Spectrum-Based Fault Localization in Deployed Embedded Systems with Driver Interaction Models*

Ulrich Thomas Gabor[1][0000−0003−4143−2400], Simon Dierl[1][0000−0001−9730−9335], and Olaf Spinczyk[2]

[1] TU Dortmund, Department of Computer Science, 44227 Dortmund, Germany
{ulrich.gabor,simon.dierl}@tu-dortmund.de
[2] Osnabrück University, Institute of Computer Science, 49090 Osnabrück, Germany
olaf.spinczyk@uni-osnabrueck.de

Abstract. Software faults are still a problem especially in deployed systems. We propose a new approach to monitor a deployed embedded system with another embedded system, which acts autonomously and isolated. The monitoring system generates reports containing probable fault locations, which can later be obtained without requiring expensive debugging hardware or continuous access to the monitored embedded system. For this, we assessed failure-detection oracles, transaction detectors and suspiciousness metrics and evaluated them in a practical combustion engine scenario. Especially, we propose a driver interaction model to capture correct interaction with periphery and use it as oracle. Our results show that for the repetitive behavior of an engine control unit, simple approaches perform best.

Keywords: Reliability · Fault Tolerance · Software Reliability · Embedded Software · Software Quality

1 Introduction

Despite the continuous efforts of software engineering researchers, software faults are still a problem in modern software development [19] leading to failures. They can bring down spacecrafts [27] or whole data centers[3] and with the increasing number of pervasive embedded systems in households, e.g., smart speakers, failures are becoming more in absolute numbers due to the sheer number of devices.

While debugging techniques have improved in recent years, for example due to better update mechanisms and systematic capture of crash/trace logs, regularly there is still a lot of manual effort required to deal with logs, bug reports and traces. Since manual labor is not only time-consuming, but also expensive,

* The final authenticated version is available online at https://doi.org/10.1007/978-3-030-26601-1_7

[3] https://status.aws.amazon.com/s3-20080720.html

the vision is to automatically assess such bug-related data and pinpoint the most probable fault location for the expensive skilled developer.

Spectrum-based fault localization (SBFL) [24] is one approach serving this purpose and is based on the fact that if a failure is observed, the faulty component of a program must have been executed and therefore should be present in information regarding this run. A program **spectrum** [24] entails execution information from a specific perspective, e.g., which components were executed. Such multiple spectra can be obtained for multiple runs together with an error vector containing if the corresponding run failed or succeeded.

The idea is to find a relation between the spectra and the error vector, such that components can be ranked for examination. The function mapping spectra and an error vector to a **suspiciousness** [15] is called **suspiciousness metric** [31]. For efficient computation, some suspiciousness metrics depend only on aggregated information. For each component an **aggregated matrix** [3] can be constructed, which lists the four counts for "component was or was not executed in a succeeding or failing run".

These definitions can be modified in granularity such that a component is not a module or file, but a called function. Another variant is that a run is actually not a full run, but a run is split into transactions and each transaction is assessed as failing or succeeding.

1.1 Motivation

Observing already deployed embedded systems can be challenging, because access to deployed systems is often restricted, for example, because the device is not accessible physically. Attaching debuggers to every deployed system may also not be an option due to regularly high costs of debuggers and it might not be possible to reach the devices at all, besides scheduling an on-site appointment.

Our motivation is to be able to assess information on another small, cheap embedded system which we will call monitor and only transmit or save an aggregated fault localization report. These reports can then be transmitted using little bandwidth or can be downloaded at the next on-site inspection.

1.2 Requirements

The proposed idea to generate fault localization reports directly in the field should fulfill multiple requirements. First, the reports should obviously assist in localizing bugs. Second, the monitoring should work autonomously, without manual intervention, at least until the reports are obtained. Third, the monitoring must operate in isolation, because a failure in the embedded device must be assessed properly and should not lead to a failure of the monitor and vice versa. As an additional bonus, it should be possible to monitor arbitrary existing software without the need to change the source code of the monitored application.

1.3 Contributions

In this paper we present a new approach, which fulfills the aforementioned requirements. We use an additional embedded system to monitor the physically isolated application-under-test. Further, we assess multiple spectrum types and suspiciousness metrics regarding their suitability for the proposed use case and evaluate them in the context of control software for a simulated combustion engine. Since the monitoring should work autonomously, we also need an oracle to determine if a failure actually occurred and for long-running systems the execution must be split into shorter so-called transactions by a transaction detector [8]. To the best of our knowledge, we are the first to consider machine-learned driver interaction models as failure-oracle in a spectrum-based software fault localization resulting in suspiciousness rankings. These models allow us to detect failures based on the modeled communication with periphery. Finally, we use the AspectC++ compiler, which provides aspect-oriented programming (AOP) features for C++ [26], to instrument the application-under-test and make it transmit information to the monitoring device without forcing the developer to modify the source code.

1.4 Paper Organization

We will list related work in Section 2 and our new approach performing fault localization on an deployed embedded system in Section 3. Section 4 will demonstrate why our approach is feasible for our specific use case and in Section 5 we will name threats to validity, since our use-case is quite specific. Finally, Section 6 will summarize our findings.

2 Related Work

Fault localization can be done using multiple approaches. One of these approaches is model-based diagnosis. Abreu et. al use observations to construct a propositional formula, but finding an assignment, i.e., a faulty component, boils down to find a minimal hitting set [2], for which even approximations require noticeable computational power. Other approaches require manual modeling of the expected behavior [14], which can be a complex and error-prone task. Model-based approaches are therefore not the best choice for our setting.

Another approach is based on coverage-based techniques, which often require to compute full or dynamic slices, showing which statements of a program modify a variable or were executed. Xiaolin et. al have combined execution coverage of statements based on test cases with execution slices in a prototype implementation called HSFal to improve suspiciousness metrics and found their approach to reduce the average cost of examined code around 2.98 % – 31.79 % [16]. While the idea seems reasonable to us, slices are statement-based and that will likely exceed the available memory on small systems.

The most promising approach for our setting are spectrum-based fault localization approaches, where only a subset of the observable information of an application is taken into account, therefore we used this method. This approach has already successfully been used for embedded systems by Abreu et. al [5], but they inserted detectors manually in source code, which requires human effort and knowledge of the functionality of the application. Others regularly use unit tests or metamorphic testing [32] for failure detection in case unit tests are not possible. There are also approaches which combine spectrum-based with model-based techniques [3], but they also require more computational power than is available on low-cost microprocessors.

One core-component of classical spectrum-based approaches is the choice of an appropriate suspiciousness metric, where the so-called Ochiai metric outperformed other metrics when applied in real scenarios [4,18]. Recent approaches tried to combine multiple metrics by learning a weight from previous faults and applying the weights to compute suspiciousness for new faults [33], but this again requires computational power not available in the field of embedded systems. It might be possible though to perform the learning phase before deploying the system, but we have not yet examined this. Instead we just selected multiple existing metrics and compared them.

3 Methodology

Our overall methodology is shown in Fig. 1. On the left side the system-under-test is shown, which was augmented with a tracing component, transmitting trace data to the monitoring machine. In our case we have used AspectC++ to inject the tracing seamlessly into an existing application. Although this does not have to be done with AOP, it has the advantage of injecting trace code into the application without modifying the original source code.

The monitoring machine on the right performs all of the fault localization functionality. The continuous stream of information provided by the target machine is assessed by an oracle to decide if the current transaction is succeeding or failing. At the same time the stream is analyzed by a transaction detector to check if the data belongs to a new transaction, in which case the aggregation unit is informed. All information is aggregated to save space and a report based on the obtained information can be generated.

The report should rank software components according to their suspiciousness for being responsible for a failed run. A human can then inspect the components in that order.

3.1 Methodology Variants

Our method is not concrete about the used mechanisms as oracle or transaction detector. In fact, we will evaluate multiple variants in Section 4. Therefore, we will introduce multiple concepts in the rest of this section, which are evaluated regarding their individual efficacy later.

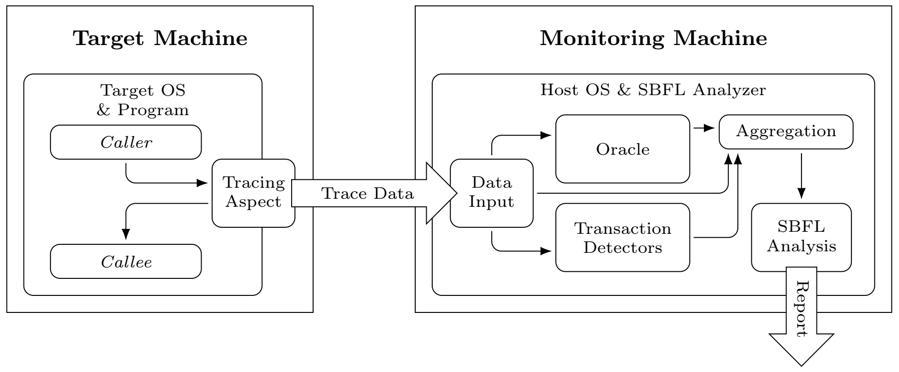

Fig. 1. High-level architecture of the analysis framework yielding a fault localization report

3.2 Observed Entities For Spectrum Generation

Section 1 introduced the concept of "components" in different granularity. We will use the called functions of a program as "components" to be able to pinpoint faults with little human effort.

Additionally, we will assess the benefit of using the extension **method call sequence hit spectra** (MCSHS) [9,30] in our evaluation, where "components" are method call sequences of length z. For example, for the method hit sequence $\langle f(), g(), h() \rangle$ and $z = 2$ the following two "components" would be marked as executed $\{(f(), g()),\ (g(), g())\}$. With $z = 1$ the extension is disabled and only called functions without their predecessors will be used.

3.3 Transaction Detector

We have assessed the well-known timing-based transaction detector, which splits the input stream every second into separate transactions as proposed by Abreu et. al [1]. This is necessary if a system cannot easily provide separate runs, for example because it runs continuously. Since results with these transactions were already good, see Section 4, we have not examined other approaches.

3.4 Failure-Detection Oracle

Our idea for a failure-detecting oracle is based on the fact that one of the main purposes of an embedded system is to interact with its environment. We use this fact to learn how correct interaction with periphery looks like and use this model to decide, if an embedded system still behaves as expected or failed. We call this model **driver interaction model (DIM)**, which is a finite state machine where all states are accepting and the input symbols are communication messages to the hardware. A DIM for a fictional wireless chipset driver can be seen in Fig. 2.

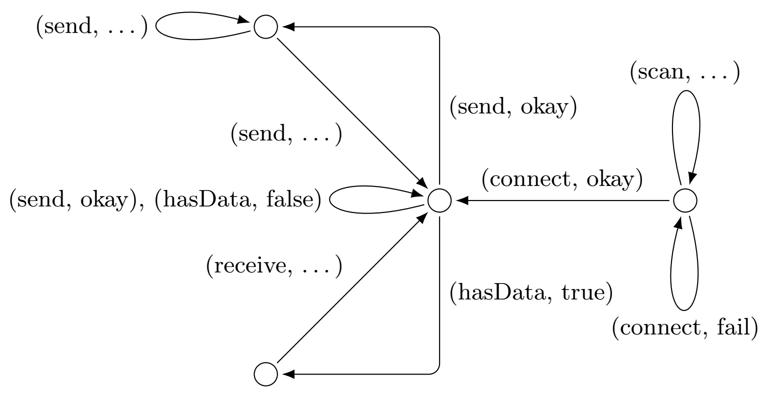

Fig. 2. A fictional DIM for a wireless chipset driver

We propose that either a correct implementation or the successful executions during development, e.g., supported by unit tests, can be used to learn a DIM, which can then be deployed with the monitoring system. In the unfamiliar environment our model will detect potential failures, which can be used by the developer to either fix a fault or to improve the DIM iteratively. Our approach is therefore applicable in multiple scenarios, for example helping a developer implement new software, but also checking a re-implementation or software variant.

For that, once a driver interaction model was learned, it can be saved. It can then be loaded later again and augmented with new information, but it is also possible to manually modify or check the model according to a specification.

Learning driver interaction models is a CPU- and memory-intensive task and requires multiple runs as reference data, which is another reason why we propose to learn the model beforehand and deploy it on the monitoring machine. We used a variance of the k-tails algorithm [6] to merge equivalent states during learning. Regularly, two states are equivalent if their k-tails are equal, i.e., the same words of length k are accepted beginning from the two states. We relaxed this and consider states for merging if the source state's k-tail is contained in the target state's k-tail to obtain more realistic models. Since the learning data is based on sequences of method calls, each node (besides an initial node) has exactly one k-tail and merging nodes will not lead to more k-tails, otherwise the nodes would not have been merged. This observation allows to merge states during learning using a map from k-tails to already inferred nodes. Learning a new node is done by first checking if the k-tail is present in the map. If it is, the two nodes are merged; if not, a new node is created and the map updated with this newly created node.

This algorithm is used to merge similar traces. Consider the most basic example, where identical message sequences occur in two traces, e.g., *(connect, okay)* $\rightarrow$ *(send, okay)* $\rightarrow$ *(send, . . .)*. It is likely that in both traces the periphery's internal state was identical, therefore the origin states are merged.

Such a driver interaction model can be used as oracle by following transitions and in case this is not possible, a failure is detected. If the execution should continue after a failure was detected, we just try to find fitting transitions for the next events starting in the erroneous state. A reset threshold r is used and after r successful consecutive transitions, the oracle is used to detect failures again. If not specified otherwise, we used the learning parameter k also as reset threshold, as this is the first intuitive choice.

Interaction can be observed on different abstraction levels, e.g., binary communication or already decoded packages, and the data can be obtained either directly in the driver or via an external sniffer.

We also examined **software behavior graphs (SBG)** [20], a graph where nodes correspond to functions and an edge (s, t) exists, if function s calls function t. They can be learned from trace data and during fault localization encountering a missing edge can be interpreted as a failure. Since the evaluation in Section 4 shows that they do not perform well in our setting, we will not go into details.

3.5 Thread Separation

Since modern software is often executed in parallel and even embedded systems make use of multi-core CPUs nowadays, it is sometimes necessary or at least helpful to know which CPU or thread caused a failure. We investigated a mechanism to isolate parallel executions. Since we did not want to expect that an embedded operating system provides a thread abstraction, we configured our approach to use the CPU core id as identifier, although it is possible to use a thread id in systems where this abstraction exists.

3.6 Failure Indexing

Spectrum-based fault localization (SBFL) is guided by the idea to localize one fault, whereas in practice often multiple faults will be present simultaneously. If enough computational power is present, this is not a problem, as one fault can be fixed, and then another SBFL experiment can be run. In long-running, isolated systems, this is not that easy, therefore it is preferable to collect information regarding multiple faults at the same time, if possible. One problem is, how the observed effects can be pinpointed to one of multiple faults. One idea is to use the information which oracle detected the failure caused by a fault or how it detected the failure to distinguish different causes. This is called **failure indexing**. We will assess in the next section if this feature helps in our use case.

4 Evaluation and Application

We have implemented our approach and performed multiple experiments to determine the best choice and configuration regarding oracles/detectors and parameters presented in the last section.

4.1 Testbed

We have implemented our approach exemplarily based on the embedded operating system CyPhOS [7], running the application *EMSComponent* [25] simulating an engine control unit on a Wandboard Quad – a development board hosting an i.MX6 CPU providing the ARM Cortex-A9 instruction set and 2 GB of RAM. We used the same operating system and board to deploy the monitoring system. To simulate a combustion engine, EMSBench[4] [17] was used, which was deployed on an STM32F4-Discovery development board – a low-power development board featuring an ARM Cortex-M3 CPU.

To trace the application we used AspectC++, which can be easily integrated into the CyPhOS build process. It was used to transmit information about called methods and driver interaction events to the monitoring machine. Since AspectC++ automatically numbers relevant methods sequentially (join point ID, short: JPID), we used these numbers to identify functions. Although this is not necessary for our approach, it is convenient to transmit information efficiently. We also used AspectC++ to modify the low-level GPIO driver to obtain trace data for the driver-interaction-model oracle.

Since most suspiciousness metrics are easy to compute, we included five metrics in total: Barinel [3], D^* [28], Ochiai [23], Op2 [21] and Tarantula [15], where D^* is parametrized with $* \in \mathbb{N}$.

To compute these metrics we only need the aggregated matrices as described in Section 1. Further, we reduced memory occupation by only storing the lower row of aggregated matrices, the one which counts the succeeding/failing transactions the component participated in, and two counters for succeeding and failing over all transactions. The upper row can be recomputed from these. Actually, storing the spectra and aggregated matrix requires some more thought to not exhaust the available memory. We have used a variation of the trie data structure [12] to be able to use method call sequences as keys for storing.

Our approach uses a domain-agnostic oracle and transaction detector. As a baseline for our experiments, we implemented domain-specific counterparts. The domain-specific oracle detects an error in CyPhOS by monitoring the crash handler. Since CyPhOS is a component-based operating system, it was possible to use calls of the function to switch between two components as an identifier for a new transaction. We will later refer to this transaction detector as “OSC”.

We used our own tool [13] to inject software faults according to a well-known fault model [10, 22], without the addition for more realistic faults based on software metrics. We filtered faults which were not useful, e.g., resulted in images that did not boot or modifications in dead code, resulting in 41 patches inserting faults into the application, see Fig. 2 for an explanation of the corresponding abbreviations.

[4] `https://www.informatik.uni-augsburg.de/en/chairs/sik/research/running/emsbench/`

4.2 Experiments

We performed multiple experiments to determine the best oracle and the best configuration for our use case. For that, we have performed the following steps:

- Generate patch files to inject software faults into our application.
- Learn a driver interaction model and a software behavior graph from successful runs to determine the quality of the oracles regarding detection of failures triggered by the previously generated faults, and assess the impact of parameter k of our k-tails learning algorithm.
- Determine speed and therefore feasible configurations for continuous monitoring.
- Determine whether thread separation and which transaction detection mechanism performs best.
- Assess found configurations as a whole.

Table 1. Oracle-detected errors for the injected faults per driver interaction model (DIM) with different k, software behavior graph (SBG) and domain-specific CPU exception detector

Fault Type	Oracle	Fault Type	Oracle	Fault Type	Oracle
MFC-1	2-DIM	MIES-2		MLPA-2	
MFC-2	1, 2-DIM	MIES-4		MLPA-10	1, 2-DIM
MFC-4		MIES-6	2-DIM	MLPA-26	1, 2-DIM
MFC-5		MIES-7	2-DIM	MLPA-29	2-DIM
MFC-6	1, 2-DIM	MIES-11	2-DIM	MLPA-30	2-DIM
MIA-2		MIFS-1		MRS-1	
MIA-3		MIFS-3		MRS-9	2-DIM
MIA-5		MIFS-5		MRS-10	CPU ex. & SBG
MIA-6		MIFS-7	2-DIM	MRS-15	
MIA-7		MIFS-8	2-DIM	MRS-18	2-DIM
MIEB-2	1, 2-DIM	MLOC-3	1, 2-DIM & SBG	MVIV-7	
MIEB-3	1, 2-DIM	MLOC-4	1, 2-DIM	MVIV-15	
MIEB-4	2-DIM	MLOC-5			
MIEB-7	2-DIM	MLOC-8			
MIEB-10	2-DIM				

To learn driver interaction models, we used 15 runs of the EMSComponent for k-tails with $k = 1, \ldots, 10$, but already for $k = 3$ the resulting oracle returns false-positives. Therefore, we only used $k = 1$ and $k = 2$, where for $k = 2$ the oracle detected 53.7 % of the errors, whereas for $k = 1$ only 19.5 % were detected. Table 1 shows detected errors caused by injected faults for the driver interaction model (DIM) with $k = 1$ or 2 in comparison to the domain-specific CPU exception detector, which detected only one error, and the software behavior graph (SBG), which detected two errors. Since only our driver interaction model with

Table 2. Fault types used for software fault injection in our experiments

MFC	Missing function call
MIA	Missing if construct around statements
MIEB	Missing if construct plus statements plus else before statements
MIES	Missing if construct plus statements plus else plus statements
MIFS	Missing if construct plus statements
MLOC	Missing OR clause in branch condition
MLPA	Missing small and localized part of the algorithm
MRS	Missing return statement
MVIV	Missing variable initialization using a value

learning parameter $k = 2$ was able to identify a notable number of errors at all, we used this oracle for all further experiments.

Next, we evaluated the parameter specifying the length of method call sequences with and without failure indexing and thread separation, so that trace information can be processed on another device without loosing information. The maximum baud rate of the used UART is 460,800, which leads to a maximum of 57,600 B/s or, with our encoding, 3,840 event messages per second. Table 3 shows the resulting numbers, where the processing numbers high enough to cope with the maximum baud rate are highlighted bold. As can be seen, only for a method call sequence length of $z = 1$ (see Section 3.2) the speed is always above the necessary computation border. Since it may be possible to improve the implementation and therefore increase the processing speed or use buffers on the monitoring device to puffer trace bursts, we did not drop sequence length of 2 altogether, but will assess it in later experiments. Although, it remains an open question if the computations can be actually improved to fulfill the real-time requirements.

Table 3. Processing speeds for different campaign configurations

Sequence Length	**Failure Indexing**	**Thread Separation**	**Avg. B/s**	**Min. B/s**
2	yes	yes	11 196	5 441
2	yes	no	7 445	3 344
2	no	yes	20 788	10 032
2	no	no	12 813	5 777
1	yes	yes	**81 335**	48 819
1	yes	no	**68 147**	39 683
1	no	yes	**146 787**	**95 129**
1	no	no	**127 094**	**76 523**

Our overall goal is to find the best combination and configuration of techniques and parameters for our fault-localization approach. We will use the well-known EXAM metric [29] to assess the performance. The EXAM metric specifies in percent what proportion of the reported code positions have to be examined before actually encountering the faulty position. Therefore, a value of 0 % is best.

We first compared the six suspiciousness metrics mentioned in the introduction, see Section 1, each with and without thread separation and different transaction detection approaches in Table 4. We show the average EXAM score over all faults and its standard deviation. Since multiple experiments resulted in a perfect EXAM score of 0 %, we also show the number of these experiments.

One can draw at least two conclusions from these experiments. First, activating thread separation leads to worse EXAM scores in nearly all cases. Although our workload was mostly single-threaded it is still a surprise that this feature is mostly hindering. Our best guess at the moment is that activating this feature led to a high number of successful transactions on other cores which had a negative influence on the metrics. This would explain the good performance of Barinel and Tarantula metric, which become more accurate when the number of successful runs vastly exceed the number of runs of the faulty component. Since our workload was not suitable to examine this problem further, we leave this for future work. Second, the timer-based transaction detector outperformed the domain-agnostic component-based detector ("OSC"), although the influence is not as severe as that of thread separation.

Table 4. Accuracy results in EXAM metric for different analysis configurations

		#0 %	**Avg.**	**σ**	**#0 %**	**Avg.**	**σ**	**#0 %**	**Avg.**	**σ**
Thread	**Trans.**		Barinel			D^2			D^3	
Sep.	**Detect.**		Ochiai			Op2			Tarantula	
off	OSC	11	5.1 %	8.1 pp	13	4.7 %	8.2 pp	13	4.7 %	8.2 pp
		13	4.7 %	8.2 pp	13	4.7 %	8.2 pp	11	5.1 %	8.1 pp
off	timer	13	2.9 %	4.1 pp	**15**	**2.7 %**	**4.7 pp**	**15**	**2.7 %**	**4.7 pp**
		14	3.0 %	4.7 pp	14	3.0 %	4.7 pp	12	3.6 %	4.7 pp
on	OSC	1	43.2 %	21.2 pp	1	44.5 %	18.4 pp	0	43.1 %	15.3 pp
		1	44.5 %	18.4 pp	0	43.1 %	14.6 pp	1	43.2 %	21.2 pp
on	timer	13	5.6 %	10.5 pp	0	27.4 %	8.8 pp	0	27.8 %	8.8 pp
		0	27.8 %	8.7 pp	0	28.3 %	8.7 pp	12	6.0 %	10.5 pp

Since the results of Table 4 are quite clear, we configured our method without thread separation and with the timer-based transaction detector to assess some remaining questions in a last experiment. First, we analyzed whether using parameters as part of the call sequence improves or worsens the results. Second, we analyzed whether reducing the reset threshold has notable impact. Lastly, we analyzed whether increasing the sequence length, and therefore actually using the MCSHS extension, improves fault localization. We performed the last experiment despite having assessed already that our current implementation is not able to cope with the trace data on time, but it may be possible to improve the runtime of our method.

The results of our last experiments are shown in Table 5. As can be seen, none of the experiments leads to significantly better results. If we ignore function parameters and for example compare D^2 and D^3 with Table 4 it can be seen that only the standard deviation improves, whereas the number of zero percent occurrences is reduced, i.e., some faulty components are not inspected first anymore. Reducing the reset threshold to 1 actually worsens the results a bit. Regarding the MCSHS extension, if we use the averaging proposed in the corresponding paper [9], where each function is assigned a suspiciousness based on the suspiciousness values of all sequences it appeared in, the results are significantly worse than before, see the last row. If we instead use an approach where each method occurring in the sequence is examined and therefore the suspiciousness of the first occurrence of the faulty method in any sequence is significant, the results improve a little bit, see the results highlighted bold, but we might lose trace data due to missing real-time requirements, cf. Fig. 3. Additionally, depending on the selected examination strategy, the number of methods to check is doubled with $k = 2$. We can conclude that for our repetitive application the reset threshold is not relevant and that using the MCSHS extension, if it could cope with the trace data in real-time, would not improve the results significantly.

Table 5. Accuracy results in EXAM metric for variants of the "ideal" configuration

	#0 %	**Avg**	**σ**	**#0 %**	**Avg**	**σ**	**#0 %**	**Avg**	**σ**
		Barinel			D^2			D^3	
Mode		Ochiai			Op2			Tarantula	
Ignore	12	2.9 %	3.8 pp	13	2.7 %	3.9 pp	13	2.7 %	3.9 pp
Parameters	12	3.0 %	4.0 pp	12	3.0 %	4.0 pp	10	3.6 %	4.0 pp
Reset Threshold	12	3.2 %	4.2 pp	14	3.0 %	4.7 pp	14	3.0 %	4.7 pp
	13	3.3 %	4.7 pp	14	3.0 %	4.7 pp	11	3.9 %	4.7 pp
Seq. Len. $z = 2$	12	3.8 %	5.5 pp	14	2.7 %	4.9 pp	14	2.7 %	4.9 pp
Best EXAM-Score	13	2.9 %	4.9 pp	**14**	**2.4 %**	**4.6 pp**	11	4.1 %	5.5 pp
Seq. Len. $z = 2$	9	18.8 %	24.1 pp	10	17.1 %	22.2 pp	10	19.2 %	23.3 pp
Avg EXAM-Score	9	17.9 %	23.7 pp	10	21.1 %	28.0 pp	8	19.3 %	24.3 pp

5 Threats to Validity

While our results are promising, we used a specific setting hindering the extension of these results to other experiments. Foremost, the repetitive work of the engine control unit is very helpful when using statistical methods. In a setting where the software or device has a wider range of functionality, our results may not hold. However, many embedded systems implement control loops and are therefore similar to our experiment. Second, the used fault model has a great impact on the results. While we used a fault model widely accepted in the fault-injection community it still may not accurately represent real faults of specific scenarios.

Another problem can be that our instrumentation to transfer trace data to another system might have an impact on the timing behavior of the system-under-test. This can lead to modified behavior or cause it to violate real-time constraints.

6 Conclusion

In this paper we have assessed spectrum-based fault localization techniques especially in the setting of deployed embedded systems, where continuous access to the systems and expensive debugging hardware is not an option. We have compared multiple known suspiciousness metrics using the EXAM score by applying them in an engine-control-unit scenario and found that regularly less than 10 % of probable fault locations have to be analyzed to find the fault. Further, we showed how oracles other than unit tests can be used to decide if an execution was successful or failing, and for that case demonstrated a new form of passively learned automaton, the driver interaction model, which can be used to learn correct behavior when interacting with periphery. We have compared this oracle with others and were able to show that our approach works well in our setting.

While our driver interaction models already performed good, during qualitative examination we found that it may still be possible to improve their representativeness. In future work it may be promising to try to improve their accuracy and expressiveness by using probabilistic or extended probabilistic automata [11] instead.

In general it seems that a spectrum-based approach based on behavior-comparing oracles seems useful above-average in a setting where the application behavior is repetitive, where only limited memory is available to store (aggregated) trace data and where observable interaction with external units takes place. In this case even simple approaches already provide good results, i.e., it is often enough to analyze the single most suspiciousness component to already find the underlying fault.

Acknowledgement. We thank Erwin Schoitsch, Austrian Institute of Technology (AIT), for his valuable feedback and dedicated effort to improve this paper.

References

1. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: Program spectra analysis in embedded software: A case study. Tech. Rep. TUD-SERG-2006-007, Software Engineering Research Group, Delft University of Technology (2006), `http://arxiv.org/abs/cs/0607116`
2. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: An observation-based model for fault localization. In: Proceedings of the 2008 International Workshop on Dynamic Analysis: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008). pp. 64–70. WODA '08, ACM, New York, NY, USA (2008). https://doi.org/10.1145/1401827.1401841

3. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: Spectrum-based multiple fault localization. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering. pp. 88–99. ASE '09, IEEE Computer Society, Washington, DC, USA (2009). https://doi.org/10.1109/ASE.2009.25
4. Abreu, R., Zoeteweij, P., Gemund, A.J.V.: An evaluation of similarity coefficients for software fault localization. In: 2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06). pp. 39–46 (Dec 2006). https://doi.org/10.1109/PRDC.2006.18
5. Abreu, R., Zoeteweij, P., Golsteijn, R., van Gemund, A.J.: A practical evaluation of spectrum-based fault localization. Journal of Systems and Software **82**(11), 1780–1792 (2009). https://doi.org/10.1016/j.jss.2009.06.035, sI: TAIC PART 2007 and MUTATION 2007
6. Biermann, A.W., Feldman, J.A.: On the synthesis of finite-state machines from samples of their behavior. IEEE Transactions on Computers **C-21**(6), 592–597 (Jun 1972). https://doi.org/10.1109/TC.1972.5009015
7. Borghorst, H., Spinczyk, O.: CyPhOS – A component-based cache-aware multi-core operating system. In: Proceedings of the 32th International Conference on Architecture of Computing Systems (ARCS '19) (2019), to appear
8. Casanova, P., Schmerl, B., Garlan, D., Abreu, R.: Architecture-based run-time fault diagnosis. In: Crnkovic, I., Gruhn, V., Book, M. (eds.) Software Architecture. pp. 261–277. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
9. Dallmeier, V., Lindig, C., Zeller, A.: Lightweight Defect Localization for Java, pp. 528–550. Springer, Berlin, Heidelberg (2005). https://doi.org/10.1007/11531142_23
10. Durães, J.A., Madeira, H.S.: Emulation of software faults: A field data study and a practical approach. IEEE Transactions on Software Engineering **32**(11), 849–867 (Nov 2006). https://doi.org/10.1109/TSE.2006.113
11. Emam, S.S., Miller, J.: Inferring extended probabilistic finite-state automaton models from software executions. ACM Trans. Softw. Eng. Methodol. **27**(1), 4:1–4:39 (Jun 2018). https://doi.org/10.1145/3196883
12. Fredkin, E.: Trie memory. Commun. ACM **3**(9), 490–499 (Sep 1960). https://doi.org/10.1145/367390.367400
13. Gabor, U.T., Siegert, D., Spinczyk, O.: Software-fault injection in source code with Clang. In: Proceedings of the 32th International Conference on Architecture of Computing Systems (ARCS '19), Workshop Proceedings (2019), to appear
14. Hooman, J., Hendriks, T.: Model-based run-time error detection. In: Giese, H. (ed.) Models in Software Engineering. pp. 225–236. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69073-3_24
15. Jones, J.A., Harrold, M.J.: Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. pp. 273–282. ASE '05, ACM, New York, NY, USA (2005). https://doi.org/10.1145/1101908.1101949
16. Ju, X., Jiang, S., Chen, X., Wang, X., Zhang, Y., Cao, H.: HSFal: Effective fault localization using hybrid spectrum of full slices and execution slices. Journal of Systems and Software **90**, 3–17 (2014). https://doi.org/10.1016/j.jss.2013.11.1109
17. Kluge, F., Ungerer, T.: EMSBench: Benchmark und Testumgebung für reaktive Systeme. In: Halang, W.A., Spinczyk, O. (eds.) Betriebssysteme und Echtzeit. pp. 11–20. Informatik aktuell, Springer (2015). https://doi.org/10.1007/978-3-662-48611-5_2

18. Le, T.D.B., Thung, F., Lo, D.: Theory and practice, do they match? A case with spectrum-based fault localization. In: 2013 IEEE International Conference on Software Maintenance. pp. 380–383 (Sep 2013). https://doi.org/10.1109/ICSM.2013.52
19. Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y., Zhai, C.: Have things changed now?: An empirical study of bug characteristics in modern open source software. In: Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability. pp. 25–33. ASID '06, ACM, New York, NY, USA (2006). https://doi.org/10.1145/1181309.1181314
20. Liu, C., Yan, X., Yu, H., Han, J., Yu, P.S.: Mining behavior graphs for "backtrace" of noncrashing bugs. In: Proceedings of the 2005 SIAM International Conference on Data Mining. pp. 286–297. Society for Industrial and Applied Mathematics (2005). https://doi.org/10.1137/1.9781611972757.26
21. Naish, L., Lee, H.J., Ramamohanarao, K.: A model for spectra-based software diagnosis. ACM Trans. Softw. Eng. Methodol. **20**(3), 11:1–11:32 (Aug 2011). https://doi.org/10.1145/2000791.2000795
22. Natella, R., Cotroneo, D., Duraes, J.A., Madeira, H.S.: On fault representativeness of software fault injection. IEEE Transactions on Software Engineering **39**(1), 80–96 (Jan 2013). https://doi.org/10.1109/TSE.2011.124
23. Ochiai, A.: Zoogeographical studies on the soleoid fishes found in japan and its neighbouring regions—II. Bulletin of the Japanese Society for the Science of Fish **22**(9), 526–530. https://doi.org/10.2331/suisan.22.526
24. Reps, T., Ball, T., Das, M., Larus, J.: The use of program profiling for software maintenance with applications to the year 2000 problem. In: Jazayeri, M., Schauer, H. (eds.) Software Engineering — ESEC/FSE'97. pp. 432–449. Springer Berlin Heidelberg, Berlin, Heidelberg (1997). https://doi.org/10.1007/3-540-63531-9_29
25. Schulte-Althoff, T.: Validierung des Echtzeitverhaltens des ereignisbasierten Betriebssystems CyPhOS am Beispiel einer Motorsteuerung (2017), https://ess.cs.tu-dortmund.de/Teaching/Theses/
26. Spinczyk, O., Lohmann, D.: The design and implementation of AspectC++. Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software **20**(7), 636–651 (Oct 2007). https://doi.org/10.1016/j.knosys.2007.05.004
27. Stephenson, A.G., LaPiana, L.S., Mulville, D.R., Rutledge, P.J., Bauer, F.H., Folta, D., Norvig, P.: Mars climate orbiter mishap investigation board phase I report (Nov 1999), https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf
28. Wong, W.E., Debroy, V., Gao, R., Li, Y.: The DStar method for effective software fault localization. IEEE Transactions on Reliability **63**(1), 290–308 (Mar 2014). https://doi.org/10.1109/TR.2013.2285319
29. Wong, W.E., Debroy, V., Xu, D.: Towards better fault localization: A crosstab-based statistical approach. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) **42**(3), 378–396 (May 2012). https://doi.org/10.1109/TSMCC.2011.2118751
30. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. IEEE Transactions on Software Engineering **PP**(99), 1–1 (2016). https://doi.org/10.1109/TSE.2016.2521368
31. Xie, X., Chen, T.Y., Kuo, F.C., Xu, B.: A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Trans. Softw. Eng. Methodol. **22**(4), 31:1–31:40 (Oct 2013). https://doi.org/10.1145/2522920.2522924
32. Xie, X., Wong, W.E., Chen, T.Y., Xu, B.: Spectrum-based fault localization: Testing oracles are no longer mandatory. In: 2011 11th International Conference on Quality Software. pp. 1–10 (Jul 2011). https://doi.org/10.1109/QSIC.2011.20

33. Xuan, J., Monperrus, M.: Learning to combine multiple ranking metrics for fault localization. In: 2014 IEEE International Conference on Software Maintenance and Evolution. pp. 191–200 (Sep 2014). https://doi.org/10.1109/ICSME.2014.41