Characterization of stack behavior under soft errors

Junchi Ma, Yun Wang

School of Computer Science and Engineering, Southeast University, Nanjing 211189, China
Key Laboratory of Computer Network and Information Integration, Ministry of Education

***Abstract*—As process technology scales, electronic devices become more susceptible to soft error induced by radiation. The stack in the memory implements procedure calls and its behavior under soft error has not been studied yet. To analyze the effects of soft error on the stack behavior, we conduct a series of fault injection experiment in the IA-32 instruction set architecture. The injection targets are the *ESP* register (used as the stack pointer) and the *EBP* register (used as the stack-frame base pointer). We obtain a few important observations from the fault injection experiment. Results show that injections on *ESP* lead to silent data corruption (SDC) or benign only if the flipped *ESP* points to another return address when executing the *RET* instruction, otherwise most of the injections cause crash. The injected bits of these SDC and benign cases are distributed in the particular bits (4-7) and the reason for the distribution is given. Moreover, flipped *EBP* may cause a series of infinite return operations, which is defined as return cycle. We describe the basic mechanism of return cycle and the essential condition for its occurrence.**

I. Introduction

With aggressive shrinking of transistor dimensions, soft error is an increasing threat to hardware reliability [1][2][3]. Conventional hardware-only solutions such as hardware redundancy are no longer feasible due to power constraints [4]. As a result, researchers have explored resilient software to enable programs to execute on unreliable hardware. The most general scheme is software-based instruction duplication [5][6]. To design and deploy software-level resilience schemes, one needs to understand the effects of hardware faults on software and give configurable error mitigation solutions.

The execution of a program is composed of a series of calls to procedures. Since calls affect the data flow and the control flow, they are important to the correctness of the program under soft errors. Calls are usually implemented by using stack. Characterizing the stack's behavior is necessary to understand the effects of hardware faults on the software. The objective of this study is to understand how the stack responds to soft errors. To the best of our knowledge, the stack has not been characterized in prior work.

Due to calls, the stack is divided into frames and each stack frame contains local variables, parameters to be passed to another procedure, and procedure linking information. Throughout the paper we make use of the IA-32 instruction set architecture [7]. The processor provides two pointers for stack operations: the stack pointer and the stack-frame base pointer. The stack pointer (contained in the *ESP* register) serves as an indirect memory operand pointing to the top of the stack at any time. The stack-frame base pointer (contained in the *EBP* register) points to the bottom of the current stack frame and it is often used to reference all the procedure parameters and local variables in the current stack frame.

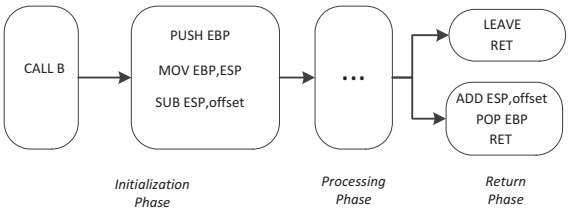

Fig. 1. A snippet of typical assembly code of a call to procedure

In this paper, we seek to characterize the stack behaviors by using fault injections. Since the two pointers represent the state of the stack, their corresponding architectural registers, the *ESP* register and the *EBP* register, are chosen as the targets of injections. We obtain certain interesting observations by concluding the results of fault injections. The major observations include:

- Flipped *ESP* may lead to silent data corruption (SDC) and benign only if the flipped *ESP* points to another return address when executing *RET* instruction, otherwise most of the injections lead to crash. Moreover, only faults in the particular bits (4-7) of *ESP* cause SDC or benign and we explain the bit distribution.
- Flipped *EBP* may cause a return cycle, meaning the return operations of a series of procedures are repeated indefinitely. We describe how a return cycle is produced and give the essential condition of the occurrence of a return cycle.

SDC occurs without any symptoms thus it is hard to detect. While most of erroneous *ESP* and *EBP* cause crash, with this approach we can find the particular *ESP* and *EBP* that cause SDC. The observations provide a vulnerability profile of *ESP* and *EBP* and identify the SDC-causing bits.

II. Preliminaries

In this section, we introduce the entire process of a call to procedure to understand the functionality of *EBP* and *ESP*. A snippet of typical assembly code of a call is shown in Fig.1. We list the essential instructions in a calling operation and explain the effects of the instructions on the stack.

Assume procedure A calls procedure B. The transfer of invoking starts by using a *CALL* instruction. *CALL B* pushes the return address of procedure A to the stack and loads the starting address of procedure B in the *EIP* register. The return

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

address points to the instruction where execution of procedure A should resume following a return from procedure B.

The execution of procedure B is divided into three phases, namely, initialization phase, processing phase and return phase. The initialization phase starts by using *PUSH EBP* to save old frame pointer. Then it initializes *EBP* and *ESP* for the stack frame of procedure B. *MOV EBP,ESP* copies the contents of *ESP* into *EBP*. *SUB ESP,offset* reserves place for local uses. Then procedure B enters the processing phase, which is the body of a procedure defined by the programmer. The processing phase does not interfere with the analysis, so it is omitted.

When the processing phase is over, the return phase starts. *EBP* and *ESP* have to be restored to match the state of procedure A. There are mainly two ways to restore *EBP* and *ESP*. The first way is to use *LEAVE* instruction, we call it L-way. The effect of *LEAVE* equals executing *MOV ESP,EBP* and *POP EBP*. *MOV ESP,EBP* restores *ESP* and *POP EBP* restores *EBP*. The other way is to use *ADD ESP,offset* and *POP EBP*, we call it A-way. *ADD ESP,offset* frees space allocated for procedure B and thus *ESP* is restored.

In the end, *RET* loads the return address of procedure A and returns. *EIP* is set to the return address of procedure A and procedure A resumes execution.

III. Experimental Methodology

The fault model we assume is a single bit flip within the *ESP* register or the *EBP* register. We use an injection map to guide the injection. Each entry of the injection map includes the instruction identifier, the register identifier (*ESP* or *EBP*) and the bit. The injection map is generated by analyzing the trace of fault-free execution. Once an instruction is found to access *ESP* or *EBP*, 32 entries with the instruction identifier are inserted into the injection map. Each bit of *ESP* or *EBP* (32bits) owns an entry to characterize the effect of bit on the propagation. In each injection run, an entry is loaded and the target bit of the register identifier is injected after the execution of the instruction with the instruction identifier.

In our experiment, we employ the dynamic instrument tool Pin for fault injection [8]. Pin is a dynamic binary instrumentation framework for the IA-32 and x86-64 instruction-set architectures that enables the creation of dynamic program analysis tools. We inject faults after the instruction is executed and thus the injection point is set to IPOINT_AFTER. The trace of executed instruction is recorded for further investigation. The item of trace includes the sequence number of instructions, the value of operands, etc. The trace files take up 35GB of disk space in total.

The experiment is conducted on Dell Precision T1700 Workstation with Intel i7 processor running Ubuntu10.04, and GCC of the version 4.4.3 is used for compilation.

The benchmark studied here is from the Siemens suite of programs [9]. The Siemens program considered is schedule, which is priority schedulers. The output is restricted to the result that is printed to the screen.

TABLE I
Percentage of each type

	Crash	Benign	SDC	Hang
ESP	80.2	15.5	4.0	0.3
EBP	83.3	14.4	1.5	0.8
RET-control *ESP*	98.5	0.9	0.5	0.1
Non-*RET*-control *ESP*	69.9	23.7	5.9	0.5
RET-control *EBP*	98.3	0.2	0.1	1.4
Non-*RET*-control *EBP*	66.1	30.7	3.1	0.1

The injection result is categorized into four types [4]: (1) benign, meaning that the program gets the right output, (2) crash, which means the error causes the program to stop execution, (3) hang, which means resource is exhausted but the program still cannot finish execution, (4) SDC, which means the program generates erroneous output. The output is compared with that of the fault-free execution, SDC happens if any divergence is found, thus soft computing [10] is not considered in this paper.

The results of ESP and EBP are discussed separately in the following two sections.

IV. The injection results of *ESP*

In this section we discuss the injection results of *ESP*. The percentage of each type is shown in the Table.I. The percentage of crash reaches 80.2% and the main reason for the high rate of crash is found by analyzing the traces. It shows that 55.5% of injections end execution after *RET* instruction and cause crash, thus the loading of return address is seldom accomplished with an erroneous *ESP*. We then analyze how the loading of return address is affected by *ESP*.

A. The effect of ESP

We apply data dependence graph to analyze how *ESP* affects the loading of return address. The data dependence graph of L-way restoring and A-way restoring is shown in Fig.2. The line starting from a box with the site name denotes the value stored in the site. Black node represents the value written to a site and white node represents the value read from a site. The edge denotes the dependence between nodes. The white node depends on the value stored in the corresponding site and the address of the site if there is one. The value of site after a write operation depends on the black node. Moreover, the value produced by an instruction depends on some source operands of the instruction. We use a tuple $(i, u, \circ \setminus \bullet)$ to represent the node in the Fig.2, where i denotes the instruction and u denotes the site that is read($\circ$) or written($\bullet$) by i. For example, $(\{RET\}, return\ address, \circ)$ denotes the value read from return address by *RET*. $(\{RET\}, return\ address, \circ)$ depends on $(\{RET\}, ESP, \circ)$ and $(\{CALL\ B\}, return\ address, \bullet)$. $(\{RET\}, ESP, \circ)$ is the address of return address and $(\{CALL\ B\}, return\ address, \bullet)$ determines the value of return address stored on the stack.

By using the graph, we can find how $(\{RET\}, return\ address, \circ)$ is affected by *ESP* and *EBP*. Red lines are used to mark the

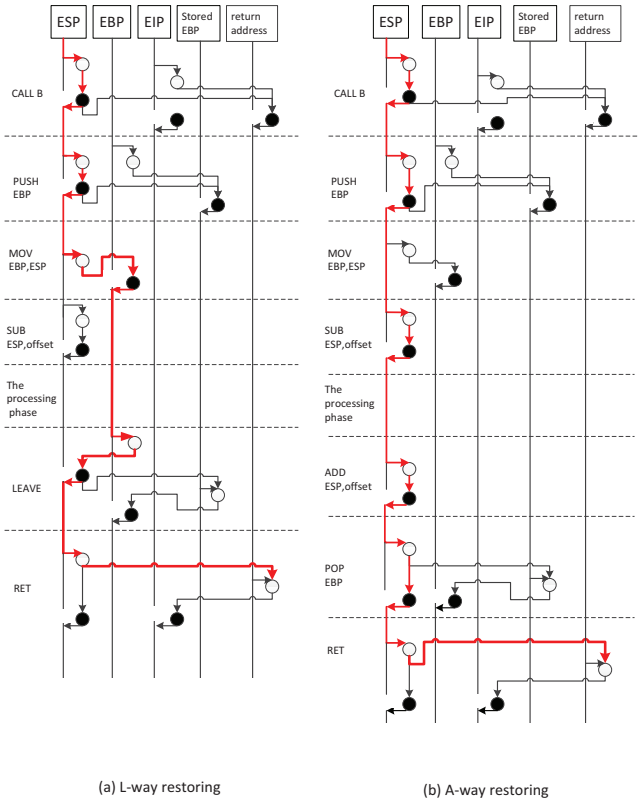

Fig. 2. The data dependence graph corresponding to the code of a call to procedure B

dependence path. In the A-way restoring, we can find $(\{RET\}, return\ \ address, \circ)$ depends on $(\{POP\ EBP\}, ESP, \bullet)$. Further, $(\{POP\ EBP\}, ESP, \bullet)$ depends on $(\{ADD\ \ ESP, offset\}, ESP, \bullet)$ that writes *ESP* prior to *POP EBP*. Tracing back like this, we find the dependence chain begins in $(\{CALL\ B\}, ESP, \bullet)$.

The L-way restoring incurs a varied propagation. $(\{RET\}, return\ \ address, \circ)$ depends on $(\{LEAVE\}, ESP, \bullet)$. $(\{LEAVE\}, ESP, \bullet)$ then depends on $(\{LEAVE\}, EBP, \circ)$ since the *LEAVE* instruction copies the content of *EBP* to *ESP*. *EBP* dominates the propagation until the node of $(\{MOV\ EBP, ESP\}, ESP, \circ)$.

To conclude, in A-way restoring, $(\{RET\}, return\ address, \circ)$ depends on *ESP* only, and in L-way restoring, $(\{RET\}, return\ address, \circ)$ is affected by both *ESP* and *EBP*. *ESP* affects $(\{RET\}, return\ address, \circ)$ during the execution from *CALL B* to *MOV EBP,ESP* and from *LEAVE* to *RET*. *EBP* affects $(\{RET\}, return\ address, \circ)$ during the execution from *MOV EBP,ESP* to *LEAVE*.

To characterize the effect on return address, we categorize *ESP* into two portions, *RET*-control and non-*RET*-control, based on whether it affects $(\{RET\}, return\ address, \circ)$. The categorized results are shown in Table.I. The injection results reveal that most injections on *RET*-control *ESP* lead to crash, and the ratio of crash (98.5%) is much higher than that of injections on non-*RET*-control *ESP* (69.9%). It can be inferred that the high rate of crash is due to the dysfunction of locating return address for called procedure.

B. Source of SDC and benign

We further analyze the portion of *RET*-control *ESP* and try to explain the SDC and benign cases. r' is used to represent the return address read by *RET*. We categorize the injection cases into following types based on r'.

- The execution never reaches *RET* of the injected procedure. 51.0% of the results belong to the category and all of them lead to crash.
- r' is not allocated. 33.9% of the results belong to the category and all of them cause segmentation errors.
- r' points to a non-executable address. 4.5% of the results falls in the category and all of them belong to crash.
- r' points to an executable address, we further divide the cases based on whether r' is a pushed return address.

  When r' is not a return address, r' can be an intermediate variable allocated by the compiler and all of the results belong to crash.

  When r' is a return address, the results show that SDC takes up 17.7% and benign takes up 29.7%, thus we obtain a observation.

Observation 1: The results of injecting *RET*-control *ESP* show it leads to benign or SDC only if r' is a return address.

We then explain the observation. First of all, if neither *ESP* nor *EBP* matches the stack frame of the running procedure, it can easily break the stack and cause crash. Common operations of a procedure like reading a local variable or returning always need to access the stack by using *EBP* or *ESP*. Using an unmatched *EBP* to access local variables can easily cause segmentation error. Besides, an improper return address will be obtained due to a pair of unmatched *ESP* and *EBP* (no matter the procedure applies A-way or L-way restoring), then it is very likely to cause crash.

A return address lies at the bottom of a stack frame and next to the top of another stack frame. If *ESP* doesn't points to a return address when executing *RET*, then after *RET*, *ESP* is incremented and it should not point to top of stack frame of any procedures. Moreover, when executing *POP EBP*, the top-of-stack value can be any variable, thus it can be considered that *EBP* is set to a random value. Neither *EBP* nor *ESP* matches the stack of the running procedure after *RET*, as we discussed before, it can easily cause crash.

We take an example to show how SDC or benign is caused, which is shown in Fig.3. We assume that there are four procedures, procedure A, B, C, D. Procedure A called procedure B, procedure B called procedure C and procedure C called procedure D. The procedure D finished the execution and returned, and the procedure C is running now. The procedure C will read the return address in $[ESP]$ to return to procedure B. The another two legal return addresses on the stack are marked with ESP' and ESP''. $[ESP']$ stores the return address of procedure A and $[ESP'']$ stores the return address of procedure C. The procedure D returned but we assume $[ESP'']$ has not

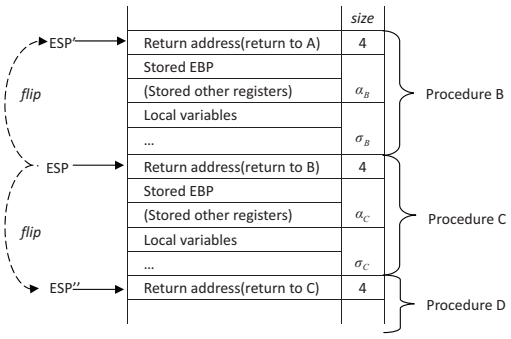

Fig. 3. The stack of SDC or benign cases when executing *RET*

been refreshed, so the return address of procedure C is still available.

Therefore there are two possible flips that cause *ESP* to point to another return address in this example. If *ESP* is changed into *ESP′*, it will read the return address of procedure A and return to procedure A; if changed into *ESP″*, it will read the return address of procedure C and return to procedure C.

The erroneous return of the example bears a resemblance to the buffer overflow [11]. The buffer overflow changes the return address to point to the attack code. When the procedure returns, instead of jumping back to where it was called from, it jumps to the attack code. The difference is that the return address in the example is not overwritten but the erroneous *ESP* cause the program to read another return address.

C. The flipped bit distribution of SDC and benign cases

We obtain another observation when analyzing the flipped bit distribution of SDC and benign cases.

Observation 2: The experiment shows that SDC or benign occurs only when injecting the bits (4-7) of *RET*-control *ESP*. None of the injections on lower bits (0-3) lead to SDC or benign.

The distribution of the flipped bits is explained here. Because of the fault model assumed in this paper, only one bit is flipped. The flip satisfies the condition below when *ESP* is changed to *ESP′* in Fig.3. We use α_B to denote the size of the stored registers of procedure B and σ_B to denote the size of allocated space of procedure B, which is equal to the *offset* in the instruction of *sub ESP,offset* of procedure B.

$$ESP' - ESP = \sigma_B + \alpha_B + 4 = 2^k, \tag{1}$$

where $k = 0, 1, 2, ..., 31$. Fig.3 can explain Eq.1 in a direct way. An return address takes up 4 bytes in the memory. In Eq.1, 4 denotes the size of the return address of procedure B. σ_B and α_B denotes the size of allocated space and the stored registers of procedure B separately, thus $\sigma_B + \alpha_B + 4$ equals the distance between *ESP* and *ESP′*. Since *ESP′* $>$ *ESP*, the flip must be from 0 to 1.

Similarly, we can get the equation for the flip that changes *ESP* to *ESP″*. Different from the flip that changes *ESP* to *ESP′*, it must be from 1 to 0.

$$ESP - ESP'' = \sigma_C + \alpha_C + 4 = 2^k. \tag{2}$$

In the proposed example, the stack frame of procedure B is between *ESP* and *ESP′*. Further, there can be more stack frames between *ESP* and *ESP′*, thus we gets Eq.3.

$$ESP' - ESP = \Sigma_{j=0}^{bet_n}(\sigma_{p_j} + \alpha_{p_j} + 4) = 2^k, \tag{3}$$

where bet_n represents the number of the procedures that between *ESP* and *ESP′*. When $bet_n = 1$, we gets the lower bound of k. Because *EBP* is always stored on the stack and it takes up 4 bytes, $\therefore \forall procedure\ p_j, \alpha_{p_j} \geq 4$. $\because \sigma_{p_j} \geq 0 \wedge \alpha_{p_j} + 4 \geq 8, \sigma_{p_j} + \alpha_{p_j} + 4 \geq 8 \therefore k \geq 3$. A procedure seldom has no allocated space, thus if $\sigma_{p_j} > 0$, then $\sigma_{p_j} + \alpha_{p_j} + 4 > 8 \therefore k > 3$, which explains the minimum k observed in the experiment is 4.

The maximum stack size is 8MB for our platform and for most 32-bit Linux systems, thus *ESP′* $-$ *ESP* must be less than 8MB (2^{23}). *ESP′* $-$ *ESP* $= 2^k < 2^{23}$, $\therefore k \leq 22$. However, the stack used is always much smaller than 8MB. Besides, when k is getting larger, it is harder to satisfy Eq.3 since stack of that size ($> 2^k$ bytes) is rare. The maximum k observed in the experiment is 7.

Thus the flipped bit bound ($3 \leq k \leq 22$) is obtained, and the experimental results satisfy the bound.

V. The injection results of *EBP*

We also divide the results of *EBP* into two portions, based on whether it affects return addresses. In A-way restoring, all instances of *EBP* are non-*RET*-control, and in L-way restoring, *EBP* is *RET*-control during the execution from *MOV EBP,ESP* to *LEAVE*. The categorized results are also shown in Table.I. The injection results reveal that most injections on *RET*-control *EBP* lead to crash, and the ratio of crash (98.3%) is much higher than that of injections on non-*RET*-control *EBP* (66.1%). The percentage of crash is very close to *ESP*'s percentage. We then analyze the portion of *RET*-control *EBP*.

A. Hang caused by return cycles

The percentage of hang of injections on *RET*-control *EBP* is much higher than that of injections on non-*RET*-control *EBP*, reaching 1.4%. After investigating the hang cases, we obtain the following observation.

Observation 3: Flipped *EBP* can make the return operations of a series of procedures repeated indefinitely.

We call this situation *return cycle* and take an example to show how a return cycle is formed.

Assume procedure A calls procedure B. The stack of the example is shown in Fig.4. *EBP* is flipped to *EBP′* before procedure A calls procedure B and *EBP′* accidently points to the location that will store the *EBP′* in the stack frame of procedure B. After executing *PUSH EBP*, *EBP′* is stored in the stack frame of procedure B and thus $[EBP'] = EBP'$.

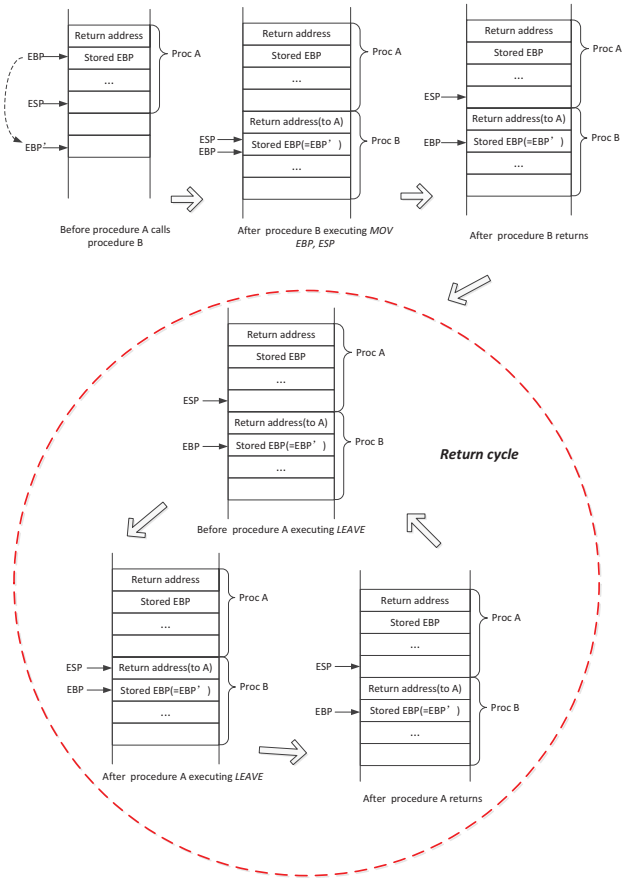

Fig. 4. The stacks when a return cycle is produced

After executing *MOV EBP,ESP*, *EBP* is set to *ESP* (the value of *EBP* doesn't change actually). *EBP* matches the stack frame of procedure B, therefore procedure B is not affected by the erroneous *EBP* during the processing phase. When procedure B returns, *EBP* is restored and thus $EBP = EBP'$, then the execution of procedure A is resumed.

Since procedure A applies L-way restoring, restoring is affected by the erroneous *EBP*. Before *LEAVE*, $EBP = EBP'$, so *EBP* is restored to the value stored in $[EBP']$. After *LEAVE*, $EBP = EBP'$, $ESP = EBP' + 4$, thus the value of *EBP* remains the same. The *RET* instruction reads return address from $[EBP' + 4]$, so the return address of procedure A is read and it then returns to procedure A. When procedure A executes *LEAVE* and *RET* again, EBP' is restored and it still returns to procedure A. The return cycle of $A \rightarrow A \rightarrow A \cdots$ is formed, leading to hang.

The key point of forming a return cycle is the erroneous restoring and returning can be repeated. Without soft errors, *EBP* always moves to higher memory after the restoring of *EBP* since the current stack frame is reclaimed. However, in the example, after the restoring of *EBP* of procedure A, *EBP* doesn't move at all. The location storing *EBP* is not refreshed, therefore *EBP* is restored to EBP' every time and the restoring can be repeated. Besides, due to applying L-way restoring, *ESP* is also affected by *EBP* and thus the return address of procedure A is loaded when *RET* is executed, therefore the returning can be repeated.

In the proposed example, the return cycle only has one procedure, we also find the cases in the experiment whose return cycles contain two procedures. We take a case for explanation. Procedure A calls procedure B first and then procedure B calls procedure C. Before calling procedure B, *EBP* is flipped and points to the stack location that will store the *EBP* of procedure B in the stack frame of procedure C. When procedure B returns, the erroneous *EBP* is restored and then the execution of procedure A is resumed. When procedure A returns, the *EBP* of procedure B is restored and it returns to procedure B since the erroneous *EBP* points to the stored *EBP* of procedure B, and it forms a return cycle $B \rightarrow A \rightarrow B \rightarrow A \rightarrow B \rightarrow A \cdots$.

To conclude the return cycle cases studied, to cause a return cycle it need to satisfy three conditions.

- The flip occurs before a call, so an erroneous *EBP* is stored in the memory by the called procedure.
- The flipped *EBP* points to a pushed *EBP* which belongs to one of the procedures in the return cycle. For instance, the flipped *EBP* points to the pushed *EBP* of procedure A in the first example. Moreover, since the flipped *EBP* points to the stack frame of a called procedure, which is from a lower memory, the flip have to be from 1 to 0.
- None of the procedures crashes. If any of them crashes, the return cycle is broken. The procedure being injected is more likely to crash since *EBP* of its stack frame is corrupted while other procedures of the return cycle are not directly affected by the erroneous *EBP* and thus seldom crash.

Although we find return cycles which have two procedures at most, there could be more procedures in a return cycle if it satisfies the conditions.

B. Source of SDC and benign cases

It is found that SDC or benign occurs when r' is a pushed return address, which is the same as the observation obtained when we inject *ESP*. We conclude a new observation based on the value of *EBP*. The *LEAVE* instruction can be decomposed into *MOV ESP,EBP* and *POP EBP*. If *ESP* points to a return address before executing *RET*, *EBP* points to a pushed *EBP* before executing *LEAVE*.

Observation 4: For *RET*-control *EBP*, it leads to SDC or benign only if flipped *EBP* points to a pushed *EBP* before executing *LEAVE*.

Considering the conditions that cause return cycles, it can be found that the timing of injection impacts the results of injection. Assume the flipped *EBP* points to a pushed *EBP*, if the flip is injected before a call operation, the result is likely to be hang; otherwise the flip is likely to cause SDC or benign.

VI. Related Work

There has been a series of work which applies fault injection to characterize the vulnerability of programs. The researchers

focus on varied aspects and we haven't found any of them share the same goal with us.

CriticalFault [12] grouped the immediate effects into three types by examining the injection results: faulty control (affecting conditional branches), faulty address (affecting the memory address) and faulty data (the rest cases). A significant fraction of faulty control cases lead to benign due to Y-branches effect [13]. Faulty address cases show a different feature that a significant fraction of faulty address cases lead to crash. The effect of fault data is more complex than other types. If a fault in faulty data case propagates to memory address computations or control instructions, it would have a similar behavior as its consumers; if it propagates to data-only computation, it is likely to result in SDC.

Faults resulting in SDC produce corrupted program output without leaving any trace of failure behind, and thus are hard to detect. By applying injection tool Relyzer [14], Hari reveals that only a small fraction of static instructions cause most SDCs and these static instructions may belong to only a few procedures [15]. This observation can be also found in SDCTune [4]. Modeling the error propagation in a program, SDCTune concludes that the SDC proneness of a variable depends on the fault propagation in its data dependency chain. While the prior work focuses on identifying the portions of program or variables that affect the output, we find that disordered return behaviors can also affect the output, and thus our work complements the source of SDC.

Cook characterizes instruction-level derating [16], meaning that an instruction can compute using incorrect data and still produce correct results. Instruction-level derating is a significant reason to cause benign cases.

Gu provides an experimental study of Linux kernel behavior in the presence of errors [17]. Errors are injected to the instruction stream of selected kernel procedures so the injection may do damage to the system. Crashes may require reformatting the file system, while in other work most crashes just end the execution of an application.

VII. Conclusion

In this paper, we characterize the stack behaviors under soft errors. The pointers that the stack uses frequently, *ESP* and *EBP*, are targeted in the injection experiment. The results of *ESP* and *EBP* are categorized by whether they are related to the loading of return addresses. Many injections on *RET*-control *ESP* lead to crash since the *RET* instruction cannot load a proper return address. The SDC and benign cases are investigated and they have a common feature that the flipped *ESP* points to another return address when executing *RET*. An unexpected procedure continues executing instead of the caller procedure. Moreover, we find this only happens when injecting bits 4-7 due to the distance between the two return addresses.

We also get insight into the hang cases and find that many hang cases are caused by return cycle, meaning the return operations of a series of procedures are repeated indefinitely. We describe how return cycle is formed through certain examples and obtain the essential conditions for return cycle. Due to these observations, a parity bit or an SEC code needs to be used to protect ESP and EBP. Without hardened hardware, compiler-based replication is also feasible but it may bring more overhead.

The benchmark we study only applies the statement of *return* to end the execution of a procedure, but there may be other ways like applying the procedure of "exit". The C library procedure "void exit(int status)" terminates the calling process immediately. If the procedure of exit is applied, *RET* instruction is not the only way to return from a procedure, which makes it more complicated to analyze the control transfer. We will consider this situation in the future work.

References

[1] J. Walters, K. Zick, and M. French, "A practical characterization of a nasa spacecube application through fault emulation and laser testing," in *Dependable Systems and Networks (DSN)*. IEEE, 2013, pp. 1–8.

[2] S. Mittal and J. S. Vetter, "A survey of techniques for modeling and improving reliability of computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 4, pp. 1226–1238, April 2016.

[3] S. Mirkhani, S. Mitra, C. Y. Cher *et al.*, "Efficient soft error vulnerability estimation of complex designs," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2015, pp. 103–108.

[4] Q. Lu, K. Pattabiraman, M. S. Gupta *et al.*, "SDCTune: A model for predicting the SDC proneness of an application for configurable protection," in *Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, Oct 2014, pp. 1–10.

[5] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *Reliability, IEEE Transactions on*, vol. 51, no. 1, pp. 63–75, 2002.

[6] G. A. Reis, J. Chang, N. Vachharajani *et al.*, "SWIFT: Software implemented fault tolerance," in *Code Generation and Optimization (CGO)*. IEEE Computer Society, 2005, pp. 243–254.

[7] Intel, *The Intel®64 and IA-32 Architectures Software Developers Manual, Volume 1: Basic Architecture*, 2006.

[8] C.-K. Luk, R. Cohn, R. Muth *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[9] M. Hutchins, H. Foster, T. Goradia *et al.*, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *International Conference on Software Engineering (ICSE)*. IEEE Computer Society Press, 1994, pp. 191–200.

[10] A. Thomas and K. Pattabiraman, "Error detector placement for soft computation," in *Dependable Systems and Networks (DSN)*. IEEE, 2013, pp. 1–8.

[11] C. Cowan, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *USENIX Security Symposium, San Antonio, TX, USA, January 26-29*, 1998.

[12] M.-L. L. Xin Xu, "Understanding soft error propagation using efficient vulnerability-driven fault injection," in *Dependable Systems and Networks (DSN)*. IEEE, 2012, pp. 1–12.

[13] N. Wang, M. Fertig, and S. Patel, "Y-branches: when you come to a fork in the road, take it," in *Parallel Architectures and Compilation Techniques (PACT)*, Sept 2003, pp. 56–66.

[14] S. K. S. Hari, S. V. Adve, H. Naeimi *et al.*, "Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2012, pp. 123–134.

[15] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *Dependable Systems and Networks (DSN)*. IEEE, 2012, pp. 1–12.

[16] J. J. Cook and C. B. Zilles, "A characterization of instruction-level error derating and its implications for error detection," in *Dependable Systems and Networks (DSN)*. IEEE, 2008, pp. 482–491.

[17] W. Gu, Z. Kalbarczyk, R. K. Iyer *et al.*, "Characterization of linux kernel behavior under errors," in *Dependable Systems and Networks (DSN)*. IEEE, 2003, pp. 459–468.