Bug Hunting in Sensor Network Applications

Raimondas Sasnauskas, Jó Ágila Bitsch Link,
Muhammad Hamad Alizai, and Klaus Wehrle
Distributed Systems Group
RWTH Aachen University, Germany
{lastname}@cs.rwth-aachen.de

ABSTRACT

Testing sensor network applications is an essential and a difficult task. Due to their distributed and faulty nature, severe resource constraints, unobservable interactions, and limited human interaction, sensor networks, make monitoring and debugging of applications strenuous and more challenging.

In this paper we present KleeNet — a Klee based platform independent bug hunting tool for sensor network applications before deployment — which can automatically test applications for all possible inputs, and hence, ensures memory safety for TinyOS based applications. Upon finding a bug, KleeNet generates a concrete test case with real input values identifying a specific error path in a program. Additionally, we show that KleeNet integrates well into TinyOS application development life cycle with minimum manual effort, making it easy for developers to test their applications.

1. INTRODUCTION

As with any software, testing of wireless sensor network applications is an essential part of development life cycle. The main goal of this engineering task remains the same: finding and fixing program bugs as early as possible.

Currently, sensor network application developers are confronted with a number of domain specific complications. The constrained memory and CPU resources on sensor nodes result in using low-level, type-unsafe languages without dynamic type checking and memory protection. Similarly, because the applications are highly data-flow oriented, the correct exception handling at full coverage is a challenging task. Moreover, due to highly distributed and faulty nature of sensor nodes, some of the program bugs are detected only after the software is deployed.

C language has been the main choice for developing sensor network applications. It provides great flexibility, expressiveness, and in particular, the required low resource footprint. However, the absence of dynamic type checking necessitates very careful programming because many sensor OS's do not support memory protection. We argue that, besides the particular sensor application semantic, the majority of bugs comes from general programming flaws such as memory out-of-bounds references, null pointer dereferences, or wrong type conversions. Especially, the widely used casting between pointers to structures may lead to well-hidden type errors. But in most cases, the given language features are simply misused by novice programmers.

Starting with *lint* [8] tool nearly three decades ago, there has been enormous effort spent on the error removal in C programs. Many techniques have been developed ranging from static code analysis, formal verification, to full state model checking. Numerous tools exist and are freely available to use [4, 5, 7, 15]. Hence, our first step was to employ them for testing sensor network software written in widely spread TinyOS platform [12]. We encountered the following problems due to which the available tools are mostly not used at all, and the developers fall back on manual code debugging techniques:

- Sensor network applications are tightly integrated with the whole operating system leading to time-consuming manual code modification in order to perform the actual testing.
- Most of the tools perform only static code analysis with limited support for C semantics. Since sensor network applications mainly process data from the environment, possible runtime errors might stay undetected.
- None of the tested tools can offer a push-button bug finding technology and the usage learning curve is mostly too steep for a typical developer.

We have discovered that recent research efforts in the area of C code checking are now targeting the usability and full automation as primary design goals [1, 2]. The philosophy is to detect only definite errors, but automatically, at full execution path coverage, and with minimum manual effort by employing symbolic checking techniques [9]. We think that these efforts could finally achieve the integration of sound testing tools into the application development process.

2. RELATED WORK

To the best of our knowledge, currently there are no frameworks for automatic bug detection in sensor network applications *before deployment* at bit-accurate C semantics. For bringing memory safe executions of applications at *runtime*, we only know of the related efforts [6, 10], where the sole representative with fined-grained memory safety at C level is Safe TinyOS.

Safe TinyOS adds dynamic memory checks during compilation which allows to catch unsafe pointer and array operations without corrupting the RAM. Overall, this results in

Features	Safe TOS	KleeNet
Automatic code instrumentation	−	+
Target platform independence	−	+
Assembly level bug detection	+	−
Off-line bug detection	−	+
All possible input values checked	−	+
Automatic test case generation	−	+
Runtime safety enforcement	+	−
No additional resource usage	−	+

Table 1: Comparison: Safe TinyOS and KleeNet

13% increase in the code size and 5.2% increase in CPU usage. As with any dynamic assertion checking, Safe TinyOS can detect program bugs only eventually after the software is deployed. Therefore, still many corner-case bugs circumvent this testing technique. KleeNet, on the other hand offers offline bug detection with automatic code instrumentation. In doing so, it doesn't consume any system resources and ensures memory safety by treating the inputs symbolically i.e. checks any program variable for its all possible input values. As the core engine of KleeNet interprets a virtual instruction set, it cannot detect hardware platform dependent assembly level bugs nor enforce runtime memory safety. Thus, KleeNet complements the beneficial features of Safe TinyOS allowing altogether even more rigorous application testing (see Table 1).

Overall, we make following contributions: First, we integrate an effective bug finding tool into the event-driven TinyOS programming model with usability as a primary goal. Second, we show that, apart from the general checks already available, Klee can easily be extended to incorporate further checks useful for testing sensor network applications. And third, we practically demonstrate that sound testing techniques can be used *throughout* the application development process with minimum manual effort.

3. SYSTEM OVERVIEW

In this section, we present an overview of our system. First, we introduce Klee, which we use for symbolically executing C applications. We continue by discussing how we automatically instrument source code without the help of application developers. We conclude this section by presenting our extension of Klee to enable struct type checking in sensor network applications.

3.1 Klee

Klee is a symbolic execution tool for C programs based on LLVM [11]. In contrast to common runtime testing where the program input is (manually) generated, Klee runs the code on symbolic input initially allowed to be "anything". The programmer only needs to specify which memory locations in his code are input-derived, e.g. an incoming network packet. During code execution, all paths and operations on symbolic variables are tracked. If a bug is detected, Klee automatically generates a test case with concrete values causing the bug. For example, consider a simple application code in listing 1 and the associated KleeNet output in listing 2.

At the current state of its implementation, Klee reports only memory reference and division by zero errors. It has been developed to be scalable and to support all sorts of unsafe type operations including pointer casts and pointer arithmetics.

```
...
call Timer1.startPeriodic(500);
...
int a[10];
unsigned i;
klee_make_symbolic_name(&i, sizeof(i), "i");

event void Timer1.fired()
{
    call Leds.led1Toggle();
    // here we violate memory safety
    // on the 11th signal of this event
    if (i < 11)
        a[i++] = 1;
}
...
```

Listing 1: Array index out of bounds bug

```
$ make kleenet test
KLEE: ERROR: memory error: out of bound pointer
$ make kleenet display
BlinkFailC$i: '10'
```

Listing 2: KleeNet detects the memory error and generates a concrete test case

3.2 Automatic Code Instrumentation

As discussed earlier, one of the major limitations associated with most software testing tools is the lack of user friendly interfaces. Most of the available tools are either not properly integrated into software development process, or they even require manual code modifications for testing the code.

One of our major design objectives is to provide an easy to use bug finding tool for sensor network applications which is strongly integrated in the software development life cycle. For this purpose, we use grammar based automatic code instrumentation. We extend ANTLR [13] based GNU C grammar to automatically insert symbolic annotations (i.e. to mark the memory locations to be checked by Klee) in the C source code. The user only needs to provide a high level configuration stating the variable names that has to be checked inside the code. However, providing a configuration to insert annotations in the code to perform additional checks is optional, as our solution performs some built-in checks to detect common bugs in sensor network applications. For example, struct type cast checking (discussed in section 3.3) and memory checks on received packet buffers.

3.3 Struct type checking

Type conversions in C using type casts is a very common practice, and, definitely not an error. But since type safety is not guaranteed, programmers can interpret each memory region to be of any type. Especially, the casts between pointers to different structure types make the code maintenance difficult [3, 14].

One of the main objective of sensor network applications is to collect and process data from the sensors. The received data is at first available only as an untyped bit stream. Afterwards the pointer to this data is casted to a known structure type based on particular bit fields. During code execution further casts on this memory location are executed. We have extended the functionality of Klee to check the struct type equality during casting operations. This check is optional, but nevertheless the warnings as shown in listing 4 are useful and facilitate program comprehension.

```
...
NewRoute* msg = (NewRoute*) payload;
// message processing
call Queue.enqueue(msg);
...
RouteUpdate* msg =
  (RouteUpdate*) call Queue.dequeue();
// further message processing
...
```

Listing 3: Casting between pointers to different structure types

```
$ make kleenet test struct
KLEE: WARNING: Struct types don't match
KLEE: %struct.NewRoute* -> %struct.RouteUpdate*
```

Listing 4: KleeNet warnings

4. INTEGRATION INTO TINYOS

We decided to integrate Klee into TinyOS by adding a virtual *KleeNet* platform based on the TinyOS *null* platform. This approach allows us to easily add different modules to a platform, that automatically marks sensor value input and incoming packets as symbolic.

Since TinyOS applications are event-driven, parts of the code are executed only eventually after particular events are fired. In order to cover all possible program control flow paths during testing, we have extended our virtual platform with an automatic event signaling mechanism. Once an application is booted, all implemented events are signaled and processed. Finally, after processing the last event TinyOS scheduler is stopped.

Figure 1 shows an overview of KleeNet's build process. First, a user can optionally specify in a configuration file which variables in his code should be marked as symbolic. For this purpose we parse the C-code after NesC compilation and insert calls to `klee_make_symbolic` function. Second, all incoming packet buffers are also marked symbolic automatically. Third, the instrumented code is then passed to Klee which builds the C-object file. Finally, Klee interprets this object file and terminates when no bug is detected. Otherwise a test case with real input values is automatically generated. Running this concrete test case with the unmodified version of the code will cause the deployed sensor network application follow the same path and hit the same bug.

Please note that, as we apply code instrumentation to C source-code, therefore, this process is neither bound to a certain hardware platform nor to TinyOS and NesC. Hence, our approach can easily be integrated into any other sensor network development platform and operating system.

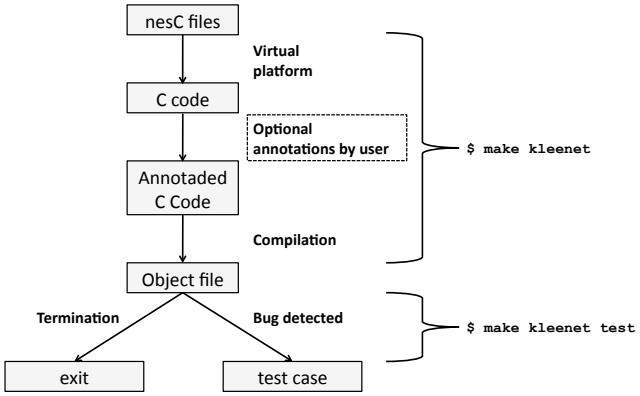

Figure 1: Integration of KleeNet into TinyOS platform

5. EVALUATION

We first checked the BlinkFail application from the TinyOS source repository. It is used to test if the Safe TinyOS toolchain installation is working properly. After marking the array index variable as symbolic we immediately detected the known out of bound pointer error.

```
...
int ratio;

event message_t* Receive.receive(message_t* bufPtr,
        void* payload, uint8_t len) {
  if(len!=sizeof(receive_fail_msg_t)){return bufPtr;}
  else{
    receive_fail_msg_t* rcm=
      (receive_fail_msg_t*)payload;
    // possible division by zero here,
    // depending on received message
    ratio = rcm->good / rcm->total;
    return bufPtr;
   }
}
...
```

Listing 5: Possible division by zero error

```
$ make kleenet test
KLEE: ERROR: divide by zero
```

Listing 6: KleeNet detects the div by zero error

In listing 5, we demonstrate the usability of KleeNet for finding possible bugs without annotating application source code any further. A received message is processed without sanitizing the received message which is a typical fault of students new to programming. KleeNet rapidly detects this mistake and warns the developer (Lst. 6).

Overall, after our initial tests, we have confirmed the following key benefits of KleeNet:

Usability: A programmer can test the code with minimum manual effort and without any previous knowledge about the checking tool.

Coverage: KleeNet covers all possible execution paths and checks all possible data values before application deployment.

Integration: KleeNet is invoked by simply adding an extra build flag enabling the *permanent* code checking during the application development process.

Efficiency: It is fast for everyday use.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented our technique and prototype implementation for automatic testing of sensor network applications before deployment. It is important to fully test (taking all possible inputs into account) embedded applications, such as sensor networks, where the cost of occurring undetected errors after deployment could be fatal. We have demonstrated that it is possible to close the gap between the testing and development community by providing a user-friendly, automated bug finding tool which is strongly integrated in the system development life cycle. We gave an overview of our system design and of the preliminary evaluation results achieved.

Strenuous deployment requirements and resource constrained nature of sensor hardware, e.g. inadequate power supply and limited computational power, demand even more rigorous testing of sensor network applications. Incorporating further useful checks in Klee, such as runtime monitoring of long loops and computationally intensive tasks by adding time annotations, is future work. Similarly, verifying the distributed behavior of sensor network protocols — such as correct state transitions — remains to be addressed. Moreover, apart from TinyOS, we will apply our solution to other sensor network operating systems and development platforms.

7. REFERENCES

[1] D. Babic and A. J. Hu. Calysto: scalable and precise extended static checking. In *ICSE '08: Proc. of the 30th international conference on Software engineering*, 2008.

[2] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *CCS '06: Proc. of 13th ACM conf. on Computer and communications security*, 2006.

[3] S. Chandra and T. Reps. Physical type checking for C. *SIGSOFT Softw. Eng. Notes*, 1999.

[4] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *CCS '02: Proc. of the 9th ACM conference on Computer and communications security*.

[5] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, 2004.

[6] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient memory safety for TinyOS. In *SenSys '07: Proc. of the 5th international conference on Embedded networked sensor systems*, 2007.

[7] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-Safety Proofs for Systems Code. In *CAV '02: Proc. of the 14th International Conference on Computer Aided Verification*, 2002.

[8] S. Johnson. Lint, a C program checker. Computer science technical report 65, Bell Laboratories, 1977.

[9] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 1976.

[10] R. Kumar, E. Kohler, and M. Srivastava. Harbor: software-based memory protection for sensor nodes. In *IPSN '07: Proc. of the 6th international conference on Information processing in sensor networks*, 2007.

[11] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04: Proc. of the international symposium on Code generation and optimization*, 2004.

[12] 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.

[13] T. J. Parr and R. W. Quong. Antlr: a predicated-ll(k) parser generator. *Software: Practice and Experience*, July 1995.

[14] H. Shen, J. Wang, L. Ping, and K. Sun. Securing C Programs by Dynamic Type Checking. In *ISPEC*, 2006.

[15] N. Volanschi. A Portable Compiler-Integrated Approach to Permanent Checking. In *ASE '06: Proc. of the 21st IEEE/ACM International Conference on Automated Software Engineering*, 2006.