were performed on the device. Starting from the set of data structures left in a memory image by past Activity launches, Timeliner infers *Activity* transitions based on the relative memory layout of those data structures. Our results show that Timeliner is highly accurate in reconstructing past activities of a user. Moreover, we show through a suite of case studies that Timeliner is applicable to a variety of crime investigation scenarios.

3 "ONE AND ONE MAKE ELEVEN": ACCOMPLICE-ASSISTED MASQUERADE ATTACK ON CAN

The controller area network (CAN) is a wired broadcast network widely utilized in modern automobiles and other automatic control systems to enable real-time communication among electronic control units (ECUs). As automobiles become more connected, various attack surfaces for adversaries [6–8, 10, 13, 14] have opened up, allowing compromise of in-vehicle ECUs. After gaining foothold in an ECU, the attacker can carry out a variety of attacks [8, 9, 13–15] over the CAN bus, which lacks mainstream network security capabilities due to resource constraints. In particular, a compromised ECU (e.g., Telematics Control Unit) may impersonate a benign ECU (e.g., Engine Control Unit) that cannot be directly compromised, and forge the latter's CAN messages to disrupt safety-critical functions (e.g., engine speed). We call such attacks *ECU masquerade attacks*.

To defend against ECU masquerade attacks, various CAN intrusion detection systems (IDS) have been proposed to detect malicious/compromised ECUs on an automobile CAN [21]. The IDS approach is the dominant CAN defense because an IDS runs as an independent entity on the CAN bus and does not burden the resource-constrained ECUs/network with non-trivial computation/communication overhead. Traditional message (anomaly)-based IDS (MIDS) [11, 22, 23] utilize features such as message frequency and payload values to identify anomalies on CAN. Modern state-of-the-art CAN IDS leverage physical characteristics of ECUs, such as clock-skew (CIDS [21]) and voltage (VIDS [18–20, 48]), to determine if a message is generated by a legitimate ECU. While MIDS and CIDS have been shown vulnerable to persistent attackers [21, 49], VIDS have been proved highly effective in detecting ECU masquerade attacks, due to the robustness and non-imitability of an ECU's voltage fingerprint. Particularly, VIDS equipped with high-frequency voltage sampling, virtually put the CAN traffic under a “microscope” for high-resolution voltage monitoring and attacker detection. As such, they are being actively researched by the automotive industry, e.g., the VIDS called Scission [18] has been developed by Bosch that is also the developer of the CAN standard [50].

In this paper, we present a novel masquerade attack strategy called DUET which evades the state-of-the-art VIDS. As its name indicates, DUET involves a duo of “attacker” and “accomplice” ECUs, which work together to manipulate, silence and then impersonate a victim ECU. Different from the “lone wolf” style (i.e., by a single attacker ECU) of existing masquerade attacks, the duo assist each other to evade VIDS detection. DUET follows a three-stage attack strategy, which includes *voltage fingerprint manipulation*, *persistent victim bus-off*, and *voltage fingerprint-based impersonation*. In each stage, either the attacker ECU (or “attacker” for short) or accomplice ECU (accomplice) plays the primary attacking role whereas the other one covers up their activities from VIDS.

DUET first performs voltage fingerprint manipulation (Stage 1) in which the attacker, with the help of the accomplice, transmits simultaneously with the victim and stealthily corrupts the voltage fingerprints of the victim’s messages. In the process, by gradually poisoning its training set, DUET tricks the VIDS into learning a distorted fingerprint of “victim + attacker”. Thereafter, whenever spoofed messages need to be sent, DUET performs persistent victim bus-off (Stage 2) in which the duo induce transmission errors in the victim’s messages and force the victim to stop all its transmissions on the CAN bus. Finally, DUET performs voltage fingerprint-based impersonation (Stage 3) in which the accomplice sends the spoofed messages while the attacker corrupts their voltage fingerprints. DUET ensures that the VIDS classifies the distorted fingerprint of “accomplice + attacker” as the distorted fingerprint of “victim + attacker”. Hence, execution of the three DUET stages leads to a stealthy, advanced persistent threat (APT)-style masquerade attack.

DUET employs two new attack tactics that exploit fundamental deficiencies of the CAN protocol: (1) *Voltage corruption* enables the attacker to transmit simultaneously with the victim and accomplice, and corrupt their voltage samples in Stages 1 and 3 of DUET, respectively. This tactic exploits the periodicity of CAN messages, the predictability of CAN message content and the "one-shot transmission mode" of CAN controllers. (2) *Passive error regeneration* exploits a flaw in the error handling mechanism of CAN, allowing regeneration of an error by blocking the transmission for the original error. This tactic causes a flash flood of errors which stealthily and swiftly silences the victim in Stage 2 of DUET.

We highlight that DUET exploits *common* deficiencies of the CAN protocol and characteristics of the CAN traffic. Hence DUET can be launched against any automobile employing the CAN bus. Also, in Stage 1 of DUET, the attacker superimposes its attack message over the victim's message and distorts the victim's voltage samples. This enables DUET to successfully evade all state-of-the-art (training-based) VIDS, *irrespective* of the features and classification algorithms used in them. Furthermore, state-of-the-art defenses against training set poisoning [51, 52] cannot prevent DUET.

To safeguard against DUET, we propose an efficient defense, called *Ra*ndomized *I*dentifier *D*efense (RAID). Breaking away from the "attack vs. IDS" arms race, RAID takes a simple, orthogonal approach by randomizing a part of the CAN message identifier (ID). RAID ensures that the attacker in DUET will either win or lose the arbitration while transmitting on the bus, and will not be able to transmit simultaneously with the victim. As a result, the attacker cannot corrupt the victim's voltage fingerprint or cause errors in the victim's transmission, which prevents (not just detects) DUET. In this way, RAID equips ECUs with the necessary tool to make the existing VIDS effective against APT-style attacks.

We have evaluated DUET and RAID in real CAN environments, which include our own testbed with ECU prototypes and CAN buses in two cars. Our evaluation shows that after corrupting voltage samples corresponding to only two bytes of the victim's payload, DUET demonstrates an impersonation *success rate* of at least 76% against the VIDS which analyzes the fingerprint of each message [18] and 95% against the VIDS which analyzes the cumulative fingerprint of eight messages [20]. Our evaluation of

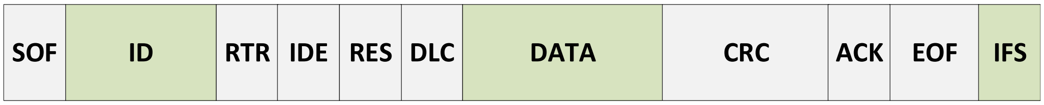


Figure 3.1.: Standard and extended frame formats in CAN.

RaID shows that it prevents Duet while incurring negligible computation overhead, and no more than an increase of 13% in bus load and 50 μs in message delays during the training mode of VIDS.

We summarize our main contributions as follows.

- We propose Duet as a three-stage masquerade attack strategy through which a duo of attacker and accomplice evade CAN VIDS by exploiting novel tactics.
- Our analytical and evaluation results illustrate that Duet is effective against all state-of-the-art VIDS irrespective of features and classification algorithms utilized in them.
- We advocate the development of cost-effective defenses that break away from the "attack vs. IDS" arms race, and propose RaID, an efficient and effective defense, for complementing VIDS and preventing Duet.

3.1 Background

Data Transmission. In CAN, the bits 1 and 0 are called recessive and dominant bits, respectively. CAN employs a twisted wire pair called CAN high (CAN-H) and CAN low (CAN-L) so that the recessive (1) and dominant (0) bits are represented by the differential (between CAN-H and CAN-L) voltages of 0 V and 2 V, respectively. We note that the transmission of a recessive bit (with 0 V) effectively remains *transparent* to other ECUs and the CAN bus. When multiple ECUs transmit their bits concurrently, the resultant bit is equal to the logical `AND` of the transmitted bits and the voltages on the CAN-H and CAN-L correspond to the resultant bit [50].

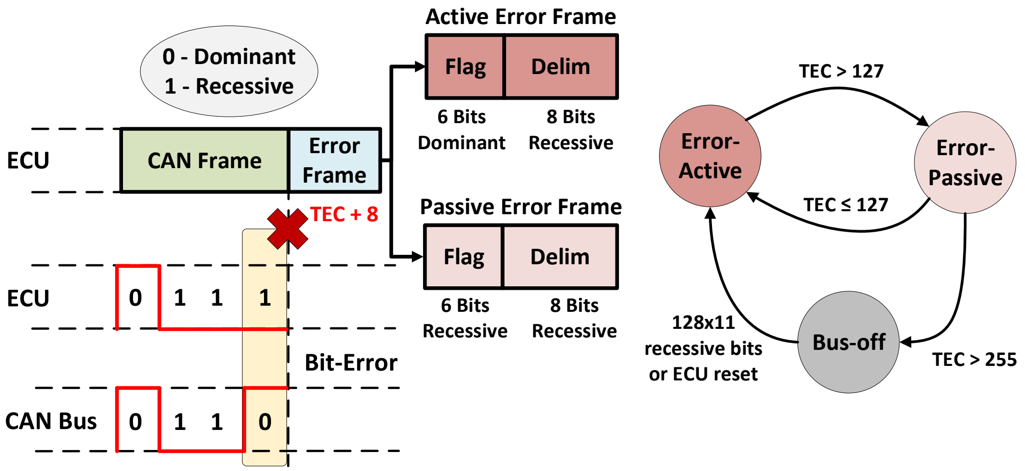


Figure 3.2.: Error handling mechanism in CAN.

CAN messages are transmitted in either the standard or extended frame format (Figure 3.1). We note that an ECU can transmit different messages with different formats on the same bus. While a frame in the standard format is identified by an 11-bit ID-A field, a frame in the extended format is identified by a 29-bit ID which is composed of two fields: an 11-bit ID-A field and an 18-bit ID-B field. In both formats, the length of the data field can be from zero to eight bytes as indicated in the data length code in the control (CTL) field.

Arbitration. The CAN protocol employs a robust arbitration procedure among ECUs, which allows only one ECU to transmit the message payload (the data field) on the bus. Each ECU starts transmitting on the bus by synchronizing with the start-of-frame (SOF) field. For the ID fields, each ECU sends an ID bit on the bus, and then reads it back from the bus. If an ECU reads a dominant bit while it transmitted a recessive bit, it loses the arbitration, and stops transmitting. Since each of the messages transmitted by ECUs is allotted a *unique* ID, only one ECU wins the arbitration and continues to transmit the following fields including the control and data fields.

Error Handling. Communications on CAN may be corrupted due to many factors such as software/hardware faults and electromagnetic interference from other ECUs or nearby devices. During any transmission other than the arbitration field, a *bit-error* occurs when the transmitted bit is not equal to the resultant bit on the bus (Figure 3.2). To tolerate such errors, CAN employs a unique error handling mechanism: Each ECU maintains a *transmit error counter* (TEC). While an error detected during transmission increases the TEC by eight, a successful transmission decreases the TEC by one.

Based on the TEC value, the ECU enters one of the three states: *error-active*, *error-passive* and *bus-off* (Figure 3.2). The ECU transitions from error-active to error-passive state when its TEC goes beyond 127, from error-passive to bus-off state when its TEC exceeds 255, and from bus-off to error-active either on ECU reset or on observing 128 instances of 11 recessive bits. In the bus-off state, the ECU cannot transmit/receive any data. As such, the ECU causing the fault is stopped from continuously corrupting the communication.

When a bit-error occurs during transmission (Figure 3.2), the ECU will behave differently in the error-active and error-passive states: The error-active ECU will transmit an *active error* frame consisting of six dominant bits and eight recessive bits. The error-passive ECU will transmit a *passive error* frame consisting of 14 recessive bits. Also, for successive transmissions in the error-passive state, an ECU is required to have an 8-bit suspend transmission field in addition to the regular 3-bit inter-frame spacing (IFS) shown in Figure 3.1.

Simultaneous Transmission and Preceded ID. A simultaneous transmission comprises of two or more concurrently transmitted messages which continue to overlap beyond their arbitration fields. Cho and Shin [53] showed that an attacker can deliberately perform simultaneous transmission with a victim if the attacker synchronizes its SOF field and utilizes the same message ID as a victim message. To achieve synchronization, a *preceded ID* is employed [53], which is a message injected right before the periodic transmission of a target message. The preceded ID message forces any other message to wait for the completion of its transmission. This way, although the victim and attacker may generate their messages at slightly different times, when

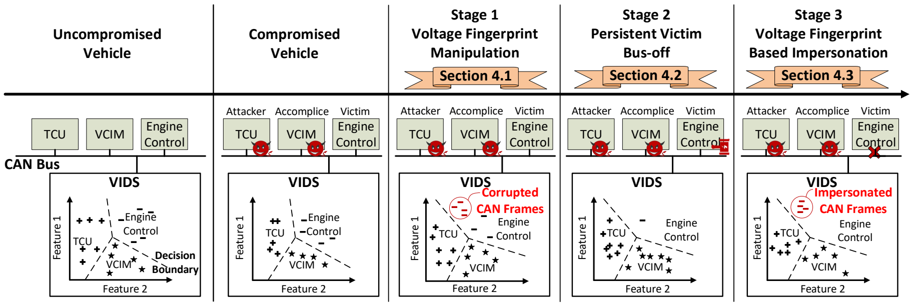


Figure 3.3.: Overview of a DUET attack where the Telematics Control Unit (TCU) and Vehicle Communication Interface Module (VCIM), compromised using their wireless interface, collaborate to masquerade the Engine Control Unit and evade the VIDS.

the preceded ID message is transmitted, they will start transmitting the SOF fields of their messages right after the IFS field of the preceded ID.

VIDS. VIDS [18–20, 48] extract features from measured message voltage characteristics of each ECU during transmission of CAN messages and learn a supervised model to infer the message source. Due to the robustness and non-imitability of an ECU's voltage characteristics, VIDS have been shown highly effective in detecting compromised, unmonitored, and newly added ECUs on a CAN. However, the message voltage characteristics and hence the ECU fingerprints vary with time due to aging effects, changing environmental/workload factors, and firmware updates [20]. For instance, the VIDS presented in [48] is robust to fingerprint variations up to only 6°C fluctuations in ambient temperature, while another VIDS [19] yields poor predictions beyond 10°C temperature differences. Since such variations may happen within a few hours/days [54], the model must be adapted accordingly. Hence, VIDS utilize online learning [20], incremental learning [19, 48], or periodic model retraining [18] to update the model.

Table 3.1.: Effectiveness of DUET against existing VIDS.

VIDS	**Sampling Frequency**	**Number of Features**	**Anomaly Detection**	**Vulnerable to DUET**
Viden (2017) [20]	50 KS/s	6	per 8 messages	✓
VoltageIDS (2018) [19]	2.5 GS/s	64	per message	✓
Scission (2018) [18]	20 MS/s	48	per message	✓
SIMPLE (2019) [48]	500 KS/s	32	per message	✓

3.2 DUET Overview

VIDS provide the state-of-the-art defense against existing ECU masquerade attacks. In this paper, we describe DUET, a novel masquerade attack which evades detection from all existing VIDS as illustrated in Table 3.1. DUET is carried out in a stealthy, multi-stage fashion based on an APT-style strategy. It involves two compromised ECUs – *attacker* and *accomplice* – with the goal of impersonating an uncompromised victim ECU (*victim*) without being detected. To achieve this, the duo work together to distort the voltage fingerprint of the victim in Stage 1 (Section 3.3.1), knock the victim off the bus in Stage 2 (Section 3.3.2), and then impersonate the victim using its distorted voltage fingerprint in Stage 3 (Section 3.3.3), all under the watch of VIDS as illustrated in Figure 3.3.

Characteristics of CAN Traffic. DUET exploits three common characteristics of vehicular CAN traffic: (1) static IDs, (2) message periodicity, and (3) *predictable payload-prefix* (PREP), which is the predictable set of bits representing constants, counters, and multi-valued numbers after the arbitration fields of CAN messages (Figure 3.1). We confirm these characteristics using the CAN traffic from five vehicles (of four different brands). Table 3.2 summarizes our findings, with detailed results in Section 3.8. We have also validated the prevalence of these characteristics in other modern vehicles using the reverse engineered data available at [55].

Attack Tactics. DUET is enabled by two novel attack tactics leveraged in its three attack stages.

Table 3.2.: Common characteristics of CAN traffic (each number is a message count).

	2011 Chevrolet Impala	2011 Chevrolet Cruze (Bus-1)	2011 Chevrolet Cruze (Bus-2)	2012 Toyota Camry	2012 Honda Civic	2010 Dodge Ram
Total messages	50	88	27	42	45	55
Periodic	50	88	27	42	45	51
PREP $\geq$ 1 byte	49	83	25	42	42	50

Tactic 1: Voltage Corruption. We discover that when the attacker is in the error-passive state and performs simultaneous transmission with an ECU, it will successfully corrupt the ECU's voltage samples, without leaving a trace on the bus. The attacker exploits this tactic to corrupt the voltage fingerprint of the victim during voltage fingerprint manipulation (Stage 1 of Duet) and that of the accomplice during voltage fingerprint-based impersonation (Stage 3). This enables the Duet duo to evade VIDS while impersonating the victim.

Tactic 2: Passive Error Regeneration. We discover a flaw in the CAN protocol regarding the victim's TEC increase. If the attacker blocks the transmission of a victim's passive error frame, the victim will regenerate another passive error frame and raise its TEC. Duet exploits this flaw to stealthily and swiftly push the victim into the bus-off state in Stage 2.

Attack Model. We assume that the adversary behind Duet makes a one-time reverse engineering effort to infer all CAN messages, patterns in their payloads, and their source ECUs in the *target* vehicle or in a vehicle with the same make and model as shown in [55–57]. We also follow the attack model in prior art [18, 20, 21, 53] and assume that the adversary behind Duet is capable of achieving arbitrary code execution on at least two ECUs of a vehicle CAN. Previous works have demonstrated the compromise of ECU(s) through various remote and physical attacks [9–11]. Other works showed APT-style stealthy compromise of multiple ECUs, and their persistence over reboots and ECU firmware flashing/updates [6–8, 13, 14]. A compromised ECU can read and inject messages on the bus through its CAN controller. Since the

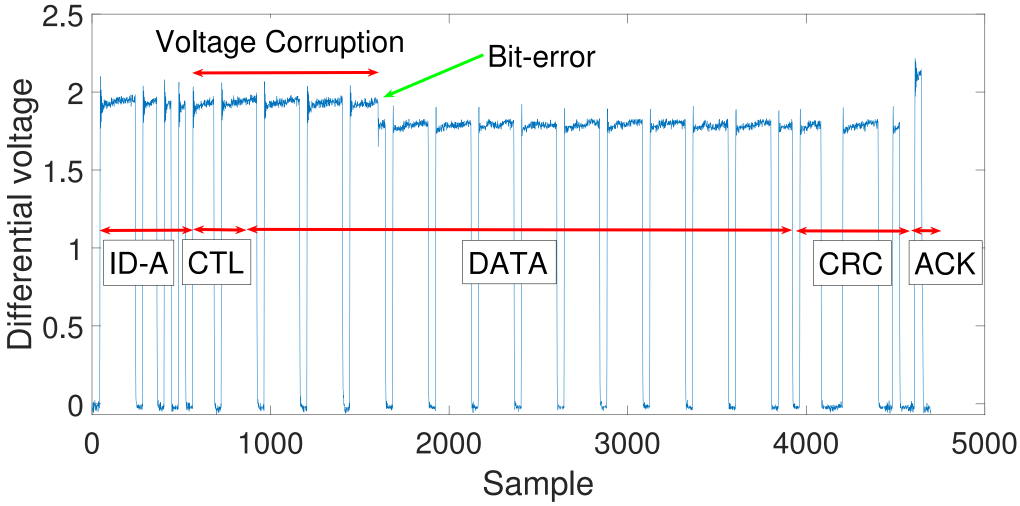


Figure 3.4.: Example of voltage corruption in an ECU's frame.

controller is implemented in hardware, the compromised ECU is still subject to the CAN transmission mechanisms and protocol specifications. We point out that *not all* ECUs are directly compromisable [8], making the masquerade attack valuable to adversaries. Finally, we let the compromised ECUs utilize their own periodic messages to indicate the transitions between different attack stages of Duet.

3.3 Detailed Design of Duet

3.3.1 Voltage Fingerprint Manipulation

The first stage of Duet, voltage fingerprint manipulation, is carried out by the attacker and accomplice to stealthily distort the voltage fingerprint of the victim ECU. We point out that Duet is the first effort in training set poisoning on the CAN bus. While the machine learning (ML)-based IDS [58, 59] including those used in automotive control systems [60] are susceptible to data poisoning attacks, Duet faces two CAN-specific challenges. First, voltage samples generated by the victim cannot be directly controlled by the attacker. Second, the attacker/accomplice cannot

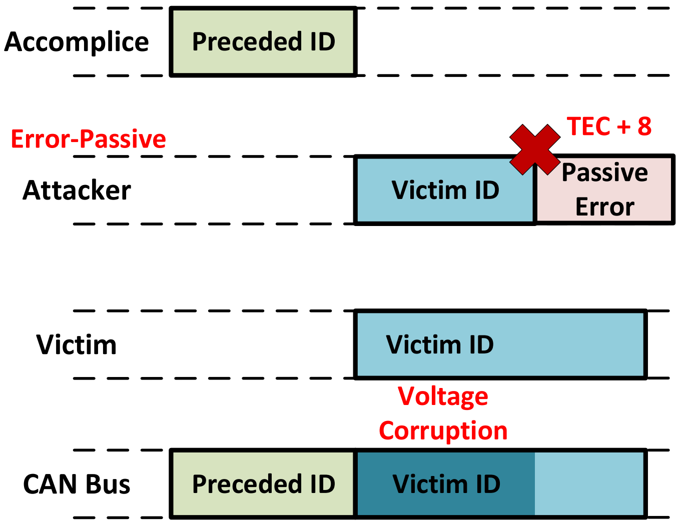


Figure 3.5.: Voltage fingerprint manipulation (Stage 1 of DUET).

modify its own voltage fingerprint [18]. To address these challenges, we introduce the *Voltage Corruption* tactic that enables the attacker to poison the training data with the help of an accomplice.

Tactic 1: Voltage Corruption. We observe a unique behavior when an attacker in error-passive state performs simultaneous transmission with an ECU (during which the attacker transmits the same bits as the ECU) until a specific bit location, where the attacker and the ECU transmit the recessive bit and dominant bit, respectively. At that location in the attacker's frame, a bit-error occurs which terminates the attacker's transmission. Since the resultant bit is dominant, the ECU does not observe the bit-error and continues transmission. Before the bit-error, the attacker's transmission overlaps with the ECU's without hindering it; whereas, after the bit-error, the attacker waits until the end of the ECU's transmission and then transmits a passive error frame. Since this error frame consists of only recessive bits, it is *transparent* to the ECU and leaves no trace on the bus. An example illustrating the impact of voltage corruption on the differential voltage on CAN is shown in Figure 3.4: The error-passive attacker transmits simultaneously with an ECU, and causes a bit-error after two bytes of data payload. It is clear in Figure 3.4 that voltage values (resulting from the overlap between the attacker's and ECU's transmissions) of the corrupted bits before the bit-error are higher than voltage values (of the benign bits corresponding to the ECU only) after the bit-error.

In Stage 1 of DUET, the attacker utilizes the voltage corruption tactic on the victim and distorts the victim's voltage fingerprint. Specifically, as shown in Figure 3.5, the accomplice first assists the attacker to attain the error-passive state. Then the attacker and accomplice independently estimate the time-of-transmission of the victim's message by exploiting its periodic behavior. The accomplice injects a preceded ID frame right before the transmission of the victim's message to help the attacker synchronize with the victim's transmission. The error-passive attacker utilizes the victim's ID in its attack message and transmits its attack message simultaneously with the victim's message. This corrupts the victim's voltage samples. Since VIDS learn the victim's fingerprint from the training set of (now corrupted) voltage measurements, the distorted *fingerprint of the simultaneous transmission* of "victim + attacker" is mistaken for the victim's fingerprint by VIDS.

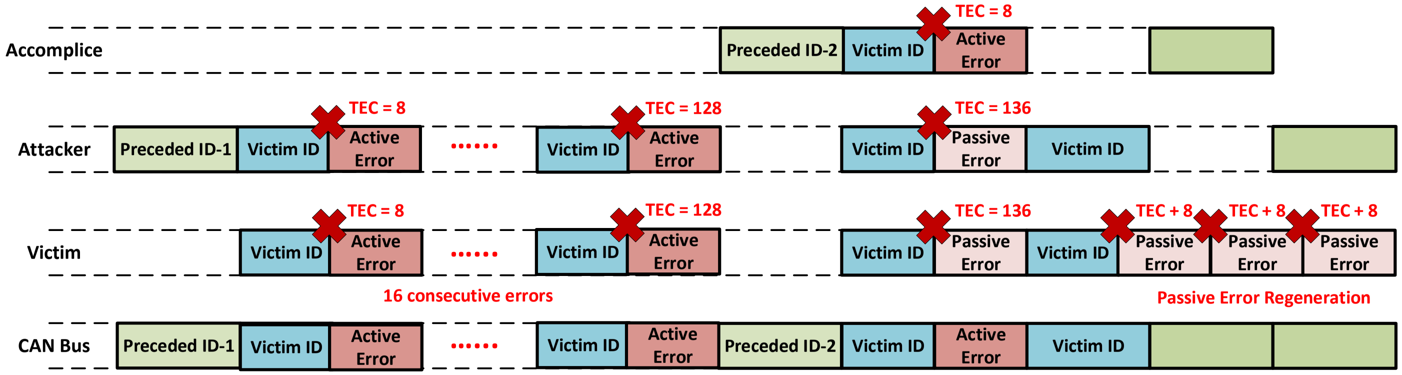


Figure 3.6.: Single instance of persistent victim bus-off (Stage 2 of DUET).

Fingerprint Manipulation. To manipulate the distorted victim's fingerprint, the attacker can control the bits corrupted in the victim's message. The CAN traffic from the five cars we profiled (Section 3.8) reveals an interesting observation that makes this feasible: The content after the arbitration field associated with a given message ID contain a *predictable payload-prefix* (PREP) of bits representing constants, counters, and multi-valued numbers which can be reliably predicted in advance. Since the payload length for a specific message ID remains the same, the PREP consists of at least six constant bits of the control field. With offline reverse-engineering of PREP in the victim's payload, the attacker can readily identify additional PREP bits in the data field. Then the attacker can utilize a selected portion of the same PREP in its attack message and enforce the bit-error *at any desired location.*

Manipulation Rate and Stealth. Stage 1 of DUET has a critical objective: corrupting the PREP bits of the victim's messages in the training set without raising a VIDS alarm. This objective can be effectively fulfilled if messages utilized for training VIDS can be identified, e.g. when they contain distinct IDs, such as those in [19]. However, a VIDS might be trained silently (without any observable indication on the bus) with regular CAN messages; thus, identifying the training set messages can be impractical. In such cases, manipulating victim's fingerprint is a particularly difficult endeavor because a significant corruption in messages not used for training would clearly be detected as an attack. To address these challenges, DUET exploits the fact that a VIDS requires online, incremental or periodic retraining to account for changing environmental and weather conditions [54]. Therein DUET manipulates *all* victim's messages and sets the manipulation rate *lower* than the retraining frequency of VIDS, where the manipulation rate is defined as the frequency of increasing the number of corrupted bits from zero to the desired bits in PREP. In fact, DUET increases the corruption one bit at a time and tricks VIDS into learning an increasingly manipulated fingerprint of victim without raising an alarm (Section 3.6.2).

Further, without special precaution, voltage corruption by the attacker will exhibit an anomaly which can be detected by VIDS: The attacker ECU's CAN controller

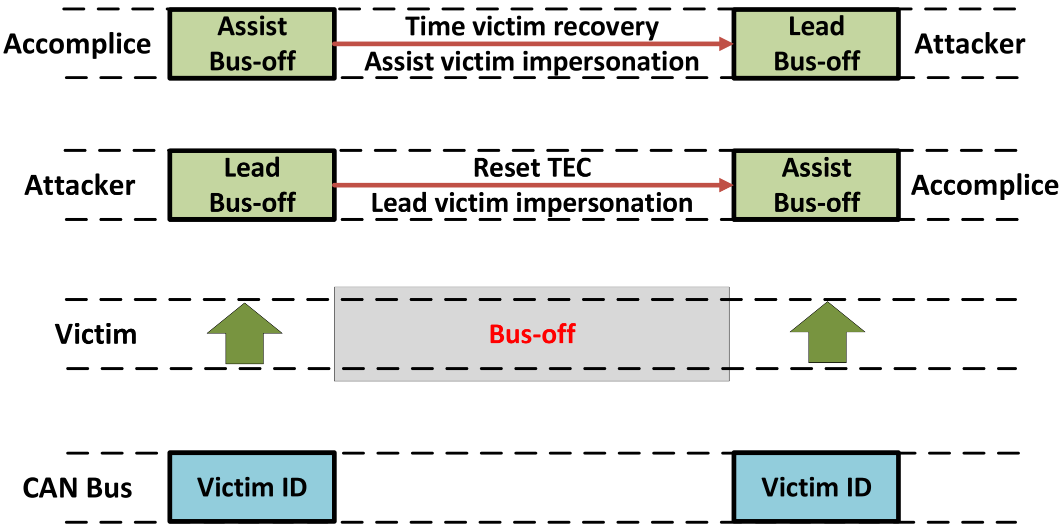


Figure 3.7.: Roles alternation in consecutive instances of DUET.

attempts to retransmit after encountering the bit-error during voltage corruption. This leads to the retransmission of the attack message which contains the victim's ID with the attacker's fingerprint. DUET eliminates this anomaly by exploiting a feature called *one-shot transmission mode*, available in all the popular CAN controllers [61, 62]. In this mode, the attacker attempts to transmit the attack message only once, and does not retransmit even after encountering the bit-error. Hence, the attacker evades detection while performing voltage corruption.

Accomplice's Role. The accomplice is vital for helping the attacker transition to error-passive state. Further, in the error-passive state, the attacker needs an extra 8-bit suspend transmission field between successive transmissions. This hinders the ability of the attacker to transmit the preceded ID frame to synchronize with the victim as well as perform voltage corruption of the victim's message. Hence, the accomplice must help the attacker with its preceded ID frame.

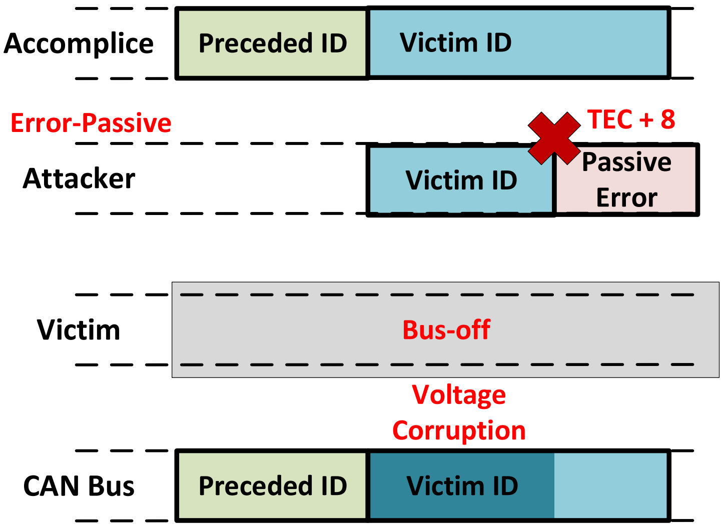


Figure 3.8.: Voltage fingerprint-based impersonation (Stage 3).

3.3.2 Persistent Victim Bus-off

After Stage 1, DUET needs to do one more stunt at run-time, before impersonating the victim: shun the victim from the CAN bus. In Stage 2, DUET exploits CAN's error-handling mechanism (Section 3.1) and persistently imposes the bus-off state on the victim. Specifically, as shown in Figure 3.6, the attacker (in error-active state) first injects a preceded ID frame before the transmission of the victim's message to enable synchronization with the victim. The attacker then transmits its attack message simultaneously with the victim, which causes bit-error at the location where their transmitted bits are different. Both the victim and attacker encounter this active error, and hence they ceaselessly attempt to retransmit, which causes more errors propelling both of them to error-passive state [53].

Now the accomplice joins the action with a preceded ID frame and then forces a bit-error which further increases TECs of the victim and attacker. The accomplice's action is necessary to ensure that the attacker has a high TEC (>127) and remains in error-passive state for Stage 3 of DUET. Then the attacker sends another attack message with a dominant bit while the victim transmits a recessive bit. Upon this bit-error, the victim raises a passive error, but the attacker completes the transmission of its attack message. Thereafter, to transition the error-passive victim to bus-off state, we introduce the novel tactic called *passive error regeneration*.

Tactic 2: Passive Error Regeneration. We discover and exploit a critical CAN protocol flaw that allows the attacker to stealthily and swiftly increment an error-passive victim's TEC. When a victim in error-passive state encounters a bit-error, it attempts to transmit a passive error frame. Since this frame consists of 14 consecutive recessive bits, it can only be successfully transmitted in a 14-bit interval where no dominant bit is transmitted. The CAN protocol assumes that the victim should be able to find such an interval to transmit its 14 recessive bits, before the start of the next message. However, the attacker and accomplice in DUET nullify this assumption. As per the CAN protocol, the number of recessive bits between two consecutive frames comprises of the 1-bit ACK delimiter, 7-bit end-of-frame (EOF) field, 3-bit inter-frame space (IFS), and a variable (≥ 0) bus-idle period. By transmitting two frames back-to-back, the number of recessive bits between two frames can be reduced to just 11 (which corresponds to zero bus-idle period). This results in transmission failure of the victim's 14-bit passive error frame. Hence, the passive error frame's transmission itself encounters a bit-error due to the dominant SOF bit of the next frame, causing the victim to increase its TEC and generate a new passive error frame.

In DUET, the attacker and accomplice exploit this CAN protocol flaw for enforcing such self-defeating behavior of the victim repeatedly to swiftly push the victim from error-passive to bus-off state. As the error-passive victim attempts to transmit its passive error frame, the attacker and the accomplice clutter the CAN bus with benign traffic of their own, along with other benign traffic on the bus, causing regeneration of passive error frames – until the victim enters bus-off state. We highlight that the regeneration of passive error frames results in quick increase of the victim's TEC without any significant effort by the attacker/accomplice and without any observable anomaly on the bus. This concludes an instance of the persistent victim bus-off.

Attacker-Accomplice Role Alternation. After pushed to the bus-off state, the victim remains in that state for only a short period of time (128 instances of 11 recessive bits or ECU reset). Hence, the victim must be "bus'ed-off" again. To re-bus-off the victim, one of the Duet duo must measure the time of the victim's recovery either by counting the sequences of 11 recessive bits as described in Section 3.9 or by waiting for ECU reset. Since the attacker's TEC was just increased as a result of the previous bus-off instance, it must reset its TEC and cannot lead the next attack. Therefore, as shown in Figure 3.7, the *accomplice* times the victim's recovery and leads the next instance of bus-off as the attacker, whereas the original attacker takes on the accomplice's role. To further mitigate the inherent uncertainty in timing the victim's recovery, Duet allows the victim to transmit one buffered message. This provides an error margin of one frame in the timing procedure and enables the Duet duo to re-bus-off the victim by precisely attacking the victim's next message, which immediately follows the buffered message. Hence, the role alternation and timing procedure ensures that the victim remains in the bus-off state persistently.

Stealth. Without any provision, the 16 active errors (shown in Figure 3.6) will occur at exactly the same bit location. Such "identical bit-error locations" can serve as an anomaly to detect simultaneous transmission by multiple ECUs. Hence, similar to Stage 1, the attacker utilizes the one-shot transmission mode to prevent retransmission of the same attack message. While enqueueing a new attack message for simultaneous transmission, the attacker randomizes the bit-error locations among the bits in PREP to invalidate the anomaly. Hence, the CAN under Duet's Stage 2 behaves the same as if it is experiencing short, transient errors, which are common during normal vehicle operations due to physical factors such as electromagnetic interference (e.g., sudden braking can lead to 12 consecutive active error frames [63]) or bus termination (e.g., resistance changes with temperature or aging [64]). As a result, these errors will not be reported by VIDS.

3.3.3 Voltage Fingerprint-Based Impersonation

With the victim in bus-off state, the duo inject forged messages to impersonate the victim in Stage 3. Duet succeeds if receiver ECUs accept these messages *without* VIDS raising any alarm. To achieve this, the attacker, which is in error-passive state at the end of Stage 2, utilizes the voltage corruption tactic (Tactic 1 in Section 3.3.1) on the accomplice. As shown in Figure 3.8, the attacker and accomplice synchronize using the preceded ID frame transmitted by the accomplice. Then, the accomplice injects a (forged) victim's message, and the attacker transmits simultaneously corrupting the accomplice's voltage samples. Hence, VIDS record the distorted fingerprint of the simultaneous transmission of "accomplice + attacker". Also, similar to Stage 1. the attacker prevents retransmission of its attack message by using the one-shot transmission mode, and regulates the length of the superposition of its attack message and the accomplice's forged message.

Stealth. Recall that in Stage 1 of Duet, VIDS have been tricked into believing that the victim's fingerprint is that of simultaneous transmission by "victim + attacker". Since the features corresponding to the two distorted fingerprints ("victim + attacker" and "accomplice + attacker") are generated by the superposition of features of two ECUs, they are significantly different from the features of any *single* ECU currently on the bus. Therefore, Duet succeeds to stealthily impersonate the (manipulated) fingerprint of the victim by ensuring that the fingerprint of simultaneous transmission by "accomplice + attacker" is classified as that of "victim + attacker" by VIDS. The formal and empirical analysis of Duet's success rate are presented in Section 3.4 and Section 3.6.3, respectively.

3.4 Analysis of Stealth of Duet against VIDS

Voltage Distribution. Figure 3.9 illustrates the distinct characteristics of the voltage samples (corresponding to dominant bits) in an *individual transmission* (by the victim, attacker, or accomplice) and a *simultaneous transmission* (by the "victim

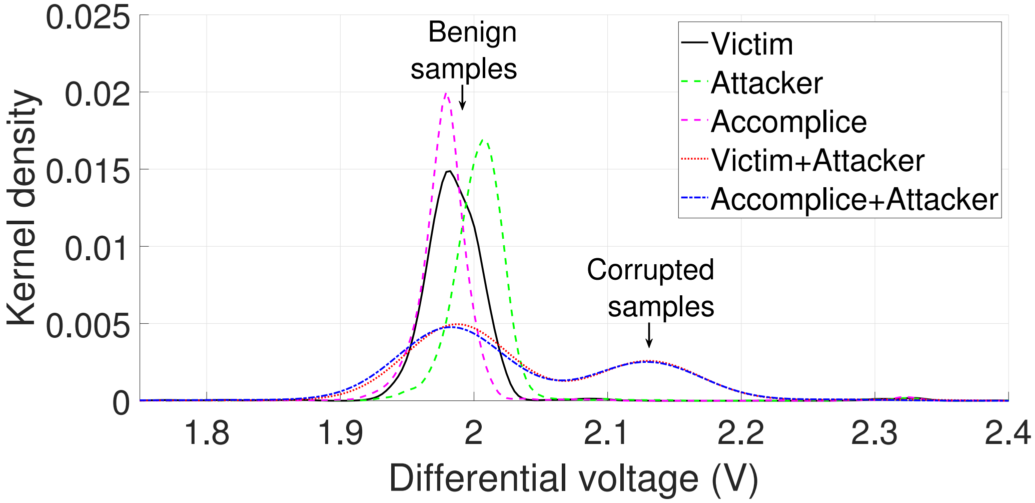


Figure 3.9.: Voltage distribution observed by VIDS when the attacker corrupts 2 bytes of payload of victim/accomplice.

+ attacker" or "accomplice + attacker"). On one hand, voltage samples of an individual transmission closely fit a *unimodal* Gaussian distribution which is consistent with the findings in prior art [20]. On the other hand, we discover that voltage samples in a simultaneous transmission follow a *bimodal* Gaussian distribution. This is because in a simultaneous transmission, the voltage corruption (which is limited by PREP) results in two sets of voltage samples in a CAN frame: corrupted and benign samples, such that corrupted samples have higher voltage values than benign samples (Figure 3.4).

Root Cause for Successful Impersonation. In Figure 3.9, we observe that bimodal distributions of any two simultaneous transmissions are statistically "closer" to each other than a unimodal distribution of any individual transmission. In other words, the values of features (such as mean and standard deviation of samples) of a simultaneous transmission are closer to those of another simultaneous transmission than those of an individual transmission. Hence, VIDS (which utilize a multitude of such features to identify the source) are likely to *misclassify* a simultaneous transmission as another simultaneous transmission. Exploiting this fundamental shortcoming of VIDS, Duet utilizes a simultaneous transmission ("victim + attacker") to create a manipulated fingerprint of the victim, and then impersonates the fingerprint with another simultaneous transmission ("accomplice + attacker"). Note that each individual feature of the resultant "victim + attacker" and "accomplice + attacker" transmission after voltage corruption does *not* need to be identical for the two fingerprints. VIDS can be evaded if the fingerprint (which is the combination of all features) of "accomplice + attacker" lies within the decision boundaries learned for the "victim + attacker" as illustrated in an example shown in Section 3.10.

Formal Model for Duet. We now discuss the stealth of Duet against VIDS by formally defining a metric called *success rate*, which quantifies the ability of Duet to get an impersonated message to evade VIDS. Also, we describe various parameters controlled by the attacker and accomplice, and model the objective of Duet as an optimization problem.

VIDS. We first define a VIDS and its functionality. Let the training data be represented as $\mathbf{D}_o = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$, where $\mathbf{x}_i \in \mathbb{R}^q$ represents a fingerprint/vector of q features, $y_i \in [1, C]$ represents the class, C represents the total number of classes, and N represents the number of samples utilized for training. In the training mode, the VIDS classifier learns the model $\theta_o = \text{argmin}_{\theta \in \Theta}\ \mathcal{O}_t(\mathbf{D}_o, \theta)$, where Θ represents the hypothesis space, and $\mathcal{O}_t$ represents the objective function of the classifier in the training mode. During the operation mode, for a given vector of features $\mathbf{x}_*$ and the model θ_o, the classifier outputs $y_* = \text{argmax}_{c \in [1,C]}\ \mathcal{O}_d(c \mid \mathbf{x}_*, \theta_o)$, where $\mathcal{O}_d$ represents the objective function of the classifier in the operation mode. Further, the VIDS employs a function (denoted by $\mathcal{F}$) to raise an alarm for an unknown (i.e., significantly distinct from previously observed) fingerprint during the training/operation mode. Let the VIDS raise an alarm for a fingerprint $\mathbf{x}_*$, if $\mathcal{F}(\mathbf{x}_*) > \tau$, where τ represents the detection threshold.

Attack Success. We now formally define the "success" of Duet. Let the class of the victim be y_v, the space of the feature vectors corresponding to the simultaneous transmission of the victim and attacker be $\mathbf{X}_{v+a}$, and the space of the feature vectors corresponding to the simultaneous transmission of the accomplice and attacker be $\mathbf{X}_{a+a}$. In Stage 1, DUET makes the VIDS record a manipulated training set (denoted by $\mathbf{D}_m$) which includes the samples $(\mathbf{x}_{\hat{v}}, y_v)$, such that $\mathbf{x}_{\hat{v}} \in \mathbf{X}_{v+a}$. The classifier learns the manipulated model (denoted by θ_m) using this training set in Stage 1. When the manipulated model is utilized by the VIDS in Stage 3, DUET succeeds if the classifier outputs y_v given the input $\mathbf{x}_{\hat{a}} \in \mathbf{X}_{a+a}$.

Success Rate. The success rate of DUET is limited by four factors. First, DUET needs to ensure that the function of the VIDS to detect unknown fingerprint does not raise an alarm for the distorted fingerprint of the simultaneous transmission employed during Stages 1 and 3. Second, the length of PREP in the victim's message (denoted by L) limits the number of bits that can be corrupted by DUET in each frame of victim. The third and fourth factors are the *message timing accuracy* of the victim's message at the accomplice (denoted by p_{acc}), and that of the accomplice's message at the attacker (denoted by p_{att}). Recall that during Stage 1 of DUET (Figure 3.5), the accomplice estimates the time of transmission of the victim, and the attacker follows the preceded ID frame transmitted by the accomplice. Hence, the values of p_{acc} and p_{att} determine the number of successfully corrupted messages of victim. Also, in Stage 3 of DUET (Figure 3.8), the attacker follows the preceded ID frame transmitted by the accomplice, and hence the value of p_{att} determines the number of successfully corrupted (and forged) messages of the accomplice.

Optimization Problem. The attacker in DUET can control two parameters to enhance its success rate: (1) It can corrupt l_v bits of each of the victim's messages during Stage 1. (2) It can corrupt l_a bits in each of the messages of the accomplice during Stage 3. As such, the problem of finding the best values of l_v and l_a to maximize the probability (denoted by Pr) of DUET's success rate can be defined as:

$$\underset{l_v, l_a \in [0,L]}{\operatorname{argmax}} \ \Pr\left(\frac{y_v = \operatorname{argmax}_{c\in[1,C]} \ \mathcal{O}_d(c \mid \mathbf{x}_{\hat{a}}, \theta_m)}{p_{acc}, p_{att}, \mathbf{x}_{\hat{v}} \in \mathbf{X}_{v+a}, \mathbf{x}_{\hat{a}} \in \mathbf{X}_{a+a}}\right),$$

$$\text{s.t. } \theta_m = \underset{\theta\in\Theta}{\operatorname{argmin}} \ \mathcal{O}_t(\mathbf{D}_m, \theta), \ \mathcal{F}(\mathbf{x}_{\hat{v}}) \le \tau, \ \mathcal{F}(\mathbf{x}_{\hat{a}}) \le \tau.$$

3.5 Implementation Details

Duet Implementation. Duet is written in 1400 lines of C++ code. Utilizing only 950 bytes of memory, and 13 KB of flash storage, Duet is lightweight and can be easily deployed in existing ECUs. The Duet code and an attack demo (on a real car) can be accessed at `https://github.com/CAN-Bus-Duet/CAN-Bus-Duet`.

VIDS Implementation. We evaluate the stealth of Duet against two representative state-of-the-art VIDS: (1) *Viden* [20] with online learning and (2) *Scission* [18] with periodic retraining. The voltage samples for Viden and Scission are collected by sampling the CAN bus at 50 kS/s and 20 MS/s, respectively. Viden computes three features (50^{th}, 75^{th}, and 90^{th} percentiles) using samples corresponding to CAN-H, and three features (10^{th}, 25^{th}, and 50^{th} percentiles) using samples corresponding to CAN-L. Using differential voltage values, Scission computes 24 features (e.g., mean, variance, skewness, and kurtosis) in the time domain, and 24 features in the frequency domain. While Viden utilizes a random forest classifier, Scission employs a logistic regression-based classifier to identify the source of a message. We also consider modified versions of Scission by replacing the logistic regression with other mainstream ML algorithms including support vector machine (SVM), naive Bayes, and random forest classifiers. More details about these VIDS are presented in Section 3.10.

Evaluation Platforms. We evaluate Duet through comprehensive experiments on our lab testbed and two real vehicles.

Lab Testbed. We set up a CAN bus testbed with 10 nodes. Each node is an ECU based on an Arduino UNO board with a 16 MHz Microchip and a SeeedStudio CAN shield with a MCP2515 CAN controller. Two of those nodes play the attacker and accomplice ECUs, one plays the victim targeted by Duet, and the remaining six nodes represent other uncompromised ECUs. The VIDS (i.e., Viden and Scission) run in a laptop connected to the bus. The voltage samples for Viden and Scission are collected by an uncompromised node and the Tektronix DPO2014 oscilloscope (with 200 MHz bandwidth and 8 bits vertical resolution), respectively. The standard CAN

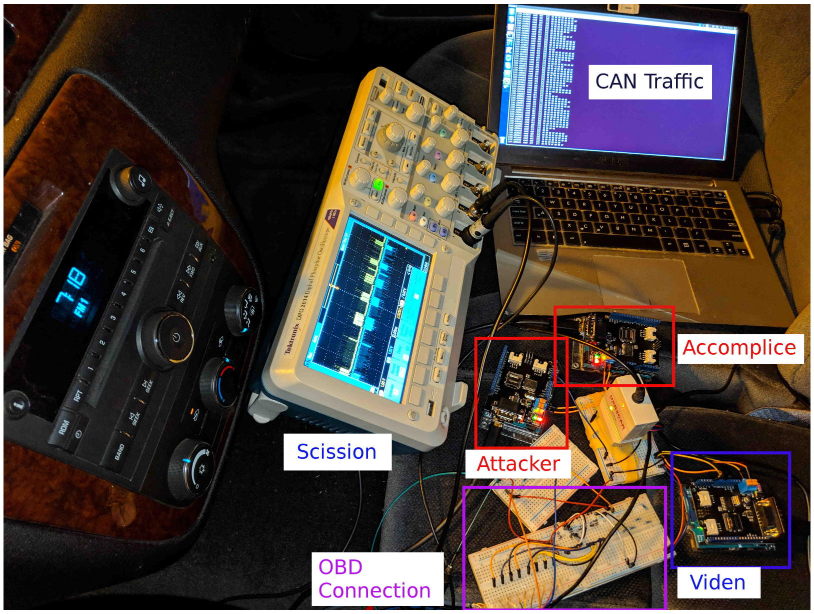


Figure 3.10.: Experimental setup within a real vehicle.

bus speed of 500 kbps is set in the testbed. We follow the benchmark proposed in [65] to generate a total of 60 message with different sizes and periodicity for these nodes, resulting in 50% bus load. We remove the built-in resistances in CAN shields, and terminate the bus with a 120 Ω resistance at each end. We utilize a stub resistance of 2.4 kΩ for connecting the oscilloscope to the bus.

Impala. The 2011 Chevrolet Impala car contains a CAN bus operating at 500 kbps with four ECUs. The CAN bus traffic comprises of 50 messages resulting in 35% bus load. The experimental setup in the Impala is shown in Figure 3.10. Through a customized OBD connector, we connect the vehicular CAN bus to two external ECUs acting as the attacker and accomplice. We also connect another ECU and the oscilloscope for recording voltage samples for Viden and Scission, respectively. With a CAN USB adapter (USB2CAN), a laptop is used for recording the bus traffic.

Cruze. The 2013 Chevrolet Cruze car contains *two* CAN buses operating at 500 kbps: *Bus*-1 supports six ECUs which transmit 88 messages resulting in 61% bus load and *Bus*-2 supports three ECUs which transmit 27 messages resulting in 34% bus

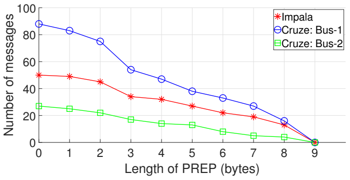


Figure 3.11.: Cumulative number of messages with the given PREP.

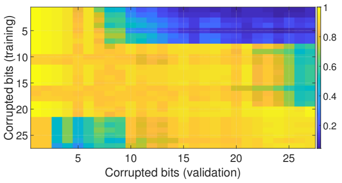


Figure 3.12.: Viden's classification accuracy during Stage 1 of DUET.

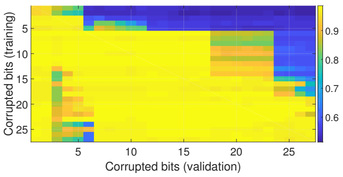


Figure 3.13.: Scission's classification accuracy during Stage 1 of DUET.

load. To record the traffic and voltage samples, we utilize a setup similar to the one shown in Figure 3.10.

3.6 Evaluation of DUET

We first analyze the characteristics of CAN messages which enable the voltage corruption tactic. We then evaluate DUET using four criteria to demonstrate its effectiveness as a masquerade attack: (1) stealth during manipulation, (2) stealth during impersonation, (3) *swiftness* which measures the time required to initialize impersonation, and (4) *persistence* which measures the duration of successful impersonation.

3.6.1 Feasibility of Voltage Corruption

DUET's voltage corruption is feasible only if the attacker is able to transmit simultaneously with the victim's messages beyond their arbitration fields. This condition is readily satisfied by real-world vehicles due to the predictable message timing (due to periodicity) of CAN messages and the existence of PREP in those messages. The generality of these conditions are confirmed by profiling the CAN traffic in five cars (details in Table 3.2 and Section 3.8), and utilizing publicly available reverse-engineering results for recent cars [55]. The generality of PREP is also corroborated by studies of CAN traffic in the existing literature. For example, the researchers in [66] found

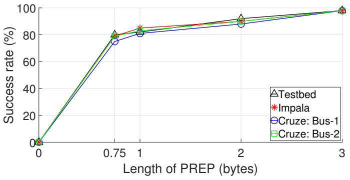


Figure 3.14.: Duet's success rate against Viden on different platforms.

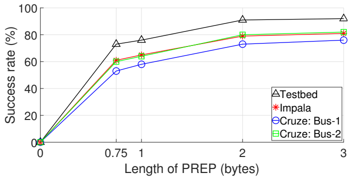


Figure 3.15.: Duet's success rate against Scission on different platforms.

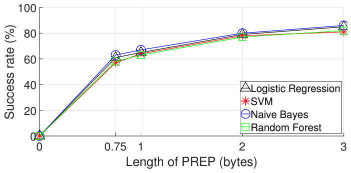


Figure 3.16.: Duet's success rate against Scission with different ML algorithms.

Table 3.3.: Message timing accuracy in Duet.

Platform	**Messages**	**Busload**	p_{acc}	p_{att}
Lab Testbed	60	50%	87%	92%
Impala	50	35%	81%	89%
Cruze: Bus-1	88	61%	60%	85%
Cruze: Bus-2	27	34%	80%	90%

that out of 456 total bytes of payload across all CAN messages in a 2012 Ford Focus, 338 bytes belonged to predictable categories (i.e, constants, counters and multi-valued numbers), and only 118 bytes belonged to unpredictable categories (i.e., sensor readings and unclassified bits). Here, we validate these results through our analysis of the traffic on the three CAN buses in the two experimental cars. For each bus, the data was collected for 15 minutes in the stationary car, and for another 15 minutes in the moving car.

Predictable Message Timing. Table 3.3 presents the average values of timing accuracy of the victim's message at the accomplice (p_{acc}) and the accomplice's message at the attacker (p_{att}), in all four evaluation platforms. Due to the highest busload in the Cruze: Bus-1, its p_{acc} and p_{att} are the lowest among the platforms. We also observe that although the testbed experiments were conducted with higher busload than Cruze: Bus-2 and Impala, its p_{acc} and p_{att} are higher. This is attributed to the larger jitter in message transmission by real ECUs compared with that by ECU prototypes in our testbed. We note that the factors affecting the timing accuracy, which include the number of CAN messages, busload, and message periodicities are independent of the control state of the car, i.e., they do not change while the car is moving or is stationary.

PREP. We calculate the length of PREP in a CAN message as follows. For each bit location after the arbitration field in the message, we calculate the conditional entropy, i.e., the randomness in the current bit given the bits (at the same location in the message) in the previous 16 messages. We consider the bit to be *predictable* if the conditional entropy is less than 0.01. We note that while the conditional entropy is one for a randomly generated bit, it is equal to zero for the bit which remains constant in all messages. The length of PREP is given by the number of bits from the first bit in the control field to the first *unpredictable* bit with conditional entropy more than 0.01. Figure 3.11 presents the *cumulative* number of messages containing different lengths of PREP on the CAN buses of our experimental vehicles. For example, out of 88 messages on Cruze: Bus-1, 75 messages have at least two bytes of PREP. This means that the attacker can readily continue to transmit simultaneously with one of these 75 messages until two bytes after the arbitration field.

3.6.2 Stealth in Stage 1 of DUET against VIDS

Figures 3.12 and 3.13 present the classification accuracy of Viden and Scission, respectively, for different number of corrupted bits in the victim's messages utilized for the training set (y-axis) and those utilized for the validation set (x-axis). These results depict that DUET can effectively trick VIDS into correctly classifying the validation set's messages with *slightly more* corrupted bits after VIDS learn the model from the corrupted training set's messages. Therefore, DUET evades VIDS by enhancing the voltage corruption at a manipulation rate lower than the retraining frequency. In other words, DUET takes downward steps of one bit from the upper-left corner and stays just above the diagonal in these figures so that both VIDS fail to detect the Duet's attempts of manipulating the victim's fingerprint. We note that the fine control over the number of corrupted bits in the victim's messages enables Duet to stealthily carry out the fingerprint manipulation.

Further, Duet deliberately does not corrupt all the victim's messages so that VIDS inherently learn from both corrupted and uncorrupted victim's messages in the training set. This way, as shown in Figures 3.12 and 3.13, the victim's *uncorrupted* messages (missed by Duet due to message timing inaccuracies shown in Table 3.3) in the validation set are correctly classified by VIDS, and will not cause any alarm. These results validate that Duet successfully makes VIDS learn the manipulated fingerprint slowly but stealthily and effectively.

3.6.3 Stealth in Stage 3 of Duet against VIDS

Success Rate. Figures 3.14 and 3.15 present the *per-message success rates* of Duet against Viden and Scission in the four CAN platforms, respectively. We observe that with larger PREP, the distorted fingerprints of the simultaneous transmissions are more distinct from the fingerprints of individual transmissions, leading to higher success rate. The success rate against Viden reaches more than 75% with the corruption of *only* control bits (i.e., PREP = 0.75 byte), and 95% with just three bytes of PREP (Figure 3.14). Similarly, the success rate against Scission reaches more than 50% by corrupting only control bits, and 76% with three bytes of PREP (Figure 3.15). We also observe similar performance of Duet against different ML algorithms (for Scission) as illustrated in Figure 3.16.

Limitation. Duet fails whenever the attacker and accomplice fail to transmit simultaneously in Stage 3 due to imperfect timing (Table 3.3). Hence, Duet's lower success rate on the real cars, in comparison with the testbed, can be attributed to the lower timing accuracy due to higher busload and tranmission jitter. On one hand, the highest success rate against Viden reaches 100% since Viden makes its decision by taking an aggregate of samples received over eight messages. On the other hand, the highest success rate against Scission is limited by the value of p_{att} shown in Table 3.3 as Scission detects the source of each message individually. Nevertheless, the success rates of DUET against VIDS (shown in Figures 3.14 and 3.15) render them ineffective for all practical purposes, specifically in safety-critical applications.

Evading VIDS Alarm. Although DUET's per-message success rate against a VIDS may not be 100%, DUET can still evade it successfully. This is because a real-world VIDS must consider encountering messages that are corrupted by electromagnetic interference and temperature changes, and must minimize the resultant false alarms. For example, Scission employs a mechanism with an *alarm counter* which is incremented by four for each suspicious (wrongly classified) message and decremented by one for each trustworthy (correctly classified) message, raising an alarm if the alarm counter exceeds 200. Such a mechanism of alarm-raising over aggregate traffic makes it *even easier* for DUET to evade Scission, as any per-message success rate of more than 80% would keep the alarm counter near its expected value of zero. As a result, DUET does not cause any alarm on three of the evaluated platforms, and even on the fourth one (Cruze: Bus-1), Scission will raise an alarm only after an average of 1000 $\left(\approx \frac{200}{(1-0.76)\cdot 4-0.76\cdot 1}\right)$ spoofed messages with three bytes of PREP (Figure 3.15). For a brake control message with a 10 ms period, this translates into a sizable 10 seconds of alarm delay, which is sufficient to accomplish malicious activities, such as disabling brakes to cause an accident [8].

3.6.4 Swiftness-Persistence of Stage 2 of DUET

We now evaluate DUET under two more criteria – *swiftness* and *persistence*. While swiftness refers to the speed of the attack in bus'ing-off the victim, persistence refers to the ability to persistently stop the victim's transmission when it is being impersonated. We highlight our contributions by comparing DUET with the original

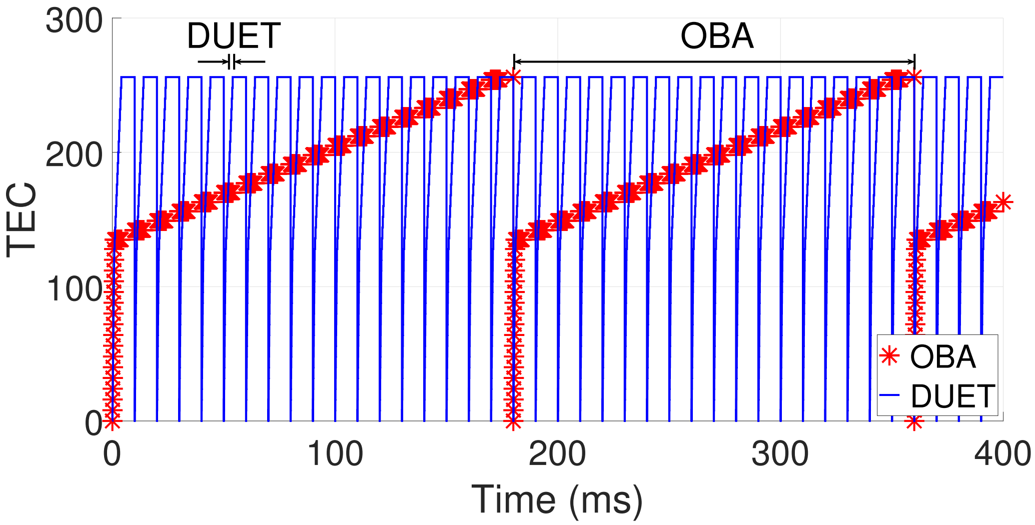


Figure 3.17.: TEC of the victim under OBA and Duet.

bus-off attack (OBA) which represents the state-of-the-art technique for victim suppression [53].

Swiftness. The OBA consists of two phases, the first one to transition the victim to the error-passive state with TEC > 127, and the second for transitioning to the bus-off state. The first phase is instantaneous and finishes within a few milliseconds. However, in the second phase, the attacker is able to target only one victim's transmission at a time thereby increasing the victim's TEC by seven. As a result, OBA takes 18 transmission periods and is extremely slow to bus-off the victim. For example, considering the victim's message with 10 ms periodicity, OBA takes 180 ms time to bus-off the victim in our testbed (Figure 3.17). In contrast, Duet can bus-off the victim by attacking a *single* transmission. In Stage 2, Duet induces 16 active errors initially like the first phase in OBA, but then utilizes the novel passive error regeneration tactic to swiftly push the victim to the bus-off state. The victim's TEC is instantly raised to 256 within 5 ms as shown in Figure 3.17.

Persistence. We measure persistence by the percentage of time the victim is successfully suppressed, i.e., either in the bus-off state or being actively prevented from transmission. Recall that the recovery of a victim ECU from the bus-off state can happen through one of the two methods: (1) after observing 128 instances of 11 recessive bits, or (2) ECU-reset which takes a constant amount of time. While our testbed follows the first method which happens quickly (∼10 ms), the vehicles implement the much longer ECU-reset which takes 150 ms on Cruze: Bus-1, 60 ms on Cruze: Bus-2, and 100 ms on the Impala CAN bus. In both methods, when the victim recovers from the bus-off state, it transmits the buffered messages and then starts its regular communication.

We compare OBA and Duet under both recovery methods for a message with a 10 ms period. Although OBA does not explicitly address victim recovery, we assume that OBA suppresses the victim again when it recovers. While OBA takes a long time to bus-off the victim (180 ms) and allows the victim ample time (175 ms) to perform regular communications, Duet only allows the transmission of one buffered message (0.25 ms) and then pushes the victim to the bus-off state again. In the first recovery method, OBA suppresses the victim for 15 ms (10 ms bus-off and 5 ms suppressed transmission), achieving only $\frac{15}{15+175} = 7.9\%$ persistence. Although Duet also suppresses the victim for 15 ms, it achieves a much higher persistence of $\frac{15}{15+0.25} = 98.4\%$. Similarly, in the second recovery method, OBA achieves persistence of 47.0%, 27.1% and 37.5%, but Duet achieves much higher persistence of 99.8%, 99.6% and 99.8% on the Cruze: Bus-1, Cruze: Bus-2, and the Impala CAN bus, respectively.

3.7 Proposed Defense: RAID

Duet is a powerful masqueraded attack which not only evades existing VIDS, but also evades message (anomaly)-based IDS (MIDS) as demonstrated in Section 3.11. It might be possible however to augment the existing IDS to build a Duet-aware IDS. For example, a VIDS may be modified to record fine-grained changes in the voltage values within message bits, and use it to detect the voltage corruption in Duet. Yet, since Duet is effective even with the corruption of only a few bits, the modified VIDS must find a tedious balance between detecting Duet and avoiding false alarms. We discuss some other defenses against Duet in Section 3.12. However, we go beyond the "attack vs. IDS" arms race to address the root cause that makes Duet feasible, and develop an effective defense which also prevents other potential attacks.

Duet exploits a major deficiency of the CAN protocol: Each message on CAN is allotted a unique identifier (ID) which remains the same for its lifetime. This static nature of the ID enables priority scheduling and deterministic latency for messages on the bus. It also ensures robust arbitration on simultaneous transmissions of two different messages by two ECUs. However, from the adversary's perspective, this means that the same ID is set in the arbitration field of all periodic messages. Such predictability allows the attacker to perform simultaneous transmission with the victim using the same ID. Given the attacker's unrestricted maneuverability during the transmission of control and data fields of the victim's message, Duet succeeds in masquerading the victim.

We propose a novel lightweight defense called Randomized Identifier Defense (RaID) which mitigates the aforementioned deficiency of CAN. Different from (and orthogonal to) IDS, RaID *prevents* Duet by restricting the attacker's fundamental ability to transmit simultaneously with the victim.

RaID Design. The high-speed CANs in vehicles typically utilize the standard frame format (Section 3.8). Under RaID, every ECU on such a CAN upgrades its standard frames to extended frames (Figure 3.1) during VIDS retraining or on occurrence of a transmission error. The 11-bit ID-A field of the standard frame is mapped to the ID-A field of the extended frame. Then, the ID-B field of the extended frame is set as an 18-bit nonce, generated using a cryptographically secure pseudo-random bit generator (PRBG) [67]. At a receiver ECU, while the ID-A field is utilized for the identification of messages, the bits in the ID-B field are discarded. This way, RaID ensures that the sender randomizes the arbitration fields of its messages, which prevents any simultaneous transmission.

RaID Evaluation. We now analyze and evaluate RaID's effectiveness against Duet and its impact on CAN.

Effect on Voltage Fingerprint Manipulation. In Stage 1 of DUET, the attacker needs to predict the arbitration field (which includes ID-A and ID-B fields) of the victim's message utilized for VIDS retraining. With RAID's 18-bit randomness in the ID-B field, the probability that the attacker can find a collision between its guess and the complete arbitration field of the victim's message is 2^{-18}. Thus, the probability of successfully manipulating the voltage fingerprint of any victim during the training mode of a VIDS, which learns the fingerprint over N messages, is $2^{-18 \cdot N}$. This probability is negligibly small for a large N. For instance, RAID reduces the success rate of DUET against Scission (with $N = 200$) to *zero percent* (from at least 76%) on all evaluation platforms.

Effect on Persistent Victim Bus-off. In Stage 2 of DUET, the attacker induces consecutive active errors in an attempt to bus-off the victim. With RAID, upon detecting a transmission error, the victim first upgrades the frame of the re-transmitted message from the standard to the extended format, and then switches back to communicate with the standard frame format after successfully transmitting the message. Therefore, to successfully push the victim to the error-passive state and employ the tactic of passive error regeneration, the attacker must correctly guess the nonce in the ID-B field for at least eigth times, leading to a very low success rate of 2^{-144}. Hence, RAID prevents the victim suppression carried out by DUET.

Table 3.4.: Mapping of attacks into the three-stage attack strategy, and performance of existing VIDS against them.

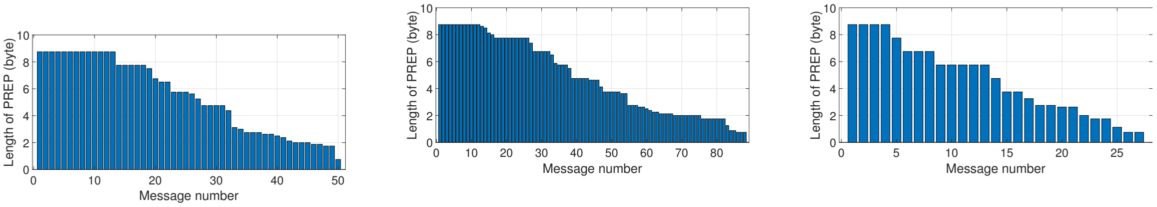
Attacks	**Attack Stages**			**VIDS Defense**
	Fingerprint Manipulation	**Victim Suppression**	**Victim Impersonation**	
Message Impersonation [6–10, 13, 14]	N/A	Unified Diagnostic Services	Unified Diagnostic Services	Detected
Cho et al. [53]	N/A	Original bus-off	N/A	Detected
Sagong et al. [49]	N/A	N/A	Clock skew impersonation	Detected
DUET	Voltage manipulation	Persistent bus-off	Voltage impersonation	*Undetected*

Computation Overhead. The implementation of RAID on a commodity ECU will incur some computation cost in generating the pseudo-random nonce. We estimate this cost with the running time of the built-in PRBG in the Arduino UNO board [68], a single invocation of which is able to generate 32 random bits in 50 μs. We note that PRBG can be executed during the ECU idle time, and the results can be stored and used when needed (i.e., during VIDS retraining and on transmission errors). Hence, RAID effectively produces negligible computation overhead during message transmission.

Communication Overhead. Since a standard frame is shorter than its extended version, RAID suffers communication overhead during VIDS retraining. This increases the bus-load which we measure via simulation, based on *real* traffic traces collected from the three CAN buses in our vehicles. RAID increases the load of the three buses by 13% (61% to 74%, for Cruze: Bus-1), 7% (34% to 41%, for Cruze: Bus-2), and 7% (35% to 42%, for Impala CAN bus) respectively. RAID also increases the end-to-end message latency by 50 μs (25 bit-periods on a 500 kbps CAN bus) which is in an acceptable range of the general automotive deadline tolerances [69].

3.8 CAN Traffic Analysis

DUET exploits three key characteristics of CAN messages: (1) static identifiers, (2) periodic messages, and (3) predictable message content. In automobiles, messages (with information related to engine, brake, steering and other critical equipments) are exclusively transmitted on the high-speed CAN bus in the standard format. Conventionally, each particular type of message always contains the same ID. This characteristic is motivated by the safety-critical requirement of providing robust message arbitration and minimizing the worst-case delay (with theoretical guarantees) in the communication of messages [69]. Further, the messages on the CAN are periodic because the fine granularity of periodic message communication is required for making safety-critical collaborative decisions (e.g., control of accelerator, brake and steering)



(a) 2011 Chevrolet Impala.

(b) 2013 Chevrolet Cruze: Bus-1.

(c) 2013 Chevrolet Cruze: Bus-2.

Figure 3.18.: Length of PREP in the CAN messages of the experimental vehicles.

in the vehicles. Hence, the next time-of-transmission of a periodic message can be readily estimated by knowing the current time-of-transmission of the message on the bus. The third key characteristic is the existence of PREP in the CAN messages which makes the voltage corruption tactic in DUET feasible. We validate these characteristics by analyzing the CAN traffic in two experimental and three non-experimental vehicles.

3.8.1 Experimental Vehicles

We have performed extensive experiments on the CAN buses of two vehicles: 2011 Chevrolet Impala and 2013 Chevrolet Cruze. We note that while the non-periodic messages (e.g., door lock/unlock) are usually communicated on the *low-speed secondary* CAN bus or local interconnect network, most of the safety-critical messages (e.g., brakes, steering, and engine speed) are communicated periodically on the *high-speed primary* CAN bus. We record and analyze the CAN traffic by connecting our experimental setup (Figure 3.10) to the primary CAN buses of these vehicles using their OBD II ports. While we could access only one primary CAN bus in the Impala, two CAN buses are accessible in the Cruze. The traffic traces from these vehicles are available at [70].

While there are 50 messages on the Impala CAN bus, there are 88 messages on the Bus-1 and 27 messages on the Bus-2 in the Cruze. All messages in these

Table 3.5.: Message periodicity in experimental vehicles.

2011 Chevrolet Impala (50 messages)

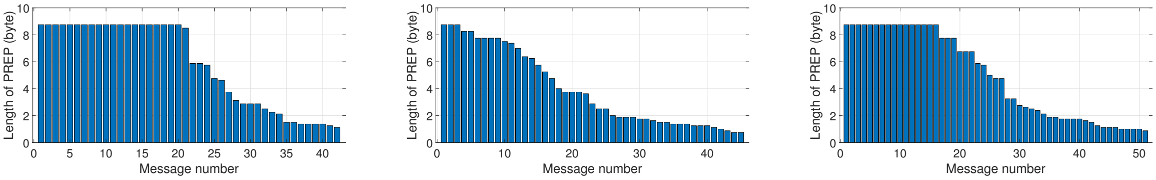
Period (ms)	**9**	**10**	**12.5**	**18**	**25**	**30**	**50**	**100**	**250**	**500**	**1000**	**5000**
No. of messages	3	1	4	4	4	3	4	9	3	3	11	1

2013 Chevrolet Cruze: Bus-1 (88 messages)

Period (ms)	**10**	**12.5**	**20**	**25**	**50**	**100**	**250**	**500**	**1000**	**5000**		
No. of messages	6	10	6	14	8	16	4	6	17	1		

2013 Chevrolet Cruze: Bus-2 (27 messages)

Period (ms)	**10**	**20**	**50**	**100**
No. of messages	10	5	2	10



(a) 2012 Toyota Camry.

(b) 2012 Honda Civic.

(c) 2010 Dodge Ram.

Figure 3.19.: Length of PREP in the CAN messages of non-experimental vehicles.

three CAN buses are transmitted in the standard frame format with static identifiers. Also, all message in these two vehicles are periodic with their periodicity presented in Table 3.5. Further, each message with a specific ID is transmitted with the same length of the message payload, which means that the bits in the control field remain

Table 3.6.: Message periodicity in non-experimental vehicles.

2012 Toyota Camry (42 messages)

Period (ms)	**10**	**20**	**30**	**100**	**200**	**300**
No. of messages	4	3	5	1	2	2
Period (ms)	**500**	**1000**	**2000**	**5000**		
No. of messages	3	20	1	1		

2012 Honda Civic (45 messages)

Period (ms)	**10**	**20**	**40**	**100**	**200**	**300**
No. of messages	11	7	6	11	2	8

2010 Dodge Ram (51 messages)

Period (ms)	**10**	**20**	**50**	**60**	**100**	**200**
No. of messages	3	17	1	1	11	1
Period (ms)	**300**	**500**	**1000**	**2000**		
No. of messages	3	3	7	4		

the same. Additionally, we observe the message payloads contain constants, counters and predictable contents. Figure 3.18 presents the length of PREP for messages on the three buses. We observe that most of the messages on these CAN buses have at least one byte of PREP.

3.8.2 Non-Experimental Vehicles

We also analyze the CAN traffic data (published at [71] by other independent researchers) of three other vehicles: 2012 Toyota Camry, 2012 Honda Civic, and 2010 Dodge Ram. For the convenience of readers, these traces of CAN traffic are also made available at [70]. While the Camry has 42 messages on its CAN bus, there are

Algorithm 3: Victim recovery timing.

```
                                    ▷ Get time at bus-off, in number of bit-periods
busOffTime $T_0 \leftarrow getCurrentTime()$
timeSpentReceivingFrames $T_f \leftarrow 0$
numReceivedFrames $N_f \leftarrow 0$
num11bitSequences $N_s \leftarrow 0$
                                    ▷ Recovery at 128 instances of 11 recessive bits
while $N_s < 128$ do
    if $CAN.availableFrame()$ then
        CANFrame $frame \leftarrow CAN.readFrame()$
        $N_f \leftarrow N_f + 1$
                                    ▷ Transmission time for received CAN frame
        $T_f \leftarrow T_f + transmissionTime(frame)$
                                    ▷ Each received frame also provides 11 recessive bits
    $N_s \leftarrow \frac{getCurrentTime() - T_0 - T_f}{11} + N_f$
```

45 messages in the Civic. All the message in the Camry and Civic are periodic with their respective periodicity shown in Table 3.6. In the Ram, we observe 51 periodic messages and 4 non-periodic messages. Also, each message in these three vehicles is transmitted in the standard frame format. Further, Figure 3.19 illustrates the distribution of lengths of PREP in the CAN messages of the three non-experimental vehicles. We observe that most of the messages have at least one byte of PREP. Additionally, when compared with the Civic, the Camry and the Ram have significantly higher number of messages in which all data bits can be readily predicted resulting in the PREP of 8.75 bytes.

3.9 Victim Recovery Timing

To re-bus-off the victim, the DUET duo must measure the time of the victim's recovery by counting the sequences of 11 recessive bits as described in Algorithm 3.

3.10 Details of VIDS

We evaluate DUET against two state-of-the-art VIDS: (1) Viden [20] and (2) Scission [18], which have practical fingerprinting technique that can be implemented on a device less powerful (and costly) than an oscilloscope. We do not evaluate *VoltageIDS* [19] as it requires a high sampling frequency of 2.5 GS/s, making it impractical for commodity vehicles. Further, we do not evaluate SIMPLE [48] since it fundamentally employs a subset of features employed in Scission.

Viden. We follow the details provided in [20] to implement Viden. Due to the low sampling rate of 50 KS/s in Viden and the need to relate the samples back to the message ID in real-time, the number of measured voltage samples per message is limited to eight samples total, four for CAN-H and four for CAN-L. Viden employs three features for CAN-H (50th, 75th, and 90th percentiles of CAN-H samples) and another three features for CAN-L (10th, 25th, and 50th percentiles of CAN-L samples). Each of the three features in CAN-H or CAN-L is calculated over at least 30 voltage samples. Hence, one set of the six features is obtained over eight messages. During the training mode, we record CAN-H and CAN-L voltage samples corresponding to 1600 messages transmitted by each ECU. We compute the thresholds (corresponding to CAN-H and CAN-L) to exclude the samples corresponding to the acknowledgement. For each ECU, we then compute 200 sets of the six features. We finally train a 200-tree random forest classifier with these six features. During the operation mode, we collect the voltage samples of eight messages on the CAN bus, compute the six features from those samples and classify their source using the trained model.

Scission. We follow the details provided in [18] to implement Scission. During the training mode, we record differential voltage samples corresponding to 200 messages transmitted by each ECU. Using the samples in each message, we compute the 24 time-domain features and another 24 frequency-domain features for each message. We then obtain the standardized values for each of the features. We pre-process the features to find the 18 most significant features by utilizing the Relief-F algorithm.

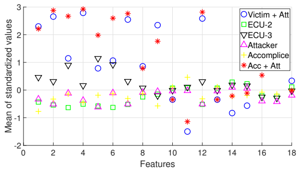


Figure 3.20.: Effect of DUET on the features utilized in Scission.

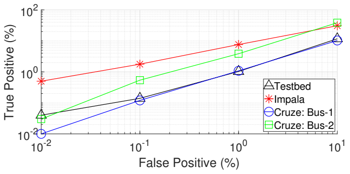


Figure 3.21.: ROC curves for Muter-MIDS against DUET.

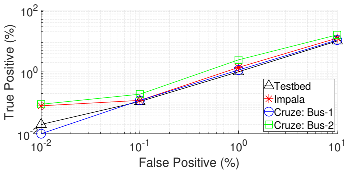


Figure 3.22.: ROC curves for Song-MIDS against DUET.

We finally train a multinomial logistic regression model with the 18 features. During the operation mode, we record voltage samples of a message on the CAN bus, compute the 18 features of the message and classify the source of the message using the trained model.

Illustrative Experiment. We illustrate the stealth of DUET by presenting the results of an experiment with Scission. In this experiment, we launch DUET on one of the three ECUs in the Cruze: Bus-2 using our experiment setup (Figure 3.10). We then record the values of the 18 features employed in Scission corresponding to the victim ECU, the two remaining benign ECUs, the attacker and the accomplice. Figure 3.20 presents the mean of the standardized values of the features utilized in Scission where the length of PREP is three bytes. We observe that the values corresponding to "victim + attacker" (recorded during the training mode) and "accomplice + attacker" (recorded during the operation mode) are significantly different from values corresponding to other ECUs. Hence, after Scission learns the manipulated fingerprint of the victim by utilizing the features of "victim + attacker" in the training mode, it classifies the fingerprint of "accomplice + attacker" as that of the victim during the operation mode.

3.11 Stealth Against MIDS

Potential Anomalies. Unlike existing attacks [6, 8, 9, 13, 53], Duet evades MIDS presented in prior art [11, 22, 23, 66, 72] by employing novel strategies, specifically the passive error regeneration tactic and the one-shot transmission mode. Some MIDS [66, 72] watch for inconsistencies in payloads of CAN messages. However, Duet strategically evades them by persistently suppressing all but one transmission from the victim (Section 3.3.2) and replacing them with forged messages. MIDS [11, 22, 23] which detect message injection by noticing an increased frequency of messages also fail to detect Duet as neither the attacker nor the accomplice inject any extraneous message on the bus. Further, Duet does not cause any inconsistency in features related to CAN protocol and CAN frame format [22]. In fact, the Duet duo carry out their malicious activity while adhering to the CAN protocol.

In terms of CAN bus traffic, the only anomalous behavior revealed by the Duet duo is the change in transmission time of their own messages. Hence, only MIDS which examine the fine-grained inter-message transmission time, like Muter-MIDS [22] and Song-MIDS [23], can potentially detect Duet. To evade such MIDS, Duet needs to conceal two potential anomalies in the timing of messages: (1) preceded ID frames enabling simultaneous transmissions in the three stages of Duet, and (2) the *clutter traffic* blocking the transmission of the victim's passive error frame in Stage 2. The accomplice requires one preceded ID frame in each of the three stages, while the attacker requires only one preceded ID frame in Stage 2. For these preceded ID frames, the attacker and accomplice can delay/expedite any of their benign messages (recall these are compromised ECUs) with the same periodicity. In Stage 2, a total of 14 messages are required as clutter traffic. The attacker and accomplice can again delay/expedite any of their benign messages to clutter the bus. Further, regular messages from other ECUs also act as clutter traffic. Optimal solutions (with minimum anomalies on the bus) of these problems are required for evading MIDS. We note that

Table 3.7.: Messages belonging to three ECUs on Cruze: Bus-1.

	ECUs		
Period (ms)	**Victim**	**Attacker**	**Accomplice**
10	0	5	0
12.5	7	0	2
20	0	5	0
25	2	0	5
50	4	6	0
100	2	4	0
250	3	1	0
500	3	0	1
1000	0	5	1
5000	0	1	0

such solutions can be easily found offline by reverse engineering the same/similar type of vehicle. Below we present a concrete example of selecting such clutter traffic.

Illustrative Experiment. We present the periodicity of messages transmitted by three ECUs on Cruze: Bus-1 in Table 3.7. We assign those ECUs the roles of the victim, the attacker and the accomplice, and explore the potential messages that could be utilized in DUET for evading MIDS. Let VIDS utilize one of the victim's message with 12.5 ms period for training. In that case, one of the two messages of the accomplice with periods of 12.5 ms can be selected as the preceded ID frame during Stage 1 of DUET without alerting MIDS. In Stage 2 of DUET, let the attacker target one of the victim's messages with period of 12.5 ms. Note that after one instance of bus-off, the victim will remain in sleep for 150 ms. In this case, one of the 11 messages of the attacker with periods of 10 ms and 50 ms can be selected as the preceded ID-1 frame (Figure 3.6) to synchronize with the victim's message. Also, one of the seven accomplice's messages with periods 12.5 ms and 25 ms can be utilized as the preceded ID-2 frame. Further, the attacker and accomplice can utilize

23 messages with lower periods than 50 ms to fill the traffic on the bus to block the transmission of the victim's passive error frame. Thereafter, in Stage 3 of Duet, the accomplice transmits the messages belonging to the victim at desirable instances of time. This means that similar to Stage 1 of Duet, the accomplice can select appropriate preceded ID frames among its messages which do not raise an alarm at MIDS.

Implementation. We evaluate Duet against two state-of-the-art MIDS: (1) Muter et al. [22] (*Muter-MIDS*) and (2) Song et al. [23] (*Song-MIDS*). Muter-MIDS employs a binary test to check whether the inter-message transmission time corresponding to a received CAN message lies within the defined upper and lower bounds. Then it accumulates the number of such anomalies (failed tests) over a specified period of an *inspection window*. Song-MIDS records an anomaly if it detects two back-to-back messages (with zero bus-idle period) over a similar inspection window. Both MIDS raise an alarm when the number of anomalies recorded over the inspection window exceeds a detection threshold. In our evaluation of Muter-MIDS, we set the upper bound as $\mu + 2\sigma$ and lower bound as $\mu - 2\sigma$, where μ and σ represent the mean and standard deviation of inter-message transmission times, respectively. We also utilize an inspection window of 0.5 s. The two MIDS run in a laptop connected to the bus.

Evaluation Results. We present the performance of Muter-MIDS and Song-MIDS against Duet in Figures 3.21 and 3.22, respectively. In these figures, receiver operating characteristic (ROC) curves are generated by modifying the detection thresholds employed by MIDS. Here, a true positive refers to the event in which MIDS successfully detect Duet, and a false positive refers to the event in which MIDS raise a false alarm for an attack after inspecting benign CAN traffic. Our results illustrate that for both Muter-MIDS and Song-MIDS, the true positive rate is less than 1% (i.e., the success rate of Duet in evading MIDS is more than 99%) at 0.01% false positive rate (which translates to one false alarm per 1.5 hours). Duet comfortably stays below MIDS's radar since the clutter traffic is required only when the victim recovers from

Table 3.8.: Computation cost (in μs) for MAC schemes.

Algorithm	**Hashing cost (per byte)**	**Finalization cost (per operation)**	**Total cost (8-byte data)**
SHA512	130.42	16938.76	17982.06
SHA256	44.75	2895.89	3253.89
SHA3_512	112.82	8116.07	9018.63
SHA3_256	60.13	8099.97	8581.01
Blake2b	73.34	9397.35	9984.07
Blake2s	20.40	1317.48	1480.68
GHASH	74.68	9.1	606.54
Poly1305	24.67	463	857.56
AES128 (CBC-MAC)	-	-	488
AES192 (CBC-MAC)	-	-	588
AES256 (CBC-MAC)	-	-	688

the bus-off state. In fact, for a MIDS to be deployable in a real car, it will need to have a much lower false positive rate which will imply higher success rate for DUET.

3.12 Potential Defenses against DUET

Message Authentication Code (MAC). Conventionally, a cryptographic MAC is utilized to defend against masquerade attacks. To explore the feasibility of a MAC scheme in CAN, we utilize the Arduino UNO board to evaluate multiple MAC schemes [73]. Our evaluation includes the hash-based schemes as well as the block cipher-based schemes. For the hash algorithms, the total computation cost is the sum of hashing cost and finalization cost. For each hash algorithm, although the hashing cost increases with the length of the message data, the finalization cost is fixed per operation of the algorithm, regardless of the number of data bytes. We consider a CAN message with eight data bytes to calculate the total computation cost of generating a MAC with each algorithm. We present our results in Table 3.8.

We note that a MAC scheme requires a secure key agreement protocol and counter synchronization mechanism among ECUs which are difficult to perform on the resource-constrained CAN [16, 73]. Unlike the MAC scheme, the ECUs in RAID do not need to share any key to generate the nonce. Further, the generation and verification of the authentication code requires the resource-constrained ECUs to perform computationally-intensive cryptographic operations. For example, computing the hash-based MAC on an Arduino Uno with $SHA256$ and $SHA3\_256$ takes 3.25 ms and 8.58 ms, respectively (Table 3.8). Furthermore, the communication overhead (>0.25 ms) of MAC on the bandwidth-limited CAN bus is non-trivial [74]. This means that the end-to-end message latency in a MAC scheme is significantly higher than that in RAID. Hence, we assert that a VIDS complemented with RAID is a more practical solution than a MAC scheme for protecting the CAN bus against masquerade attacks.

Transmission Time Randomization. The simultaneous transmission of the victim and attacker in Stage 1 of DUET might be deemed preventable by randomizing the transmission time of the victim. However, the accomplice may readily counter such a defense and re-enable the simultaneous transmission by employing *multiple* (instead of just one) preceded ID frames. Moreover, transmission time randomization adversely affects the priority scheduling of messages on CAN, and may result in significantly degraded worst-case real-time responsiveness of the system [69]. RAID does not suffer from such adverse effects since the message priority is preserved by mapping the bits of the ID-A field in the standard frame to those in the extended frame.

Message (Anomaly)-Based Defense. Another solution that a MIDS might implement is to record the number of bit-errors on the bus and track the victim's TEC to detect when a victim is suppressed. However, unlike RAID, such a defense will only detect the victim suppression and cannot prevent it.

3.13 Discussion

Relevance. Prior efforts [8–10] have illustrated various techniques for compromising in-vehicle ECUs. We build upon these efforts and present a powerful ECU masquerade attack, Duet, which follows an APT-style CAN attack strategy and evades any training-based VIDS by poisoning its training set.

Generalisability. The various CAN characteristics leveraged by Duet are native in all CAN buses, e.g., the periodicity of CAN messages and predictability of message contents are commonly observed, as confirmed by our study of CAN traffic from five different vehicles (Table 3.2). Finally, the flaws that enable our tactics are also fundamental to the CAN protocol and not specific to any vehicle model/maker.

Secure VIDS Retraining. To provide a secure training mode for VIDS, Kneib et al. propose to utilize message authentication along with other existing defenses against training set poisoning attacks [18]. We note that Duet will successfully manipulate the fingerprint of the victim's messages *with* authentication since the attacker can still corrupt voltage samples in the payload. Also, existing defenses against poisoning attacks utilize techniques to remove outliers in the training samples [51, 52]. Since Duet can poison a significant portion (>50%) of the training samples (as presented in Table 3.3), the poisoned samples are no longer outliers and cannot be eliminated by the such defenses. We do acknowledge that the encryption of the payload may limit the length of the PREP to 0.75 byte, i.e., only the bits in the control field. As shown in Figures 3.14 and 3.15, this may lower the success rate of Duet, but *cannot* completely prevent Duet.

Retraining Frequency. In Stage 1, Duet increases the number of corrupted bits (from zero to the desired bits) in the victim's messages at a rate slower than the retraining frequency of VIDS. As such, on one hand, Stage 1 of Duet will quickly conclude against an online training-based VIDS, e.g, within a few seconds against Viden [20] which learns from each message. On the other hand, it will take a relatively longer time to conclude against a periodic learning-based VIDS, e.g., within a few days against Scission [18] which may have a daily training schedule. Nevertheless, Duet tricks VIDS into learning an increasingly manipulated fingerprint of the victim and ensures that the fingerprint is manipulated stealthily (Section 3.6.2). We point out that such "low-and-slow" nature of Duet is aligned with other in-vehicle APTs [75].

3.14 Summary

We have presented Duet, a stealthy, multi-stage ECU masquerade attack strategy that successfully evades state-of-the-art VIDS defenses. Through evaluation results from real CAN buses and ECUs, we demonstrate the power of the "attacker + accomplice" duo – in comparison with existing "lone-attacker" attacks – in distorting the victim's voltage fingerprint (which is considered "immutable" in prior research), suppressing the victim, and impersonating the victim for message spoofing. All these attack stages are enabled by generic attack tactics. By proposing the RaID defense against Duet, we advocate the development of orthogonal (to IDS), cost-effective defenses that break away from the "attack vs. IDS" arms race.

4 CANDID: PROTECTING CONTROLLER AREA NETWORK VIA INTRA-NETWORK DIALECTING

The state-of-the-art automobiles are equipped with hundreds of sensors and actuators which are administered by electronic control units (ECUs) including brake, engine and steering control units. These ECUs employ the controller area network (CAN) protocol for real-time communication and coordination while performing safety-critical automotive functions such as adaptive cruise control and lane keeping assist. Although CAN has been designed to be robust against electromagnetic interference and random errors, it has very little, if any, protection against malicious attacks (e.g., issuing a malevolent command to an actuator, reporting a wrong sensor value or disabling a legitimate ECU) [10].

The lack of even fundamental security countermeasures make CAN vulnerable to a range of traditional attacks such as denial-of-service, replay, and message injection [5–12]. These attacks exploit the static message identifier/header, the predictable message payload, and deterministic transmission time of periodic messages. Through these attacks, an attacker can easily disrupt/disable legitimate ECUs at will [53, 76], and inject messages to manipulate crucial automotive components, e.g., brake, engine and steering. Moreover, these attacks can be easily replicated not only to similar vehicles, but also to other automotive systems because of the uniformity provided by the CAN protocol.

To secure CAN, researchers have proposed to employ data encryption and authentication by utilizing the link layer features [74, 77–79], and additional hardware support [80, 81]). However, the traditional data encryption and authentication schemes lead to unacceptable end-to-end latency for messages as they put on additional communication overhead to the bandwidth-constrained CAN bus and computation overhead to the resource-constrained ECUs. Moreover, to limit the worst-case response time, the existing literature provides no mechanism for encrypting the message identifier and altering the transmission time of messages. Lightweight defense techniques utilizing intrusion detection systems (IDS) have also been proposed [11, 21, 82–84]. These IDS are able to detect compromised ECUs injecting malicious messages, or newly added ECUs on the bus.

However, these defenses have many shortcomings. First, all proposed cryptographic solutions are computationally expensive and impractical for resource-constrained environments. Second, novel attacks such as bus-off [53] and DUET are still effective, being able to utilize the static components of CAN messages, such as identifier, payload, and the IFS (inter-frame space), which is the fixed duration between two consecutive messages. Third, the homogeneity of CAN configuration across all vehicles of the same make and model and the static nature of various components allows an attacker to reverse-engineer a single vehicle offline and easily scale any attack.

To solve these shortcomings, we propose CANDID, a protocol dialecting defense. By modifying various components of CAN messages, CANDID creates a unique "dialect" for each vehicle. This limits the scalability of any attack to a single vehicle. Further, by modification of the id and IFS fields, attacks such as bus-off and DUET are prevented. Any modification of various CAN components is called a policy, and a dialect contains a list of different policies applied in a deployment. CANDID provides a software-only solution that can be utilized by manufacturers to provide unique dialects for each deployment. By providing a modified CANDID-enabled CAN library, CANDID can be easily used by the manufacturers without modification to their application code. For deployment, CANDID includes a separate *Dialect Controller* ECU which facilitates dialect execution and verifies dialect compliance, and a *Dialect agent* on each ECU in the deployment.

While executing a dialect in a deployment, CANDID faces various challenges. First, to be able to control IFS, CANDID must be able to accurately initiate transmission when the ECU is idle. Second, to be able to apply dynamic modifications to the CAN messages, CANDID must update a buffered message for transmission whenever a new message is received. Third, modification of various fields such as identifier and IFS hinders their purpose, which is arbitration and synchronization for arbitration respectively. Fourth, not all dialects can be implemented in a CAN bus environment because of resource constraints, and we must identify if a dialect can be deployed in a particular CAN bus environment.

To solve these challenges, CANDID makes various contributions. First, CANDID performs ECU state tracking by utilizing features that are available from current off-the-shelf hardware. Second, to free up identifier from its role of arbitration, we introduce the concept of utilizing IFS for arbitration, without sacrificing worst-case deadline guarantees. Third, to enable efficient cryptographic policies, we maintain and utilize a synchronized stream-cipher. Fourth, we provide a time constraint that can be used to verify if a dialect can be executed in a particular CAN bus environment.

To evaluate CANDID, we have showcased dialects with various policies over two different attack models as case studies, against a compromised existing ECU and an external ECU. The two case studies describe various policies highlighting the importance of modifying CAN components and the impact of CANDID. Our results show that CANDID requires minimal computational and communication resources. Further, the dialects in our case studies do not introduce any extra network traffic, and impact the inter-message arrival rate with a smaller jitter than which occurs naturally in a vehicle.

We summarize our main contributions as follows.

- We propose CANDID, a protocol dialecting defense to address various shortcomings in the CAN environment.
- By utilizing ECU state tracking by a Dialect agent, CANDID allows unique dialects in a CAN bus which can be verified by a Dialect controller.
- CANDID enables new capabilities, such as utilizing IFS for arbitration, and a synchronized stream cipher for cryptographic solutions.

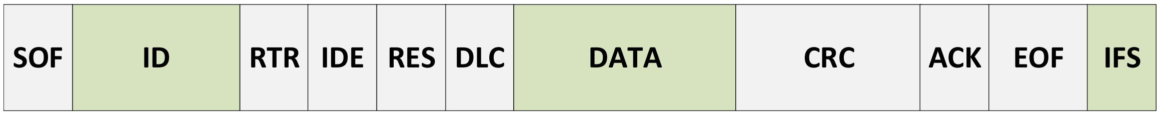


Figure 4.1.: CAN message format.

4.1 Background

CAN Frame. CAN messages are communicated in either standard or extended format as shown in Figure 4.1. While a message in the standard frame format is identified by an 11-bit identifier in the ID-A field, a message in the extended frame format is identified by a 29-bit identifier which is composed of two fields: an 11-bit ID-A field and an 18-bit ID-B field. In both formats, a message carries the payload in the data field which can be of length from zero to eight bytes as indicated in the data length code (DLC) field. In CAN, the bit values 1 and 0 are called recessive and dominant bits, respectively. CAN employs a twisted wire pair called *CAN-H* and *CAN-L*. The recessive and dominant bits are represented by the differential (between CAN-H and CAN-L) voltages of 0 V and 2 V, respectively.

Protocol Stack. The protocol stack followed by an ECU can be broadly classified into three components: application, controller and transceiver. After generating the data for a particular sensor/actuator (e.g., steering status), the application software assigns an identifier to the message, and then pushes the identifier and the payload to the transmit buffer in the controller. Thereafter, the CAN controller is responsible for calculating cyclic redundancy check (CRC) bits, properly formatting the frame and transmitting the message as per the CAN protocol. Finally, the transceiver is responsible for setting the bus voltage for transmitting each bit. While receiving a message, the transceiver reads the bus voltage and decodes it to a bit. After accumulating a message in the receive-buffer, the controller checks the correctness of the frame format and the CRC bits. If the controller does not detect any error, the message identifier and payload are forwarded to the application.

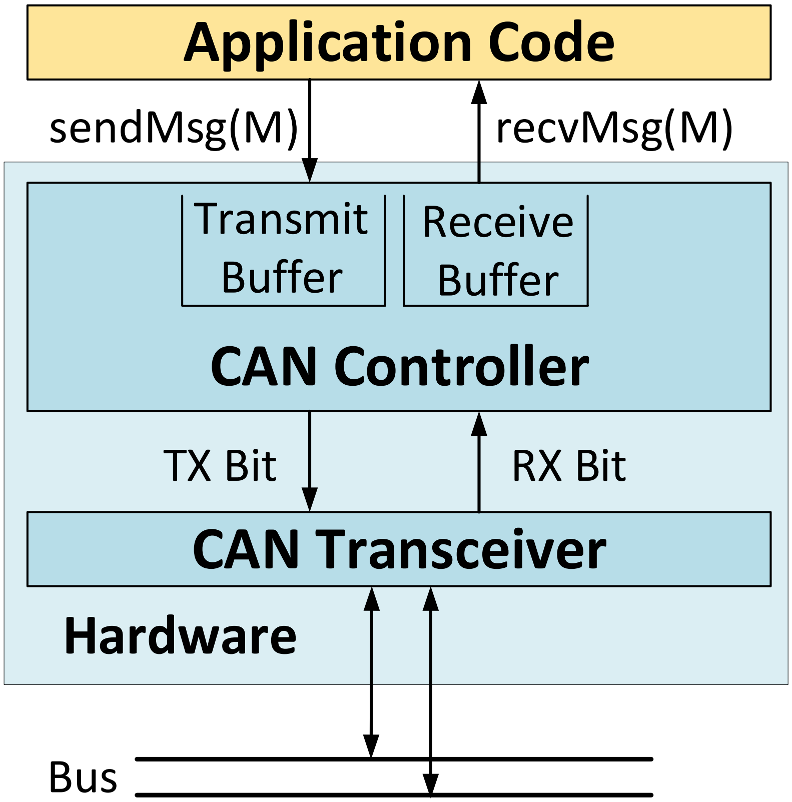
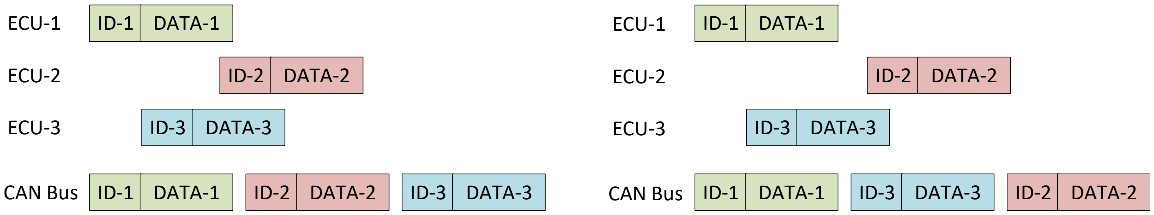


Figure 4.2.: Overview of CAN.

ID-Based Arbitration. When the controllers of two ECUs attempt to transmit their messages (in their transmit-buffers) on the bus at the same time, they employ the following arbitration procedure which allows only one ECU to transmit its message on the bus. Each ECU starts transmitting on the bus by synchronizing with the start-of-frame (SOF) field. For the ID fields, each ECU sends an ID bit on the bus, and then reads it back from the bus. If an ECU reads a dominant bit while it transmitted a recessive bit, it loses the arbitration, and stops transmitting. Since each of the messages transmitted by ECUs is allotted a *unique* ID, only one ECU wins the arbitration and continues to transmit the rest of the frame.



(a) MessageSequenceA. (b) MessageSequenceB.

Figure 4.3.: ID sequence.

4.2 Motivation

Lack of Stream Cipher-Based Solutions. Many CAN bus nodes are incapable of supporting cryptographic solutions. According to our measurement study, many ECUs in control systems are based on low-end microcontrollers (e.g., STMicroelectronics STM8 [85] with 16 MHz frequency and 8 KB RAM) which cannot perform cryptographic computation and preserve real-time properties. For instance, it takes the STM8 controller more than 1 ms to encrypt each payload with AES-CTR and 3 ms to authenticate with SHA256. Unfortunately, many systems require the CAN bus to achieve significantly higher throughput (up to 1 Mbps). CAN message have real-time constraints, which are defined as the deadlines for receiving messages. For instance, many car manufacturers require that an ECU can receive a CAN frame, process it, and respond within a 1 ms window. Due to its resource and real-time constraints, ECUs call for more lightweight approaches than traditional cryptographic solutions for securing the network and consequently the physical system.

Traditional cryptography-based defenses are unaffordable in CAN because of the resource constraints of low-end ECUs, stringent latency requirements of CAN messages, and bandwidth-constraints of CAN bus. One lightweight solution that is potentially suitable for CAN is to employ a fast stream cipher (e.g., ChaCha) or a fast block cipher (e.g., Speck) turned into a stream cipher through cipher feedback mode, output feedback mode or counter mode. Through such a solution, the ECUs can generate a keystream *without waiting* for the payload (plaintext), and then the keystream can be simply XORed with the payload to get the ciphertext. This ensures that the payload does not suffer from the delay caused in the computation of the keystream. However, this solution cannot be readily implemented in CAN because ECUs cannot synchronize and pipeline their messages on the CAN bus. For instance, for two cases shown in Figure 4.3a and Figure 4.3b, the keystream must be different, and the payload of the messages with ID-2 and ID-3 must be updated. However, they have already been pushed in the transmit buffers and will transmit with wrongly encrypted plaintext.

Modification of Static CAN Components. One of the problems with CAN bus deployments has been the use of static IDs and payload format. This homogeneity allows an attacker to reverse-engineer a vehicle and scale an attack to all vehicles of the same make and model. A software diversification approach of modification of various CAN components to create a unique protocol dialect for each vehicle would reduce this scalability. However, modification of various CAN message components such as ID and IFS faces challenges because of their unique functions.

ID field serves the purpose of arbitration on the CAN bus. Any modification of ID values that does not preserve the priority order will violate the worst-case response time guarantees that a real-time system is carefully designed around. However, preserving the priority order for ID values leaves them easily reverse engineered. To solve this issue, we must decouple priority from the ID field.

Although the CAN standard sets IFS between two consecutive messages to be exactly 3, we note that the CAN standard is not violated for IFS values to be set higher. We also note that if two messages with different IFS attempt transmission together, the one with the lower IFS will transmit first, and will have a higher priority level. Interestingly, IFS-based arbitration can still provide the same worst-case deadline guarantees as with ID-based arbitration, as only messages of same and lower priority levels can transmit before a pending transmission, regardless of the ID values. Note that this does not require any modification to the CAN hardware, and is perfectly compliant with the CAN protocol. Hence, usage of IFS for arbitration frees up the ID field for various use cases, while still providing the same deadline guarantees. However, control of the IFS field is not possible with current CAN libraries or hardware.

4.3 Policies and Dialects

CANDID provides a software-only framework to allow run-time modification of ID, Data, and IFS fields of the CAN messages. The run-time nature of the modifications allows them to dynamically change with the traffic on the CAN bus. This is effected by maintaining a *context* of the bus as a function of the past CAN bus traffic. The context can then be utilized for applying modifications on CAN messages. The context-dependent modification of CAN messages enabled by CANDID allows various applications which would not be practical otherwise. For example, per-message encryption is cost-prohibitive for a CAN environment. However, CANDID enables efficient symmetric encryption by utilizing a shared synchronous stream cipher. The stream cipher's keystream is kept synchronized by utilizing the number of messages transmitted on the bus (context) as the offset in the shared keystream. We discuss this application further in Case Study B (Section 4.7.2).

Policies. CANDID defines any modification of various CAN components as a *policy*. A policy P consists of five components: $\{P_c, P_\delta, P_\alpha, P_\beta, P_t\}$. P_c is the context maintained by the policy P, and belongs to the context space C. P_δ is the context update function which is used to update the context after a message is newly transmitted on the bus. It is defined as $P_\delta : \mathcal{M} \times \mathcal{C} \rightarrow \mathcal{C}$, taking the transmitted message in the message space M and the current context P_c as input, and outputs an updated context value. P_α is the policy encoder function that applies appropriate modifications on a message. It is defined as $P_\alpha : \mathcal{M} \times \mathcal{C} \rightarrow \mathcal{M}$, taking an unmodified message and the current context as the input, and outputs a modified message to be transmitted on the CAN bus. P_β, the policy decoder function, is the inverse of P_α.

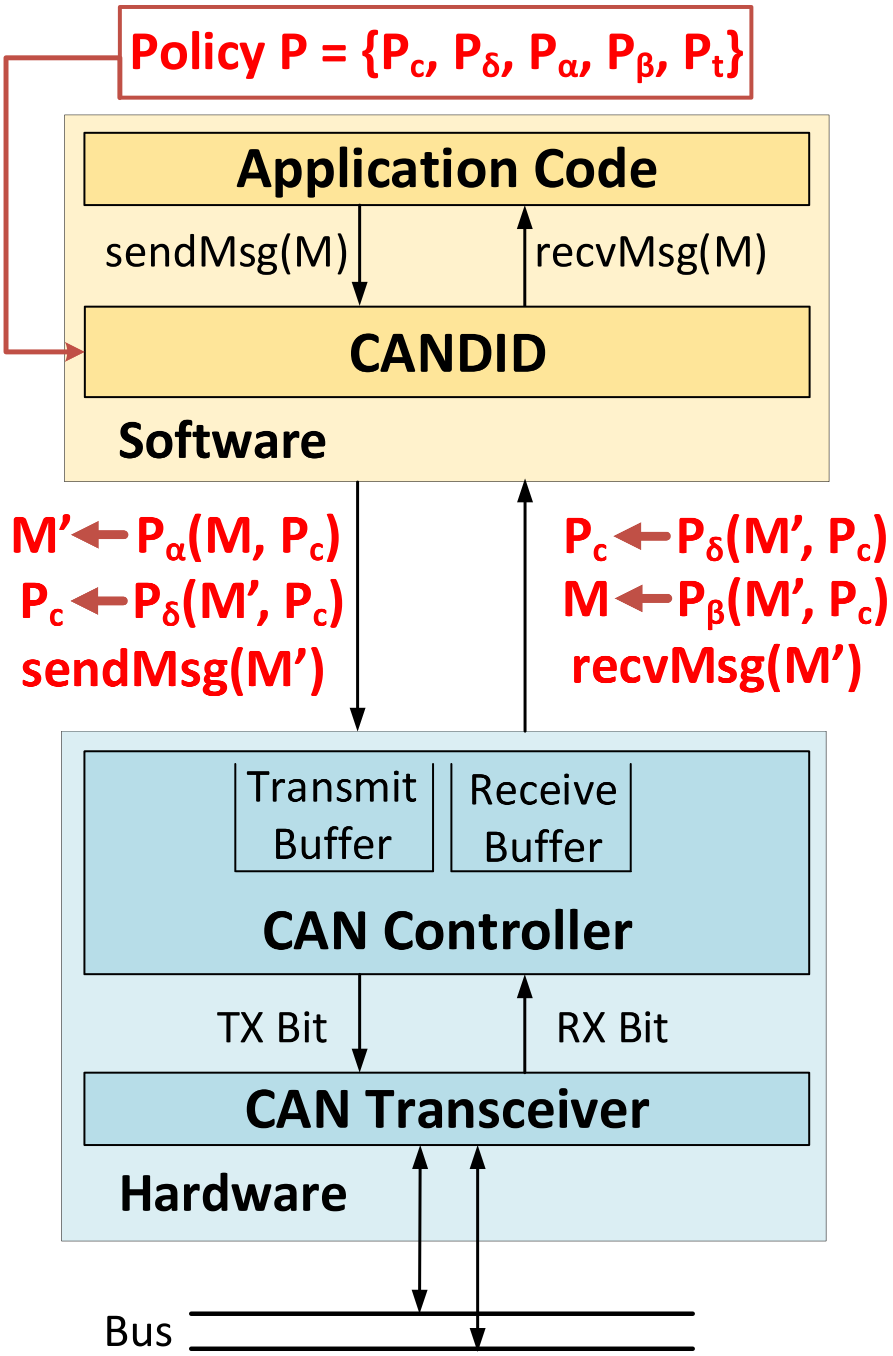


Figure 4.4.: Overview of CANDID architecture with a single policy dialect.

It is defined as $P_\alpha : \mathcal{M} \times \mathcal{C} \to \mathcal{M}$, taking a modified message transmitted on the bus and the current context as the input, and outputs an unmodified message. Finally, P_t is the execution time of policy P, counting the time taken by both context update and policy encoder functions.

Dialects. CANDID can apply multiple policies in a CAN bus environment. A set of policies applied on a system is called a *dialect*. A dialect D is represented as $\{P^{(1)}, P^{(2)}, ..., P^{(n)}\}$ where $P^{(i)}$ are different policies to be applied in order, and n is the number of policies in the dialect. When a dialect D is applied in a CAN bus deployment, all ECUs on the bus must be programmed with CANDID and configured with the same dialect D. Further, CANDID can apply different dialects to different deployments of the same CAN bus environment. For example, two cars of the same make, model, and year can apply different CAN dialects. This makes reverse-engineering time intensive and provides a diversification defense to the CAN environment. Finally, the execution time for a dialect D can be represented as $\sum_{i=1}^{n} P_t^{(i)}$. Ensuring the execution time of a dialect is bounded is very important as CAN environments are resource constrained and cannot apply computationally intensive dialects/policies. An example is shown in Figure 4.4, where a dialect consisting of a single policy is applied on the CAN bus.

4.4 CANDID Architecture

A CAN bus with CANDID consists of CANDID-agents added on each existing ECU and a separate CANDID-controller ECU. The agents on each ECU are responsible for applying a dialect to the CANDID deployment, while the controller is responsible for facilitating the dialect application, and verification of policy compliance.

Agents. The agents are software-only and do not require any modifications in current ECU hardware. They leverage the functionality described in the CAN specification and available in off-the-shelf CAN controllers. The agents are available to manufacturers as a CANDID-enabled CAN library to integrate during the firmware compilation stage. Figure 4.4 shows the architecture for an ECU, where CANDID-agents provides a transparent layer that sits between the hardware interface of the CAN controller and the application code. As a result, the current ECU firmware can be easily recompiled with the CANDID-enabled CAN library and a configured dialect, and no change is required in the application code of the firmware.

Controller. The controller is a separate ECU on the CAN bus. It serves two major functions. First, it facilitates the dialect on the CAN bus. The controller maintains the contexts for different policies for different ECUs. If an ECU needs to reset, the controller then provides the newly reset ECU the up-to-date context for various policies. It communicates this information using asymmetric encryption and public/private key pairs maintained per-ECU, with the public key stored with the controller, and the private key with the ECU. We note that the controller does not serve any role during regular dialect application, and is not a performance bottleneck. The second function the controller serves is to verify the compliance of policies in the dialect. With maintained policy contexts, the controller is able to verify if certain policies are being followed. The controller can be utilized as an anomaly detection system for policies whose violation signals anomalies. We note that not all policies need to be verified, for example a randomization policy would not need verification.

Deployment. CANDID is available to manufacturers as a CANDID-enabled CAN library that can be used as a drop-in replacement for the CAN library used during firmware compilation. The dialect is input as a configuration to the library, and the resultant firmware binaries contain the CANDID agents configured with the appropriate dialect. The CANDID controller is also available with its source code, and can be compiled with the dialect as input configuration. We note that while CANDID is provided as a source-code only solution which can be easily used by manufacturers, it can also be deployed in situations where only the firmware binaries are available. CANDID requires only a few interrupt service routines and the send and receive functions for the CAN bus to be rewritten, and any firmware rewriting tool can be utilized for this purpose.

Threat Model. CAN bus environments can have attackers with various capabilities. An attacker may remotely or physically compromise existing ECUs on the CAN bus, or may physically attach their own ECU to the exposed OBD-II port on the CAN bus. In a compromised existing ECU, the attacker has to follow the CAN protocol, but has the capability to send arbitrary messages, or read all messages on the bus. With their own device, the attacker may even violate the CAN protocol. Under this attack model, we assume that the CANDID-controller is not compromised. This is relatively easy to ensure as the CANDID-controller's attack surface is limited to the CAN bus, it can be physically secured or made hard to access, and the firmware update process can be cryptographically secured. We do not place any assumptions on compromised existing ECUs. In those cases, the attacker has full access to all the maintained policy contexts, and stored cryptographic keys, including the one used for communication with the CANDID-controller. With this basic threat model, in Section 4.7.2 and Section 4.7.2, we look at specialized dialects that can be deployed for specific attack models.

4.5 Design of CANDID

We look at how CANDID implements dialects without requiring specialized hardware. Using current off-the-shelf CAN controllers, CANDID must control the ID, Data, and IFS fields to implement context-dependent policies.

4.5.1 ECU State Tracking

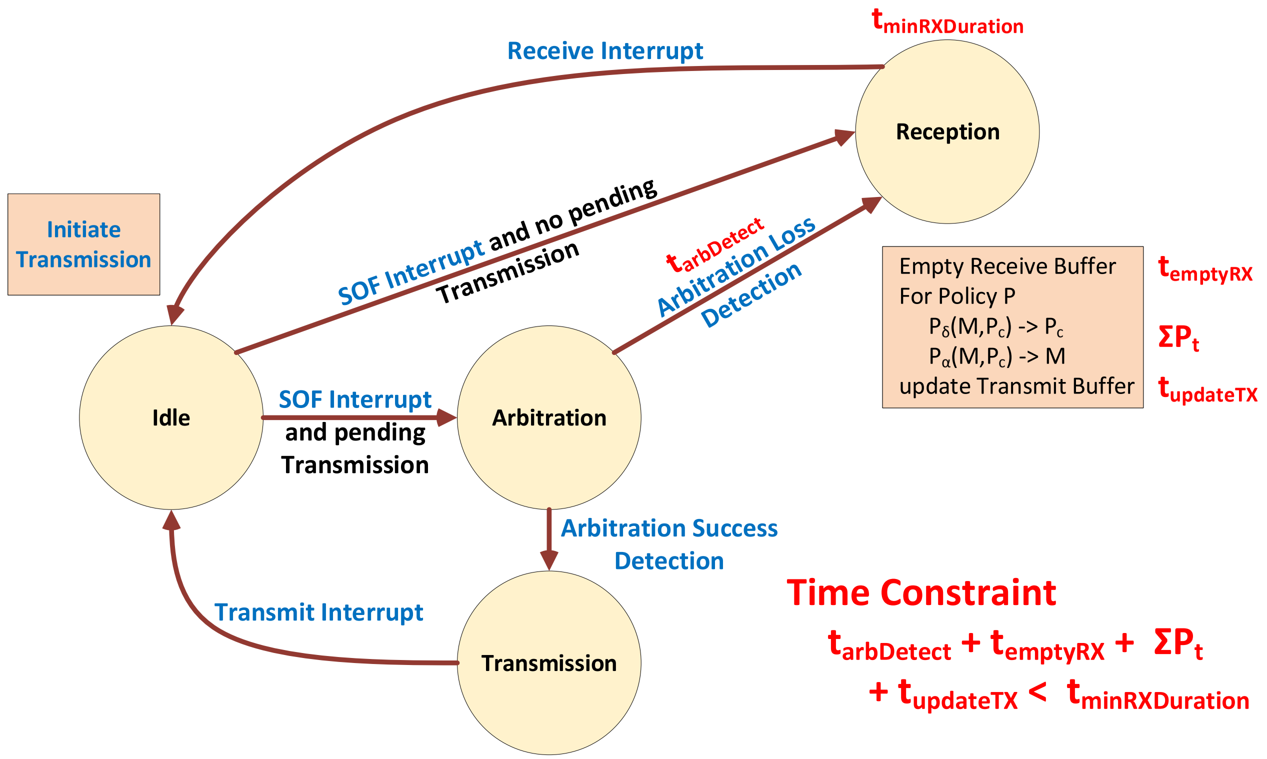


Figure 4.5.: CANDID enables dialect application via ECU state tracking.

Application of a dialect in CANDID faces some challenges. Firstly, for IFS to be controlled, CANDID must accurately track the completion of message communication on the bus, and then programmatically initiate a transmission to achieve a desired value of IFS. Both these actions must be carried out with high accuracy. Secondly, upon reception of a message, CANDID must update the contexts of various policies and reapply policies with updated contexts on any buffered message. This requires CANDID to accurately track when a message is being received, and perform the time-sensitive dialect application. These two reasons, (1) controlling IFS and (2) reapplying dialect, requires CANDID to perform careful tracking of the state of an ECU.

ECU States. An ECU can be in one of the four states– *transmission*, *reception*, *arbitration* and *idle*. The ECU remains in the idle state until there is a message to be transmitted/received. From the idle state, the ECU moves to the reception state when another ECU starts transmitting a message on the bus and the ECU has no transmission buffered in the CAN controller. When the ECU has got a message to transmit, it moves from the idle state to the arbitration state. If the ECU wins the arbitration, it moves to the transmission state and starts transmitting its message; otherwise, if it loses the arbitration, it moves to the reception state as it receives the message on the bus. Upon the completion of the message communication, the ECU moves from the transmission or reception state to the idle state. Figure 4.5 shows the state machine and its various transitions.

To detect the state machine transitions quickly, and also cause them accurately when needed, CANDID relies on five features from the standard CAN hardware (controller and transceiver). These features are (1) SOF interrupt, (2) Receive interrupt, (3) Transmit interrupt, (4) Initiate transmission, and (5) Arbitration success detection. These features do not require any specialized hardware, and are available from existing CAN controllers. Figure 4.5 shows how ECU state transitions are detected with these five features. We now discuss them one by one.

SOF (Start-of-frame) Interrupt. To track the transition of an ECU from idle to reception/arbitration states, CANDID needs to detect when a message has initiated transmission. There are several ways this can be achieved. One way is to utilize the RX output pin from the CAN transceiver. As a new message is communicated on CAN, the RX pin will feature a dominant SOF bit as part of the CAN message, coming after a sequence of recessive bits as the bus was idle beforehand. By attaching an interrupt on the value change of RX pin, CANDID can detect the SOF bit. Another approach is to utilize a feature from TTCAN (Time Triggered CAN), an extension of regular CAN, that stores the time when a SOF bit is raised in a register. By checking modifications in the value of the register, CANDID can detect message communication on the bus. A third way is to utilize the SOF signal that a few CAN controllers [61] provide, specifically to support TTCAN implementations. Any of the three methods can be utilized by CANDID.

Receive Interrupt. An interrupt is provided by CAN controllers on successful reception of a message. This is needed to track the transition from reception state to the idle state. There are two methods to detect receive interrupts. First, CAN controllers provide receive interrupts through the general interrupt pin, and also potentially through a dedicated pin. Second, receive interrupts can also be detected through the RX output pin of the transceiver. Each message terminates with the EOF (End-of-frame) sequence with 7 recessive bits, and such a sequence cannot happen in a non-idle bus otherwise. Both methods for receive interrupt are quick and accurate, allowing fine and timely control of IFS field.

Transmit Interrupt. Similar to receive interrupt, this interrupt is raised on the successful transmission of a message. This interrupt signals the transition from the transmission to the idle state. As like receive interrupts, there are two methods to detect transmit interrupts. One way is to use one of the CAN controller interrupt pins, either general or dedicated. Another is to utilize CAN transceiver's RX pin to detect the EOF sequence. Again, both methods are quick and accurate, and can be utilized by CANDID to control IFS.

Transmission Initiation. For accurate control of IFS field, one must not only detect the termination of past communication, but also be able to initiate a new one. CAN controllers provide the capability to initiate transmission, either through a SPI (Serial Peripheral Interface) call, or through dedicated pins on the CAN controller. Both methods are able to control the IFS length programmatically.

Arbitration Success Detection. If the ECU loses arbitration while transmitting a message, it will reattempt transmission afterwards. However, as a new message has been communicated on the bus (the one which won the arbitration), the context of various policies must be updated, and the policies be reapplied. Hence, it is essential to differentiate success and loss during arbitration. This can be achieved by reading the CAN transceiver's RX pins to identify the ID which won the arbitration, and deduce if the ECU won or lost. Another approach that may be used is to utilize specific registers from the CAN controller that store success and loss of arbitration.

Time Constraints. CAN bus is a resource constrained environment which does not afford much computational power. The computational limitations place a hard limit on the execution time of the dialects during a particular state transition. When an attempted transmission has lost arbitration, and the ECU is in the reception state, CANDID must perform a bunch of actions. First, it must clear the receive buffer (if full) to prevent overflow. CANDID saves the context and the received message to perform the unapply the dialect later. Second, it must update the context to account for the message currently being received. And lastly, if a message is awaiting transmission, it must update the message by applying the dialect with the new updated context. All these actions must be performed in the reception state. The time limit can be formulated as:

$$t_{arbDetect} + t_{emptyRX} + \sum_{i=1}^{n} P_t^{(i)} + t_{updateTX} < t_{minRXDuration} \tag{4.1}$$

, where $t_{arbDetect}$ is the computational time taken to detect arbitration loss, $t_{emptyRX}$ is the time taken to clear the receive buffer, $\sum_{i=1}^{n} P_t^{(i)}$ is the dialect execution time,

$t_{updateTX}$ is the time taken to push the updated message in the transmit buffer, and $t_{minRXDuration}$ is the minimum time the ECU will stay in the reception state.

To check if a dialect is feasible for a certain architecture, this time constraint can be verified offline during the compilation stage. CANDID includes a tool that can be guarantee the feasibility of a dialect for a certain architecture, and can be easily used by the manufacturers.

4.5.2 Interrupt Service Routines

As CANDID is enabled by quick state-tracking, it is necessary that the interrupt service routines are not delayed unnecessarily. To enforce this, CANDID maintains the various interrupts in strict priority levels. Only interrupts of higher priority are allowed to interrupt an existing interrupt. This priority hierarchy is kept as SOF $>$ Receive $=$ Transmit $>$ System/Timer interrupts, i.e. CANDID ensures its own interrupts are given a higher priority over the timer and other system interrupts. Even if the target architecture does not support interrupt masks to disable interrupts, such as the AVR which we implement CANDID upon, we can enforce these priority levels by selectively enabling and disabling interrupts. For example, we enable interrupts during all interrupts except SOF interrupt service routine, and disable system/timer interrupts during receive and transmit interrupt service routines.

Interrupt Safety. Interrupt-safety is also an important consideration owing to the complexity of the interrupt service routines in CANDID, which requires them to have access to the CAN hardware, and the various policies and their contexts. However, as even the non-interrupt code, as well as multiple interrupt service routines share these resources, we must employ several techniques to keep CANDID interrupt-safe.

Interrupt-safe Data Structures. To reduce the number of shared resources in CANDID, we utilize interrupt-safe data structures wherever possible. Specifically, we utilize circular buffers for storing transmit requests and received messages. This allows interrupt-safe appends and removals, which CANDID utilizes to handle the shared resource safely.

Enabling/Disabling Interrupts. Requirement for atomicity is reduced by disabling and enabling interrupts, whenever certain variables such as the ECU state need to be updated. However, as CANDID must not delay any interrupt by a large duration of time, we ensure that interrupts are disabled for only a few clock cycles.

Delayed Critical Section. Even with the above solutions, CANDID still has critical sections across multiple interrupt service routines and non-interrupt code that share resources. For example, both the CAN library's readMsg function called from application code and the SOF interrupt service routine will attempt to clear the receive buffer. Also, one cannot merely delay the interrupt, as CANDID must still service the time-sensitive parts of the interrupt, such as the check for arbitration loss in SOF interrupt service routine. To solve this issue, CANDID marks a boolean to detect critical section execution, and delays the execution of critical sections in the interrupt service routines. CANDID first executes the time sensitive but non-critical section of the interrupt service routine, then finishes execution of the critical section in the non-interrupt code, and finally executes the delayed critical section from the interrupt service routine.

4.6 Implementation

ECU and Testbed Setup. We implement CANDID on ECUs based on Arduino Uno utilizing MCP2515 CAN controller and the MCP2551 CAN transceiver. The ECUs have a 16 MHz processor, 2 KB dynamic memory, and 32 KB program memory. The SPI communication between the Arduino Uno and the CAN controller utilizes a 8 MHz clock. The testbed consists of 6 ECUs, each programmed with CANDID. The ECUs are connected through a high speed CAN bus operating at 500 kbps, which is the standard for high-speed automotive buses.

ECU State Tracking. For the five features, (1) SOF interrupt, (2) Receive interrupt, (3) Transmit interrupt, (4) Initiate transmission, and (5) Arbitration success detection, we utilize features provided by the CAN controller and the CAN transceiver. We utilize the inbuilt SOF signal in the CAN controller. We also utilize the dedicated pin for Receive interrupt in the CAN controller. For the transmit interrupt, we utilize the RX output pin from the CAN transceiver to detect the EOF sequence. Initiation of transmission is carried out through the dedicated pin in the CAN controller. Finally, arbitration success detection is carried out by reading the RX pin to identify the ID which won the arbitration. We note that access to these pins from the controller and transceiver do not increase the attack surface of the bus and other ECUs, as the controller pins are intended features, and the RX pin from the transceiver is a read-only pin. Also, while we have utilized these mechanisms for the five features, CANDID can be implemented via any of the other mechanisms described in Section 4.5.1.

Time Constraints. We calculate the various constants in Equation 4.1 for our setup. The computational time taken to detect arbitration loss ($t_{arbDetect}$), is 16μs. The time taken to read from and clear the receive buffer comes out to be 14μs. The time taken to push an updated message is also 14μs. The minimum time an ECU will stay in the reception state (after arbitration) is at least 38 bit periods. This is calculated as 15 bit CRC checksum, 7 bit EOF sequence, 4 bit control field, 8 bit for minimum 1 byte payload, and 4 bit of ACK and various delimiters. Note that a payload of 0 byte is possible, but not used on any high-speed vehicular CAN buses. Hence, $t_{minRXDuration}$ is equal to 38 bit periods, which is 76μs for our 500kbps CAN bus. The equation then simplifies to

$$\sum_{i=1}^{n} P_t^{(i)} < 32\mu s \tag{4.2}$$

with $\sum_{i=1}^{n} P_t^{(i)}$ as the dialect execution time.

Evaluation Setup. For evaluation, we emulate the traffic from real vehicles on our testbed. We utilize three CAN buses, two from 2013 Chevrolet Cruze, and one from a 2011 Chevrolet Impala for our evaluation. The first bus Bus1, is a bus on the Cruze, also operating at 500 kbps. It consists of 88 messages of varying periodicities being transmitted by six ECUs, with a 61% bus load. The second bus Bus2, also a bus on the Cruze, is also operating at 500 kbps. It consists of 27 messages of varying periodicities being transmitted by three ECUs, with a 34% bus load. The third bus Bus3, is a bus on the Impala operating at 500 kbps. It consists of 50 messages of varying periodicities being transmitted by four ECUs, with a 35% bus load. Note that while we utilize Chevrolet vehicles for our evaluation setup, CANDID is generic and does not use any characteristic unique to Chevrolet CAN buses.

4.7 Evaluation

4.7.1 General Evaluation

Identity Dialect. We first perform a general evaluation for CANDID with an identity dialect that does not modify any components, but does perform state tracking.

ECU. We find that the resource usage of our ECUs stays small after deploying them with CANDID. The dynamic memory usage (global variables) increases 0.39 KB (from 0.38 KB to 0.77 KB), and program memory usage increases 3.1 KB (from 4.6 KB to 7.7 KB) on implementing CANDID. As a result, CANDID easily fits the resource requirements of such low-end devices.

Bus. The identity dialect does not produce any extra traffic on the bus. In addition to that, we compare the jitter in our testbeds to the jitter in the real vehicle. We find in Figure 4.7 that the jitter in our testbed is a fraction of the jitter that exists in the real vehicle. This is because real vehicle ECUs have other sources of jitter besides the queuing and arbitration delays.

IFS Control. A novel contribution of CANDID is the ability to control IFS between CAN messages. In this section, we check the accuracy of IFS control in CANDID. Our evaluation in setting IFS values in our testbed shows that CANDID

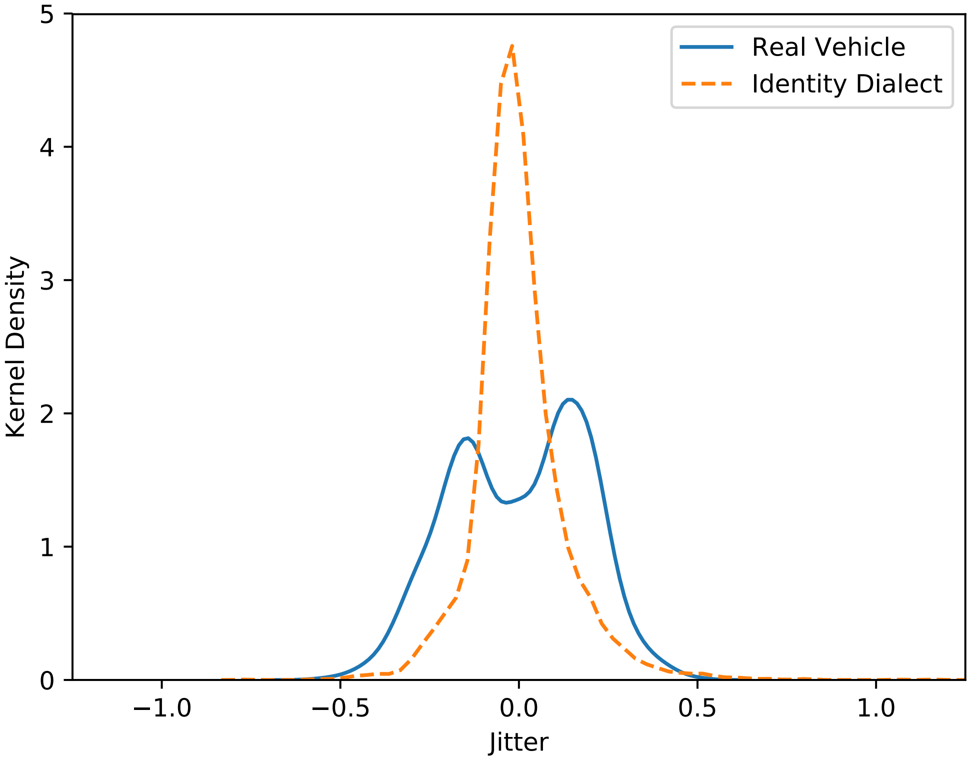


Figure 4.6.: Jitter Distribution for an ID (0x0C1) on Chevrolet Cruze (Bus1).

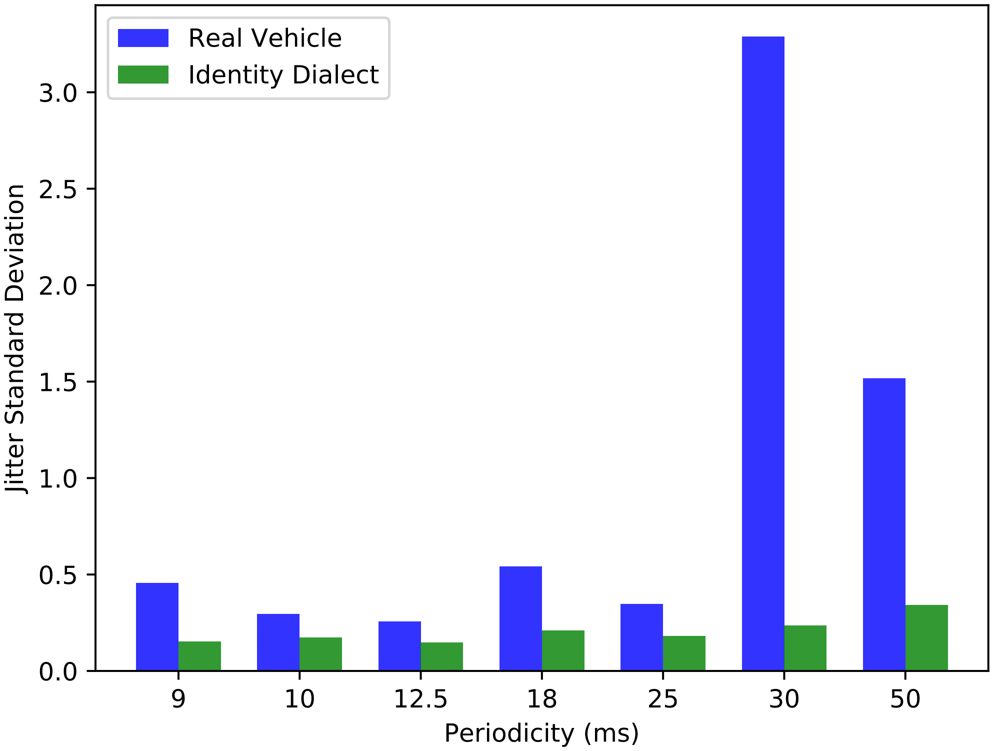


Figure 4.7.: Jitter Standard Deviation by Periodicity on Impala (Bus3).

is 100% accurate. This may look counter-intuitive, as the accuracy of IFS control should be challenged by the clock-skew between different ECUs. As the clocks of two ECUs are slightly skewed, their estimated length of IFS differs if the bus has been empty for a long time. However, the clocks of ECUs skew very slowly [21], of the order of 100 parts per million, i.e. 1 bit period per 10,000 bit periods. This means that clock-skew will have an impact only for a 10ms idle period, and our real vehicular buses and hence our testbeds are never idle for that long. Hence, we are able to control and set IFS to arbitrary values with 100% accuracy.

4.7.2 Case Studies

To evaluate the attack-resilience and implementation overhead of dialects generated by CANDID, we present the following two case studies – (1) attacker controlling a compromised ECU on the bus, (2) attacker with direct access (e.g., through the OBD-II port) to the bus.

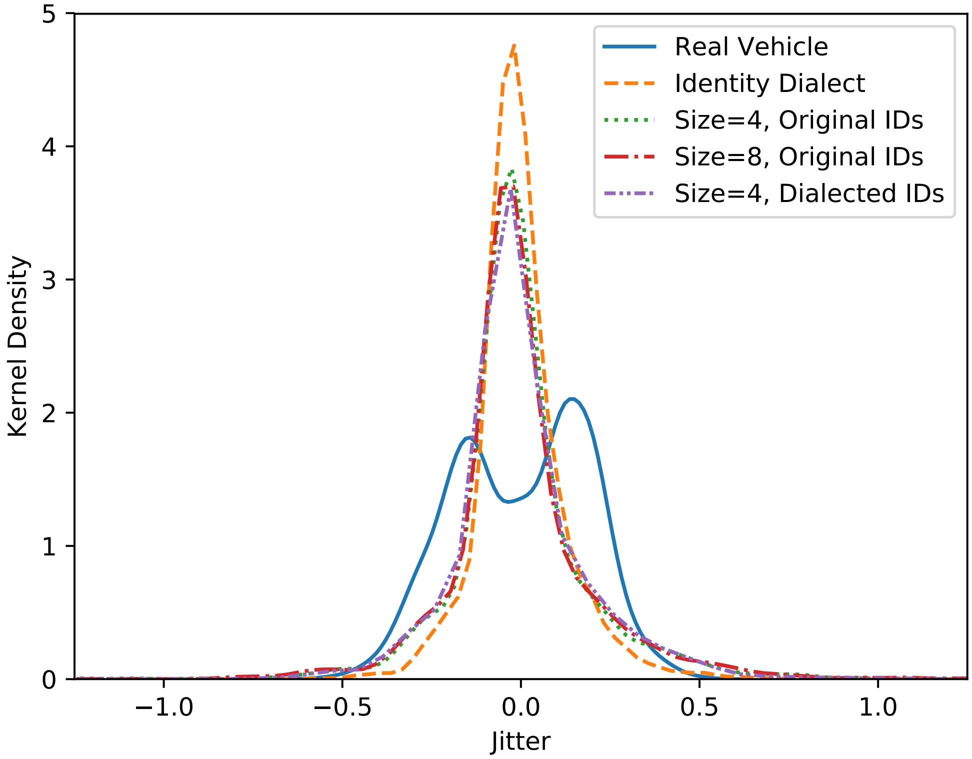


Figure 4.8.: Jitter Distribution for an ID (0x0C1) on Chevrolet Cruze (Bus1) for Case Study: Compromised Existing ECU.

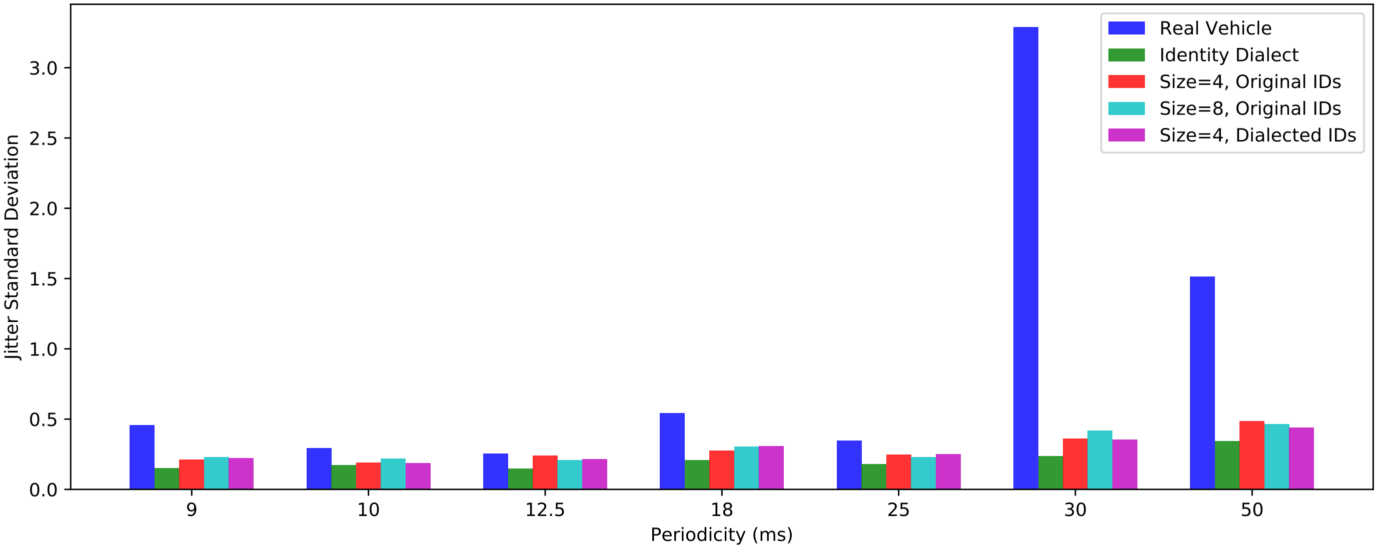


Figure 4.9.: Jitter Standard Deviation by Periodicity on Impala (Bus3) for Case Study: Compromised Existing ECU.

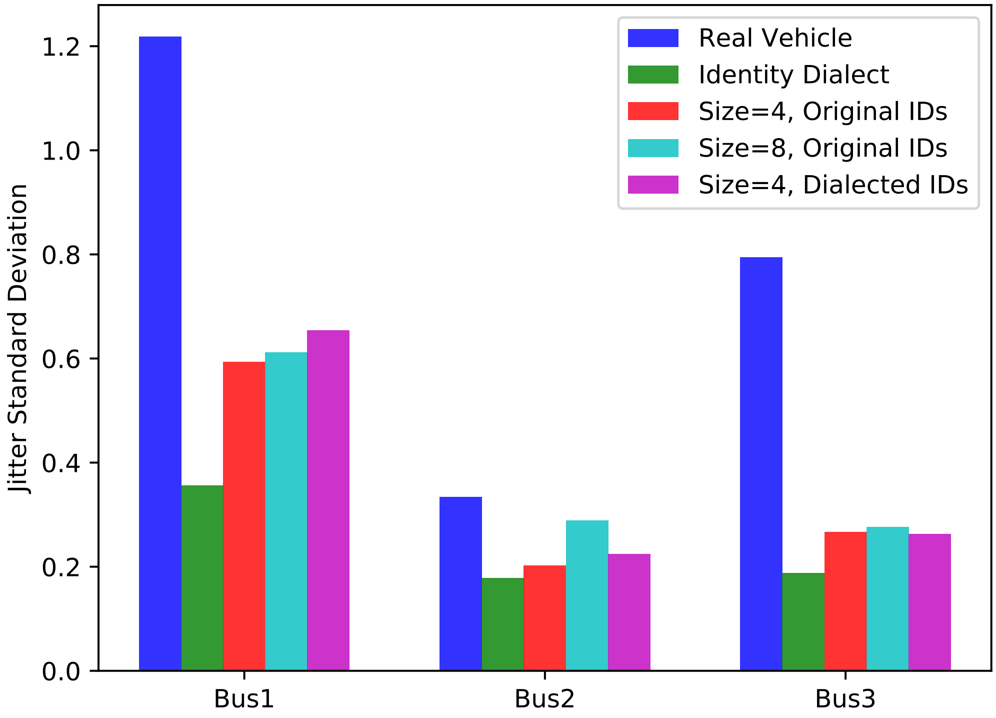


Figure 4.10.: Jitter Standard Deviation on different buses for Case Study: Compromised Existing ECU.

Compromised Existing ECU

An attacker with a compromised ECU on the CAN bus has been repeatedly shown to be practical [6–8, 10, 13, 14] and able to inject messages and control [8, 9, 13–15] various vehicular functionality. The attacker can also suppress another ECU from transmitting, utilizing the bus-off attack [53]. We note that the message injection and bus-off attacks are successful because of static components of the CAN bus. While message injection is effective because reverse engineering is easy and scalable from only one vehicle, the bus-off utilizes the constant IFS (=3) between consecutive messages to induce errors.

To address such an attacker, we look towards randomization policies. While Payload (format) and IFS are easily randomized by CANDID, the ID field faces restrictions. As IDs are used for arbitration, the randomization must preserve the priority order, and can be easily reverse engineered. To address this issue, we show how IFS can be utilized for arbitration.

Priority Scheduling without ID. CANDID allows us to utilize IFS for priority scheduling on a CAN bus. We make two observations that enable this, (1) if two ECUs have messages queued when the bus is busy, the message with the lower IFS will be able to transmit, and (2) while there are tens of IDs present on a CAN bus, there are generally only around 5 deadline/priority levels [86, 87]. Also, since IFS control is very accurate, the IFS-based priority scheduling is also very reliable. This makes it possible to free ID from the role of arbitration.

Dialect. We deploy a randomization-based dialect on our CAN bus, whose policies are described below.

IFS-based Priority Scheduling. IFS values are assigned through a policy encoder function f that assigns IFS to 5 priority levels. The policy is represented as $\{\varnothing, \varnothing, f, \varnothing, 0\}$, where $f(M).ifs \in \{0, 3, 6, 9, 12\}$.

ID Randomization. For each ID $a \in ID_{space}$ used on the CAN bus, we assign a set of possible values $ID_a \subset ID_{space}$. Note that for two distinct ID a and b, ID_a and ID_b should be disjoint. Further, ID_a and ID_b do not need to preserve relative ordering for priority, making ID-only reverse engineering impossible. The policy is represented as $\{\varnothing, \varnothing, f, f^{-1}, 0\}$, where $f(M).id \in ID_{M.id}$.

Payload Randomization. The policy is represented as $\{\varnothing, \varnothing, f, f^{-1}, 0\}$, where $f(M).payload$ is a static permutation of $M.payload$.

IFS Randomization. IFS values are also randomized. The policy is represented as $\{\varnothing, \varnothing, f, \varnothing, 0\}$, where $f(M).ifs = M.ifs + r$, with $r \in \{0, 1, 2\}$.

Time Constraint. As the dialect is completely context-independent, it does not need to update contexts and reapply policies. Hence it satisfies the time constraint.

Attack Resilience. By deploying different dialects in different vehicles with randomized Payload (format) and ID values, the scalability of the attack is defeated. By randomizing IDs (through IFS-based priority scheduling) without maintaining priority ordering, reverse-engineering within a single vehicle is also made difficult. Finally, by randomizing IFS values, victim suppression via bus-off is prevented.

Performance. CANDID produces minimal overhead (over the identity dialect) for the ECU, primarily due to memory and computational requirements for randomizing. Further, the bus load is also not increased.

Message Delays. The only metric that is affected is the delays in message transmissions. These occur because CANDID controls IFS by delaying the message to set its value. We plot the standard deviation in inter-message arrival rates of the periodic messages (which are all CAN messages) on all buses. Our results show that the increase in jitter by our dialect is a fraction of the jitter that exists in real vehicles.

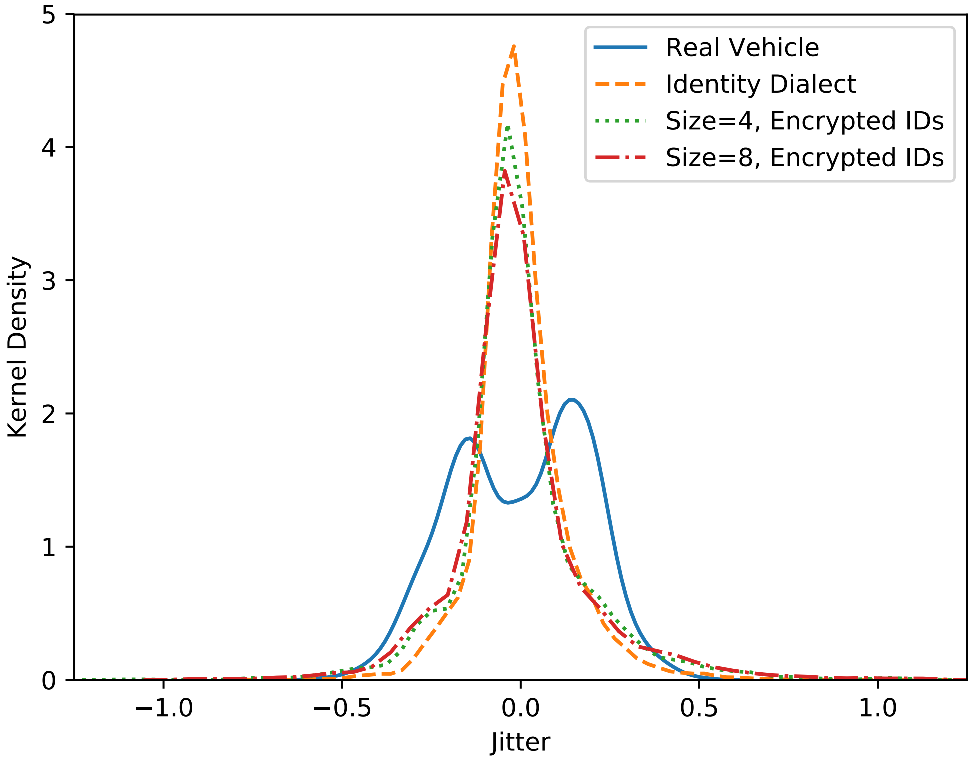


Figure 4.11.: Jitter Distribution for an ID (0x0C1) on Chevrolet Cruze (Bus1) for Case Study: External ECU.

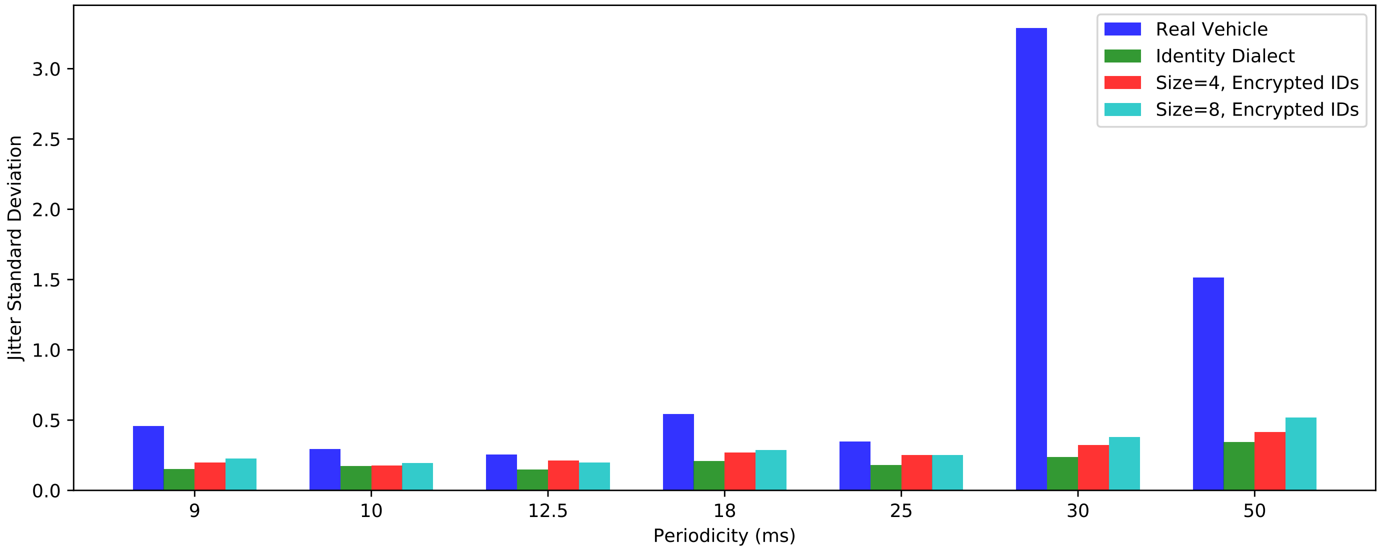


Figure 4.12.: Jitter Standard Deviation by Periodicity on Impala (Bus3) for Case Study: External ECU.

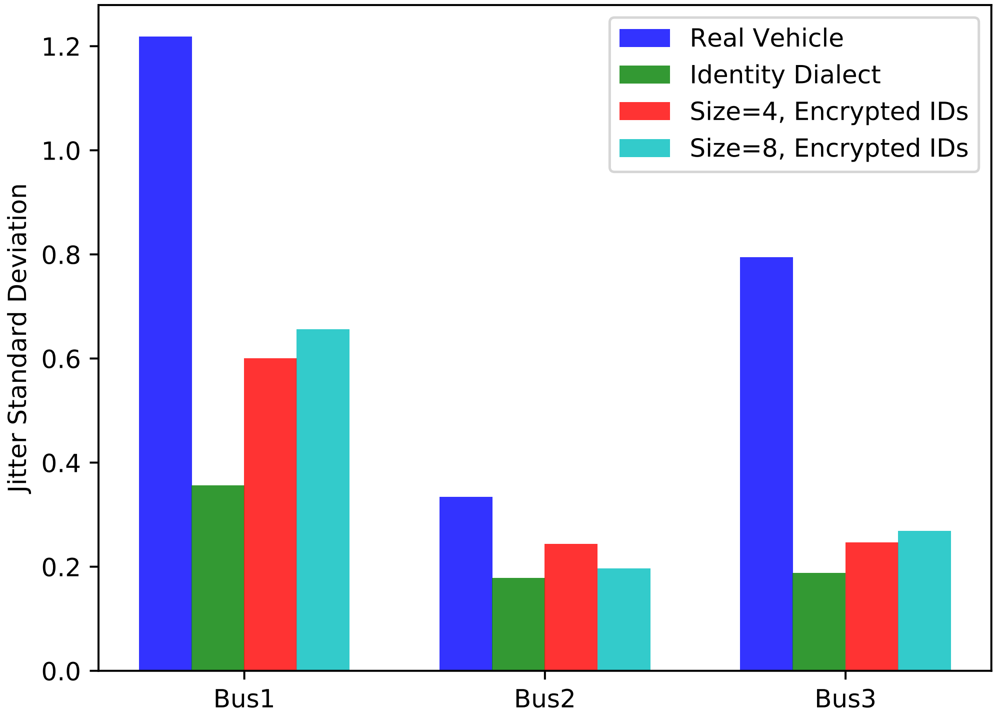


Figure 4.13.: Jitter Standard Deviation on different buses for Case Study: External ECU.

External ECU

An attacker with physical access to the vehicle can carry out the “evil maid” attack, by attaching an external ECU to the bus through the On-Board Diagnostics (OBD)-II port. The attacker thus skips the difficulty of finding an exploit, has remote access, and control of a sophisticated ECU on the bus. However, this attacker has not compromised an existing ECU on the bus, and the ECUs can maintain shared secrets for cryptographic solutions.

Synchronous Stream Cipher. The primary difficulty of implementing cryptographic solutions [16, 17, 74, 78, 88–91] on CAN has been the overhead, which the resource-constrained ECU cannot meet. However, this is primarily because cryptographic solutions have traditionally been applied per-message, whose fixed cost is then repeated for each message. Instead, with CANDID, it is possible to have the ECUs maintain a synchronous stream cipher. This allows the CAN bus to have efficient symmetric encryption by treating the messages on the bus as the plaintext.

Dialect. We deploy a encryption-based dialect on our CAN bus. The synchronous stream cipher has the shared keystream K, with the policy-context maintaining the offset in the keystream. Further, we define a context-update function $f(c, m) = c+80$, which increases the offset 80 bit for every message transmission on the bus. We choose 80 bit (10 bytes) as it is enough to encrypt ID (11 bit), encrypt Payload (<64 bit), and set IFS (0 to 3). The context for different policies are initialized as follows, 0 for ID, 11 for Payload, and 75 for IFS. The policies are described below.

IFS-based Priority Scheduling. We utilize the same policy from Section 4.7.2. This frees up the ID for encryption.

ID Encryption. The policy is represented as $\{c, f, g, g, P_t^{(2)}\}$, where $g(M).id = M.id \oplus K[c : c+11]$ for keystream K, context-update function f, and the decryption function being the same as the encryption function.

Payload Encryption. The policy is represented as $\{c, f, g, g, P_t^{(3)}\}$, where $g(M).payload = M.payload \oplus K[c : c+64]$ for keystream K, context-update function f, and the decryption function being the same as the encryption function.

IFS Anomaly Detection. The legitimate ECU sets IFS from the keystream, but the attacker will set IFS to a random value. This allows us to perform anomaly detection, as over a large number of attacker's messages, the probability of detection increases. The policy is represented as $\{c, f, g, g^{-1}, P_t^{(4)}\}$, where $g(M).ifs = M.ifs + K[c : c+5] \pmod 3$ for keystream K, and context-update function f.

For the evaluation, we consider two stream-ciphers. (1) ChaCha20, which is adopted in Chrome for TLS [92] and (2) Speck (256 bit key, ECB mode).

Time Constraint. The dialect execution time $\sum_{i=1}^{n} P_t^{(i)}$ in this case is the time taken to read, store, and xor 10 bytes, along with an update of the policy-context.

This comes out to roughly $4\mu s$, which is comfortably below the upper limit set in Equation 4.2.

Attack Resilience. Through encryption of both ID and Payload, we have made both reverse-engineering and message injection impossible. Further, any brute force attempt to find collision will also be quickly detected through the use of anomaly detection via IFS.

Performance. For the ECU overhead, the program memory usage increases by 0.45 KB, and the dynamic memory usage increases by 0.31 KB for ChaCha20, and by 0.42 KB for Speck. The computational overhead is increased as the keystream must be generated in idle time, and comes out to around 150μs for ChaCha20 and 100μs for Speck2. This corresponds to a processor usage of 45%, 27%, and 28% for ChaCha20, and 30%, 18%, and 18% for Speck2 on our three buses. Also, Just as before, CANDID does not cause any increase in the busload.

Message Delays. Similar to Section 4.7.2, messages transmissions may be delayed because of setting IFS values. For this case study too, we plot the standard deviation in inter-message arrival rates of the periodic messages on all buses. Our results again show that the overhead introduced by our dialect is a fraction of the jitter that exists in real vehicles.

5 RELATED WORK

5.1 Timeliner

Timeline reconstruction is of interest to both cyber and traditional crime investigations. This interest is reflected in the wide variety of work done for creating timelines [93–96], making better tools for editing and visualization [97, 98], and correlating sources together to infer semantics in a timeline [99–101]. However, all these methods are dependent on various logs and database files that are formatted independently by applications making their timeline recovery highly application-specific. Further, these logs and database files are limited to a small set of events that are logged. Even widely used commercial tools Oxygen [102] and Cellebrite [103] are application-specific and are limited to these small sets of events. Further, reliance on system level logging is untrustworthy as major phone manufacturers turn off Android features that reveal forensic information [4]. Timeliner, on the other hand, is application-generic and can reliably reconstruct a wider variety of actions into the timeline *from only a single image of volatile memory.*

Timeliner is more related to RetroScope [29], a memory forensics technique capable of reconstructing historical and temporally ordered GUI screens. However, Timeliner differs from RetroScope in two aspects. First, while RetroScope is limited to reconstruction for a single running (at the time of memory snapshot) app, Timeliner works *across* all apps and can construct a device-wide timeline of app activities (including terminated apps). Second, RetroScope reconstructs screens, which are renderings of GUI content, while Timeliner reconstructs *Activities*, which are abstractions of user actions/events. As such Timeliner and RetroScope perfectly complement each other, with Timeliner reconstructing the skeleton of a crime story involving multiple apps and RetroScope re-rendering the activity details within each app.

Memory forensics has been applied extensively to the Android platform. Mostly these applications have focused on recovering raw data structures: app-specific login credentials, JVM control structures, raw Java objects, text messages, buffered media content, and a variety of application-specific data [28, 104–109]. Recovery of the raw data structures is performed via value-based [26, 110–113] (relying on constants and expected values) or structure-based [114–117] (relying on pointer constraints) scanning. In particular, SigGraph [118] recovers data structure instances using probabilistic analysis on the whole memory image. On the other hand, data structure recovery is only the first, preparatory step in Timeliner's timeline recovery.

Various memory introspection and memory analysis techniques have been used to determine malware and virus activity by observing kernel data structures [119, 120] or by identifying data structure signatures for polymorphic viruses [121]. However, while these techniques either rely on active introspection or recover only live kernel and virus data structures, Timeliner recovers and orders past app activities, including activities with no references from live data structures, using only a single memory image.

A number of recent works have gone beyond merely recovering raw data structures towards full-utilization of their content. DSCRETE [122] recovers a single data structure instance and utilizes binary analysis and code reuse to transform it into a human-understandable form. DEC0DE [27] also operates on a single data structure at a time, recovering call log entries using a finite state machine. Tools such as HOWARD [123], REWARDS [124], and TIE [125], infer data structure definitions in binary programs. DIMSUM [126] utilizes probabilistic inference to identify data structures without page mapping information. VCR [28] recovers media content using vendor generic signatures, and GUITAR [127] pieces back together various data structures to retrieve an application's GUI. As a new, complementary addition to the above tool set, Timeliner leverages spatial memory layout information to infer temporal ordering of user Activities.

5.2 Controller Area Network

Attacks. In many existing attacks [6, 8, 9, 13], the attacker injects spurious CAN messages to control the vehicle without victim suppression, which makes the attack easily detectable. The conventional approach to victim suppression is to exploit vehicle's Unified Diagnostic Services [6, 8, 10, 13] and is hence easily preventable by restricting the use of diagnostics only in protected environments (e.g., repair shops). In contrast, DUET does not require vehicle diagnostics and is more flexible. Another method to suppress a victim is to utilize the bus error-handling mechanism [53]. However, not only is the original bus-off attack detectable by VIDS [19], it is also slow in suppressing the victim and hence ineffective in stopping the victim's transmission persistently. DUET employs the passive error regeneration tactic in Stage 2 to rapidly and persistently bus-off the victim, all while evading VIDS.

While ECU's voltage fingerprinting can be utilized to detect all known ECU masquerade attacks, existing voltage fingerprint-based impersonation methods (described in [18–20]) against such VIDS involve a *lone* attacker changing its own fingerprint indirectly (heating/cooling ECU). Hence, they are unable to significantly alter the attacker's fingerprint or target the victim's fingerprint. As a result, VIDS are able to detect such impersonation attempts. In fact, DUET is the first work to successfully counter VIDS, and does so through a duo of attacker and accomplice ECUs manipulating the victim's voltage fingerprint and then impersonating the victim. Table 3.4 summarizes the strategy and detectability of various attacks, highlighting the novelty and stealth of DUET.

Defense. Various cryptographic solutions [16, 17] have been proposed to secure CAN by encrypting and authenticating message payloads. Unfortunately, cryptographic solutions remain impractical as vehicular CAN employs resource-constrained ECUs and remains bandwidth-constrained. In contrast, CANDID includes context-dependent modification of CAN components which enables efficient symmetric cryptography with a shared stream cipher. More practical and deployable CAN defenses favor signature and fingerprint-based IDS, such as MIDS [11, 22, 23], CIDS [21] and VIDS [18–20, 48]). While MIDS and CIDS have been shown vulnerable to impersonation attacks [21,49], VIDS are still considered the state-of-the-art defenses. However, as shown, VIDS are not effective against DUET. RAID complements existing VIDS and provides an orthogonal, lightweight, and effective defense that not only *prevents* (instead of just detects) DUET, but also – in conjunction with VIDS – detects or prevents all aforementioned attacks. Unlike RAID, CANDID through its ECU state tracking and control of IFS enables developers to create new policies for intrusion detection.

6 CONCLUSION

Cyber-physical systems are used to handle the interactions of the physical-world and the cyber-world. Actuators are used to influence the physical-world from the cyber-world, with sensors working in the reverse direction. Traditionally, the temporal component of the physical-world has been restricted to the analysis of deadlines of the actuator messages and the responsivity of the control-loop algorithm reacting to the sensor inputs. However, the temporal component on its own interacts with the cyber-world, and has not been studied.

In this dissertation, we investigate the relationship of the temporal component of the physical-world with the cyber-world. We first study how temporal order of actions leaves an impact in the cyber-world, with application in forensics. We next look at Controller Area Network (CAN), a broadcast communication network, where the temporal control of messages enables new attack and defense capabilities.

In particular, Timeliner presents an Android memory forensics technique that reconstructs a timeline of past user-actions from a single static memory image collected post-crime. By identifying the key set of data structures left in a memory image by user-actions, Timeliner is able to recover the list of past user-actions. Then, Timeliner infers the temporal order of user-actions from the spatial order of their memory allocations. Our results show that Timeliner is highly accurate in reconstructing up to an hour of past activities

DUET presents an attack capability enabled by the temporal components of messages by two adversaries, an attacker and an accomplice. DUET is able to stealthily manipulate the physical-world voltage fingerprint learnt by a voltage-based IDS, persistently suppress the victim, and then impersonate the manipulated fingerprint. RAID presents a lightweight and efficient defense to DUET, which can complement existing IDS defenses.

CANDID presents defensive capabilities enabled by control of various CAN message components, the most important of which is temporal. Through its ECU state tracking mechanism, CANDID enables new applications such as using inter-frame space for arbitration, freeing up CAN identifiers for other applications, or enabling lightweight stream-cipher based symmetric encryption in the resource constrained CAN bus environment. Our results show that CANDID is able to provide these capabilities at minimal performance overhead.

In conclusion, this dissertation casts light on the complex relationship between the temporal component of the physical-world and the cyber-world. As a result, Timeliner, DUET, and CANDID enable new capabilities for forensics, attack, and defensive applications.

REFERENCES

[1] Commonwealth v. Phifer. SJC-11242, (2012).

[2] Nissen v. Pierce County (Majority). Washingotn S. Ct. 90875-3, (2015).

[3] Hamilton v. State. Oklahoma S. Ct. F-2015-529, (2016).

[4] Devices without UsageStats API. `http://stackoverflow.com/questions/32135903/devices-without-apps-using-usage-data-or-android-settings-usage-access-setting`.

[5] Ivan Studnia, Vincent Nicomette, Eric Alata, Yves Deswarte, Mohamed Kaâniche, and Youssef Laarouchi. Survey on security threats and protection mechanisms in embedded automotive networks. In *Dependable Systems and Networks Workshop (DSN-W), 2013 43rd Annual IEEE/IFIP Conference on*, pages 1–12. IEEE, 2013.

[6] C. Miller and C. Valasek. Adventures in automotive networks and control units. *Def Con*, 21:260–264, 2013.

[7] C. Miller and C. Valasek. A survey of remote automotive attack surfaces. *Black Hat USA*, 2014:94, 2014.

[8] C. Miller and C. Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015:91, 2015.

[9] K. Koscher, A. Czeskis, F. Roesner, et al. Experimental security analysis of a modern automobile. In *IEEE Symposium on Security and Privacy (S&P)*, pages 447–462, 2010.

[10] S. Checkoway, D. Mccoy, B. Kantor, et al. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*, pages 77–92, 2011.

[11] T. Hoppe, S. Kiltz, and J. Dittmann. Security threats to automotive CAN networks–practical examples and selected short-term countermeasures. *Reliability Engineering & System Safety*, 96(1):11–25, 2011.

[12] Yelizaveta Burakova, Bill Hass, Leif Millar, and André Weimerskirch. Truck hacking: An experimental analysis of the sae j1939 standard. In *WOOT*, 2016.

[13] S. Nie, L. Liu, and Y. Du. Free-fall: Hacking Tesla from wireless to CAN bus. *Briefing, Black Hat USA*, 2017.

[14] S. Nie, L. Liu, Y. Du, and W. Zhang. Over-the-air: How we remotely compromised the gateway, BCM, and autopilot ECUs of Tesla cars. *Briefing, Black Hat USA*, 2018.

[15] S. Woo, H. J. Jo, and D. H. Lee. A practical wireless attack on the connected car and security protocol for in-vehicle CAN. *IEEE Transactions on Intelligent Transportation Systems*, 16(2):993–1006, 2015.

[16] B. Groza and P. Murvay. Security solutions for the controller area network: Bringing authentication to in-vehicle networks. *IEEE Vehicular Technology Magazine*, 13(1):40–47, 2018.

[17] Q. Hu and F. Luo. Review of secure communication approaches for in-vehicle network. *International Journal of Automotive Technology*, 19(5):879–894, 2018.

[18] M. Kneib and C. Huth. Scission: Signal characteristic-based sender identification and intrusion detection in automotive networks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 787–800, 2018.

[19] W. Choi, K. Joo, H. J. Jo, et al. VoltageIDS: Low-level communication characteristics for automotive intrusion detection system. *IEEE Transactions on Information Forensics and Security*, 13(8):2114–2129, 2018.

[20] K.-T. Cho and K. G. Shin. Viden: Attacker identification on in-vehicle networks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1109–1123, 2017.

[21] K.-T. Cho and K. G. Shin. Fingerprinting electronic control units for vehicle intrusion detection. In *USENIX Security Symposium*, pages 911–927, 2016.

[22] M. Müter, André Groll, and Felix C Freiling. A structured approach to anomaly detection for in-vehicle networks. In *Sixth International Conference on Information Assurance and Security (IAS)*, pages 92–98, 2010.

[23] H. M. Song, H. R. Kim, and H. K. Kim. Intrusion detection system based on the analysis of time intervals of CAN messages for in-vehicle network. In *International Conference on Information Networking (ICOIN)*, pages 63–68, 2016.

[24] Rohit Bhatia, Brendan Saltaformaggio, Seung Jei Yang, Aisha I Ali-Gombe, Xiangyu Zhang, Dongyan Xu, and Golden G Richard III. Tipped off by your memory allocator: Device-wide user activity sequencing from android memory images. In *Proceedings of Network and Distributed System Security Symposium*, 2018.

[25] Ross M Gardner and Tom Bevel. *Practical crime scene analysis and reconstruction.* CRC Press, 2009.

[26] The Volatility Foundation. `http://www.volatilityfoundation.org/`.

[27] Robert Walls, Brian N Levine, and Erik G Learned-Miller. Forensic triage for mobile phones with DEC0DE. In *Proceedings of USENIX Security Symposium*, 2011.

[28] Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. VCR: App-agnostic recovery of photographic evidence from android device memory images. In *Proceedings of ACM Conference on Computer and Communications Security*, 2015.

[29] Brendan Saltaformaggio, Rohit Bhatia, Xiangyu Zhang, Dongyan Xu, and Golden G Richard III. Screen after previous screens: Spatial-temporal recreation of android app displays from memory images. In *Proceedings of USENIX Security Symposium*, 2016.

[30] Activity API Documentation. `https://developer.android.com/reference/android/app/Activity.html`.

[31] Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *Journal of the ACM (JACM)*, 44(4):585–591, 1997.

[32] Kendall-tau distance. `https://en.wikipedia.org/wiki/Kendall_tau_distance`.

[33] Android version market shares. `https://fossbytes.com/most-popular-android-versions-always-updated/`.

[34] Jemalloc. `https://github.com/jemalloc/jemalloc/wiki/Background`.

[35] mozjemalloc. `https://github.com/mozilla/gecko-dev/blob/master/memory/build/mozjemalloc.cpp`.

[36] Chrysaor malware. `https://android-developers.googleblog.com/2017/04/an-investigation-of-chrysaor-malware-on.html`.

[37] Lipizzan malware. `https://android-developers.googleblog.com/2017/07/from-chrysaor-to-lipizzan-blocking-new.html`.

[38] Mexico spyware. `https://www.theguardian.com/world/2017/jun/19/mexico-cellphone-software-spying-journalists-activists`.

[39] Domestic spying. `http://www.independent.co.uk/news/uk/home-news/exclusive-abusers-using-spyware-apps-to-monitor-partners-reaches-epidemic-proportions-9945881.html`.

[40] Theonespy. `https://www.theonespy.com`.

[41] FBI. Brian P. Regan espionage. `https://www.fbi.gov/history/famous-cases/brian-p-regan-espionage`, 2001.

[42] Gillette corporate espionage. `http://www.fraud-magazine.com/article.aspx?id=2147483718`.

[43] Dangers of distracted driving. `https://www.fcc.gov/consumers/guides/dangers-texting-while-driving`.

[44] Textalyzer. `http://www.nytimes.com/2016/04/28/science/driving-texting-safety-textalyzer.html`.

[45] Mccann investigations. `http://www.mccanninvestigations.com/media/6310/case_study_texting_and_driving.pdf`.

[46] Tesla autopilot accident. `https://www.theguardian.com/technology/2016/jul/01/tesla-driver-killed-autopilot-self-driving-car-harry-potter`.

[47] Judy Harrison. Death of 15-year-old nichole cable was kidnapping gone wrong, affidavit says. `http://bangordailynews.com/2013/05/29/news/bangor/death-of-15-year-old-nichole-cable-was-kidnapping-gone-wrong-affidavit-says/`, 2013.

[48] M. Foruhandeh, Y. Man, R. Gerdes, et al. SIMPLE: Single-frame based physical layer identification for intrusion detection and prevention on in-vehicle networks. In *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[49] S. U. Sagong, X. Ying, A. Clark, et al. Cloaking the clock: Emulating clock skew in controller area networks. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, pages 32–42, 2018.

[50] R. Bosch. CAN specification - Version 2.0, 1991.

[51] M. Jagielski, A. Oprea, B. Biggio, et al. Manipulating machine learning: Poisoning attacks and countermeasures for regression learning. In *IEEE Symposium on Security and Privacy (S&P)*, pages 19–35, 2018.

[52] S. Mei and X. Zhu. Using machine teaching to identify optimal training-set attacks on machine learners. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 2871–2877, 2015.

[53] K.-T. Cho and K. G. Shin. Error handling of in-vehicle networks makes them vulnerable. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1044–1055, 2016.

[54] Diurnal temperature variation. `https://en.wikipedia.org/wiki/Diurnal_temperature_variation`, 2019. [Online; accessed August 1, 2019].

[55] Comma.ai. opendbc: Democratize access to car decoder rings. `https://github.com/commaai/opendbc`, 2019. [Online; accessed August 1, 2019].

[56] M. Marchetti and D. Stabili. READ: Reverse engineering of automotive data frames. *IEEE Transactions on Information Forensics and Security*, 14(4):1083–1097, 2019.

[57] M. D. Pesé, T. Stacer, C. A. Campos, et al. LibreCAN: Automated CAN message translator. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2283–2300, 2019.

[58] B. Biggio, B. Nelson, and P. Laskov. Poisoning attacks against support vector machines. In *Proceedings of the 29th International Coference on International Conference on Machine Learning*, pages 1467–1474, 2012.

[59] N. Šrndic and P. Laskov. Practical evasion of a learning-based classifier: A case study. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 197–211, 2014.

[60] J. Petit and S. E. Shladover. Potential cyberattacks on automated vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 16(2):546–556, 2014.

[61] Microchip. MCP2515: Stand-alone CAN controller with SPI interface. `http://ww1.microchip.com/downloads/en/devicedoc/21801e.pdf`, 2007. [Online; accessed August 1, 2019].

[62] Philips. SJA1000: Stand-alone CAN controller. `https://www.nxp.com/docs/en/data-sheet/SJA1000.pdf`, 2000. [Online; accessed August 1, 2019].

[63] M. Pous, A. Atienza, and F. Silva. EMI radiated characterization of an hybrid bus. In *10th International Symposium on Electromagnetic Compatibility*, pages 208–213, 2011.

[64] Vector CANtech, Inc. Common high speed physical layer problems. `https://assets.vector.com/cms/content/know-how/_application-notes/AN-ANI-1-115_HS_Physical_Layer_Problems.pdf`, 2003. [Online; accessed August 1, 2019].

[65] S. Kramer, D. Ziegenbein, and A. Hamann. Real world automotive benchmarks for free. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

[66] M. Markovitz and A. Wool. Field classification, modeling and anomaly detection in unknown CAN bus networks. *Vehicular Communications*, 9:43–52, 2017.

[67] P. P. Lopez, E. S. Millan, J. C. V. Lubbe, and L. A. Entrena. Cryptographically secure pseudo-random bit generator for RFID tags. In *International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 1–6, 2010.

[68] Arduino. Random function. `https://www.arduino.cc/reference/en/language/functions/random-numbers/random/`, 2019. [Online; accessed August 1, 2019].

[69] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien. Controller area network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007.

[70] Anonymous. Duet: Supplementary files. `https://github.com/CAN-Bus-Duet/CAN-Bus-Duet`, 2019. [Online; accessed August 1, 2019].

[71] Crash Reconstruction Research Consortium. CAN traffic data. `http://tucrrc.utulsa.edu`, 2019. [Online; accessed August 1, 2019].

[72] M. Kang and J. Kang. A novel intrusion detection method using deep neural network for in-vehicle network security. In *IEEE 83rd Vehicular Technology Conference (VTC Spring)*, pages 1–5, 2016.

[73] P.-S. Murvay, A. Matei, C. Solomon, and B. Groza. Development of an AUTOSAR compliant cryptographic library on state-of-the-art automotive grade controllers. In *IEEE International Conference on Availability, Reliability and Security (ARES)*, pages 117–126, 2016.

[74] B. Groza, S. Murvay, A. V. Herrewege, and I. Verbauwhede. LiBrA-CAN: A lightweight broadcast authentication protocol for controller area networks. In *International Conference on Cryptology and Network Security*, pages 185–200. Springer, 2012.

[75] A. Lima, F. Rocha, M. Völp, and P. E. Veríssimo. Towards safe and secure autonomous and cooperative vehicle ecosystems. In *Proceedings of the 2nd ACM Workshop on Cyber-Physical Systems Security and Privacy*, pages 59–70, 2016.

[76] Tsvika Dagan and Avishai Wool. Parrot, a software-only anti-spoofing defense system for the CAN bus. *ESCAR EUROPE*, 2016.

[77] AutoSAR Secure Onboard Communication. `https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_SecureOnboardCommunication.pdf`.

[78] A. Van Herrewege, D. Singelee, and I. Verbauwhede. CANAuth-a simple, backward compatible broadcast authentication protocol for CAN bus. In *ECRYPT Workshop on Lightweight Cryptography*, 2011.

[79] Bosch Secure Communication for CAN. `http://www.bosch-semiconductors.com/ip-modules/can-ip-modules/plug-and-secure-communication-for-can/`.

[80] SPC58 Hardware Security Module. `www.st.com/resource/en/brochure/brspc58c.pdf`.

[81] Telemaco3 HW Crypto Engine. `http://www.st.com/resource/en/data_brief/sta1195.pdf`.

[82] M. Müter and N. Asaj. Entropy-based anomaly detection for in-vehicle networks. In *IEEE Intelligent Vehicles Symposium (IV)*, pages 1110–1115, 2011.

[83] Pal-Stefan Murvay and Bogdan Groza. Source identification using signal characteristics in controller area networks. *IEEE Signal Processing Letters*, 21(4):395–399, 2014.

[84] Wonsuk Choi, Hyo Jin Jo, Samuel Woo, Ji Young Chun, Jooyoung Park, and Dong Hoon Lee. Identifying ecus using inimitable characteristics of signals in controller area networks. *arXiv preprint arXiv:1607.00497*, 2016.

[85] STM8 Microcontrollers. `http://www.st.com/content/st_com/en/products/microcontrollers/stm8-8-bit-mcus.html?querycriteria=productId=SC1244`.

[86] K. Tindell and A. Burns. Guaranteeing message latencies on control area network (CAN). In *International CAN Conference*, 1994.

[87] K. Tindell, A. Burns, and A. J. Wellings. Calculating controller area network (CAN) message response times. *Control Engineering Practice*, 3(8):1163–1169, 1995.

[88] K. Han, S. D. Potluri, and K. G. Shin. On authentication in a connected vehicle: Secure integration of mobile devices with vehicular networks. In *ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, pages 160–169, 2013.

[89] S. Jain and J. Guajardo. Physical layer group key agreement for automotive controller area networks. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, pages 85–105, 2016.

[90] A. Groll and C. Ruland. Secure and authentic communication on existing in-vehicle networks. In *IEEE Intelligent Vehicles Symposium*, pages 1093–1097, 2009.

[91] K. Kang, Y. Baek, S. Lee, and S. H. Son. An attack-resilient source authentication protocol in controller area network. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 109–118, 2017.

[92] Speeding up and strengthening https connections for chrome on android. `https://security.googleblog.com/2014/04/speeding-up-and-strengthening-https.html`, 2014. [Online; accessed August 1, 2019].

[93] Yoan Chabot, Aurélie Bertaux, Christophe Nicolle, and Tahar Kechadi. Automatic timeline construction for computer forensics purposes. In *IEEE Joint Intelligence and Security Informatics Conference (ISI-EISIC 2014), 24-26 September, the Hague, Netherlands*. Institute of Electrical and Electronics Engineers, 2014.

[94] K Guðjónsson. Mastering the super timeline with log2timeline. *SANS Institute*, 2010.

[95] Christopher Hargreaves and Jonathan Patterson. An automated timeline reconstruction approach for digital forensic investigations. *Digital Investigation*, 9:S69–S79, 2012.

[96] Alex Levinson, Bill Stackpole, and Daryl Johnson. Third party application forensics on apple mobile devices. In *System Sciences (HICSS), 2011 44th Hawaii International Conference on*, pages 1–9. IEEE, 2011.

[97] Florian P Buchholz and Courtney Falk. Design and implementation of zeitline: a forensic timeline editor. In *DFRWS*, 2005.

[98] Jens Olsson and Martin Boldt. Computer forensic timeline visualization tool. *digital investigation*, 6:S78–S87, 2009.

[99] Kevin Chen, Andrew Clark, Olivier De Vel, and George Mohay. Ecf-event correlation for forensics. 2003.

[100] Bradley Schatz, George Mohay, and Andrew Clark. Rich event representation for computer forensics. In *Proceedings of the Fifth Asia-Pacific Industrial Engineering and Management Systems Conference (APIEMS 2004)*, volume 2, pages 1–16. Citeseer, 2004.

[101] MNA Khan and Ian Wakeman. Machine learning for post-event timeline reconstruction. In *First Conference on Advances in Computer Security and Forensics Liverpool, UK*, pages 112–121. Citeseer, 2006.

[102] Oxygen Forensic Analyst Timeline. `http://www.oxygen-forensic.com/en/products/oxygen-forensic-detective/analyst/timeline`.

[103] Cellebrite UFED. `https://www.cellebrite.com/en/solutions/pro-series/`.

[104] Dimitris Apostolopoulos, Giannis Marinakis, Christoforos Ntantogian, and Christos Xenakis. Discovering authentication credentials in volatile memory of android mobile devices. In *Collaborative, Trusted and Privacy-Aware e/m-Services*. 2013.

[105] 504ENSICS Labs. Dalvik Inspector. `http://www.504ensics.com/automated-volatility-plugin-generation-with-dalvik-inspector/`, 2013.

[106] Holger Macht. Live memory forensics on android with volatility. *Friedrich-Alexander University Erlangen-Nuremberg*, 2013.

[107] Vrizlynn LL Thing, Kian-Yong Ng, and Ee-Chien Chang. Live memory forensics of mobile phones. *Digital Investigation*, 7, 2010.

[108] Christian Hilgers, Holger Macht, Tilo Muller, and Michael Spreitzenbarth. Post-mortem memory analysis of cold-booted android devices. In *Proceedings of IT Security Incident Management & IT Forensics (IMF)*, 2014.

[109] Seung Jei Yang, Jung Ho Choi, Ki Bom Kim, Rohit Bhatia, Brendan Saltaformaggio, and Dongyan Xu. Live acquisition of main memory data from android smartphones and smartwatches. *Digital Investigation*, 23:50–62, 2017.

[110] Andreas Schuster. Searching for processes and threads in microsoft windows memory dumps. *Digital Investigation*, 3, 2006.

[111] Nick L Petroni Jr, Aaron Walters, Timothy Fraser, and William A Arbaugh. FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation*, 3, 2006.

[112] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. Robust signatures for kernel data structures. In *Proceedings of ACM Conference on Computer and Communications Security*, 2009.

[113] C Bugcheck. Grepexec: Grepping executive objects from pool memory. In *Proceedings of Digital Forensic Research Workshop*, 2006.

[114] Andrew Case, Andrew Cristina, Lodovico Marziale, Golden G Richard, and Vassil Roussev. FACE: Automated digital evidence discovery and correlation. *Digital Investigation*, 5, 2008.

[115] Martim Carbone, Weidong Cui, Long Lu, Wenke Lee, Marcus Peinado, and Xuxian Jiang. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of ACM Conference on Computer and Communications Security*, 2009.

[116] Paul Movall, Ward Nelson, and Shaun Wetzstein. Linux physical memory analysis. In *Proceedings of USENIX Annual Technical Conference, FREENIX Track*, 2005.

[117] Junyuan Zeng, Yangchun Fu, Kenneth A. Miller, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Obfuscation resilient binary code reuse through trace-oriented programming. In *Proceedings of ACM Conference on Computer and Communications Security*, 2013.

[118] Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang. SigGraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *Proceedings of Network and Distributed System Security Symposium*, 2011.

[119] Tal Garfinkel, Mendel Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of Network and Distributed System Security Symposium*, 2003.

[120] Nick L Petroni Jr, Timothy Fraser, AAron Walters, and William A Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proceedings of USENIX Security Symposium*, 2006.

[121] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T King. Digging for data structures. In *Proceedings of Symposium on Operating Systems Design and Implementation*, 2008.

[122] Brendan Saltaformaggio, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. DSCRETE: Automatic rendering of forensic information from memory images via application logic reuse. In *Proceedings of USENIX Security Symposium*, 2014.

[123] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of Network and Distributed System Security Symposium*, 2011.

[124] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of Network and Distributed System Security Symposium*, 2010.

[125] JongHyup Lee, Thanassis Avgerinos, and David Brumley. TIE: Principled reverse engineering of types in binary programs. In *Proceedings of Network and Distributed System Security Symposium*, 2011.

[126] Zhiqiang Lin, Junghwan Rhee, Chao Wu, Xiangyu Zhang, and Dongyan Xu. DIMSUM: Discovering semantic data of interest from un-mappable memory with confidence. In *Proceedings of Network and Distributed System Security Symposium*, 2012.

[127] Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. GUITAR: Piecing together android app GUIs from memory images. In *Proceedings of ACM Conference on Computer and Communications Security*, 2015.