SECURING IOT TRAFFIC FROM ENCRYPTION BYPASSING USING UNOBSERVABLE COMMUNICATION

Jacob Baird, Brendan Lane

1. Background

The rapid proliferation of Internet-of-Things (IoT) poses security challenges of the utmost importance. With an enormous number of IoT sensors generating an equally large volume of data, effective means of securing data communications between IoT devices is increasingly necessary. By 2030, the estimated number of IoT sensors worldwide will reach 1 trillion (Marjani et. Al, 2017). Today, technologies such as IPsec, SSL/TLS and DTLS enable secure end-to-end channels in IoT (Bonetto et. al., 2012). However, network traffic analysis can gather and utilize metadata including communication endpoints, message timing and location details, which when combined with a-priori knowledge and processed by machine learning algorithms, can yield such extensive insight that end-to-end encryption is essentially bypassed (Staudmayer, Pohls and Wojcik, 2018). As privacy is recognized as a human right by the European Union, IoT devices operating within the EU must adhere to strict security standards, and such leaking of sensitive information is unacceptable. Therefore, end-to-end encryption is not enough to secure IoT communications and a new approach to securing IoT communications is necessary.

One technique to implement to solve the problem of network traffic analysis extracting sensitive information from metadata is unobservable communication. There has been extensive work published on the topic of unobservable communication in recent years (Staudmayer, Pohls and Wojcik, 2018; Bauer and Staudmayer, 2017; Balaji and Prabha, 2013; Angel and Setty, 2016). Unobservable communication is achieved through implementation of the Dining Cryptographers Net protocol. (Bauer and Staudmayer, 2017). Using this approach, communication is anonymized, and metadata is hidden from attackers.

This paper explores and analyzes the usefulness of the DC-net protocol in ensuring unobservable communication and protecting IoT communications. This exploration is motivated by the importance of security an ever-expanding Internet-of-Things. The DC-net protocol was first proposed by David Chaum over 30 years ago, and while its initial implementation experienced large overheads, it has undergone many changes in the past three decades to better ensure feasibility and effectiveness (Bauer and Staudmayer, 2017).

2. Related Works

One implementation of unobservable communication is the Pung communication system (Angel and Setty, 2016). Pung was developed in response to fears of mass surveillance and ISPs disclosing users' information without their consent. Although this problem is not the same as the one facing IoT, its solution may be applicable to IoT (i.e. hiding users' metadata).

Pung enables users, analogous to IoT devices from the IoT viewpoint, to communicate with each other by storing and retrieving messages in a key-value store without any other parties including Pung servers themselves knowing.

A downside of Pung in the IoT lens is the overhead associated with storing and retrieving messages in a key-value store. The overhead lies not only in administering the key-value stores themselves but also in runtime needed to store and retrieve message in an environment of otherwise high-velocity and high-volume data.

Another implementation of unobservable communication is the UnObservable Secure Proactive Routing Protocol (Balaji and Prabha, 2013). UOSPR was developed to be used in mobile Ad-hoc networks in which moving nodes are connected over lossy channels. Certain IoT networks, such as networks of autonomous vehicles, fit this description.

UOSPR features group signatures which allow one member of a group to sign a message on behalf of its group. This provides the member a certain degree of anonymity as a receiver or any eavesdropping party cannot ascertain from which member the message originated. UOSPR also features ID-based encryption. ID-based encryption involves public key generation based on some publicly known feature of a member's identity. Private key generation is handled by a trusted third party.

A downside of UOSPR is the association of a user's public key with its identity. This would enable another party to determine the sender or intended recipient of a message by associating the public key used to encrypt the message with known users' and their identities.

An implementation of interest to us is achieved via message scheduling (Herraveld, 2012). Using this approach, multiple messages may be passed in a decentralized manner, one at a time, on an agreed-upon schedule.

To begin, every host in the networks decides upon a number of messages to be passed in total, *p*, and the number of messages it would like to send that round itself, *q*. To construct its own schedule, it then initializes a *p*-bit sequence of 0s and flips *q* bits at random. Each host broadcasts its schedule to every other host, where the XOR sum of every schedule is calculated in order to compute a collective schedule. A bit of 0 in the collective schedule indicates either two or no hosts chose this position, and this position is skipped. A bit of 1 indicates that a single host chose this position.

Position	1	2	3	4	5	6	7	8
Host A	0	0	1	1	0	0	0	0
Host B	1	0	0	0	0	0	1	0
Host C	1	0	0	1	0	0	0	0
Collective	0	0	1	0	0	0	1	0

Position	1	2	3	4	5	6	7	8
Host A	0	0	1	1	0	0	0	0
Host B	1	0	0	0	0	0	1	0
Host C	1	0	0	1	0	0	0	0
Collective	0	0	1	0	0	0	1	0

Once each host has a copy of the collective schedule, hosts than calculate the AND of the collective schedule and their individual schedule, which becomes the host's new individual schedule. Starting from bit 1 in this schedule, and skipping all 0s in the collective schedule, a host will send a message only when the bit in their individual schedule is 1. Once a message is verified to have been sent from within the DC-network, each host moves to the next bit in the collective schedule to repeat this process.

A downside of this implementation is the possibility of more than 2 hosts choosing the same position in their individual schedules. If the number of hosts colliding is odd, this conflict will go unnoticed. This requires hosts choosing a large value of *p* in relation to *q*. Another downside is the requirement that only one message be sent within the network at a time. This reduces message throughput and the problem is exacerbated for large numbers of hosts.

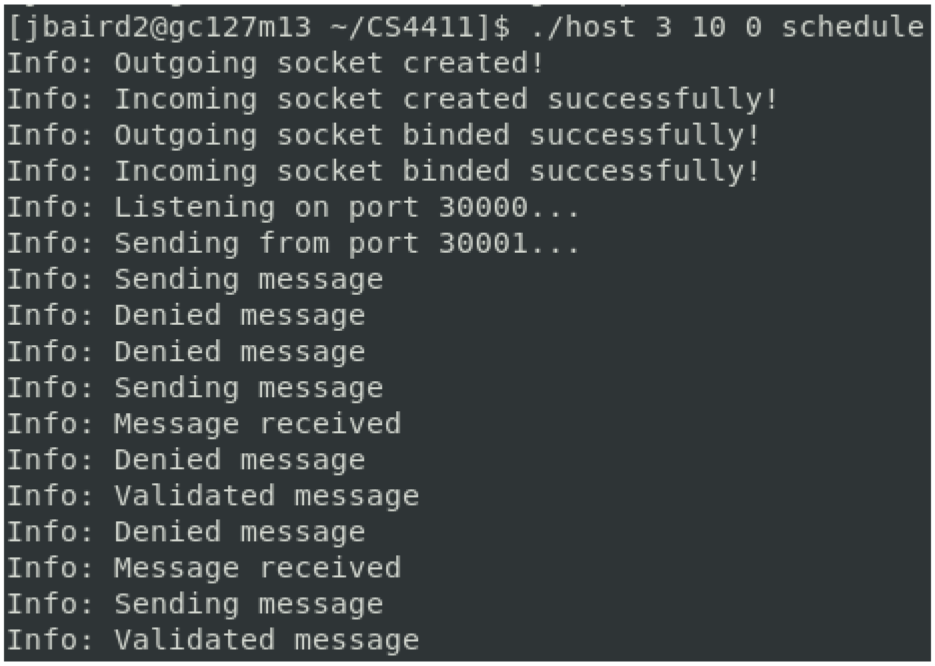
3. Implementation

Due to the shortcomings of the implementation proposed by Herraveld, we propose our own implementation which avoids message collisions and increases throughput at the price of increased memory consumption in otherwise limited IoT devices.

In our implementation, verification is performed by checking the hash of a message with sent message hashes stored locally. I.e., when host *A* sends a message *M* to host *B, A* stored the hash *H* of this message locally. Once *B* receives this message, it broadcasts the hash *H* of this message. Once *A* receives this *H* it will compare it to all hashes stored locally. If it finds a match, it will remove this local copy of this hash and respond with the negation of the XOR of its left and right shared keys concatenated with *H*, otherwise it will respond with the XOR of its left and right shared keys concatenated with *H*. Once *B* receives all responses, it can deduce whether the original sender is trusted.

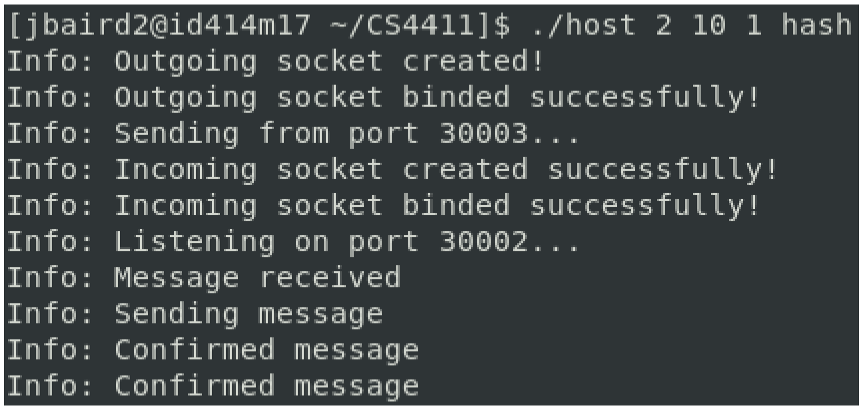
The purpose of hashing messages in this approach is to not divulge the contents of a message while still enabling a host to determine whether it sent the message originally. The benefit of this approach is a greater throughput of messages in the network as multiple messages may travel across the network simultaneously.

Below is a screenshot of the "schedule" protocol in action:



```
[jbaird2@gc127m13 ~/CS4411]$ ./host 3 10 0 schedule
Info: Outgoing socket created!
Info: Incoming socket created successfully!
Info: Outgoing socket binded successfully!
Info: Incoming socket binded successfully!
Info: Listening on port 30000...
Info: Sending from port 30001...
Info: Sending message
Info: Denied message
Info: Denied message
Info: Sending message
Info: Message received
Info: Denied message
Info: Validated message
Info: Denied message
Info: Message received
Info: Sending message
Info: Validated message
```

Below is a screenshot of the "hash" protocol in action:



```
[jbaird2@id414m17 ~/CS4411]$ ./host 2 10 1 hash
Info: Outgoing socket created!
Info: Outgoing socket binded successfully!
Info: Sending from port 30003...
Info: Incoming socket created successfully!
Info: Incoming socket binded successfully!
Info: Listening on port 30002...
Info: Message received
Info: Sending message
Info: Confirmed message
Info: Confirmed message
```

4. Hypothesis

The hypothesis is our implementation of a DC-network will be more runtime performant and allow greater message throughput than the model proposed by Herraveld while requiring more memory. It follows that our implementation will be more appropriate on IoT devices with larger memory which also require a greater message throughput.

5. Experimental Design

Initially, the plan for experimentation was to generate an IoT simulation and implement a DC-net protocol with that simulation. However, further review of available resources has shown that there is little in the way of DC-net protocols available for use with a simulator. In fact, a study published in 2019 lists only two existing DC-net implementations at the time of their work, neither of which was created for use with IoT (Staudmayer, Pohls and Wojcik, 2019). The study further identifies itself as the first and only implementation of a DC-net within an IoT simulation, which further research for this paper deemed to be true.

Because of this, developing an experimental design for this paper has proven difficult. The aforementioned 2019 study, however, states in its conclusions that it found DC-net protocols to be a worthwhile implementation when balancing privacy with overhead, meaning further attempts at implementation could prove worthwhile.

To simulate both implementations of a DC-network, we created an executable file from scratch in C which simulates a single host. This host spawns a child thread to listen to a UDP socket and spawn its own child thread to handle a message once it is received. The message handler will verify the authenticity of a sender or verify whether the host sent a particular message, depending on the nature of the received message.

The executable file takes as input 4 fields. These fields are:

- Number of hosts: The number of hosts that are running in the network
- Number of messages: The number of messages the host must send
- Index: The host's index within the network
- Protocol: The implementation to use (either "schedule" or "hash")

Hosts are called from a shell script.

The following table contains the values used to benchmark each of the protocols:

Number of hosts	Number of messages
4, 8, 16, 32	1, 10, 100, 1000

Each combination of the above variables is evaluated 30 times for both the "schedule" and "hash" protocols.

Runtime is measure by storing the current time in milliseconds, via the C **time.h** library, at the beginning and the end of the execution of the executable file and printing the difference.

Memory consumption is measured by the maximum size of the hash list stored by the "hash" implementation. This number is multiplied by the size of a hash, 64 bytes, and printed at the end of execution. The rationale for this measurement technique lies in the stack consuming the greatest amount of space on the heap. By this approach, the "schedule" protocol will have no memory consumption as it does not store hashes locally. Because the "schedule" protocol is intended for devices with tight memory constraints, this measurement is not problematic.

6. Experimental Result and Discussion

The following graphs contain the results of the benchmarking experiments:

Figure 1: Average runtime of hosts for "schedule" protocol

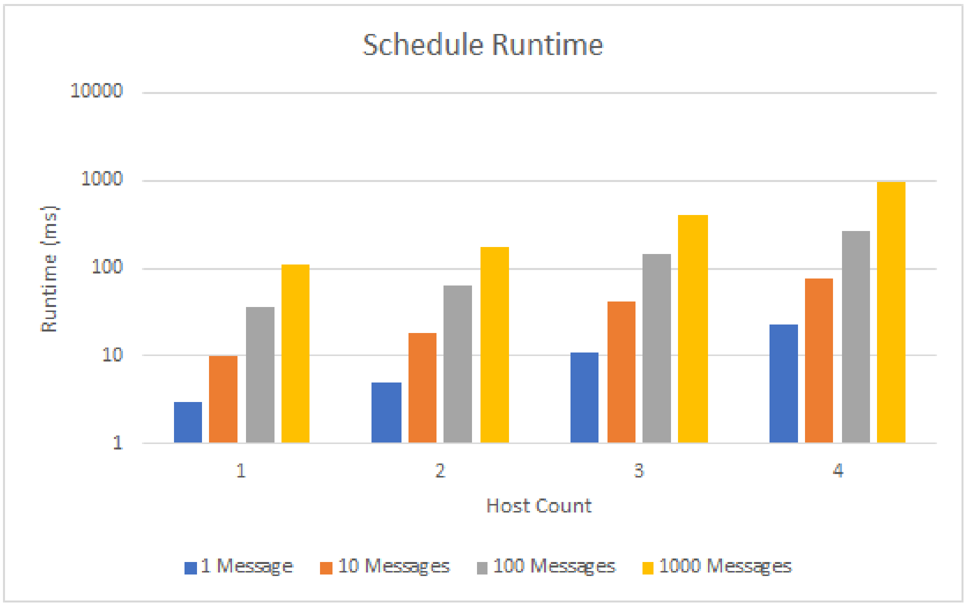


Figure 2: Average runtime of hosts for "hash" protocol

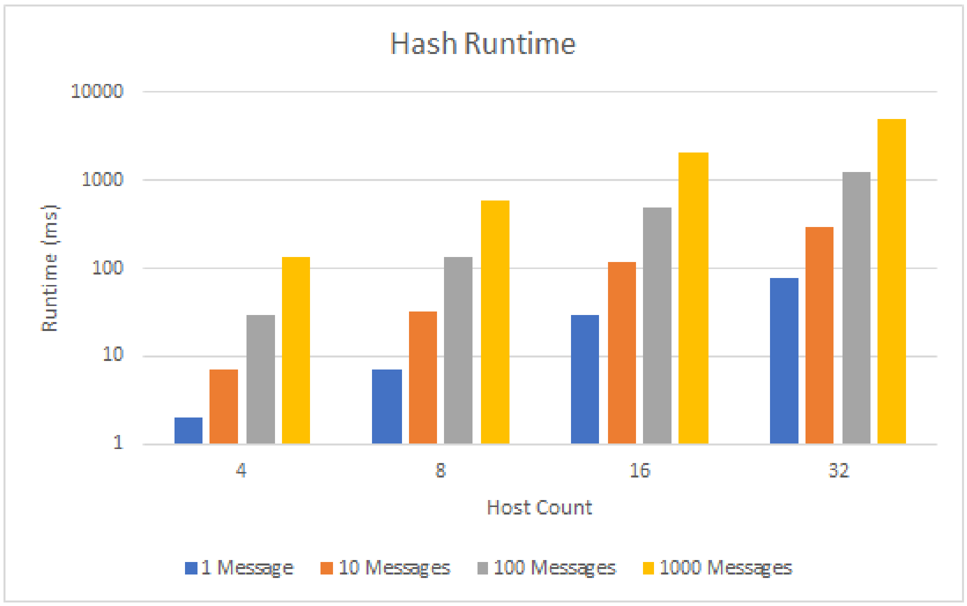
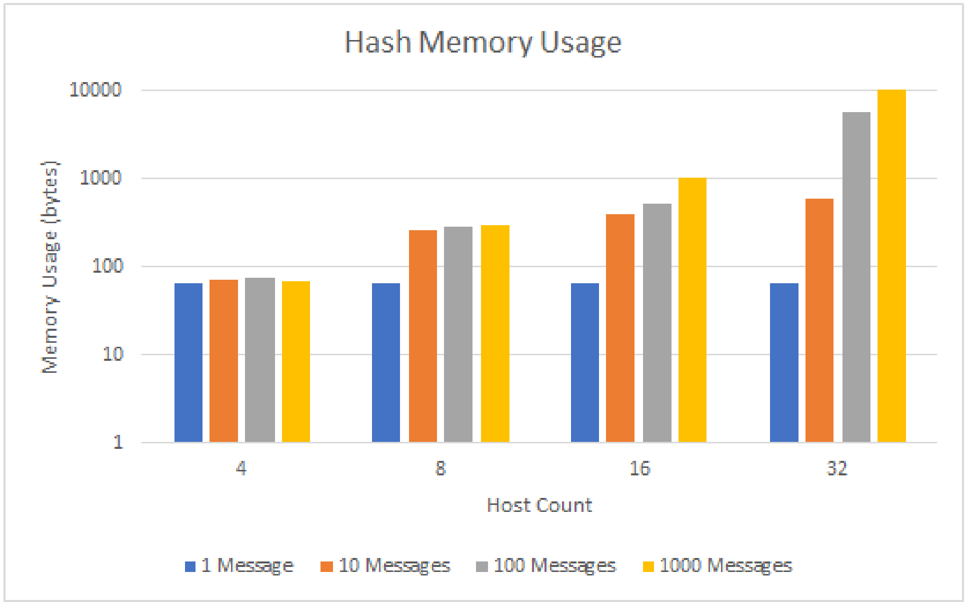


Figure 3: Average memory usage of hosts for “hash” protocol



From the above graphs its can be observed that the existing “scheduling” protocol is more performant for both runtime and memory usage. Because the experiments involved simulating up to 32 hosts on a single core system, the protocol which requires fewer computation is favorable. However, in a setting in which transmission and propagation delays are non-negligible, and on hosts supporting hyperthreading or multiple cores, the benefit of our approach may be more pronounced.

Under no circumstances can we definitively say our “hash” approach is better than the “schedule” approach. Although the prospect of fewer message collisions and multiple messages passing simultaneously, it is simply not feasible by our approach.

We suspect that the overhead of calculating, storing and searching for hashed messages has the greatest affect on our runtime.

Bibliography

Harreveld, T. (2012). *Dining Cryptographer Networks*.

Marjani, M. et. al. (2017). *Big IoT Data Analytics: Architecture, Opportunities, and Open Research Challenges*.

Bonetto, R. et. al. (2012). *Secure communication for smart IoT objects: Protocol stacks, use cases and practical examples.*

Staudmayer, R.C., Pohls, H.C. and Wojcik, M. (2018). *The road to privacy in IoT: beyond encryption and signatures, towards unobservable communication.*

Bauer, J. and Staudmayer, R.C. (2017). *From Dining Cryptographers to dining things: Unobservable communication in the IoT.*

Balaji, S. and Prabha, M. M. (2013). *UOSPR: UnObservable secure proactive routing protocol for fast and secure transmission using B.A.T.M.A.N.*

Angel, S., Setty, S. (2016). *Unobservable Communication over Fully Untrusted Infrastructure*.

Staudmayer, R.C., Pohls, H.C. and Wojcik, M. (2019). *What it takes to boost Internet of Things privacy beyond encryption with unobservable communication: a survey and lessons learned from the first implementation of DC-net.*