Implementing Secure & Reliable Software

Seminar - SS 2014

Topic Descriptions

OVERVIEW
Languages for Specifying Safety-Critical Software Requirements ........................................ 2
Fault-tolerant Wireless Sensor Networks .................................................................................. 2
Parallel Symbolic Execution ...................................................................................................... 2
Exploiting Locality in Peer-to-Peer Overlay Networks ............................................................... 3
Analysis and Verification of Runtime Behavior of Automotive SW ........................................... 3
Model-based Security Testing .................................................................................................... 3
The Role of Abductive Reasoning in Dealing with Embedded System Failures ......................... 4
Trusted Applications on Untrusted Operating Systems .............................................................. 4
Verifying failure mitigation mechanisms of safety critical systems .......................................... 5
Synchronization Avoidance Mechanisms and Consistency Semantics ....................................... 5
An Empirical Comparison of the Fault-Detection Capabilities of Internal Oracles ...................... 6

Contact
Thorsten Piper
Tel. 06151-16-3414
Room: S2|02 E219
piper@cs.tu-darmstadt.de
Languages for Specifying Safety-Critical Software Requirements

The verification of a software implementation (e.g. by testing or formal methods) requires a specification that describes how the implementation should behave. Especially for formal verification attempts, specification languages with precise semantics are required. Several such specification languages exist and some are frequently advocated for use in safety-critical software engineering, as they ease verification. “Z” is one such language. The task of this seminar report is to provide an overview on formal specification languages suitable for the development of safety-critical software, their core features, and application fields.

Fault-tolerant Wireless Sensor Networks

Wireless Sensor Networks (WSNs) are prone to node and link failures and unexpected communication delays. The reliability of WSNs becomes especially important when they are to be employed in a critical infrastructure system. In this seminar, we study how to make the communication, routing and processing in WSNs resilient to faults and failures.


Parallel Symbolic Execution

In the realm of implementation of dependable and reliable software, formal verification constitutes a major phase in the development process. Although model checking has proven to be efficient in finding bugs and flaws in real systems it remains of limited use due to the fact that it requires concrete inputs from the user. This means that in order to completely verify a software, one has to run the model checker for every possible input. Symbolic execution, on the other hand, is a technique that works using the same principle as model checking apart from the fact that it covers all possible input ranges in one run of the verification. Although not trivially realizable, the parallelization of Symbolic execution can considerably speed up the verification process. In this seminar report, the student is required read about the concept of symbolic execution, understand the challenges faced to implement it and discuss some existing work.

Exploiting Locality in Peer-to-Peer Overlay Networks

Peer-to-Peer protocols represent the basis of numerous applications. As such systems may span across continents or the whole world, various approaches exist to improve location based searches and overlay maintenance, which consider metrics based on network latencies, geographic information, or operator domains. The student should survey recent approaches and address in particular their performance and reliability aspects.


Analysis and Verification of Runtime Behavior of Automotive SW

AUTOSAR is an evolving standard for automotive software architecture, featuring a component-based design throughout its different abstraction layers (OS, Runtime Environment and Application). At this point, AUTOSAR components do not have any standardized timing relation with their tasks. This causes unpredictable and non-deterministic runtime behavior, which can only be analyzed and verified after integration phase. Within this seminar work, current approaches to runtime analysis (static or experimental) of embedded automotive systems shall be surveyed, especially focusing on worst-case execution time (WCET) analysis. An introduction to common scheduling algorithms employed by the operating systems of these systems will also be part of the work.

Model-based Security Testing

Software security is a software quality issue that continues to grow in importance as software systems manage continually increasing amounts of critical corporate and personal information. Testing helps to verify whether the behavior of a product or system conforms to the security features claimed by the manufacturer. Presently, developing and executing security functional tests is time-consuming and costly. Consequently, methods and tools for automating security functional testing are developed, relying on a model-based approach to automate security functional testing. Within this seminar work, the range of model-based security testing approaches shall be surveyed and compared, also emphasizing their benefits and drawbacks against non model-based testing approaches.
The Role of Abductive Reasoning in Dealing with Embedded System Failures

While deductive and inductive reasoning are fairly well understood and frequently applied concepts in logical discussions, arguments, and formal proofs, abduction is less frequently discussed. Just as induction, abduction is a form of logical inference under uncertainty, introduced as a complementary concept to deduction and induction by the American mathematician Charles Sanders Peirce. The core idea of abduction is that if there is an unexpected observation (such as a system failure), humans can find an explanation for that observation so that the observation directly follows as a deductive inference from what they know about the world they are living in. This generation of an explanation for an observation is called abduction or abductive reasoning. Abductive reasoning is an important ability for finding effective work-arounds for system failures: abduce the root cause of the failure, then circumvent it. For embedded systems, especially in ubiquitous computing applications, this way of crafting workarounds is complicated. A major goal of the engineer is to embed the system in its environment, i.e. to make it indistinguishable from what it is embedded in. This clearly hampers abducability. The root cause of an observed failure is hard to see, because the embedded system, which is its source, is hidden. The goal of this seminar report is to give an overview of this problem, its implications, and possible solutions.

Trusted Applications on Untrusted Operating Systems

Applications that need to be trusted, e.g., because they handle sensitive data, can be thoroughly designed, tested and validated to be resistant against attacks. A development process that incorporates trustworthiness offers a certain degree of confidence that the developed applications are indeed trustworthy. However, such applications are often executed on commodity operating systems (OSs), which usually contain a large body of unverified trusted code including the kernel itself as well as device drivers and other system software. Since the OS controls all system resources and applications have to trust the OS, a compromise of the OS usually also compromises all trusted applications. In recent years, systems [1, 2] have been proposed that allow for trustworthy applications even if the OS they are executing on is not trustworthy, compromised or even hostile. The objective of this seminar is to discuss and compare two or more recent approaches [1, 2] that protect applications against untrusted operating systems.

**Verifying failure mitigation mechanisms of safety critical systems**

As the failure of safety-critical systems could potentially result in hazardous behavior (e.g., loss of life, damage to the environment or property), these systems usually include mechanisms for the mitigation of safety related failures. In this seminar work, the student shall survey different techniques for the verification of failure mitigation mechanisms and discuss them comparatively.

**Synchronization Avoidance Mechanisms and Consistency Semantics**

Replication is a widely used technique to achieve fault tolerance and high availability. This comes with the cost of managing the states of the replicated services, also called replicas, such that they remain consistent. Different notions of consistency exist. For instance, strong consistency requires that the replicas remain exact copies of each others at any time. To guarantee such a strong condition replicas have to synchronize after each operation, which results in a considerable overhead. Eventual consistency, on the other hand, depending on the nature of the operation request received from the client, synchronization might not be required, and it is sufficient to assume that the state of the replicas will eventual converge to the same state. Depending on nature of the request received from the client synchronization might not be needed to achieve a certain level of consistency. In a effort to achieve better performance, a hybrid approach exist where strong consistency and eventual consistency co-exist and enforced according to the nature of the involved requests. A major problem in designing such an approach lies in the decision whether an request should be strongly consistent (i.e., requires synchronization) or eventually consistent. It turns out that the decision, in most cases, cannot be statically made and thus a dynamic solutions are of interest. In this seminar, the student is required to investigate different approaches to implement the hybrid combination of strong and eventual consistency and therefore avoid synchronization when not needed and discuss the pros and cons.


An Empirical Comparison of the Fault-Detection Capabilities of Internal Oracles

Modern computer systems are prone to various classes of runtime faults due to their reliance on features such as concurrency and peripheral devices such as sensors. Testing remains a common method for uncovering faults in these systems, but many runtime faults are difficult to detect using typical testing oracles that monitor only program output. In this seminar work, the student shall evaluate the effectiveness of fault detection of internal test oracles, which monitor aspects of internal program and system states, and provide a comparison to output-based oracles.