KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks Before Deployment

Raimondas Sasnauskas*, Olaf Landsiedel*, Muhammad Hamad Alizai*,
Carsten Weise‡, Stefan Kowalewski‡, Klaus Wehrle*
*Distributed Systems Group, ‡Embedded Software Laboratory
RWTH Aachen University, Germany
{lastname}@cs.rwth-aachen.de

ABSTRACT

Complex interactions and the distributed nature of wireless sensor networks make automated testing and debugging before deployment a necessity. A main challenge is to detect bugs that occur due to non-deterministic events, such as node reboots or packet duplicates. Often, these events have the potential to drive a sensor network and its applications into corner-case situations, exhibiting bugs that are hard to detect using existing testing and debugging techniques.

In this paper, we present *KleeNet*, a debugging environment that effectively discovers such bugs *before deployment*. KleeNet executes unmodified sensor network applications on symbolic input and automatically injects non-deterministic failures. As a result, KleeNet generates distributed execution paths at high-coverage, including low-probability corner-case situations. As a case study, we integrated KleeNet into the Contiki OS and show its effectiveness by detecting four insidious bugs in the μIP TCP/IP protocol stack. One of these bugs is critical and lead to refusal of further connections.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification, Assertion Checkers; D.2.5 [**Testing and Debugging**]: Testing tools, Distributed Debugging, Symbolic execution

General Terms

Design, Reliability, Verification

Keywords

Automated Protocol Testing, Failure Detection, Experimentation, Wireless Sensor Networks

1. INTRODUCTION

Wireless Sensor Networks (WSN) are envisioned to be deployed in the absence of permanent network infrastructure and in environments with limited or no human accessibility [27,30,33]. Operating complex distributed protocols over lossy links and potentially unreliable nodes, WSNs demand extensive testing and debugging before deployment. After deployment, bugs are very difficult to detect, costly to fix, and can potentially cause an operational outage.

For example, a recent WSN deployment in the Swiss Alps [3] experienced sporadic packet loss on all GSM nodes simultaneously. This was caused by a bug in the GPRS drivers of the WSN sink node used for collecting measurements. The bug prevented it from reconnecting to the cellular network after connection loss. It did not occur during testing before deployment as the test site had a very good GSM connectivity. Similarly, a bug in the flash driver of Deluge [11] caused a three-day network-outage during a deployment on an active volcano in Ecuador [37]: Due to this bug, rebooting after remote reprogramming failed breaking the network for three days until each node was manually reprogrammed on the volcano.

Overall, examples of bugs detected during deployments [3,16,33,37] indicate that bugs are often revealed in corner-cases, that were not tested sufficiently before deployment. To explore these corner-cases during the debugging of wireless sensor networks, we present *KleeNet*, a debugging environment for high-coverage testing of sensor network applications *before deployment*. It enables the detection of bugs that result from complex interactions of multiple nodes, non-deterministic events in the network, and unpredictable data inputs. Built on the symbolic virtual machine KLEE [4], KleeNet makes the following four key contributions and facilitates rigorous testing of distributed WSN applications and protocols:

- **Coverage:** KleeNet enables symbolic execution of unmodified distributed sensor network applications. It considers symbolic input values from the environment and generates execution paths of participating nodes at high-coverage.
- **Non-determinism:** KleeNet injects symbolic, non-deterministic events such as loss, duplication and corruption of packets and node failures automatically. These events can appear at any point in time, and thus drive the sensor network execution into corner-case situations.

- **Distributed Assertions:** KleeNet allows to formulate intuitive assertions about the distributed state of a sensor network. If an execution path violates an assertion, KleeNet automatically generates a test case to reproduce the bug and allows to easily narrow down its root cause.
- **Repeatability:** Automatically generating test cases for each bug found, KleeNet allows to reproduce the execution that led to a bug and to replay distributed systems. We believe that this seamless transition between the testing and real execution makes KleeNet a powerful and attractive debugging environment.

We demonstrate the contribution and effectiveness of KleeNet with four insidious bugs discovered in Contiki's μIP TCP/IP stack [8]—a protocol stack for resource-constrained devices. It has been actively used for years in many open-source and commercial projects worldwide, such as wireless sensor networks and (pico) satellites. One of the detected bugs is crucial—it lead to a dead TCP protocol state refusing any further connections to the affected node.

The remainder of this paper is structured as follows. Section 2 discusses existing tools for testing and debugging of WSNs and relates our approach. We introduce the basic concept of KleeNet in Section 3 and provide the required background on symbolic execution in Section 4. Section 5 details the design of KleeNet and Section 6 introduces KleeNet extensions along with optimizations to reduce the number of states required for the evaluation of distributed systems. Next, Section 7 describes the implementation and usability of KleeNet. We present evaluation results and the detected bugs in Section 8 and conclude in Section 9.

2. RELATED WORK

Existing approaches of debugging wireless sensor networks fall into two categories: testing *before* and *after* deployment.

Testing before deployment. Compiler tools are the first step to detect and troubleshoot local node errors, such as out-of-bounds memory access, wrong type conversions and possible race conditions. Safe TinyOS [7] is an example of a compilation toolchain enforcing type-safe applications in TinyOS [18]. It warns the developer about unsafe code portions during compilation and additionally instruments the code with safety annotations preventing memory corruption at runtime. However, such compiler tools are only suitable to identify local implementation errors and are unable to detect distributed bugs that result from complex node interactions.

Employing formal methods [2, 21, 34], code analysis approaches attempt to verify the correctness of sensor network applications by extracting a model from the application code that is later fed into a model checking environment. These tools face the state explosion problem due to non-determinism in distributed systems such as wireless sensor networks. Furthermore, model extraction requires manual effort, making formal validation a laborious task. The symbolic execution based tool FSMGen [14] comes closest to our work. It automatically derives a high-level system representation from a single TinyOS application in the form of state machines. In contrast, KleeNet detects bugs in the distributed interaction of sensor network applications and models non-deterministic events such as packet loss and node reboot to drive the sensor network into corner-cases.

Next to validation, simulation and emulation based tools provide a convenient way to test sensor network applications. Discrete event simulators—such as TOSSIM [17]—provide functional debugging support and are easy to use. Full-system simulators, including Avrora [31], COOJA [22], S^{2}DB [36], and ATEMU [23], offer additional fidelity in terms of timing, hardware characteristics and memory access. Although being systematically used in many projects, such simulation based testing lacks high input coverage and simulation of scenarios that lead to the occurrence of low-probability, but often severe bugs.

Testing after deployment. After sensor nodes have been programmed and deployed, memory safe execution of applications is one of the most crucial concerns. For example, Safe TinyOS and Neutron [6] automatically enforce run-time memory safety and efficiently recover from memory bugs. In addition, semi-automated approaches, such as TraceSQL [5] and NodeMD [15], offer simple annotations to trace and avoid critical errors in specific parts of the code. Finally, interactive debugging systems such as Marionette [38] and Clairvoyant [40] enable GDB like tracing capabilities at runtime. However, these approaches focus on local state only and often interfere with the application execution itself.

Sympathy [24], Nucleus [32], EnviroLog [20], Dustminer [12], PDA [25] and others [19, 29, 35] (passively or actively) collect and analyze information about node states and their communication. While these traces provide insight into a deployed network and its distributed interaction, it is often difficult to narrow down a failure with certainty or to repeat the scenario causing it [35]. MDB [29]—a post-mortem debugger for macroprograms—enables high-fidelity analysis of distributed systems using *hypothetical changes*. However, all these approaches detect failures in a particular execution trace or analyze slight modifications to it.

In contrast, KleeNet enables the testing of applications before deployment: it provides (1) high input and execution path coverage through symbolic analysis techniques, (2) execution accuracy by executing unmodified applications, (3) a network model to analyze distributed node behavior, and (4) non-deterministic network and node failures to discover new, often corner-case distributed execution paths.

For completion, we relate our contribution to existing symbolic execution techniques in Section 4.

3. BASIC CONCEPT

In this section we present an overview of KleeNet and its debugging process. Using a simple example scenario, we demonstrate the architectural challenges and highlight our key contributions.

Consider a scenario with three communicating nodes, successively placed so that each of them is directly connected to its neighbors only (Case 0 in Figure 1). Assume that node A begins the communication by broadcasting a data packet to its neighbor B. Upon receiving the packet, node B first determines the validity of the packet, i.e., it calculates the header checksum. As the packet can contain arbitrary data, the validation branches into two execution paths, namely "*packet invalid*" and "*packet valid*". Hence, KleeNet follows both program execution paths at node B separately. In case of an invalid packet, B discards it (see Case 1 in Figure 1). While manually testing the correct handling of arbitrary program input is time-consuming and challenging, KleeNet explores such execution paths automatically.

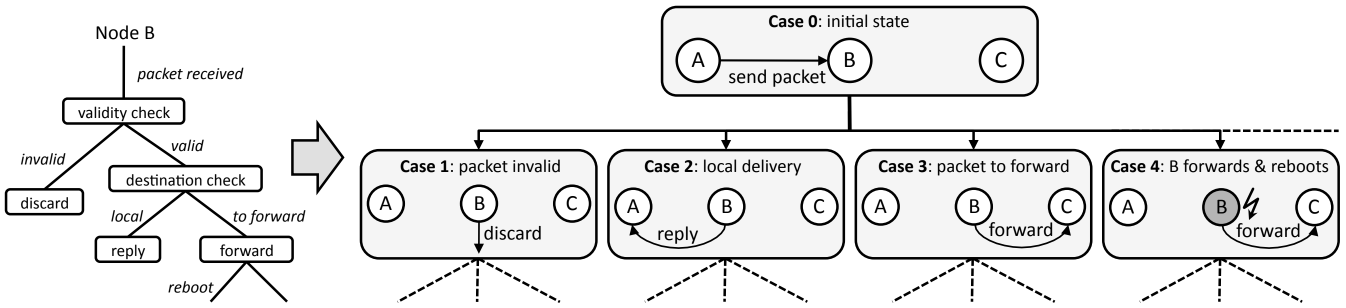


Figure 1: KleeNet's debugging approach. The initial execution path (A broadcasts a packet to B) splits after packet reception at node B into four execution paths: packet invalid, local delivery, packet to forward, and packet to forward followed by a reboot of node B. Following these execution paths, KleeNet provides a high program coverage including non-deterministic failures such as reboots and packet losses.

In the case of a valid packet, B next checks the destination of the packet. Here, B again splits the execution path to "*packet to forward*" and "*local delivery*". Case 2 in Figure 1 represents the latter path where node B consumes the packet and sends a reply back to node A. Whereas, in Case 3, B forwards the packet to its next neighbor C. These two execution paths represent a node's protocol behavior for valid packets and are covered by KleeNet as well.

Cases 1–3 result from B's functional decisions due to the uncertain packet data. In deployed wireless sensor networks, applications have to be aware of non-determinism: For example, a node might experience a reboot, receive duplicate packets or packets might get lost due to interference. Case 4 illustrates one of these failures: After forwarding the packet to C, B encounters a reboot and looses its state information. As a result, a new node execution path occurs, which can reveal further aspects of a distributed protocol execution. Such communication aspects are either impossible to observe in traditional pre-deployment testing mechanisms, or require laborious manual effort by the developers to generate them. In contrast, KleeNet—based on high-coverage symbolic execution—automatically injects such non-deterministic and low-probability events into the program flow and thus explores corner-case situations.

4. SYMBOLIC EXECUTION

Our debugging approach is built on the symbolic virtual machine KLEE [4]. Therefore, we give a short overview on symbolic execution. This background information is necessary to explain the context of KleeNet's environment as well as to understand the challenges that influenced our design decisions.

Symbolic execution [13] allows the automatic exploration of execution paths in complex applications. The idea is to execute program code on symbolic input, e.g., incoming network packets, which are initially allowed to take any value. During symbolic execution code is executed normally until it reaches a symbolic value. At this point, program execution is branched into all possible states and execution is resumed for each branch. For example, if the program flow reaches the code line `addr = packet->src` where the packet is symbolic, the symbolic execution engine assigns `addr` with a constraint containing symbolic memory access. Next, upon reaching a symbolic branch, e.g., checking whether `addr` equals `myAddr`, the engine forks the active program path and follows both program paths independently with additional path constraints, namely `addr == myAddr` and `addr != myAddr`, respectively. If an explored application path terminates or detects a bug such as a buffer overflow, a test case with the respective input values that led to this execution path is generated. The easy way of bug reproduction makes this testing approach very effective and attractive for many developers.

Existing symbolic execution frameworks such as KLEE, JPF-SE [1], and CUTE [28] are designed to explore non-distributed applications including the GNU core utilities `ls` and `mkdir`, and the `Vim` editor. For interaction with the environment, e.g., file system or network socket they assemble models that abstract from the complexity of the underlying hardware, system libraries, and network communication logic. Thus, in contrast to our work, these tools do not support symbolic distributed execution, which controls and groups independently explored execution paths into valid distributed execution scenarios.

In our previous work [26], which is also built on top of KLEE, we showed that it is possible to symbolically execute a single TinyOS instance and check applications for data input safety before deployment. In this paper we extend this preliminary work to enable symbolic execution of distributed sensor network applications.

Moreover, we designed non-deterministic failure models that uncover new distributed execution paths.

5. DESIGN

Detection of low-probability bugs in WSNs demands a high-coverage of possible distributed execution paths. In the following we discuss the design of KleeNet and its techniques that enable automated generation of such paths in distributed systems. These paths also include non-deterministic events such as packet loss or node failure.

KleeNet provides the following four mechanisms to generate high-coverage paths and to test the state of a distributed system (see Figure 2):

- **Symbolic Input:** Testing behavior of applications on arbitrary environment input (packets, sensor data) enhances software reliability. We extend this concept with node and network models to reflect the distributed nature of WSNs.

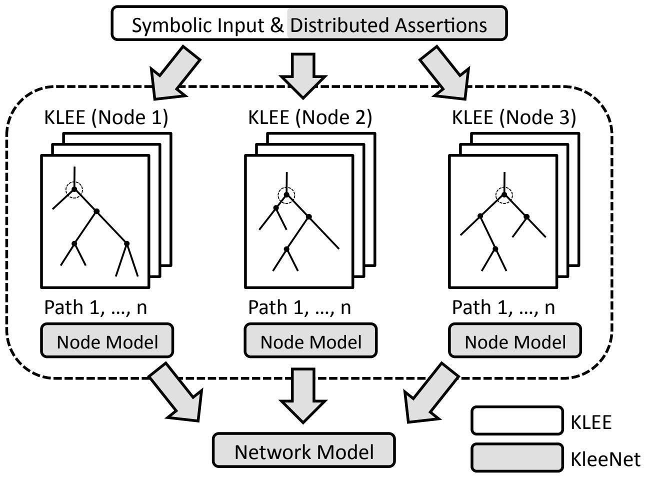


Figure 2: KleeNet overview. KleeNet manages distributed program execution, communication, and user-defined distributed assertions. Extensions of KleeNet to KLEE are shaded.

- **Node Model:** The node model represents the behavior of individual nodes in KleeNet including outage and reboot. Modeling such low-probability events allow KleeNet to drive sensor network applications into corner-cases of their execution.
- **Network Model:** The network model describes the interaction of nodes, such as sending and receiving packets, and corresponding non-deterministic failures, such as packet loss, duplication and corruption. In our evaluation we show that complex interaction bugs can arise from such events and drive the distributed systems into invalid states or even render deployed network non-operational.
- **Distributed Assertions:** Distributed assertions allow to permanently check the distributed state of a wireless sensor network. With the help of these assertions KleeNet detects when a sensor network reaches undefined states.

In the remainder of this section we discuss individual design choices and their complexity.

5.1 Symbolic Input

Symbolic input is the first step to explore the execution paths of a distributed application. In KleeNet, a developer can mark variables or data structures to be symbolic. Sensor network applications are driven by the input from the environment. Hence, the most critical execution scenarios appear by marking network packets and sensor data as symbolic. Thus, assuming a loss-free network, symbolic input is a convenient way to test an application's functional correctness at an early stage of the development.

For example, a developer of a new protocol stack might want to receive a packet with a symbolic header to analyze which execution paths header processing code will take, and how the stack responds to a particular data input. Hence, it allows to test whether a protocol stack or application remains in valid states independent of the data input, and gives early feedback to the developers.

Complexity: Assuming a worst case execution time of c, symbolic execution along an n-ary branch results in growth of $\mathcal{O}(c^n)$. Therefore, for a sensor network setup with m nodes the resulting path complexity is $\mathcal{O}(c^{n \cdot m})$. Despite this worst case complexity, we observed that most sensor network applications, including their communication protocols, do not exhibit this exponential growth. This observation can be explained by the resource-constrained nature of sensor network applications: In practice, protocols commonly only have a small number of possible execution paths. As a result, the number of possible protocol states remains manageable [14].

5.2 Node Model

KleeNet models the behavior of individual nodes, such as node reboots and node outage. Introducing these non-deterministic events allows KleeNet to drive protocols and applications into corner-cases of their execution.

During their lifetime, sensor nodes may experience reboot and outage due to memory bugs, hardware failures or a power outage. Typically, these unexpected events cause a node to loose all program state. If neighboring nodes do not detect this reboot or outage reliably, this may lead to undefined states or even to operational outages of a deployed distributed system [39].

Node Reboot and Outage: To cover these low-probability situations we introduce symbolic reboot events in the node model of KleeNet. When a reboot event is triggered, KleeNet branches the active execution path into two paths: In the one branch we continue normal execution while in the other, we reboot the node by clearing its state and reinitializing the deployed application. Hence, this branch re-executes the application from the initial state while all other nodes in the network continue with their uninterrupted execution. Similarly, KleeNet allows symbolic outage events to occur during the program flow, creating additional execution paths. In doing so, we reveal corner-case execution paths increasing the overall coverage in a distributed system. Moreover, by exploiting our domain knowledge of sensor network applications, we reduce the overall complexity.

Complexity: In contrast to other non-deterministic failures, node outage does not directly increase the number of execution paths. A node reboot forks the active application execution path and brings it to its initial state, i.e., to an already visited state. Nonetheless, this state plays a different role in the distributed node communication.

5.3 Network Model

Next to modeling the behavior of individual nodes, KleeNet models their interaction. Hence, KleeNet provides a network model with non-deterministic events such as packet loss, corruption and duplication.

Packet Loss: In wireless sensor network deployments, packet loss is a common failure faced by applications. To test applications against packet losses, KleeNet's network model introduces symbolic packet drops. A symbolic drop conceptually discards arbitrary packets traversing the network. This allows KleeNet to effectively discover additional distributed execution paths beyond the sensor network behavior induced by the deterministic execution of applications. Hence, it achieves much higher coverage than in the case of ideal network conditions or when using a network model with random packet drops.

Packet Duplication: Similar to packet drops, KleeNet injects packet duplicates—a frequent phenomenon observed in deployed sensor networks—in the network model. For example, in collection tree protocols, such as CTP [10], packet duplication is a well known problem that has ruinous effects over multiple hops as duplication is exponential: If each hop produces one duplicate on average, then there will be two packets at the first hop, four at the second, and eight at the third. Therefore, by employing symbolic duplicates, we can test such protocol robustness at different steps of distributed execution. In Section 8 we demonstrate that in stateful communication protocols, packet duplicates must be considered very carefully to avoid complex interaction bugs.

Packet Corruption: Symbolic packet corruption events is a further feature of KleeNet. As with other non-deterministic failures, KleeNet can corrupt any packet traversing the network model. Again, the resulting execution paths discover new distributed execution paths.

Complexity: The choice between delivering, dropping, or corrupting a packet trebles the number of investigated execution paths. Thus, worst-case complexity stays within $\mathcal{O}(c^{n \cdot m})$. In all distributed scenarios where we detected bugs our network model dropped or duplicated at most 20 packets. Any subsequent failure events were leading to redundant execution paths showing the same, already discovered, distributed application behavior. In that sense, KleeNet leverages domain knowledge to make the symbolic execution more effective for this specific application domain.

Overall, modeling the non-determinism of node and network behavior allows KleeNet to cover execution paths beyond normal program execution and to find bugs in corner-cases. This high degree of coverage is very challenging and laborious to achieve with conventional testing tools such as network simulators and testbeds.

5.4 Distributed Assertions

Depending on a application, even a small sensor network setup has the potential to produce hundreds or even thousands of distributed execution paths. Manual analysis of each of these scenarios becomes quickly a very laborious task.

In KleeNet, we extend the C like assertions provided by KLEE to check predicates on distributed node states. This is very useful for protocol testing, e.g. to permanently check if the communicating nodes reside in well-defined states during sensor network execution. As an example, we can specify the following distributed assertion in the code of node A: If node A adds a new parent node (node B in this case), then node B should also have node A in his children set:

```
if (parentID != NULL) {
  assert(NODE(parentID, ''isChild'', myID));
}
```

Note that this assertion calls the function `isChild(myID)` at node B whereas `parentID` and `myID` are variables in A's code. Thus, if any of the distributed execution scenarios violates this assertion, a developer can directly replay the suspicious scenario and narrow down the root cause of the problem. Furthermore, KleeNet's distributed assertions can be easily extended to formulate high-level sensor network predicates as presented in the PDA paper [25].

Nonetheless, specifying distributed assertions in KleeNet is a manual step requiring knowledge of application logic and appropriate data structures.

6. KLEENET EXTENSIONS

After discussing the design of KleeNet, we next discuss our main extensions to the KLEE engine and further introduce optimizations to reduce the number of execution paths required to obtain all valid distributed execution scenarios.

6.1 Symbolic Distributed Execution

From the software architecture point of view, KLEE is not designed to test the interaction of distributed applications. Instead, each of the emerging program execution paths is explored independently from all others. An application being tested in KLEE communicates with its environment both via native library calls and using runtime models (e.g. file system, socket library). However, these models cannot influence the execution of other program paths and vice versa. For example, if several paths of a TCP client/server application reach the blocking `accept` socket call, the affected execution paths will neither know that there might be a path calling `connect` nor be able to synchronize or exchange data between these paths. To overcome this issue, KleeNet provides a number of extensions to KLEE enabling symbolic execution and interaction of distributed applications.

Execution path control: KleeNet offers hooks to suspend, resume, and manually fork execution paths. This functionality is required by the network model to control the communication, i.e., to synchronize the program flow of tested applications.

Distributed scenario isolation: All emerging execution paths in KLEE are treated equally within a single set. In KleeNet, each of the execution paths belongs to a particular distributed execution scenario. Thus, this allows to share the data (e.g. send/receive data packets, trigger events) between these paths and check distributed assertions.

6.2 Optimizations

As long as sensor nodes exchange non-symbolic data, KleeNet's network model acts like any other simulator network model. However, upon reception of symbolic input or when symbolic failures occur, the execution paths of affected nodes automatically branch into new ones. These new paths must be considered by the network model to keep the execution of the distributed system consistent.

The naive approach to tackle this situation is straightforward: every time an execution path branches all other affected execution paths are forked as well. This causes a quadratic growth of execution paths and quickly leads to path explosion. However, many of these newly created paths are redundant and unnecessarily waste memory resources. Consider an example with three nodes (A_1, B_1, C_1) where node B_1 branches into four execution paths during its execution (see Figure 3). With the naive approach, four distributed execution scenarios ($\{A_1, B_1, C_1\}$, $\{A_2, B_2, C_2\}$, $\{A_3, B_3, C_3\}$, $\{A_4, B_4, C_4\}$) are generated with six additional path forks (Figure 4(a)).

In KleeNet, we optimize this naive approach by removing redundant branches and forking execution paths *on demand*. Our network model tracks application paths which request packet transmission and afterwards forks destination nodes before delivering the data, if necessary. Therefore, we only consider execution paths that are involved in a particular communication while the rest of the sensor network remains unchanged. For the exemplary scenario discussed above, KleeNet's network model generates four distributed execu-

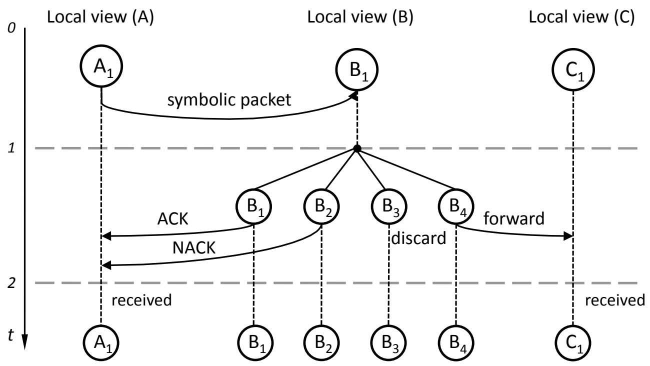
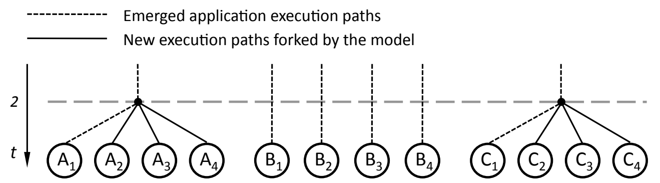
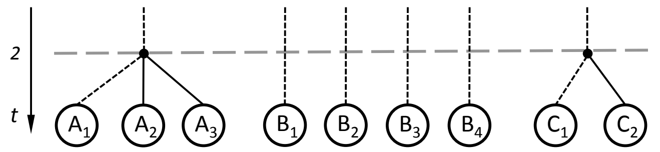


Figure 3: Upon reception of a symbolic packet, B forks into four execution paths.


Emerged application execution paths
New execution paths forked by the model
2
t

(a) Naive approach: all affected execution paths are forked.



(b) KleeNets reduces the number of states and forks execution paths on demand: Node A is forked twice, whereas node C only once.

Figure 4: Comparison of the naive approach and optimizations provided by KleeNet.

tion paths ($\{A_2, B_1, C_1\}$ – ACK, $\{A_3, B_2, C_1\}$ – NACK, $\{A_1, B_3, C_1\}$ – discard, $\{A_1, B_4, C_2\}$ – forward) with only three additional forks (see Figure 4(b)). Compared to the naive approach, we significantly decrease the number of execution paths. This is especially efficient in larger sensor network setups where the transmission of a packet only affects a small number of neighboring nodes.

Conceptually, a distributed execution scenario in KleeNet is a set of related execution paths. KleeNet detects and manages these sets throughout the sensor network execution. Packets exchanged between nodes of the same path set are not visible in other sets. For example, the ACK packet sent by B_1 to A_2 in the first path set is not observed in any other set.

7. IMPLEMENTATION

In this section we discuss the realization of node and network models in KleeNet, its integration into the Contiki OS, and usability aspects.

7.1 Node and Network Model

The node and network models of KleeNet base on a hardware abstraction layer, similar to TOSSIM [17]. Hence, they provide hooks to (1) handle node boot-up, outage, and resets, (2) to interact with the environment via sensors and (3)

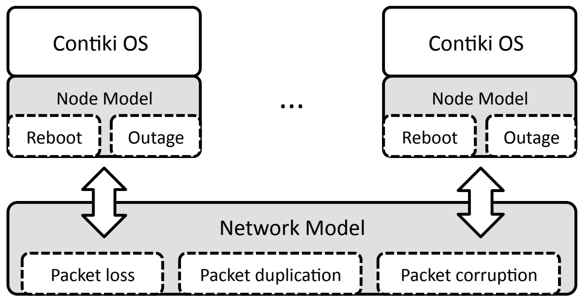


Figure 5: Contiki *kleenet* platform. The non-deterministic failures are triggered within network and node models.

to communicate, i.e., to send and receive packets. KleeNet's network model abstracts from the MAC-layer and hence considers network and application-level behavior only.

In contrast to existing models as in TOSSIM, KleeNet models node and network events symbolically. Hence, failures can appear at "any" point in time, creating new distributed execution paths (see Section 5.3). For example, packet loss in KleeNet is not determined by a channel model. Instead, KleeNet forks the execution path and considers both cases, i.e., successful delivery and packet loss. Modeling of packet duplicates and corruption as well as node failures is realized similarly. To ease its usability, KleeNet provides a flexible plug-in architecture, allowing users to integrate own failure models, such as packet reordering and delay.

The code that injects failures is executed both in the network model (e.g. within packet transmission code for packet related failures) and in the main OS scheduler using symbolic timer events (node failures only). In the latter case, the granularity of the symbolic timer must be carefully chosen to limit the state explosion.

For node connectivity, KleeNet allows a developer to define the topology according to testing requirements. Hence, a user can either select complex, multi-hop radio models (list of node pairs) or a simple model where all nodes can reach each other.

7.2 Case Study: Integration in Contiki

KleeNet has a platform-independent design, and therefore, it can easily be ported to any sensor operating system[1]. It can even evaluate sensor networks with nodes executing different operating systems and applications. As a case study, we present KleeNet's integration into the Contiki [9] operating system.

We integrated KleeNet into Contiki by adding two additional build platforms, namely *kleenet* (see Figure 5) and *replay*. Both platforms are based on the existing *native* platform which executes Contiki on an ordinary Linux machine.

KleeNet Platform: One of our key design goals is to allow developers to test their applications with low manual effort. In KleeNet, users specify a simple configuration file describing the desired testing scenario consisting of (1) the applications of each node, (2) desired failure classes (see Table 1). To model scenarios beyond direct connectivity,

[1]The only prerequisite is the LLVM (Low Level Virtual Machine) front-end for the WSN OS programming language.

ID	app	loss	dupl	corrupt	reboot	outage
1	tcpclient	1	1	0	1	0
2	tcpserver	1	1	0	1	0

Table 1: Sample scenario configuration file with symbolic packet losses, duplicates, and node reboots enabled.

users may optionally specify a network topology to model complex, multi-hop communication. Symbolic data and distributed assertions are specified directly in the source code as they operate on application data structures.

To activate KleeNet, a user is only required to type `make TARGET=kleenet` to compose his test scenario. Next, `make runtest` starts the exploration process and generates test cases for all discovered execution paths.

Replay Platform: A test case in KleeNet specifies the input, such as packets received, and non-deterministic events, such as node failures, that led to the violation of an (distributed) assertion.

To replay this test case in a real system, KleeNet additionally provides a so-called replay platform. It represents a native (i.e. non-symbolic) version of the applications under test. Hence, executing a test case on this platform provides the exact execution path (assuming deterministic code) that lead to a bug. As this native version is a set of Linux binaries, the developer can rely on well-known tools such as `gdb`, `valgrind` and `wireshark` to easily identify the root cause of a bug. Overall, the replay platform is very similar to a simulator, which executes a given scenario setup with a predefined set of parameters.

7.3 Usability

A key feature of KleeNet's integration in Contiki is the seamless transition between the testing, replay, and deployment platforms. Already at the early stage of application development, KleeNet allows a developer to define a sensor network scenario including distributed assertions that permanently check the functionality of the distributed system. If KleeNet encounters a bug, a developer switches to the replay platform and executes the unmodified sensor network scenario again with the test case generated by KleeNet. Hence, the resulting sensor network execution follows the same path and hits the same bug. After fixing bugs, the code can be deployed and executed on real motes without the need for modifications.

Overall, we experienced that the combination of replaying a distributed system in KleeNet and the possibility to use standard tools for analysis of program execution is a very effective strategy to narrow down, fix, and report the root causes of complex interaction bugs discovered during testing.

8. EVALUATION

This section demonstrates the efficacy of KleeNet as a debugging tool. For our evaluation, we use a comprehensive set of standard Contiki applications, eminently the widespread μIP TCP/IP stack[2]. Using KleeNet, we discovered four insidious interaction bugs[3] that occurred due to typical wireless communication constraints such as packet loss and duplicates. These bugs may lead to detrimental situations, such as causing a total refusal to connect to an active communication partner ever again.

[2]From here onwards we use the term μIP as an abbreviation for the TCP/IP protocol stack of Contiki.

[3]All bug reports were confirmed and fixed by the Contiki developers.

Runtime	6 sec, 14–15 MB RAM (3 scenarios)
Scenario 1	TCP listen port not found, reset
Scenario 2	TCP connection established (port 7777)
Scenario 3	TCP connection established (port 0)

Table 2: Symbolic TCP destination port.

In the remainder of this section we detail on the bugs discovered and illustrate the techniques used to identify each. While highlighting KleeNet's debugging effectiveness, we also discuss its limitations experienced during debugging of an established protocol stack.

8.1 Case 1: Symbolic Packet Data

As a first scenario, we tested the robustness of μIP on arbitrary input (see Section 5.1). By marking packet data as symbolic in KleeNet, we show how incorrect handling of TCP ports in μIP led to the establishment of a valid connection to a closed (not listening) TCP port.

Communication Scenario: Our evaluation scenario involved two Contiki nodes running simple TCP client and server applications, respectively. The server application initializes the μIP stack and listens on port 7777 for incoming connections. The client application initializes its stack, connects to the server, sends one data packet, and closes the connection. This test aimed at verifying the state consistency of both applications: If the client enters the connected state, the server should enter the connected state as well. Similarly, the same assertion should be valid for the *not* connected case. This type of statement allows developers to describe the functional correctness of their systems very intuitively.

Test Setup: We began our state consistency tests by marking the TCP destination port of the client as symbolic. In addition, we specified the following assertion in the connected and not connected states of the client's application:

```
assert(connected == NODE(2, ''connected''));
```

As described in Section 5.4, KleeNet's assertions check the overall state of a distributed system, e.g., in this test case they check the states of the client and the server. Hence, `NODE(2, "connected")` represents the global symbol **`connected`** in the server's (node id #2) application.

Bug #1: In this test two-node setup, KleeNet discovered three unique distributed execution paths (see Table 2): (1) connection refused due to trying to connect to an inactive port, (2) successful connection to port 7777, and surprisingly (3) a successful connection to port 0.

For the latter unforeseen path, KleeNet triggered the distributed assertion violation indicating that the client was in the connected state although the server application was not aware of the connected client. After replaying the test case we saw that the client successfully established a connection to server port 0, which, according to IANA[4], is a valid TCP port. Moreover, the sent packet was acknowledged by the server's TCP/IP stack and even a subsequent disconnection

[4]Internet Assigned Numbers Authority

Runtime	8 sec, 14–15 MB RAM (7 scenarios)
5 scenarios	TCP client connected
2 scenarios	TCP client not connected

Table 3: Symbolic packet loss.

proceeded correctly. However, the only server application was running on port 7777. It turned out that, during bootstrap, the μIP stack initializes all listen ports to 0. Upon a connection request to port 0, it assumes the port to be listening without checking for a running (server) application. Hence, the connection is established and the client can send arbitrary data that is acknowledged by the stack.

Discussion: Executing distributed applications on symbolic input highlights the core feature of KleeNet. It allows to achieve high-coverage of distributed scenarios while executing unmodified code. To the best of our knowledge, none of the available sensor network debugging tools can achieve this level of automatic coverage for scenarios with more than one node. For example, using a simulator we would require to execute the same test with 65536 different ports to verify the state consistency discussed in this section.

8.2 Case 2: Symbolic Packet Loss

In the following example, we show how a packet loss can lead to dead protocol states in μIP, thereby, causing a refusal to reconnect to an active communication partner.

Communication Scenario: We use the same two-node communication scenario as in the previous example. Additionally, we set the destination port of the TCP client to 7777. In this scenario, we aimed at testing the robustness of connection establishment in applications against non-deterministic failures, namely packet loss. The state consistency, i.e, our distributed assertions, are specified as in case 1: If the client is connected, the server should be connected as well. For the *not* connected case, we modified the distributed assertion as follows: Regardless of the state of the server, the client should always trigger the assertion violation if it fails to connect to the server or the connection process gets aborted. This assertion was motivated by the assumption, that the TCP connection establishment process should be resistant against arbitrary packet loss.

Test Setup: In this setup, we enabled symbolic packet loss for both nodes, i.e., for connection requests and replies. The two assertions are defined as follows:

```
/* connected state */
assert(connected == NODE(2, ''connected''));
...
/* not connected and aborted states */
assert(0);
```

A total of 7 distributed execution scenarios was generated (see Table 3). KleeNet triggered assertion violations for two of these scenarios, which we discuss in the following.

Bug #2: During replay, both scenarios violating our assertions demonstrated the same behavior: They showed that the client could not connect to the server, if the first reply from the server, i.e., the SYN/ACK packet, was lost.

Figure 6 depicts the timing diagram of message exchange for one of the communication scenarios: The client initially sends a SYN packet to the server. The server replies with a SYN/ACK packet, but this packet gets lost in the network. As a result, the retransmission timer fires and triggers the

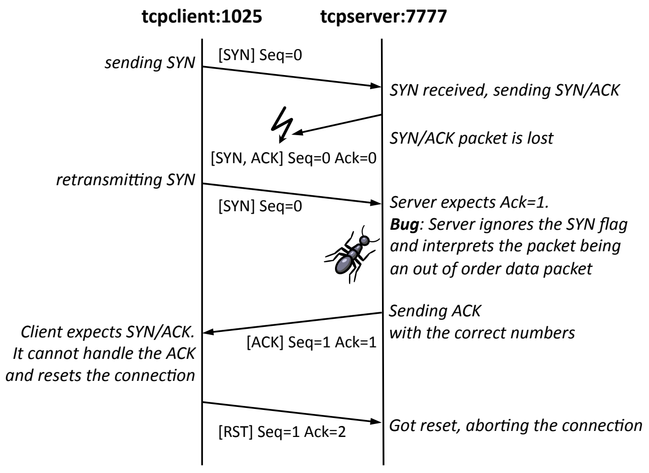

tcpclient:1025
tcpserver:7777
sending SYN
[SYN] Seq=0
SYN received, sending SYN/ACK
SYN/ACK packet is lost
[SYN, ACK] Seq=0 Ack=0
retransmitting SYN
[SYN] Seq=0
Server expects Ack=1.
Bug: Server ignores the SYN flag and interprets the packet being an out of order data packet
Sending ACK with the correct numbers
Client expects SYN/ACK. It cannot handle the ACK and resets the connection
[ACK] Seq=1 Ack=1
[RST] Seq=1 Ack=2
Got reset, aborting the connection

Figure 6: Bug #2: A lost SYN/ACK packet causes a TCP connection reset.

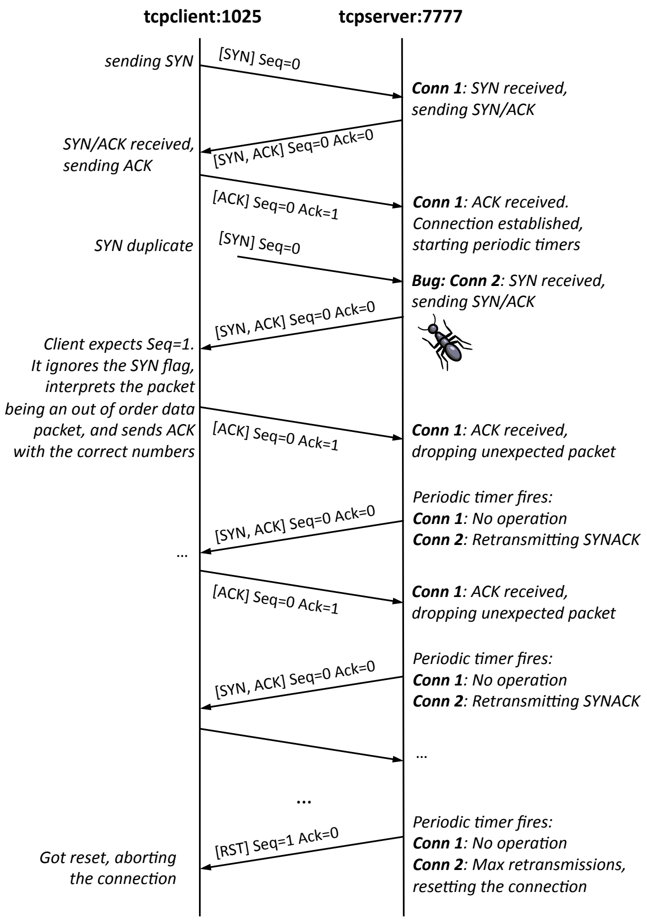
client to send the SYN packet again. Upon reception of the subsequent SYN packet, the server erroneously ignores the SYN flag in the packet. As a result, the server interprets the packet being out of order as it does not contain an expected ACK value of 1. Consequently, the server sends an ACK packet with the expected sequence number `Ack=1` back to the client. However, the client is still waiting for the SYN/ACK packet from the server and thus cannot handle the ACK packet. Therefore, it resets the connection by sending a reset packet to avoid any further inconsistencies. As seen in the timing diagram, the bug occurred due to wrong handling of subsequent SYN packets in the SYN_RCVD state of μIP.

Bug #3: After this bug was confirmed by the μIP maintainers, further discussion indicated a much more severe, third bug. For RAM constrained nodes allowing only one TCP connection a developer reported the following case: He experienced situations in which a deployed node got permanently stuck in the SYN_RCVD state, i.e., it did not leave this state nor accepted new connections. Hence, this bug caused a refusal to reconnect and required a reset.

However, it was not yet clear how this situation occurred. In the reported scenario, the deployed nodes were periodically queried with a script running in a web browser. Hence, during replay, we exchanged our μIP client application with a Linux telnet client and again triggered the SYN/ACK packet loss at the μIP server. In contrast to the μIP stack, the Linux TCP/IP stack does not reset the connection upon receiving the unexpected ACK packet during connection establishment. Instead, it retransmits the SYN packet until the maximum number of retransmissions is reached and announces a connection timeout to the application. Thus, the μIP stack of the server application gets stuck in the SYN_RCVD state forever. This was unexpected because the μIP stack implements a periodic timer for all active connections that are not in the CLOSED state.

During further analysis, we discovered the following bug: the periodic connection timer is started *only after* the first connection reaches the ESTABLISHED state. This finding explained why single-connection nodes eventually reach this dead state.

Discussion: Overall, KleeNet does not abstract from timer predicates, as these form an important building block



tcpclient:1025
tcpserver:7777
sending SYN
[SYN] Seq=0
Conn 1: SYN received, sending SYN/ACK
SYN/ACK received, sending ACK
[SYN, ACK] Seq=0 Ack=0
[ACK] Seq=0 Ack=1
Conn 1: ACK received. Connection established, starting periodic timers
SYN duplicate
[SYN] Seq=0
Bug: Conn 2: SYN received, sending SYN/ACK
[SYN, ACK] Seq=0 Ack=0
Client expects Seq=1. It ignores the SYN flag, interprets the packet being an out of order data packet, and sends ACK with the correct numbers
[ACK] Seq=0 Ack=1
Conn 1: ACK received, dropping unexpected packet
Periodic timer fires: Conn 1: No operation Conn 2: Retransmitting SYNACK
[SYN, ACK] Seq=0 Ack=0
...
[ACK] Seq=0 Ack=1
Conn 1: ACK received, dropping unexpected packet
Periodic timer fires: Conn 1: No operation Conn 2: Retransmitting SYNACK
[SYN, ACK] Seq=0 Ack=0
...
...
Periodic timer fires: Conn 1: No operation Conn 2: Max retransmissions, resetting the connection
[RST] Seq=1 Ack=0
Got reset, aborting the connection

Figure 7: Bug #4: A SYN packet duplicate leads to undefined protocol behavior.

of many protocol implementations. In contrast, other tools such as FSMGen [14] simulate timers only partially and hence are unable to catch communication scenarios similar to the one described above.

8.3 Case 3: Symbolic Packet Duplicates

In the following example we show how a duplicate packet erroneously opened a new TCP connection at the server due to a wrong handling of SYN packets.

Communication Scenario: After submitting the report of the previously described bug, the following patch was proposed by the developers. The idea was not to pass SYN packets to active TCP connections, i.e., connections that are not in the state CLOSED. This appeared to be the correct fix and KleeNet could not detect the previously described bug. After patching the μIP stack, we defined new tests based on symbolic packet duplicates.

Test Setup: The state consistency, i.e, our distributed assertions, are specified as in case 2. Hence, we test for failures in connection setup, i.e., failures due to timeouts or reset messages.

Based on symbolic packet duplicates, KleeNet generated a total of 14 distributed execution scenarios (see Table 4). Of these, nine scenarios triggered assertion violations. Their execution traces revealed that a duplicate SYN packet caused the connection establishment to fail.

Runtime	77 sec, 14–15 MB RAM (14 scenarios)
5 scenarios	TCP client connected
9 scenarios	TCP connection aborted

Table 4: Symbolic packet duplicates.

Bug #4: During replay, TCP exhibited a strange behavior resulting in a connection reset before a connection could be established. A duplicate SYN packet arriving at one particular point in time erroneously opened a new connection with the same connection identifiers, i.e., same IP addresses and port numbers. This resulted in the client receiving packets from both server connections, whereas the data from the client arrived at the first connection of the server only. Consequently, this led to the following undefined protocol behavior (see Figure 7): The first TCP connection was successfully established, but the second connection continuously retransmitted SYN/ACK packets to the client. As the μIP stack passes the incoming data to the first matching connection, the ACK packet from the client never arrived at the second connection. As a result, after reaching the number of maximum SYN/ACK retransmissions, the second connection sent a reset packet (RST) to the client.

Discussion: In this scenario, KleeNet used symbolic packet duplicates to detect a bug which was introduced by the proposed patch. Subsequently, we proposed a new patch that fixed the problem and appears to be resistant against packet losses and duplicates.

8.4 Case 4: Symbolic Node Outage

In the final example, we show how a node outage during an active TCP connection may lead to a stale protocol state within the μIP stack.

Communication Scenario: We use the same communication scenario as described in the previous examples and additionally enabled symbolic nodes outages.

Test Setup: To test handling of node outages in μIP, we defined the following state consistency check: If a node does not reach the not connected state after 2 minutes of communication, KleeNet should trigger the assertion violation.

```
/* 2 minutes timer handling code */
assert(connected);
```

Observation #5: KleeNet generated an execution path that violated our assertion. During replay, we noticed that the outage of the client after successful connection establishment caused this violation. The periodic timer at the server was still running and did not time out (see Figure 8).

After discussing this issue with the developers it turned out that this is not a bug in μIP. Contiki applications *must* be designed to check connections for activity. Thus, (1) a server application must send data on the connection to receive an RST packet if the client is back alive, or (2) employ a watchdog to reset the connection.

Discussion: Overall, our evaluation proves the flexibility and efficiency of KleeNet as a debugging tool. It is important to highlight that all the bugs presented in this paper are very difficult to detect using conventional pre- and post-deployment debugging techniques. For example, the behavior resulting from two of the bugs (bug #2 and #3) have long been observed during deployments and reported on the

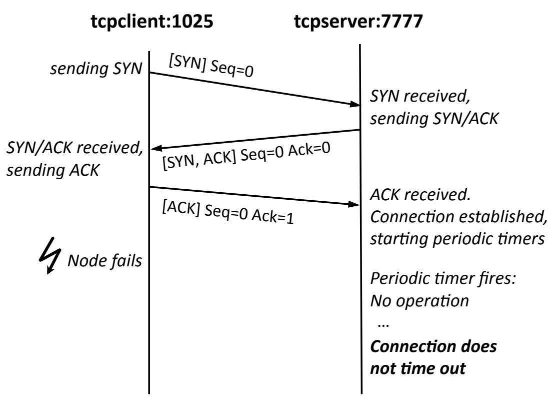

tcpclient:1025
tcpserver:7777
sending SYN
[SYN] Seq=0
SYN received, sending SYN/ACK
SYN/ACK received, sending ACK
[SYN, ACK] Seq=0 Ack=0
ACK received. Connection established, starting periodic timers
[ACK] Seq=0 Ack=1
Node fails
Periodic timer fires: No operation
...
Connection does not time out

Figure 8: Observation #5: Server connection does not time out.

mailing list. However, their root causes could not be found. KleeNet allows to specify simple distributed assertions to discover the root causes of such behaviors. Moreover, our evaluation illustrates that KleeNet is beneficial throughout the application development life cycle, particularly during regression testing by catching new bugs and verifying the absence of old ones.

8.5 Limitations and Discussion

Overall, we used KleeNet to test a variety of Contiki's protocols such as (`ARP`, `IP`, `ICMP`, `TCP`) and applications including `ping`, `telnet`, `shell`, `webbrowser`, and `webserver`. The distributed scenarios were composed of a small number of nodes, connected either directly or over a μIP router in between. During testing, we experienced the following issues with our debugging approach.

Symbolic Input: As discussed in Section 5.1, the runtime and memory complexity of KleeNet heavily depend on the type of applications and the number of symbolic inputs. Furthermore, symbolic non-deterministic failures double the number of particular execution paths. Hence, even with relatively small-sized symbolic inputs, we experienced that some applications quickly emerge into thousands of execution paths. However, by applying domain knowledge—as discussed in the example below—we can efficiently reduce this execution complexity to a reasonable level, and as a result, resolve this limitation of KleeNet.

For example, we ran a test scenario with three nodes running a `wget` on a `shell`, a μIP router, and a `webserver` application, respectively. Via the `shell` we sent a symbolic HTTP request of 80 characters to the `webserver`. Due to the character by character parsing of this symbolic request at the `shell`, KleeNet reached the configured 1GB memory cap after 22 hours of execution generating thousands of test cases. However, none of the test cases showed a connection request to the `webserver` because the symbolic execution was still busy with the symbolic request parsing. To limit the state explosion, we reduced the size of symbolic data to the requested filename (e.g. six characters) only. This allowed the client to establish HTTP sessions with the `webserver` and to receive HTML documents through the network.

Automatism: Manual effort involved in specifying symbolic data and distributed assertions illustrates another facet of KleeNet. A user has to possess the knowledge of the distributed application logic to setup a meaningful testing scenario correctly. Only the non-deterministic node and network failures are injected completely automatically. Nevertheless, we expect the developers to accomplish this manual step as symbolic input (e.g. the initial data packet) induces the most interesting distributed execution paths.

Application domain: As discussed in Section 7.1, KleeNet is not designed for MAC-level debugging, that would surely have a large impact on fast execution state growth. Rather, we see KleeNet as a protocol and application testing tool at small scales where the early testing process can effectively reveal interesting design and implementation details.

9. CONCLUSIONS

In this paper we presented *KleeNet*, an effective debugging approach enabling high-coverage testing of execution paths in distributed applications and protocol stacks. In WSNs, undesirable events such as node failures, packet corruption, loss and duplication might lead to complex interaction bugs or liveness violations. Based on symbolic input and automatically injecting non-deterministic failures, KleeNet generates high-coverage traces of WSNs and their applications. By default, KleeNet—as based on KLEE—detects memory and division by zero errors. For checking distributed states, the developers must specify distributed assertions. Once these assertions are violated, KleeNet allows to replay the failure scenarios. We demonstrated the effectiveness of KleeNet with four insidious interaction bugs discovered in Contiki's μIP protocol stack.

After concluding our ongoing integration of KleeNet into TinyOS, we plan to test more complex and large-scale distributed scenarios. Such scenarios will, for example, include collection and dissemination trees, point-to-point routing protocols, and 6LoWPAN. We aim to detect new bugs and to identify further scalability challenges and limitations of our approach.

Overall, KleeNet offers a testing environment enabling *permanent, high-coverage* code checking throughout the development life cycle of WSNs at low manual effort. Furthermore, we believe that KleeNet—by utilizing a seamless transition between testing, replay, and deployment—significantly eases automated testing of wireless sensor networks.

10. ACKNOWLEDGMENTS

Many thanks to Koen Langendoen for shepherding this paper and helping us to improve its structure and quality. This work is partly supported by DFG UMIC research cluster of RWTH Aachen University and DFG Emmy Noether Programme.

11. REFERENCES

[1] S. Anand, C. S. Pasareanu, and W. Visser. JPF-SE: A Symbolic Execution Extension to Java Pathfinder. In *TACAS*, 2007.

[2] P. Ballarini and A. Miller. Model Checking Medium Access Control for Sensor Networks. In *ISOLA*, 2006.

[3] G. Barrenetxea, F. Ingelrest, G. Schaefer, and M. Vetterli. The Hitchhiker's Guide to Successful Wireless Sensor Network Deployments. In *SenSys*, 2008.

[4] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of

High-Coverage Tests for Complex Systems Programs. In *OSDI*, 2008.

[5] Q. Cao, T. Abdelzaher, J. Stankovic, K. Whitehouse, and L. Luo. Declarative Tracepoints: A Programmable and Application Independent Debugging System for Wireless Sensor Networks. In *SenSys*, 2008.

[6] Y. Chen, O. Gnawali, M. Kazandjieva, P. Levis, and J. Regehr. Surviving Sensor Network Software Faults. In *SOSP*, 2009.

[7] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient Memory Safety for TinyOS. In *SenSys*, 2007.

[8] A. Dunkels. Full TCP/IP for 8-bit Architectures. In *MobiSys*, 2003.

[9] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *LCN*, 2004.

[10] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection Tree Protocol. In *SenSys*, 2009.

[11] J. W. Hui and D. Culler. The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In *SenSys*, 2004.

[12] M. M. H. Khan, H. K. Le, H. Ahmadi, T. F. Abdelzaher, and J. Han. Dustminer: Troubleshooting Interactive Complexity Bugs in Sensor Networks. In *SenSys*, 2008.

[13] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[14] N. Kothari, T. Millstein, and R. Govindan. Deriving State Machines from TinyOS Programs Using Symbolic Execution. In *IPSN/SPOTS*, 2008.

[15] V. Krunic, E. Trumpler, and R. Han. NodeMD: Diagnosing Node-level Faults in Remote Wireless Sensor Systems. In *MobiSys*, 2007.

[16] K. Langendoen, A. Baggio, and O. Visser. Murphy Loves Potatoes: Experiences from a Pilot Sensor Network Deployment in Precision Agriculture. In *WPDRTS*, 2006.

[17] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *SenSys*, 2003.

[18] 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*, pages 115–148. 2005.

[19] K. Liu, M. Li, X. Yang, and M. Jiang. Passive Diagnosis for Wireless Sensor Networks. In *SenSys*, 2008.

[20] L. Luo, T. He, G. Zhou, L. Gu, T. F. Abdelzaher, and J. A. Stankovic. Achieving Repeatability of Asynchronous Events in Wireless Sensor Networks with EnviroLog. In *INFOCOM*, 2006.

[21] P. C. Ölveczky and S. Thorvaldsen. Formal Modeling, Performance Estimation, and Model Checking of Wireless Sensor Network Algorithms in Real-Time Maude. *Theor. Comput. Sci.*, 410(2-3):254–280, 2009.

[22] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level Sensor Network Simulation with COOJA. *EWSN*, 2006.

[23] J. Polley, D. Blazakis, J. McGee, D. Rusk, J. Baras, and M. Karir. ATEMU: A Fine-grained Sensor Network Simulator. In *SECON*, 2004.

[24] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the Sensor Network Debugger. In *SenSys*, 2005.

[25] K. Römer and J. Ma. PDA: Passive Distributed Assertions for Sensor Networks. In *IPSN/SPOTS*, 2009.

[26] R. Sasnauskas, J. A. B. Link, M. H. Alizai, and K. Wehrle. Poster Abstract: KleeNet: Automatic Bug Hunting in Sensor Network Applications. In *SenSys*, 2008.

[27] L. Selavo, A. Wood, Q. Cao, T. Sookoor, H. Liu, A. Srinivasan, Y. Wu, W. Kang, J. Stankovic, D. Young, and J. Porter. LUSTER: Wireless Sensor Network for Environmental Research. In *SenSys*, 2007.

[28] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/FSE-13*, 2005.

[29] T. Sookoor, T. Hnat, P. Hooimeijer, W. Weimer, and K. Whitehouse. Macrodebugging: Global Views of Distributed Program Execution. In *SenSys*, 2009.

[30] R. Szewczyk, J. Polastre, A. M. Mainwaring, and D. E. Culler. Lessons from a Sensor Network Expedition. In *EWSN*, 2004.

[31] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: Scalable Sensor Network Simulation with Precise Timing. In *IPSN/SPOTS*, 2005.

[32] G. Tolle and D. Culler. Design of An Application-cooperative Management System for Wireless Sensor Networks. In *EWSN*, 2005.

[33] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong. A Macroscope in the Redwoods. In *SenSys*, 2005.

[34] P. Völgyesi, M. Maróti, S. Dóra, E. Osses, and A. Lédeczi. Software Composition and Verification for Sensor Networks. *Sci. Comput. Program.*, 56(1-2):191–210, 2005.

[35] M. Wachs, J. I. Choi, J. W. Lee, K. Srinivasan, Z. Chen, M. Jain, and P. Levis. Visibility: A New Metric for Protocol Design. In *SenSys*, 2007.

[36] Y. Wen, R. Wolski, and S. Gurun. S^2DB: a Novel Simulation-based Debugger for Sensor Network Applications. In *EMSOFT*, 2006.

[37] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and Yield in a Volcano Monitoring Sensor Network. In *OSDI*, 2006.

[38] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: Using RPC for Interactive Development and Debugging of Wireless Embedded Networks. In *IPSN/SPOTS*, 2006.

[39] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *NSDI*, 2009.

[40] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: a Comprehensive Source-level Debugger for Wireless Sensor Networks. In *SenSys*, 2007.