On the Impact of Injection Triggers for OS Robustness Evaluation[*]

Andréas Johansson, Neeraj Suri
Dept. of CS, TU-Darmstadt, Germany
{aja,suri}@informatik.tu-darmstadt.de

Brendan Murphy
Microsoft Research, Cambridge, UK
bmurphy@microsoft.com

Abstract

The traditional method of software robustness evaluation, through error injection, is for errors to be injected at reaching a specific code location. This paper studies what impact varying the time for error injection has on evaluation of software (specifically Operating Systems (OS)) robustness. A strategy to guide the appropriate error injection timing is proposed, based on the observation that the operational usage profile of a driver shows a high degree of regularity in the calls being made. Using the concept of call blocks (i.e., a distinct sequence of calls made to the driver), the trigger for injection can be used to guide injections into different system states, corresponding to the operations carried out. A real-world case study compares the effectiveness of the proposed strategy to a traditional location-based approach, and shows that significantly more useful insights can be gained using the proposed approach.

1. Introduction

Fault injection (FI) has been identified as a useful and effective technique for robustness evaluation [1, 5, 8, 12]. The purpose of robustness evaluation is to establish the failure modes of the system, identify error propagation paths and to expose possible vulnerabilities related to external perturbations. A key parameter for FI is the time of injection. The effectiveness of the injection (its ability to reveal a vulnerability) depends on the state of the system at the time of injection. A failure of a system is more likely if a fault (or error) is inserted while the system is processing a critical path (holding shared resources for instance).

With device drivers considered as a prominent cause of OS failures [3, 15, 19], the focus of this paper is on OS robustness evaluation w.r.t. faults in the interface between device drivers and the OS. Several proposals have been made on how to evaluate the robustness of a system in the presence of faults in drivers, for instance [8, 9, 10]. A promising technique for such evaluation is fault injection. For FI three key questions need to be answered: a) which faults (or errors) to inject, b) where to inject them, and c) when to inject them. Several previous efforts have focused especially on the first two questions. Although the need for controlling the time of injection has been identified as important, it is to a large degree a manual process entailing selection of triggering events. The selection of appropriate triggers is difficult and system dependent. Traditionally, mainly two generic strategies have been employed to select the events triggering an injection: *first-occurrence* (also location-based) and *time-triggered*. First-occurrence is based on the premise that since we cannot generally know when the system is in a vulnerable state, we identify (code) locations for injections and inject the error as soon as the system reaches these locations. This is particularly useful for software (SW) faults, which by nature are tied to a code location. The error duration can then vary from transient to permanent. The time-triggered approach defines a timeout, after which an error is injected at a specific location, or possibly a randomly selected one. First occurrence can then be applied after the timeout. The key issue here is then to set timeouts appropriately, alternatively do sufficiently many injections with a random distribution in time. This approach is particularly useful when simulating physical faults such as radiation or EMI.

This paper proposes a novel technique for controlling the timing of error injection in the interface between drivers and the OS using the usage profile of the driver. Calls invoked on the driver are divided into *call blocks*, repeating subsequences of calls. Additionally, the lifetime of the driver is divided into phases, such as initialization, working and clean up phases. Our proposed injection methodology uses this timing model to select the call blocks and phases where errors are injected, thereby enabling more fine-grained control over the time of injection, compared to first occurrence and time-triggered injection.

The contribution of this paper is two-fold, a) it proposes a novel timing model for driver-based fault injection, based on the operational usage profile of the system, and b) it substantiates the value of controlling the timing of error injec-

[*]This research has been supported, in part, by Microsoft Research, EU FP6 NoE ReSIST, EU FP6 IP DECOS and DFG TUD GK MM.

tion via extensive experimentation on a real system.

The structure of the paper is as follows: Sections 2 and 3 introduce the system and error models used. Section 4 then defines the key concepts of the usage profiling approach: call blocks and call strings. The experimental setup is detailed in Section 5 and the used evaluation criteria is presented in Section 6. The results of the case study are presented in Section 7 and further discussed and interpreted in Section and 8. Section 9 presents relevant related work and the paper is summarized in Section 10.

2. System Model

The system model consists of a set of interacting SW components. Interactions across components take place using specified interfaces. An interface is a set of interaction points, or *services*. Interaction takes place by one component invoking the service of another component. The system as a whole also provides services to a user (which in turn can be another system). A failure of the system is observed when the system service can no longer be provided as stipulated. One can define a set of disjoint failure modes for the system services.

In this context, robustness evaluation aims to investigate how the system handles faults occurring in external component interactions. Typically, an external component or system is identified as the source of faults, and errors are introduced in the interface between this component and the target system. As the invocation volume for such interactions can be very high, a selection of which interactions to target must be performed. As mentioned, a common approach is to first identify services to target and then inject using first occurrence.

For the case study presented in this paper the targeted component is a device driver and the system where the outputs are observed is the operating system together with the workload (applications) running on it. Perturbations are introduced in the interfaces between the device drivers and the OS. Drivers provide *services* for the OS to use. Applications call the OS to perform specific *operations* using the devices attached to the system, which translates into one or more calls to driver services.

For each targeted device driver we identify the services defining the exported functionality provided by the driver (d_x in Figure 1). The OS uses these services to interact with the driver. The driver completes these service requests using services provided by other components, mainly the OS itself, but also other libraries shipped with the OS and other drivers (s_x in Figure 1).

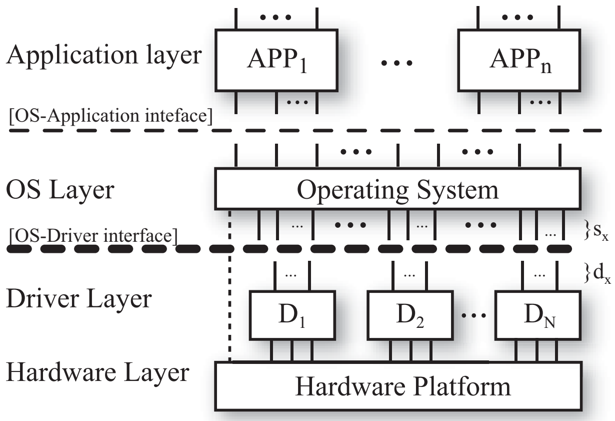

Figure 1. The system model.

3. Error Model

The errors we inject represent the subset of software (and hardware) errors that manifest themselves at the OS-Driver interface. A service in this interface is typically invoked multiple times during the lifetime of the system. Errors at the interface level represent multiple potential faults (and fault locations) in the driver. Consecutive invocations of a service by a driver may stem from different locations in the code and thus occur when the driver is in different states. It is therefore very reasonable that multiple invocations are targeted, i.e., that the time of injection is controlled and used to guide the injections. Note that this does not in general hold for code level injections, where the location of injection corresponds to the location of the fault.

For this study, bit-flips are used to represent the faults in drivers. Bit-flips were chosen based on their effectiveness in provoking system vulnerabilities, as reported for instance in [10]. Bit-flips have also been successfully used for instance in [5] and [21]. It is important to note that the goals of a robustness evaluation is to provide clear failure modes of the system, to find and expose robustness vulnerabilities and to guide further QA resources across system components. For these purposes bit-flips are well suited.

Previous studies have shown that many driver errors manifest as memory access errors, which are possible to detect using sandboxing techniques [19]. However, still some errors can slip through over the interfaces to other components and these are the target of this study.

As previously mentioned the target location for the errors is the OS-Driver interface. This location was chosen as it is a well defined interface, which is suitable for robustness evaluation as it defines an input interface for the OS. It also allows injection of errors without access to the source code of the driver.

Errors are injected in the parameters used in service invocations, by manipulation of the values used. A bit is flipped in the value used and the call is passed on to the OS. We target each parameter carrying a value from the driver to the OS. For each parameter we target all bits once, giving a total of 32 injections per parameter for types using all 32 bits, such as integers. However, our FI environment previously detailed in [9, 10] additionally allows injecting errors also in the members of C-structs, commonly used in the OS-Driver interface, which is based on C.

We target a transient error model, i.e., an error appears for one invocation of a service and then disappears, leaving subsequent invocations error-free. Since we assume that base line functional testing has been performed for each component our error model represents hard-to-find Heisenbugs as well as external perturbations.

4. Driver Usage Profile

A service request from an application translates to one or more calls into the driver by the OS. In practice only some sequences of calls make sense from a semantic point of view. It doesn't, for instance, make sense for an application to try to read a file from disk before it has been opened, although there is nothing stopping it from trying. Since such problems can be expected to have been found during testing, the operational behavior of the driver can be broken down into a series of calls to the driver, where certain subsequences are more frequent than others. Each such subsequence represents common sequences of calls made from the application, thereby defining what the driver is "doing", such as "reading", or "setting connection parameters" etc. We call each of these discrete blocks of calls a *call block*.

The operational usage profile of a driver is a list of service invocations carried out on the driver (the d_x services in Figure 1). Note that this definition differs from the traditional definition of operational profile by Musa [16]. For completing the request sent by the OS the driver makes use of OS services and other help libraries (s_x in Figure 1). These calls to other services form the interface between the driver and the OS, which is the target for our robustness evaluation. As the same OS service may be invoked multiple times, i.e., called from different locations within the driver and with different internal states, it is pertinent to target as many of them as possible. By guiding the injection using the usage profile we get a much finer granularity and control of injection timing.

4.1. Call Strings and Call Blocks

The usage profile of a driver can be illustrated as a time-service diagram, like in Figure 2. The services invoked ($a - d$) are shown as rectangles. By assigning a token to each service from a predefined alphabet, the series of calls made to the driver can be represented as a string, the *call string*. The call string in Figure 2 is *ababcdabdab*. Note that we do assume sequential operation of the driver. Further discussion on this is presented in Section 8.

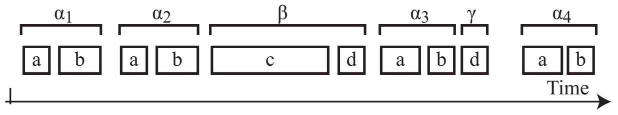

Figure 2. Example of invoked services.

In the example in Figure 2 it can be seen that services a and b are invoked multiple times during the execution of the component. Each time a is called it is followed by an invocation of b; a and b thus form a *call block*, which is repeated during the execution of the component. The sequences cd and d are not repeating and cannot be added to any other call block. These sequences form the non-repeating call blocks. The whole call string can therefore be split into call blocks as indicated in Figure 2 as the sequence **(ab)**$\{2\}$**cd(ab)(d)(ab)**[1]. The assignment of call blocks is typically done through a combination of identification of repeating blocks and a priori knowledge regarding the functionality of the driver.

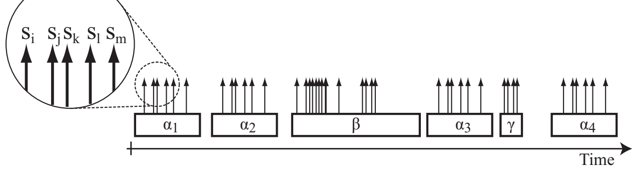

Figure 3. Driver invoked services.

Figure 3 shows the same call blocks as Figure 2, α, β and γ. When injecting in service s_i, *first occurrence would only inject an error in the first invocation of* s_i in α. Subsequent invocations of s_i, in α_1 or elsewhere are not targeted, unless the error is made permanent. Using call blocks allows us to inject errors in multiple invocations of a service.

4.2. Operational Phases

Generally the lifetime of a driver can be divided into three disjoint phases, as seen in Figure 4. First the driver executes an initialization phase, where the driver registers its presence with the OS and initializes software and device specific resources. Then follows a number of blocks where the driver performs work on behalf of applications (indirectly) or the OS directly, the working phase. At the end is a clean up phase where the driver deregisters with the OS and may free any resources allocated to it.

[1]The syntax $(ab)\{2\}$ means that the symbols ab are repeated twice.

The concept of operational phases becomes important when selecting which call blocks to inject errors into. It is to be expected that failures in the initialization and clean up phases have more severe consequences, since in these phases the driver interacts with many OS services, which may affect the state of the system, not only the driver.

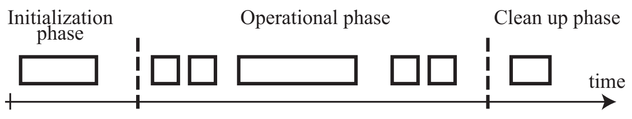

Figure 4. The operational phases of a driver.

4.3. Injection Targets

Injections are targeted at each call block. When a call block is repeated multiple times, a filtering may take place to reduce the number of injections needed. Preferably at least one call block per phase is targeted, when possible. For each targeted call block, first occurrence injection is done on *each service invoked at least once within the call block.*

5. Experimental Setup

To evaluate the influence of the time of injection on the outcome of the robustness evaluation this section presents a case study where two drivers have been evaluated with injections targeting all unique call blocks. The two drivers, a serial port driver (`cerfio_serial.dll`) and a network card driver (`91C111.dll`), are built for Windows CE .Net, running on an Intel XScale reference board. The experimental setup is illustrated in Figure 6. The reference boards are connected to a host computer handling logging of the results. Both drivers are provided by the reference board vendor and are shipped with the hardware in binary form, prohibiting source code modifications.

5.1. Targeted Drivers

The two drivers studied were selected as they represent two common types of functionality found in most modern OS's. They are also different, with the serial port driver relying heavily on the OS to perform its operations, whereas the network card driver uses the OS mostly during the initialization phase. This is clearly shown in Figure 5 which indicates the serial driver invoking a higher number of services and more frequently than the network card driver. It is therefore to be expected that the utility of the proposed call block approach will show up most clearly for the serial port driver.

For the serial port driver 41 services are used by the driver. A service is invoked 30.5 times on average for the

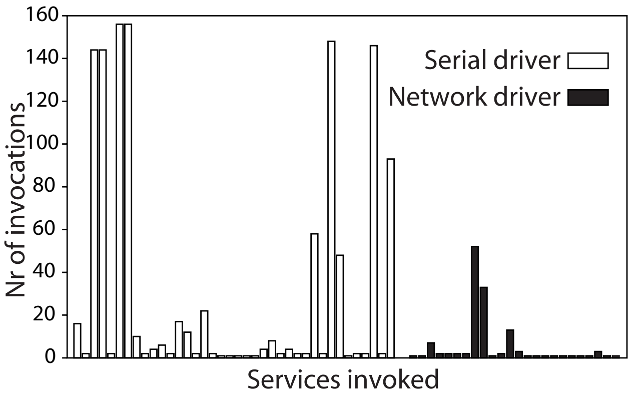

Figure 5. Driver invocation profiles.

given workload with a standard deviation of 53.5 and median of 2. This reflects the fact that there are some services that are used frequently (for reading/writing, synchronization etc.) and some only once or twice (like configuration of the device). The same data for the network card driver shows an average number of invocations of 5.4 with a median of 1 and standard deviation of 11.7.

Device drivers for Windows CE are implemented as dynamic link libraries which are loaded into the memory of a dedicated driver process at system load time. Both drivers implement the standard stream interface used for such drivers in Windows CE which exports the ten functions. Table 1 show the exported services by the serial port driver. The COM prefix is specific to the serial port driver and is different for other drivers. The order in which drivers are loaded in the system is configured using the registry. In Windows CE network card drivers do not use the normal stream interface to interact with the OS, instead the driver uses the `NDIS.dll` library for implementing the needed functionality. We have therefore built an intermediate NDIS driver that tracks all NDIS calls to the driver. As drivers in Windows CE are implemented as dynamic link libraries (Dll's) a dedicated function exported by the driver (`DllMain` for the serial driver and `DriverEntry` for the network driver invoked as the first thing when it is loaded into the system. Typically some basic setup of synchronization objects etc. is done in this function. On top of this both drivers provide additional initialization functions called by the OS at a later time.

Apart from the exported services each driver makes use of services provided by the OS and other system libraries, possibly including other drivers. This is the interface we target for FI.

5.2. Experimental Process

The tracker module keeps track of which calls have been made to the driver and notifies the injector module when the

Table 1. Stream interface for serial driver.

Number	Name
0	COM_Init
1	COM_Deinit
2	COM_Open
3	COM_Close
4	COM_Read
5	COM_Write
6	COM_Seek
7	COM_IOControl
8	COM_PowerDown
9	COM_PowerUp

targeted call block has been reached. The injector module then injects the error using first occurrence injection. For the first occurrence experiments the injector module is configured to ignore the instructions from the tracker module and instead inject according to the first occurrence strategy. The system also has a manager module, handling setup and selection of injection cases, and a host computer for storing the log files produced for each injection.

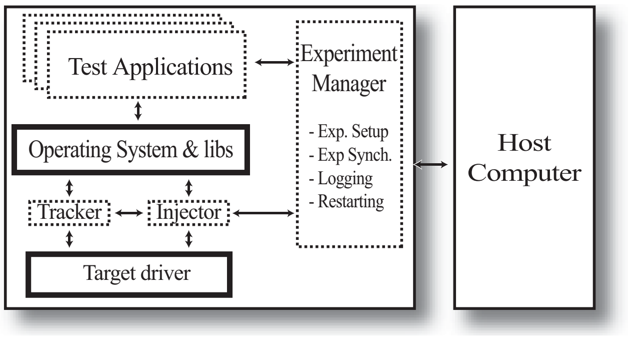

Figure 6. The experimental setup.

For each experiment run (injections for a specific driver and call block) a new image of the whole system is built, downloaded and stored in flash memory on the reference board. When the system boots up it loads the image from flash into RAM, thus guaranteeing that each injection is performed on a fresh copy of the correct OS image. The first time the system boots up it configures all the injections that can be performed and stores them in a designated flash device. It then proceeds and runs the workload and profiles the system filter out non-activated injections (see next section). Logs are sent to the host computer over either Ethernet or serial communication, depending on which driver is targeted. After each injection the system is rebooted. This way dormant errors are not present at the time of injection. However, it does limit the ability to study effects of multiple errors and possible state degradation effects.

5.3. Pre-profiling

For any FI approach, keeping the number of activated injections (those where the injected fault turns into an error) high is desirable as it reduces the time needed to conduct the experiments. To achieve a 100% activation rate a pre-profiling phase is employed before each injection campaign. The system is executed without injecting any error, recording which services are used and in which call blocks. All injection cases which will not be activated can thus be filtered out a priori, either because the service is not used at all or because it is not used in the targeted call block.

5.4. Workload

To drive the experiments and to make sure that the targeted drivers are used, a workload consisting of a set of test applications are executed on the system as indicated in Figure 6. There are two types of test applications, OS level test applications exercising generic OS services, such as memory management, process and thread management, file system operations etc. The second type is device specific test applications, dedicated to specific devices attached to the system. For the reference boards used in this study we have dedicated test applications for attached external storage (CompactFlash), serial and Ethernet communication with the host computer. The device specific test applications first set up the application level setup required to use the driver, then sends and receives data and finally closes the communication. This mimics the operational phases of the driver itself, with initialization, working and cleanup phases as shown previously in Figure 4 and we consequently call these the operational phases of the test application.

5.5. Injection

At system boot time the targeted service for injection is read from a configuration file. As the driver is loaded, a wrapper is built enabling tracking and injection of errors into that service. As the service is triggered for injection, a selected bit in the specified parameter of the service is flipped. The remaining bits are left unchanged. Once the bit is flipped the system continues running. The error is only injected once. Subsequent service invocations are not targeted.

For first-occurrence injections the error is injected the first time the service is invoked. For the proposed call block approach the injection is performed only after the specified call block has been reached. The actual injection is exactly the same, only the triggering even differs.

5.6. Identifying Call Strings & Call Blocks

To find the call blocks the drivers are first exercised without any errors injected. The tracker module records the calls made, and in which order, forming the call string for the driver. For the setup used in this paper, the workload is deterministic, i.e., the same call string is generated every time. This simplifies the problem of tracking the call blocks as it is reduced to counting the calls, and triggering injection when a specified number of invocations have taken place.

Serial port driver: For the serial driver the test application first writes a string of characters to the serial port and an application on the connected host computer reads them and sends the same characters back, which are read, character by character. This is then repeated once more. On top of this, the `DllMain` function is called before any other call to the driver, forming a call block by itself. In total the call string for the serial port driver contains 152 tokens.

Figure 7 shows the call string for the serial driver. The number for each service invoked is taken from Table 1. It shows the run starting with a call to `DllMain` (`D`) followed by service 0 (Init) which both belong to the initialization phase of the driver (and the workload). The working phase of the driver consists of a sequence of *Open* calls and calls for writing characters to the port, followed by a repeated sequence of reads and control calls. The clean up phase of the workload then ends with a call to *Close*. This pattern then repeats (open + writing + reading) a second time, but without the initialization calls. The application consistently gives rise to the same call string. A discussion on non-determinism and concurrency is found in Section 8.

D02775(747){*23*}732775(747){*23*}73

Figure 7. The serial driver call string.

Figure 8 illustrates the call string. Five distinct call blocks have been identified ($\delta, \alpha, \beta, \gamma$ and ω), with all but the first two (δ and α) repeating. We target each repeat once and for the consecutive repeats of γ we target one of the repeats, in the first run the first block, and in the second run the sixth. For the sake of discussion in the following sections we term the targeted call blocks $\delta, \alpha, \beta_1, \gamma_1, \omega_1, \beta_2, \gamma_2$ and ω_2 as shown along the x-axis in Figure 8, here indicating time in terms of call block ordering.

Network card driver: The workload for the network card driver works in a similar fashion as the serial port driver, by sending packets of data to the host machine and then listening for incoming packets (the same data sent back). Similarly to to the serial port driver we have manually inspected the call string to the driver and identified eight call blocks with two of them repeating, as illustrated in Figure 9. The call blocks are named in a similar fashion

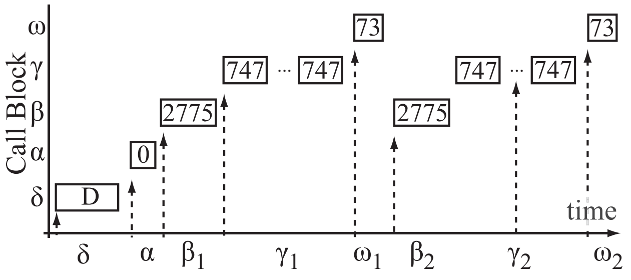

Figure 8. The call blocks for the serial port.

to the call blocks for the serial port driver. The network card call string contains 59 tokens.

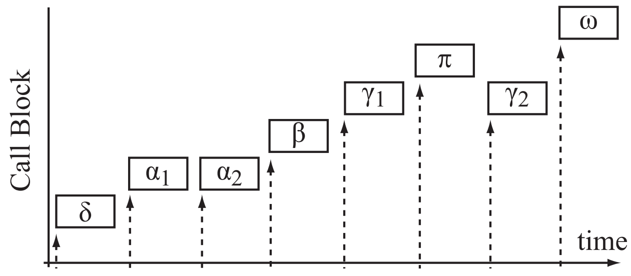

Figure 9. The network driver call blocks.

6. Evaluation

To evaluate the efficiency of the call block based injection we will perform experiments with the proposed technique and compare it with a classical first occurrence based injection. The comparison will focus on the ability to find severe failures and on the required number of injections (i.e., the execution time of the experiments).

The comparison uses failure mode analysis. The failure modes are defined from a user perspective, i.e., based on the service provision of the system. Failure mode analysis is a common approach used to classify FI experiments, see for instance [2, 13, 18] Four failure classes are defined as follows, based on the behavior during the golden run of the test applications.

Class NF: No visible deviation of service behavior, i.e., *No Failure* class. Three distinguishable explanations account for an injection resulting in this class, namely a) the error location not being activated; b) the error being overwritten by the system; or c) the error being dormant, i.e., still present in the system but propagation is not yet activated.

Class 1: A deviation from golden run behavior visible for the OS services, but still satisfies the OS service specification. Examples of class 1 outcomes include returning valid error codes, propagation of data errors also fall into this category.

Class 2: The specification of the service is violated, like an unforseen hang or crash of the *application*. An application is considered hung after 40 seconds of non-responsiveness, exceeding 100% of normal execution time. Note that the rest of the system is still functioning after the failure.

Class 3: The OS becomes irresponsive due to a crash or a hang. No further progress is possible.

7. Results

For each of the two drivers we carried out FI experiments as previously described. Table 2 details the results obtained, with names as defined in Section 5.6. Using these results Figure 10 and 11 graphically compares the results obtained with the call block approach to those obtained with first occurrence. For a trigger to be useful it should find many high severity failures (Class 3) with as few injections as possible.

Serial port driver: Figure 10 shows the distribution across failure classes for each of the call blocks (and first occurrence, FO) used together with the number of injections carried out for each call block. The NF class, which amounts to 75-95% of the cases is not shown in the figure.

The call blocks showing the lowest ratio of Class 3 failures are γ_1 and γ_2. These call blocks correspond to the working phase of the driver, where it is sending and receiving data. In this phase one can expect the system to be built to be better prepared to fluctuations in behavior. The figure also clearly shows a small difference in the number of Class 2 failures seen, with γ_2 have a slightly higher ratio.

In contrast to the working phase, the initialization and clean up phases show higher ratios of Class 3 failures. In these phases the driver is interacting with the OS using sensitive OS services. Whereas ω_1 and ω_2 show close to identical distributions, β_2 shows more Class 1 failures than β_1.

Figure 10 also shows that the δ call block is less prone to Class 3 failures than the other initialization phase α. This can be explained by the fact that during load time drivers are discouraged to do any complex operations, minimizing the calls being made to the OS only initializing synchronization objects and other lightweight operations. As a consequence we see fewer Class 3 failures, but since these operations may now fail, we see a rise in Class 2 failures as the driver is unable to do any progress.

Looking at the specific services where injecting errors actually gives rise to Class 3 failures Table 3 shows the results for the serial port driver. Out of a total of 41 services being targeted for this driver, injections in 13 services caused Class 3 failures. From Table 3 it can be seen that five services which did not show any Class 3 failures for the first occurrence injections, do show failures when injected

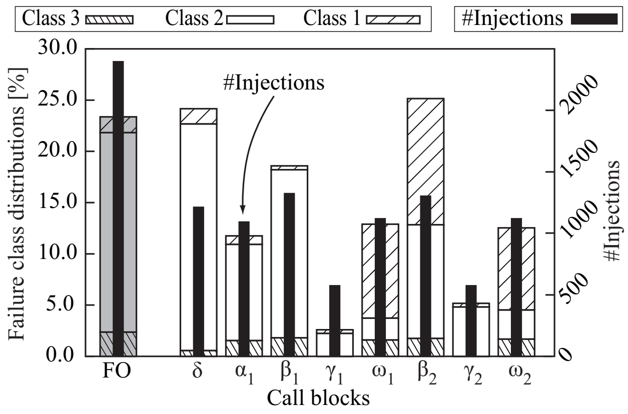

Figure 10. Failure class distribution and number of injections for each call block of the serial port driver.

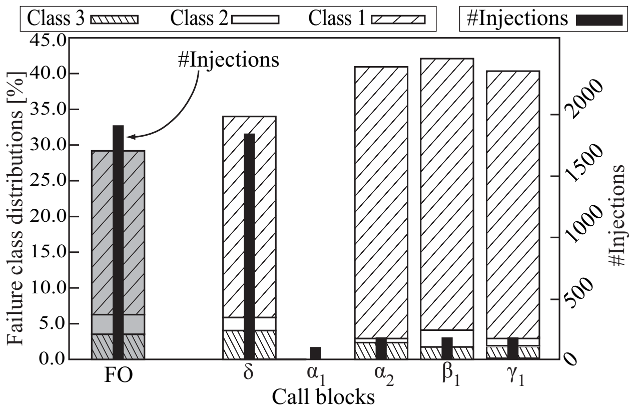

Figure 11. Failure class distribution and number of injections for each call block of the network card driver.

in later call blocks. This shows that *choosing the triggering event for injection for the serial port driver does have a large impact on the results obtained* and that *first occurrence is not sufficient for a comprehensive evaluation.*

Network card driver: Table 2 and Figure 11 show that for the network card driver the call block δ shows a very similar distribution to the first occurrence approach (FO). The call blocks π, γ_2 and ω_2 defined in Section 5.6 did not have any injections performed as no service invocations take place. These call blocks are not shown. Call blocks α_2, β_1 and γ_1 show very similar behavior, with a distinctly lower number of injections. Call block α_1 does not show any failures at all.

As speculated in Section 5.1 the impact of the call block strategy is minimal for the network card driver. No new Class 3 failures where found and the δ call block shows a very similar behavior to the first occurrence (in contrast to

Table 2. Injection results for the serial (cerfio_serial.dll) and the network card driver (91C111.dll).

Driver Name	Call Blocks	#Injections	NF		C1		C2		C3	
			#	%	#	%	#	%	#	%
cerfio_serial.dll	FO	2436	1872	76.85%	37	1.52%	467	19.17%	60	2.46%
	δ	1217	923	75.84%	18	1.48%	269	22.10%	7	0.58%
	α	1098	969	88.25%	9	0.82%	103	9.38%	17	1.55%
	β_1	1356	1082	79.79%	5	0.37%	218	16.08%	51	3.76%
	γ_1	580	565	97.41%	2	0.34%	13	2.24%	0	0.00%
	ω_1	1124	979	87.10%	103	9.16%	24	2.14%	18	1.60%
	β_2	1328	979	73.72%	161	12.12%	145	10.92%	43	3.24%
	γ_2	580	550	94.83%	2	0.34%	28	4.83%	0	0.00%
	ω_2	1125	984	87.47%	90	8.00%	32	2.84%	19	1.69%
91C111.dll	FO	1820	1289	70.82%	417	22.91%	50	2.75%	64	3.52%
	δ	1756	1159	66.00%	494	28.13%	32	1.82%	71	4.04%
	α_1	96	96	100.00%	0	0.00%	0	0.00%	0	0.00%
	α_2	171	101	59.06%	65	38.01%	1	0.58%	4	2.34%
	β_1	171	99	57.89%	65	38.01%	4	2.34%	3	1.75%
	γ_1	171	102	59.65%	64	37.43%	2	1.17%	3	1.75%

the serial port driver), which *confirms that for drivers which perform few calls, and especially during the initialization phase, first occurrence is to be preferred*.

8. Discussion

This section interprets the results presented in the previous section and discusses some of the issues involved. Focus of the discussion is on the most severe failures from a robustness point of view, the Class 3 failures.

Driver types: The results show a distinct difference between the two drivers. Where for the serial driver a significant number of additional services experienced Class 3 failures the network card driver gave no additional services using the call block strategy. As explained in Section 5.1 the serial port driver relies heavily on the OS to perform its services. As expected, the call block approach is only more effective for drivers which use many services multiple times throughout the lifetime of the driver, which is not the case for the network card driver used in this study. *A profiling of the targeted drivers (like in Figure 5) should therefore be conducted before triggers are selected*, to minimize the time for implementation and number of injections required. Note that such profiling can be conducted *prior* to injection!

Comparing with first occurrence: The first occurrence approach has several advantages compared to using call blocks: it uses fewer injections, it is useful for code level faults and it is simple to implement. Table 4 shows a quantitative comparison of the two approaches.

The number of injections needed for the call block approach is significantly higher than that for first occurrence, even though the pre-profiling reduces the number of injections needed for each call block. However, the higher price is inherently inevitable given the fact that the call block approach injects in multiple invocations of a service, whereas first occurrence only injects in the first invocation, i.e., the first occurrence injections is a subset of the injections performed when all call blocks are targeted.

The additional number of injections give rise to a trade-off with the usefulness of the results gained. Section 7 clearly shows that there are indeed services found to be vulnerable only using the call block approach. Using first occurrence found eight services for the serial port driver to be vulnerable. Using the call block approach additionally five more services were identified, an increase of more than 60%. On the other hand, for drivers that do not use the OS services as much, or mostly during initialization, the call block approach gives no benefit.

Selecting Call Blocks: Figure 10 and 11 show that not all call blocks are effective at identifying Class 3 failures. By inspecting the call blocks that do not show any such failures it becomes evident that these are mostly in the working phase of the driver as well as the workload. To reduce the number of injections performed one could focus on mostly the initialization and clean up phases.

Finding Call Blocks: Depending on the workload used for the evaluation the call strings induced varies in length. For the case study presented in this paper manual inspection was sufficient to deduce relevant call blocks. However, when call strings are significantly longer and when knowledge about the semantics of services is lacking, manual inspection may not be a feasible option. For this case some level of automation is required. The repeating nature of call blocks can be used to identify them within the call string.

Table 3. The services having Class 3 failures for the serial port driver.

Service/Call block	FO	δ	α	β_1	γ_1	ω_1	β_2	γ_2	ω_2
CreateThread	x			x			x		
DisableThreadLibraryCalls	x	x							
EventModify						**X**			**X**
FreeLibrary	x	x							
HalTranslateBusAddress			**X**						
InitializeCriticalSection		**X**							
InterlockedDecrement									**X**
LoadLibraryW	x	x							
LocalAlloc	x	x							
memcpy	x			x			x		
memset	x			x			x		
SetProcPermissions	x			x			x		
TransBusAddrToStatic			**X**						

Table 4. Comparing the first occurrence and call block approaches.

Trigger	Serial driver		Network driver	
	#Injections	#C3	#Injections	#C3
First occ.	2436	**8**	1820	12
Call blocks	8408	**13**	2356	12

Several algorithms and data structures have been developed for similar problems, like identification of repeating substrings in dna (see for instance [6]), which can be applied here as well. A future extension of this work is indeed to develop such algorithms.

Determinism: A call block represents an operation carried out by the driver and may consist of more than one service invocation. To define the call blocks for a driver the call string needs to be stable, i.e., the invocation pattern needs to be deterministic. The workload used for the case study is indeed deterministic and actually most benchmarks are, since they should provide repeatable results. An application used for a specific product may not be deterministic and may therefore have to be transformed in such a way that the invocation pattern it induces is deterministic.

Concurrency: Drivers can in general be accessed concurrently by multiple applications, depending on the semantics of the driver. For instance, the serial driver used in the case study uses a non-sharable semantics (only one application can have access to the serial port at a given time), whereas other drivers can be accessed concurrently. In this paper we have deliberately focused on the simpler case, where only single applications access the driver at any given moment in time. In order to handle concurrent accesses the call strings relating to different applications need to be kept apart and treated individually. Call blocks can then be defined for each of them individually. However, the interleaving of the accesses may still be problematic. Whether such a situation is desirable or not from a robustness evaluation point of view can be discussed, as it potentially leads to undesirable non-determinism as described above.

Workload selection: As the phases and the call blocks form the basis for the usage profile it is important to identify a representative workload for the experiments, representative of the expected workload of the system once it becomes operational, if known, or representative of common applications within the targeted area if not. In the latter case it is important to use a diverse workload that exercises the systems in all possible ways that may be required of real applications. Load and stress scenarios may also be part of the workload, but it is important that the workload can be executed without any problems in error-free settings, such that a golden-run comparison can be made.

9. Related Work

Multiple robustness studies of OSs and other software systems have been carried out. [8, 14] studies the impact of software faults in device drivers using code mutations. Here not only the type of errors injected (code level) but also the location of the injection differs from ours. We have opted for a more general technique by using the interfaces between components for injection.

Interface level errors where used for instance in [1] and [2] for the Linux kernel and for micro-kernel-based systems respectively. It was also used for user level applications for instance in [4, 13, 20].

Injection timing has been identified to be of great importance to the effectiveness of fault injection. Whittaker notes that injecting faults into SW interfaces is important for testing robust and reliable systems, but that it is very hard to know where and when to inject [22]. [21] injects errors into CPU registers and memory locations by flipping bits. The goal is to select the time of injection to maximize the activation rate of errors. Both workload based and path-based injection is studied. The path-based approach has more similarities with our technique but it is based on the assembler code of the program. The execution paths for given inputs are found a priori and faults are selected along these paths, using the code location as trigger. Also [7] considers the time of injection, but again based on source code level errors and not from the interface level as our approach. In [17] the authors injects bit-flips in the stack area corresponding to the parameters in a call using a randomly selected time of injection. We believe that with the approach presented in this paper increases the effectiveness of the injections by controlling the time of injection to maximize the effects of the injections. Multiple other tools allow controlling the injection of errors to some degree, for instance FERRARI [11] which allows a user to set both spatial (location) and timed triggers. The spatial trigger injects an error after a location is invoked n time. This is similar to our proposed method, but our approach does provide guidance on selecting n, using the call block concept.

10. Conclusions & Outlook

This paper presents a novel approach for selection of injection triggers for OS robustness evaluation. The usage profile of a driver (a list of invocations to the driver) ia split into disjoint call blocks and each block is targeted for injection. The results presented establish the fact that controlling the time of injection is crucial. Applying the approach to Windows CE shows that a significantly higher number of services can be identified as vulnerable, compared to a traditional first occurrence based approach. A profiling of the invocation pattern of the driver can be conducted to give insights on whether call block triggers or a traditional first occurrence trigger will be more effective. Such profiling can be conducted before any injections have taken place. Furthermore, inspection of the results show the initialization and clean up phases of the drivers having a distinctly higher number of vulnerabilities than the working phase.

Future extensions of our work include enlarging the set of models and drivers studied. We are also looking into methods for automatic detection of call blocks from call strings. Applying the technique in a larger industrial scenario would allow for more detailed evaluation of the efficacy of the used models and study of the bug revealing capabilities of different models.

References

[1] A. Albinet, et. al., Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel. In *Proc. of DSN*, pp. 807–816, 2004.

[2] J. Arlat, et. al., Dependability of COTS Microkernel-Based Systems. *IEEE Trans. on Computers*, 51(2):138–163, Feb. 2002.

[3] A. Chou, et. al., An Empirical Study of Operating System Errors. In *Proc. of SOSP*, pp. 73–88, 2001.

[4] C. Fetzer and Z. Xiao. An Automated Approach to Increasing the Robustness of C Libraries. In *Proc. of DSN*, pp. 155–164, 2002.

[5] W. Gu, et. al., Characterization of Linux Kernel Behavior under Errors. In *Proc. of DSN*, pp. 459 – 468, 2003.

[6] D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997.

[7] J. Christmansson, et. al., An Experimental Comparison of Fault and Error Injection. In *Proc. of ISSRE*, pp. 378–396, 1998.

[8] J. Durães and H. Madeira. Multidimensional Characterization of the Impact of Faulty Drivers on the Operating System Behavior. *IEICE Trans.*, E86-D(12):2563–2570, Dec. 2003.

[9] A. Johansson and N. Suri. Error Propagation in Operating Systems. In *Proc. of DSN*, pp. 86–95, 2005.

[10] A. Johansson, et. al.,. On the Selection of Error Model(s) for OS Robustness Evaluation. In *Proc. of DSN*, 2007.

[11] G. A. Kanawati, et. al., FERRARI: A Flexible Software-Based Fault and Error Injection System. *IEEE Trans. on Computers*, 44(2):248–260, Feb 1995.

[12] K. Kanoun, et. al., Benchmarking the Dependability of Windows and Linux using PostMark Workloads. In *Proc. of ISSRE*, pp. 11–20, 2005.

[13] P. Koopman and J. DeVale. Comparing the Robustness of POSIX Operating Systems. In *Proc. of FTCS*, pp. 72–79, 1999.

[14] W. lun Kao, et. al., FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults. *IEEE Trans. on Software Engineering*, 19(11):1105–1118, Nov 1993.

[15] B. Murphy and B. Levidow. Windows 2000 dependability. In *Proc. of the Workshop on Dependable Networks and OS*, pp. D20–28, 2000.

[16] J. Musa. Operational Profiles in Software-Reliability Engineering. *IEEE Software*, pp. 14–32, Mar. 1993.

[17] M. Rodriguez, A. Albinet, and J. Arlat. MAFALDA-RT: A Tool for Dependability Assessment of Real-Time Systems. In *Proc. of DSN*, pp. 267–272, 2002.

[18] A. Steininger and H. Schweinzer. A Model for the Analysis of the Fault Injection Process. In *Proc. of FTCS*, pp. 186–195, 1995.

[19] M. M. Swift, et. al. Improving the Reliability of Commodity Operating Systems. In *Proc. of SOSP*, pp. 207–222, 2003.

[20] T. Tsai and N. Singh. Reliability Testing of Applications on Windows NT. In *Proc. of DSN*, pp. 427–436, 2000.

[21] T. K. Tsai, et. al., Stress-Based and Path-Based Fault Injection. *IEEE Trans. on Computers*, 48(11):1183–1201, Nov 1999.

[22] J. A. Whittaker. *How to Break Software*. Addison-Wesley, 2003.