S2: a Small Delta and Small Memory Differencing Algorithm for Reprogramming Resource-constrained IoT Devices

Borui Li, Chenghao Tong, Yi Gao, and Wei Dong
College of Computer Science, Zhejiang University, China
Alibaba-Zhejiang University Joint Institute of Frontier Technologies, China
Email: {*borui.li, tongch, gaoyi, dongw*}@zju.edu.cn

Abstract—**Incremental reprogramming is one of the key features for managing resource-constrained IoT devices. Nevertheless, existing approaches fall flat in RAM and flask usage due to the increasing firmware size of contemporary IoT applications. In this paper, we advocate S2, a differencing algorithm for reprogramming resource-constrained IoT devices. S2 achieves small memory and flash footprints by leveraging a topological sort based in-place reconstruction mechanism and stream reconstruction technique, as well as smaller delta size by a prediction-based encoding. Evaluation shows that S2 uses 33.3% less RAM while reducing at most 42.5% delta size than state-of-the-arts.**

I. Introduction

A majority of IoT devices trade higher computation ability for a longer battery lifetime. For example, the STM32-H743 microcontroller (MCU) with only 512KB RAM and 2MB flash, which is widely-adopted in IoT devices, achieves months of lifetime with a single battery.

System software of the IoT device needs to be regularly updated for new functionalities or bug fixes. Over-the-air reprogramming is generally deemed as the critical technology for IoT device management because it removes human from the loop of manually re-collecting, "burning" and re-deploying devices. Furthermore, incremental reprogramming [1] gains popularity by lowering the networking cost and reducing the energy consumption. Incremental reprogramming only transmits the *delta* between the old and new version to the device. The delta, including series of copy and add commands, is generated by a *differencing algorithm* and typically much smaller than the complete firmware. Once the delta is received, a patching algorithm on the device *reconstructs* the new version with the old firmware and the delta.

However, due to the higher complexity of IoT applications nowadays, the firmware size increases dramatically, which poses new challenges for state-of-the-art incremental reprogramming approaches. For example, a firmware with an image classification algorithm on the aforementioned STM32-H743 MCU consumes 490KB (95.7%) RAM and 1.9MB (95%) flash. Existing differencing algorithms fail to reprogram under this circumstance because of (1) *Limited RAM*. Algorithms need to read the whole old file to the RAM, which is unfeasible. (2) *Limited flash space*. Algorithms typically do not overwrite the old version in the flash but write the reconstructed version to a different block because overwriting may fail some subsequent copy commands for the lack of necessary data.

To address the above two issues and further reduce the delta size, we advocate S2, a differencing algorithm with small footprints. We summarize the contributions as follows:

- To alleviate the flash pressure, S2 leverages an in-place reconstruction mechanism which allows overwriting the old file without influencing the successive copy commands. The data dependency among copies is solved by interchanging and modifying the copy commands in the original delta using a topological sort based approach.
- S2 proposes a stream reconstruction technique to avoid reading the whole old version that lowers memory usage.
- S2 further reduces the delta size by compressing the commands with a prediction-based encoding.

We conduct experiments using binaries from real IoT projects and results show that S2 uses 33.3% less memory while reducing at most 42.5% delta size than existing works.

II. The S2 Algorithm

S2 algorithm includes the delta generation algorithm (S2Diff) and the reconstruction algorithm (S2Patch).

Preliminaries of the delta file. Generally, the delta file consists of two kinds of commands: COPY(f,t,l) and ADD(t,l). The COPY copies l bytes from offset f of old file to offset t of the new file. The ADD adds l bytes to offset t of the new file. The additive data bytes are also included in the delta file.

S2Diff algorithm. Algorithm 1 shows how S2Diff generates the delta file. Firstly, S2 uses an existing differencing algorithm to obtain the original command sets and additive data. Note that S2 is not bond to a dedicated differencing algorithm, any algorithm that conforms the copy/add mechanism is feasible. The original command set may fail when used for in-place reconstruction because data dependencies may exist among COPYs. Hence, S2 solves it via a topological sort based approach and perform a prediction-based encoding for the commands, which we will explain in detail below. Finally, S2 generates the delta header and concatenates it with the commands and data compressed by `xz`.

Topological sort based dependency solving. The COPY command is executed sequentially when in-place reconstruction. When a subsequent COPY$_B$ uses the data which is overwritten

Algorithm 1: S2DIFF algorithm.

```
Input: Old binary file $F_{old}$, new binary file $F_{new}$
Output: Delta file of the two binaries $Delta$
1 Apply differencing algorithm on $F_{old}$ and $F_{new}$ to obtain the COPY set
    $C_{copy}$, ADD set $C_{add}$ and additive data $D_{add}$;
2 Apply topological sort on $C_{copy}$ to solve data dependency;
3 for $c(f,t,l) \in C_{copy}$ do
4     Update $f$ and $t$ using the difference between the predicted value and
        real value of $f$ and $t$;
5 for $c(t,l) \in C_{add}$ do
6     Update $t$ using the difference between the predicted value and real
        value of $t$;
7 $hdr \leftarrow$ Command count of COPY and ADD, the file offset of additive data
    and the MD5 value of delta;
8 ret $Delta \leftarrow (hdr \mid$ XZ-compressed $C_{copy}$, $C_{add}$ and $D_{add})$;
```

by a former COPY$_A$, a data dependency occurs. To solve the dependency, we can manually place COPY$_B$ earlier than COPY$_A$. For a large set of COPYs, we leverage a topological sort based approach inspired by [2]. We organize the COPY set as a graph where a vertex denotes a COPY and the directed edge from A to B denotes A should be executed earlier than B. Then, we perform topological sort on the graph and generate the final COPY sequence following the topological order. If there are cycles in the graph, we break them by modifying some COPY to ADD in the cycles.

Prediction-based encoding. Each address parameter (i.e., f and t) in the original COPY(f,t,l) and ADD(t,l) is encoded with 8 bytes to support the differencing of large files (<4GB). Nevertheless, we have the observation that the parameters of the ADD sequence (i.e., t) are monotonically increasing, and so do the parameters of COPY (before the topological sort). Hence we re-encode the address parameters with a prediction model. Given a t of COPY to encode, we predict its value using several previous ts and encode t in 2 bytes with the residual of the prediction. In S2, we use the moving average (MA) model because it efficiently transferred and executed on the resource-constrained devices with satisfying accuracy. It is worth noting that the parameters of COPY after topological sort are not strictly monotonically increasing due to the permutation but are still increasing to a large extent. Hence, we use more historical values for COPY in MA to obtain better accuracy.

S2PATCH algorithm. Algorithm 2 illustrates how S2 reconstructs the binary on the device. When the delta is received, S2 first calculates its MD5 value to check the integrity of the delta. Then S2 reads the necessary information from the header and performs the stream reconstruction which we will describe below. Finally, S2 reboots the device to the new image.

Stream reconstruction. S2 proposes stream reconstruction to minimize memory usage. Take the COPY reconstruction as an example. S2 first decompresses the f, t and l using xz and recovers the original value of f and t with MA algorithm. Finally, S2 performs copy following the restored command.

III. EVALUATION

We use eight real IoT projects as the benchmarks to evaluate the memory usage and the size of generated delta file of S2. These benchmarks are based on AliOS Things operating system (AOS) and classified in two categories: (1) benchmarks with small changes between versions (#1-#4, e.g., changing the output period of a serial print application from 1s to 2s), and (2) benchmarks with major changes (#5-#8, e.g., an HTTP data upload application in AOS 2.1 to that in AOS 3.0).

Algorithm 2: S2PATCH algorithm.

```
Input: Delta file $Delta$, old binary file $F_{old}$
Output: Reprogramming result: $Success/Failed$
1 if MD5 check of $Delta$ failed then ret $Failed$;
2 Obtain the command count of COPY $n_{copy}$ and ADD $n_{add}$, and the offset
    of additive data $o_{add}$ via header of $Delta$;
3 for $i \leftarrow 1$ to $n_{copy}$ do            // Stream COPY reconstruction
4     XZ-decompress $l$ and residual of $f$ and $t$ from $Delta$;
5     Restore the value of $f$, $t$ from prediction residual and obtain the
        entry of COPY $c_{copy}[i](f,t,l)$;
6     Copy $l$ bytes from offset $f$ of $F_{old}$ to offset $t$;
7 for $i \leftarrow 1$ to $n_{add}$ do             // Stream ADD reconstruction
8     XZ-decompress $l$ and residual of $t$ from $Delta$;
9     Restored the value of $t$ from prediction residual and obtain the entry
        of ADD $c_{add}[i](t,l)$;
10    Add the Xz-decompressed $l$ bytes from offset $o_{add}$ of $Delta$ to offset
        $t$ of the device;
11    $o_{add} \leftarrow o_{add} + l$;
12 Reboot the device and boot to the reprogrammed image;
13 ret $Success$;
```

TABLE I
EVALUATION RESULTS OF S2 ALGORITHM COMPARED WITH THE BASELINES ON DELTA SIZE AND MEMORY USAGE (UNIT: KB).

#	Old→New	Delta Size			Memory Usage		
		AOS	bsdiff	S2	AOS	bsdiff	S2
1	304.3 → 304.3	1.4	0.32	0.32	144	688.6	96
2	551.6 → 551.6	2.4	0.35	0.35	144	1183.2	96
3	304.3 → 338.1	141.9	109.5	105.5	144	722.4	96
4	304.3 → 338.1	142.3	108.6	104.9	144	722.4	96
5	382.9 → 88.2	292.3	199.8	169.1	144	951.1	96
6	446.9 → 553.0	344.9	154.0	143.0	144	1079.9	96
7	553.0 → 446.9	77.7	84.8	75.2	144	1079.9	96
8	553.0 → 605.3	304.3	245.4	238.3	144	1236.3	96
Avg. var. vs. AOS		**0**	**-38.3%**	**-42.5%**	**0**	**+565.3%**	**-33.3%**

We use two baselines to compare with S2: (1) the original bsdiff featured with the in-place algorithm in [2]; (2) the default incremental reprogramming algorithm in AOS. bsdiff [3] algorithm is specifically optimized for binaries and widely-adopted in the industry (the default differencing algorithm of Google Play [4]). The AOS achieves incremental reprogramming for large files by splitting the original new version and old version to a series of 64KB segment pairs and performing bsdiff on each pair.

Table I shows the evaluation results. We can observe that S2 reduces the delta size by 42.5% and 11.0% against AOS and bsdiff, separately. S2 achieves a 33.3% RAM reduction compared with AOS. The bsdiff consumes considerable RAM mainly because it lacks stream reconstruction technique.

Acknowledgment. This work is supported by the National Key R&D Program of China under Grant No. 2019YFB1600700.

REFERENCES

[1] W. Dong *et al.*, "Optimizing relocatable code for efficient software update in networked embedded systems," *ACM TOSN*, 2015.
[2] R. Burns *et al.*, "In-place reconstruction of version differences," *IEEE TKDE*, 2003.
[3] C. Percival, "Naıve differences of executable code," 2003.
[4] Android Developers Blog, "Improvements for smaller app downloads on google play," 2016.