Memory Safety for Embedded Devices with nesCheck

Daniele Midi
Dept. of Computer Science
Purdue University
West Lafayette, IN, USA
dmidi@purdue.edu

Mathias Payer
Dept. of Computer Science
Purdue University
West Lafayette, IN, USA
mpayer@purdue.edu

Elisa Bertino
Dept. of Computer Science
Purdue University
West Lafayette, IN, USA
bertino@purdue.edu

ABSTRACT

Applications for TinyOS, a popular operating system for embedded systems and wireless sensor networks, are written in *nesC*, a C dialect prone to the same type and memory safety vulnerabilities as C. While availability and integrity are critical requirements, the distributed and concurrent nature of such applications, combined with the intrinsic unsafety of the language, makes those security goals hard to achieve. Traditional memory safety techniques cannot be applied, due to the strict platform constraints and hardware differences of embedded systems.

We design *nesCheck*, an approach that combines static analysis and dynamic checking to automatically enforce memory safety on nesC programs without requiring source modifications. nesCheck analyzes the source code, identifies the minimal conservative set of vulnerable pointers, finds static memory bugs, and instruments the code with the required dynamic runtime checks. Our prototype extends the existing TinyOS compiler toolchain with LLVM-based passes. Our evaluation shows that nesCheck effectively and efficiently enforces memory protection, catching all memory errors with an overhead of 0.84% on energy, 5.3% on code size, up to 8.4% on performance, and 16.7% on RAM.

1. INTRODUCTION

Wireless Sensor Networks (WSNs) are deployed in critical, real-time applications, where availability and integrity are of paramount importance. WSN nodes are embedded systems that often manage confidential information, such as private keys and aggregated data, making confidentiality and integrity key requirements. However, the distributed and concurrent nature of WSN applications, together with the intrinsic type and memory unsafety of C/C++, make it hard to achieve these security goals.

TinyOS [21] is an open source operating system designed for low-power wireless embedded systems, such as WSN motes and smart meters [19]. TinyOS programs consist of separate software components statically linked through interfaces. Common components include routing and packet radio communication, sensor measurements, and storage. The language used to program TinyOS applications is *nesC*, a dialect of the C language optimized for the resource constraints of low-power embedded devices [13]. Because of the strict constraints in terms of memory, storage, and energy, neither TinyOS nor the underlying hardware provide any memory protection or virtual memory mechanism between processes or kernel and user-space. Moreover, the *nesC* language makes it easy to write memory-unsafe code, inheriting all the type and memory safety problems of C.

Memory corruption in the software running on a single node may allow an attacker to take over the node, read private data, or even disseminate incorrect data and degrade the entire network. Note that embedded platforms do not have code injection protection or ASLR, so a holistic defense like memory safety becomes even more important. More critically, since all the nodes run the same software image, an attacker may exploit a single vulnerability to take control of every node in the network. Concrete examples of such devastating attacks have been shown for Harvard-architecture-based sensor nodes such as the MicaZ motes [11], as well as Von Neumann-architecture-based ones such as the popular TelosB motes [15]. In these attacks, a well-crafted network packet sent to a vulnerable node can take control of the node and propagate as a self-replicating worm to the entire network through multi-hop communications [12, 14, 16, 32, 36]. All of these critical attacks would be prevented by enforcing memory safety.

Existing memory safety techniques [26, 24, 4, 29] are not applicable to embedded systems, nesC, or TinyOS, as they are designed for general-purpose systems. Adapting them to embedded systems is unfeasible without extensive redesign. In fact, they have heavy requirements in storage, memory, presence of an MMU, and hardware memory-protection mechanisms. However, embedded systems fundamentally differ from regular computing systems, having scarce memory, no MMU, no protection against code injection and code reuse, and often not even a distinction between kernel-space and user-space. For example, widespread sensor motes like the Memsic TelosB [23] only provide 10kB of RAM and 48kB of program flash memory; previously proposed memory safety approaches result in significantly bigger code size and more intensive memory usage. Moreover, the performance degradation that many existing solutions impose is not acceptable for energy-constrained, real-time WSNs applications. In fact, solutions such as CCured reported slowdowns ranging from 20% to 200% [26]. Given the resource

ASIA CCS '17, April 02 - 06, 2017, Abu Dhabi, United Arab Emirates

ISBN 978-1-4503-4944-4/17/04...$15.00

DOI: http://dx.doi.org/10.1145/3052973.3053014

constraints, a straightforward porting of common implementations for memory protection techniques to embedded systems is infeasible. A tailored solution for memory and type safety for TinyOS applications is therefore needed.

For TinyOS applications, the code for applications, libraries, and operating system is entirely available at compile time. This allows us to effectively leverage whole-program static analysis to ensure memory safety with a fallback to dynamic checking instrumentation if we run into aliasing issues. Moreover, by statically identifying and removing unnecessary checks for memory accesses that will never result in memory errors, it is possible to achieve low performance overhead.

Based on such considerations, we design *nesCheck*, a novel scheme that combines static analysis, type inference, and dynamic instrumentation techniques to enforce memory safety on embedded systems for existing nesC programs. The goal of nesCheck is to protect embedded software against memory vulnerabilities with negligible overhead and without requiring any source code modification. nesCheck statically analyzes the source code, identifies the potentially dangerous pointer variables, automatically infers the minimum set of dynamic runtime checks needed to enforce memory safety based on pointer access flow, and instruments the code appropriately.

On one hand, solutions that enforce memory safety entirely dynamically have resource requirements that cannot be satisfied on embedded platforms; on the other hand, static solutions based on whole-program analysis are not applied in practice as they lead to exponential state explosion. nesCheck is novel in tailoring its design to the challenges and constraints of the embedded world, where whole-program analysis is feasible but memory and performance overhead are of concern, by relying entirely on static analysis alone whenever possible and falling back on runtime protection only when needed. Instead of porting incremental defenses, our approach enforces low-overhead spatial memory safety for all code running on embedded systems.

To evaluate our approach, we implement nesCheck as a combined static analysis/dynamic checker on top of the LLVM compiler framework. The static analysis infers types and removes as many checks as possible while the dynamic checker enforces safety. We then integrate our checker into the existing nesC toolchain. We evaluate nesCheck on standard TinyOS application benchmarks, and show that it effectively enforces memory safety on WSN applications, while minimizing the runtime performance overhead (0.84% on energy, 5.3% on code size, up to 8.4% on performance, and 16.7% on RAM). These benchmarks are the standard benchmarks for evaluating WSN and present realistic usage scenarios for embedded systems. Porting full desktop benchmarks like SPEC CPU2006 is unrealistic due to the hardware and performance constraints of embedded devices (just the test input of a single benchmark is orders of magnitudes larger than all the available memory on the target platforms).

The contributions of our work are: (i) Design of an interprocedural whole-program static analysis mechanism, based on type tracking and pointer usage, without the need for programmer annotations; (ii) Design of dynamic instrumentation for efficient memory safety enforcement on highly constrained embedded platforms, without MMU or kernel/user space separation; (iii) Evaluation of efficiency and effectiveness of our approach through implementation prototype.

2. ADVERSARIAL MODEL

We assume that the attacker can inject and intercept arbitrary packets in the network. We also assume that the application has memory vulnerabilities known to the attacker. She will exploit them to take control of a node by means of code injection/reuse attacks or leak private information from the node. The attacker has the power to compromise the integrity, availability, or confidentiality of the node.

Physical attacks targeting the nodes, hardware attacks, or flashing/programming individual nodes with a malicious firmware are out of scope.

3. BACKGROUND

3.1 Memory Safety Vulnerabilities

The root cause of all memory safety vulnerabilities is the dereferencing of invalid pointers. There are two main categories of memory safety vulnerabilities: *spatial* memory safety vulnerabilities, resulting from pointers pointing to addresses outside the bounds of the allocated memory area, and *temporal* memory safety vulnerabilities, resulting from the usage of pointers after the corresponding memory areas are deallocated (e.g. *use-after-free* errors).

Our current prototype of nesCheck targets spatial memory safety, but can be extended to enforce temporal safety as well, by lock and key mechanisms [25]. However, as memory in well-developed WSN applications is allocated statically instead of dynamically, temporal safety errors are not a pressing issue for applications that comply with the development guidelines for TinyOS. This includes all the applications that ship with the standard distribution of TinyOS, as well as most larger-scale WSN applications. Examples of the memory vulnerabilities that nesCheck protects against are out-of-bounds accesses to pointers on the stack and heap, uninitialized uses, and null dereferencing.

3.2 TinyOS

nesC. nesC is an event-driven dialect of C. Its additional features include the concept that programs are built out of *components*, statically linked through interfaces.

Dynamic allocation. In the early versions of TinyOS, no dynamic memory allocation was allowed. This constraint, partially relaxed in recent releases, is still highly encouraged, as the lack of memory protection and separation can easily lead to involuntary stack smashing when the heap grows into the stack [37]. Specialized components (e.g., TinyAlloc), were introduced to support dynamic allocation, but behind the scenes they simply manage a large chunk of pre-allocated memory. Disabling dynamic allocation has the advantage, from a memory safety standpoint, that most required information is available at compile-time, and little work is left for dynamic detection.

Compilation and execution model. The standard TinyOS compilation pipeline is composed of several steps. First, the nesC code is processed and all the required components, including the operating system, are linked together. Under this model, all code, libraries, and OS components are statically known at compile time. The resulting single nesC code is cross-compiled to C code, in turn compiled natively into a binary image for the specific target platform. Such single binary image – containing both user code and OS code – runs as a single executable, assuming complete control over the hardware at all times. The memory address

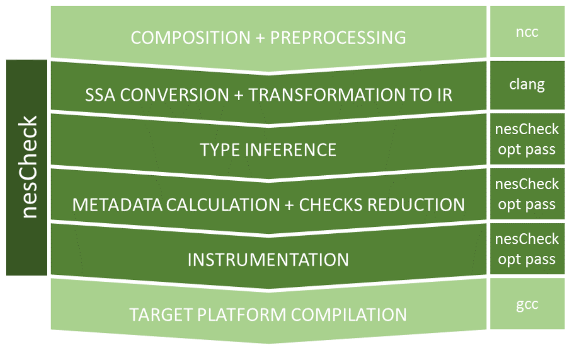

COMPOSITION + PREPROCESSING
ncc
SSA CONVERSION + TRANSFORMATION TO IR
clang
TYPE INFERENCE
nesCheck opt pass
METADATA CALCULATION + CHECKS REDUCTION
nesCheck opt pass
INSTRUMENTATION
nesCheck opt pass
TARGET PLATFORM COMPILATION
gcc
nesCheck

Figure 1: nesCheck pipeline (lighter blocks are existing steps of the nesC compiler toolchain).

space is shared among all components, both user and system code. For this reason, the official development guidelines for TinyOS recommend to (i) keep the state of the various components private, (ii) communicate only through exposed interfaces, and (iii) avoid transferring pointers between different pieces of code. All these characteristics of the TinyOS compilation and execution model make it a particularly good fit for static analysis

4. THE NESCHECK APPROACH

Figure 1 shows the architecture of the final pipeline for nesCheck. Our main memory safety goals are listed below. nesCheck performs both static bug detection – for memory accesses that will always result in a violation regardless of the execution path – and runtime bug catching – for memory accesses that could potentially lead to memory corruptions, depending on the execution flow. **(i) Bugs:** (Static) Find all *statically provable* memory bugs and report them as errors; **(ii) Vulnerabilities:** (Static) Find all *potentially* unsafe memory accesses, determine and exclude those that will never result in a memory corruption (in a conservative way), and report the remaining ones as warnings; **(iii) Checks:** (Dynamic) Instrument all remaining vulnerable locations with dynamic runtime checks, and catch all memory errors at runtime. We provide a proof sketch about nesCheck's memory safety in Appendix A

4.1 Static Analysis

nesCheck uses static analysis in order to enforce an extended type system on the pointer variables, and subsequently compute and propagate metadata for the vulnerable pointers. Our approach uses an inter-procedural whole-program analysis technique, carried out on the Static Single Assignment (SSA) form [9] representation of the code. In SSA form, each variable is written to at most once, introducing a fresh variable every time the value is updated with a destructive assignment operation. While the code is in SSA form, the heap remains in non-SSA form, meaning that the same memory location can be written to multiple times through the same and different pointers. Therefore, even though in SSA form each variable is only assigned once, a new value is assigned with a `store` operation to a memory location previously loaded with a `load` operation, making it possible to connect together different instructions operating on the same logical variable.

4.1.1 *Extended Type System and Type Inference*

In order to provide type safety, identify the potentially dangerous memory accesses, and avoid dynamic checks on the provably safe operations, it is necessary to understand the role played by the various pointers in the code and their interrelations. We thus enforce a type system inspired by CCured [26], that categorizes pointers according to their usage into different classes with specific characteristics. The pointer types that we consider are the following: **(i) Safe pointer to τ:** it can only be null or point to a value of type τ. At runtime, it may only need a null-pointer check. **(ii) Sequence pointer to τ:** like a Safe pointer, it can be null or point to a value of type τ. However, a Sequence pointer can also be interpreted as an integer, and be manipulated via pointer arithmetic. At runtime, it may need a null-pointer check, as well as a bounds check if casted to a safe pointer of base type τ. **(iii) Dynamic pointer:** it is a pointer that cannot be statically typed. At runtime, it may need null-pointer and bounds checks, as well as dynamic type checks.

The type inference engine gathers information from the source code to classify pointer declarations according to the extended type system. The engine focuses on all locations in which pointer variables are used and classifies them, in a fixpoint iteration, by analyzing their usage. Our type inference algorithm is shown in Algorithm 1.

The type inference algorithm uses 3 rules:

① All pointers are classified as *Safe* upon their declaration;

② Safe pointers subsequently used in pointer arithmetic are re-classified as a *Sequence*;

③ Safe or Sequence pointers interpreted with different types in different locations are re-classified as a *Dynamic*. This includes casting between different levels of indirection (e.g., `int**` to `int*`), and between different root types (e.g., `int*` to `void*`).

nesCheck's type inference engine effectively enforces a total ordering $Dynamic \prec Sequence \prec Safe$ on pointer types, so the type of a pointer is updated only if the new type is more restrictive. For example, assume the following code:

```
int *arr, *p, n;
arr = malloc(5 * sizeof(int));
n = (int)arr;
p = arr[3];
```

Algorithm 1: nesCheck's type inference algorithm

```
foreach declaration of pointer variable p do
    classify(p, SAFE);
foreach instruction I using pointer p do
    r ← result_of(I);
    if I performs pointer arithmetic then
        classify(p, SEQ);
        classify(r, SAFE);
    if I casts p to incompatible type then
        classify(p, DYN);
        classify(r, DYN);
```

The pointer `*arr` is classified as Safe upon declaration. When casted from `int*` to `int`, `*arr` is reclassified as Dynamic since $Dynamic \prec Safe$ holds according to the total ordering. However, when used in pointer arithmetic, the type of `*arr` is not changed as the total ordering constraint $Sequence \prec Dynamic$ is not satisfied.

Note that no extra rules are necessary for several non-obvious cases, because the analysis runs on SSA form. For instance, indirect calls (e.g., callbacks or function pointers) are classified as Dynamic by nesCheck's type inference because of the use of `void*` pointers. Another case includes pointers to pointers, or pointers to structs containing pointers. If the inner type is classified as Dynamic, the outer type must be classified as Dynamic as well. A concrete example of this is `int * q1 * q2`, where `q1` and `q2` are pointer kinds. If `q2` is Dynamic, then `q1` should also be Dynamic. The three rules presented suffice in correctly classifying these pointers, since an access to that pointer as a whole will result in two subsequent `load` instructions, that will propagate the Dynamic classification between the different levels of indirection.

After the type inference completes, all the pointers are classified. The rules guarantee that the final assignments are a conservative over-approximation, potentially classifying non-Dynamic pointers as Dynamic pointers, but never the opposite. This fundamental property ensures the correctness of the memory safety enforcement. The subsequent optimizations will compensate the potential performance degradation of conservative classification.

Type Inference Validation. We discuss the nesCheck type inference engine based on the example used in CCured [26]:

```
1  int **a;
2  int i;
3  int acc;
4  int **p;
5  int *e;
6  acc = 0;
7  for (i=0; i<100; i++) {
8      p = a + i;
9      e = *p;
10     while ((int) e % 2 == 0)
11         e = *(int**)e;
12     acc += ((int)e >> 1);
13 }
```

The program sums an array of "boxed integers", a data type with double interpretation: when odd, its 31 most significant bits can be interpreted as an integer, otherwise it represents the pointer to another boxed integer.

The expected behavior of the type inference engine is to classify `**a` as Sequence – since it is used in pointer arithmetic at line 8 – and `*e` as Dynamic – since it is casted and used with different types at different locations (i.e., as pointer at line 11 and as integer at line 12). All the other pointers should be classified as Safe. It is possible to verify that, according to our type inference algorithm, the correct classification of `**a` as Sequence pointer is achieved by the application of Rule ②, while `*e` is correctly classified as Dynamic by Rule ③ applied at line 12. All the other pointers are classified as Safe upon their declaration, by Rule ①, and their classification never changes.

Through this example and others constructed specifically to exercise unusual pointer usages, we verify that nesCheck correctly classifies all of the pointer types according to our extended type system.

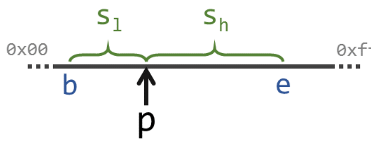
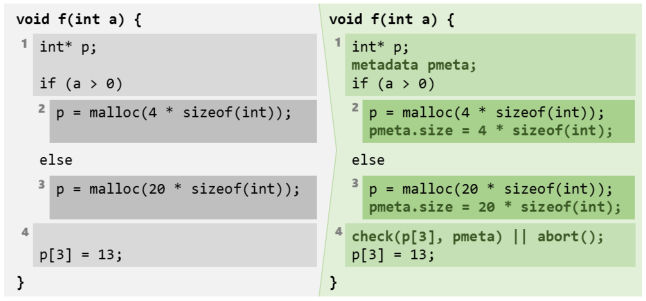


Figure 2: Comparison of bounds metadata in nesCheck to the traditional approach.



```
void f(int a) {
1  int* p;

   if (a > 0)
   2  p = malloc(4 * sizeof(int));

   else
   3  p = malloc(20 * sizeof(int));

4
   p[3] = 13;
}
```

```
void f(int a) {
1  int* p;
   metadata pmeta;
   if (a > 0)
   2  p = malloc(4 * sizeof(int));
      pmeta.size = 4 * sizeof(int);
   else
   3  p = malloc(20 * sizeof(int));
      pmeta.size = 20 * sizeof(int);
4  check(p[3], pmeta) || abort();
   p[3] = 13;
}
```

Figure 3: Explicit metadata variables.

4.1.2 Metadata Computation and Propagation

The metadata maintained by nesCheck for each pointer contains information about the memory area to which such pointer points. Differently from the traditional tracking of base b and bound e for each pointer, nesCheck's metadata includes the size of the areas towards both lower and higher memory addresses (denoted with s_l and s_h, respectively), with respect to the current address stored in the pointer variable. Figure 2 shows a graphical comparison of our metadata structure and the more traditional one. As an example, let `int* p` be a pointer to an array of 5 integers, and assume `int* p1 = &p[2]`. The metadata for `p` will be ($s_l = 0, s_h = 5 \cdot \texttt{sizeof(int)}$), while the metadata for `p1` will be ($s_h = (5-2) \cdot \texttt{sizeof(int)}, s_l = (5-2-1) \cdot \texttt{sizeof(int)}$). This construction simplifies bounds checking by checking only one "side" instead of two if we can infer the direction of a sequential pointer; for example, in a common scenario such as a monotonically increasing (or decreasing) loop, the compiler can safely remove one check.

nesCheck computes the metadata information for each pointer with different strategies, depending on the specific pointer. For static allocations, such as arrays of fixed size or pointers to `structs`, nesCheck directly computes the size of the allocated memory. While dynamic memory allocation is discouraged in TinyOS, nesCheck supports it for completeness. For dynamically allocated memory, the size is computed and updated by keeping track of the parameters of calls to functions such as `malloc()`, `realloc()`, `calloc()`, and `free()`. In cases where a local pointer can point to different memory areas depending on dynamic control flow conditions, nesCheck generates and injects an explicit variable to hold the metadata for this pointer, depending on the control flow paths. Figure 3 shows a concrete example of this scenario – with the original source code on the left and the instrumented one on the right – where function `f()` performs different allocations for pointer `*p` depending on the value of the function parameter `a`. Explicit metadata variables are needed for pointers accessed in basic blocks different than the one they were defined in. Detecting this behavior is possible as the heap is in non-SSA form. nesCheck is thus capable of connecting the same logical variable at the different locations (i.e., variable declaration and assignments in disjoint branches). In Figure 3, the different basic blocks are highlighted as separate, numbered solid boxes. `*p` is declared in block 1, but is initialized in block 2 or 3, and accessed in block 4.

4.1.3 Metadata Table

A Metadata Table associates specific memory addresses with their metadata information. Efficient data structures, e.g., hashmaps, often use large virtual address spaces [24]. Embedded devices do not have a virtual memory management mechanism; however, all the pointers that will need an entry in the Metadata Table are known at compile time, so nesCheck optimizes its data structure by using a dense, array-based Binary Search Tree. Moreover, for code that follows TinyOS's design guidelines and therefore does not make use of dynamic memory allocation, this data structure can be entirely preallocated for a statically-defined size.

We decouple metadata from the pointers – compared to *fat pointers* used in prior work [26, 17, 1, 10, 27, 35] – in order to achieve a uniform memory representation for all pointers. Moreover, since the search tree is, on average, tiny compared to the total number of pointers, keeping it separate allows nesCheck to choose the optimal data structure.

4.2 Dynamic Instrumentation

Dynamic checks can detect all memory errors since they have full runtime view and dynamic information when they are executed. In nesCheck, the metadata for each pointer is set to zero upon declaration, then always kept up-to-date with the actual offsets of the pointer in its memory area.

Every time a dynamic check is necessary, the respective memory access instruction is instrumented to be preceded by a bounds check. A failed check will terminate the execution and reboot the node, preventing memory corruptions. With no memory separation nor difference between kernel-land and user-land, continuing the software execution after a memory error can have unpredictable, arbitrarily bad outcomes. Rebooting is the only safe fault-handling action to prevent further memory corruption and potential compromising of the entire network on such constrained platforms. The attacker could try to exploit the same vulnerability again, and achieve at best a Denial of Service (DoS). Compared to probabilistic defenses, the attacker will never succeed against memory safety. In a debugging scenario, it would be possible to extend our prototype to send an error report message to the base station, including more details about the code location that caused the error. Our current prototype supports the explicit printing of details about the error location on screen when the code is run in a simulator (more about the TinyOS simulator in Section 6).

Frequent updates and lookups in the table incur high performance overhead. nesCheck optimizes by adding instrumentation to more directly propagate the metadata.

Functions taking pointers as parameters: A pointer appearing as a parameter in a function will assume different values for different callers of the function. Consequently, the pointer will also inherit different metadata properties depending on the pointer that is passed as actual parameter at every different call site. nesCheck enhances the signature of all the functions that have pointer parameters to include additional parameters for the metadata. Note that variadic functions are still supported by updates and lookups in the metadata table. As an example, a function with a signature such as `void f(int* p)` is enhanced to `void f(int* p, metadata pmeta)`, where `metadata` is the type of the data structure holding nesCheck's metadata information. Finally, the pointer parameter is associated with the metadata parameter as its own metadata.

Functions returning pointers: If a function returns a pointer, metadata propagation must also be enabled through the return value. nesCheck enhances the signatures of such functions and their `return` instructions, from a single value to a structure containing the original returned value plus its attached metadata. Thus, the sample function signature `int* f()` will be instrumented into `{int*, metadata} f()`. All the return instructions will consequently be transformed from `return p;` into `return {p, pmeta};`, where `pmeta` is the metadata information for pointer `p`. Lastly, all call sites for this function must be instrumented to take into account the change in return type, unpack the two pieces of data from the structure (i.e., the pointer and its metadata) and associate one with the other.

4.3 Running Example

In this section, we present the working of the core components of nesCheck on a program example (shown in Figure 4) that is small – for ease of detailed discussion and manual analysis of expected behaviors – but stress-intensive in the number of advanced features and memory error corner-cases included. We include casting of pointer types to and from integers, index-based access of memory areas, usage of pointers with incompatible types depending on specified conditions, and dynamic memory allocation as well, even if discouraged by TinyOS, to ensure the correctness of nesCheck even in face of wrong programming styles.

While the analysis and instrumentation of the program in nesCheck is carried out sequentially one entire function at a time, here we follow the execution flow for a more effective presentation. First of all, nesCheck rewrites the signatures for `testMT_aux` to `{foo_t*, meta_t} testMT_aux(int* p, meta_t pmeta)`, and instruments similarly `testMetadataTable` and `assignLoop`.

In `main()`, nesCheck infers the size for the metadata of `*arr` to be 5 integers, from the parameter of `malloc()`. The subsequent call to `testMetadataTable()` is then updated for its new signature (adding as second parameter the metadata for `*arr`), avoiding the need for metadata table accesses.

The `testMetadataTable` function initially leverages the support function `testMT_aux` for obtaining a pointer to `struct foo_t`, using the characteristic TinyOS pattern of global variables in place of dynamic allocation. The field `f->bar` is aliased to `*p`, and this time the metadata propagation requires metadata table accesses, as the pointer is in a struct. The execution resumes in the `testMetadataTable` function. The storing of a numerical value inside the array member of the `struct foo_t bla` at line 14 is actually translated by Clang into a sequence of `GetElementPtr` statements. Whenever necessary, such instructions are instrumented by dynamic runtime checks and metadata table lookups.

Following the execution, the function `testDynamically-`

```
 1 typedef struct foo {
 2  int   a;
 3  int* bar;
 4 } foo_t;
 5 foo_t myfoo;
 6
 7 foo_t* testMT_aux(int* p) {
 8  foo_t* f = &myfoo;
 9  f->bar = p;
10  return f;
11 }
12 void testMetadataTable(int* p) {
13  foo_t* f = testMT_aux(p);
14  (f->bar)[2] = 13;
15 }
16 void assignLoop(int* p) {
17  int i;
18  for (i = 0; i < 4; i++)
19   *(p + i) = i;
20 }
21 void testDynamicAliasing(int n) {
22  int* p;
23  int a[4];
24  int b[12];
25  if (n < 1) p = a;
26  else       p = b;
27  assignLoop(&(p[1]));
28 }
29 int main() {
30  int* arr = malloc(5 * sizeof(int));
31  testMetadataTable(arr);
32  testDynamicAliasing(0);
33 }
```

Figure 4: Representative example for the stress-intensive microbenchmark.

`Aliasing`, conceived to stress-test common dynamic aliasing scenarios, is first instrumented with explicit metadata variables, as presented in Section 4.1.2. Then, `assignLoop()` tries to assign numeric values to the first 4 elements of the array, resulting in an out-of-bounds memory violation. However, an injected dynamic runtime check at line 19 will catch the out-of-bounds access to the 4^{th} element of the array, and the execution will be diverted to a trap function.

5. IMPLEMENTATION

The implementation of nesCheck leverages the existing TinyOS compiler toolchain and extends it with custom components built on Clang [5] and optimization passes from the LLVM suite [18]. The technologies used are highlighted next to each pipeline block in Figure 1.

The nesC source code is initially processed by *ncc*, the nesC compiler, that links the different nesC components together through their interfaces and translates the result to a single C source code file. The C source is then transformed into the LLVM Intermediate Representation (IR) language. Such IR is a well-specified code representation offering an abstraction layer between the source programming language used (nesC/C) and the actual target platform code. Then, the IR is passed to our nesCheck *Static Analyzer*, based on an LLVM target-independent Optimization Pass.

The nesCheck *Analysis State Manager* component maintains the analysis state throughout the different steps, and propagates information between the various components. Most of the metadata is kept in memory by the Analysis State Manager, and looked up and injected only when needed for the appropriate instrumentation.

As a last step, the minimal set of required runtime checks for the memory-manipulating instructions is computed, and the code is instrumented accordingly. The LLVM IR uses, in general, two separate instructions for pointer dereferencing: a `GetElementPtr` instruction to calculate the memory address of the location to be accessed, and a `Load` or `Store` instruction to actually access this memory location and, respectively, place the resulting value in a variable or store a value into the location. nesCheck's instrumentation adds a bounds check conditional branch before the `GetElementPtr` instruction, and a trap function to be invoked whenever the runtime check fails, to terminate the execution and reboot the node, preventing memory corruptions.

Whenever nesCheck statically determines that any execution of the instruction being instrumented will result in a failure of the check – i.e., the condition can be statically determined to be always false – the user is alerted that a constant memory bug is present, providing her with insights useful to inspect and fix the bug.

The rest of the pipeline, after the instrumentation, resumes the original TinyOS compilation toolchain, having the instrumented code go through the *gcc* compiler to obtain the final native binary for the desired target platform.

6. EVALUATION

The TinyOS development platform ships with several sample applications, such as radio communication, sensing, hardware interaction. As done by most other TinyOS research works [8, 4, 3, 29, 20], we use these applications as benchmark suite for evaluating nesCheck. In our experiments, we instrument all executed code, including that of the TinyOS operating system itself. Table 1 provides details on each program in our benchmark suite. We first use these applications *as-is* to evaluate the performance overhead. Then, we evaluate the overall effectiveness of nesCheck by randomly injecting memory bugs in the benchmark applications and verifying that all of them are caught statically or at runtime.

We evaluate nesCheck on several static metrics – such as the number of pointer variables, their inferred type classification, and the number of dynamic check instrumentations – and dynamic metrics – such as the overhead of nesCheck in terms of program size, memory, execution performance, and energy consumption.

To evaluate performance, we compiled the applications for TOSSIM [20], a discrete event simulator, *de facto*-standard tool for TinyOS WSNs. TOSSIM simulates the behavior of TinyOS accurately down to a very low level and precisely times interrupts. This allowed us to perform the evaluation in a controlled environment, through repeatable experiments, and to increase the number of runs for each experiment, while still maintaining a realistic distributed embedded software execution. Each of the evaluation results has been obtained by averaging 25 independent runs of each test.

Type Inference. The results in Figure 5 show that, on average, 81% of the variables are classified as Safe, 13%

Application	LOC	Description
BaseStation	5684	Simple Active Message bridge between the serial and radio links.
Blink	5505	Blinks the 3 LEDs on the mote.
MultihopOscilloscope	11728	Data collection: samples default sensor, broadcasts a message every few readings.
Null	4261	An empty skeleton application, useful to test the build environment functionality.
Oscilloscope	6868	Data collection: radio broadcasts a message every 10 readings of default sensor.
Powerup	4306	Turns on red LED on powerup, to test deploy of app on hardware mote.
RadioCountToLeds	6751	Broadcasts a 4Hz counter and displays every received counter on the LEDs.
RadioSenseToLeds	6808	Broadcasts default sensor readings, displays every received counter on the LEDs.
Sense	5699	Periodically samples the default sensor and displays the bottom bits on the LEDs.

Table 1: TinyOS standard applications used as benchmark for nesCheck's evaluation.

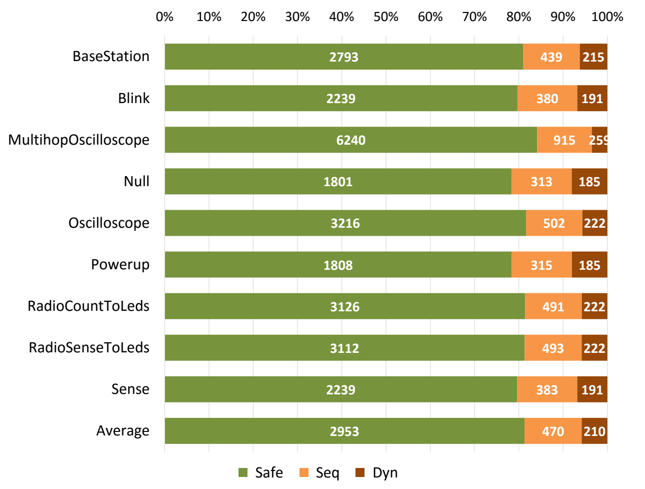


Figure 5: Pointer classification results for the TinyOS sample apps benchmark.

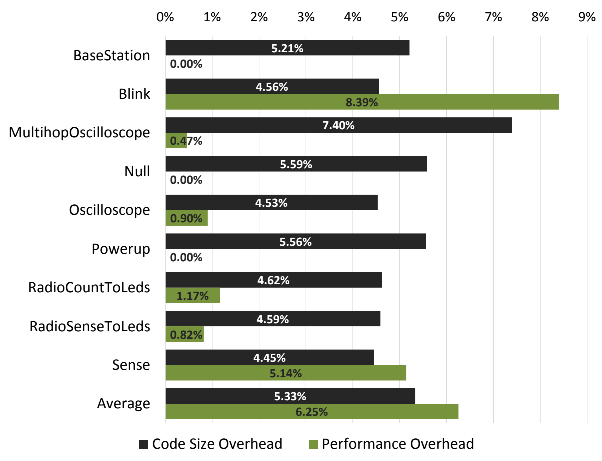


Figure 6: Code size and performance overhead for the instrumented TinyOS apps, including TOSSIM.

as Sequence, and 6% as Dynamic. A large number of dynamic runtime checks can thus already be skipped as immediate consequence of the type system inference. Note that, since the analysis is conservative, some pointers classified as dynamic might not be so; however, as shown in the performance evaluation afterwards, this does not degrade the efficiency of our approach.

The average total number of analyzed variables, across all the TinyOS sample applications in the benchmark, is 3,633, a small number that further supports our design choice of whole-program static analysis.

Code Size and Performance Overhead. We investigate the overhead of nesCheck's instrumentation in terms of code size and performance, and the results are shown in Figure 6. The programs in the benchmark total to 57,610 lines of code. The size overhead is measured in additional bytes of the memory-safe executable produced by nesCheck vs. the uninstrumented one, both including the code for the TOSSIM simulator infrastructure. The code size of the uninstrumented programs averages to 228,761 bytes, and the instrumentation adds only 12,201 bytes (5.3%) of overhead on average. This result shows that nesCheck is suitable for the instrumentation of programs to be deployed even on devices very constrained in ROM.

We also measure the performance overhead of nesCheck through the TOSSIM simulator for TinyOS. This tool is used by a simulation driver program by repeatedly asking it to execute the next event from the simulation queue. The duration of each event and the total number of events depend on the complexity of the computation to be executed. Therefore, we measure the overhead of nesCheck's instrumentation by fixing the total simulation time to 30 real seconds, running the simulation of the original and instrumented applications, and then measuring the number of simulated seconds actually executed. In three cases (`BaseStation`, `Null` and `PowerUp`), since the applications are merely sample "skeleton" programs to guide developers, no real events were happening after the initial program startup. Therefore, for those programs the reported overhead is 0, and we do not consider them in our averages for the performance overhead. For all the other applications that continuously process events, we observe that TOSSIM executes more simulated seconds (in the span of 30 real seconds) for simpler programs than for more complex ones. For example, the simple `Blink` program is executed for 120185.86 simulated seconds, while the more complex `RadioSenseToLeds` program only reaches 6878.08 simulated seconds (both uninstrumented). In fact, this confirms the intuition that fewer events can be processed in the same time span when the computation of each event is more complex. On average, nesCheck introduces a performance overhead of 6.2%. We note that the maximum overhead (incurred by the `Blink` application) is still quite low, at 8.4%. We believe that this overhead is acceptable for WSN applications.

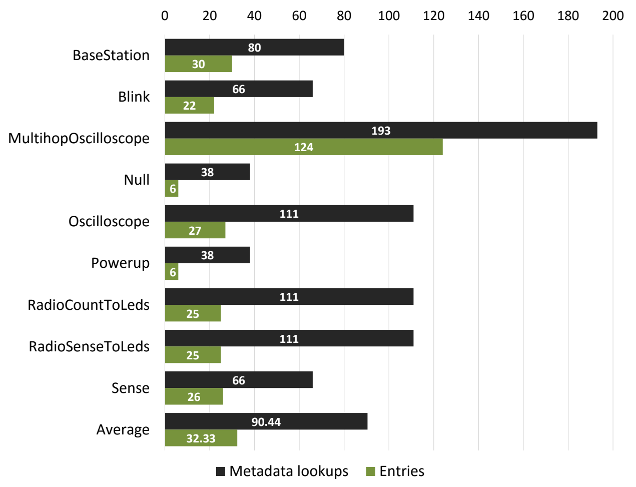


Figure 7: Metadata table entry lookups vs. actual metadata table entries required by the instrumentation.

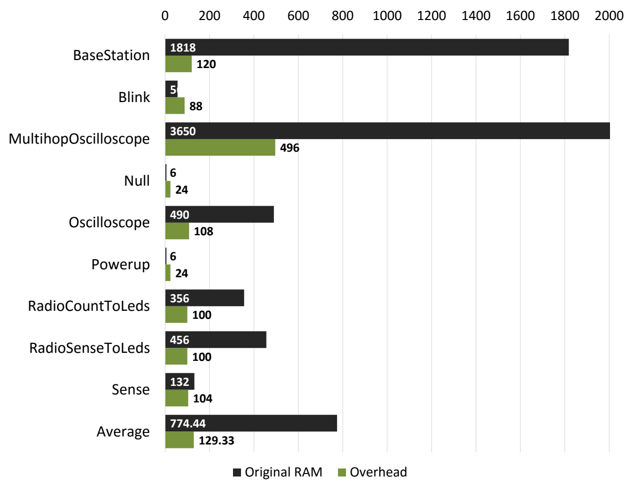


Figure 8: RAM occupation of uninstrumented programs and memory overhead of nesCheck (all in bytes).

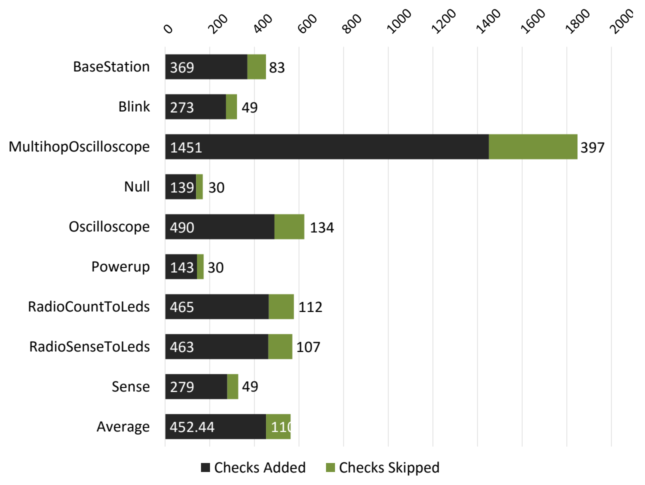


Figure 9: Checks added and checks skipped in the instrumented TinyOS sample apps benchmark.

Memory Overhead. As discussed in Section 4.1.3, some of the pointers require entries in a separated metadata table. We thus measure the impact of this additional data on the memory of the embedded devices. Figure 7 and Figure 8 present our results on the memory overhead of nesCheck for the TinyOS applications benchmark. In particular, Figure 7 shows the number of metadata lookups added to the code and the number of actual metadata table entries required for each application. On average, nesCheck added only 90 metadata table entry lookup instrumentation points during the instrumentation. Given the SSA form, there is a direct relationship between the number of memory accesses and the number of analyzed variables; therefore, we compare the number of metadata lookups with the total number of variables analyzed by nesCheck, and see that it amounts to just 2%. When only comparing to the Dynamic pointers, it amounts to 41%, which still represents a significant memory saving. Many of such lookups, furthermore, refer to the same logical variable, and thus point to the same entry in the metadata table. Thus, in fact, only 32 distinct entries are needed on average in the metadata table, constituting approximately 1/3 of the total lookup instrumentations for each program. With these collected metrics, we measure the effective RAM overhead of nesCheck for each application by comparing the RAM occupation of the uninstrumented program – as reported by the nesC toolchain when compiling for the TelosB motes platform [23] – with the size of the metadata table in the instrumented version – representing the effective memory overhead. Figure 8 presents both these metrics side by side for ease of presentation. The numbers vary greatly for the different applications, as the number of metadata table entries is completely dependent on the data structures used by each program. However, the average overhead is 16%, and in all cases the total memory requirement remains significantly below the 10kb RAM limit of the TelosB platform chosen for this experiment.

Checks Reduction. As part of our experimental analysis, we collected statistics about the number of runtime checks added to the programs during the instrumentation, together with the checks that are removed as part of nesCheck's check reduction. This metric includes the reduction both due to pointer kind classification and our additional analysis of pointer usage, which leads to removing additional checks whenever memory accesses can be statically verified as safe. As shown in Figure 9, the complete analysis and instrumentation of nesCheck for all the TinyOS applications overall reduces, on average, the required checks by 20% of the total potentially vulnerable locations, greatly reducing the performance overhead in enforcing memory safety. For the whole benchmark suite, an average of 452 checks are added, and 110 are skipped.

Energy Overhead. The power consumption for the various operations – such as computation, radio communication, standby or sleep – varies across the different sensor mote hardware platforms. However, on all platforms, the majority of the power consumption is always caused by wireless transmission and reception, as well as the transitions between the on and off states of the radio. Shnayder *et al.*, for example,

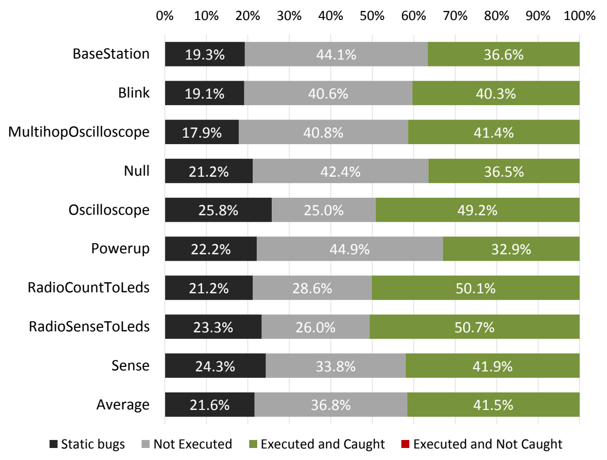


Figure 10: Fault injection results on TinyOS benchmark.

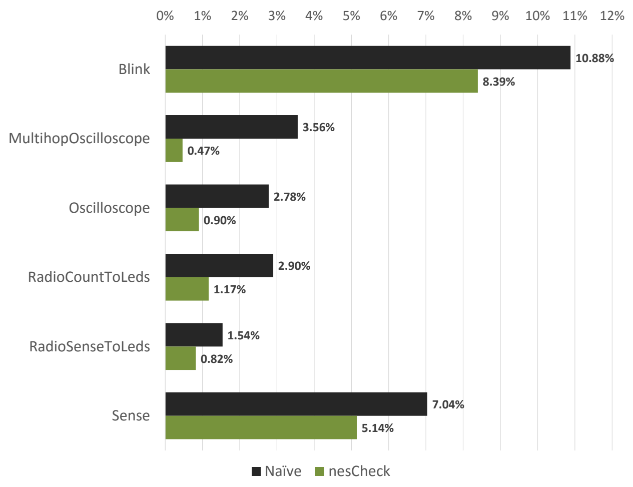


Figure 11: Naive vs. optimized instrumentation on TinyOS benchmark.

quantitatively measure that, in many cases, active CPU cycles in WSN applications are very small, and have negligible effect on total power consumption [30]. The instrumentation of nesCheck in TinyOS programs does not introduce any additional radio communication, while instead adding some runtime computation for the dynamic checks. Therefore, the energy overhead is, intuitively, proportional to the performance overhead that we measured in our experiments by a factor of CPU energy consumption.

Since measuring the energy consumption directly on the motes' hardware is difficult [33], to quantify this metric we leverage the energy model proposed by Polastre *et al.* [28]. We refer to the MicaZ motes hardware platform datasheet [22] (being the platform simulated by TOSSIM), and multiply the battery voltage by current draw and time. With those calculations, the energy overhead for nesCheck amounts on average to 0.84%, a negligible quantity that supports our analytical expectations.

Fault Injection. To evaluate the effectiveness of nesCheck in preventing all memory errors, we randomly injected memory vulnerabilities and bugs in the TinyOS applications. We injected 500 random faults in each applications in the TinyOS benchmark, for a total of 4, 500 faults. In particular, each time we selected one random memory access, altered its indices to produce a memory error, and included an extra printing instruction to mark the moment when the memory fault occurs; we then instrumented the application and executed it. We expected nesCheck to correctly find the fault, either statically or at runtime, and prevent the out-of-bounds access. Figure 10 shows our results. On average, 22% of the injected faults were statically caught at compile time. 37% of the faults were injected in areas of the code that were not executed at runtime. For the injected faults that were executed at runtime (41% on average), 100% were correctly caught by the dynamic checks placed by the nesCheck's instrumentation, i.e., no fault occurred and went uncaught.

Naive vs. Optimized Approach. While a direct comparison of nesCheck with traditional techniques such as SoftBound or CCured is infeasible due to (i) constraints of embedded systems, and (ii) the missing implementation of SoftBound or CCured for embedded systems, we measure the performance benefits of nesCheck's check reduction to get an estimate of the improvement over those traditional techniques. We run nesCheck with ("optimized") and without ("naive") check reduction optimizations, and run it on all the applications in the benchmark (excluding those that did not yield events in our performance overhead evaluation) Figure 11 shows a comparison of the overhead of the naive and optimized executions of the instrumented programs. We observe an overhead reduction of 41.13% on average, showing how nesCheck's check reduction effectively leads to significant performance improvements.

7. LIMITATIONS

Currently, a sensor node instrumented with nesCheck is rebooted when a dynamic check fails. Since this might not always be the best option, in the future, we plan to work on more advanced, programmer-guided recovery mechanisms, with the goal of maintaining the network as functional as possible even in the presence of memory errors.

An attacker could attempt to repeatedly trigger a memory error, which nesCheck's protection would counter by rebooting the software, to achieve DoS. However, nesCheck's memory safety guarantees ensure that a memory safety error cannot be used for malicious intents (except for DoS).

More powerful computing platforms (e.g., Raspberry PI) are becoming increasingly available. However, they are impractical for common WSN application purposes, with significantly higher cost, energy requirements, and size, as compared to low-power WSN nodes. The latter have the advantages of being cheap, easily replaceable, deployable in bulk, and in need of little energy. Even when such more advanced devices will become sufficiently cost-effective for large deployments, efficiency would still remain a critical concern for memory safety techniques, as the number and scale of applications deployed on them would consequently increase as well. We plan to work in this direction to investigate how nesCheck can be ported to more powerful platforms, and leverage the additional capabilities of these platforms to further improve performance.

Our evaluation did not find instances of memory bugs in TinyOS – a reasonable outcome since the code base is small and has been widely used for several years (i.e., bugs were fixed), while at the same time seeing very few modifications (i.e., no new bugs). Also, at runtime, nesCheck evaluates only the executed code paths, not all possible code paths.

The current prototype of nesCheck enforces spatial memory safety. Our approach could, however, be extended to also enforce temporal memory safety. Note that most WSN applications do not use dynamic memory allocation, and are therefore fully protected by spatial safety alone. Nevertheless, we plan to extend our implementation of nesCheck to explicitly address temporal safety, and design mechanisms tailored for embedded platforms to enforce it.

When determining the set of vulnerable pointers, (a) false negatives (i.e., pointers marked as Safe when they are not) cannot occur, as nesCheck is conservative in case of ambiguity, and (b) false positives (i.e., pointers not marked as Safe when they are) do not compromise the security invariants but only cause performance degradation. Our experimental analysis shows that nesCheck's overall overhead is small.

Lastly, the scalability of the system, and further overhead reduction, are of great importance. We plan to investigate whether the integration of Bounded Model Checking techniques [2] in nesCheck helps in that direction, as it would enable the use of formal verification techniques for proving the safety of seemingly dangerous memory accesses, therefore further reducing the overhead. Note that there are several issues that make formal verification on embedded software hard. Several patterns – such as direct communication with hardware registries for sensing, network packets, frequent interrupts, or the use of bit fields – cause the search space for formal verification to quickly explode. Dynamic checks are able to cope with these patterns, at the price of performance.

8. RELATED WORK

Memory safety is an ongoing research topic [34]. Attacks to WSN software through memory vulnerabilities have been widely investigated. Against common belief that Harvard-architecture devices would prevent code injection attacks, Francillon *et al.* [11] showed a detailed exploit for code injection without size limitation through carefully crafted network packets. Giannetsos *et al.* carried out a similar study [15], targeting Von Neumann-architecture devices. These two works cover most common architectures for WSNs, able to exploit, for example, both MicaZ and TelosB motes.

From the defense point of view, research work has typically taken three different directions: runtime protection, formal analysis and symbolic execution. nesCheck uses an approach that enhances the runtime protection class of mechanisms with static analysis techniques.

Runtime protection. Necula *et al.* introduce an extended type system for CCured [26]. CCured uses pointer classification as a static analysis technique to infer safe pointers that do not need bounds checks; however, it instruments all non-safe pointers in the code with runtime checks, potentially generating many unnecessary checks. nesCheck overcomes this issue by leveraging more extensive static analysis techniques to conservatively detect whether some of the sequence pointers can be left unchecked too, as well as detecting statically-recognizable memory violations. SoftBound [24] is a compile-time approach that instruments C code to enforce spatial memory safety by (i) keeping track of the properties of each the pointed memory area, and (ii) wrapping each memory access with a bounds check. Its design is geared towards platforms with large amounts of memory and virtual memory mechanisms, as it needs to maintain metadata about each pointer at runtime. Such resources are not available on constrained embedded devices. nesCheck leverages static analysis (see Section 4.1) to minimize the number of pointers whose metadata is managed in-memory at runtime, replacing the global metadata with local stack variables or conservatively removing the metadata completely (for specific pointers). While nesCheck too leverages dynamic runtime checking to enforce memory safety, it tailors and optimizes this approach to the specific characteristics of nesC applications in order to improve performance. Compared to the notable solutions just discussed, as well as other traditional ones, nesCheck works for embedded software, being designed specifically for their constraints, challenges, and advantages. One of the most relevant approaches for memory protection in WSN applications – and TinyOS in particular – is Safe TinyOS [8]. Cooprider *et al.* investigate issues related to the implementation of memory protection for TinyOS programs by formalizing the problem and the requirements, and developing optimizations that make runtime checks more viable under the strict performance constraints of WSN software. Safe TinyOS relies on the Deputy source-to-source compiler [7] to infer necessary information for the code instrumentation. Note that the Deputy project is no longer maintained. Safe TinyOS, however, puts much of the analysis burden on the programmers, requiring them to either annotate the code with specific type definitions and safety guidelines, or to declare entire components as "trusted" and therefore skipped by the tool. nesCheck, on the other hand, automates the entire process, with no need for source code modifications. Also, nesCheck reduces the potential runtime overhead by removing unnecessary checks before the instrumentation.

Formal analysis. Bucur *et al.* [4] propose a source-to-source transformation tool to make TinyOS code processable by the CBMC [6] bounded model checking [2] proving tool. The well-known limitations of formal verification, in particular the search space explosion, are inherited by this approach too. Even though Bucur *et al.* propose several optimizations to reduce the complexity to be handled, large-scale applications can still suffer by long times for analysis and potential undecidability if the state becomes too big to be handled.

Symbolic execution. Sasnauskas *et al.* [29] build an approach on top of the Klee symbolic execution framework to debug TinyOS applications before deployment. Just like for the formal analysis-based approaches, the bottlenecks for these designs are: (i) the need for a good model definition of the application to be tested, and (ii) the rapid explosion of the search state. If either part of the design results in a non-complete coverage of every possible vulnerability, then not all the bugs can be effectively identified. Conversely, since nesCheck leverages runtime checks for all the memory accesses that cannot be statically proven as safe, in a conservative way, nesCheck is guaranteed to always catch all the potential vulnerabilities and prevent memory corruption.

Hardware. Francillon *et al.* [12] propose a hardware modification to split the stack in a control flow stack and a data stack. While this is an interesting idea, it would require hardware manufacturers to change the platform (an economically burdensome path unlikely to be pursuable).

nesCheck's software-only approach does not require changes to the hardware platform, nor to the source code.

9. CONCLUSIONS

We presented nesCheck, an approach that combines whole-program static analysis and dynamic checking techniques to efficiently enforce memory safety on nesC programs, without requiring any source modification. nesCheck implements techniques to determine the presence of static memory bugs in the code, as well as to instrument it with the required set of runtime checks. It focuses on minimizing the overhead for the dynamic checks, considering the strict constraints of embedded systems. Through extensive evaluation benchmarks – both on specifically constructed programs and the standard TinyOS applications – we showed the effectiveness of nesCheck in enforcing memory protection while minimizing the runtime performance overhead (0.84% on energy, 5.3% on code size, up to 8.4% on performance, and 16.7% on RAM).

Acknowledgments

We thank our shepherd Lucas Davi and the anonymous reviewers for their insightful comments, as well as Scott A. Carr, Nathan Burow, and Gregory Essertel for helpful discussions about different aspects of the work. The work reported in this paper has been supported, in part, by the National Science Foundation under grants CNS-1547358, CNS-1513783, and CNS-1657711, and by Purdue PLM Center.

10. REFERENCES

[1] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *ACM PLDI*, 1994.

[2] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. *Symbolic model checking without BDDs*. Springer, 1999.

[3] D. Bucur. Intelligible tinyos sensor systems: Explanations for embedded software. In *Modeling and Using Context*, pages 54–66. Springer, 2011.

[4] D. Bucur and M. Kwiatkowska. On software verification for sensor nodes. *Journal of Systems and Software*, 84(10):1693–1707, 2011.

[5] Clang. clang: a C language family frontend for LLVM. http://clang.llvm.org/.

[6] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ansi-c programs. In *TACAS*. Springer, 2004.

[7] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *Progr. Languages and Systems*. Springer, 2007.

[8] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient memory safety for tinyos. In *SenSys*. ACM, 2007.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 1991.

[10] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. In *ASPLOS*, 2008.

[11] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *ACM CCS*, 2008.

[12] A. Francillon, D. Perito, and C. Castelluccia. Defending embedded systems against control flow attacks. In *1st ACM workshop on Secure execution of untrusted code*, 2009.

[13] 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 *ACM SIGPLAN PLDI*, 2003.

[14] T. Giannetsos and T. Dimitriou. Spy-sense: Spyware tool for executing stealthy exploits against sensor networks. In *ACM HotWSN Workshop*, 2013.

[15] T. Giannetsos, T. Dimitriou, I. Krontiris, and N. R. Prasad. Arbitrary code injection through self-propagating worms in von neumann architecture devices. *Comput. J.*, 53(10):1576–1593, Dec. 2010.

[16] T. Goodspeed. Stack overflow exploits for wireless sensor networks over 802.15.4, 2008.

[17] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference*, 2002.

[18] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, 2004.

[19] P. Levis. Experiences from a decade of tinyos development. In *USENIX OSDI*, 2012.

[20] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: Accurate and scalable simulation of entire tinyos applications. In *SenSys*. ACM, 2003.

[21] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. Tinyos: An operating system for sensor networks. In *Ambient Intelligence*, 2005.

[22] Memsic. Micaz datasheet. http://www.memsic.com/userfiles/files/Datasheets/WSN/micaz_datasheet-t.pdf.

[23] Memsic. Telosb datasheet. http://www.memsic.com/userfiles/files/Datasheets/WSN/telosb_datasheet.pdf.

[24] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. In *ACM Sigplan Notices*, pages 245–258. ACM, 2009.

[25] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Cets: Compiler enforced temporal safety for c. *SIGPLAN Notices*, 45(8):31–40, June 2010.

[26] G. C. Necula, S. McPeak, and W. Weimer. Ccured: Type-safe retrofitting of legacy code. *ACM SIGPLAN Notices*, 37(1), 2002.

[27] H. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in c programs. *Softw. Pract. Exper.*, 27(1):87–110, Jan. 1997.

[28] J. Polastre, J. Hill, and D. Culler. Versatile Low Power Media Access for Wireless Sensor Networks. In *ACM SenSys*, pages 95–107, 2004.

[29] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle. Kleenet: discovering insidious interaction bugs in wireless sensor networks before deployment. In *ACM/IEEE IPSN*, 2010.

[30] V. Shnayder, M. Hempstead, B.-r. Chen, G. W. Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *ACM SenSys*, 2004.

[31] Softbound website. http://www.cis.upenn.edu/acg/softbound/.

[32] B. Sun, D. Shrestha, G. Yan, and Y. Xiao. Self-propagate mal-packets in wireless sensor networks: Dynamics and defense implications. In *IEEE GLOBECOM*, 2008.

[33] V. Sundaram, P. Eugster, and X. Zhang. Prius: Generic hybrid trace compression for wireless sensor networks. In *ACM SenSys*, 2012.

[34] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, 2013.

[35] W. Xu, D. C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of c programs. In *ACM FSE*, 2004.

[36] Y. Yang, S. Zhu, and G. Cao. Improving sensor network immunity under worm attacks: a software diversity approach. In *ACM MobiHoc*, 2008.

[37] R. Züger. Paging in tinyos, 2006.

APPENDIX

A. PROOF OF SAFETY

In this section, we sketch a formal proof of memory safety for nesCheck. First, we give the intuition of the rules for type inference. Then, we follow the general structure of the proof of SoftBound [31], while focusing on the features relevant for nesCheck. We tackle the complexity of the nesC language by focusing the proof on an abstract subset of nesC that captures most of the fundamental primitives. We keep the formalism, operational semantics tractation and proof short, while still remaining sound in showing safety.

A.1 Grammar and Operational Semantics

The syntax we use models programs in their processed IR form, already reduced to atomic data types (`int` and pointers) and simple operations. Table 2 shows the grammar we consider for our proof. We use *LHS* and *RHS* to denote left-hand side and right-hand side, respectively. Note that, while most WSN applications do not use dynamic memory allocation, we include it in our formal grammar for the sake of generality. In our simplified operational semantics, we consider an environment E that models the stack with a map S from variable names to addresses and types, models the type inference with a map Γ from variable names to pointer categories, and models the heap with a partial map M from addresses to values.

Atomic Types	t	::=	`int` | p*
Pointer Types	p	::=	t | s | `void`
Struct Types	s	::=	`struct{`f `;` f`}`
Struct Fields	f	::=	(*id*:*t*)
LHS Expr.	*lhs*	::=	x | **lhs* | *lhs*.*id* | *lhs*->*id*
RHS Expr.	*rhs*	::=	*val* | *rhs*+*rhs* | &*lhs* | (*a*)*rhs*
		|	`sizeof(`p`)` | `malloc(`*rhs*`)` | *lhs*
Commands	c	::=	c ; c | *lhs* = *rhs*

Table 2: Grammar used in the formal proof of safety.

Using `some` and `none` to denote presence or absence of a value, we model nesC's memory access primitives as follows: (i) `read` M l: if l is an allocated memory location, return `some`, otherwise return `none`; (ii) `write` M l v: if l is an allocated memory location, set the content to the value v; (iii) `malloc` M s: if M has an available region of size s, allocate and return it, otherwise fail.

The normal C operational semantics processes assignments by writing the result of the expression in the RHS operand to the address calculated from the LHS operand. For this proof, we extend the traditional operational semantics of C by including new outcomes for operations (that include memory errors) and tracking of pointers metadata. A result r can therefore be: (i) $v_{(sl,sh)}$, a value v with the attached metadata for the size of the memory region towards the lower (sl) and higher (sh) memory addresses (see Section 4.1.2); (ii) a memory address l; (iii) `Success`; (iv) `MemoryError` if a bounds check failed; (v) `MemoryExhaustion` if M did not have enough free memory upon a `malloc` operation.

Using the above definitions, we formalize nesCheck's operational semantics with four classes of rules. First, the rules for type inference and propagation. Second, the $(E, lhs) \Rightarrow_l r : a$ rule specifies how LHS expressions are evaluated (no changes to the environment). Third, the $(E, rhs) \Rightarrow_r (E', r : a)$ rule specifies how RHS expressions are evaluated (potential changes to the environment; if successful, r is $v_{(sl,sh)}$). Lastly, $(E, c) \Rightarrow_c (E', r : a)$ is the rule to execute commands (r must be a success or failure result). Here we omit rules straightforwardly representing standard C semantics, and just show the rules most relevant for nesCheck's semantics.

A.2 Type Inference

We present some of the rules for type inference, that formalize the rules presented in Section 4.1.1. For example, pointer arithmetic on a Safe or Sequence pointer causes the result to be of Sequence kind, while casting a Safe or Sequence pointer to an incompatible type (Section 4.1.1 defines two types as "incompatible" when, for example, they have different levels of indirection or have same level of indirection but different root types) results in a Dynamic pointer:

$$\text{T}\,\frac{\Gamma(x) = \tau \qquad \tau \in \{\texttt{Safe}, \texttt{Seq}, \texttt{Dyn}\}}{\Gamma \vdash x : \tau}$$

$$\text{ArithT1}\,\frac{\Gamma \vdash e_1 : \tau \qquad \tau \in \{\texttt{Safe}, \texttt{Seq}\} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{Seq}} \qquad \text{ArithT2}\,\frac{\Gamma \vdash e_1 : \tau \qquad \tau = \texttt{Dyn} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{Dyn}}$$

$$\text{IllegCast}\,\frac{(E, x) \Rightarrow_l l : t \qquad \mathit{incompatible}(t, t')}{\Gamma \vdash (t')x : \texttt{Dyn}}$$

No memory access to Safe pointers is subject to dynamic bounds checks; conversely, all memory accesses to Dynamic pointers are instrumented with runtime checks. For a memory access to a Sequence pointer, if nesCheck can statically determine that it will never result in an out of bounds operation, it will not be instrumented with a dynamic check. We therefore define a predicate $safe(*p)$ that is true ($\top$) iff the memory access $*p$ does not require bounds checks, false ($\bot$) otherwise:

$$safe(*p) = \begin{cases} \top & \text{if } \Gamma \vdash p : \texttt{Safe} \vee \\ & \quad (\Gamma \vdash p : \texttt{Seq} \wedge *p \text{ not out of bounds}) \\ \bot & \text{otherwise} \end{cases}$$

nesCheck declares a memory access as never out of bounds only if the in-memory metadata propagated up to that access statically indicates the safety of the operation. The result of the predicate $safe(*p)$ for memory accesses on that pointer removes the need to carry the pointer type over to the dynamic bounds checks rules. The formal proof of CCured [26] shows it is safe to leave memory accesses uninstrumented, and the same proof also applies in our case to statically-provable Sequence pointers accesses.

A.3 Dynamic Bounds Checks

The bounds checking operational rules are very similar to those of SoftBound, so we omit most of them for brevity. We present here the rules for the evaluation of a pointer dereference operation, both in case of success:

$$\text{DerefSuccess}\frac{\begin{array}{c}(E, lhs) \Rightarrow_l p : t* \\ \texttt{read}\ (E.M)\ p = \texttt{some}\ v_{(sl,sh)} \\ safe(p) \vee (sl \geq 0 \wedge sh \geq sizeof(t))\end{array}}{(E, *lhs) \Rightarrow_l v : t}$$

and in the case of memory error (failed bounds check):

$$\text{DerefFail}\frac{\begin{array}{c}(E, lhs) \Rightarrow_l p : t* \\ \texttt{read}\ (E.M)\ p = \texttt{some}\ v_{(sl,sh)} \\ sl < 0 \vee sh < sizeof(t)\end{array}}{(E, *lhs) \Rightarrow_l \texttt{MemoryError} : t}$$

Other rules, such as those for type casts and pointer arithmetic, need to ensure that metadata information is propagated correctly:

$$\text{PtrArithm}\frac{\begin{array}{c}(E, ptr) \Rightarrow_r (E', l_{(sl,sh)} : p*) \\ (E', val) \Rightarrow_r (E'', off_{(sl',sh')} : \texttt{int}) \\ l' = l + off * sizeof(p) \\ sl' = sl + off * sizeof(p) \\ sh' = sh - off * sizeof(p)\end{array}}{(E, ptr + val) \Rightarrow_r (E', l'_{(sl',sh')} : p*)}$$

$$\text{TypeCast}\frac{\begin{array}{c}(E, rhs) \Rightarrow_r (E', v_{(sl,sh)} : t) \\ t' \neq \texttt{int}\end{array}}{(E, (t')rhs) \Rightarrow_r (E', v_{(sl,sh)} : t')}$$

For the formal rule for integer-to-pointer cast, we follow SoftBound's approach of zeroing out the metadata to avoid potentially undefined behaviors:

$$\text{TypeCastIntToPtr}\frac{\begin{array}{c}(E, rhs) \Rightarrow_r (E', v_{(sl,sh)} : t) \\ t = \texttt{int} \\ (sl', sh') = (0, 0)\end{array}}{(E, (t')rhs) \Rightarrow_r (E', v_{(sl',sh')} : t')}$$

With this support infrastructure of rules in place, we note that the operational rules for values that are valid at runtime and need runtime bounds checks are fully equivalent to their corresponding rules in SoftBound's formal model [24]. Therefore, they satisfy the same safety invariants and ensure memory safety for those values, as proven for SoftBound (in Theorems 4.1 and 4.2, and Corollary 4.1 in [24]). Reducing our formal definitions and methodology to the respective ones in the SoftBound paper allows for a proof by reduction, with the aim of presenting a set of theoretical concepts that the reader is already familiar with, and that can rely on the full formalization of SoftBound.

While adding bounds checks to every memory access is surely sound, as shown by the proof in SoftBound, by combining the latter with the proof in CCured we improve the performance overhead by removing unnecessary checks while still remaining sound. Thus, given the operational semantics rules above, every memory access in nesCheck is either safe at runtime – resulting in a correct access – or causes the application to stop – due to a detected memory error. Therefore, the nesC applications analyzed and instrumented by nesCheck fulfill the set memory safety goals.

B. SOURCE CODE

The source code for our implementation of nesCheck will be made available at https://github.com/HexHive/nesCheck.