Fast Simulation Based Testing of Anti-Tearing Mechanisms for Small Embedded Systems

Johannes Loinig, Christian Steger, and Reinhold Weiss
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
Email: \{johannes.loinig, steger, rweiss\}@tugraz.at

Ernst Haselsteiner
NXP Semiconductors Austria GmbH
Gratkorn, Austria
Email: ernst.haselsteiner@nxp.com

Abstract—Small embedded systems are often powered by unreliable power supplies like energy harvesting systems (e.g., for sensor nodes) or external power supplies (for smart cards). For secure embedded systems a sudden loss of power can violate data integrity. The power has just to drop when data is written to non-volatile memory. Thinking about a byte array in a smart card representing some digital money of an e-purse, this becomes obvious. In order to guarantee data integrity a secure embedded system has to provide an anti-tearing mechanism. Testing this mechanism is very difficult, extensive, and requires deep inside knowledge into its implementation details.

In this work we show how a simulation of an embedded system can be used to test the anti-tearing mechanism. High-level test cases are used to generate test vectors automatically. The proposed approach allows a fast and comprehensive test of the anti-tearing mechanism. We\textsuperscript{1} explain our proposed mechanism on the basis of a case study of a smart card system. However, the mechanism is general enough to be used for secure embedded systems of any kind.

I. INTRODUCTION

Small embedded systems have often no reliable power supply. Typical examples are sensor nodes with energy harvesting systems and smart cards which are externally powered by the smart card reader. In the first case the gained power is dependent on the environment wherein the system operates, e.g., the lighting conditions in case of photovoltaic cells. In the case of smart cards, the user may just remove the card form the reader. In both cases the execution of the embedded application will be aborted immediately. Consequently, the application can not rely on its execution to its regular end.

However, it is important that a system is able to guarantee integrity for information stored in persistent memory. This is especially important if the system is used for secure use cases as banking cards or e-government applications. For example, the amount of digital money stored on a banking card must stay consistent in any case of operation. To guarantee the necessary data integrity in case of sudden power loss most smart cards have to support atomic write operations to persistent memory. Anti-tearing mechanisms, or also called transaction mechanisms (TM), provide this functionality.

\textsuperscript{1}This paper is a result of a project which is funded by the Austrian Federal Ministry for Transport, Innovation, and Technology (contract FFG 816464).

A TM is usually part of the operating system (OS). Therefore, an application does not need to take care about tearing-events on its own. For example, in case of Java Cards [1] (a smart card system including a Java virtual machine) each assignment to a variable located in the persistent memory is defined to be atomic.

Testing a TM can be very difficult and may require a lot of deep knowledge about the implementation details. Additionally, the TM runs on an embedded system which is supposed to be a secure environment. Hence, observability and controllability of the system for testing purposes is strongly limited. Having a reliable, implementation-independent, and rapidly executed test in place would be a great benefit.

In this work we propose an approach which fulfills these requirements. Test vectors are generated automatically from high-level test cases which are independent from the TM’s implementation. A fast simulation based execution allows a rapid and simple test setup and execution. Furthermore, we evaluate the performance of the test mechanism in comparison to a common fault injection based testing approach.

The remainder of this paper is structured as follows. First, we summarize related work, explain anti-tearing mechanisms, and give our motivation. Then, we describe a simulation based fault injection method which is the basis for further comparison. After that, we propose our test vector based approach including test vector generation and compaction. Finally, we introduce our proof of concept implementation and evaluate our proposed approach on theoretical basis and simulation results.

II. RELATED WORK AND MOTIVATION

The requirement to write data atomically to persistent memory is well known for smart cards. However, as already mentioned, it is not restricted to them. Smart cards are very small embedded devices with limited computational resources: some few hundred bytes of RAM and some few hundred kilobytes of ROM and persistent memory (typically EEPROM). Smart cards are usually powered via inductive coupling by the smart card reader what raised the need for reliable and efficient TMs.

In the following section we explain some published TMs for smart cards. This list is not complete, there are more

1
mechanisms published especially in patents. Afterwards, we summarize related work about verification methods for atomicity.

A. Anti-Tearing Mechanisms

The basic concept of the introduced anti-tearing mechanisms is a backup strategy. Simply overwriting values in persistent memory is very insecure [2]. If the execution is interrupted (e.g., caused by a breakdown of the power supply), the old value may be deleted, but the new value may not yet be in a consistent state. Thus, the basic principle is to make a backup of important data before writing the new data. When the system boots up it searches for existing backups. If a backup exists (in case of a tearing event) the system restores the data to bring the memory back to a consistent state. For the remainder of this paper we call these phases the Backup Phase (BP) and the Restore Phase (RP).

Oestreicher explains two basic mechanisms in [3]: Old Value Logging (OVL) and New Value Logging (NVL). Both mechanisms use a transaction buffer (TB) in the persistent memory to store the backup information. In the BP OVL saves the old value going to be overwritten, its type, and its address to the TB. This is a roll-backward strategy as the old value can be restored in case of a tearing event. NVL saves the new value, type, and address to the TB. Thus, the RP is able to finalize the write operations if necessary. This is a roll-forward strategy.

The patent [4] describes a different concept based on memory redundancy. The persistent memory is split into multiple parts. Each data field is allocated in all these parts. A write operation does not overwrite the old data directly but stores the new values in an according data field on a different memory part. If the write operation was successful the field’s reference is changed to the new location. The old data field may be overwritten with the next atomic write operation. The critical write operation in this mechanism is the update of the reference. As the reference to persistent data fields is also located in the persistent memory it can not be overwritten without tearing countermeasures. An OVL or a NVL may be used to update the reference in a secure way.

B. Verification of Atomicity

Verification and testing of software in small embedded systems is very crucial. The software (e.g., the operating system) is often stored in the ROM during fabrication. Thus, it is practically impossible to update the software in case of a bug that should be fixed [5, page 574]. Additionally, highly secure use cases require a fully trustworthy implementation which makes testing even more important.

In [5, page 586] it is mentioned that testing the TM is a typical black box test. An atomic operation is executed and interrupted at multiple specified points in time. After each interruption it is checked if the system is still in a consistent state. This is explained in more detail in Section III. However, it is mentioned that the number of needed tests is fairly large.

There exists a lot of related work around the TM for Java Cards [1], [6]. The authors of [7] and [8] used the Java Modeling Language to model the Java Card TM. The concepts allow formal verification of atomicity of Java Card applications. However, a fully functional and reliable TM is assumed to be in place when the verified application is executed.

In [9] the author introduces a method to formally prove embedded C code of anti tearing mechanisms. First, a transition system is extracted automatically from a formal specification of the atomic function to be verified. Then, a program verification tool (Caduceus) is used to evaluate if the implementation fulfills the transition system. The verification tool requires that the C code is annotated with functional properties like the precondition and the post-condition.

Simulation based testing is common for hardware development. Simulated fault injection allows observing the system in situations which are difficult to reproduce on real hardware. Rothhart et al. has proposed a high-level fault injection approach for smart cards in [10]. The system was modeled in SystemC [11]. Faults can be injected into functional blocks and interconnections without recompilation of the model.

A simulation based test environment must provide a suitable performance if it should be used during the development process. Misera et al. describes how to achieve higher testing performance for SystemC based fault injection [12]. The basic principle of the performance increase is to parallelize tests for rather low level modules.

C. Motivation of our Work

We were not able to find related work about fast and reliable verification of transaction mechanisms. The found formal verification methods are done on a high abstraction level. As great parts of operating systems are implemented at least partially in assembler these approaches may not be applicable for verification during development. In addition these approaches require additional high-level information like functional properties. This information may not always be available during a development process.

Fault injection is a promising approach. A tearing event does not need a complicated fault model and can be easily applied on a simulated embedded system. However, we will show in this work that a fault injection based approach only may not be the best choice. First, we explain the basics of the fault injection based approach. After that, we explain how the test performance can be increased by reduction of code that is executed by the simulated system. Furthermore, our approach allows a reduction of test cases which is not easily possible with a basic fault injection based approach.

III. A SIMULATION BASED FAULT INJECTION APPROACH

In this section we explain how to move a common test for anti-tearing mechanisms for smart cards explained in [5] to a simulation based environment. Furthermore, we identify the advantages and disadvantages of this proposed test method.

A. Fault Injection Without Simulation

As mentioned before, [5] describes a black box test for anti-tearing mechanisms of smart cards. Figure 1 shows this
basic test approach. The host system sends a command that causes the smart card to start execution of the application. The host turns off the power supply of the smart card at multiple points in time \( t_i \). This aborts the execution on the smart card’s application. The implementation of the test seems to be trivial.

In detail, it may be difficult to determine the ‘right’ values for \( t_i \). Notice that a secure embedded system is designed not to leak information which may be used for attacks. Thus, the choice of values for \( t_i \) can be, in the worst case, more or less just guessing. Additionally, as this is a pure black box test, not only the right values have to be guessed, the right number of tests is also difficult to determine.

It should be mentioned that just switching off the reader may turn up as not so easy as expected. A commercial smart card reader may not provide a command to switch off the power supply. Furthermore, smart card readers do usually not provide a mechanism to define exact arbitrary timing for commands as the timing is defined by the underlying protocol like ISO 14443.

In detail, the explained approach is more complex. The fault injection module as shown in Figure 2 must be able to decide if a fault injection should be executed. Therefore, it has to identify the write operations and distinguish between:

- write operations that have to cause a tearing event,
- write operations that already have caused a tearing event and are thus tested,
- and write operations that should not cause a tearing event because they are out of scope of the active test case.

Latter ones are, for example, write operations in the RP. Notice that an interrupted BP is followed by a RP and the RP also includes write operations which may be necessary to be tested in a separated test case.

### C. Estimation of the Test Effort

The number of test cases strongly depends on the implementation of the embedded system. Different data types which can be written in an atomic way may have different implementations in the TM. Thus, they have to be tested separately. Additionally, the system can provide different kinds of atomic write operations. Java Cards, for example, provide atomicity for single variable assignments and for groups of assignments. Latter ones are called transactions [1] and are encapsulated within the `JCSYSTEM.beginTransaction()` and `JCSYSTEM.commitTransaction()` function calls. Finally, there may be functions like `arrayCopy()` that are atomic by definition. Each of these write operations for Java Cards are conceptually different. It is obvious that they may be implemented at least partially independently and that they require different test cases.

Furthermore, these test cases consist of multiple write operations and each of these write operations results in a simulated tearing event. Last but not least, each test case starts again from the beginning after a tearing event (after the RP). Thus, huge parts of the system are simulated again and again.

Therefore, the test effort can be very high in total. This is a clear drawback of the proposed approach. We describe a more efficient test mechanism in the next section.
IV. A TEST VECTOR BASED APPROACH

For our proposed test vector based approach we define the internal state of the embedded system by the content of the TB. The state is evaluated during the RP. Both, BP and RP are modifying the state during their execution. Figure 3 shows the state diagrams for the BP and the RP. The BP always starts in the invalid state and ends in the invalid state if the BP is executed to its end without occurred tearing event. The RP may start at any state because it must be assumed that a tearing event has interrupted an atomic write operation. If the RP is executed without interruption invalid is the final state.

The state diagram in Figure 3 includes the state written and partially valid. Latter state is needed if more than one data field must be written like for mentioned Java Card transactions. In such a case the backups are written and validated sequentially one after another. However, the state diagram is very general and does not include states for different data types.

As shown, the BP does not depend on the state of the TB. The executed instructions in the BP only depend on the data which has to be written. This data is defined by the test case. In the strict sense the BP also depends on the initial state of the persistent memory. This initial state can easily be pre-defined and is therefore out of interest for our test approach.

Contrary to the BP, the RP depends on the state of the TM only. The test case can only influence the RP by the content of the TB. This is obvious as the RP runs at startup of the system before a test case is executed.

A. Test Vector Generation

In our proposed vector based test approach we take advantage of these two properties. If the repeated execution of code leads to the same write operations, and therefore to the same TB state, there is no need to run it several times. The RP would perform anyway the same operations as before. Therefore, we run code that is modifying the TB’s state only once and record the write operations. These records represent the TB’s state changes. Afterwards we use the obtained write operations to create test vectors. Each test vector contains addresses of persistent values and related new values to be changed before the test is executed. An overview is shown in Figure 4 a).

![Diagram of test vector generation process]

Fig. 4. A Test Vector Based Approach. a) Test Vector Generation b) Test Execution

The test case is executed only once. All write operations to the persistent memory are reported by the simulation. One tearing event may be simulated to run the RP and to store its write operations as well (not shown in the figure). Afterwards the stored write operations are combined cumulatively to test vectors as shown in Figure 5. This is done to generate test vectors that are independent from each other.

Because the test vectors are independent it is possible to run them in any arbitrary order. This also allows to skip or merge test vectors which are identified to cause execution of identical code parts. The compaction process shown in Figure 4 runs a static analysis on the test vectors and identifies test vectors that can be skipped. This may require that the compaction process considers implementation details of the TM.

B. Test Execution

To apply a test vector, its associated write operations are performed by the simulation environment like shown in Figure 4 b). Notice that these write operations are executed by the simulation environment and not by the simulated model. Therefore, this can be performed very fast. Afterwards the simulated system starts up, the RP is executed, and the consistency check can be done. This is repeated for each test
vector and is equivalent to the model’s simulation after an injected tearing event described in Section III.

This proposed test method requires less code to be executed in the simulated system. The simulation environment must be able to record write operations to the persistent memory and it must provide a mechanism to write data to selected addresses without running the model. Both requirements are independent from the TM. Therefore, the test vectors can be generated and applied without knowledge of implementation details of the TM.

V. IMPLEMENTATION

We implemented our test environment for the anti-tearing mechanisms of a complete Java Card OS. The OS runs on a cycle accurate and memory accurate instruction set simulation and provides NVL and OVL functionality. However, cycle accuracy is not needed for our proposed approach. The simulated smart card was modeled with SystemC.

The test cases are implemented in a Java Card application. Furthermore, this application also includes the consistency checks after the RP. Several test cases for different types (byte, short ...) and atomic write operations (transactions, arrayCopy() ...) are implemented.

The fault injection module described in Section III and the record functionality and test vector generation explained in Section IV are implemented in the EEPROM module of the smart card’s SystemC model. Test vectors are stored byte-wise in a file. So far, the appliance of the test vectors is done manually.

The test vector compaction is just done rudimentarily as the detailed compaction process is out of scope of this work. Only test vectors that are identical (e.g., if the TB is erased) are combined in our proof-of-concept implementation.

VI. EVALUATION AND EXPERIMENTAL RESULTS

First, we want to evaluate if our proposed test vector based approach is in general faster than a simple fault injection based approach. Equation 1 describes the time $t_{fi}$ which is needed to execute test cases with the fault injection approach where:

- $n$ is the number of test cases emerging from the different data types and write operations,
- $x_{BP}$ and $x_{RP}$ are the number of write cycles during BP and RP,
- $t_{BP_i}$ and $t_{RP_i}$ are the execution time of BP and RP until the $i$-th tearing event is injected,
- and $t_{BP}$ and $t_{RP}$ are the execution time of BP and RP in total.

The two sums in the equation represent the partially executed (interrupted by a tearing event) test runs for the BP and the RP. The last term represents the execution of the RP for every injected tearing event.

$$t_{fi} = n \cdot \sum_i t_{BP_i} + \sum_i t_{RP_i} + (x_{BP} + x_{RP}) \cdot t_{RP}$$  \hspace{1cm} (1)

Equation 2 describes the time $t_{tv}$ which our proposed test vector based approach needs for execution. In this equation $v$ is the number of generated test vectors. The worst case is that $v = x_{BP} + x_{RP}$ if no test vector compaction is performed and all test vectors are used. For the further evaluation we assume this case.

$$t_{tv} = n \cdot (t_{BP} + t_{RP} + v \cdot t_{RP})$$  \hspace{1cm} (2)

If we use these equations to evaluate $t_{fi} > t_{tv}$ we come to the result shown in Equation 3. This equation holds for all known TMs based on backup strategy. Therefore, we can argue for all TMs based on a backup strategy, that our test vector based mechanism is in general faster than a fault injection mechanism.

$$\sum_i t_{BP_i} + \sum_i t_{RP_i} > 0$$  \hspace{1cm} (3)
We executed both of our proposed testing approaches with the test environment described in Section V. The experimental results are shown in Figure 6. Notice that the absolute simulation times are out of interest as they are anyway strongly dependent on the abstraction level of the model and the performance of the simulation environment. Therefore, we have normalized the values in the graphs: the shortest time value for the fault injection approach is normalized to 1. All other values are relative to this basis value.

The graph in Figure 6 shows the expected linear behavior explained in Equation 1 and Equation 2. As can be seen, our test vector based approach scales much better than the regular fault injection method. For 16 test cases with more than 70 write operations, respectively simulated tearing events or test vectors, we achieved a significant performance gain of more than 20%.

Figure 7 shows the measured and normalized execution time for testing after the test vector compaction. As compaction reduces the number of test vectors, and thus the number of write cycles, we chose the number of test cases for the x-axis. However, these results have to be interpreted with care. The result of the test vector compaction strongly depends on the choice of test cases. Thus, it would have been possible to construct or select test cases with better or worse results as shown in the graph. Therefore, the graph is just given for seek of completeness to show that even a very simple compaction method results in a notable performance gain. For our very basic compaction method we achieved up to 13% faster execution times in comparison to the test vector based approach without compaction.

VII. CONCLUSION

In this work we explained the importance of anti-tearing mechanisms for embedded system without reliable power supply. Furthermore, we summarized existing mechanisms and common basics. These basics were used to propose two testing approaches: a simulation based fault injection approach educated from a very similar black box test, and an implementation independent test vector based approach.

The test vector based approach reduces unnecessary code re-execution during testing without loss of test coverage. Thus, it is able to achieve better performance for a large number of test cases. The applied test vectors are generated implicitly by the system under test. This allows implementation independent testing of the TM. Furthermore, our approach allows reducing the number of test cases by test vector compaction.

We evaluated our proposed mechanism on a theoretical basis and by usage of a proof of concept implementation of a Java Card test environment based on a SystemC simulation. The results show that a significant performance gain can be achieved without loss of test coverage.

REFERENCES