Vol.2014-OS-130 No.15
2014/7/28

Linux へのソフトウェアフォールトインジェクションの有用性の調査

菊池 伸郎[1,a)]　吉村　剛[1]　佐久間　亮[1]　河野　健二[1]

概要：オペレーティングシステム (OS) には未だにバグが残されており, バグの実行によるクラッシュが避けられない. そのため OS のクラッシュを隠蔽する信頼性向上手法が提案されている. 信頼性向上手法の定量的評価には OS のクラッシュを統計的に有意な回数引き起こす必要がある. しかしテスト段階で OS のソースコードから発現しやすいバグは取り除かれており, クラッシュを起こすことは困難である. そこで信頼性向上手法の評価を簡単にするためにソフトウェアフォールトインジェクション (SFI) が用いられている. SFI は擬似的なバグをソースコードに挿入し，クラッシュを引き起こす手法である. SFI が引き起こすクラッシュは現実のクラッシュと近い必要がある. なぜならば信頼性向上手法は現実のクラッシュ を対象にして設計されており, 現実のクラッシュと SFI によるクラッシュに差があると結果が不当に悪くなるためである. しかし OS に対する SFI がどの程度現実に近いクラッシュを起こしているのかの評価が未だにされていない. 本研究では現実のクラッシュと SFI のクラッシュを比較することにより, SFI がどの程度現実に近いクラッシュを起こしているのか評価する. 調査により SFI では現実のクラッシュを再現しきれていないことを示した.

キーワード：ソフトウェアフォールトインジェクション, バグ, ディペンダビリティ

1. はじめに

オペレーティングシステム (OS) には高い信頼性が求められている．アプリケーションは OS の提供する機能を利用して動作するため，アプリケーションの信頼性が高くとも OS のクラッシュにより停止するためである．アプリケーションの停止はサービスの停止につながるため，多大な損害につながり，例えば仲買業のシステムの停止は 1 時間に約 6.4 億円の損害を出す [1].

しかし OS のソースコードには多くのバグがあることが知られており，Linux カーネルには比較的簡単な静的解析で見つかるバグですら約 700 個存在することが報告されている [2]. そのためソースコード中のバグによる OS のクラッシュは避けられない.

そこで OS のクラッシュを隠蔽し被害を最小限にするため，多くの信頼性向上手法が提案されている [3], [4], [5], [6], [7].

例えば Shadow Driver [3] という信頼性向上手法がある. Shadow Driver はデバイスドライバのクラッシュの検知・復旧を行い，デバイスドライバのクラッシュによって OS 全体のクラッシュが引き起こされないようにする．Shadow Driver のような信頼性向上手法は，どの程度バグに対して有効であるのかを定量的に評価する必要がある．このような定量的な評価を行うには，十分な回数 OS をクラッシュさせ，統計的に有意な結果を得る必要がある.

このような定量的な評価を容易にするためにソフトウェアフォールトインジェクション (SFI) という手法が知られている．SFI は擬似的なフォールトをコードに挿入する手法であり，クラッシュなどの障害を引き起こしやすくすることで信頼性向上手法の定量的評価を容易に行うことができるようにする技術である．SFI が引き起こすクラッシュは現実のクラッシュに近い必要がある．なぜなら，SFI によって引き起こされる障害が現実のソフトウェアシステムで発生する障害と大きく異なっていると，SFI を用いた評価結果そのものの信頼性が揺らぐこととなる．たとえば，Shadow Driver ではドライバのクラッシュには対応可能であるもののドライバのハングに大しては何も対応が取られていない．もし，SFI によってドライバのクラッシュばかりが引き起こされ，ドライバのハングが発生しなかったとすると，Shadow Driver は極めて高い確率で障害に対処できるという結果となる．現実には，ドライバがハングする

[1] 慶應義塾大学
Keio Uniersity
[a)] kikuchi@sslab.ics.keio.ac.jp

場合もあり，このような結果は望ましくない．

しかし SFI によるクラッシュが現実のクラッシュの性質に近いかが明らかになっていない．本稿ではでは現実のクラッシュログと SFI によるクラッシュログとの定量的な比較を行い，どの程度 SFI が現実に近いクラッシュを起こせるのかを評価する．Linux のファイルシステムに対し SFI を行って得られたクラッシュログと，RedHat 社が収集している Linux のクラッシュログのうち，ファイルシステムと関連したものとを比較する．SFI は SAFE [8] というフォールトインジェクタを用いる．OS に対し広く使用されているフォールトインジェクタ [9] は 1992 年の OS のバグの調査 [10] をもとに作成されているため，バグのモデルが古い．SAFE は 2006 年のバグの調査 [11] をもとに作成されているため，新しいモデルのバグを挿入可能な最先端のフォールトインジェクタである．なお，調査対象をファイルシステムとしたのは，文献 [2] において多くのバグが存在するサブシステムと指摘されているためである．

調査によって SFI によるクラッシュは現実のクラッシュとは性質が違うことが示された．現実のクラッシュログ約 5 万件のうち，ファイルシステムに関係のあるクラッシュログ約 300 件と，SFI によるクラッシュログ約 5,500 件を比較した．その結果クラッシュをする原因の比率が異なることや，クラッシュ時のコールトレースの状態が異なることがわかった．

本論文の構成を以下に示す．2 章では SFI に関する関連研究を紹介する．3 章では SFI 実験環境について説明する．4 章では現実のクラッシュログと SFI のクラッシュログとの比較を行う．5 章ではまとめと今後の課題を述べる．

2. 関連研究

SFI は実際のクラッシュを再現する必要がある．そのため SFI によるクラッシュを現実のクラッシュに近づける研究が行われている．本稿では SFI によるクラッシュを現実のクラッシュと比較し，解析することによって，フォールトインジェクタが現実のクラッシュを引き起こしているかを調査している．他の研究では異なるアプローチから，SFI の性能の調査や SFI の性能の向上手法を研究している．

文献 [11] では最先端のフォールトインジェクタである G-SWFIT を提案している．現実のバグの性質を再現すれば現実のクラッシュの性質を再現できるいうと考えから，まず現実のバグの性質を調査している．そして現実のバグの性質を調査することから新たなバグの分類指標を提案している．vim, bash, Linux カーネルなどの 12 種類の実用ソフトウェアの 668 件の修正ログを分析し，ODC[12] のバグの分類を拡張し新たなバグの分類を作成している．そして新たに作成した分類の中から，現実のソースコード中に存在頻度の高い 13 種類のバグを SFI で挿入すべきであると示し，これら 13 種類のバグを挿入するフォールトインジェクタ G-SWFIT を提案している．G-SWFIT は実行ファイルのバイナリを操作することによってバグを挿入する．バグ挿入のためのバイナリ解析からバイナリの変更までの一連の操作をオペレータと定義し，G-SWFIT が挿入する 13 種類のバグを挿入するためのオペレータを作成している．図 1 にオペレータの一覧を示す．[11] では G-SWFIT のオペレータが文献 [11] 中では，バグの分類定義の通りに G-SWFIT がバグを正確に挿入ができるかの調査をしており，高い精度で意図した箇所に正しくバグの挿入ができていると示している．

表 1 G-SWFIT のバグ挿入オペレータ

Operator	Fault type addressed
OMFC	MFC - Missing Function call
OMVIV	MVIV - Missing variable initialization using a value
OMVAV	MVAV - Missing variable assignment using a value
OMVAE	MVAE - Missing variable assignment using an expression
OMIA	MIA - Missing if construct around statements
OMIFS	MIFS - Missing If construct plus statements
OMIEB	MIEB - Missing if construct plus statements plus else before statements
OMLAC	MLAC - Missing "AND EXPR" in expression used as branch condition
OMLOC	MLOC - Missing "OR EXPR" in expression used as branch condition
OMLPA	MLPA - Missing small and localized part of the algorithm
OWVAV	WVAV - Wrong value assigned to variable
OWPFV	WPFV - Wrong variable used in parameter of function call
OWAEP	WAEP - Wrong arithmetic expression in parameter of a function call

文献 [8] ではバイナリレベルのフォールトインジェクタとソースコードレベルのフォールトインジェクタの性能比較を行っている．バイナリレベルのフォールトインジェクタはソースコードが入手できないサードパーティ製のソフトウェアに対して使用可能である．そのためバイナリレベルで SFI が行える方が利便性が高い．しかしコンパイル時に情報が失われる，最適化によってコードが変更されるといった問題からバイナリレベルでは SFI が正確にできない恐れがある．そこでバイナリレベルではソースコードレベルに対してどの程度 SFI の性能に差があるかの評価を行っている．そしてバイナリレベルの SFI はソースコードレベルの SFI に対して大きく性能が落ちることを示している．文献 [8] ではバイナリレベルフォールトインジェクタとして G-SWFIT を用いている．またソースコードレベルフォールトインジェクタとして，G-SWFIT と同じ種類のバグをソースコードレベルで挿入するようにデザインされている SAFE を用いている．本稿ではフォールトインジェクタがどの程度現実クラッシュの性質を再現できるかを調査する．そのため，最も正確に現実のバグを再現できる最先端のフォールトインジェクタに対し調査を行う．バイナリレベルの G-SWFIT よりもソースコードレベルの SAFE の方が正確にバグを挿入できることから，本稿では SAFE をフォールトインジェクタとして使用する．

文献 [13] ではフォールトインジェクタ G-SWFIT の性

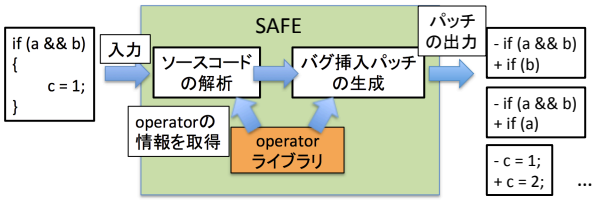

図 1 SAFE の動作の流れ

能を代表性の観点から調査している．現実のバグはテスト段階で取り除かれているはずであるため，テストケースを実行してもバグは発現しない．文献 [13] ではフォールトインジェクタが挿入するバグがテストケースで発現しないなら現実のバグに近いバグを挿入できていると考えている．そこで全体の 50% 以下のテストケースでしか発現しなかったバグは代表性があり，実際のバグに近いと考え，G-SWFIT が挿入するバグの代表性の調査を行っている．その結果，G-SWFIT が挿入するバグは高い代表性を持つことを示している．文献 [13] の研究では挿入したバグがクラッシュを引き起こすかどうかのみでフォールトインエジェクタの性能を評価している．本稿ではクラッシュ時の様子をクラッシュログを見て詳細に分析し，SFI によるクラッシュの性質が現実のクラッシュに近いかどうかを評価している．

3. 実験環境

本稿で使用するフォールトインジェクタ SAFE はソースコードを解析することによって，バグの挿入を行う．図 1 に SAFE の動作の流れを示す．SAFE はソースコードを入力とし，文献 [8]，[11] で定められたオペレータに従いソースコードを解析しバグの挿入可能個所を探す．そしてバグを挿入するためのパッチを出力する．SAFE にはソースコードが必要であるため，本稿の実験対象はソースコードが入手可能な OS である Linux とし，安定板であり広く使用されている Linux 2.6.32.60 を用いる．Linux のファイルシステムにはバグが多い [2]．そのためファイルシステムが原因で OS がクラッシュすることが多く，本稿で現実のクラッシュログとして使用する RedHat 社が収集しているクラッシュログも大部分の原因がファイルシステムであると考えられる．そこで RedHat 社が収集しているクラッシュログが多くが利用できると考えられるため，本稿では Linux のファイルシステムを調査対象にする．調査対象とするファイルシステムは広く使用されている ext2, ext3, ext4, xfs, fat の 5 つである．

図 2 に SAFE による SFI の例を示す．図 2 は本稿の実験環境で実際にクラッシュを引き起こしたパッチの例である．図 2 では SAFE の OMVIV オペレータによる初期化忘れのバグを挿入している．本来 NULL で初期化されている parent を初期化しないようにソースコードを変更す

```
339| void ext2_rsv_window_add(struct super_block *sb,
340|                 struct ext2_reserve_window_node *rsv)
341| {
347|       struct rb_node * parent = NULL;
348|       struct ext2_reserve_window_node *this;
349|
365|       rb_link_node(node, parent, p);
366|       rb_insert_color(node, root);
367| }
```

```
@@ -347,1 +347,1 @@
-       struct rb_node * parent = NULL;
+       struct rb_node * parent = parent;
```

初期化忘れのバグの挿入

```
339| void ext2_rsv_window_add(struct super_block *sb,
340|                 struct ext2_reserve_window_node *rsv)
341| {
347|       struct rb_node * parent = parent; //初期化の削除
348|       struct ext2_reserve_window_node *this;
349|
365|       rb_link_node(node, parent, p); //エラー発生
366|       rb_insert_color(node, root);
367| }
```

図 2 SAFE による SFI の例

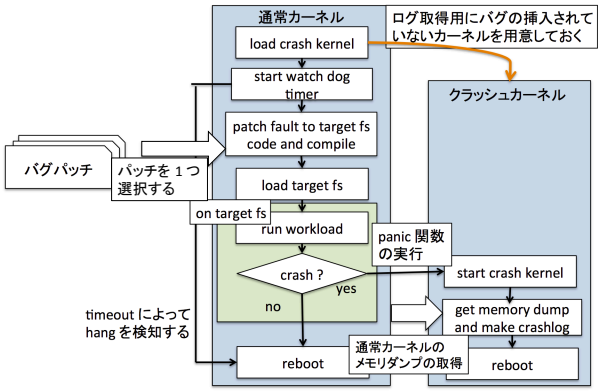

図 3 実験環境の動作の流れ

るパッチを当てる．この変更を施すことによって 365 行目の rb_link_node において parent が誤った値を保持したまま実行されることがある．その際にクラッシュが発生する．

本稿では Linux カーネルのソースコードにおいて各ファイルシステムのディレクトリ内のファイルに変更を加えるパッチのみを用いる．SAFE はファイルシステムディレクトリ以外のファイルにも変更を行うパッチを生成する．例えば include/linux/mm.h や arch/x86/include/asm/processor.h などのファイルへのパッチを生成する．ファイルシステムディレクトリ内のファイル以外への変更は，ファイルシステムのバグを再現にはならないと考え，ファイルシステムディレクトリ外のファイルの変更を行うパッチは使用しない．各ファイルシステムのパッチ生成状況を表 2 に示す．なお，パッチの生成数が 0 であるオペレータは記載していない．

実験の流れを説明する．実験システムの動作の流れを図 3 に示す．本稿の実験システムは自動で SFI を行いクラッシュログを取得する．最初に実験システムは，マシン起動直後にクラッシュカーネルをメモリにロードする．本稿の実験ではカーネルに対して SFI を行う．そのためカーネルにエラーが起き，正しくクラッシュログを取ることができない恐れがある．そこで，クラッシュログ取得用のカー

表 2 ファイルシステム，オペレータごとのパッチの生成数（個）

file system	OMFC	OMVIV	OMVAV	OMVAE	OMIA	OMIFS	OMIEB	OMLAC	OMLOC	OMLPA	OWVAV	OWPFV	OWAEP
ext2	376	70	146	441	472	461	70	96	68	940	146	611	98
ext3	780	152	292	962	968	936	113	215	126	1679	292	1212	215
ext4	759	188	267	1313	1299	184	1216	239	111	2045	267	1512	239
fat	254	23	234	491	389	350	70	148	30	1002	234	681	148
xfs	2432	399	1780	3365	3312	3138	1018	922	620	8657	1780	3399	922

```
BUG: unable to handle kernel NULL pointer dereference at (null)
IP: [<ffffffff811d2042>] rb_insert_color+0xd6/0xe5
...
Pid: 2046, comm: mount Not tainted 2.6.32.60 #1 PowerEdge T410
RIP: 0010:[<ffffffff811d2042>]  [<ffffffff811d2042>] rb_insert_color+0xd6/0xe5
RSP: 0018:ffff880129e2bc20  EFLAGS: 00010246
...
Call Trace:
 [<ffffffffa0226612>] ext3_rsv_window_add+0x5e/0x60 [ext3]
 [<ffffffffa0234544>] ext3_fill_super+0xaf7/0x171b [ext3]
 [<ffffffff8127c938>] ? get_device+0x19/0x1f
 [<ffffffff8111998c>] ? sget+0x388/0x39a
 [<ffffffff81119c0f>] get_sb_bdev+0x139/0x19c
 [<ffffffffa0233a4d>] ? ext3_fill_super+0x0/0x171b [ext3]
 [<ffffffffa0231364>] ext3_get_sb+0x18/0x1a [ext3]
 [<ffffffff811193c0>] vfs_kern_mount+0xa9/0x168
 [<ffffffff811194e7>] do_kern_mount+0x4d/0xed
 [<ffffffff81131c46>] do_mount+0x731/0x793
 [<ffffffff810e79d2>] ? strndup_user+0x5d/0x87
 [<ffffffff81131d30>] sys_mount+0x88/0xc2
 [<ffffffff81011cf2>] system_call_fastpath+0x16/0x1b
```

図 4 oops の例

ネルを用いて，正確にクラッシュログの取得を行う．次に watch dog timer を起動させる．本稿の実験システムは途中でハングを起こす可能性がある．そのため watch dog timer を用いてハングを検知することによって，実験システムの停止を回避する．次に SAFE を用いて生成しておいたパッチを実験対象のファイルシステムに 1 つ当て，バグを挿入する．その後，実験対象のファイルシステムの機能を使用するようなワークロードを実行する．ワークロードは Linux の機能を広く利用する UnixBench を用いる．ワークロード実行中にクラッシュが起きたら，クラッシュカーネルに操作が移り，クラッシュ時のメモリダンプを収集し，クラッシュログを抽出する．全行程が終了したのち，マシンを再起動する．以上のシステムを物理マシン 10 台上で実行する．なお，実験に使用した物理マシンの CPU アーキテクチャは x86_64 である．

クラッシュログには oops メッセージが含まれている．oops メッセージの例を図 4 に示す．oops メッセージにはクラッシュの原因や，クラッシュ直前のコールトレースの状態といった情報が含まれている．oops は 1 行目にクラッシュ原因が書かれており，図 4 には NULL ポインタ参照がクラッシュ原因であると記述されている．Pid の行には Linux のバージョンが 2.6.32.60 であると記述されている．またクラッシュ時に実行されていた関数が rb_insert_color である．コールトレースの情報を見ると，システムコールを呼び出す関数である system_call_fastpath によりカーネルに操作が切り替わり，mount システムコールが実行されていることがわかる．また ext3_get_sb などの関数には関数名の右にモジュール情報が付加されており，ext3 モジュールの関数であることがわかる．本稿のクラッシュログの比較では，この oops メッセージを用いる．

4. クラッシュログの比較

SFI によるクラッシュログは各ファイルシステム毎，各オペレータ毎に表 3 に示すように収集した．以下から現実のクラッシュログと SFI によるクラッシュログの比較，分析を行う．

RedHat 社が収集しているクラッシュログには様々な CPU アーキテクチャのログや，様々なカーネルバージョンのログが含まれている．そこで本稿では x86 もしくは x86_64 アーキテクチャである，Linux のバージョンが 2.6.32 台である，ファイルシステムに関連性が高い，これら 3 つの条件を満たすログを用いる．以上の条件に該当するクラッシュログを判別するために，oops メッセージに含まれるレジスタの情報を用いて CPU アーキテクチャを特定する．また oops メッセージに含まれる Linux カーネルバージョンの情報を用いてバージョンを特定する．またクラッシュがファイルシステムに関連性が高いかを判断するために，コールトレースの情報を用いる．oops にはコールトレース内の関数がどのモジュールの関数であるかが記述されている．そこで本稿では，コールトレース中に 1 つでもファイルシステムモジュールによる関数が含まれているクラッシュログをファイルシステムに関連性の高いクラッシュログであると判断する．RedHat 社のクラッシュログのうち，311 件が以上の 3 条件を満たしており，この 311 件のクラッシュログを本稿では，現実のクラッシュログとして用いる．なお 311 件の内訳は，ext2 が 7，ext3 が 145，ext4 が 140，fat が 15，xfs が 4 件である．

4.1 クラッシュ原因の比較

oops メッセージにはクラッシュ原因が書かれている．本研究で扱うクラッシュログ中に書かれているクラッシュ原因は，以下の 6 種類である．

- **unable to handle NULL pointer dereference**：NULL ポインタ参照時に起きる．本稿中では NULL と表記する．
- **unable to handle kernel paging request**：ページサイズ以上のアドレスにアクセスする際に起きる．本稿中では ERR_PTR を表記する．

表 3　オペレータ毎のクラッシュ数

file system	OMFC	OMVIV	OMVAV	OMVAE	OMIA	OMIFS	OMIEB	OMLAC	OMLOC	OMLPA	OWVAV	OWPFV	OWAEP
ext2	32	31	0	137	0	58	0	0	0	0	0	90	0
ext3	60	10	27	156	33	65	0	6	3	192	27	208	9
ext4	17	53	76	422	99	106	5	27	8	2301	95	312	16
fat	23	9	51	101	21	15	0	6	6	224	45	23	26
xfs	97	0	4	64	0	11	0	0	0	0	0	19	0

表 4　クラッシュ原因ごとのクラッシュ数

file system	NULL	ERR_PTR	kernel BUG	DIVIDE	GENERAL
RedHat	229	11	22	0	49
all	1739	307	3311	32	81
ext2	224	59	79	4	16
ext3	599	73	80	27	3
ext4	382	66	3063	1	20
fat	325	99	85	0	41
xfs	209	10	4	0	1

表 5　コールトレース最下位の関数の数

file system	syscall hadler	child_rip	kernel_thread_helper	その他
RedHat	275	8	19	9
all	4101	1337	0	28
ext2	376	1	0	5
ext3	770	1	0	8
ext4	2218	1303	0	10
fat	543	2	0	5
xfs	194	30	0	0

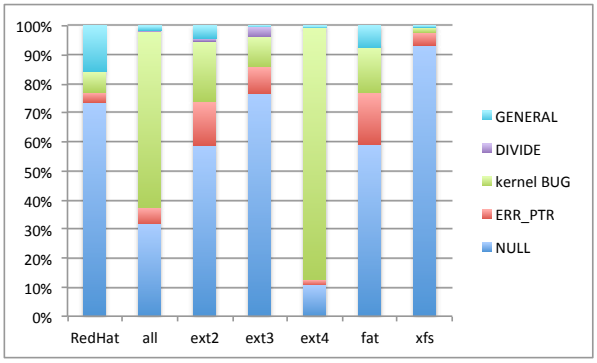

図 5　クラッシュ原因の割合

- **general protection fault**：NULL や ERR_PTR などで定義されていない保護違反が起きた際に起きる．本稿中では GENERAL と表記する．
- **kernel BUG**：ソースコード中の BUG_ON() 関数の実行により起きる．BUG_ON() はチェックする値を引数とし，誤った値であると検出するとカーネルの動作を停止させる関数である．
- **divide error**：0 除算を行った際に起きる．本稿中では DIVIDE と表記する．

クラッシュ原因はクラッシュ時のカーネルの状態を表す要素の一つであり，SFI が起こすクラッシュにおいて現実の状態と一致すべきである．そこでクラッシュ原因について現実のクラッシュログと SFI によるクラッシュログとで，原因毎の個数と割合を比較した．結果を表 4 と図 5 に示す．図 5 では，一番左の棒が RedHat 社が収集している現実のクラッシュログの結果であり，それ以外は SFI によるクラッシュログの結果である．all は SFI による全てのクラッシュログをまとめた場合の結果であり，右 5 つの棒は SFI 対象のファイルシステム毎に結果を分けて表示している．現実のクラッシュログでは NULL の割合が最も多くなった．ext4 以外のファイルシステムでも NULL の割合が最大であり，現実のクラッシュと似た傾向を示した．ext4 は kernel BUG の割合が最大になった．ext3，ext4 は現実のクラッシュログ中でも約半数を占めるが，そのことを考慮すると現実のクラッシュログでは NULL の割合は高く，kernel BUG の割合は低くなった．また，現実のクラッシュログが 16% の割合で GENERAL を発生させているのに対して，SFI によるクラッシュログにおいては高々 8% の割合にしかならなかった．以上からクラッシュ原因に関して，現実のクラッシュログと SFI によるクラッシュログの性質は一致しなかった．

4.2　コールトレース最下位の関数の比較

SFI が現実のクラッシュを再現するためには，現実のコールトレースの状態を再現することが求められる．そこでコールトレース最下位の関数を比較を行う．コールトレース最下位の関数は，どの関数によってカーネルに操作が移り，これからどのような操作がカーネルで行われるかを表す．コールトレース最下位の関数以降に呼び出される関数は，コールトレース最下位の関数が呼んでいる関数になるため，コールトレース最下位の関数は以降にどの関数が呼び出されるかを決定付ける．そのためコールトレース最下位の関数は，コールトレースに記録される情報を決定付ける．そこで，oops のコールトレースがどの関数から始まっているかを比較することにより，コールトレースがどの程度類似するかを調査した．結果を表 5，図 6 に示す．syscall_fast_path や sysenter_do_call などのシステムコールを呼び出すためのハンドラを syscall handler と記述する．現実のコールトレースも，SFI によるコールトレースも syscall handler によって開始されクラッシュする結果が多かった．現実のコールトレースでは最下位の関数のうち 6% が kernel_thread_helper であったが，SFI によるコールトレースでは kernel_thread_helper が最下位になるログが存在しなかった．以上からコールトレースについて kernel_thread_helper など一部の関数については，コールトレース最下位の関数の性質が一致しないため，現実のクラッシュログと SFI によるクラッシュログとで性質は一致しないことを示した．

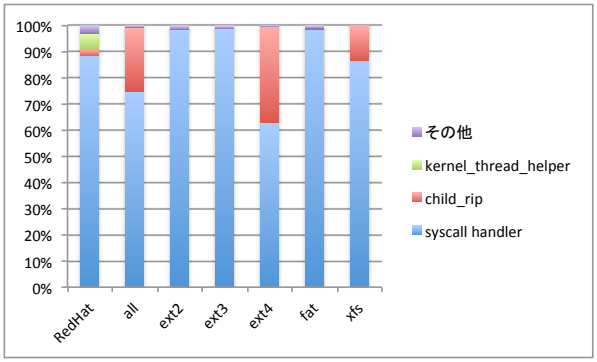

図 6 コールトレース最下位の関数の割合

表 6 ラッシュ時に実行されているモジュールの数

file system	fs/own	mm	fs	fs/jbd	fs/jbd2	include/linux	lib	その他
RedHat	85	47	23	23	2	31	59	40
all	2059	1715	332	268	122	104	51	131
ext2	155	0	62	0	0	15	25	28
ext3	128	5	57	268	1	65	10	48
ext4	1492	1707	9	0	121	17	6	16
fat	216	3	86	0	0	6	10	11
xfs	68	0	118	0	0	1	0	26

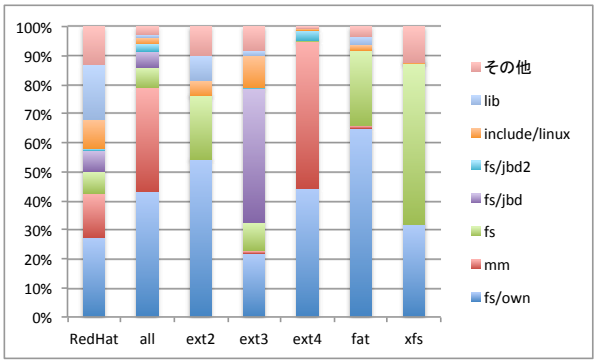

図 7 クラッシュ時に実行されているモジュールの割合

4.3 クラッシュ時に実行されているモジュールの比較

クラッシュがどの関数によって引き起こされるかを再現することは，クラッシュ時のカーネルの状態を再現するために重要である．そこでクラッシュが起きた時に実行されていた関数を比較したいが，クラッシュ時に実行されている関数の比較は，実行されている関数の種類が多いため難しい．そこで，関数レベルよりも粒度の荒いモジュールレベルでの調査を行い，クラッシュ時にどのモジュールが実行されていたかを比較した．本稿ではクラッシュ時に実行されていた関数が定義されているファイルを含むディレクトリをモジュールとして考える．結果を表 6，図 7 に示す． 表 6，図 7 において fs/own の own には対象のファイルシステムの名前が入る．例えば ext2 の場合は fs/own は fs/ext2 を意味することになる．SFI による実験において，ext4 では mm の割合が大きいが他のファイルシステムでは大きな値を示さなかった，ext2，fat，xfs では fs が比較的大きな割合を占めるが ext3，ext4 では大きな割合を占めないなどファイルシステム毎に大きく性質が異なっていた．また現実のクラッシュログの傾向との一致を示すファイルシステムはなかった．現実のクラッシュログは lib によるクラッシュが比較的大きな割合を示すが，現実のクラッシュログにおいて lib に対して比較的高いクラッシュ率を示す ext2 が占める割合は少なく，現実のクラッシュログのファイルシステム毎の存在比率を考慮しても一致しないためである．以上からクラッシュ時に実行されているモジュールについて，現実のクラッシュログと SFI によるクラッシュログとでは性質が一致しなかった．

5. まとめと今後の課題

最先端のフォールトインジェクタである SAFE の Linux カーネルに対する性能の評価を行った．本稿では Linux に対して SAFE を用いた際に得られるクラッシュログと，RedHat 社が収集している現実のクラッシュログとを比較し，どの程度 SAFE が引き起こすクラッシュが現実のクラッシュと性質が近いかを評価した．その結果，SAFE が引き起こすクラッシュは現実のクラッシュとは性質が異なることを示した．

得られたクラッシュログの数が少ないファイルシステムに対しては引き続き実験を行い，統計的に有意な程度にクラッシュログを収集していく予定である．また，本稿で行った分析以外の手法を用いて，更に現実のクラッシュログと，SFI によるクラッシュログの比較・分析を進めて行く予定である．

参考文献

[1] Jann, J., Burugula, R. S., Wu, C. and El Maghraoui, K.: An OS-Hypervisor Infrastructure for Automated OS Crash Diagnosis and Recovery in a Virtualized Environment, *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, IEEE, pp. 195–202 (2012).

[2] Palix, N., Thomas, G., Saha, S., Calvès, C., Lawall, J. and Muller, G.: Faults in Linux: Ten years later, *ACM SIGARCH Computer Architecture News*, Vol. 39, No. 1, ACM, pp. 305–318 (2011).

[3] Swift, M. M., Annamalai, M., Bershad, B. N. and Levy, H. M.: Recovering device drivers, *ACM Transactions on Computer Systems (TOCS)*, Vol. 24, No. 4, pp. 333–360 (2006).

[4] Depoutovitch, A. and Stumm, M.: Otherworld: giving applications a chance to survive OS kernel crashes, *Proceedings of the 5th European conference on Computer systems*, ACM, pp. 181–194 (2010).

[5] Swift, M. M., Martin, S., Levy, H. M. and Eggers, S. J.: Nooks: An architecture for reliable device drivers, *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, ACM, pp. 102–107 (2002).

[6] David, F. M., Chan, E., Carlyle, J. C. and Campbell, R. H.: CuriOS: Improving Reliability through Operating System Structure., *OSDI*, pp. 59–72 (2008).

[7] Kadav, A., Renzelmann, M. J. and Swift, M. M.: Fine-grained fault tolerance using device checkpoints, *ACM*

SIGARCH Computer Architecture News, Vol. 41, No. 1, ACM, pp. 473–484 (2013).

[8] Cotroneo, D., Lanzaro, A., Natella, R. and Barbosa, R.: Experimental analysis of binary-level software fault injection in complex software, *Dependable Computing Conference (EDCC), 2012 Ninth European*, IEEE, pp. 162–172 (2012).

[9] Chen, P. M., Ng, W. T., Chandra, S., Aycock, C., Rajamani, G. and Lowell, D.: *The Rio file cache: Surviving operating system crashes*, Vol. 31, No. 9, ACM (1996).

[10] Lee, I., Tang, D., Iyer, R. K. et al.: *Measurement and Analysis of Operating System Fault Tolerance*, Citeseer (1992).

[11] Duraes, J. A. and Madeira, H. S.: Emulation of software faults: A field data study and a practical approach, *Software Engineering, IEEE Transactions on*, Vol. 32, No. 11, pp. 849–867 (2006).

[12] Lyu, M. R. et al.: *Handbook of software reliability engineering*, Vol. 222, IEEE computer society press CA (1996).

[13] Natella, R., Cotroneo, D., Duraes, J. and Madeira, H.: Representativeness analysis of injected software faults in complex software, *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, IEEE, pp. 437–446 (2010).