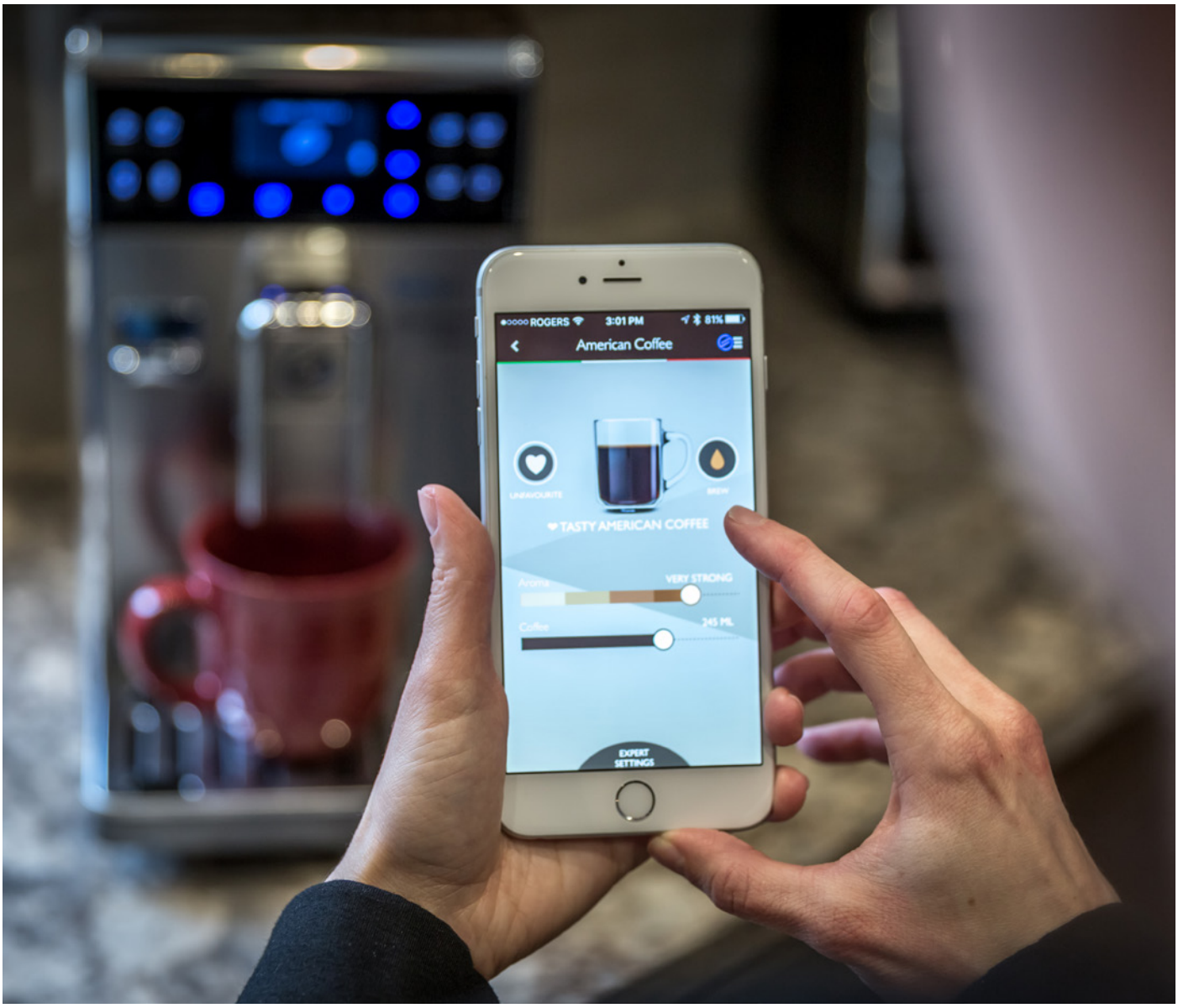
September 2017 COMPUTER SCIENCE APPLIED Issue #6

THE MORNING PAPER
QUARTERLY REVIEW

GREY FAILURE: REAL-WORLD EXPERIENCE RUNNING MICROSOFT AZURE

TAKING OVER A CITY WITH PHILIPS HUE SMART LAMPS

WHY RUST SHOULD BE YOUR SYSTEMS PROGRAMMING LANGUAGE OF CHOICE



A selection of papers from the world of computer science, as featured by Adrian Colyer on The Morning Paper blog

InfoQ queue

CONTENTS

Issue #6, September 2017

WELCOME...

Welcome to the latest Quarterly Review! I've chosen quite a mix of topics for this edition, covering experiences running production clouds, IoT and (in)security, privacy, systems programming, and mechanical sympathy.

Adrian Colyer

In 'Gray failure: the Achilles heel of cloud scale systems' we get a fascinating insight into the subtle causes of production problems in clouds at scale, based on experiences running Microsoft Azure. Gray failure happens when the monitoring in place thinks that a system is healthy, but clients of the system are experiencing degraded performance nonetheless. It's gray failure that ultimately turns out to be behind most production incidents. Addressing gray failure may call for a rethink on how we observe system health.

'Trajectory recovery from Ash: user privacy is NOT preserved in aggregated mobility data' is a great demonstration of just how hard it is to maintain privacy when releasing aggregated data sets. Starting from aggregate information simply revealing the number of users at a given location (e.g., a cell tower) at a given point in time, the authors show step-by-step how to recover individual user trajectories, and from there to identify the users associated with them. Until they showed me how, I wouldn't have thought it was possible!

Reading 'IoT goes nuclear: creating a ZigBee chain reaction' was another 'oh my goodness' moment for me. Exploiting the ZigBee protocol, the authors show how to take over a city using Philips Hue smart lamps. In a world of peer-to-peer networked devices with sufficient density, unstoppable worms can spread very rapidly over large areas.

In 'System programming in Rust: beyond safety' the authors make the case for Rust as the systems programming language of choice. We all know about

the type-safety arguments, but in this paper the authors demonstrate three other ways in which Rust can aid the development of systems software: software fault isolation, information flow control, and safe traversals of pointer-linked data structures. In the discussion that followed the original write-up on the blog, many people pointed out that strictly Rust has an affine type system, not a linear one. But don't let that put you off!

A trillion-edged graph is getting pretty big. In 'Mosaic: processing a trillion edge graph on a single machine' the authors remind us that sometimes scale-up can be a very cost-effective alternative to scale out. The 'single machine' that they use to process the graph is quite a special one, with Intel Xeon Phi co-processors and NVMe storage, but there's no denying the results when you design a system in sympathy with such hardware.

I hope you enjoy reading these selections as much as I did when I first came across them.

GRAY FAILURE: THE ACHILLES' HEEL OF CLOUD-SCALE SYSTEMS

If you're going to fail, fail properly, dammit! All this limping along in degraded mode, doing your best to mask problems, turns out to be one of the key causes of major availability breakdowns and performance anomalies in cloud-scale systems.

Huang et al. (HotOS XVI 2017)

"Gray failure: the Achilles' heel of cloud-scale systems" by Huang et al. (HotOS XVI 2017) is a short piece that discusses Microsoft Azure experiences with such so-called grey failures.

...Cloud practitioners are frequently challenged by gray failure: component failures whose manifestations are fairly subtle and thus defy quick and definitive detection. Examples of gray failure are severe performance degradation, random packet loss, flaky I/O, memory thrashing, capacity pressure, and non-fatal exceptions.

(All quotes from Huang et al. 2017.)

Just like all other kinds of failures, the larger your scale, the more common grey failures become.

A DEFINITION OF GREY FAILURE

First we need to understand the enemy. Anecdotes of grey failure have been known for years, but the term still lacks a precise definition.

We consider modeling and defining gray failure a prerequisite to addressing the problem.

Our first-hand experience with production cloud systems reveals that gray failure is behind most cloud incidents

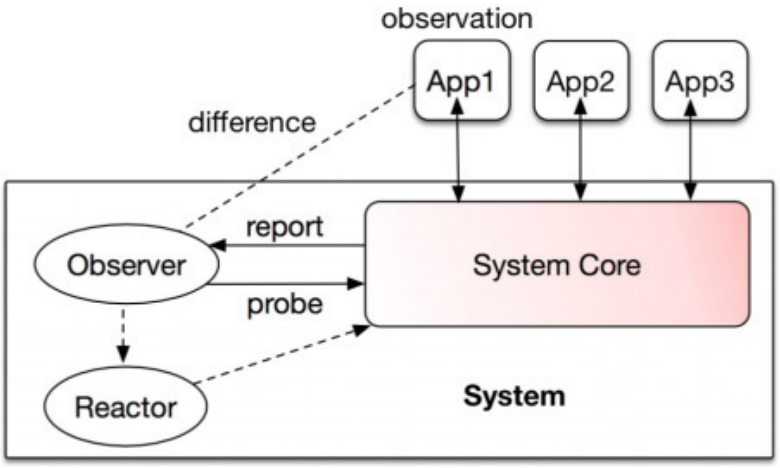


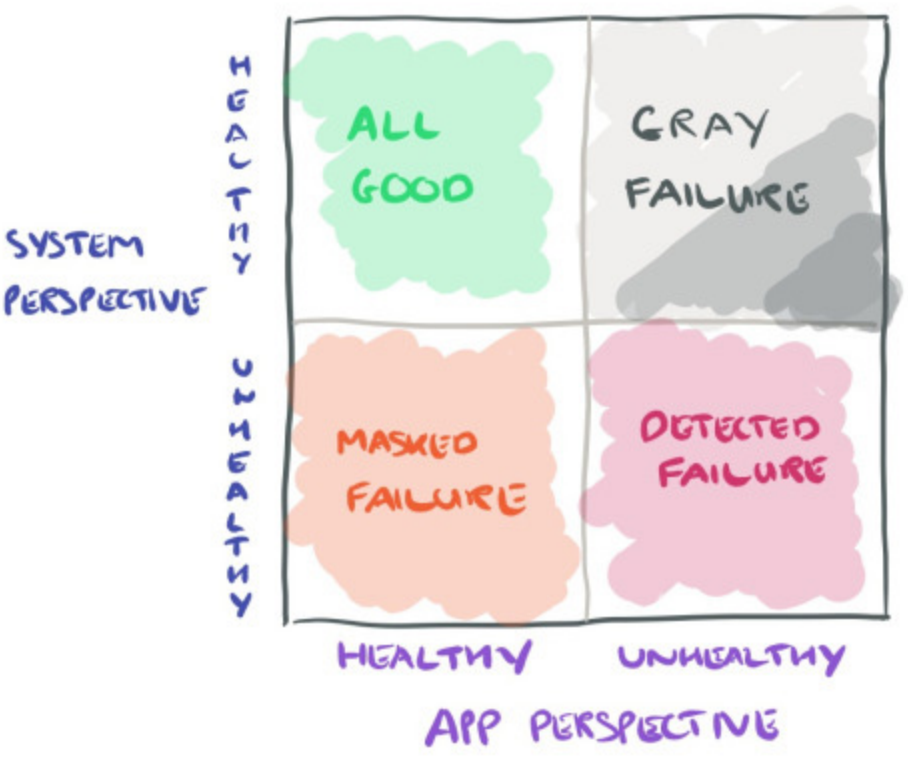
Figure 2: An abstract model to characterize gray failure

The model proposed by the authors is the result of an extensive study of real incidents in Azure cloud services. It's all a question of perspective.

Consider some system being monitored by a failure detector (the observer in figure 2). If the observer detects a fault, a reactor takes action (for example, restarting components). Meanwhile, there are all kinds of clients (apps in figure 2) that are using the system. By virtue of interacting with it, these apps make their own observations of the health of the system (by detecting slow response, reports of errors, etc.). Cloud systems may be used simultaneously by many different kinds of apps, each with its own perspective on the health of the system.

We define gray failure as a form of differential observability. More precisely, a system is defined to experience gray failure when at least one app makes the observation that the system is unhealthy, but the observer observes that the system is healthy.

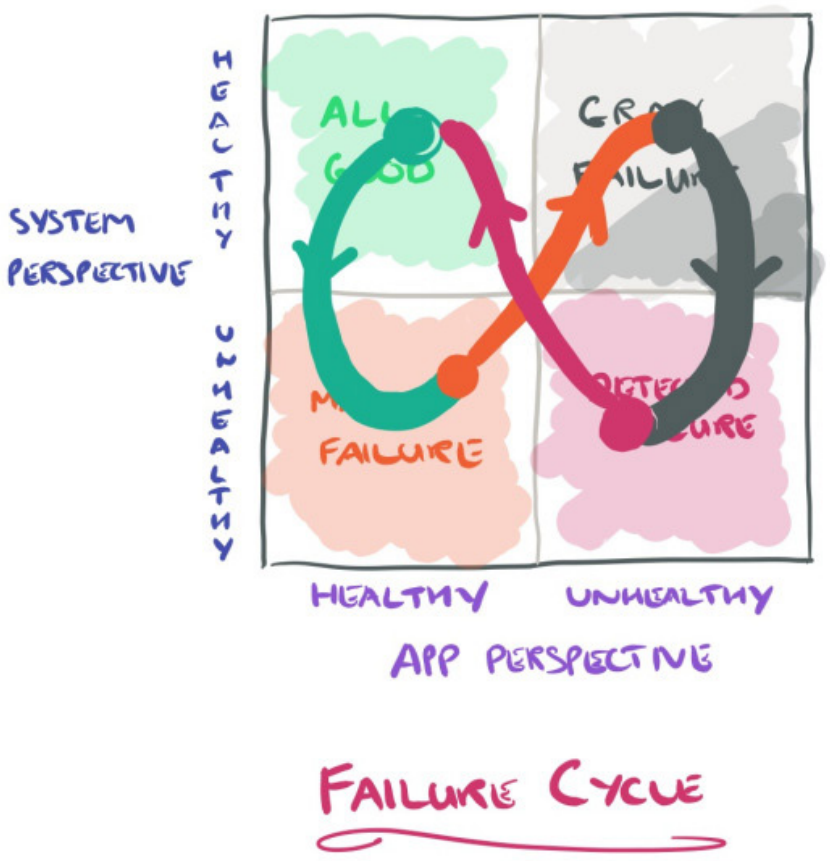
Considering the system and its client apps, we can draw the following quadrant of grey:



There's also a cycle of grey that many systems tend to follow.

...Initially the system experiences minor faults (latent failure) that it tends to suppress. Gradually, the system transits into a degraded mode (gray failure) that is externally visible but which the observer does not see. Eventually the degradation may reach a point that takes the system down (complete failure), at which point the observer also realizes the problem. A typical example is a memory leak.

During a grey failure with intermittent misbehaviour, we can go round the cycle many times.



GREY FAILURES

Let's look at some examples of grey failure from the Azure world to make this all a bit more concrete.

MORE REDUNDANCY == LESS AVAILABILITY!?

Datacentre networks can route around failing switches, but typically do not reroute packets when switches experience intermittent failures such as random and silent packet drops. The grey failure results in application glitches or increased latency.

As a consequence, we sometimes see cases were increasing redundancy actually lowers availability.

As with tail latency, the chances of experiencing this increase exponentially with the fan-out factor (number of services involved) in processing a given request. If we have n core switches (or indeed, any n resources used to process requests) then the probability that a certain switch is traversed by a request is given by $1 - \left(\frac{n-1}{n}\right)^m$ where m is the fan-out factor.

*This probability rapidly approaches 100% as **m** becomes large, meaning each such request has a high probability of involving every core switch. Thus a gray failure at any core switch will delay nearly every front-end request.*

The more switches you have (for example, for increased redun-

We sometimes see cases where increasing redundancy actually lowers availability

dancy), the more likely at least one of them will experience a grey failure!

WORKS FOR ME!

The authors describe incidents where unhealthy VMs are internally experiencing severe network issues, but the failure detector is receiving heartbeats via a host agent that communicates with the VM using local RPC. Thus the problem is never spotted and no recover happens until a user reports an issue. This leads to long gaps between the point of user impact and the point of resolution.

WHEN THE CURE IS WORSE THAN THE DISEASE

An Azure storage data server was experiencing severe capacity constraint, but a bug prevented the storage manager from detecting the condition. As the storage manager sent more requests to the overloaded data server, it eventually crashed and rebooted, which did not fix the underlying problem so the cycle repeated.

The failure detector did pick up on the continual rebooting and concluded the data server was irreparable and so took it out of service. This put pressure on the remaining healthy servers, causing more servers to degrade and experience the same fate.

Naturally, this eventually led to a catastrophic cascading failure.

(It sounds like they need a backpressure mechanism!)

FINGER-POINTING AT DAWN

VMs run in compute clusters, and their virtual disks lie in storage clusters accessed over the network. Sometimes a storage or network issue means a VM can't access its virtual disk, which can lead to a crash.

If no failure detector detects the underlying problem with the storage or network, the compute-cluster failure detector may incorrectly attribute the failure to the compute stack in the VM. For this reason, such gray failure is challenging to diagnose and respond to. Indeed, we have encountered cases where teams responsible for different subsystems blame each other for the incidents since no one has clear evidence of the true cause.

PROTECTING AGAINST GREY FAILURES

First, note that the standard distributed systems mechanisms for availability don't work well with grey failures because such failures are not part of their failure model.

A myriad of techniques have been proposed to leverage redundancy and replication to tolerate component faults, e.g., primary/backup replication, RAID, Paxos, and chain replication. Many of these techniques assume a simple failure model: fail-stop. Different from fail-stop, a component experiencing gray failure appears to be still working but is in fact experiencing severe issues. Such discrepancy can negatively im-

pact traditional techniques and cause fault-tolerance anomalies.

The next thought that went through my mind was "Isn't this just inadequate monitoring?" If you want to understand how well your website is performing, for example, send it external requests similar to the ones that users will submit. Under that hypothesis, the differential observability problem is caused by insufficient/inappropriate observations that don't capture what truly matters to the client.

Although it would be ideal to eliminate differential observability completely by letting the system measure what its apps observe, it is practically infeasible. In a multi-tenant cloud system that supports various applications and different workloads, it is unreasonable for a system to track how it is used by all applications.

There's a deeper problem lurking here too, which goes back to the binary fail-stop model. Failure detectors typically send an "it failed" message that then initiates recovery, but grey failure is not such a black-and-white issue.

A natural solution to gray failure is to close the observation gaps between the system and the apps it services. In particular, system observers have traditionally focused on gathering information reliably about whether components are up or down. But, gray failure makes these not just simple black-or-white judgements. Therefore, we advocate moving from singular failure detection (e.g., with heartbeats) to multi-dimensional health monitoring.

Another way to increase the chances of closing the observation gap is to combine observations from many different vantage points.

...Since gray failure is often due to isolated observations of an observer, leveraging the observations from a large number of different components that are complementary to each other can help uncover gray failure rapidly. Indeed, many gray failure cases we investigated are only detectable in a distributed fashion because each individual component has only a partial view of the entire system (so the gray failures are intrinsic).

Exactly how to do this is an open question. If the observations are taken close to the core of the system, for example, they may not see what the clients see. But if taken near the apps, built-in fault tolerance mechanisms may mask faults and delay detection of gray failures.

We envision an independent plane that is outside the boundaries of the core system but nevertheless connected to the observer or reactor.

This sounds very like a classic centralised-control-style solution to me. (I could well be wrong; we only have this one sentence to go on!) I wonder what a solution starting from the premises of promise theory might look like instead?

A final technique that may help with grey failure comes through understanding how the grey-failure cycle evolves over time. If these time-series patterns can be learned, then we can predict more serious failures and react earlier. Since many faults truly are benign, though, care must be taken not to do more harm than good.

THE LAST WORD

As cloud systems continue to scale, the overlooked gray failure problem becomes an acute pain in achieving high availability. Understanding this problem domain is thus of paramount importance.... We argue that to address the problem, it is crucial to reduce differential observability, a key trait of gray failure.

TRAJECTORY RECOVERY FROM ASH: USER PRIVACY IS NOT PRESERVED IN AGGREGATED MOBILITY DATA

Xu et al. (WWW 2017)

Borrowing a little from Simon Wardley's marvellous Enterprise IT Adoption Cycle, here's roughly how my understanding progressed as I read through "Trajectory Recovery from Ash: User Privacy Is NOT Preserved in Aggregated Mobility Data" from Xu et al. (WWW 2017):

Huh? What? How? Nooooo. Oh, no. Oh, s*@\#!

Xu et al. show us that even in a dataset in which you might initially think there is no chance of leaking information about individuals, they can recover data about individual users

with between 73% and 91% accuracy — even in datasets that aggregate data on tens of thousands to hundreds of thousands of users! Their particular context is mobile location data, but underpinning the discovery mechanism is a reliance on two key characteristics:

1. Individuals tend to do the same things over and over (regularity) — i.e., there are patterns in the data relating to given individuals.

2. These patterns are different across different users (uniqueness).

Therefore, statistical data with similar features is likely to suffer from the same privacy breach. Unfortunately, these two features are quite common in the traces left by humans, which have been reported in numerous scenarios, such as credit card records, mobile application usage, and even web browsing. Hence, such a privacy problem is potentially severe and universal, which calls for immediate attention from both academia and industry.

(All quotes from Xu et al. 2017.)

As we'll see, there are a few other details that matter, which in my mind might make it harder to transfer to other problem domains — chiefly the ability to define appropriate cost functions — but even if it does relate only to location-based data, it's still a very big deal.

Let's take a look at what's in a typical aggregated mobility dataset, and then go on to see how Xu et al. managed to blow it wide open.

AGGREGATED MOBILITY DATA (ASH)

In an attempt to preserve user privacy, owners of mobility data tend to publish only aggregated data — for example, the number of users covered by a cellular tower at a particular timestamp. The statistical data is of broad interest with applications in epidemic controlling, transportation scheduling, business intelligence, and more. The aggregated data is sometimes called the "ash" of the original trajectories.

The operators believe that such aggregation will preserve users' privacy while providing useful statistical information for academic research and commercial usage.

To publish aggregated mobility data, first the original mobility records of individual mobile users are grouped by time slots (windows), and then aggregate statistics are computed for each window (e.g., the number of mobile users covered by each base station).

Two desirable properties are assumed to hold from following such a process:

3. No individual information can be directly acquired from the datasets, with the aggregated mobility data complying with the k-anonymity privacy model.

4. The aggregate statistics are accurate.

Although the privacy leakage in publishing anonymized individual's mobility records has been recognized and extensively studied, the privacy issue in releasing aggregated mobility data remains unknown.

Looking at this from a differential privacy perspective, you might already be wondering about the robustness of being able to detect, for example, whether or not a particular individual is represented in the dataset. But what we're about to do goes much further. Take a dataset collected over some time period, which covers a total of M locations (places). In each time slot, we have the number of users in each

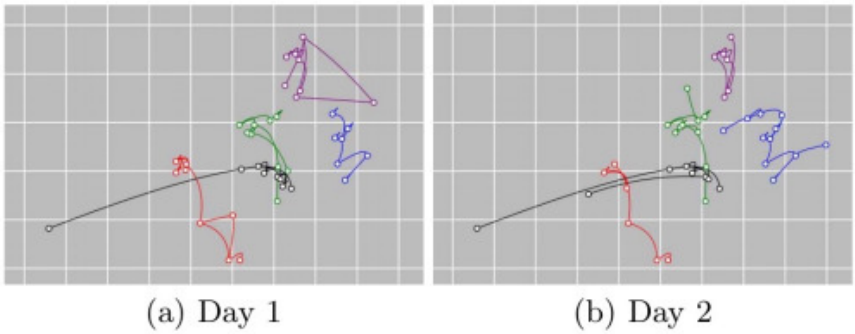


Figure 1: Mobility trajectories of five randomly selected mobile users in two days.

of those places (i.e., an M-dimensional vector). That's all you get. From this information only, it's possible to recover 73%-91% of all the individual users' trajectories — i.e., which locations they were in at which times, and hence how they moved between them.

Huh? What? How?

REVEALING INDIVIDUAL USERS FROM AGGREGATE DATA

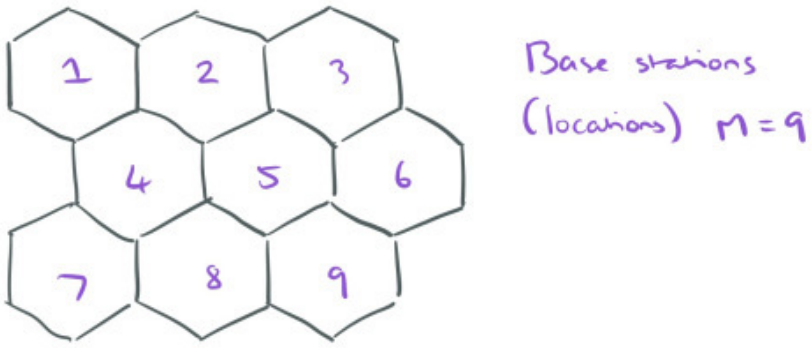
The first thing that's easy to work out is how many individual users there are at any point in time: just sum up all user counts at each place at one timestamp. Call that N.

Now, consider what other clues might be available in the data. Individual users tend to have fairly coherent mobility trajectories (what they do today, they're likely to do again tomorrow), and these trajectories are different across different users. See, for example, the trajectories of five randomly selected selected users over two days in the figure above. Each user's pattern is unique, with strong similarity across both days.

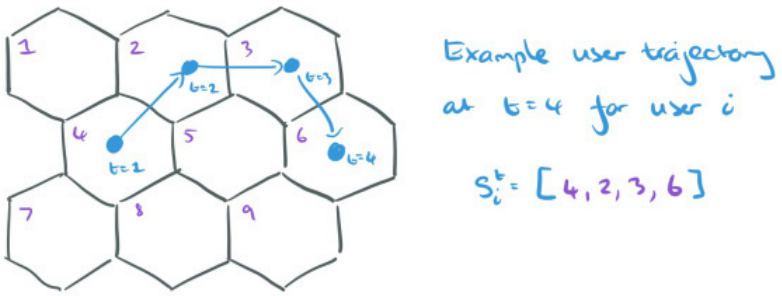
The key to recovering individual trajectories is to exploit these twin characteristics of regularity for individual users and unique patterns across users. Even so, it seems a stretch when all we've got is user counts by time and place!

We're going to build up estimates for the trajectories of each of the N users, one time step at a time. Let the estimate for the trajectory of the ith individual user at time step t — called s_i^t — be represented by a sequence of locations $[q_i^1, q_i^2, q_i^3, \ldots, q_i^t]$.

For example, here's a world with M=9 locations:



And here's a trajectory in that world for a user to time t, represented as a sequence of t locations:

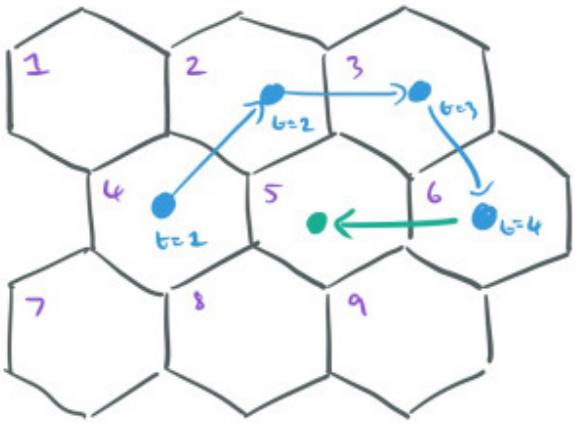


We want to decide on the most likely location for each user at time step t+1, subject to the overall constraint that we know how many users in total there must be at each location at time t+1.

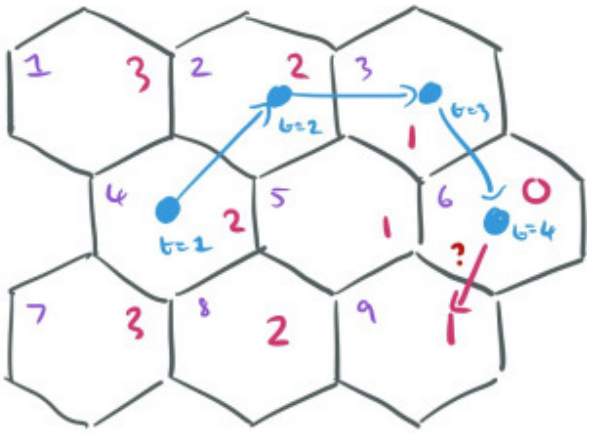
The description of how this next part works is very terse in the paper itself, but with a little bit of reverse engineering on my part, I believe I've reached an understanding. The secret is to formulate the problem as one of completing a decision matrix. This decision matrix takes a special form: it has N rows (one for each user trajectory to date), and N columns. Say we know that there must be two users in location 1 and four users in location 2. Then the first two columns in the matrix will represent the two user slots available in location 1, and the next four columns will represent the four slots available in location 2, and so on.

In the decision matrix X^t, $x^t_{i,j} = 1$ if next location for trajectory i is the location identified by column j, and 0 otherwise. A valid completion of the matrix has every row adding up to one (each trajectory is assigned one and only one next location), and every column adding up to one (each slot is filled by one and only one user).

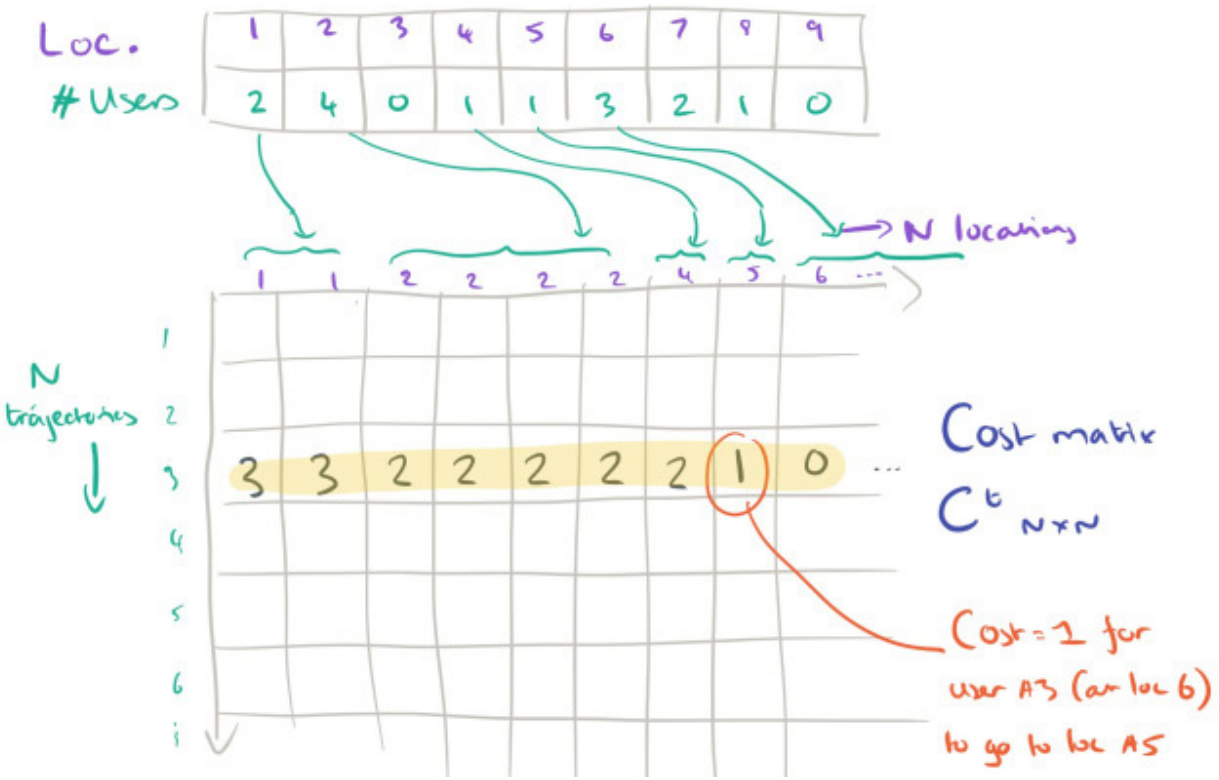
Thus the example decision matrix above indicates that the next location for the trajectory of user #3 has been assigned as location 5.



When we complete the decision matrix, we don't just make random assignments of slots, of course! That's where the cost matrix C^c comes in. The cost matrix is also a matrix of size NxN, with the same row and column structure as the decision matrix. Instead of being filled with 1's and 0's though, $c^t_{i,j}$ contains a value representing the cost of moving to the location for slot j given the trajectory so far for user i .Take, for example, the trajectory for a user in the illustration below, which currently finishes in location 6. We simply might set the cost of moving to each potential next location to the number of hops in the grid to get there (red numbers). The actual cost functions used are more complex than this; this example is just to help you get the idea.



Here then is what a subsection of the cost matrix for the user in the above sketch might look like:



We'll return to to how to define the cost functions in a moment. For now, note that the problem has now become the following:

$$\text{Minimize} \quad \sum_{i=1}^{N}\sum_{j=1}^{N} c_{i,j}^{t} \times x_{i,j}^{t}, \tag{1}$$

$$\text{subject to} \quad x_{i,j}^{t} = \{0,1\}, \sum_{i=1}^{N} x_{i,j}^{t} = 1, \sum_{j=1}^{N} x_{i,j}^{t} = 1.$$

The above-formulated problem is equivalent to the Linear Sum Assignment Problem, which has been extensively studied and can be solved in polynomial time with the Hungarian algorithm.

Space prevents me from explaining the Hungarian algorithm, but the Wikipedia link above does a pretty good job of it (it's actually pretty straightforward, check out the "Matrix interpretation" section to see how it maps in our case).

We can recover each step of individual trajectories in polynomial time

Thus far then, we've discovered a way to represent the problem such that we can recover each step of individual trajectories in polynomial time, so long as we can define suitable cost matrices.

Noooooo.

BUILDING COST MATRICES: IT'S NIGHT AND DAY

At night, people tend not to move around very much, as illustrated by these plots from the operator and app datasets used in the evaluation:

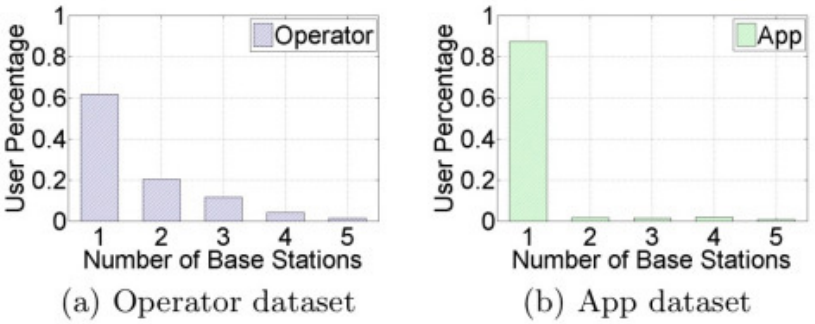


Figure 4: Percentage of users with different number of nighttime locations.

Not only that, but the nighttime location of individual users tends to be one of their top most visited locations (often the top location):

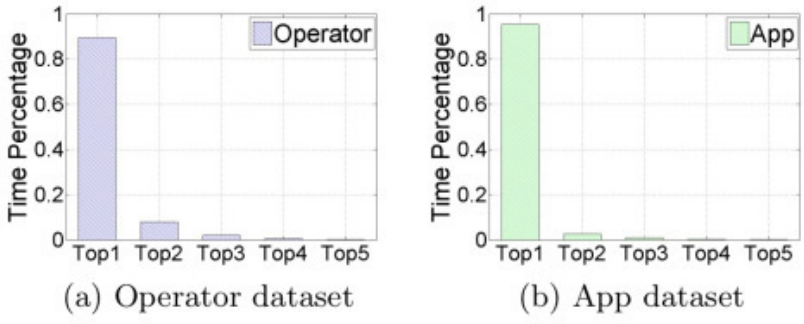


Figure 5: Percentage of time staying in the most frequent locations during nighttime.

For nighttime then, it makes sense to use the distance between the location of the user trajectory at time t and the location being considered for time t+1 as the cost.

In the daytime, people tend to move about.

The key insight is the continuity of human mobility, which enables the estimation of next location using the current location and velocity.

Let the estimated next location using this process be l, then we can use as the cost function the distance between the l and the location being considered for time t+1.

It is worth noting that the Hungarian algorithm is currently the most efficient algorithm to solve [the linear sum assignment problem], but still has computational complexity of $O(n^3)$. To speed things up, we adopt a suboptimal solution to reduce the dimension of the cost matrix by taking out the pairs of trajectories and location points with cost below a predefined threshold and directly linking them together.

LINKING TRAJECTORIES ACROSS DAYS

Using the night and day approaches, we can recover mobile users' sub-trajectories for each day. Now we need to link sub-trajectories together across days. Here we exploit the regularity exhibited in the day-after-day movements of individual users.

Specifically, we use the information gain of connecting two sub-trajectories to measure their similarities.

The entropy in a trajectory is modelled based on the frequency of visiting different locations, and the information gain from linking two sub-trajectories is modelled as the difference between the entropy of the combined trajectory and the sum of the entropies of the individual trajectories divided by two. For one user, we should see relatively little information gain if the two trajectories are similar, whereas for different users we should see much larger information gain. And indeed we do:

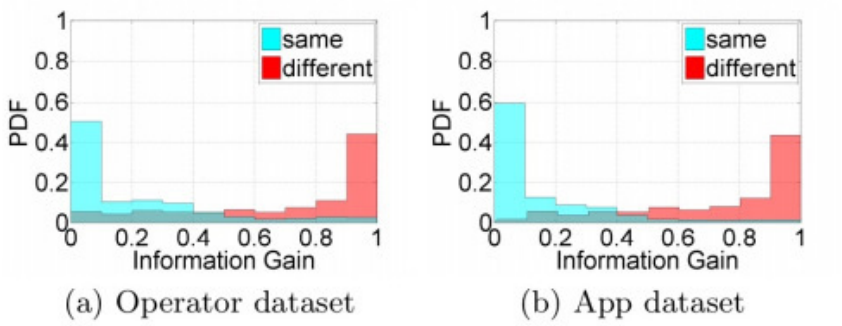


Figure 7: The PDF of information gain in linking across dats sub-trajectories contributed by a single user or a different user.

To conclude, we design an unsupervised attack framework that utilizes the universal characteristics of human mobility to recover individuals' trajectories in aggregated mobility datasets. Since the proposed framework does not require any prior information of the target datasets, it can be easily applied on other aggregated mobility datasets.

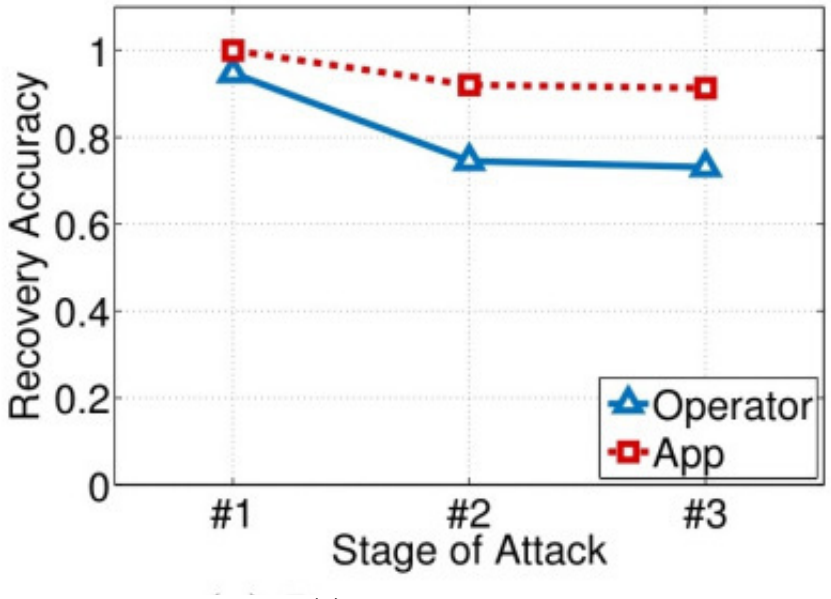
Once the individual trajectories are separated out, it has been shown to be comparatively easy, with the help of small amounts of external data such as credit card records, to re-identify individual users (associated them with trajectories).

Oh, no.

EVALUATION

The authors evaluate the technique on two real-world datasets. The app dataset contains data for 15,000 users collected by a mobile app that records a user's location over a two-week period when activated. The operator dataset contains data from a major mobile-network operator for 100,000 users over a one-week period. Tests are run on aggregate data produced from these datasets, and the recovered trajectories are compared against ground truth.

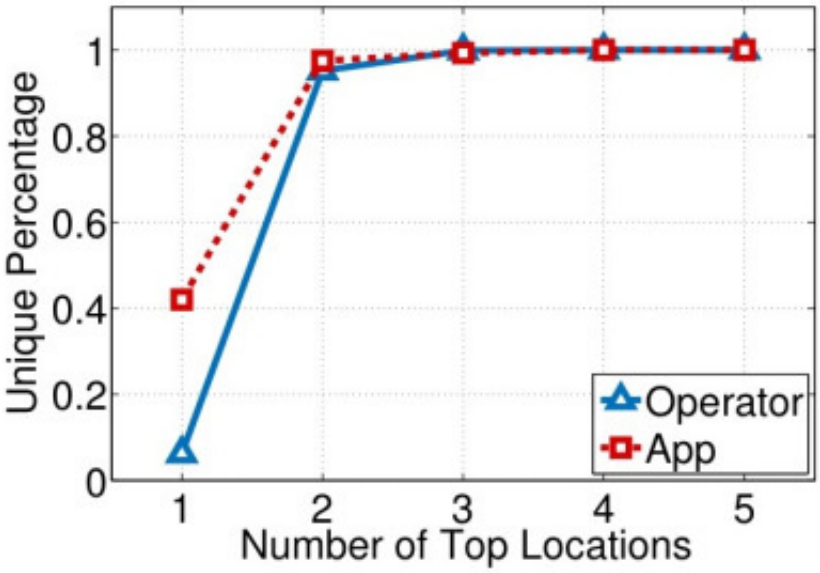
In the figures that follow, stage #1 represents nighttime trajectory recovery, stage #2 daytime trajectory recovery, and stage #3 the linking of sub-trajectories across days. Here we can see the recovery accuracy for the two datasets:



(a) Recovery accuracy

For the app dataset, 98% of nighttime trajectories are correctly recovered, falling to 91% accuracy by the final step. The corresponding figures for the operator dataset are 95% and 73%.

For the recovered trajectories, the following chart shows the percentage that can be uniquely identified given just the top-k locations for k of 1 to 5.



(c) Uniqueness

From the results, we can observe that given the two most frequent locations of the recovered trajectories, over 95% of them can be uniquely distinguished. Therefore, the results indicate that the recovered trajectories are very unique and vulnerable to be re-identified with little external information.

To put that more plainly, given the aggregated dataset, and knowledge of your home and work locations, there's a very good chance I can recover your full movements!!!

Oh, s*@\#!

Decreasing the spatial resolution (i.e., using more coarse-grained locations) actually increases the chances of successful trajectory recovery (but only to the location granularity of course). It's harder to link these recovered trajectories to individual people, though, as human mobility becomes less unique in coarser-grained datasets.

Decreasing the temporal resolution (only releasing data for larger time windows) increases both the chances of successful trajectory recovery and of re-identification.

The best defence for preserving privacy in aggregated mobility datasets should come as no surprise to you: we need to add some carefully designed random noise. What that careful design is, though, we're not told!

...A well-designed perturbation scheme can reduce the regularity and uniqueness of mobile users' trajectories, which has the potential for preserving mobile users' privacy in aggregated mobility data.

IOT GOES NUCLEAR: CREATING A ZIGBEE CHAIN REACTION

You probably don't need another reminder about the woeful state of security in IoT, but "IoT Goes Nuclear: Creating a ZigBee Chain Reaction" by Ronen et al. (IEEE Security and Privacy 2017) may well give you further pause for thought about the implications.

Ronen et al. (IEEE Security and Privacy 2017)

The opening paragraph sounds like something out of science fiction – except that it's a demonstrated reality today:

> ***Within the next few years, billions of IoT devices will densely populate our cities. In this paper, we describe a new type of threat in which adjacent IoT devices will infect each other with a worm that will rapidly spread over large areas, provided that the density of compatible IoT devices exceeds a certain critical mass.***

(All quotes from Ronen et al. 2017.)

The ZigBee protocol is the radio link of choice for many IoT devices since it is simple and widely available at low cost and with low power consumption. It has much lower bandwidth than Wi-Fi, but that doesn't matter for many IoT applications. The popular Philips Hue smart lamps use ZigBee, for example.

Suppose you could build a worm that jumps directly from one lamp to another using their ZigBee wireless connectivity and their physical proximity. If the installed base of lamps in a city is sufficiently dense, you could take them all over in no time, with the worm spreading like a physical

Within the next few years, billions of IoT devices will densely populate our cities

virus. The authors of today's paper demonstrate exactly how to build such a worm:

...We developed and verified such an infection using the popular Philips Hue smart lamps as a platform.... The attack can start by plugging in a single infected bulb anywhere in the city, and then catastrophically spread everywhere within minutes.

If plugging in an infected bulb is too much hassle, the authors also demonstrate how to take over bulbs by wardriving around in a car or by warflying a drone. (I get the impression they had fun with this one!) Based on percolation theory, we can estimate that if a city such as Paris has around 15,000 or more randomly located smart lamps, then the attack will spread everywhere within the city.

What we demonstrate in this paper is that even IoT devices made by big companies with deep knowledge of security, which are protected by industry-standard cryptographic techniques, can be misused by hackers can rapidly cause city-wide disruptions which are very difficult to stop and investigate.

I just p3wned your city!

The whole paper is a wonderful tour de force that combines two main elements:

1. A correlation power analysis (CPA) attack against the CCM encryption mode used to encrypt and verify firmware updates allowed the authors to encrypt, sign, and upload malicious OTA updates to infect lamps.

2. A takeover attack allowed the authors to take control over lamps from long distances without using custom hardware.

Let's look at these two components in turn, and then revisit the analysis that shows how easily such a worm can spread through a city given a relatively low density of installed bulbs. We'll conclude by considering some of the attacks this enables and appeal again for better IoT security.

UPLOADING MALICIOUS FIRMWARE

ZigBee provides an OTA update feature that the authors exploit. They started out by finding several different older Hue models that had previously had software updates. By plugging these in and letting them update, a recording of the update process can be made for analysis, which enabled discovery of the hex code that OTA requires to be sent over the air. The team then wrote a Python script to impersonate a light bulb and ask for an OTA update using the code.

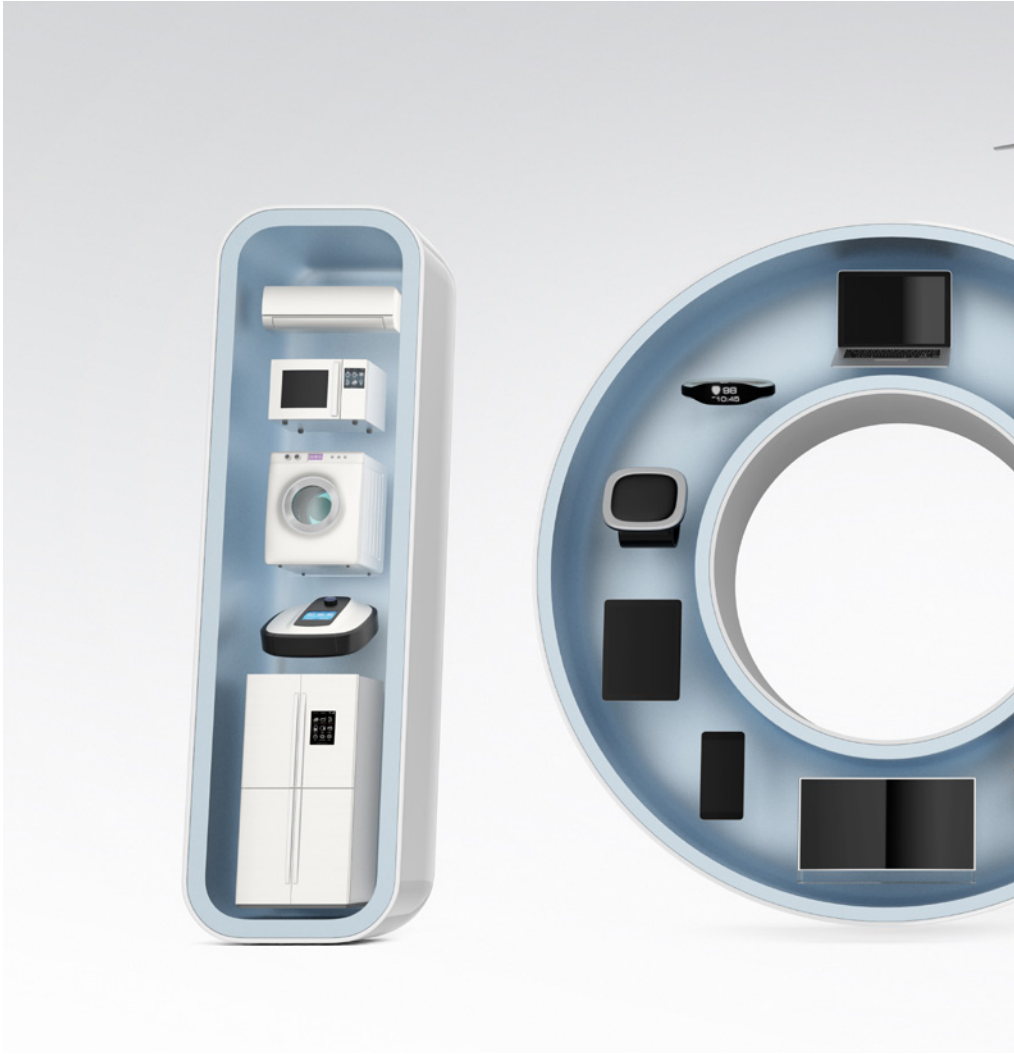
The firmware image that we recorded was not dependent on our impersonated light-bulb MAC address. This brought us to the conclusion that the software is protected by a single key that is shared at least between all the lamps from a specific model.

(This is a bad idea, but is very common!) On this assumption of symmetric cryptography, recovering the encryption and authentication keys from any one Philips Hue bulb will allow an attacker to perform a software update to any other light bulb and upload malicious code — at least, once the OTA firmware image structure has been reverse engineered, which the authors also did.

To get those keys, the authors developed a side-channel power analysis attack to break the Philips bootloader:

Side channel power analysis measures the power consumed by a digital device as it performs operations. With this attack it is possible to infer something about the data being processed by the device, which was first demonstrated as a method of breaking cryptographic algorithms by Kocher et al.

Section 6 of the paper provides the full details of how differential power analysis (DPA) and correlation power analysis (CPA) were used to recover the key bits. We don't really have space to cover that here (and besides, I'm not sure I could do it justice) but it's well worth reading if you're interested, and it's a great demonstration of the weakness of security by obscurity even when you think you're doing everything else right.



REACHING BULBS FROM A DISTANCE

At this point, if we can communicate with a bulb, we can upload our firmware to it. Communication normally requires close physical proximity though.

[The ZigBee chips in the Hue lamps] will ignore any request to reset or to change their affiliation unless it is sent from a ZigBee transmitter, which is only a few centimeters away from the lamp. Even though the attacker can try to spoof such a proximity test by using very-high-power transmitters, the fact that the received power decreases quadratically with the distance makes such brute force attacks very hard.

Section 7 in the paper details how the authors managed to bypass the proximity check in the Atmel ZigBee chip's Touchlink implementation. It all boils down to an assumption in the code that a message with a zero-value transaction ID has already been rejected as invalid. However, this check is actually only carried out for "scan request" messages and not for any other message in the protocol.

This means that we can send any other message assuming zero value for the Transaction and Re-

sponse IDs and it will be received and processed as a valid message by the light.

So the team send a unicast "Reset to Factory New Request" command to the light with a zero Transaction ID. Upon receiving the message, the light undergoes a factory reset and begins scanning for a new network to join by sending ZigBee Beacon request messages.

...We respond with a ZigBee Beacon message with the Association Permit flag set to true. This causes the light to start a ZigBee association process and join our network.

Here's further description of this part of the attack.

We installed three Philips Hue lamps in offices at the first floor of our faculty building. We successfully tested our full attack from a car, which was parked across the lawn at a distance of about 50 meters.... We then successfully tested our attack while "wardriving" the car at the far edge of the lawn.

Then, the researchers installed five smart bulbs in the third floor of an office building and mounted the attack kit onto a DJI drone. Successful takeover was achieved with a flyby (warflying) and attack code that caused all the lamps to repeatedly signal SOS in Morse code while the drone hovered in front of the building.

Building a self-spreading worm is now possible by combining the ability to sign and encrypt arbitrary binaries with our knowledge of the long-rage takeover attack vulnerability.

HOW MANY BULBS DO YOU NEED BEFORE THE INFECTION TAKES OVER A WHOLE CITY?

Once you can spread from bulb to bulb, all you need to do is infect one seed bulb (or several) and let nature take its course. Recall from some of the network theory we've looked at previously that in a random graph, once a certain number of edges have been added, a phase change occurs in which a single giant connected component emerges. If we model this by assuming that N smart lamps (nodes) are placed randomly within a city, and an edge between any two lamps that are less than distance D apart, then the field of percolation theory can tell us the critical value of N (the percolation threshold) at which this giant connected component should emerge.

For the city of Paris, with a total area of 105 square kilometers, and assuming D is 100 meters, we find that the critical mass of installed lamps N in the whole city of Paris needs to be about 15,000.

Since the Philips Hue smart lamps are very popular in Europe and especially in affluent areas such as Paris, there is a very good chance

that this threshold has in fact already been exceeded, and thus the city is already vulnerable to massive infections via the ZigBee chain reaction described in this paper.

YOU TOOK OVER MY BULB — SO WHAT?

Here are four scenarios:

- A bricking attack makes the attack irreversible such that any effect caused by the worm (blackout, constant flickering, etc.) is permanent. Once the worm is in place, it can determine what further OTA updates to accept, if any.

- Wireless-network jamming uses the ZigBee band, which overlaps with Wi-Fi. By going into test mode, which transmits a continuous wave signal without first checking for a clear channel, it would be possible to mount DoS attacks disrupting all Wi-Fi or other 2.4-GHz traffic.

- Ronen and Shamir demonstrated data infiltration and exfiltration using Philips Hue lamps at a rate of about 10 KB per day. Using infected lamps, similar covert channels can be created but at much higher rates.

- Let's get a bit more serious.... It's possible to use the Philips Hue to trigger epileptic seizures or to drive LEDs at frequencies that increase long-term discomfort in humans. Imagine this happening simultaneously across a whole city!

CAN WE PLEASE TAKE IOT SECURITY A LITTLE MORE SERIOUSLY NOW?

Our attacks exploit a specific implementation bug of the Touchlink commission protocol and a specific design for the OTA update process, but they are just an example of the way security in IoT is designed today. The Atmel code we reviewed was very well written and documented, but it is extremely difficult to implement complex state machines of such protocols without any bugs. The main problem is in the insecure design of the ZLL standard itself. We believe this will not be the last bug or attack found against ZLL commissioning.... The sharp contrast between the open and inclusive manner in which the TLS 1.3 standard was designed and the secretive work on the ZigBee 3.0 specification that is still not open to the public is a big part of the problem.

The attack is also another reminder that "security by obscurity has failed time after time" and that any reuse of keys across devices is a big security risk.

We should work together to use the knowledge we gained to protect IoT devices or we might face in the near future large scale attacks that will affect every part of our lives.

SYSTEM PROGRAMMING IN RUST: BEYOND SAFETY

In "System Programming in Rust: Beyond Safety", Balasubramanian et al. (HotOS 2017) set out the case why we should switch to Rust for all of our systems programming.

Balasubramanian et al. (HotOS 2017)

Despite many advances in programming languages, clean-slate operating systems, hypervisors, key-value stores, web servers, [and] network and storage frameworks are still developed in C, a programming language that is in many ways closer to assembly than to a modern high-level language. Today, the price of running unsafe code is high.... Why are we still using C?

(All quotes from Balasubramanian et al. 2017.)

About two thirds of the 2017 CVEs relating to the Linux kernel can be attributed to the use of an unsafe language, and pervasive use of pointer aliasing, pointer arithmetic, and unsafe type casts defeat the use of software verification tools.

So why are we still using C? Because safe languages have overheads that are too high for many use cases, argue the authors. (And because of familiarity and large existing C codebases, I would add.)

Is it reasonable to sacrifice safety for performance, or should we prioritize safety and accept its overhead? Recent developments in programming languages suggest that this might be a false dilemma, as it is possible to achieve both performance and safety without compromising on either.

Enter Rust!

Rust achieves both safety and performance by embracing linear types. In the Rust ownership model, a variable that is bound to an object acquires ownership of that object. When the variable goes out of scope, the object is deallocated. Ownership can be transferred to another variable, but doing so destroys the original binding. You can also borrow an object within, breaking the binding, but only within the syntactic scope of the declaration, and when it does not exceed the scope of the primary binding.

This ownership model eliminates pointer aliasing. There is an unsafe subset of the language that is not subject to the single-ownership restrictions (best left to the standard library to implement, e.g., doubly-linked lists).

It is possible to achieve both performance and safety without compromising on either

There is also a mechanism for safe read-only aliasing that involves wrapping the object with a reference-counted type, `Rc` or `Arc`. And if you must have write aliasing (for example, to a shared resource), you can enforce this dynamically by wrapping the object with the `Mutex` type:

In contrast to conventional languages, this form of aliasing is explicit in the object's type signature....

Several projects have demonstrated Rust's suitability for building low-level high-performance systems, but the authors of this paper want us to go further and consider additional benefits for systems programming that are enabled by Rust's type system: software fault isolation (SFI); program analysis, especially static information flow control (IFC); and safe traversal of pointer-linked data structures, which enables automation of tasks such as checkpointing.

(Recall that we looked at timely dataflow as used by Mosaic in May 2017. Timely dataflow is implemented in Rust.)

SOFTWARE FAULT ISOLATION

SFI is the idea of enforcing process-like boundaries around program modules in software, without relying on hardware protection. Modern SFI implementations enable low-enough cost isolation in some applications (e.g., browser plugins and some device drivers), but "their overhead becomes unacceptable in applications that require high-throughput communication across protection boundaries."

Consider, for example, a network processing framework that forwards packets through a pipeline of filters that should ideally be isolated from each other.

Sending data across protection boundaries requires copying it, which is unacceptable in a line-rate system.

You can avoid copying with a tagged shared heap, but this introduces other run-time overheads (up to 100%) for tag validation on each pointer dereference.

Rust's single ownership model allows us to implement zero-copy SFI. The Rust compiler ensures that, once a pointer has been passed across isolation boundaries, it can no longer be accessed by the sender.

The authors demonstrate how to build an SFI library in Rust, which supports secure communication across isolation boundaries with negligible overhead. The library exports two data types: protection domains (PDs) and remote references (rrefs).

Arguments and return values of remote invocations follow the usual Rust semantics: borrowed references are accessible to the target PD for the duration of the call; all other arguments change their ownership permanently. The sole exception is remote references: the object pointed to by

an rref stays in its original domain and can only be accessed from the domain holding the reference via remote invocation.

Remote references are weak references to a reference table. All remote invocations are proxied through this table.

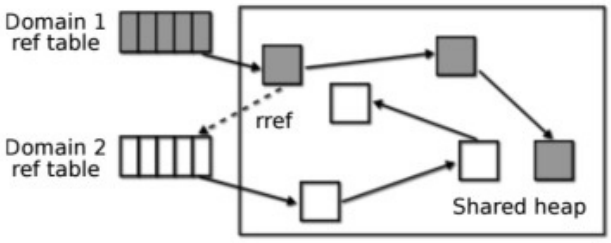


Figure 1: All PDs share the common heap. Cross-domain refernces (rrefs) are mediated by the reference table.

If a panic occurs inside a domain, its domain reference table is cleared and a user-provided function is invoked to re-initialize it from a clean slate.

The cost of this isolation is accessed in the context of a network processing framework using null filters that forward batches of packets without doing any work on them.

The overhead grows from 90 CPU cycles for one-packet batches to 122 cycles for 256-packet batches, which is roughly the cost of two or three L3 cache accesses.

INFORMATION FLOW CONTROL

Static IFC enforces confidentiality by tracking the progress of sensitive data through a program. It doesn't require a great leap of imagination to see that aliasing make this problem much harder — you have to keep track of all aliases to a variable and everywhere that they are used.

Modern alias analysis achieves efficiency by sacrificing precision, posing a major barrier to accurate IFC. By restricting aliasing, Rust sidesteps the problem.

The authors implement a proof-of-concept IFC for Rust. Rust macros transform a program into an abstract representation in which the value of each variable is simply represented by its security label. Input variables are initialised with user-provided labels, arithmetic expressions over secure values compute an upper bound of their arguments, and an auxiliary program counter tracks the flow of information via branching on labelled variables.

The resulting abstract program is verified using a verifier implemented as an extension of SMACK. This is able to detect problems such as the secret data leak in the following program fragment.

```
9 let mut buf = Buffer::new();
10 #[label(non-secret)] //security annotation for IFC
11 let nonsec = vec![1,2,3];
12 #[label(secret)] //security annotation for IFC
13 let sec = vec![4,5,6];
14 buf.append(nonsec);
15 buf.append(sec); // buf now contains secret data
16 println!("{:?}",buf.data);//ERROR:leaks secret data
17 //println!("{:?}",nonsec);
```

SAFE TRAVERSALS

...Checkpointing, transactions, replication, multi-version concurrency, etc. involve snapshotting parts of program state. This, in turn, requires traversing pointer-linked data structures in memory. Ideally one would like to generate this functionality automatically and for arbitrary user-defined data types. However, doing so in a robust way can be complicated in the presence of aliasing.

In Rust, the problem is much simpler. By default, all references in Rust are unique owners and can be safely traversed without extra checks. If aliasing is present, it is detectable through the use of `Rc` and `Arc` wrappers, which makes these wrappers a convenient place to deal with aliasing with minimal changes to user code and without expensive lookups.

The authors built an automatic checkpointing library for Rust following this observation using a `Checkpointable` trait (interface) and a custom implementation for `Rc` (`Arc` could be extended similarly). A compiler plugin automatically generates an implementation of the `Checkpointable` trait for types composed of scalar values and references to other checkpointable types.

Our library adds the checkpointing capability to arbitrary user-defined data types; in particular, it checkpoints objects with internal aliases correctly and efficiently.

THE LAST WORD

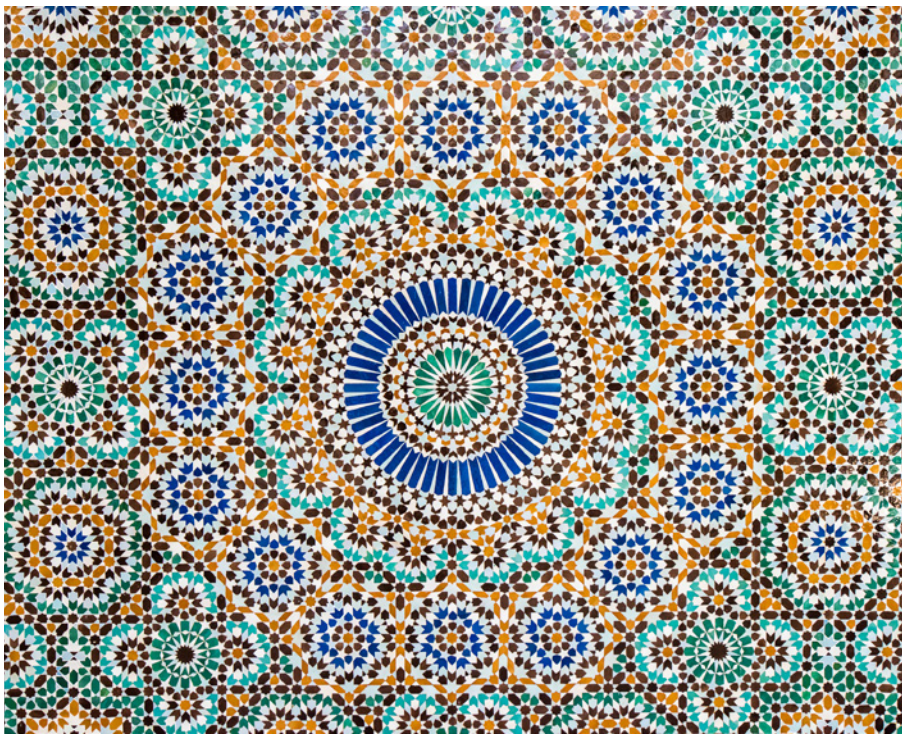
We show that Rust enables system programmers to implement powerful security and reliability mechanisms like SFI, IFC, and automatic checkpointing more efficiently than any conventional language. This is just the tip of the iceberg: we believe that further exploration of linear types in the context of real systems will yield more game-changing discoveries.

MOSAIC: PROCESSING A TRILLION-EDGE GRAPH ON A SINGLE MACHINE

Unless your graph is bigger than Facebook's, you can process it on a single machine! So say Maass et al., in "Mosaic: Processing a trillion-edge graph on a single machine" (Proceedings of the Twelfth European Conference on Computer Systems — EuroSys 2017).

Maass et al. (Proceedings of the Twelfth European Conference on Computer Systems — EuroSys 2017)

With the inception of the Internet, large-scale graphs comprising web graphs or social networks have become common. For example, Facebook recently reported their largest social graph comprises 1.4 billion vertices and 1 trillion edges. To process such graphs, they ran a distributed graph-processing engine, Giraph, on 200 machines. But, with Mosaic, we are able to process large graphs, even proportional to Facebook's graph, on a single machine.



In this case it's quite a special machine, with Intel Xeon Phi co-processors and NVMe storage. But it's really not that expensive: the Xeon Phi used in the paper costs around $549 and a 1.2-TB Intel SSD 750 costs around $750. How much do large distributed clusters cost in comparison — especially when using expensive interconnects and large amounts of RAM?

So Mosaic costs less, but it also consistently outperforms other state-of-the-art out-of-core (secondary-storage) engines by 3.2 to 58.6 times, and shows comparable performance to distributed graph engines. At one-trillion-edge scale, Mosaic can run an iteration of PageRank in 21 min-

utes (after paying a fairly hefty one-off set-up cost).

(And remember, if you have a less-than-a-trillion-edges scale problem, say just a few billion edges, you can do an awful lot with just a single thread too!) Another advantage of the single-machine design is a much simpler approach to fault tolerance....

Handling fault tolerance is as simple as checkpointing the intermediate stale data (i.e., vertex array). Further, the read-only vertex array for the current iteration can be written to disk parallel to the graph processing; it only requires a barrier on each superstep. Recovery is also trivial; processing can resume with the last checkpoint of the vertex array.

There's a lot to this paper. Perhaps the two most central aspects are design sympathy for modern hardware and the Hilbert-ordered tiling scheme used to divide up the work. So I'm going to concentrate mostly on those in the space available.

EXPLOITING MODERN HARDWARE

Mosaic combines fast host processors for concentrated memory-intensive operations, with coprocessors for compute and I/O-intensive components. The chosen coprocessors for edge processing are Intel Xeon Phis. In the first generation (Knights Corner), a Xeon Phi has up to 61 cores with four hardware threads each and a 512-bit single-instruction, multiple-data (SIMD) unit per core. To handle the amount of data needed, Mosaic exploits NVMe devices that allow terabytes of storage with up to 10 times the throughput of SSDs. PCIe-attached NVMe devices can deliver nearly a million IOPS per device with high bandwidth (e.g. Intel SSD 750) — the challenge is to exhaust this available bandwidth.

As we've looked at before in the context of data stores, taking advantage of this kind of hardware requires different design trade-offs:

We would like to emphasize that existing out-of-core engines cannot directly improve their performance without a serious redesign. For example, GraphChi improves the performance only by 2-3% when switched from SSDs to NVMe devices or even RAM disks.

SCALING TO 1 TRILLION EDGES

That's a lot of edges. And we also know that real-world graphs can be highly skewed, following a power-law distribution. A vertex-centric approach makes for a nice programming model ("think like a vertex"), but locality becomes an issue when locating outgoing edges. Mosaic takes some inspiration from COST here:

To overcome the issue of non-local vertex accesses, the edges can be traversed in an order that preserves vertex locality using, for example, the Hilbert order in COST using delta encoding.

Instead of a global Hilbert ordering of edges (1 trillion edges, remember), Mosaic divides the graph into tiles (batches of local graphs) and uses Hilbert ordering for tiles.

A Hilbert curve is a continuous fractal space-filling curve. Imagine a big square table (an adjacency matrix) where the rows and columns represent source vertices and target vertices respectively. An edge from vertex 2 to vertex 4 will therefore appear in row 2, column 4. If we traverse this table in Hilbert order (following the path of the curve), we get a nice property whereby points close to each other along the curve also have nearby (x,y) values — i.e., include similar vertices.

The following illustration shows a sample adjacency matrix, the Hilbert-curve path through the table, and the first two tiles that have been extracted.

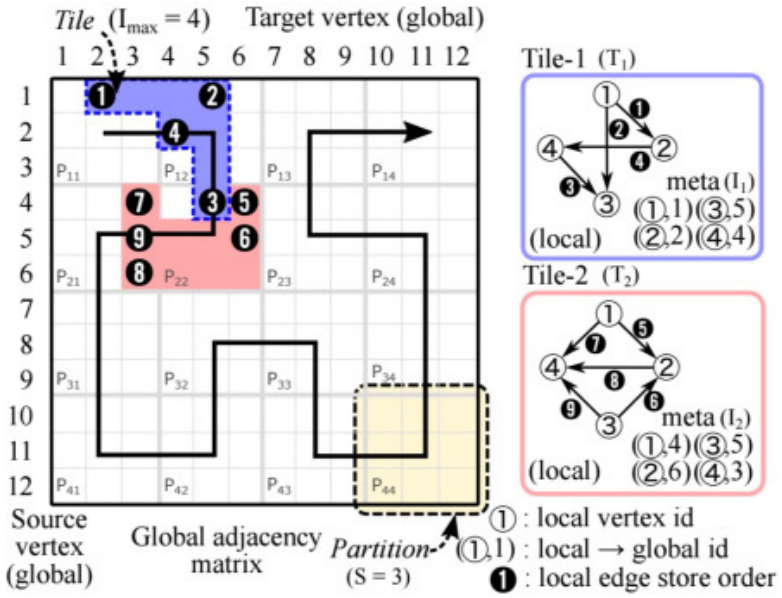


Figure 1: The Hilbert-ordered tiling and the data format of a tile in MOSAIC. There are two tiles (blue and red regions) illustrated by following the order of the Hilbert curve (arrow). Internally, each tile enodes a local graph using local, short vertex identifiers. The meta index I_j translates between the local and global vertex identifiers. For example, edge ❷, linking vertex 1 to 5, is sorted into tile T_1, locallt mapping the verices 1 to ① and 5 to ③.

Also note that the edge space is divided into partitions (labelled P_{11}, P_{12} etc. in the figure above). It's important to note again here that Mosaic does not process every single edge in Hilbert order (that would require a global sorting step); instead, it statically partitions the adjacency matrix into 2^{16} x 2^{16} blocks (partitions), and then pro-

cesses the partitions themselves in Hilbert order. The tile structure is populated by taking a stream of partitions as input.

We consume partitions following the Hilbert order, and add as many edges as possible into a tile until its index structure reaches the maximum capacity to fully utilize the vertex identifier in a local graph.... This conversion scheme is an embarrassingly parallel task.

At run time, Mosaic processes multiple tiles in parallel on four Xeon Phis. Each has 61 cores, giving 244 processing instances running in parallel.

Due to this scale of concurrent access to tiles, the host processors are able to exploit the locality of the shared vertex states associated with the tiles currently being processed, keeping large parts of these states in the cache.

When processing a tile, neighbouring tiles are pre-fetched from NVMe devices to memory in the background, following the Hilbert order.

SYSTEM COMPONENTS

The full Mosaic system looks like this:

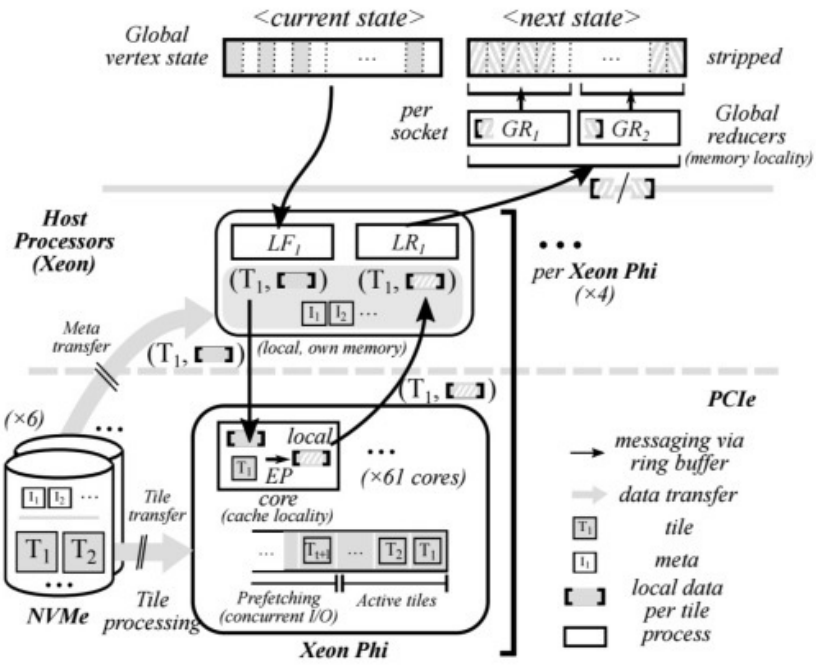


Figure 3: MOSAIC's components and the data flow between them. The components are split intro *scale-out* for xeon Phi (*local fetcher* (LF), *local reducer* (LR) and *edge processor* (EP)) and scale-up for host (global reducer (GR)). The *edge processor* runs on a Xeon Phi, operating on local graphs while host components operate on the global graph. The vertex state is available as both a read-only *current state* as well as write-only *next* state.

The Xeon Phi coprocessors run local fetchers, edge processors, and reducers. Each Phi has one fetcher and one reducer, and an edge processor per core. The local reducer retrieves the computed responses from the coprocessors and aggregates them before sending them back for global processing.

The host processor runs global reducers that are assigned partitions of the global vertex state and received and process input from local reducers.

As modern systems have multiple NUMA domains, Mosaic assigns disjoint regions of the global vertex state array to dedicated cores running on each NUMA socket, allowing for large, concurrent NUMA transfers in accessing the global memory.

PROGRAMMING MODEL

Mosaic uses the numerous yet slower co-processor cores to perform edge processing on local graphs, and the faster host processors to reduce the computation results to global vertex states.

To exploit such parallelism, two key properties are required in Mosaic's programming abstraction, namely commutativity and associativity. This allows Mosaic to schedule computation and reduce operations in any order.

The API is similar to the Gather-Apply-Scatter model, but extended to a Pull-Reduce-Apply (PRA) model:

Pull(e) — For every edge (u,v), Pull(e) computes the result of the edge e by applying an algorithm-specific function on the value of the source vertex u and the related data such as in-degrees or out-degrees.

Reduce(v1,v2) — This takes two values for the same vertex and combines them into a single output. It's invoked by edge processors on coprocessors, and global reducers on the host.

Apply(v): After reducing all local updates to the global array, Apply(v) runs on each vertex state in the array, allowing the graph algorithm to perform non-associative operations. Global reducers run this at the end of each iteration.

It all fits together something like this.

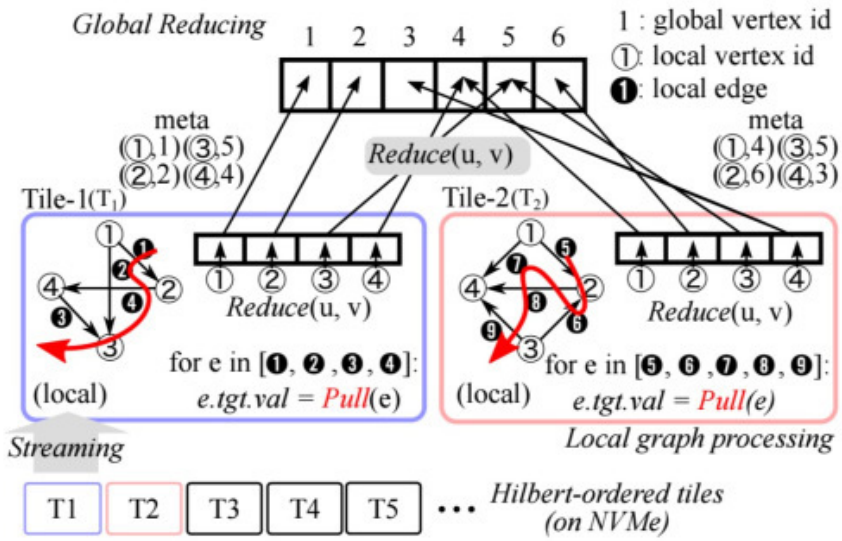


Figure 5: The execution model of MOSAIC: it runs *Pull( )* on local graphs while *Reduce( )* is being employed to merge vertex states, for both the local as well as the global graphs.

Mosaic consistently outperforms other state-of-the-art out of core engines by 3.2 to 58.6 times, and shows comparable performance to distributed graph engines

And here's what PageRank looks like using the Mosaic programming model:

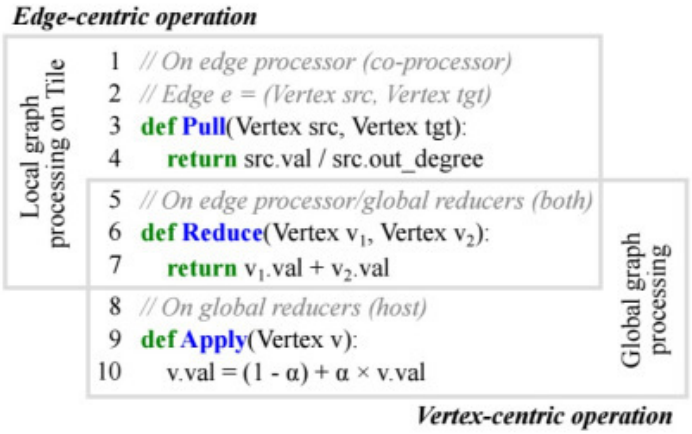


Figure 4: The Pagerank implementation on MOSAIC. *Pull( )* operates on edges, returning the impact of the source vertex. *Reduce( )* accumulates both local and global impacts, while *Apply( )* applies the damping factor α to the vertices on each iteration.

EXPERIMENTS

The team implemented seven popular graph algorithms and tested them on two different classes of machines — a Vortex gaming PC and a Ramjet workstation — using six different real-world and synthetic datasets, including a synthetic trillion-edge graph following the distribution of Facebook's social graph. The Vortex machine has one Xeon Phi, whereas Ramjet has four.

Graph	#vertices	#edges	Raw data	MOSAIC	
				Data size (reduction, bytes/edge)	Prep. time
*rmat24	16.8 M	0.3 B	2.0 GB	1.1 GB (-45.0%, 4.4)	2 m 10 s
twitter	41.6 M	1.5 B	10.9 GB	7.7 GB (-29.4%, 5.6)	2 m 24 s
*rmat27	134.2 M	2.1 B	16.0 GB	11.1 GB (-30.6%, 5.5)	3 m 31 s
uk2007-05	105.8 M	3.7 B	27.9 GB	8.7 GB (-68.8%, 2.5)	4 m 12 s
hyperlink14	1,724.6 M	64.4 B	480.0 GB	152.4 GB (-68.3%, 2.5)	50 m 55 s
*rmat-trillion	4,294.9 M	1,000.0 B	8,000.0 GB	4,816.7 GB (-39.8%, 5.2)	30 h 32 m

Table 4: The graph datasets used for MOSAIC's evaluation. The data size of MOSAIC represents the size of complete, self-contained information of each graph dataset, including tiles and meta-data generated in the conversion step. The * mark indicates synthetic datasets. Each dataset can be effciently encoded with 29.4-68.8% of its original size due to MOSAIC tile structure.

The execution times are shown in the following table.

Graph	#edges (ratio)	PR‡		BFS†		WCC†		SpMV‡		TC‡		SSSP†		BP‡	
		vortex	ramjet	vortex	ramjet	vortex	ramjet	vortex	ramjet	vortex	ramjet	vortex	ramjet	vortex	ramjet
*rmat24	0.3 B (1×)	0.72 s	0.31 s	18.04 s	3.52 s	18.51 s	3.92 s	0.58 s	0.30 s	2.06 s	1.46 s	45.32 s	11.71 s	1.09 s	0.47 s
twitter	1.5 B (5×)	4.99 s	1.87 s	51.65 s	11.20 s	59.46 s	18.58 s	4.59 s	1.66 s	13.90 s	9.42 s	269.99 s	54.06 s	7.78 s	5.34 s
*rmat27	2.1 B (8×)	6.28 s	3.06 s	71.79 s	17.02 s	78.07 s	22.18 s	6.02 s	2.74 s	22.10 s	16.52 s	353.75 s	101.95 s	10.64 s	8.42 s
uk2007-05	3.7 B (14×)	5.75 s	1.76 s	11.35 s	1.56 s	18.85 s	5.34 s	5.68 s	1.49 s	11.32 s	4.31 s	11.41 s	2.05 s	15.02 s	4.57 s
hyperlink14	64.4 B (240×)	100.85 s	21.62 s	15.61 s	6.55 s	2302.39 s	708.12 s	85.45 s	19.28 s	-	68.03 s	17.32 s	8.68 s	-	70.67 s
*rmat-trillion	1,000.0 B (3,726×)	-	1246.59 s	-	3941.50 s	-	7369.39 s	-	1210.67 s	-	5660.35 s	-	-	-	-

Table 5: The execution time for running graph algorithms on MOSAIC with real and synthetic (marked ⋆) datasets. We report the seconds per iteration for iterative algorithms (‡: PR, SpMV, TC, BP), while reporting the total time taken for traversal algorithms (†: BFS, WCC, SSSP). TC and BP need more than 64 GB of RAM for the hyperlink14 graph and thus do not run on vortex. We omit results for rmat-trillion on SSSP and BP as a weighted dataset of rmat-trillion would exceed 8 TB which currently cannot be stored in our hardware setup.

Mosaic shows 686-2,978 M edges/sec processing capability depending on datasets, which is even comparable to other in-memory engines (e.g. 695-1,390M edges/sec in Polymer) and distributed engines (e.g. 2,770-6,335 M edges/sec for McSherry et al.'s Pagerank-only in-memory cluster system.

On Ramjet, Mosaic performs one iteration of Pagerank in 21 minutes.

Overall, here's how Mosaic stacks up against some of the other options out there:

Dataset	Single machine								Distributed systems		
	Out-of-core				In-memory		GPGPU		Out-of-core	In-memory	
	MOSAIC	GraphChi †	X-Stream †	GridGraph †	Polymer	Ligra	TOTEM	GTS	Chaos$_{32}$	GraphX$_{16}$	McSherry$_{16}$
rmat24	0.31 s	14.86 s (47.9×)	4.36 s (14.1×)	1.12 s (3.6×)	0.37 s	0.25 s	-	-	-	-	-
twitter	1.87 s	65.81 s (35.2×)	19.57 s (10.5×)	5.99 s (3.2×)	1.06 s	2.91 s	0.56 s	0.72 s	-	12.2 s	0.53 s
rmat27	3.06 s	100.02 s (32.7×)	27.57 s (9.0×)	8.38 s (2.7×)	1.93 s	6.13 s	1.09 s	1.42 s	28.44 s	-	
uk2007-05	1.76 s	103.18 s (58.6×)	52.39 s (29.8×)	14.84 s (8.4×)	-	-	0.85 s	1.24 s	-	8.30 s	0.59 s

Table 6: The execution time for one iteration of Pagerank on out-of-core, in-memory engines and GPGPU systems running either on a single machine or on distributed systems (subscript indicates number of nodes). Note the results for other out-of-core engines (indicated by †) are conducted using six NVMes in a RAID 0 on ramjet. We take the numbers for the GPGPU (from [33]), in-memory systems and the distributed systems from the respective publications as an overview of different architectural choices. We include a specialized in-memory, cluster Pagerank system developed by McSherry et al. [43] as an upper bound comparison for in-memory, distributed processing and show the GraphX numbers on the same system for comparison. MOSAIC runs on ramjet with Xeon Phis and NVMes. MOSAIC outperforms the state-of-the-art out-of-core engines by 3.2–58.6× while showing comparable performance to GPGPU, in-memory and out-of-core distributed systems.

Compared to GPGPU systems, Mosaic is slower by a factor of up to 3.3, but can scale to much larger graphs.

Mosaic is approximately one order of magnitude faster than distributed out-of-core systems.

Mosaic's performance is competitive with distributed in-memory systems (and beats GraphX by 4.7x-6.5x). These systems pay heavily for distribution.

Converting datasets to the tile structure takes two to four minutes for datasets up to about 30 GB. For hyperlink14, with 21 M edges and 480 GB of data it takes 51 minutes. For the trillion-edge graph (8,000 GB), it takes about 30 hours. However, after only a small number of iterations of Pagerank, Mosaic is beating systems that don't have a preprocessing step, showing a quick return on investment.

Finally, let's talk about the COST. It's very nice to see the authors addressing this explicitly. On uk2007-05, Ramjet matches a single-threaded host-only in-memory implementation with 31 Xeon Phi cores (and ultimately can go up to 4.6x faster than the single-threaded solution). For the Twitter graph, Mosaic needs 18 Xeon Phi cores to match single-thread performance, and ultimately can go up to 3.86x faster.

HERE IS WHAT WE'VE COVERED IN THE PREVIOUS ISSUES

TOWARD SUSTAINABLE INSIGHTS, OR WHY POLYGAMY IS BAD FOR YOU

WHEN DNNS GO WRONG - ADVERSARIAL EXAMPLES AND WHAT WE CAN LEARN FROM THEM

THOU SHALT NOT DEPEND ON ME: ANALYSING THE USE OF OUTDATED JAVASCRIPT LIBRARIES ON THE WEB

REDUNDANCY DOES NOT IMPLY FAULT TOLERANCE: ANALYSIS OF DISTRIBUTED STORAGE REACTIONS TO SINGLE ERRORS AND CORRUPTIONS

THE CURIOUS CASE OF THE PDF CONVERTER THAT LIKES MOZART

SIMPLE TESTING CAN PREVENT MOST CRITICAL FAILURES

TOWARDS DEEP SYMBOLIC REINFORCEMENT LEARNING

WHEN CSI MEETS PUBLIC WIFI: INFERRING YOUR MOBILE PHONE PASSWORD VIA WIFI SIGNALS

KRAKEN: LEVERAGING LIVE TRAFFIC TESTS TO IDENTIFY AND RESOLVE RESOURCE UTILISATION BOTTLENECKS IN LARGE SCALE WEB SERVICE

MORPHEUS: TOWARDS AUTOMATED SLOS FOR ENTERPRISE CLUSTERS