Michał MOSDORF, Janusz SOSNOWSKI
INSTITUTE OF COMPUTER SCIENCE, WARSAW UNIVERSITY OF TECHNOLOGY,
ul. Nowowiejska 15/19, 00-665 Warsaw

Fault injection in embedded systems using GNU Debugger

Mgr inż. Michał MOSDORF

PHD student at Institute of Computer Science of Faculty of Electronics and Information Technology. Graduate of Computer Science at Faculty of Electronics and Information Technology (2009). Conducts research in field of software reliability in embedded systems environment.

e-mail: m.mosdorf@ii.pw.edu.pl

Prof. dr hab. inż. Janusz SOSNOWSKI

Graduated from the Faculty of Electronics and Information Technology at Warsaw University of Technology. Received professor title in 2006. Currently employed in professor position at Computer Science Institute of Warsaw University of Technology. He is author and coauthor of 200 publications. His scientific interests concern computer systems reliability, architecture and interfaces design.

e-mail: jss@ii.pw.edu.pl

Abstract

The paper presents the technique of simulating faults in embedded systems. It consists of PC software that performs fault injection through the JTAG interface controlled by GNU Debugger (GDB) server for a chosen platform. This approach can be easily adopted to various platforms due to a wide support of GDB project for many architectures. The experimental results for ARM architecture show high controllability of the fault injection process and measured time overhead in the implemented injector.

Keywords: fault injection, JTAG interface, GDB project, microcontrollers.

Symulacja błędów w systemach wbudowanych z wykorzystaniem GDB

Streszczenie

Praca przedstawia technikę symulacji błędów dla systemów wbudowanych wykorzystującą interfejs JTAG sterowany za pomocą oprogramowania „GNU Debugger" przygotowanego dla danej platformy mikroprocesorowej. Opracowana architektura symulatora błędów została przedstawiona na rys. 1. Zaprezentowane rozwiązanie umożliwia symulację błędów typu bit-flip oraz błędów trwałych za pomocą mechanizmów breakpoint oraz watchpoint. Obserwacja wyników symulacji została zrealizowana za pomocą programowego mechanizmu breakpoint. W ramach pracy zweryfikowano koncepcję dla współczesnych mikroprocesorów z rdzeniem ARM7TDMI oraz zaprezentowano rezultaty symulacji błędów dla wybranych obszarów pamięci SRAM oraz rejestru PC procesora. Podejście to może być łatwo dostosowane do różnych platform systemów wbudowanych wspieranych przez projekt GDB. Przeprowadzone eksperymenty symulacyjne potwierdziły ich dużą sterowalność. W pracy przedyskutowano również efektywność opracowanej metody symulacji błędów oraz przedstawiono wyniki pomiarów opóźnień związanych z symulacją błędów oraz obserwacją wykonywania programu wynoszące odpowiednio 52ms i 42ms.

Słowa kluczowe: symulacja błędów, interfejs JTAG, projekt GDB, mikrokontrolery.

1. Introduction

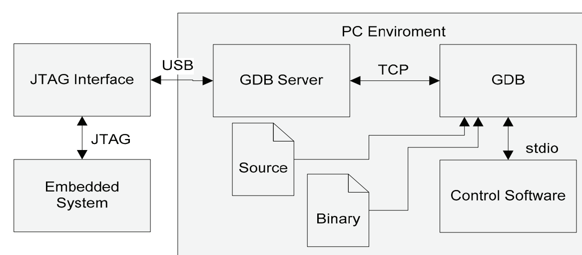
Fault injection is a useful tool for evaluating system dependability in the presence of faults. It provides the possibility of simulating system faults, tracking fault propagation, examining system susceptibility to faults and checking effectiveness of error detection and correction mechanisms. Literature distinguishes 2 main approaches to implementing fault injection [8]. Hardware implemented fault injection (HWIFI) performs fault injection by means of special hardware equipment and physical processes like a laser beam or electromagnetic field. Software implemented fault injection (SWIFI) emulates faults by means of various debugging mechanisms (e.g. software debuggers, JTAG or NEXUS interfaces) in real or in modeled systems [1, 2, 6, 7].

In this paper the authors focus on SWIFI technique performed in the environment of embedded systems. Limited resources in these systems create some implementation problems with SWIFI. There are only a few reports in the literature, so further research and new experience are still needed in this issue. The authors particularly concentrate on very popular ARM7 microcontrollers family [10]. In the presented solution fault injection is performed through JTAG interface available in the ARM platform. Fault injection is controlled by PC software that interfaces with GNU Debugger (GDB) [5] for ARM devices. This approach can be widely adopted to different hardware platforms supported by GDB project. This usually requires some modifications within the processor JTAG interface and the controlling GDB server.

The paper is organized as follows: Section 2 outlines architecture of the designed fault injector, Section 3 presents experimental results related to embedded program testing in ARM7 microcontroller environment. The final conclusions and future work perspectives are presented in Section 5.

2. GDB-Based Fault Injector

The developed Fault Injector architecture is presented in Fig. 1. Direct access to the tested embedded system is realized by JTAG probe compatible with a given architecture. In our studies we used J-Link pro device manufactured by SEGGER company [11].



Rys. 1. Architektura systemu do symulacji błędów opartego na GDB
Fig. 1. Architecture of GDB-Based fault injector system

JTAG interface is controlled by GDB remote server compatible with the tested embedded system architecture. This part of software is usually prepared by a JTAG manufacturer. The GDB remote server provides a GDB compliant serial interface that is used by open source GNU Debugger for remote debugging realized through TCP connection. Fault injection in the tested software is realized by GDB that provides easy access to tested processor memories and registers. Fault injection is controlled by the developed application implemented in .Net called "Control Software". Control procedures use standard input and output methods provided by NET Process class that is used for communication with GDB. This software is responsible for triggering fault injection events, performing fault injection through GDB commands and checking fault injection effects. The presented system needs access to a binary file and source codes of the embedded program. GDB uses the source code to provide

debug symbol information. A binary code is used for programming the device during its initialization before a fault injection experiment. The microcontroller program download is initialized by Control Software commands.

Portability of this approach to fault injection in embedded systems is provided by a standard GDB interface used for debugging various architectures. At this point GNU Debugger project community provides support for many processor architectures including e.g. ARM, PowerPC or MIPS [5]. For other platforms than ARM, only the JTAG interface and a corresponding GDB server needs to be modified.

In the presented architecture fault injection can be triggered by three different schemes:.

- *Time trigger*: In this approach fault injection moment is selected by the Control Software. After trigger event Control Software suspends a processor by executing break instruction through GDB.
- *Breakpoint*: In this scheme controlled bit-flips can be simulated at deliberately chosen source code locations. This can be very useful when simulating faults in function local variables or stack memory. This approach requires setting software or hardware break point in the source code. Having reached the breakpoint, the processor stops and GDB passes appropriate information do Control Software.
- *Watchpoint*: In this approach fault injection moment is defined by programmed watch point for a chosen processor memory address. Watchpoints can be reported during read or writes from the specified memory address. Watchpoint triggers provide possibility of simulating permanent memory errors (Section 3).

Having reached a injection trigger, the tested program is suspended and the Control Software is activated. In case of the time trigger the program is stopped externally by GDB command. In case of breakpoint/watchpoint triggers the program stop is caused by processor internal debug mechanisms. The activated Control Software reads a selected memory cell and then writes memory with overridden value (injection of a fault e.g. single bit flip). Then the tested program is resumed by GDB command.

Tracing the effect of fault injection (observability) is provided by breakpoint/watchpoint mechanisms that allow monitoring of the program progress by checking states of selected variables. The number of available hardware breakpoints in the tested processor is limited (e.g. 2 hardware breakpoints for ARM7). This limitation is overcome by using software breakpoint mechanism provided by J-Link [11]. In case of more complex ARM processors observation of injection results can be archived by Embedded Trace Buffer (ETB) support in the used JTAG interface.

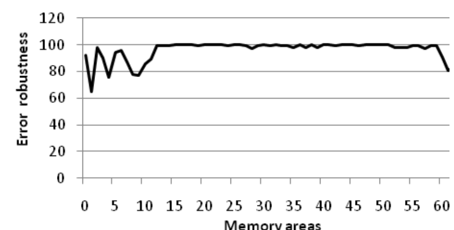
The idea of this fault injector was inspired by the drawbacks of our previous fault injector [12] which was based on OpenOCD and usbScarab2 JTAG interface. It provided only time trigger for generating fault injections. Moreover, the observation of test results needed some instrumentation of the tested application, it included trace messages sent via an additional RS232 interface. The presented fault injector is more universal, it does not need software instrumentation and additional result observation channel.

The developed fault injector requires suspension of the tested program (externally or internally) to inject faults or to observe their effects. This unfortunately introduces temporal overhead that may disturb real time character of the tested embedded system. The authors measured the time required for handling breakpoints used for observational purposes (reading one 32 bit variable) and breakpoints used for bit-flip injection. The time overhead related to handling these events was measured as an average for total 2258 events of each type. The Control Software registered the time of receiving breakpoint occurrence notification from GDB and the time of resuming processor work by GDB command. The average time required for handling the observational breakpoint that included reading a 32-bit variable was 42 ms. In case of fault injection that additionally required writing a32-bit variable the average time was 52 ms.

3. Experimental Results

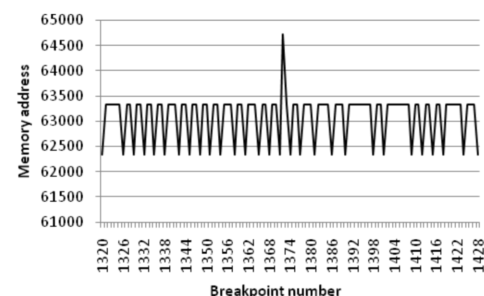
For the demonstration purpose there were performed tests on a real time embedded system based on an AT91SAM7S256 microcontroller (CPU clock – 48MHz, SRAM memory size 64KB). Software executed on the processor was responsible for performing a set of cryptographic algorithms based on elliptic curve cryptography (ECC) [4]**Błąd! Nie można odnaleźć źródła odwołania.**. These were: Diffie-Helman key exchange, Digital Signature Algorithm (signature and verification) and ElGamal (encryption and decryption). Implementation was based on the GMP arithmetic library [9]. Proper execution of each of three algorithms was self checked by the program. For the ARM platform, program binary size was 140KB, total static memory usage was 4KB. This software was developed in the scope of a project on biomedical data collection (described in [3])**Błąd! Nie można odnaleźć źródła odwołania.**.

In the performed experiments we were especially interested in checking the impact of faults with the microcontroller RAM memory. Having injected bit-flip errors with equal distribution within the whole RAM address space, we found that the most sensitive were memory cells within addresses 0-10k (static global and dynamically allocated variables) and 63k-64k (the program stack area holding function calls data and local variables). This is illustrated in Fig. 1 which shows the percentage of injected faults resulting in no operational error (practically 100% for most memory cells (addresses 16k – 64k) and 70 -90% within the most sensitive areas. This results show the memory static and dynamic usage profile within the tested application.



Rys. 2. Odporność na błędy pamięci RAM
Fig. 2. RAM memory fault robustness

The developed injector allows tracing system resource usage in more detail. For an illustration Fig. 3 shows the dynamic activity of the program stack (memory address stored in stack pointer during program execution). In that experiment we configured the Control Software to set breakpoints at the beginning of 4 basic functions used within the tested program. These functions were responsible for performing arithmetic operations on elliptic curve points (adding, doubling and multiplying) and printing point coordinate values through a serial port.



Rys. 3. Wybrane wartości wskaźnika stosu
Fig. 3. Selected values of stack pointer

During the experiment the Control Software handled 8952 breakpoints with the following distribution of functions: 2258 –

epoint_add, 6674 – epoint_dub, 14 – epoint_mul, 6 – epoint_print. After reaching each breakpoint the Control Software read and stored state of the stack pointer. Fig. 3 shows typical fluctuation of the stack pointer state during performed calculations. For ARM microcontrollers stack is built from higher to lower memory addresses.

The highest value of the stack pointer relates to invoking epoint_mul function (around point 1371). This function performs a series of calls to functions responsible for elliptic curve adding and doubling according to the ECC multiplication algorithm. So after point 1374 there are observed lower stack states (calls to epoint_add function) and middle states (calls to epoint_dub function). Similar fluctuations before point 1371 are related to the previous activation of epoint_mull function (not shown in Fig. 3).

To trace fault susceptibility we can target the tests at specific program points. Here we present the results of simulating faults in arguments (three 32-bit pointers) of the function responsible for performing elliptic curve point multiplication, this function declaration code is as follows:

```
void epoint_mul(eccpoint_t * c, eccpoint_t * a,
mpz_t b)
```

In the performed experiment the Control Software was programmed to inject faults using breakpoint trigger set at the function entry. During this experiment additional breakpoints were used for observing injection results such as: processor exceptions, watchdog resets and algorithm errors. For each of the three variables Control Software simulated 20 random bit-flip faults. Each fault injection was preceded with an external reboot of the system. After each fault injection the program was resumed and the injection results were collected. They are shown in Tab. 1.

Tab. 1. Wyniki symulacji błędów w argumentach funkcjo odpowiedzialnej za mnożenie punktu na krzywej eliptycznej

Tab. 1. Results of fault injection performed in arguments of function responsible for fault elliptic curve point multiplication

Variable	Data abort	Prefetch abort	Watchdog	Incorrect result
a	11	0	6	3
b	6	0	14	0
c	17	3	0	0

Most fault injectors deal with transient faults (bit flips, setting, resetting a memory cell). Simulating permanent faults creates more problems. The developed injector provides this capability. We illustrate this by simulating the permanent fault in one of the cryptographic constants – elliptic curve points additive group generator. In the tested implementation group generator was represented as the following structure:

```
typedef struct ECC_POINT
{
      mpz_t x;
      mpz_t y;
      mpz_t z;
      int identity;
}eccpoint_t;
```

Variables `x,y,z` represented elliptic curve point coordinates and variable `identity` was a marker providing information if the given elliptic curve point is a point of infinity (0 in additive group). During the experiment a permanent fault in variable identity was simulated as value 1 instead of 0. This was performed using the programmed write watchpoint trigger. The value of variable identity was overridden to 1 each time just after valid write operation of the program. This fault disturbed the three cryptographic algorithms and the error was reported by the self-checking procedure.

The developed fault injection system also provides possibility of simulating faults directly in processor registers. In particular, it is possible in this way to simulate program control flow faults by disturbing the program counter register (PC). For the analyzed program there were obtained the following distribution of the test results (PC disturbed with random distribution in time): Data abort – 45%; Prefetch abort – 20%; Undefined instruction – 15%; Incorrect result – 10%; Correct result – 10%.

4. Conclusions

The paper proves that JTAG interface supported with GDB provides the capability of implementing quite universal fault injector (useful in dependability evaluation) without the need of instrumenting the tested program within the embedded system. This creates the capability of performing injection experiments in real environment. The used GDB assures an easy access to processor registers and code variables by using symbol names. Other injectors usually base on physical addresses. Moreover, it provides efficient fault triggering mechanisms and facilitates injection of permanent faults.

The performed experiments with ECC algorithms on ARM7 platform confirmed high controllability and observability of injection processes. However, this approach introduces some time overhead (typically 40-50ms per simulated fault). In many applications this is negligible. For time critical applications the authors are working on a more efficient injector.

5. References

[1] Portela-García M., López-Ongil C., Valderas M.G., Entrena L.: Fault Injection in Modern Microprocessors Using On-Chip Debugging Infrastructures, IEEE Transactions on Dependable and Secure Computing, Volume 8 issue 2, ISSN: 1545-5971.

[2] Pedro Yuste, Juan Carlos Ruiz, Lenin Lemus and Pedro Gil: Non-intrusive Software-Implemented Fault Injection in Embedded Systems”, Lecture Notes in Computer Science, 2003, Volume 2847/2003, 23-38.

[3] Mosdorf M., Zabołotny W.: Implementation of elliptic curve cryptography for 8-bit and 32-bit embedded systems – time efficiency and power consumption analysis, ACS-AIBITS 2010, PAK vol. 56, nr 8/2010, Międzyzdroje 2010.

[4] Hankerson D., Menezes A. J., Vanstone S. A.: Guide to Elliptic Curve Cryptography, Springer, 2004.

[5] Richard Stallman, Roland Pesch, Stan Shebs, et al.: Debugging with GDB, 2010-10-16.

[6] Ar lat J. , et al .: Comparison of Physical and Software-Implemented Fault Injection Techniques, IEEE Trans. on Computers, vol. 52, no.9, pp. 1115-1133, 2003.

[7] Benso A., Prinetto P.: Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation. Kluwer Academic Publishers. (2003).

[8] Peter Folkesson, Sven Svensson, Johan Karlsson: A Comparison of Injection Based and Scan Chain Implemented Fault Injection. ftcs, pp.284, The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing, 1998.

[9] The GMP team: GNU MP, The GNU Multiple Precision Arithmetic Library, Edition 4.2.4, September 18, 2008.

[10] Andrew N. Sloss, Dominic Symes, Chris Wright: ARM System Developer’s Guide, Design and Optymizing System Software, Elsevier, 2004.

[11] Segger Microcontroller GmbH & Co. KG, J-Link/J-Trace ARM, User Guide of the JTAG emulators for ARM Cores. SEGGER web page: www.segger.com, 2010.

[12] Mosdorf M., Grochowski K., Gawkowski P., Sosnowski J.: Simulating Faults In Computer Systems, ISAT 2010, Information Systems Architecture and Technology – New Developments in Web-Age Information Systems, ISBN 978-83-7493-541-8, Wrocław 2010.

otrzymano / received: 13.05.2011
przyjęto do druku / accepted: 04.07.2011 *artykuł recenzowany*