2012 9th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)

No-reboot and Zero-Flash Over-the-air Programming for Wireless Sensor Networks

Nasif Bin Shafi
School of Computing,
Queen's University
shafi@cs.queensu.ca

Kashif Ali
EECS Department,
University of California, Berkeley
kashif@eecs.berkeley.ca

Hossam S. Hassanein
School of Computing,
Queen's University
hossam@cs.queensu.ca

***Abstract*—Over-the-air reprogramming is an important aspect in the deployment and management of Wireless Sensor Networks (WSNs). However, WSNs reprogramming poses significant challenges due to scarce available energy, low computational power, and limited memory capabilities of the WSNs nodes; all are required for transmission and processing of the created patches. In existing reprogramming schemes, any change in the program layout and/or global variables, produces a significantly large patch size, hence consumes the node's limited resources. Furthermore, to apply the patch, existing schemes require rewriting internal flash, large volume of external flash, as well as rebooting the node. In this paper, we devise a novel reprogramming scheme that we call Queen's Differential (QDiff), which mitigates the effects of program layout changes and retains the maximum similarity between "old" and "new" codes using clone detection techniques. Moreover, QDiff organizes the global variables in a novel way to eliminate the effect of variable shifting. To assess the performance of Qdiff, we have carried out a TinyOS implementation using an IRIS mote platform. Our experiments show that QDiff requires near-zero external flash, and significantly lower internal flash rewriting and transmission overhead than leading existing differential reprogramming mechanisms.**

I. Introduction

Development and deployment of Wireless Sensor Networks (WSNs) is tightly coupled with the desired application(s). The WSNs nodes require code updates, i.e., reprogramming, to address any desired application changes. Main objective of an over-the-air reprogramming scheme for WSNs is to be efficient in creating patches that are small in size while minimizing their deployment and loading cost on the operational and functional performance of the nodes. Communication is the most energy consuming component in WSNs, hence, small patch size directly affects the energy usage and network lifetime.

Applying the patch against the executing code, on the mote, requires significant flash memory (both internal and external) reads and writes. Flash memory usage requires higher (2.7V) than minimal operational voltage (1.7V), at which its behavior is unpredictable. In GreenOrbs WSNs application, researchers found that for the TelosB WSN mote, the network lifetime is reduced from 40 days to 11 days when using a reprogramming protocol with full flash memory writing [1]. Therefore, reducing flash writing reduces the energy consumption of the mote, hence allowing more of the mote reprogramming while maintaining the network lifetime. Any existing reprogramming approach requires full flash memory rewriting and consumes significantly more power and time.

The loading cost for executing the new code is also minimized, in particular, the mote reboot. The restart is essentially used to perform complete code flash rewriting, even for the simplest case. Restarting of the WSN mote is energy inefficient and results in overall network instability and functional failure. For instance, in a recent deployment on Reventador Volcano, reboots led to 3-day network outage, reducing mean node uptime from over 90% to 69% [1], [2]. The only exception, at the cost of significant changes in the nesC [3] compiler, is the Elon scheme. However, the Elon scheme only works for a certain WSNs platform and operating system.

In this paper, we propose an efficient over-the-air reprogramming scheme, QDiff (Queen's Differential). The proposed QDiff scheme calculates the minimum size patch by keeping maximum similarity between two programs, old and new, using clone detection. Clone detection is used to find similar functions and variables thereby re-organizing function and variable layout to keep the maximum similarity between old and new programs. New functions or variables are padded at the end of the program. Most of the existing reprogramming mechanisms fail to handle global variable shifting due to insertion or deletion of variables. QDiff proposes a new method of re-organizing of global variables that efficiently eliminates the problem of global variable shifting. These two approaches drastically reduce the size of patch, while facilitating maximum possible function and variable layout changes. To evaluate the performance, we have implemented QDiff in TinyOS using an IRIS mote platform. Our experiments show QDiff, when compared with Stream [4] and Hermes [5], produces minimal patch size (by 250 and 12 times, respectively), lower internal flash rewriting (by 21 times), near-zero external flash and does not require node reboot. To best of our knowledge, the proposed approach is the first platform independent scheme which produces small patch with minimal flash memory rewritings and does not require mote restart.

The remainder of the paper is organized as follows. Section II highlights the technical issues and design objectives. Section III describes the related work. Section IV explains the scheme, patch creation and patch deployment of the proposed scheme. Section V provides detailed information about the implementation details, performance metrics and evaluation methodology. Finally, section VI concludes this paper.

II. Issues and Design Objectives

A. Issues

A single line of code change, in a high-level programming language, may cause a significantly larger change at the lower-level, e.g. assembly code. To address this, existing solutions [1], [6], [7] require changes in the programming language syntax or the compiler itself and hence applies only to certain platforms. A heterogeneous solution, on the other hand, although independent of compiler and mote architecture, needs to solve the following technical issues:

Functions: The compiler might change its functional layout for optimization purposes resulting in a large patch file. Moreover, these changes shift the machine code in the final ELF file. Furthermore, even a small change, e.g. insertion of single instruction, may shift all of its subsequent instructions and branch addresses. This will yield a significantly large patch file whose deployment would require full flash rewriting.

Global variables: During the creation of an executable file, the compiler considers the global variables in two contexts: initialized and uninitialized. Initialized global variables are kept in the data section, and uninitialized global variables are kept in the bss section. To optimize memory space, both of these sections (data and bss) are kept back-to-back without any slots in the memory space. Due to this, any addition in data section, e.g., new variable, shifts the address of all variables within the bss section. Furthermore, the shift also modifies all instructions that reference the shifted variables, hence results in a very large patch file. Moreover, for every compilation, there is no assurance that a compiler will keep the same variable layout.

Relative jumps: The relative jump range is calculated using the current memory location. Based on current memory address a relative jump can either be a positive offset or a negative offset. Positive offset means that the jump target address is higher than the current address. Negative offset means that the jump target address is lower than the current address. Addition or removal of any new or existing instructions, between the current memory location and the target location, will result in a change in the jump offsets. This happens even if the instruction at the target location has not been modified, hence resulting in an unnecessarily large patch file.

Indirect addresses: Indirect addressing is an important part of the Reduced Instruction Set Computing (RISC) machine. The RISC architecture does not allow direct access to the memory location. Memory locations are accessed or modified using registers. Indirect addressing provides fast access to large data structures, e.g. arrays, linked lists, union, etc. However, any changes in the global variable layout will also change the corresponding indirect instructions and must be taken care of in order to produce correct patch files.

B. Design Objectives

The main objectives of the reprogramming scheme are as follows:

- Minimizing code difference – small difference means less energy and time to reprogram the network.
- Minimizing flash memory writing – flash memory writing is energy hungry and reducing flash memory writing means better energy and time efficiency.
- Minimizing external flash – means reducing cost as well as reducing the requirement for board real estate.
- Supporting heterogeneity – many different hardware platforms and operating systems are available for wireless sensor networks, with diverse specifications.
- Eliminating the need for mote reboot after applying the patch on mote software.

III. Related Work

Reprogramming schemes in WSNs can be classified as system level, module level, virtual machine and differential approaches.

In the differential reprogramming scheme, the base-station generates a patch using the difference between the old and updated program. Rsync [8] is a well known differential update scheme that is widely used in desktops and servers. The Rsync divides the program into different blocks and calculates the hash value of those blocks. The hash values are then matched to determine the block's insertion, deletion, or modification. In reference [9], authors present a Rsync-based scheme to generate a small sized patch. However, this scheme does not produce an optimized result as it discards ELF file structure while calculating the difference. It also requires rewriting the whole flash memory as well as large amount of external flash memory. Moreover, the scheme is not optimized for call, jump and function rearrangement.

In reference [7], authors propose a differential scheme, which changes the linking procedure and keeps a slop region after each function to reduce the patch size caused by call and jump instruction. This scheme creates a small patch, however it causes fragmentation of flash memory. Moreover, since the size of the slop space is heuristically assigned, which leads to inefficient utilization of flash memory.

Zephyr [6] keeps a jump table via which all call and jump go to their destination. This mechanism reduces patch size along with mote's energy and processing requirements. However, the scheme shows poor performance with applications with loop, which is common in WSNs. Hermes [5], an improvement to Zephyr, removes the slop region for global variables and jump table at the cost of a complex bootloader. A complex bootloader requires significantly larger boot memory. Moreover, Zephyr and Hermes both require complete flash memory rewriting and large amounts of external flash memory.

Elon [1] overcomes the restart requirement of earlier schemes while also reducing flash memory access. Elon divides the system as two different components: replaceable and non-replaceable. Core operating system is considered as non-replaceable and other parts are considered as replaceable components. Although Elon does not require mote restart, it is highly platform specific. For instance, currently the scheme, by design, can only work on a TelosB mote running TinyOS.

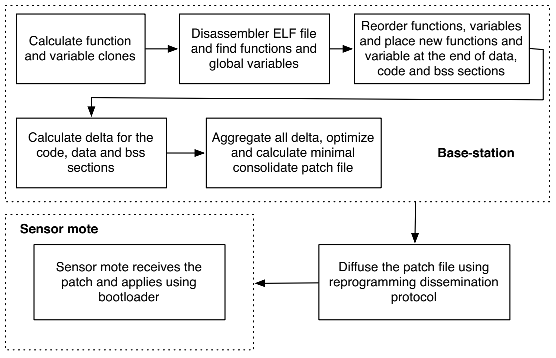

Calculate function and variable clones
Disassembler ELF file and find functions and global variables
Reorder functions, variables and place new functions and variable at the end of data, code and bss sections
Calculate delta for the code, data and bss sections
Aggregate all delta, optimize and calculate minimal consolidate patch file
Base-station
Sensor mote
Sensor mote receives the patch and applies using bootloader
Diffuse the patch file using reprogramming dissemination protocol

Fig. 1. Major steps of the QDiff differential reprogramming scheme.

Furthermore, Elon also requires significant changes in the compiler to support new programming language syntax.

R^2 [10] reduces the patch size by using an efficient implementation of dynamic loading and linking modules. The R^2 scheme maintains meta-data regarding changeable information and the difference of this meta-data is transmitted to motes. Using meta-data eliminates the need of jump table (requirement of Hermes and Zephyr) and complex bootloaders. However, this scheme requires full flash memory writing. Moreover, a significantly large amount of code flash memory is required to maintain the meta data.

All of the aforementioned schemes perform poorly when there is a change of program and variable layout, require full flash memory writing and large amounts of external flash memory.

IV. QDiff: Differential-based Reprogramming Scheme Using Clone Detection

The focus of our research is to devise an efficient differential scheme to fulfill the design objectives of no-reboot, eliminating frequent flash memory writes, support heterogeneity and produce minimum possible patch size while supporting code and variable layout changes. To this we propose a novel differential scheme, QDiff, which re-organizes global variables and makes use of clone detection techniques to determine any code dissimilarities.

To calculate the patch file, QDiff considers both executable files, i.e., the ELF file, as well as high-level source code. Making use of the high-level programming languages provide significant advantages over other schemes, which only use low-level languages because of its grammatical and syntax rules provide better flexibility in order to determine code similarities. The code similarities are determined using a clone detection tool. The reprogramming steps of QDiff are shown in Fig. 1 and are explained next.

Step-1: QDiff calculates clones between old and new files using a clone detection tool. The clone detection tool takes in the C file of old and modified programs and calculates mapping between functions and variables. The C files, in case of TinyOS programming, are generated by the compiler as it converts the nesC into C before it create the final ELF file. Mapping is a list of functions and variables cloned between the old and the new file, i.e., FunctionClonePair(FunctionOld, FunctionNew) and VariableClonePair(OldVariable, NewVariable). The clone detection algorithm can detect clones under various circumstances, e.g. change in file layout, rename of functions and variables, addition or removal of one or more lines of code, etc. For this paper, we emphasize "Type 3" clone detection [11] as it can detect addition and removal of new code, functions and variables.

Step-2: QDiff disassembles the ELF files (both old and modified), using the core dump utilities, to determine different sections, i.e., code, data and bss. In this step, mappings between functions and global variables, computed from clone detection, are calculated. Step-2 is further subdivided into: **(2a.)** Reorder functions and global variables to enhance similarity between old and new programs. Place new functions, initialized and uninitialized global variables at the end of code, data and bss section respectively. **(2b.)** Change all references to the re-ordered functions and variables. For instance, if a *call* instruction invokes a function *func1* at address *0x234* and reordering caused the change of *func1* location to *0x456*, all call instructions invoking *func1* changes their destination address to *0x456*. A similar approach is taken for the global variables. **(2c.)** Modify organization of uninitialized global variables, i.e., variables in bss section. Organize uninitialized global variables as stack fashion. Place new variables at the top of bss section. **(2d.)** Calculate delta between old and new code section. This includes clone function and newly added (or removed) functions. **(2e.)** Calculate delta between old and new data section. This includes added, deleted or modified initialized variables. **(2f.)** Calculate delta between old and new bss section. This includes added, deleted or modified uninitialized variables. **(2g.)** Aggregate all deltas and calculate the final patch file. The file is optimized to minimize the patch size and flash memory access.

Step-3: Base-station transmits the patch file over wireless medium, which is dispersed through the whole network using an existing dissemination protocol.

Step-4: Sensor mote receives the patch file and invokes the bootloader, which applies it without rebooting the mote.

A. QDiff Patch Creation Process

Differential patch creation needs to address four major technical issues: branches, global variables, relative jumps and indirect addresses. In this section, we explain QDiff's approaches to solve these issues.

1) Branches: Code change affects branch instructions in two ways. First, an insertion or deletion of new instruction(s) which may shift all the subsequent instructions. This shift changes all call and jump instructions that invokes functions or jump to the location in shifted portion. Second, the compiler may change the function order during code optimization. This reordering changes functions' locations and subsequently requires to change all *call* instructions referring to them.

QDiff solves both scenarios by appending any modification to the end of the ELF file. In case of compiler reordering and renaming, the branch related issues are solved using clone

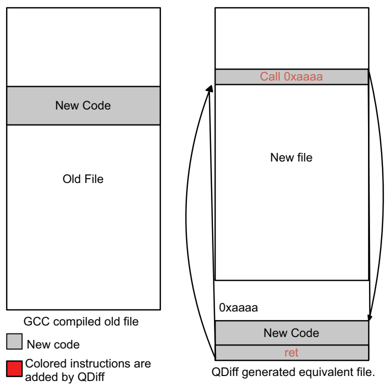


Fig. 2. Padding newly added code block at the end of program code.

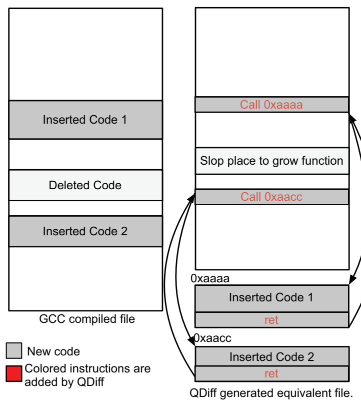


Fig. 3. Illustration of QDiff to handle simultaneous function insertion and deletion.

detection. If any new function is added, in the middle of code, QDiff moves its location and changes all branch instructions that are referring to it. The main idea is to minimize the flash memory writing as only the new functions and the instructions referring to it, need to be rewritten. On the other hand, if new instructions are added within the function, QDiff modifies the program to bring maximum similarity in two ways. First, if there is a slop region after that function, the function is extended to that slop region. This slop region is created due to the deletion of a function. Second, if there is no slop region, new instructions are moved and padded at the end of program. From the callee function, new instructions are added by QDiff; jump to the newly padded instruction and return back to the callee function. This is illustrated in Fig. 2.

Code deletion, related to branches, comes in two forms: removal of complete function or some code from within a function. In the first case, if the function is deleted from the program, slop regions are inserted in place of the deleted function. The slop regions facilitate future growth of existing functions or insertion of new functions. Moreover, this provides more similarity between the old and new program and reduces patch file size. To keep the patch file at minimum the slop regions are kept within or after the function. The simultaneous addition and deletion of function (or code within) is a multi-step process. First, the deleted functions are removed from the old program. This removal creates slop regions within the old program. Second, for the newly added code/function, all matching slop regions are first determined. Matching slop-regions is only concern with those slop-regions with size larger or equal to the newly added code/function. To break the tie, QDiff selects the largest region. The slop region is fully utilized by inserting maximum possible functions. The maximum slop region utilization problem is a variant of a typical (0,1) knapsack problem. If there is no slop region whose size is larger than the function size then the new function is padded at the end. If some code is inserted in an existing function and existing function has some slop region following it, the function is expanded. Otherwise, new instructions are padded at the end. A call-return pair is inserted into program code to jump to/from the newly padded instructions. An illustration

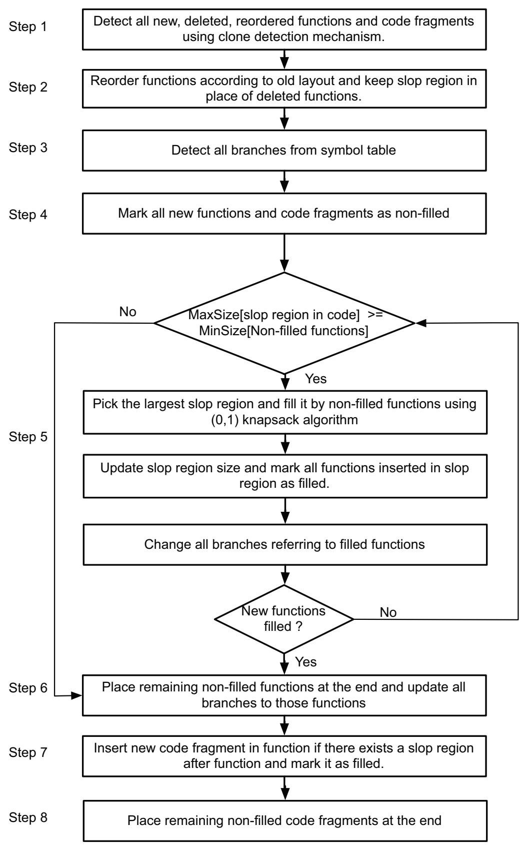

Step 1
Detect all new, deleted, reordered functions and code fragments using clone detection mechanism.
Step 2
Reorder functions according to old layout and keep slop region in place of deleted functions.
Step 3
Detect all branches from symbol table
Step 4
Mark all new functions and code fragments as non-filled
No
MaxSize[slop region in code] >= MinSize[Non-filled functions]
Yes
Step 5
Pick the largest slop region and fill it by non-filled functions using (0,1) knapsack algorithm
Update slop region size and mark all functions inserted in slop region as filled.
Change all branches referring to filled functions
New functions filled ?
No
Yes
Step 6
Place remaining non-filled functions at the end and update all branches to those functions
Step 7
Insert new code fragment in function if there exists a slop region after function and mark it as filled.
Step 8
Place remaining non-filled code fragments at the end

Fig. 4. Flowchart for fixing branches

of QDiff's approach to handle simultaneous insertion and deletion is shown in Fig. 3. The algorithm flowchart is shown in Fig. 4.

2) Global Variable: In an ELF file, global variables are kept in two separate sections, the data section for the initialized variables and the bss section for the uninitialized variables. Both data and bss sections are kept in heap, which is initialized in the RAM by the initializer function. In a traditional ELF file, data and bss sections are kept back to back, i.e., in-sequence. Due to this placement, the addition of new variables in the data

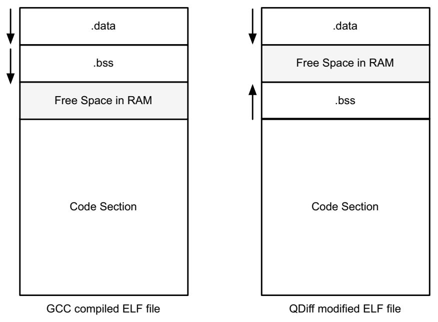


Fig. 5. Comparison of bss section in a traditional ELF and QDiff ELF file.

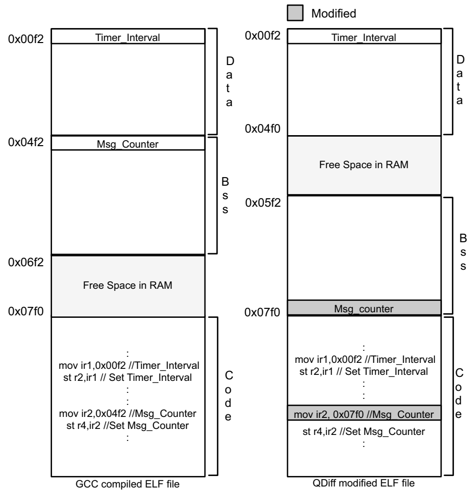
section will shift the subsequent variables in the data variables and all variables in the bss section. This shift will change the instructions involving the modified variable, due to change in variables memory addresses. This leads to a significantly large patch file. Similar is the case of variable(s) deletion.

To address the global variable problem, the QDiff scheme reverses the memory address order for the bss section while maintaining the original order for the data section. Fig. 5 shows the position of both data and bss sections (memory layouts) for a traditional ELF and the QDiff modified file. As there is free space between data and bss sections, therefore addition or deletion of global variables, either initialized or uninitialized, will not cause any shift in the variables addresses and their subsequent instructions. Addition, deletion, or re-arrangements of initialized or uninitialized global variables, similar to branch instructions, may result in code shift.

In case of compiler reordering and renaming, the global variable related issues are solved using clone detection. If any variable is deleted, it is removed from the heap, however the location is maintained as a slop region. If a variable is added, QDiff checks all slop regions, picks up the largest slop region and fills the slop region by the new variable, using (0,1) knapsack algorithm. If there is no such type of region, the new variable is added at the end. An initialized global variable is added at the bottom of data section. Whereas, an uninitialized global variable is added at the top of the bss section.

Special consideration must be given for memory access via indirect registers. The indirect registers are used for accessing arrays and large data structure. In QDiff, the addition of new array in the data section follows the traditional ELF method, i.e., placed at the bottom of the section. In case of the bss section, the base address of the array is changed while maintaining the array order. This reduces the complexity of handling indirect registers with offsets.

Changes in the initializer function (.init) is required to handle the QDiff modified data and bss section. QDiff changes the layout of bss and data in memory to stack and heap fashion which allows to grow data and bss section freely. This reduces shift of all global variables in bss section while a single variable is added in data section. The new initializer function just reads from bss section in reverse direction. The ELF organization changes are transparent to the programmer hence, require no special programming considerations.



Modified
0x00f2
Timer_Interval
Data
0x04f2
Msg_Counter
Bss
0x06f2
Free Space in RAM
0x07f0
mov ir1,0x00f2 //Timer_Interval
st r2,ir1 // Set Timer_Interval
mov ir2,0x04f2 //Msg_Counter
st r4,ir2 //Set Msg_Counter
Code
GCC compiled ELF file
0x00f2
Timer_Interval
Data
0x04f0
Free Space in RAM
0x05f2
Bss
0x07f0
Msg_counter
mov ir1,0x00f2 //Timer_Interval
st r2,ir1 // Set Timer_Interval
mov ir2, 0x07f0 //Msg_Counter
st r4,ir2 //Set Msg_Counter
Code
QDiff modified ELF file

Fig. 6. Illustration of indirect address access for traditional and QDiff ELF.

3) Indirect Addressing: Indirect addressing is a technique to access memory location using registers. Effective address of the memory location is specified in an indirect register. For example, $Timer\_Interval$ resides in address location $0x00f2$ of data section in Fig. 6. In code section, this address location is loaded in indirect register $ir1$. As $ir1$ contains memory location of $Timer\_Interval$, all $load/store$ instructions using $ir1$ effects $Timer\_interval$. In sub-sequent instruction, $st\ r2,\ ir1$, value of $Timer\_Interval$ is set by register $r2$. Typically, indirect addressing is used for faster access of data. Due to re-arrangement of bss section, QDiff modifies indirect instructions referring to bss section. Fig. 6 illustrates the working of QDiff indirect addressing, showing (left side of Fig.) the modified global variables, $Timer\_Interval$ and $Msg\_Interval$. These two variables are set, in the code section, using indirect registers $ir1$ and $ir2$, respectively. To address this, QDiff modifies (right side of Fig.) the memory location of $Msg\_Interval$. This modification however, due to QDiffs stack-based bss section, invalidates the instructions referring to it and is accordingly modified. Such modification can be performed by using two different instruction types. First, by updating indirect registers through an immediate value. Second, updating indirect through another register. QDiff generates control flow graph to detect these two type instructions and modifies them to refer to new location of corresponding variable. Fig. 6 shows the modified code section wherein the original "$mov\ ir2,\ 0x04f2$" is changed to "$mov\ ir2,\ 0x07f0$"; $0x07f0$ is the new memory location of $Msg\_Interval$ variable. The QDiff algorithm to handle indirect addressing is shown in flowchart of Fig. 7.

4) Relative Jumps: Relative jump is a position independent code. These jumps are generally smaller in size and are measured relative to current instruction pointer. Jump instructions involve either positive (address is after current instruction address) or negative (address is before current instruction address) offset. Relative jump instructions are changed if the instructions are added or deleted in between the jumping

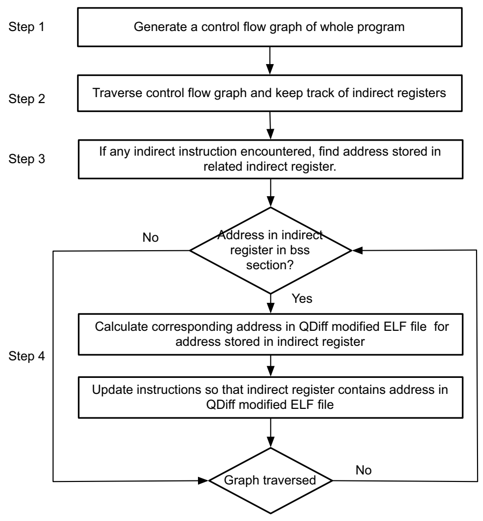

Step 1
Generate a control flow graph of whole program
Step 2
Traverse control flow graph and keep track of indirect registers
Step 3
If any indirect instruction encountered, find address stored in related indirect register.
Address in indirect register in bss section?
No
Yes
Step 4
Calculate corresponding address in QDiff modified ELF file for address stored in indirect register
Update instructions so that indirect register contains address in QDiff modified ELF file
Graph traversed
No

Fig. 7. Flowchart for fixing indirect addresses.

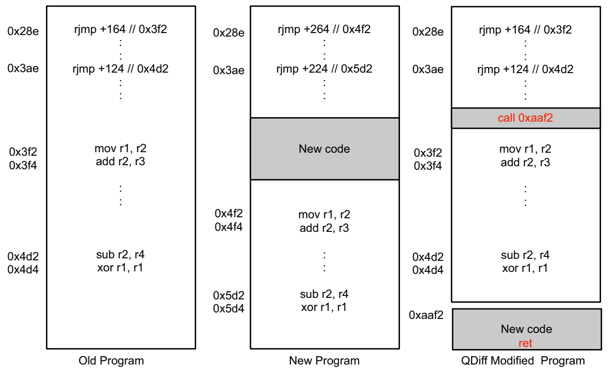


Fig. 8. Illustration of change in the relative jump instructions.

source and destination. The QDiff scheme handles relative jump by the use of basic block calculation technique. The basic block is the code block that contains at most one branching instruction, generally at the end.

Fig. 8 illustrates the changes in the relative jump instructions, due to insertion of new code. Insertion of new block code causes a change in the relative location of the two jump instruction, at address $0x28e$ and $0x3ae$. To address this change, QDiff first identifies all the basic blocks in the program code. Afterward, the basic block that causes least change in the relative jumps is identified and is moved to end of program code. Slop regions are inserted as replacement for the moved block of code. Relative jump requires special care as there might exist some relative jumps in new code that jump to some position in old program. To overcome this issue, QDiff only pads the basic block at the end. QDiff moves the new code block at $0xaaf2$, therefore, maintaining the same address for relative jumps. Fig. 9 describes the procedure to handle relative jumps.

B. QDiff Patch Deployment Process

The base-station encodes delta and transmits it over the network. Based on requirement, the base-station disseminates

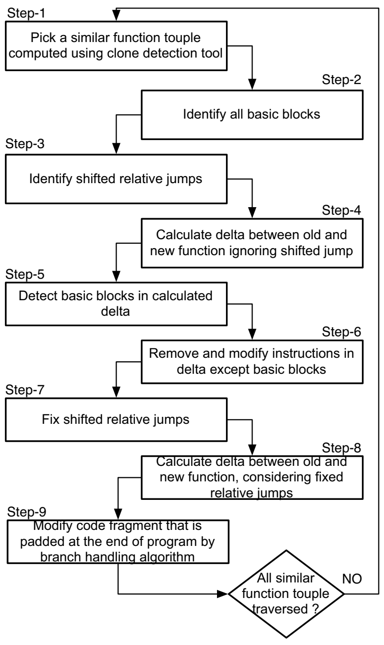

Step-1
Pick a similar function touple computed using clone detection tool
Step-2
Identify all basic blocks
Step-3
Identify shifted relative jumps
Step-4
Calculate delta between old and new function ignoring shifted jump
Step-5
Detect basic blocks in calculated delta
Step-6
Remove and modify instructions in delta except basic blocks
Step-7
Fix shifted relative jumps
Step-8
Calculate delta between old and new function, considering fixed relative jumps
Step-9
Modify code fragment that is padded at the end of program by branch handling algorithm
All similar function touple traversed ?
NO

Fig. 9. Flowchart for fixing relative jumps.

delta over the whole network or part of the network. Any transmission protocol format such as deluge, trickle and stream can be used. We assume that the transmission protocol will deliver the patch completely and reliably over the network. The delta produced by QDiff is not dependent on any particular dissemination protocol. Sensor mote receives delta and stores it in internal flash or RAM if its size is small. However, if delta size is large, it is stored in external flash memory. When sensor mote receives delta completely, it calls bootloader. Bootloader parses and applies delta on the existing code. If parsing fails, boatloader restores the old image.

V. Implementation and Evaluation

In this section, we present the implementation details of QDiff, performance metrics, experimental use cases and their results. For performance comparisons, we use Zephyr [6] and Hermes [5] due to their better efficiency than other schemes. Furthermore, both qualitative comparison and overheads are also investigated.

A. Implementation details

The QDiff implementation has two parts; first is Java implementation which runs on the base-station and second is nesC (TinyOS) implementation which runs on an IRIS mote platform. The IRIS mote uses atmega1281 microcontroller as a processing unit. IRIS has 8KB RAM, 128KB internal flash and 512KB external flash memory. In order to create a single OS image, i.e., a monolithic image, the nesC compiler converts the nesC code to a corresponding C code. This C file is later compiled and linked using an architecture specific C compiler, e.g. avr-gcc for the IRIS mote. We make use of the Bauhaus toolkit [12] to detect clones between programs. Bauhaus is a well-known clone detection tool and works for various programming languages, including C. While the Bauhaus toolkit does not work directly for assembly language, it is still used indirectly as the compiler keeps same identifier name for both C and Assembly languages. We have extended the Bauhaus toolkit to enhance the clone detections, especially to support "Type 3" clone detection. To determine clones, the ELF sections (code, bss and data) are dumped using the standard binary utilities tools, e.g. avr-objdump and avr-readelf. At the base-station, these tools are applied on the new ELF and old ELF files to create the desired patch and is then wirelessly transmitted to the motes. Google-guava API and extended Google diff-match-patch are used for analyzing dumped ELF files and determining the patch. The generated patch is made of series of commands as follows:

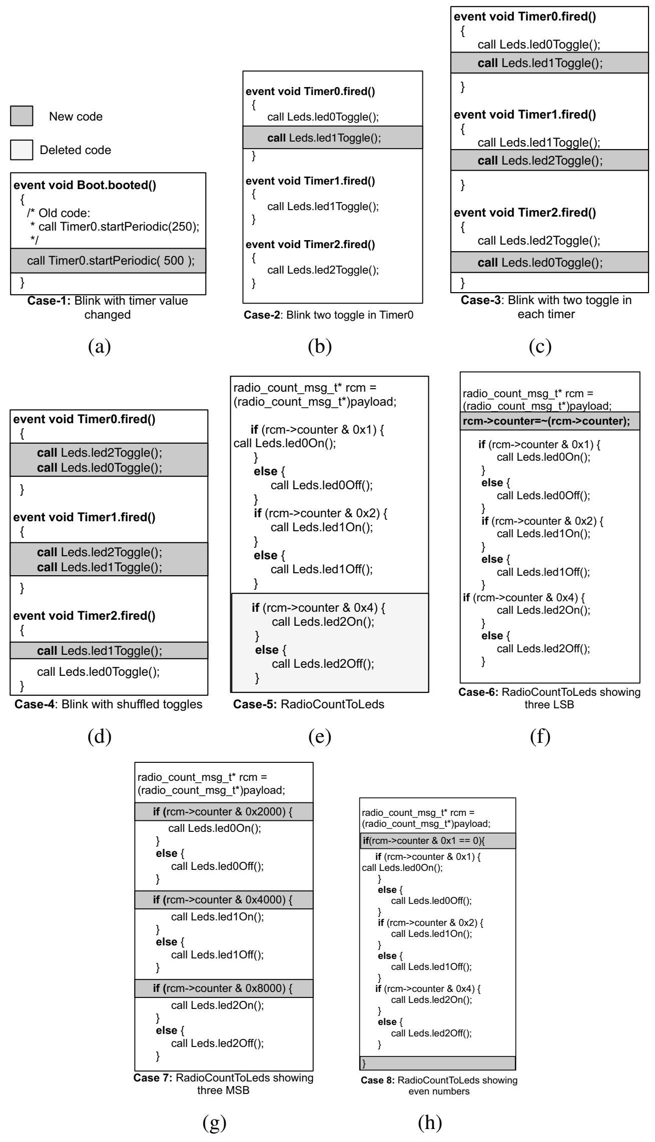
```
insert <newOffset><size=s1><data>
delete <newOffset> <size=s2>
copy <oldOffset><size=s3><newOffset>
pad <newOffset><size=s4>
repeat <newOffset><size=s5><numrepeat=n>
        <addr1> ... <addrn>
```

insert: This command inserts data of size s1 at *newOffset* memory location. This data can either be the machine code or global variables.
delete: This command erases code and variable of size *s2* from *newOffset* memory location.
copy: This command copies code and variable from the old ELF to the revised version. The address for the old and the new ELF location is specified by *oldOffset* and *newOffset*, respectively. The data size to be copied is specified by the size *s3*. This command is also used by the Hermes and Zephyr schemes to build the image for the external memory.
pad: This command pads s4 many zeros, in new image, at memory location specified by *newOffset*.
repeat: This command performs consecutive copy of an instruction with same size. Number of times and amount of data to be copied is specified by *n* and size *s5*, respectively. This command is used by the Zephyr or Hermes schemes for transmitting jump table.

The command opcode and offset address is represented using one and two bytes, respectively. We assume address length of 24 bits that can address up to 16MB and 32MB if the address is byte aligned and word aligned, respectively. The generated patch is fragmented into network packets and disseminated over the network using the dissemination protocol, e.g. Stream. The received patch, depending on its size, is stored either in the RAM (if smaller than available RAM space) or the external memory. The external memory is used only when the patch size exceeds the RAM space. We have implemented the bootloader using nesC language. The bootloader parses the received patch, builds target flash memory pages and writes them back to the internal flash memory.

B. Evaluation Scenarios

To cover various code modifications and to evaluate the performance of reprogramming protocols, we ran numerous evaluation cases. These cases are outlined in Fig. 10. The evaluation cases covers all the four major issues, i.e., branches, global variables, indirect addressing and relative jumps, that any differential scheme should address.



```
event void Boot.booted()
 {
  /* Old code:
   * call Timer0.startPeriodic(250);
   */
  call Timer0.startPeriodic( 500 );
 }
```

Case-1: Blink with timer value changed

(a)

```
event void Timer0.fired()
 {
    call Leds.led0Toggle();
    call Leds.led1Toggle();
 }
event void Timer1.fired()
 {
    call Leds.led1Toggle();
 }
event void Timer2.fired()
 {
    call Leds.led2Toggle();
 }
```

Case-2: Blink two toggle in Timer0

(b)

```
event void Timer0.fired()
 {
    call Leds.led0Toggle();
    call Leds.led1Toggle();
 }
event void Timer1.fired()
 {
    call Leds.led1Toggle();
    call Leds.led2Toggle();
 }
event void Timer2.fired()
 {
    call Leds.led2Toggle();
    call Leds.led0Toggle();
 }
```

Case-3: Blink with two toggle in each timer

(c)

```
event void Timer0.fired()
 {
    call Leds.led2Toggle();
    call Leds.led0Toggle();
 }
event void Timer1.fired()
 {
    call Leds.led2Toggle();
    call Leds.led1Toggle();
 }
event void Timer2.fired()
 {
    call Leds.led1Toggle();
    call Leds.led0Toggle();
 }
```

Case-4: Blink with shuffled toggles

(d)

```
radio_count_msg_t* rcm =
(radio_count_msg_t*)payload;

   if (rcm->counter & 0x1) {
call Leds.led0On();
   }
   else {
      call Leds.led0Off();
   }
   if (rcm->counter & 0x2) {
      call Leds.led1On();
   }
   else {
      call Leds.led1Off();
   }
   if (rcm->counter & 0x4) {
      call Leds.led2On();
   }
   else {
      call Leds.led2Off();
   }
```

Case-5: RadioCountToLeds

(e)

```
radio_count_msg_t* rcm =
(radio_count_msg_t*)payload;
rcm->counter=~(rcm->counter);
   if (rcm->counter & 0x1) {
      call Leds.led0On();
   }
   else {
      call Leds.led0Off();
   }
   if (rcm->counter & 0x2) {
      call Leds.led1On();
   }
   else {
      call Leds.led1Off();
   }
if (rcm->counter & 0x4) {
      call Leds.led2On();
   }
   else {
      call Leds.led2Off();
   }
```

Case-6: RadioCountToLeds showing three LSB

(f)

```
radio_count_msg_t* rcm =
(radio_count_msg_t*)payload;
  if (rcm->counter & 0x2000) {
     call Leds.led0On();
  }
  else {
     call Leds.led0Off();
  }
  if (rcm->counter & 0x4000) {
     call Leds.led1On();
  }
  else {
     call Leds.led1Off();
  }
  if (rcm->counter & 0x8000) {
     call Leds.led2On();
  }
  else {
     call Leds.led2Off();
  }
```

Case 7: RadioCountToLeds showing three MSB

(g)

```
radio_count_msg_t* rcm =
(radio_count_msg_t*)payload;
if(rcm->counter & 0x1 == 0){
   if (rcm->counter & 0x1) {
call Leds.led0On();
   }
   else {
      call Leds.led0Off();
   }
   if (rcm->counter & 0x2) {
      call Leds.led1On();
   }
   else {
      call Leds.led1Off();
   }
   if (rcm->counter & 0x4) {
      call Leds.led2On();
   }
   else {
      call Leds.led2Off();
   }
}
```

Case 8: RadioCountToLeds showing even numbers

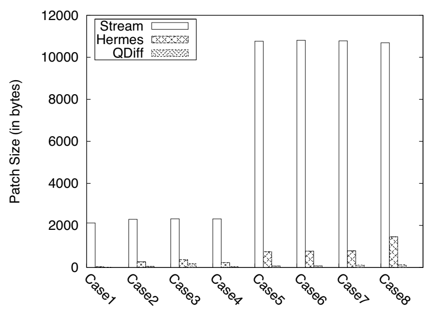
(h)

Fig. 10. Code modification cases for evaluation.

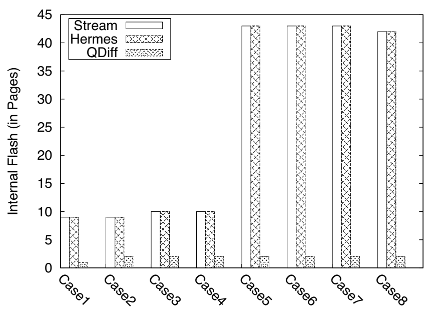
C. Performance Evaluation

In this section, we quantitatively and qualitatively analyze Stream, Hermes, and QDiff reprogramming schemes, for various evaluation scenarios, using various performance metrics.

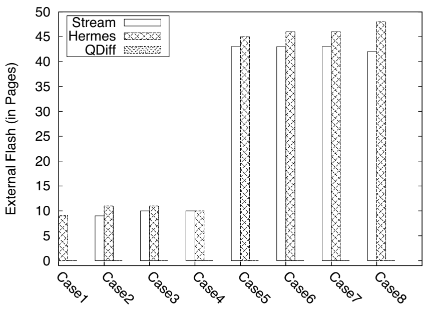
1) Patch size: Patch size refers to the file size of a generated patch. The generated patch includes both data and commands, as explained in Section V-A. The size however, does not include the overhead that may be caused as the patch size is fragmented into smaller network packets. Patch size is an important performance metric as smaller patches imply less data transmission in WSNs, hence lower energy consumptions and less reprogramming time for each mote.



(a) Patch Size



(b) Internal Flash



(c) External Flash

Fig. 11. Performance comparison for the Stream, Hermes and QDiff reprogramming schemes.

Patch sizes for all of the evaluation scenarios, using Stream, Hermes and QDiff, are shown in Fig. 11-a. QDiff outperforms Stream and Hermes up to **250** and **12** times, respectively, for all of the scenarios considered. The highest relative improvement, between QDiff and Hermes, is observed for “Case 8”. In this case, due to compiler optimizations, the function order has been changed. This is not detected by the Hermes scheme. QDiff, on the other hand, using the clone detection technique is able to detect this structural change is more efficient.

2) Internal flash writing: Internal flash writing refers to the number of flash memory pages required to be rewritten upon receiving a patch. The internal flash size also includes the overhead of using external flash to build (or modify) the newly received image. However, we do not include the overhead to load the reprogramming protocol from the external storage into internal flash. Energy consumption of flash memory writing is similar to wireless transmissions. Therefore, reducing internal flash writing reduces the energy consumption of mote, hence allowing more of the motes reprogramming while maintaining the network lifetime.

Flash memory writing is a page-base approach, in IRIS mote the size of flash page is 256 bytes. Experimental results for internal flash writing for all evaluation scenarios are shown in Fig. 11-b. Byte equivalents of internal flash writing are shown in Table I. Both Stream and Hermes schemes require complete flash rewriting of the program, and hence require the same amount of flash page writing. Stream also requires rewriting the reprogramming protocol in internal flash, and Hermes requires rebuilding the code image on external flash memory. In more complex real-life scenarios, these flash writings could be significantly higher. As shown in Table I, the proposed QDiff scheme outperforms both Stream and Hermes, by a factor of **21** times. Furthermore, the QDiff scheme, unlike other schemes, is not affected by the complexity of the use case. For instance, the frequency of internal flash writings for both Stream and Hermes jumps five times, between Case 4 and Case 5. This is because, QDiff only writes modified flash pages, whereas both Stream and Hermes rewrite the whole program flash every time.

3) External flash usage: External flash usage refers to the amount of external flash memory required for storing patch script and building new image from patch script. External flash, if present, for certain applications is mainly used for various network functions, e.g., storing, sensing and debugging information, logging network events, etc. However, it is not a mandatory requirement for other applications, e.g., real-time pressure sensing in home automation. Moreover, accessing the external flash memory for either read or write, performed via SPI bus, is both slow and power hungry. Therefore, reducing the amount of external flash, or ideally eliminating the need of it by the reprogramming scheme is of great importance.

	External Flash			Internal Flash		
	Stream	Hermes	QDiff	Stream	Hermes	QDiff
Case 1	2120	2157	0	2304	2304	256
Case 2	2292	2563	0	2304	2304	512
Case 3	2316	2687	0	2560	2560	512
Case 4	2312	2546	0	2560	2560	512
Case 5	10766	11515	0	11008	11008	512
Case 6	10810	11592	0	11008	11008	512
Case 7	10782	11575	0	11008	11008	512
Case 8	10690	12148	0	10752	10752	512

TABLE I
FLASH MEMORY (IN BYTES) REQUIREMENT COMPARISON

In the Hermes scheme, the bootloader builds the new image using the patch script on external flash. Similarly, the Stream scheme also stores both the reprogramming protocol and the patch file in the external flash. Experimental results for external flash memory usages (in pages) and their byte equivalents are shown in Fig. 11-c and Table I, respectively. Results do not include the flash memory required for storing the golden image. Both Stream and Hermes require large volume of external memory. Hermes requires more external flash as it needs to store the patch script, as well as build and store new images. However, the QDiff scheme does not require any external memory as it can build the new image using RAM. This results in significant lower energy consumption and less reprogramming time for the motes.

4) System Restart: Existing reprogramming schemes require the mote to be restarted. This restart is essential as these schemes perform complete code flash rewriting, even for the simplest case. Restarting of the WSN mote is energy inefficient

	Patch size	External flash	Flash rewrite	Hetero-geneity	System Restart
Deluge [16]	Huge	Yes	Yes	Yes	Yes
Rsync [8]	Medium	Yes	Yes	Yes	Yes
Zephyr [6]	Low	Yes	Yes	Yes	Yes
Hermes [5]	Low	Yes	Yes	Yes	Yes
Elon [1]	Medium	Yes	Yes	No	No
QDiff	Very low	No	No	Yes	No

TABLE II
QUALITATIVE COMPARISON OF VARIOUS REPROGRAMMING SCHEMES

and results in overall network instability and functional failure. For instance, in a recent deployment on Reventador Volcano, reboots led to 3-day network outage, reducing mean node uptime from over 90% to 69% [1], [2]. An exception is the Elon scheme. However, Elon only works for certain WSNs platform, e.g. TelosB. Our proposed QDiff scheme does not require mote restart and is independent of the WSNs platform and its operating system.

5) Code Overheads: A reprogramming scheme usually inserts new instructions, slop regions, jump tables and other translation schemes, which result in large program sizes. Maintaining lower code size increment is important as the mote has a limited available memory. The proposed QDiff reprogramming scheme also introduces overhead as additional jump calls. However, such overhead is insignificant as each of these jump instruction, and subsequent return instruction, take only 6 bytes. Furthermore, as the call-return instructions are only added for only few functions, the overall increment is, on average, within 3% of the total code memory.

6) Heterogeneity: Many different hardware platforms and operating systems are available for WSNs, with diverse specifications. All of the existing schemes, by design, can only support a subset of these platforms. For instance, some of the reprogramming schemes [1], [13] are specific to the TinyOS operating system, several others [14], [15] are specific to the Contiki operating system whereas some [1] are tied to a particular mote platform. However, the proposed QDiff is independent of the mote platform and its operating system. An overall qualitative comparison of Deluge, Rsync, Zephyr, Hermes, Elon and QDiff is shown in Table II.

VI. CONCLUSION

Existing reprogramming schemes transmit large amounts of information over the network, rewrite the whole internal flash memory and require large amount of external flash memory. Radio transmission and flash memory writing are two of the most energy- and time-intensive operations in wireless sensor networks. This paper describes, QDiff, a novel reprogramming scheme, which reduces the amount of information to be transmitted, reduces internal flash memory writing and eliminates necessity of external memory without modifying the compiler. Experiments show that QDiff reduces information size by a factor of 11 times and internal flash memory writing by a factor of 20 times than existing reprogramming schemes. QDiff achieves this efficiency by exploiting the strength of clone detection mechanisms and re-arranging global variables in the final executable file. QDiff works on various hardware and software platforms to support the application-specific nature of sensor networks.

ACKNOWLEDGEMENT

This research is funded by a grant from the Ontario Ministry of Economic Development and Innovation under the Ontario Research Fund-Research Excellence (ORF-RE) program.

REFERENCES

[1] W. Dong, Y. Liu, X. Wu, L. Gu, and C. Chen, “Elon: enabling efficient and long-term reprogramming for wireless sensor networks,” in *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2010, pp. 49–60.

[2] G. Werner-Allen, K. Lorincz, M. Welsh, O. Marcillo, J. Johnson, M. Ruiz, and J. Lees, “Deploying a wireless sensor network on an active volcano,” *IEEE Internet Computing*, vol. 10, pp. 18–25, 2006.

[3] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, “The nesc language: A holistic approach to networked embedded systems,” in *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI)*, 2003, pp. 1–11.

[4] R. K. Panta, S. Bagchi, and I. M. Khalil, “Efficient wireless reprogramming through reduced bandwidth usage and opportunistic sleeping,” *Ad Hoc Networks*, vol. 7, no. 1, pp. 42 – 62, 2009.

[5] R. K. Panta and S. Bagchi, “Hermes: Fast and energy efficient incremental code updates for wireless sensor networks,” in *Proc. IEEE INFOCOM*, 2009, pp. 639–647.

[6] R. Panta, S. Bagchi, and S. Midkiff, “Zephyr: Efficient incremental reprogramming of sensor nodes using function call indirections and difference computation,” in *Proc. of USENIX Annual Technical Conference*, 2009.

[7] J. Koshy and R. Pandey, “Remote incremental linking for energy-efficient reprogramming of sensor networks,” in *Proceeedings of the Second European Workshop on Wireless Sensor Networks.*, 2005, pp. 354 – 365.

[8] A. Tridgell and P. Mackeras., “The rsync algorithm. technical report available: http://samba.anu.edu.au/rsync/tech_report/tech_report.html,” Australian National University, Tech. Rep., 1998.

[9] J. Jeong and D. Culler, “Incremental network programming for wireless sensors,” in *Proc. First Annual IEEE Communications Society Conf. Sensor and Ad Hoc Communications and Networks (SECON)*, 2004, pp. 25–33.

[10] W. Dong, Y. Liu, C. Chen, J. Bu, and C. Huang, “R2: Incremental reprogramming using relocatable code in networked embedded systems,” in *Proceedings IEEE INFOCOM*, 2011, pp. 376–380.

[11] C. K. Roy, J. Cordy, and R. Koschke, “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach,” *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, May 2009.

[12] A. Raza, G. Vogel, and E. Plödereder, “Bauhaus–a tool suite for program analysis and reverse engineering,” *Reliable Software Technologies–Ada-Europe 2006*, pp. 71–82, 2006.

[13] W. Munawar, M. Alizai, O. Landsiedel, and K. Wehrle, “Dynamic TinyOS: Modular and Transparent Incremental Code-Updates for Sensor Networks,” *Proceedings of the IEEE ICC*, 2010.

[14] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, “Run-time dynamic linking for reprogramming wireless sensor networks,” in *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2006.

[15] W. Dong, C. Chen, X. Liu, J. Bu, and Y. Liu, “Dynamic linking and loading in networked embedded systems,” in *Proc. IEEE 6th Int. Conf. Mobile Adhoc and Sensor Systems (MASS)*, 2009, pp. 554–562.

[16] J. W. Hui and D. Culler, “The dynamic behavior of a data dissemination protocol for network programming at scale,” in *Proceedings of the 2nd international conference on Embedded networked sensor systems (SenSys)*, 2004, pp. 81–94.