6A-03

OS メモリダンプにおける関数引数解析機能に関する検討

高橋　香穂†　　伊藤　孝之†　　松浦　陽平†

†三菱電機株式会社　情報技術総合研究所

1　背景と目的

Linux には、カーネルがクラッシュした際に、レジスタやスタックの内容をメモリダンプとして保存する機能がある。例外発生によるシステム停止などの障害発生時には、メモリダンプを解析し、呼び出された関数やその引数を知ることが原因解明の手掛かりとなる。メモリダンプの解析には解析ツールが利用可能であるが、引数がレジスタを通して関数に渡される場合は、スタックを参照して値を知ることが出来ないため、機械的に解析できず、人手による解析を行う必要がある。

そこで、上記課題を解決するために、レジスタ渡し引数に対する機械的な解析方式について検討した。本稿ではその検討結果を述べる。

2　既存技術と課題

Linux のメモリダンプ解析ツールである crash コマンド[1]では、障害発生までに呼び出された関数を表示することが出来る。この際、引数を表示できるかどうかは、その渡し方によって異なる。渡し方は呼出規約によって決められており、スタック渡しとレジスタ渡しがある。

スタック渡しは、引数をスタックに積んで関数を呼び出し、引数を渡す方法である。この場合、図 1-(1)のように、障害発生時にも同じ場所に積まれたままであるため、解析ツールで機械的に解析することが可能である。

一方、レジスタ渡しは、引数をレジスタに格納して渡す方法であり、メモリアクセス回数を減らし処理速度の向上に貢献する。しかし、図 1-(2)のように、レジスタが処理に使用されることがあるため、スタック渡しと異なり引数が所定の場所になく、解析ツールを用いた機械的な解析が困難である。そのため、人手で解析する手間と時間がかかる。

これらの課題に対し、障害解析容易化のため、機械的にレジスタ渡しの引数を解析する機能を検討した。

Investigation of function argument analysis in a kernel memory dump.

† Information Technology R&D Center, Mitsubishi Electric Corporation.

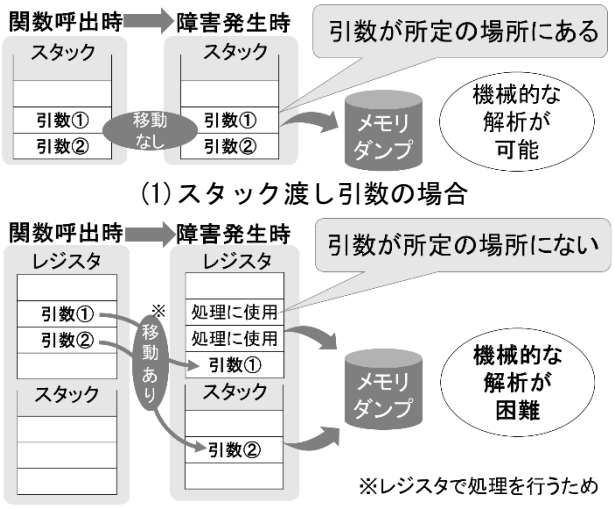

図 1 メモリダンプにおける引数格納場所

3　引数解析機能の方式案

引数解析機能の検討にあたり、(1)レジスタ値を書き出す方式、(2)命令コードをたどる方式、(3)デバッグ情報を用いる方式の 3 通りの方式案を挙げる。

(1)レジスタ値を書き出す方式

レジスタ渡し引数の場合、引数を格納したレジスタの内容が、関数内の処理により上書きされる可能性がある。これに対応するため、平常時に関数が呼ばれるたびに引数の値を書き出しておき、障害発生時にその値を取得する方式が考えられる。

例えば、レジスタ値をスタックに書き出すことで障害解析に利用する際の流れを図 2-(1)に示す。平常時において、関数が呼び出された際、引数格納レジスタ R に格納されて渡された引数を、スタック内の所定の位置に書き出す。スタックの内容は障害発生時にメモリダンプに書き出されるため、障害解析時にはこのスタックを参照して引数を取得する。

(2)命令コードをたどる方式

実行ファイルからカーネルモジュールの命令コードを取得することが可能である。これを活用し、命令コードをたどり、関数呼出時に渡さ

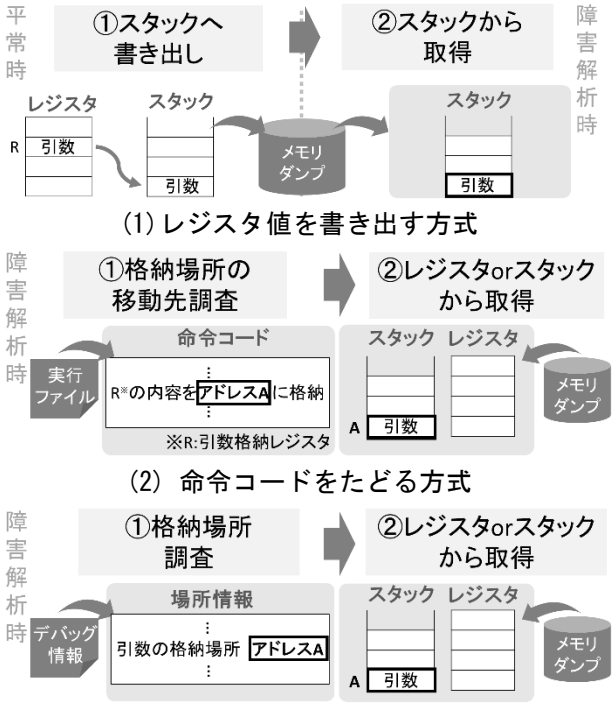

図 2　各方式案の概要

れた引数が障害発生時のレジスタやスタックのどこに格納されているのかを調べ、引数を解析する方式が考えられる。

具体的には、図 2-(2)のように、命令コードから、引数格納レジスタ R の内容がアドレス A に格納されたことを確認し、障害発生時の A に格納されている値を取得する。

なお、命令コードに分岐が含まれている場合は、どちらの分岐先が実行されたか不明であるため、引数を解析できない場合がある。すなわち、引数の移動や、移動先レジスタの上書きが実行されたか不明である場合は引数を解析できない。

(3) デバッグ情報を用いる方式

デバッグ情報とは、ソースコードと命令コードの対応を記したものである。GDB[2]では、デバッグ情報で引数の格納場所を調べ、アプリケーションの障害解析を行う。OS のメモリダンプにおいても、関数呼出時に渡された引数の障害発生時における格納場所をデバッグ情報から調べ、その値を取得する方法が考えられる。

具体的には、図 2-(3)のように、デバッグ情報で引数の格納場所がアドレス A であることを確認し、障害発生時の A の内容を取得する。

4　各方式の比較検討

前述の 3 通りの方式について比較検討を行った。本稿では、平常時の処理にオーバーヘッドを与えず、より多くの引数を解析可能である方式を採用する。

表 1 ケースごとの引数解析の比較

格納先	分岐	格納タイミング	方式(2)	方式(3)
なし	−	−	×	×
レジスタ	なし	−	○	○
	あり	分岐以前	×	○
		分岐以降	×	○
スタック	なし	−	○	○
	あり	分岐以前	○	○
		分岐以降	×	○

方式(1)は関数が呼ばれるごとに書き出し処理を行うため、オーバーヘッドを発生させる。方式(2)と(3)は平常時の処理に影響を与えないため、条件を満たす。

次に、前述の条件を満たした 2 通りの方式の内、どちらがより多くの引数を解析可能であるか比較した。格納先、分岐命令の有無、分岐命令がある場合の格納命令の実行タイミングによりケース分けし、解析可能か否かをまとめた結果を表 1 に示す。最適化された関数において、引数をその後の処理で使わない場合に、どこにも格納しないまま上書きすることがある。このように、引数がどこにも格納されていないケースは両方式共に解析不可能である。それ以外のケース、すなわちレジスタもしくはスタックに値が格納されている場合は、方式(3)では全て解析可能である。一方、方式(2)は、分岐命令がないケースと、分岐命令以前にスタックに格納されたケース以外は解析不可能である。これは、分岐命令以降に実行された命令が不明であるためである。なお、全ての引数が関数開始時にスタックに退避されている場合は、方式(2)も有効であると言える。しかし、最適化された関数は、関数開始時の引数退避処理を行わない。よって、方式(3)がより多くのケースにおいて引数を解析可能であると言える。

5　結論

本稿では、OS メモリダンプにおいて、レジスタ渡しの引数を機械的に解析するための方式を検討した。今後は、デバッグ情報を用いる引数解析機能の実現性を検証する。

参考文献

[1] Red Hat Software, Inc. “White Paper: Red Hat Crash Utility,” <https://people.redhat.com/anderson/> 2018-12-13 アクセス.

[2] Free Software Foundation, Inc. “GDB: The GNU Project Debugger,” <https://www.gnu.org/software/gdb/> 2018-12-13 アクセス.