Abstract—This paper presents FITIn, a bit-error injection tool designed for evaluating software-implemented hardware fault tolerance (SHIFT) mechanisms. Like most bit-error injection tools, FITIn injects faults at run time into the binary of a program. Unlike previous bit-error injection tools, FITIn allows a software developer to control the targets of injection campaigns at the level of a higher programming language rather than assembler. FITIn is implemented as a Valgrind plugin and has been tested for C programs. We present its architecture, demonstrate its functioning using examples from three benchmarks (Dhrystone, STAMP, and CoreMark), provide performance figures, and discuss general limitations of the approach.

I. INTRODUCTION

Traditionally, testing tools aim at detecting logical errors, i.e., human mistakes in the design or implementation of software. Bit errors, on the other hand, rarely represent logical errors but rather hardware faults or malicious attacks. Examples of bit errors in hardware include single event upsets (SEUs), caused by natural radiation or temperature instability on the chip; examples of malicious intervention include bitsquatting where internet domains are registered that differ from popular ones by one bit only. In this paper, we focus on bit errors that are caused by hardware faults but the tool we introduce, FITIn, handles bit-error attacks just as a special case.

Bit errors in hardware are measured in FIT, i.e., failure in $10^9$ hours of operation. Even though the probability is low that a bit error hits an application in an observable way, examples of prominent failures exist; the outages at Sun Microsystems and the pending Toyota lawsuit, for example, are well known [1][2]. Moreover, computer engineers expect in the near future a dramatic increase of transient or intermittent SEUs due to temperature instability on the chip or effects like hot carrier injection. Altogether, protection mechanisms against bit errors are needed not just in the classical safety-critical domains but also in server farms or whenever applications are large or long living. Testing for bit-error vulnerability, however, is complex since bit errors can strike anytime and anywhere in the code. The most common testing approach today is fault injection, a low-level variant of mutation testing where the mutation operator corresponds to a bit flip and mutation takes place in the executable at run time. While injections in a binary represent the closest approximation to hardware faults there is, those tests are too low level for software developers and make it difficult for them to take appropriate protection steps since it is hard to relate binary faults and source-level entities.

This paper presents FITIn, a tool to inject bit errors that allows the tester to select injection targets in terms of high-level program variables instead of low-level registers. Software developers can use FITIn to test in a controlled way and at source-code level both vulnerability to bit errors and any source-level protection mechanism they may employ. FITIn has been tested on x86 and AMD64 systems with Linux and Darwin operating systems. Currently, faults can be injected in variables of all types except float and double. The remainder of the paper discusses the architecture and related work in Sects. II and III, demonstrates the usage of FITIn in Sect. IV, and evaluates it for robustness and performance in Sect. V; Sect. VI concludes. A previous version of FITIn has already been presented, but it supported injections in memory only and not also in registers, and had a number of additional limitations that prohibited the kinds of experiments presented in this paper [3]. FITIn is available for download from http://www.sts.tuhh.de/research/fitin.html.

II. ARCHITECTURE

FITIn is a plugin for version 3.8.1 of the dynamic binary instrumentation (DBI) framework Valgrind [4]. Valgrind is known for a number of famous dynamic binary analysis tools such as Memcheck for spotting memory leaks and memory access errors, CacheGrind for analyzing the cache-friendliness of a program, or Helgrind for identifying race conditions. Valgrind accepts any native program on the supported platforms. Starting from the entry point, Valgrind disassembles the program into blocks of instructions, super blocks (SB). Unless modified later, each SB is translated once into an internal representation (IR). The IR is handed over to the loaded plugin, FITIn in this case, which analyzes and instruments the IR. After processing the SB, Valgrind compiles the modified IR back into native instructions and executes the SB. When finished, Valgrind continues to the following SB according to the actually executed code path, the main trait of DBI. Fig. 1 visualizes this process as control-flow graph.

![Fig. 1: Program execution of Valgrind](http://www.sts.tuhh.de/research/fitin.html)
For the recognition of error injection targets, FITIn provides a C-header file that introduces macros. The tester selects them to annotate the source code for enacting the observation of the targets. These macros make the compiler allocate a little additional space on the stack: for an encoded command and parameters. When executed by Valgrind, it forwards this client request to FITIn. Aside from these control sequences, FITIn consumes the IR of the program code, looking for uses of observed targets and waiting for performing an error injection. The error injection time is given as a numeric command line parameter, which expresses the n-th operational read, counting all active targets.

III. RELATED WORK

Many fault-injection tools exist in the literature but there are no freely available tools under active development that are ready to use on x86 platforms. Moreover, the existing tools all pursue injection strategies that differ from the one in FITIn. FERRARI uses a way of process-hijacking, waiting for an interrupt handler coming into play after an elapsed timer or a certain number of occurrences of memory addresses [5]. Xception, originally supported by dedicated debugging features of the PowerPC platform, targets registers as well as the CPU, which is not relevant to FITIn because it targets only regions that can be addressed at source level [6]. FITgrind, like FITIn, is implemented as a Valgrind-based tool, injecting bit errors into memory in a random manner [7]. GOOFI-2 also supports different injection vectors: via a specific hardware setup, by setting hardware breakpoints, or by static binary instrumentation that places software breakpoints and uses binary analysis to ensures the breakpoint is reached [8]. FITIn, in contrast, gives the developer control of the injection targets directly at the source code and allows for flexible scheduling at run time. Its strategy reduces the need for pre- and post-analyses of memory addresses and for finding links back to program variables.

Pin and DynamoRIO are two common DBI frameworks that present alternatives to Valgrind [9][10]. From a developer’s perspective, however, the advantages of Valgrind over the other ones include full abstraction of the CPU and of the operating system, implicit handling of multi-threading, an encapsulation of external events (e.g., system calls accessing process memory or registers), and support for client communication towards the framework.

IV. USING FITIN

As FITIn is working exclusively on the IR composed by Valgrind, error injections cannot be performed directly on machine instructions. FITIn flips a selected bit on data that is used as an operand of a CPU instruction. The user specifies a variable—or, via a pointer, a block of memory—in the source code, and FITIn looks for IR-encoded read instructions that access the memory corresponding to the variable. The IR of Valgrind hides whether the unit of data is stored in a register and used in later instructions or whether it is accessed directly by an instruction. This way, FITIn offers an easy way for scheduling the error injection on memory: By looking at the source code, the user places the macro FITIN_MONITOR_VARIABLE somewhere after the declaration of a variable. Next, the user counts the number of uses of this variable—respecting code branches, loops, and subroutine calls—until the end of its lifetime or until the user manually disables the variable observation by using FITIN_UNMONITOR_VARIABLE.

The example given in Fig. 2 demonstrates the testing of the integer variable a. Here, as the branching conditions are constant, 34 read operations on a can be easily spotted that increase the counter of FITIn’s scheduler. Assuming a program that only consists of the displayed code lines, a user may start Valgrind and FITIn and declare --mod-load-time=29 and --mod-bit=1. Those two command-line parameters schedule the bit flip right before the 29th access on the counter, the first subtraction from a by one, and request the second bit to be inverted (0 denotes the least significant bit). At the end of its lifetime, a will then carry the value 2147483657; without bit flip it would carry the value 2147483641.

Injecting bit errors in variables relevant for the control flow might lead to infinite loops. FITIn therefore allows the user to limit the absolute number of instructions to be executed by the client program.

V. EVALUATION

In this section, we present three case studies that demonstrate how FITIn can be used to find programs vulnerable to bit errors, evaluate FITIn’s performance and robustness, and report about discovered limitations.

A. Case Studies

The purpose of a fault-injection tool is to discover parts of programs that are vulnerable to bit errors. To evaluate FITIn, we conducted three case studies in which we used FITIn to inject bit errors in Boolean variables to test the robustness of applications against SEUs. In addition, we explored the effectiveness of FlipSafe, a SIHFT library. In particular, we used the Benchmarks Dhrystone-2.1 [11], STAMP-0.9.10 [12], and CoreMark-1.0 [13].

First, we describe the general setup of our experiments. Then, we present the specifics and findings of each individual fault-injection campaign. The Dhrystone experiments were run on a 1.73 GHz Intel Celeron M530 in 32 bit mode with 1.5 GB RAM, Linux-3.8, and GCC-4.7.3. The STAMP and CoreMark
experiments were run on a 64 bit 3.3 GHz Intel Core 2 Duo system with 6 MB L2 cache, 8 GB RAM, Linux-3.0.0, and GCC-4.6.1. To be able to use the C++ library FlipSafe, we compiled all applications with g++ even if we used FlipSafe only with Dhrystone until now.

Each fault-injection campaign started with the selection of one or more Boolean variables into which the bit errors should be injected. Then, the programs were compiled and run under FITIn without injecting errors to determine the number of read accesses to the selected variables (golden run). Next, the series of fault-injection runs was started. Each run flipped a bit of the monitored variable at a read access between the first and the total number of read accesses. After each run, the output from the program was parsed to determine whether an error could be observed or whether the program ran correctly despite the bit flip. Finally, these observations were compiled into a statistic, as depicted in Fig. 3 and Fig. 4.

In the first case study, we used a customized version of Dhrystone. To make the execution deterministic, calls to time-measurement functions were removed, the number of runs was set to a fixed number, and function Proc_4 was modified to provide better error propagation. For the experiment, we selected one Boolean variable that is accessed 151 times. Then, we conducted a series of fault injection runs, one for every access and bit of the variable. This led to 182 observable faults. After that, we changed the type of the selected variable to one of the hardened types from FlipSafe and repeated the experiment. The observed errors for the unprotected case and all Boolean types from FlipSafe are depicted in Fig. 3. Except EPBOOL, all protected types work as expected. That FITIn revealed so many observable faults for EPBOOL indicates to the developer to check the design and implementation of that type. Since these errors occur exactly if one of the bits 7, 15, 23, or 31 is flipped, FITIn helps to further pinpoint the flaw.

In the second case study, we used the STAMP suite. In contrast to the first case study, we selected all Boolean variables as potential fault-injection targets at the same time, and randomly chose one read access and bit to flip for each run. The results for 100 runs are depicted in Fig. 4. The Boolean variables in many of the STAMP suite’s applications are quite robust, but FITIn identified the program labyrinth as being susceptible to bit flips.

In the last case study, we experimented with CoreMark. To analyze the impact of bit errors on control flow, we extracted the predicates from if, for, and while statements, and stored their values in temporary variables. Then, we selected these temporary variables as fault-injection targets and started the fault-injection runs. The results are also depicted in Fig. 4, and show that, as expected, bit errors in branching conditions have a significant impact on the correctness of the application. If SEUs are a concern, the developers should implement appropriate mitigation strategies.

B. Performance

After we demonstrated how FITIn can be used to support software developers, we will now report about its performance. To that end, we compare the average time it takes to run the selected benchmark applications directly on the hardware, inside Valgrind but without FITIn (using Valgrind’s plugin none), and with FITIn. Valgrind adds overhead because it has to translate the machine code into IR and back. Additionally, it has to allow clients to monitor and manipulate the program’s execution. FITIn’s overhead is due to its instrumentation of read accesses and internal bookkeeping.

Performance measurements were conducted with the benchmarks from the case studies and Linpack. We measured the time from starting the application or FITIn on the shell until its execution terminated. For each application, we computed the arithmetic average of 100 executions. The overhead is depicted in Fig. 5 with the time for the native execution normalized to 1. Depending on the application, the execution is slowed down by a factor between 61 and 152. The measured slow down of Linpack and Dhrystone seems comparatively small. However, the internal score of these benchmarks is reduced by a factor of approximately 150, which is consistent with the slow down of the other benchmarks.

We have identified some bottlenecks related to the IR analysis of FITIn. For example, the processing time is quadratic in the number of loads from main memory in the current implementation but can be improved by using a more efficient
data structure. Thus, we are confident that we can improve FITIn’s performance in the future.

C. Robustness

The method to determine the parameter mod-load-time by counting read accesses in the source code may be invalidated by a compiler that generates an unexpected number of such accesses in the machine code. One obvious reason for deviations is code optimizations, but these can usually be disabled. Another, not so obvious, reason is the generation of excess read instructions. We observed this with GCC on MacOSX. The C code for function \( f \) given on the left of Fig. 6 was translated to the sequence of instructions shown on the right of Fig. 6. First, the value of \( a \) at \([\text{EBP}-16]\) is loaded into register EAX (expected read). Then, the value is incremented and written back to memory. After that, the value of \( a \) is unexpectedly loaded and stored again into EAX and hence deceives the unaware user.

Since FITIn does not know about the high-level source code, it cannot account for a deviating number of read accesses. If in doubt, the tester has to check the generated assembly code manually. Fortunately, these cases seem rare: we are only aware of this one. Future versions of FITIn may utilize compiler-generated debug information to relate the high-level source code to machine code and thereby mitigate compiler effects.

D. Limitations

Our experiments show that FITIn is a valuable fault-injection tool. Nevertheless, it has limitations arising from peculiarities of Valgrind. Valgrind handles variables of some floating-point types different from other variables. This different handling does not allow direct manipulations, and therefore, FITIn is unable to inject faults into these variables. In particular, this applies to the C types float and double, while 80-bit floating-point variables are unaffected. Another limitation is that accesses to loop-counter variables are not recognized for certain loops. In these cases, Valgrind distributes the instruction that accesses to the affected variable over several SBs in a way that FITIn cannot follow yet. We are currently exploring strategies to overcome these limitations.

VI. Conclusion and Future Work

While bit-error injection necessarily takes place at assembly level, FITIn presents a testing approach that is accessible to software engineers since tests are specified in terms of high-level program variables rather than low-level registers. It is possible, thus, to test in a controlled way at what point in a program a variable is vulnerable to a bit error. Inevitably, the instruction order at source-code level and assembler level do not exactly align, so that FITIn cannot even for unoptimized programs guarantee to hit the right program point. In the tests presented in this paper, however, FITIn proved to be sufficiently precise.

Future work on FITIn includes additional robustness testing on the platforms supported by Valgrind and performance improvements; additional customizations for timing and performing the bit-errors experiment; and the utilization of debugging information to reduce the amount of annotations a tester might have to do.

REFERENCES

APPENDIX
DEMONSTRATION
INSTALLATION

For installing FITIn, we currently recommend Linux Kernel 3 or newer and glibc library not newer than v2.16. Required tools are autoconf, automake, and a C compiler; if one optionally wants to run the test suite that is part of the distribution, then Ruby v1.9 or newer and the rake-gem are additionally required. Please follow the given URL\(^1\) to download the latest version of FITIn.

The distribution contains the Valgrind source (currently release 3.8.1) where FITIn is embedded in the subdirectory fitin. FITIn and Valgrind, thus, do not exist independently from each other: when building FITIn, Valgrind gets built as well.

After successfully downloading and extracting the archive file, please open a shell and move to the extraction target. Execute the following commands for completing the installation:

```bash
./autogen.sh
./configure --prefix=$pwd
make && make install
```

For running the test suite of FITIn, proceed to the subdirectory fitin/tests. Please consult the file README.md for setup and start configuration.

EXTENDED EXAMPLE

Let us consider the following example for demonstrating the use of FITIn: a control software of a passenger elevator where one section takes care of the cleared weight. For the sake of the example, the elevator is described by three bits of the variable state: for indicating power supply, the operating state, and overload. We also assume a sensor measuring the weight, using negative values to report a failure. In this case, the elevator will stop its service for safety reasons. In this section, we walk the reader through two fault-injection experiments; a video is available from the FITIn website.

As the bits for power and operating are decoupled, a hardware error could lead to an invalid live-lock situation if the operating-bit was cleared independently from the power-bit. Furthermore, corrupting the overload-bit could cause a false-alarm for too much weight.

The source code subject to this FITIn presentation is shown in Fig. 7. We are interested in the reaction of the program if we manipulated any bit of state at an arbitrary time.

We start by including fi_client.h, so that, among other ones, the macro FITIN_MONITOR_VARIABLE(var) becomes available.

Next, we take the original code and place the annotation FITIN_MONITOR_VARIABLE(state) right after the variable’s declaration in line 10; the resulting modified source code is shown in Fig. 8. Once all variables of interest have been annotated, the program needs to be recompiled. It is strongly recommended to disable compiler-side optimizations (GCC option -O0) so that the counted accesses at source-code level and internally are kept in sync.

If variables have been annotated that are declared outside of main, the user needs to provide information about the corresponding function scope so that FITIn can identify these sections. One can either specify that all annotations are in a single function or supply the source code directory. Furthermore, since no two compilers behave the same, debugging symbols should always be included into the resulting binary (GCC option -g). Otherwise, the underlying Valgrind is unable to identify procedures at run time. When the compilation process has finished, the resulting program, elevator, can be executed standalone, without FITIn, just as any other C program; in fact, we recommend such test run as a sanity check. For starting the program under FITIn, however, the program elevator has to be passed as a parameter to Valgrind, and at the same time, FITIn has to be specified as the tool that Valgrind employs. Our workflow looks like the following:

```
 gcc -g -O0 -Iinclude/valgrind main.c -o elevator
 bin/valgrind --tool=fitin --golden-run=yes ./elevator
```

We have set --golden-run=yes as the golden run will not touch any bits of the program. In a fault-injection campaign, the golden run is the baseline against which the tests are run. Further command-line options can be obtained

\(^1\)https://github.com/MarcelHB/valgrind-fitin/archive/csmr-wcre14.zip
program to return four reports and then mute for the rest of its life. After adjusting the parameters, we notice that our elevator will simply pause that second and continues to operate normally! The reason is that \texttt{state} is apparently reloaded from memory each time. As we perform the bit flip on a temporary representation, the flip does not persist and vanishes in the next iteration. For this reason, we have to add the parameter \texttt{--persist-flip=yes} to the command line. Now, the bit flip successfully induces a live lock—the output stops and we have to terminate the program. Compared to the first experiment, the last line of Fig. 10 proves that there has been much more activity after the last output. If working under less deterministic circumstances, we can tell FITIn to stop the execution on exceeding a specified limit. In this experiment, we could terminate the elevator prematurely after 105000 instructions by setting \texttt{--inst-limit=105000}.

\begin{Verbatim}
$ bin/valgrind --tool=fitin --mod-bit=1 --mod-load-time=22 --persist-flip=yes ./elevator

==3312== FITIn, A simple fault injection tool
==3312== Copyright (C) 2013, and GNU GPL’d, by Clemens Terasa, Marcel Heing-Becker
==3312== Using Valgrind-3.8.1 and LibVEX;
   rerun with -h for copyright info

==3312== Command: ./elevator
==3312== System operating.
==3312== Elevator overloaded.
==3312== System operating.
==3312== System operating.
==3312== Final report:

FITIn Totals of monitored code blocks:
FITIn Overall variable accesses: 22
FITIn Monitored variable accesses: 22
FITIn Instructions executed: 113375

Fig. 9: False alarm for overload

Fig. 10: Invalid live-lock

Fig. 8: Annotation of the example

by using the command-line option \texttt{--help}.

Now, we have to count the number of accesses to \texttt{state} to schedule our first experiment: triggering one false alarm for overload. Each loop-run implies five read operations (lines 12, 15, 19 or 21, 23, 25). We select the third iteration and need to flip the least significant bit right before the 14th access. We prepare the run as shown by the first line of Fig. 9. The output also demonstrates the success of the fault injection as we notice one message \texttt{Elevator overloaded.}. In a golden run, for example, one can count the number of accesses to monitored variables to spot significant mismatches in the number of expected and observed read accesses. As, under normal conditions, the program runs indefinitely, we have to terminate it by Ctrl-C.

At the end, FITIn prints additional statistics to the output: the overall number of counted variable accesses resulting from load operations from the memory, the accesses to variables that have been annotated by the user until the bit flip took place, and the overall number of instructions executed, excluding the ones of Valgrind, FITIn, and all function calls the outside of the intermediate representation.

In the next experiment, we attempt to put the program into an invalid live-loop state: We select the fourth iteration to be the last one correctly working and we will perform a bit flip before the read in line 15, which is the 22th access to \texttt{state}; we flip the second bit. We expect the

\begin{Verbatim}
$ bin/valgrind --tool=fitin --mod-bit=0 --mod-load-time=14 ./elevator

==3299== FITIn, A simple fault injection tool
==3299== Copyright (C) 2013, and GNU GPL’d, by Clemens Terasa, Marcel Heing-Becker
==3299== Using Valgrind-3.8.1 and LibVEX;
   rerun with -h for copyright info

==3299== Command: ./elevator
==3299== System operating.
==3299== System operating.
==3299== System operating.
==3299== System operating.

FITIn Totals of monitored code blocks:
FITIn Overall variable accesses: 170
FITIn Monitored variable accesses: 22
FITIn Instructions executed: 113375

Fig. 10: Invalid live-lock

```c
#include <fi_client.h>
#define POWER 4
#define OPERATING 2
#define OVERLOAD

int get_weight() { return ...
[FITIn] Monitored variable accesses: 22
[FITIn] Instructions executed: 113375
Fig. 10: Invalid live-lock
```