Fast Downstream to Many (Computational) RFIDs

Henko Aantjes*, Amjad Y. Majid*, Przemysław Pawełczak*, Jethro Tan*, Aaron Parks† and Joshua R. Smith†
*TU Delft, Mekelweg 4, 2628 CD Delft, NL; h.aantjes@student.tudelft.nl, {a.y.majid, p.pawelczak, j.e.t.tan}@tudelft.nl
†University of Washington, Seattle, WA 98195-2350, USA; anparks@uw.edu, jrs@cs.uw.edu

***Abstract*—We present *Stork*—an extension of the EPC C1G2 protocol allowing streaming of data to multiple Computational Radio Frequency IDentification tags (CRFIDs) simultaneously at up to 20 times faster than the prior state of the art. Stork introduces downstream attributes never before seen in (C)RFIDs: (i) fast feedback for CRFID downstream verification based on the internal EPC C1G2 memory check command—which we analytically and experimentally show to be the best possible downstream verification process based on EPC C1G2; (ii) ability to perform multi-CRFID transfer—which in our experiments speeds up downstream by more than two times compared to sequential transmission; and (iii) the use of compressed data streams—which improves firmware reprogramming times by up to 10% at large reader-to-CRFID distances.**

I. Introduction

Computational Radio Frequency IDentification (CRFID) platforms are embedded batteryless systems with communication, computation and sensing capabilities. Because CRFIDs run solely on harvested energy [1] they are considered to be a favorable replacement for conventional wireless sensors, e.g. in applications that require perennial operation in hard-to-reach/embedded/implanted locations [1], [2]. Examples of CRFIDs include the WISP [3] and Moo [4] platforms. Unique to CRFIDs, in contrast to standard sensor nodes, is the frequent loss of power (on a milliseconds time scale during operation [5, Fig. 1]).

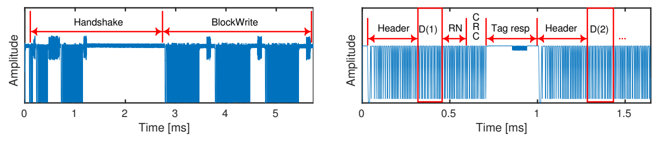
Despite rapid progress in research, many aspects of CRFIDs are still unexplored. For instance, it is hard to communicate robustly between a host (PC, controlling RFID reader) and CRFID—due to frequent power interrupts. Furthermore, key communication features (like broadcast) are not available and host-to-CRFID transmission speeds are slow, in comparison to battery-powered sensor networks. CRFIDs aside, even classical RFID systems have not yet demonstrated fast and robust data transfer and storage protocols.

A. Robust Host to (C)RFID Downstream: Background

This work focuses on *downstream* transmission in CRFID/RFID applications—sending large portions of data from a host to (many) (C)RFIDs. In the context of CRFIDs, a downstream protocol based on EPC C1G2[1] can be used for firmware updates [5], [7], [8].

Supported by the Dutch Technology Foundation STW grant 12491 and in part by US NSF grant CNS 1305072, CNS 1407583 and EEC 1028725, Intel Corporation, and the Google Faculty Research Awards program.

[1]To the best of our knowledge, all CRFIDs use EPC C1G2 protocol [6] for communication. EPC C1G2 supports backscatter, which is orders of magnitude more energy efficient than active transmissions such as those based on IEEE 802.11x [1].



(a) *Wisent* protocol message (b) `BlockWrite` closeup (R1000)

Fig. 1. State of the art (C)RFID downstream protocol (*Wisent* [5]) trace by custom-made EPC C1G2 [6] sniffer. Setup: Impinj R1000 reader (3.2.4.240 firmware), WISP 5.1. We notice large overheads due to: (a) Wisent protocol itself (handshake every EPC C1G2 data frame: `BlockWrite`)—48% overhead, and (b) limitations of Impinj R1000 firmware (unnecessary slicing of `BlockWrite` into parts, marked as $D(x)$; RN: tag handle)—85% overhead. Delay between ≈(1,3) ms in (a), is also due to Impinj firmware.

Wireless reprogramming capability would allow CRFIDs to realize their full battery-less potential in deeply embedded/implanted use cases. Other applications of a robust and fast downstream include delivery of datasets to be used at the CRFID, e.g., feature sets for pattern matching by CRFID cameras [2]. A specific example use case explored herein is the storage of maintenance history in CRFIDs integrated in airplane parts [9]. Despite some prior work exploring downstream communication, e.g. the *Wisent* protocol [5] or R^2 protocol [7], many challenges remain.

B. (C)RFID Downstream: Challenges and Contributions

We introduce three challenges facing CRFID downstream development, and a solution to address them, in a new downstream protocol called **Stork**. We frame this work by posing the following research question: *What are the speed limits of robust downstream communication for multiple (Computational) RFIDs, operating over the EPC C1G2 protocol, and how can we reach them?*

1) Slow Downstream Speeds: Existing (C)RFID downstream protocols are slow. It takes approximately a minute to transfer a ≈5 kB firmware file with Wisent [5]—the fastest protocol to date. This duration, which is an order of magnitude longer than conventional RFID tag access times, would require a procedural overhaul of industry practices in order to be accepted. As a point of reference, updating access rights to contactless door keys requires ≈4 seconds (measured using commercial door access system of [10]).

Contribution 1—**Fastest EPC C1G2 Downstream:** Stork enables immediate acknowledgments for downstream data by leveraging the ability for an EPC C1G2 `Read` command to be issued immediately after a `Write` event, eliminating the need to re-singulate the CRFID in order to get an acknowledgement. Stork demonstrates a 20 times improvement in downstream transfer speeds over prior art by more efficiently utilizing the underlying EPC C1G2 protocol in this way, making it the fastest protocol to date.

2) Lack of Multi-CRFID Capability: EPC C1G2 does not provide a means for sending the same data to many (C)RFID tags simultaneously. Even if it did, there is no strategy for optimizing the transfer acknowledgement process for multi-(C)RFID populations—something that is desirable since each host-to-(C)RFID handshake takes a significant amount of time compared to actual data transmission, see Fig. 1.

Contribution 2—**Low-Overhead Multi-CRFID Transfer:** Stork enables multi-tag transfer through opportunistic CRFID connections and introduces broadcast as an overhearing mechanism, the first (to the best of our knowledge) for CRFIDs. Overhearing allows selected nearby CRFIDs to 'listen in' to transmissions which contain data intended for other tags, such as a common firmware file or configuration data. After transmission, file segments which were not correctly overheard are identified using a new mechanism called *binary memory search*, and the missing pieces are then filled in by the reader. For even small tag populations, Stork's overhearing scheme provides more than a two times improvement in file transfer speeds to multiple tags over simple tag-for-tag transfer.

3) Uncompressed Data Streams: Transmitting data without compression often leads to inefficient utilization of the link. In the context of energy-constrained CRFIDs, there is a tradeoff between communication and computation: Decompressing a compressed data stream at the CRFID, requires more computation but allows for less data to be sent, saving on the energy cost of communication. However, decompressing at the CRFID also introduces costs in terms of energy and time. Furthermore, the transient power availability on a CRFID poses a significant hurdle in doing reliable decompression.

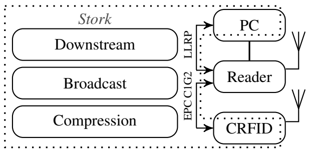
Contribution 3—**Robust Decompression at CRFID:** Stork implements the first downstream (de)compression scheme for CRFIDs, allowing a communication/computation tradeoff to be traversed for the first time. A CRFID-centric Huffman coder reduces the size of any data transferred to the CRFID. Stork addresses the issue of transient power availability by checkpointing [11], [12]. It preserves the state of program execution during mid-computation power outages in a novel *adaptive* way based on the observed energy stability. We demonstrate that transfer speeds benefit from compression when performing longer-range, i.e. low-energy region, transmissions (e.g., 10% improvement at 60 cm distance).

For results replication, source code and measurement data is accessible upon request or via [13]. The video showing Stork in action is accessible upon request or via *goo.gl/xuH9uf*.

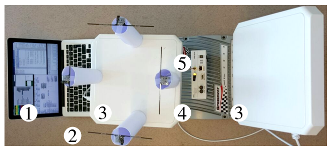
II. (C)RFID Downstream Protocol: Preliminaries

A. System Overview: Architecture, Hardware and Software

1) EPC C1G2 Downstream Architecture: Any UHF RFID system is composed of: (i) a PC or host device, translating data to be stored into a (C)RFID-compliant format and controlling message flow; (ii) an RFID reader, translating PC commands into EPC C1G2 messages; and (iii) a CRFID (or RFID), recording and/or acting on messages received from the reader. The standard bidirectional reader-to-CRFID protocol is EPC C1G2 [6], while the PC-to-reader interface is the Low Level Reader Protocol (LLRP) [14], see Fig. 2(a).



(a) *Stork* and (C)RFID architecture



(b) Experiment setup

Fig. 2. Stork: (a) features mapped to (C)RFID components, described in Section III; (b) experimental setup (refer to Section II-B1): (1) PC running Stork and EPC C1G2 sniffer, (2) four WISP(s) 5.1 on paper stands, (3) reader/sniffer antenna, (4) RFID reader, (5) software defined radio for protocol sniffing.

2) Hardware and Software: The CRFID platform used in this evaluation is the WISP 5.1: an MSP430-based CRFID with 64 kB non-volatile FRAM [3]. The reader used is the Impinj R1000[2] (firmware version 3.2.4.240) transmitting 30 dBm at 915 MHz into an RFMAX S9028PCRJ 8 dBic gain antenna. The reader is controlled by a PC using Ubuntu 14.04; refer to Fig. 2(b). The Python-based sllurp library [15] is extended to enable Stork-specific LLRP requests.

B. EPC C1G2 Sniffer

An EPC C1G2 protocol sniffer is developed which measures timing on a sub-packet level. This is used in assessing Stork performance (result in Fig. 1 is also sniffer-generated).

1) Sniffer Design: The EPC C1G2 sniffer is based on a USRP N210 software defined radio, which is connected to the same antenna type as the RFID reader for downstream transmission (Section II-A2). The PC hosts MATLAB R2016a and GNU Radio version 3.7.8.1 (Fig. 2(b)). Because the standards-compliant reader performs frequency hopping every 400 ms, the entire band over which it hops must be captured in order to recover the baseband signal. Thus, a 50 MS/s rate was selected in order to capture the entire 902–928 MHz band.

2) EPC C1G2 Message Extraction: The raw signal is smoothed with a moving average filter. A near-zero threshold is used to distinguish between transmissions and regions of inactivity. Reliable peaks are selected and used to determine an ASK demodulation threshold for each transmission. Data edge timing is extracted, and the resulting symbols are compared against EPC C1G2 symbols including *Data-0*, *Data-1*, *TRCal*, *RTCal*, and *delimiter* [6, Sec. 6.3.1.2.3]. Symbols are stored and mapped to EPC messages, e.g. `QueryRep` and `Write`. Finally, the sniffer searches for the tag's backscattered signals between reader messages. Once a backscattered signal is detected, the sniffer uses a similar decoding process to extract the timestamps and information[3]. Using this tool, we were

[2]The Impinj R420 is also compatible with the WISP but was excluded from the experiments as the only stable firmware available (4.8.3.240) had a bug disabling `BlockWrite`.

[3]We note that alternative sniffer has been described in [16, App. A], however it followed a less direct path to extract the information which led to significant processing delay [16, A.9].

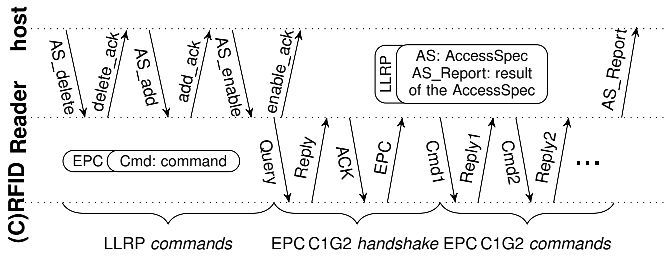


Fig. 3. Host-tag data flow. Alternating EPC C1G2 commands (instructed by LLRP AccessSpec, the one command that encapsulates EPC C1G2, which when issued one after the other enables continuous host-to-reader data streaming)—Cmd1: 'send data to the tag', Cmd2: 'report data reception to the reader', are possible only when Cmd2←Read. For Cmd2←EPC field, a new EPC C1G2 handshake is enforced, limiting downstream speed.

able to sniff EPC C1G2 frames (upstream/downstream) from $\approx$10 meters line-of-sight in an office corridor.

III. Stork: Fast Downstream to many (C)RFIDs

A. Fast and Reliable Downstream to (C)RFID with EPC C1G2

For a (C)RFID downstream protocol to be reliable, all data must be stored correctly in the (C)RFID memory.

Observation 1: In EPC C1G2, only two types of messages could potentially be used as a feedback mechanism for memory verification: (i) the EPC field itself [6, Sec. 6.3.2.1.2] and (ii) the `Read` command [6, Sec. 6.3.2.11.1].

Feedback using the EPC field was used by Wisent [5], a choice presumably inspired by *Flit* [17], which argued that the fastest transfer method for tag to reader communication (*upstream*) was by using the EPC field [17, Sec. 4.1]. The R^2 [7] did not consider any reliability check. Taking on *upstream* challenges, and starting from the above observation, we introduce the following proposition.

Proposition 1: Verifying stored data in (C)RFID memory using the `Read` command is faster than using the EPC field.

Proof: The amount of data the EPC field can hold (|EPC|) is less than what can be accessed with the `Read` command [6, Sec. 6.3.2]. Thus, even if ignoring reader-tag handshake overhead for EPC-based memory verification (see Fig. 3), it would take $m = \lceil X/|\text{EPC}|\rceil$ more messages to verify X bytes of data using the EPC field, compared to using one `Read` command[4]. □

1) Implementation: With Proposition 1 we introduce the components of Stork.

a) Data Representation: Stork uses a new input data format, with Intel HEX as a basis. Prior to transmission, data must be converted into Intel HEX format and mapped to CRFID memory addresses, for instance using TI Code Composer Studio in the case of WISP reprogramming. The Intel HEX format is composed of *address* and *data* fields. Unlike Wisent, in which the Intel HEX file was sourced directly [5, Sec. IV-B], Stork concatenates contiguous data in the HEX file into one transmission, getting around the Intel HEX format's limitation of 16 words per line.

b) Message Format: Stork reuses and adapts the Wisent message format by splitting the payload size field (l in [5, Fig. 6]) into two parts. The first part is a shortened (five bits long) payload size field, providing a maximum payload size of 32 words. This is sufficient since the `BlockWrite` command itself has a maximum length (including header) of 32 words for the Impinj R1000. The remaining three bits of the original payload size field l are used to inform the WISP in which location the `BlockWrite` memory verification CRC should be stored. This allows for up to eight different `BlockWrite` commands to be issued without interspersed `Read`-based memory checks, greatly reducing overhead.

c) Transmitted Frame Acknowledgment: By applying Proposition 1 and executing a `Read` command (i.e. reading the result location) directly after the `BlockWrite`, a message acknowledgment is executed within the same handshake sequence, see again Fig. 3. To make this approach possible we extended the sllurp library [15], amassing to $\approx$180 lines of code in python (excluding anything directly related to Stork) [13], to allow sending of multiple commands after a single handshake, i.e. in a single LLRP AccessSpec. With this extension we define two possible message verification methods: (i) **Secure Progress Method** (SPM): Performing `BlockWrite`/`Read` command pairs, such that data from a `BlockWrite` command is verified by its corresponding `Read` command; and (ii) **Minimize Overhead Method** (MOM): Performing multiple `BlockWrite` commands and one (longer) `Read` command, which checks all the data written at once in the end. Although more data can be sent with MOM in a single AccessSpec, MOM is also more vulnerable to power failures, because each `BlockWrite` will stay unchecked until the end of this operation sequence. Therefore, SPM will be used for the remainder of this work[5].

d) Message Length Throttling: A throttling mechanism for the `BlockWrite` payload is fundamental for combining high speed data transfer with CRFID mobility, as proven in [5, Sec. IV-D3]. Stork leverages throttling, with an improved frame adaptation method. Wisent EX has only a limited set of possible payload sizes, i.e. $P_W = \{1, 2, 3, 4, 6, 8, 16\}$ [5, (2)]. Stork expands this list to $P_W \subseteq P_S = \{1, 2, \ldots, 30\}$. The limit of 30 comes from the `BlockWrite` size limit imposed by the Impinj readers (both R1000 and R420), which is 32 words (2 words are reserved for Stork metadata).

For throttling up/down, a novel timeout mechanism is used, see Algorithm 1. The throttling factor (line 2) should be chosen based on the expected channel quality. The period of the timer (line 10) must be larger than the expected time of an AccessSpec, such that any errors have minimum effect.

Proposition 2: Stork is faster than Wisent [5].

Proof: See Appendix A. □

Following Proposition 2, we present a numerical example in Fig. 4(a). The Stork and Wisent speed decreases slowly

[4]In practice $m = 3$, as $|\text{EPC}| = 12\,\text{B}$ (of which at least two bytes are necessary to identify the CRFID, such that only ten bytes can be used) and $X = 64\,\text{B}$ (the size of `Read` command), which in practice it reduces to 32 B, because the maximum response time of the `Read` command is too short to load more than 32 B (due to WISP 5.1 firmware limitations).

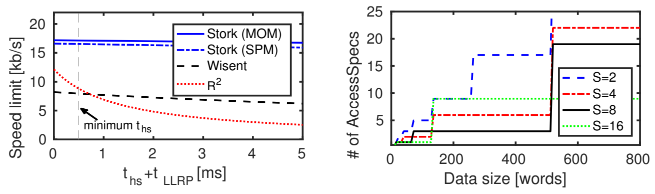
[5]A third method of providing memory verification (not implemented here) watches for timeouts in the tags' reply to `BlockWrite`.

Algorithm 1 Stork payload throttling

```
1: $P_\mathrm{m}$                                  ▷ Message payload (words); SPM method (default)
2: $s$          ▷ Increments/decrements factor of $P_\mathrm{m}$; $s = 1.2$ chosen for all experiments
3: upon receiving an AccessSpec report
4:     if $n_\mathrm{bw} = 1$ then                  ▷ Exact one BlockWrite check successful
5:         $P_\mathrm{m} \leftarrow P_\mathrm{m}$                                      ▷ Do nothing
6:     else if $n_\mathrm{bw} > 1$ then            ▷ Multiple BlockWrite checks successful
7:         $P_\mathrm{m} \leftarrow \lceil P_\mathrm{m} s \rceil$     ▷ Only if $P_\mathrm{m} < 30$; otherwise switch to MOM method
8:     else                                                          ▷ $n_\mathrm{bw} < 1$
9:         $P_\mathrm{m} \leftarrow \lfloor P_\mathrm{m}/s \rfloor$
10: upon timer overflow                               ▷ AccessSpec report timeout
11:     if tag not seen then
12:         reset timer                                  ▷ Wait until tag response
13:     else                                                   ▷ AccessSpec error
14:         $P_\mathrm{m} \leftarrow \lfloor P_\mathrm{m}/s \rfloor$
```

Algorithm 2 Stork Binary memory search (host-executed)

```
1: $\mathsf{R} = \{\mathsf{R}_1\}$  ▷ Memory location blocks; content might differ between (C)RFID and
   host; initially largest memory block
2: $\mathsf{G} = \emptyset$                                              ▷ List of good memory parts
3: $\mathsf{W} = \emptyset$                    ▷ List of wrong memory parts—to be resent by the host
4: while $\mathsf{R} \neq \emptyset$ do
5:     for all $\mathsf{R}_i$ in $\mathsf{R}$ do
6:         $\mathsf{crc_{tag}} \leftarrow$ CRCREQUEST($\mathsf{R}_i$)        ▷ Calculate and send CRC by (C)RFID
7:         $\mathsf{crc_{host}} \leftarrow$ CRC($\mathsf{R}_i$)             ▷ Calculate correct CRC by the reader
8:         $\mathsf{R} \setminus \mathsf{R}_i$                                    ▷ Remove $\mathsf{R}_i$ from $\mathsf{R}$
9:         if $\mathsf{crc_{tag}} = \mathsf{crc_{host}}$ then                        ▷ Tag data correct
10:            $\mathsf{G} \leftarrow \mathsf{R}_i$
11:        else                                                  ▷ Tag data incorrect
12:            if LENGTH($\mathsf{R}_i$) $< R_\mathrm{min}$ then                ▷ $\mathsf{R}_i$ too small to split
13:                $\mathsf{W} \leftarrow \mathsf{R}_i$                                        ▷ Send $\mathsf{R}_i$ as is
14:            else                                   ▷ Request CRCs of smaller blocks
15:                $\mathsf{R} \leftarrow$ SPLIT($\mathsf{R}_i, S$) ▷ Split $\mathsf{R}_i$ in $S$ equal parts, see Section III-B2
```



(a) Protocol-dependent downstream speed limits

(b) Maximum AccessSpec number to check the complete memory

Fig. 4. Numerical examples: (a) the sniffed parameter values of the Impinj R1000 reader, as given in Table II, combined with the protocol parameters of Table III are used in generating this result. The vertical line denotes the minimal observed time of handshake (t_hs). Observe that the benefit of MOM over SPM is marginal; (b) illustration of (1), $C_\mathrm{min} = 8$.

when the overhead of handshake (t_hs) and LLRP access time (t_LLRP) increases, because the combined (on air) packet time (i.e. Nt_msg in (3)) is large compared to $t_\mathrm{hs} + t_\mathrm{LLRP}$. R^2 suffers most from the small combined packet time.

B. Multi-CRFIDs Downstream

To stream data to multiple CRFIDs, we have designed and implemented three transmission schemes for Stork: (i) **Sequential** (baseline), where data is sent to each (C)RFID in turn/successively; (ii) **Opportunistic**, where only one AccessSpec is sent (of maximum size of 210 words, Table III) following the same process as for a single (C)RFID, to the first responding tag in a population—this will continue until all tags have received the intended data; and (iii) **Broadcasting**, described below.

Observation 2: The EPC C1G2 protocol only allows the sending of commands to RFID tags to be sequential [6, Sec. 6.3.2.11], even if the command is the same for all tags.

Proposition 3: A broadcasting scheme can be EPC C1G2-compatible.

Proof: According to the EPC C1G2 specification, a command can be sent to tag i only after a handshake [6, Sec. 6.3.2.11]. The command m_i holds a handle chosen by tag i at the handshake. However, any other tag $j \neq i$ in the vicinity may also able to receive m_i, which enables broadcasting of data to multiple tags. □

1) Broadcast Implementation: We proceed with the details.

a) Broadcast Process: The broadcasting scheme in Stork is divided as follows: (i) set all but one CRFID in overhearing mode; (ii) send all the data to the one CRFID that is not in overhearing mode; (iii) pick one CRFID and turn its overhearing mode *OFF*; (iv) check the memory of this CRFID and send missing data parts; (v) repeat previous two steps until all CRFIDs have received the correct data[6]. The complete implementation of broadcast in Stork amasses to ≈100 lines of Python code [13].

b) Broadcast Enablement in EPC C1G2: We implement a special mode for the CRFID that does not discard messages if the handle (Req_H, Fig. 3) is wrong (amassing to just ten lines of TI MSP430 assembly code [13]): If a CRFID j receives a `BlockWrite` command m_i with $i \neq j$, it will check if it is in overhearing mode. If this is the case, it will execute the command, without replying—a reply would interfere with the reply of the communicating CRFID i. Finally, to switch a tag to/from an overhearing mode Stork sends a specific `Write` command which is decoded by the CRFID as *Overhear ON* or *OFF*.

c) Memory Check: Since the (C)RFIDs listening to the broadcast cannot immediately provide feedback to each `BlockWrite` with the `Read` command (this is impossible with EPC C1G2, see Observation 2), there must be a mechanism to verify if the targets have overheard the broadcast successfully. An intuitive way would be to read the memory from each CRFID after each data fragment was sent to a 'handshaked' CRFID. Although upstream is generally faster than downstream, a *reliable and fast* upstream transfer of a large (C)RFID memory section is still an unsolved challenge [17, Sec. 8] [2, Fig. 14]. Therefore, we propose a new low-overhead (C)RFID memory check, the **binary memory search**, inspired by [18] and outlined in Algorithm 2. The idea behind the algorithm is to "zoom in" to increasingly small memory segments to isolate a corrupted block, by comparing the CRC-16 (default CRC implementation in WISP 5.1) of the sent data independently by the reader (which has a correct copy of the data) and the (C)RFID. This is needed, as starting with small memory blocks will increase the number of CRC checks needed, and thus the overall transmission time. The binary memory search algorithm implementation amasses to 300 lines of Python code [13].

To request a CRC, the `BlockWrite` command is used. The

[6]The proposed method could also be used for automatically sending smaller, differential firmware updates.

request message contains the address of the starting location (from which the CRC must be calculated), length (amount of words that need to be verified) and a parity check. A `Read` command must follow to receive the calculated CRC, because the `BlockWrite` command does not allow a custom reply. Since the maximum number of commands per AccessSpec is eight, a single AccessSpec can hold up to four memory check requests and four replies. Moreover, since the `Read` command has capacity for eight CRCs (for WISP 5.1), we let the CRFID compute by default eight CRCs of sequential memory parts per request. Therefore, up to 32 memory sections can be checked in one AccessSpec if those 32 sections can be grouped into four contiguous memory parts.

2) Speed versus Memory Splitting Factor S: Enabling overhearing comes at a price: the need to change tags to/from overhearing mode and perform memory checks. The number of these operations is determined by the memory splitting factor S (line 15 of Algorithm 2).

Proposition 4: The maximum number of AccessSpecs it takes to check all the memory for various S is

$$\sum_{n=0}^{\lceil \log_S(DC_{\text{min}}^{-1}) \rceil} \left\lceil \frac{S^n}{\min(4, 32/S)} \right\rceil, \text{ for } D \leq C_{\text{max}}, \quad (1)$$

where D is the data size (in words) that need to be checked, C_{min} is the minimum data length for which requesting a CRC is faster than sending the data itself, and C_{max} is the maximum number of words that the CRC can check at once[7].

Proof: If $D < C_{\text{min}}$, no CRC has to be requested. If $D < SC_{\text{min}}$, only one CRC will be requested, because splitting the data in S parts would make the parts smaller than C_{min}. This CRC request will be placed in a single AccessSpec. If $D \geq SC_{\text{min}}$, the first CRC will be requested, and will be split into S parts in the worst case, which will be placed in $\left\lceil \frac{S}{\min(4,32/S)} \right\rceil$ AccessSpecs. The factor $\min(4, 32/S)$ is determined by the AccessSpec limitation, i.e. it can hold up to 32 CRC requests of at most four different memory blocks. The process of this memory splitting halts just before a part would become too small, i.e. smaller than C_{min}. □

Using Proposition 1 we now compute the optimal S. Since the CRC can only be computed over data parts with length a power of two, so S must also be a power of two. Second, we set $C_{\text{min}} = 8$, since the time it takes to send eight words is comparable with the time it takes to request a CRC of eight words. We have observed that the CRFID was not capable of calculating CRCs larger than 512 words in low energy environments, so $C_{\text{max}} = 512$. Inputting these parameters we get $S = 8$ as the best value for the largest range of data lengths, as shown in Fig. 4(b).

C. Compression for CRFID Downstream

We proceed with the final component of Stork—data/firmware compression for CRFIDs.

[7]Due to either the max response time of the CRFID or to available energy.

Algorithm 3 Stork Decompression on CRFID

```
1: V    ▷ Set of variables used in the decompression—kept in volatile memory
2: $F_c, F_1 = F_2 = V$    ▷ $F_c$: consistency flag; $F_x$: copies of V—kept in non-volatile memory
3: function DECOMPRESS()    ▷ Called by MAIN() after every power failure
4:     ENSURECONSISTENCY()
5:     $S_{cr} \leftarrow 0$    ▷ The size of the checkpoint region
6:     $c \leftarrow 0$    ▷ c (counter) and $S_{cr}$ will be reset on power interrupt
7:     while LEN(data compressed) $> 0$ do
8:         if $l_{\text{buf}} \leq l_{\text{bits}}$ then    ▷ $l_{\text{bits}} = 8$ for byte
9:             buffer $\leftarrow$ READBYTEFROMDATA()    ▷ Load one byte into the buffer
10:        $l_{\text{bits}} \leftarrow$ DECODE(buffer)    ▷ Decompression function call
11:        SHIFTBUF(buffer, $l_{\text{bits}}$)    ▷ Shift buffer by $l_{\text{bits}}$
12:        $c \leftarrow c + 1$    ▷ Dynamic checkpoint step indicator
13:        if $c > S_{cr}$ then
14:            SAFELYUPDATESTATE()
15:            $c \leftarrow 0$, $S_{cr} \leftarrow S_{cr} + 1$    ▷ Dynamic checkpoint step increase
16: function ENSURECONSISTENCY()
17:     if $F_c = S_1$ then    ▷ Power loss during $S_1$
18:         $F_1 \leftarrow F_2$    ▷ Keep $F_1, F_2$ consistent
19:     else if $F_c = S_2$ then    ▷ Power loss during $S_2$
20:         $F_2 \leftarrow F_1$    ▷ Keep $F_1, F_2$ consistent
21:     $V \leftarrow F_1$    ▷ No power fail, initialize V during update
22: function SAFELYUPDATESTATE()
23:     $F_c \leftarrow S_1$    ▷ Program is state $S_1$
24:     $F_1 \leftarrow V$    ▷ Save operational variables in non-volatile memory
25:     $F_c \leftarrow S_2$    ▷ Program is state $S_2$
26:     $F_2 \leftarrow V$    ▷ Repeat variable copy for data consistency check
27:     $F_c \leftarrow$ null    ▷ State already consistent after power loss
```

1) Compression/Transmission in CRFIDs: We begin with the observation that, for CRFIDs, using a compressed data stream will always eventually (with increased frequency of energy interrupts) result in faster end-to-end communication, due to the lesser impact of energy interrupts on decompression operations as compared with transmission. To demonstrate this we use the following example. Let F be the size of a file to be sent/compressed (in bits), $c_r \in (0, 1)$ the file reduction ratio (through compression), s_t the message size into which the file must be split (i.e., the length of one `BlockWrite`), and s_d the number of data units decompressed in one decompression iteration (e.g. one nibble in our implementation). Hence, the expected number of transmission trials to complete is $N_t = Fs_t^{-1}p_t^{-1}$ and the expected number of decompression trials to complete is $N_d = Fs_d^{-1}p_d^{-1}$, where $p_x = (1 - p_e)^{l_x}$ is the respective probability of a successful unit transmission ($x = t$) and decompression ($x = d$) respectively, and l_t is the time needed to transmit a unit message, l_d is the time needed to decompress one unit of data, and p_e is the probability of power interruption per unit time (independently and uniformly distributed). Showing that sending data uncompressed is slower than sending it compressed (and decompressing it at the CRFID) is equal to showing

$$N_t l_t > N_t l_t c_r + N_d l_d \Rightarrow \frac{s_d(1 - c_r)}{(1 - p_e)^{l_t}} > \frac{s_t}{(1 - p_e)^{l_d}}. \quad (2)$$

Inequality (2) will hold if $l_t \gg l_d$, i.e. when the length of the atomic data transfer operation is larger than atomic decompression (irrespective of the numerator values in (2)), and large p_e. Using (2), we shall proceed by introducing a proof-of-concept file decompressor tailored for CRFIDs.

2) Data Compression: Stork utilizes the (lossless) Huffman algorithm to compress downstream data. The host PC converts a binary file to be transmitted into an Intel HEX file, appending a coding table at the end (e.g. see [13]). This file is then sent to CRFID tag(s) via Stork. Compression required ≈450 lines of C code and ≈250 lines of Python code on the host side.

3) Data Decompression across Power Failures: For CRFID reprogramming, the transmitted firmware image must be decompressed by the CRFID itself. In Stork, this involves a table lookup of Huffman codes in the transmitted firmware. The key challenge is to keep program state consistent in the face of power outages [12], [19]. To provide this consistency we use *checkpointing* in Stork. Prior approaches followed one of two strategies: (i) rely on external hardware to dynamically place a checkpoint and to save the program state, e.g. [11] or (ii) use of software-based checkpointing, e.g. [19].

In this work we propose a new solution called **dynamic step checkpointing** (DSC), see Algorithm 3, which combines the advantages of both approaches: (i) it relies only on software (necessary as WISP 5.1 does not have external low-power hardware for energy availability measurements, like [20]); that (ii) adapts the location of checkpoints *during* the program execution to minimize overhead of checkpointing.

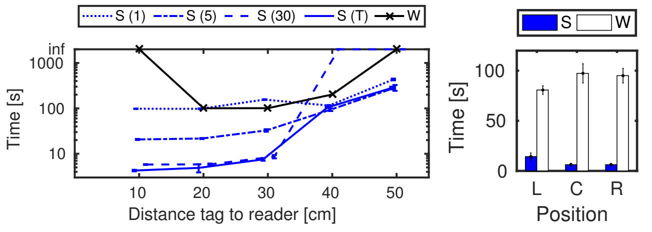
DSC is based on two ideas. First, assuming that the CRFID will keep its state, with each successful consistency check within the execution loop, the algorithm will increase the checkpointing interval by one (see line 13 of Algorithm 3). Second, to insure consistency between states, DSC will exploit the double buffering principle. The program state is saved in two different locations in non-volatile memory (see line 22) to allow restoration of the data reliably after a power failure (line 16). This simple mechanism does not require storage of *all* registers in non-volatile memory, in contrast to earlier approaches [11], [19]. Nonetheless, further in-depth investigation of DSC with other programs, compared against, e.g. [11], [19], is required. Our implementation of Huffman coding-based decompressor amasses to 280 lines of C code.

IV. Stork: Experimental Results

To demonstrate all contributions (listed in Section I-B) we proceed with the evaluation of Stork in single and multi-CRFID scenarios. Tests with commercial RFID tags were not performed, as commercial tags are not capable of firmware modifications, thus cannot be made to run the Stork protocol.

A. Host to CRFID Downstream—Single CRFID

The first set of experiments characterizes the speed of transmission as a function of distance. One WISP is positioned at distances $d = \{10, 20, \ldots, 50\}$ cm from the reader antenna, in a setup similar to Fig. 2(b). At every position we send a complete 5054 B WISP 5.1 base firmware. The results are presented in Fig. 5(a). Each experiment was repeated five times. With Stork we can wirelessly reprogram the WISP 5.1 in under ten seconds at a distance up to 30 cm, which is *ten times faster* than Wisent (and more than 20 times faster within $d = \{10, 20\}$ cm distance). Furthermore, at a distance of $d = 50$ cm, Stork is able to reprogram the WISP, while Wisent was unable to complete the transfer.



(a) Effect of frame size (b) Effect of location

Fig. 5. Single CRFID: (a) transmission of WISP 5.1 base firmware to WISP at different distances and various `BlockWrite` lengths compared against throttling (min P_W=2 was used). Wisent [5] (version EX, throttling enabled), is added for comparison; (b) impact of location differences of WISP against the reader antenna for fixed tag to reader antenna distance $d = 20$ cm, with the WISP antenna parallel to the left edge (L), right edge (R) of the reader antenna and its center (C); Legend—S: Stork, W: Wisent, T: throttle.

We have also measured Stork's performance across CRFID position changes, see Fig. 5(b), for tag to reader antenna distance of $d = 20$ cm. Despite the location-dependent channel variations, Stork consistently shows a tenfold improvement in speed over Wisent.

We do not directly compare Stork to R^2 [7]: applying the transmission speed result from [7, Sec. V] to Fig. 5(a) is not possible for two reasons. First, the experimental setup is different from ours (indoor versus outdoor). Second, R^2 does not consider memory write verification [7, Sec. III-B].

B. Reader to CRFID Transmission—Many CRFIDs

a) Multi-CRFID Transmission Method Comparison: The second set of experiments characterizes the speed of firmware transfer to up to four tags simultaneously[8] (the same firmware as used in Section IV-A), using (i) broadcasting, (ii) opportunistic, and (iii) sequential transfer schemes. All four WISPs are positioned each at fixed distances from the reader antenna, exactly as shown in Fig. 2(b). The result is presented in Fig. 6(a). Each experiment was repeated five times.

When multiple WISPs need to receive the same data, our broadcast mechanism can reduce the transfer time by at least two times at $d = \{10, 20\}$ cm compared to sequential transmission (51.2 and 112.8 seconds sequential versus 17.0 and 51.2 seconds broadcast, respectively). As the distance from the reader antenna to the CRFIDs increases, the benefit of using a broadcasting scheme diminishes, making the opportunistic transmission at $d = 40$ cm (106.7 versus 76.7 seconds) the most efficient scheme of the three. This can be explained by the increase of overhead required for memory verification when broadcasting. This observation calls for an adaptive selection of broadcasting versus opportunistic transmission in future CRFID downstream protocols. We also observe a decrease in multi-tag transmission at short distances, which we verified to be caused by a WISP demodulator power overload. We observe that this effect is not connected to the implementation of Stork, however affects all WISPs.

[8]Our experimental setup consisted of at most four tags, with this limit being imposed by practical RF propagation constraints and the high incident power needed by the WISP to maintain operation.

TABLE I
HUFFMAN COMPRESSION ALGORITHM IMPLEMENTATION AGAINST GZIP

File type	Size (B)	This work (%)	Gzip (%)
Example WISP firmware	7.07 k	10.8	35.7
Example ASCII file	1.9 M	15.6	53.9

b) Scalability of Multi-CRFID Transmission: For the same setup as previously we measured the transmission speed as a function of number of CRFIDs to transmit the data to. The number of CRFIDs in vicinity of the reader antenna is fixed, i.e. four, so if data is transmitted to x CRFIDs, then $4-x$ are still in vicinity of the reader, with the distance from CRFID to the reader also fixed ($d=20$ cm). The result is presented in Fig. 6(b). Each experiment was repeated five times. Transfer speed for all methods scales linearly with the number of tags to transmit. The benefit of using broadcasting increases with increasing number of CRFIDs, provided all CRFIDs are well positioned against the reader antenna.

C. Reader to CRFID Transmission—Firmware Compression

1) Checkpointing Overhead: To measure the benefit of using compression for downstream we will first measure the overhead of checkpointing during decompression. Thus we power the WISP directly to prevent power interrupts. The measured time needed to decompress the same WISP firmware used in the earlier experiments without checkpointing was 1.50 s—comparable to a battery-based embedded system decompression time [21, Fig. 5]. The same experiment was then repeated using checkpoints (positioned statically in line 12 of Algorithm 3) which increased the time by ≈20% (1.79 s). To measure the benefit of dynamic checkpointing, we also performed the experiment with the WISP moving back and forth from the antenna using the Gondola open source indoor mobility testbed [22], at distances $d=\{(10,\ldots,60\}$ cm (to induce repeatable varying channel conditions). Each experiment was repeated 20 times. Dynamic checkpointing reduced the minimum decompression time, when power is uninterrupted, to 1.55 sec. The maximum execution time in the experiment was reduced from 3.85 to 3.0 seconds.

2) Decompression/Transmission Time Trade-off: Second, we measured the overall benefit of compression for CRFID reprogramming, using the same setup as in Section IV-A. The results are given in Fig. 6(c). As shown in Section III-C1 the decompression reduced the overall reprogramming time by 10% at the largest reader to CRFID distance ($d=60$ cm), while it increased transmission speed at $d\leq 30$ cm by 1.65 s (30 %), Fig. 6(c). However, looking at the relative compression gain, Fig. 6(d), we see an improvement of almost 30 s at $d>50$ cm. Comparing our implementation to a benchmark, Table I, we demonstrate reasonable compression levels, although three times lower than Gzip. This shows the potential transmission savings of up to 50 % if more powerful compression methods were implemented on CRFID.

D. Stork Energy Consumption

To evaluate the energy cost of downstream transmission, we compare Stork to Wisent. Stork needs only 7.78 mJ to send 5054 B (1.54 μJ/B) versus 81.70 mJ for Wisent to send 5387 B [5, Sec. V-C2] (15.2 μJ/B). The measurement was performed with the open source measurement circuit provided by [23, Sec. 3A-2c]. Considering decompression of Stork, the energy consumption of the transmission of the compressed file decreased to 6.92 mJ, while the decompression uses 2.65 mJ.

E. Case Study: Aircraft Part Maintenance History Storage

In the final experiment, we demonstrate how Stork can enable storage of maintenance history of individual airplane parts [9], [24]. In case 1, the WISP was attached to the outermost rib of a VFW-Fokker 614 wing, separated by a 7 mm thin expanded polystyrene (to enable reception in the presence of a metal backdrop), see Fig. 7(a). In case 2, it was attached to the inner surface of the rudder, (creating a non-line of sight path)[9], see Fig. 7(b).

In both cases we have transferred the EASA 2015-0133 Airworthiness Directive, converted (prior to the transmission) to text format. The transmitted file size was 9.4 kB in Intel HEX format. The rest of the CRFID ecosystem was the same as in other experiments (refer to Section II-A). In both cases, the reader antenna was handheld, see Fig. 7, and kept at a distance of ≈40 cm and ≈20 cm from the surface of the wing for case 1 and 2, respectively. Successful file transfer times were 17.7 and 20 s for case 1 and case 2, respectively, showing the feasibility of using Stork in managing airplane part history.

V. RELATED WORK: FAST (C)RFID TRANSFER

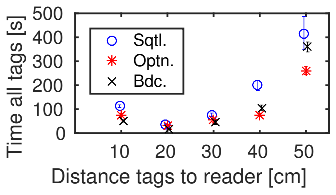
EPC C1G2-based (C)RFID Downstream Analysis: To our knowledge there are no studies that analyze the performance of the EPC C1G2 downstream. Analyses of the EPC C1G2 protocol overwhelmingly focus only on the upstream (tag to reader) transmission. Modern commercial tags used in aviation allow downstream of only a single message, fitting within one `BlockWrite`, e.g. [25, Sec. 7.3.13]. We identified no studies reporting actual speeds of streaming large portions of data (>1 kB) to commercial RFID tags.

(C)RFID-based Data Storage: We are aware of only one patent that proposes to use RFID tags as a means for data storage [26]. No experimental studies of such systems appear to have been performed so far.

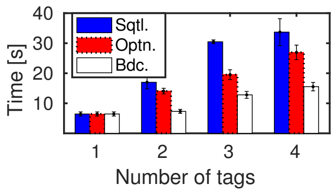
CRFID Wireless Reprogramming: FirmSwitch [8] was the first work demonstrating remote firmware execution, but without actual firmware transfer. Wisent [5] was the first protocol to demonstrate complete wireless reprogramming on top of EPC C1G2, followed by R^2 [7].

Program Construction for CRFID: There are a growing number of papers discussing CRFID programming methods that guarantee execution integrity despite power interrupts. The majority, as exemplified by Mementos [11] or Chime [12] consider hardware support (using external voltage measurement circuit). The only purely software-based system for

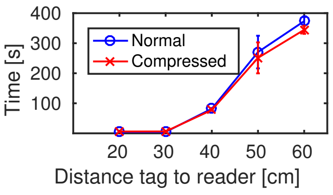
[9]We acknowledge that WISP 5.1 is not designed to withstand harsh airborne environment. Making it robust to large temperature and pressure variations is beyond the scope of this work. An example of an EPC C1G2-compliant RFID targeting aerospace industry is described in [25].



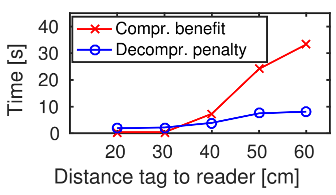
(a) Multi-CRFID transmission



(b) Multi-CRFID scalability

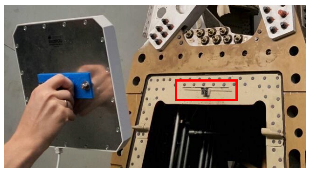


(c) Firmware compression

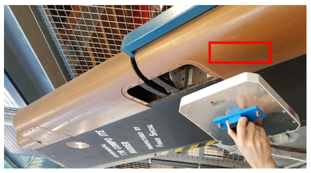


(d) Compression tradeoff

Fig. 6. Stork performance: (a) sending WISP 5.1 base firmware to four CRFIDs (positioned as in Fig. 2(b)) at a time. Broadcasting (Bdc.) improves the transmission performance compared to sequential transfer (Sqtl.) by almost 100 seconds at 50 cm distance; (b) scalability of broadcasting against opportunistic (Optn.) and sequential at 20 cm; (c) complete CRFID reprogramming times using uncompressed and compressed WISP 5.1 firmware. At longer distances the benefit of compression increases; (d) benefit (sending less data) and penalty (computation) of compression for the same experiment as in (c).



(a) VFW-Fokker 614 wing



(b) Fokker composite T28 rudder

Fig. 7. Experiment setup demonstrating transmission to (C)RFID, attached to the aircraft part, from the host. WISP 5.1 is adhesively attached to: (left) metallic, line of sight, and (right) composite, non-line of sight, aircraft part. Attachment locations marked with red rectangle. Using Stork we demonstrated transmission of 4.7 kB file in 17.7 s and 20 s for left and right scenario, respectively. Refer to Section IV-E for details.

CRFID checkpointing is DINO [19]. DINO, however, does not consider checkpoint step adaptivity, which has been introduced in Chime [12, Sec. 6.3], but the step adaptivity of Chime is only supported by dedicated hardware.

VI. Stork: Limitations and Improvements

Better Frame Adaptation: We speculate that Stork would be further improved by throttling down to `Write` message (atomic message of R^2) once shortest `BlockWrite` is reached in harsh channel conditions.

Adaptive Multi-Tag Transmission: We noted that in the multi-CRFID downstream there is no technique that minimizes transmission time for all scenarios. It would be then desired to connect proposed methods into an adaptive system.

Bi-directional Fast/Robust CRFID Transfer: Stork focused on downstream only. Thus, further research is needed on reaching the limits of bi-directional CRFID communication. A first possible approach is to integrate Stork with Flit [17] (for upstream).

Better Compression Mechanism: Stork used a simple compression mechanism based on Huffman coding to traverse the computation/communication tradeoff inherent in (C)RFID systems. This is only a first step—many other compression methods are available and should be explored in this context.

VII. Conclusions

We have introduced and characterized Stork, the fastest protocol for downstream transmission to multiple (Computational) RFIDs. Stork is based on three novel contributions: (i) an order of magnitude speedup of (C)RFID downstream transfer compared to all state-of-the-art experimental and industry protocols (20 times improvement compared against the best protocol we found), brought about by immediate feedback mechanism for data verification; (ii) implementation of an EPC C1G2-complaint broadcast mechanism which improves sequential transfer by two times, and (iii) a compression mechanism enabling faster CRFID reprogramming speeds, by 10%, supported by dynamic step checkpointing, a novel programming primitive for Transiently Powered Computers.

References

[1] S. Gollakota, *et al.*, "The emergence of RF-powered computing," *IEEE Computer*, vol. 47, no. 1, 2014.
[2] S. Naderiparizi, *et al.*, "WISPCam: A battery-free RFID camera," in *Proc. IEEE RFID*, 2015.
[3] (2016) WISP 5.0 Wiki. [Online]. Available: http://wisp5.wikispaces.com
[4] (2016) UMich Moo. [Online]. Available: http://spqr.cs.umass.edu/moo
[5] J. Tan, *et al.*, "Wisent: Robust downstream communication and storage for computational RFIDs," in *Proc. IEEE INFOCOM*, 2016.
[6] (2016) EPC radio-frequency identity protocols generation-2 UHF RFID. Version 2.0.1. [Online]. Available: http://www.gs1.org/sites/default/files/docs/epc/Gen2_Protocol_Standard.pdf
[7] D. Wu, *et al.*, "R2: Over-the-air reprogramming on computational RFIDs," in *Proc. IEEE RFID*, 2016.
[8] W. Yang, *et al.*, "Wireless firmware execution control in computational RFID systems," in *Proc. IEEE RFID*, 2015.
[9] (2016) Draft advisory circular provides guidance on RFID tags. [Online]. Available: http://www.ainonline.com/aviation-news/business-aviation/2016-01-12/draft-advisory-circular-provides-guidance-rfid-tags
[10] (2016) SALTO Systems BV peripheral products website. [Online]. Available: http://www.saltosystems.com/en/product-range/product/97/xs4-wall-readers-and-xs4-door-controllers/
[11] B. Ransford, *et al.*, "Mementos: System support for long-running computation on RFID-scale devices," in *Proc. ACM ASPLOS*, 2011.
[12] A. Mirhoseini, *et al.*, "Chime: Checkpointing long computations on intermittently energized IoT devices," *IEEE Trans. Mobile Comput.*, 2016, accepted for publication.
[13] (2017) Experiments source code. [Online]. Available: http://www.es.ewi.tudelft.nl/papers/2017-Aantjes-INFOCOM_source_code.zip
[14] (2016) EPCglobal low level reader protocol. Version 1.1. [Online]. Available: http://www.gs1.org/sites/default/files/docs/epc/llrp_1_1-standard-20101013.pdf
[15] B. Ransford. (2015) LLRP library controller. [Online]. Available: https://github.com/ransford/sllurp
[16] E. W. Fuentes, "RFIDIoT: RFID as the data link layer for the internet of things," Master's thesis, MIT, 2015.
[17] J. Gummeson, *et al.*, "Flit: A bulk transmission protocol for RFID-scale sensors," in *Proc. ACM MobiSys*, 2012.
[18] J. Redford, "Data integrity checking," Patent US 7 415 654, 2008. [Online]. Available: https://www.google.com/patents/US7415654
[19] B. Lucia and B. Ransford, "A simpler, safer programming and execution model for intermittent systems," in *Proc. ACM PLDI*, 2015.
[20] S. Naderiparizi, *et al.*, "μmonitor: In-situ energy monitoring with microwatt power consumption," in *Proc. IEEE RFID*, 2016.
[21] N. Tsiftes, *et al.*, "Efficient sensor network reprogramming through compression of executable modules," in *Proc. IEEE SECON*, 2008.

[22] M. Cattani and I. Protonotarios, "Demo abstract: Gondola—a parametric robot infrastructure for repeatable mobile experiments," in *Proc. ACM SenSys*, 2016.

[23] I. in 't Veen, *et al.*, "BLISP: Enhancing backscatter radio with active radio for computational RFIDs," in *Proc. IEEE RFID*, 2016.

[24] B. Nath, *et al.*, "RFID technology and applications," *IEEE Pervas. Comput.*, vol. 5, no. 4, Jan.–Mar. 2006.

[25] MAINtag. (2016) FLYchip 64 kbits part number 12368 specification. [Online]. Available: http://www.maintag.fr/fichiers/pdf/flychip/flychip-64kbit-user-guide-12386-20140129.pdf

[26] J. Holland, *et al.*, "System for using RFID tags as data storage devices," Patent US 20 060 279 412, 2005. [Online]. Available: https://www.google.com/patents/US20060279412

APPENDIX

Proof: We define the speed of a EPC C1G2-based downstream protocol x as the number of bits in a single LLRP AccessSpec divided by the time it takes to send them, i.e.

$$L_x \triangleq b_{\text{da}} N (N t_{\text{msg}} + t_{\text{hs}} + t_{\text{LLRP}} + t_{\text{RA}} + t_{\text{meta}} + t_{\text{check}})^{-1}, \quad (3)$$

where b_{da} is the number of bits per data message in the AccessSpec, N number of such data messages, t_{hs} is the time of setting up a connection, t_{LLRP} is the LLRP protocol overhead, t_{RA} is the random access overhead, t_{meta} is the overhead of sending meta data for the data bits, t_{msg} is the time of sending a data message (including a default acknowledgement), and t_{check} is the time of checking if the data is stored correctly. To compare any existing or future downstream protocol, and specifically, to compare Stork over Wisent [5] one needs to derive individual variables in (3) for each protocol. For completeness, in the analysis we also consider R^2 protocol [7], despite R^2 lacks data verification.

Assumptions: To simplify derivation we assume: (i) a single tag environment, i.e. $t_{\text{RA}} = 0$; (ii) no errors, i.e. no message resends present; (iii) connection setup time, random access overhead and LLRP overhead are downstream protocol independent; and (iv) the data to be send is large, such that the border effects, e.g. the very last message might be small creating an AccessSpec with large overhead, are neglected.

Message Overhead—t_{msg}: The EPC messages that are designed for downstream data transfer, `BlockWrite` and `Write` [6, Sec. 6.3.2.12.3], add b_{ov} overhead bits on each b_{da} data bits. The data in EPC C1G2 is transferred using Pulse Interval Encoding for which the symbol times of the *Data-0* (t_0) and *Data-1* (t_1), respectively, are not equal. Each message needs to be acknowledged by the tag with a b_{ack} bits message. For this message FM0 baseband encoding is used [6, Sec. 6.3.1.3], for which the symbol time $t_{\text{a}} = \frac{t_{\text{TR}}}{c_{\text{dr}}}$, where t_{TR} is a calibration value and c_{dr} a divide ratio [6, Sec. 6.3.1.2.8]. Each message also has (i) a frame sync as a preamble which defines t_0 and t_1 with a fixed time delimiter t_{deli}, i.e. $t_{\text{FS}} = t_{\text{deli}} + 2t_0 + t_1$ (ii) a preamble for the acknowledging message as a function of t_{a} and a constant c_{p}, $t_{\text{pr}} = c_{\text{p}} t_{\text{a}}$, (iii) a minimum time after a message and after the acknowledgement, $t^{\text{a}}_{\text{msg}}$ and $t^{\text{a}}_{\text{ack}}$, respectively. Thus, denoting r as zeros/ones ratio,

$$t_{\text{msg}} \triangleq t_{\text{FS}} + (rt_0 + (1-r)t_1)(b_{\text{ov}} + b_{\text{da}}) + t^{\text{a}}_{\text{msg}} + t_{\text{pr}} + t_{\text{a}} b_{\text{ack}} + t^{\text{a}}_{\text{ack}} \triangleq t_{\text{m}}(\{b_{\text{ov}}, b_{\text{da}}, b_{\text{ack}}, r, t^{\text{a}}_{\text{msg}}, t^{\text{a}}_{\text{ack}}\}). \quad (4)$$

TABLE II
LIMITS OF EPC C1G2 [6] VERSUS IMPINJ R1000 IMPLEMENTATION

Sym.	**Unit**	**Min**	**Max**	**R1000***	**SD** (σ)*
b_{ack}	bits	17	17	17	—
c_{dr}	—	8	$64/3$	$64/3$	—
t_0	μs	6.25	25	7.12	0.030
t_1	μs	$1.5t_0$	$2.0t_0$	10.64	0.026
t_{TR}	μs	$1.1(t_0+t_1)$	$3.0(t_0+t_1)$	32.48	0.035
t_{deli}	μs	11.88	13.12	12.48	0.220
$t^{\text{a}}_{\text{msg}}$	μs	$\max(t_0+t_1, 10t_{\text{a}})0.8-2$	—	171.8	0.090
$t^{\text{a}}_{\text{ack}}$	μs	$3t_{\text{a}}$	—	82.7	6.30
t_{hs}	μs	977.3	—	2600	171

* Sniffed values are averages from a single trace with over 1500 EPC C1G2 messages.

TABLE III
CRFID DOWNSTREAM PROTOCOL PARAMETERS BUILD UPON EPC C1G2

Param.	**Wisent**	**R2**	**Stork** (MOM)	**Stork** (SPM)
b_{da}	16	16	16	16
b_{ov}	58	58	58	58
N	16	1	210	120
t_{meta}	$2t_{\text{msg}}$	t_{RRN}	$14t_{\text{msg}}$	$8t_{\text{msg}}$
t_{check}	$t_{\text{RA}} + t_{\text{hs}} + Nt_{\text{msg}}$	0	t_{Read}	$4t_{\text{Read}}$

Connection Overhead—t_{hs}: The handshake [6, Sec. 6.3.2.6] consists of two messages (Query and ACK—Fig. 3), that have the same structure as t_{msg}, but different parameter set, i.e.

$$t_{\text{hs}} = t_{\text{TR}} + t_{\text{m}}(\mathbf{v}_{\text{Query}}) + t_{\text{m}}(\mathbf{v}_{\text{ACK}}), \quad (5)$$

where $\mathbf{v}_{\text{x}}$ is a parameter set of message x mapped into (4).

LLRP Protocol Overhead—t_{LLRP}: The average (Ethernet-dependent) delay of instructing the reader will be considered as fixed. We refer to [16, Sec. 4.7] for details.

Downstream Protocol Overhead—t_{meta}: Since Impinj readers split the `BlockWrite` into train of `Writes`, Fig. 1(b) [5, Sec. IV-D1], Wisent and Stork add two words of meta data to each `BlockWrite`, i.e. $t_{\text{meta}} = 2t_{\text{msg}}$. For R^2, t_{meta} is executing a `request random number` command (t_{RRN}), so $t_{\text{meta}} = t_m(\mathbf{v}_{\text{Request}})$ as it uses `Write`, which needs a random number for data encryption.

Data Verification Overhead—t_{check}: For Wisent t_{check} is the time of performing the handshake and `BlockWrite` command again, i.e. $t_{\text{check}} = t_{\text{RA}} + t_{\text{hs}} + Nt_{\text{msg}}$, and for Stork t_{check} is executing a `Read` command (t_{Read}), $t_{\text{check}} = t_m(\mathbf{v}_{\text{Read}})$. Although authors of R^2 remark about resending if data is not received successful [7, Sec. III-B], R^2 performs no verification, so $t_{\text{check}} = 0$ making it unsuitable for reliable downstream.

Downstream Protocol Overhead—N: Wisent limit is $N = 16$ words/AccessSpec. Stork is limited by seven, 30 words long `BlockWrites`/AccessSpec, i.e. $N = 210$. For R^2 $N = 1$ as it splits data in `Writes`, with no grouping.

Now, considering Stork (MOM)—with SPM conclusion being equivalent—see Section III-A1a, we insert specific parameters t_{meta}, t_{check} and N into (3) with respective vectors $\mathbf{v}_x$ provided by the EPC C1G2 standard [6] (refer also to [13]). To show that $L_{\text{Stork}} > L_{\text{R}^2}$ we need to show (after parameter assignment) that $209(t_{\text{hs}} + t_{\text{LLRP}} + t_{\text{RA}}) + 210t_{\text{RRN}} > 14t_{\text{msg}} + t_{\text{Read}}$. Proof is completed by observing that $t_{\text{hs}} \gg t_{\text{msg}}$ and $t_{\text{msg}} > t_{\text{Read}}$ as defined in [6, Sec. 6.3.2.12]. Showing that $L_{\text{Stork}} > L_{\text{Wisent}}$ is analogous. $\square$