Vol.2009-OS-112 No.20
2009/8/6

仮想マシンモニタを用いた Software Fault Injection

星 洋 平[†1] 大 山 恵 弘[†1]

セキュリティシステムのテストにおいては，テスト環境で動くソフトウェアを攻撃し，セキュリティシステムの反応を調べる作業が不可欠である．しかし，脆弱なソフトウェアとそれに対する攻撃コードを十分な数用意することは簡単ではない．様々な条件下におけるセキュリティシステムの動作を観測するための一つの有望な方法は，fault injection を行うツールによって，攻撃を受けたかのような効果をソフトウェアの中に作り出すことである．本論文では，セキュリティシステムをテストするための Software-Implemented Fault Injection (SWIFI) を実現するシステム HyperAttacker を提案する．HyperAttacker はユーザの与えるシナリオに沿ってソフトウェアに異常を注入する．監視されているソフトウェアにおいて，シナリオで指示されたイベントが発生すると，HyperAttacker はイベントに関係づけられた異常を注入する．注入できる異常には，レジスタ，ユーザメモリ領域，カーネルメモリ領域が持つ値の改変が含まれる．HyperAttacker は仮想マシンモニタ Xen を改造して実装されている．異常は 1 つの特殊な仮想マシンから，テスト対象のソフトウェアが動いている他の仮想マシンに対して注入される．ユーザは HyperAttacker の利用にあたり，テストされるソフトウェアを修正する必要はなく，そのソフトウェアを動かす環境に特殊なプログラムをインストールする必要もない．実験を行った結果，HyperAttacker が様々なセキュリティソフトウェアのテストでうまく動作することを確認した．

Software Fault Injection Using Virtual Machine Monitor

Youhei Hoshi[†1] and Yoshihiro Oyama[†1]

An essential phase in testing of security systems is to attack target software in a testing environment and check the response of the security systems. Unfortunately, it is not straightforward to obtain or develop a sufficient number of vulnerable software and attack code. One promising way for observing the behavior of security systems in various conditions is to create the effect of attacks in target software with fault injection tools. In this paper, we propose HyperAttacker, a Software-Implemented Fault Injection (SWIFI) system for testing security systems, which injects faults based on scenarios provided by users. When an event specified in a scenario occurs in monitored software, HyperAttacker injects faults associated with the event. Supported faults include updates of values in registers, user memory areas, and kernel memory areas. HyperAttacker is implemented by modifying a virtual machine monitor Xen. Faults are injected from one special virtual machine to target software running in another virtual machine. The users do not need to modify tested software or install an additional program in a testing environment. Our experimental results show that HyperAttacker worked well in testing of various kinds of security software.

1. はじめに

近年，侵入検知システム，サンドボックス，メモリ破壊検知機構などの様々なセキュリティシステムが使用されている．セキュリティシステムの開発においては，システムが期待通りの動作を示すかテストを行う必要がある．そのようなテストでは，ソフトウェアに対して実際に攻撃を仕掛け，セキュリティシステムがどう反応するかを観察することが望ましい．しかし，脆弱なソフトウェアとそれに対する攻撃コードを用意するには多大な手間がかかる．用意のための 1 つの手法は，既知の脆弱性を持つソフトウェアを入手し，それに対する攻撃コードを入手もしくは自分で開発することであるが，一般に攻撃コードの入手は簡単ではない．また，脆弱なソフトウェアを解析し，脆弱性を突くコードを開発する作業はスキルや知識を必要とし，システム開発者の負担が大きい．もう 1 つの手法は，ソフトウェアのソースコードを書き換え，意図的に脆弱性を含ませるというものである．しかし，この手法ではソフトウェアが元のものと異なってしまうという問題がある．さらに，ソースコードが必要となるため，オープンソースのソフトウェアのみが対象となるという問題もある．

一方，脆弱なソフトウェアを用意することなしにセキュリティシステムをテストする手法も考えられる．ソフトウェアを用いてシステムに異常を注入する手法である SoftwareFault Injection が知られていて，Software Fault Injection を行うソフトウェアである Software-Implemented Fault Injection (SWIFI)[2)3)9)10)19)] が開発されている．SWIFI は特殊なハードウェアを利用することなく，異常注入を行うことができる．本研究では，SWIFI を用い

†1 電気通信大学
The University of Electro-Communications

てソフトウェアに異常を注入することが可能である点に着目する．ソフトウェアに異常を注入することで．脆弱性がないソフトウェアに対しても，攻撃が行なわれた状態を作り出すことができる．既知の脆弱性を突く攻撃だけでなく，様々なプログラム部分で行なわれる，様々な種類の攻撃を模倣することができるため，多様な条件でセキュリティシステムのテストができるようになることが期待される．

既存の SWIFI は，異常を注入できる箇所が非常に限定されていたり[2)]，異常が注入される条件がランダムであったり[10)]，注入される異常がハードウェアレベルのものであったり[3)] するため，脆弱性を突く攻撃を模倣することは難しい．また，カーネルコードに異常が注入されると，システムがクラッシュしてしまうため，1 台のマシン上でテストと結果の観測の両方を行うことが難しい．

そこで本研究では，ユーザの与えるシナリオに沿って異常を注入する SWIFI である HyperAttacker を提案する．HyperAttacker は言語レベルのシナリオに沿って，特定のプログラムの特定の箇所に異常を注入できる SWIFI である．例えば,「プログラム `httpd` が関数 `f` を実行すると，スタックを破壊する」といった異常が注入できる．これにより，特定のプログラムを攻撃が行なわれた状態にすることができる．HyperAttacker は仮想マシンモニタ (VMM) である Xen[1)] を用いて実現した．HyperAttacker では，Xen 上に 2 つの VM を用意し，片方の VM で異常を注入するプログラムを，もう片方の VM で異常が注入されるプログラムを動かす．前者のプログラムは，VM をまたいで異常を注入する．

VMM の利用には多くの利点がある．第 1 の利点は，1 台のマシン上に複数の仮想マシン (VM) を作成することにより，テストと結果の観測を 1 台のマシンで行うことができる点である．第 2 の利点は，容易に様々な箇所に異常を注入できるという点である．第 3 の利点は，VM がクラッシュするようなテストを行っても，他の VM から結果を観測することができるという点である．

我々は文献 17) で HyperAttacker のプロトタイプについて述べた．プロトタイプは ptrace システムコールを用いてプロセスがプロセスを監視する方法で実装されていた．本論文で述べるシステムは，VMM を用いて実装されている．そのため，プロトタイプとは違い，すでに ptrace システムコールにより監視されているプロセスへの異常注入や，カーネルデータへの異常注入が可能である．

本論文の構成は次のようになる．2 章では HyperAttacker の設計について述べる．3 章では実装について述べ，4 章では HyperAttacker を用いた実験の結果を示す．5 章では関連研究について述べ，6 章で本論文のまとめと今後の課題について述べる．

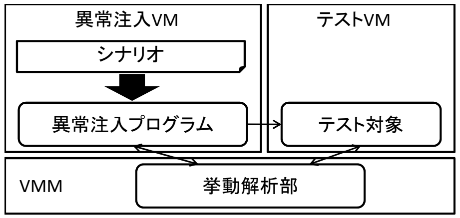

図 1 HyperAttacker の設計
Fig. 1 Design of HyperAttacker

2. 設　　計

2.1 HyperAttacker の概要

提案する SWIFI，HyperAttacker を**図 1** に示す．HyperAttacker は IA-32 アーキテクチャを対象とする．HyperAttacker を利用する際には，VMM 上で，異常注入 VM とテスト VM と呼ばれる 2 つの VM を動作させる．HyperAttacker は VMM 内にある挙動解析部と，異常注入 VM 内の異常注入プログラムからなる．挙動解析部はテスト対象の挙動を解析する．テスト対象とは異常が注入されるソフトウェアのことであり，テスト VM 内で動作させる．セキュリティシステムのテストを行なう者は異常注入プログラムに，どんな条件でどんな異常を注入するかを記述したシナリオを与える．異常注入プログラムはテスト対象の挙動とシナリオを比較し，シナリオに沿った挙動を示した時点で異常を注入する．異常注入 VM とテスト VM，挙動解析部間での通信は，すべて VMM を介して行う．テスト VM に特別なソフトウェアを導入する必要はなく，平常時に使用している環境がそのままテスト環境となる．また，挙動解析および異常注入を行うために，テスト対象のソースコード変更は必要ない．挙動解析部はシナリオで指定されたテスト対象の挙動のみを解析する．ユーザプロセスを対象とした異常注入では，テスト対象以外のプロセスには影響を与えない．

HyperAttacker は，メモリやレジスタの持つ値が別の値に改竄されるタイプの攻撃により発生する異常を扱うものとする．このタイプの攻撃の例としては，バッファオーバーフロー攻撃，rootkit によるカーネル内のデータやコードの改竄が挙げられる．不正な入力データによる Web アプリケーションへの攻撃や，DoS 攻撃，レース攻撃は対象外とする．

2.2 シナリオ

シナリオには (1) テスト対象の情報と，(2) 異常が注入される条件，(3) 条件が成立した

Vol.2009-OS-112 No.20
2009/8/6

時に実行される操作を記述する．文法を**図 2** に記載する．

target では，テスト対象の情報を記述する．情報としてはプログラム名 (*proc-name*) とプロセス ID (*proc-pid*) を用いることができる．カーネルに異常を注入するには，プログラム名に `kernel` を与える．

condition では，異常が注入される条件を指定する．cond-main には異常が注入されるトリガーとなる挙動を記述する．cond-before (指定された関数・システムコール呼び出しの直前に来た)，cond-after (呼び出しの直後に来た)，cond-head (指定された関数・システムコール実行処理の先頭に来た)，cond-tail (末尾に来た) が指定できる．cond-before の `before` に続く部分には対象の関数・システムコール名を指定する．position を指定することにより，特定の関数内での関数・システムコールの呼び出しを指定することができる．cond-option には条件に関する補助的な情報を記述する．`numoccurred` に続く部分には，指定した条件が何回成立したら異常が注入されるかを記述する．`possibility` には続く部分に異常が注入される確率を記述する．

action では，実行される操作を指定する．obj には異常が注入され，値が改竄される箇所を指定する．`eax` などのレジスタや，リターンアドレス (`retaddr`)，ランダムなアドレス (`randaddr`)，大域変数が保持する値が指すアドレス (*glovalvar*)，即値 (*immediate*) が指定できる．`randaddr` は範囲指定 (start, limit) が可能である．data では，上書きするデータを指定する．データとしては，大域変数が保持している値 (`varval`)，ランダムな値 (`randbytes`)，ユーザの指定する値 (`bytes`) が指定できる．

シナリオの記述に，ユーザは大域変数や関数の名前を使用することができる．ここで，HyperAttacker の実装は，テスト対象のバイナリやシンボル情報を利用できることを仮定している．なお，局所変数名をシナリオの記述に用いることはできない．

シナリオの例を**図 3** に示す．このシナリオではプログラム名 `httpd` のプロセスの挙動を解析することが指定されている．異常注入の条件としては,「関数 `parse_data` 中の関数 `strcpy` の 3 回目の呼び出しの直後」というものが指定されている．注入される異常は,「リターンアドレスを関数 `delete_file` のアドレスに書き換える」というものが指定されている．このシナリオにより，関数 `strcpy` の呼び出しにバッファオーバーフロー脆弱性が存在することにより，リターンアドレスが改竄される攻撃を模倣することができる．

2.3 異常注入の流れ

まず，テスト VM 内でテスト対象を実行する．その後，異常注入 VM 内で異常注入プログラムにシナリオを与えて実行する．挙動解析部は，異常注入プログラムと連携し，テスト対象の挙動を解析する．解析した情報は異常注入プログラムに送信される．異常注入プログラムは，送られてきた情報をシナリオの異常が注入される条件と比較する．条件が成立していなければ，何も行なわない．条件が成立していれば，シナリオに沿ってテスト VM の資源を操作して異常を注入する．これにより，テスト VM 内のテスト対象は，あたかも攻撃が行われたかのような挙動を示す．

scenario	→	(target condition action+)*
target	→	`name=`*proc-name* | `pid=`*proc-pid*
condition	→	cond-main cond-option*
cond-main	→	cond-before | cond-after | cond-head | cond-tail
cond-before	→	`before` { *funcname* | *syscallname* } position*
position	→	`in` *funcname*
cond-option	→	`numoccurred=`*num* | `possibility=`*num*
action	→	obj `<-` data
obj	→	regs | `retaddr` | `randaddr`(start, limit) | `varval` *globalvar* | `imm` *immediate*
regs	→	`eax` | · · · | `esp` | `*eax` | · · · | `*esp`
start	→	regs | `imm` *immediate* | `varval` *globalvar*
limit	→	regs | `imm` *immediate* | `varval` *globalvar*
data	→	`varval` *globalvar* | `randbytes` *num* | `bytes` *b1 b2* · · ·

図 2 シナリオ文法
Fig. 2 Scenario Syntax

```
target:
  name=httpd

condition:
  after strcpy in parse_data
  numoccurred=3

action:
  retaddr <- varval delete_file
```

図 3 シナリオ例
Fig. 3 Example of Scenario

2.4 HyperAttacker の使用例

HyperAttacker を用いた，セキュリティシステムのテスト例をいくつか挙げる．まず，バッ

ファオーバーフロー攻撃検出機構のテストに用いる場合について述べる．ここで，検出機構は StackGuard[5] や SSP[7] のように，ソフトウェアのいくつかの関数の先頭と末尾に検出コードを挿入するものとする．検出機構が正常に作動することを確認するとき，開発者はテスト VM 内で，検出コードが挿入されたソフトウェアを動作させる．また，ソフトウェア内の適当な関数が呼び出された直後にスタックを破壊するシナリオを HyperAttacker に与える．すると，このソフトウェアにバッファオーバーフロー攻撃が行なわれた状態が作り出され，検出機構が作動するかどうか確認することができる．

また，侵入検知システムのテストに用いることもできる．ここでテストを行なう侵入検知システムは，文献 8) のように，システムコール列を用いて異常を検出するものであるとする．開発者はテストにおいて，攻撃を示唆する異常なシステムコールをソフトウェアが発行するようにし，システムが検出し警告を出すかどうかを確かめる必要がある．しかし，ソフトウェアに手を加えて脆弱性を入れるのは手間がかかる．そこで，開発者はシステムコールが発行されたとき，一定の確率で別のシステムコールに改竄するシナリオを HyperAttacker に与える．すると，異常なシステムコールが発行される状態を作り出すことができ，システムがどの程度の精度で侵入を検知するかを効率的に評価することができる．

3. 実　　装

3.1 Xen API

HyperAttacker は，VMM に対して変更を加える箇所を最小限に止めるように実装されている．これにより，導入が容易になることが期待される．

Xen では，ユーザ空間から VMM 層にアクセスするためのインターフェースである Xen API が提供されている[4]．Xen が提供するシステム制御コマンドラインツールである `xm` も Xen API を使用している．Xen API を用いることで，他の VM で使用しているメモリをマップすることや，レジスタのロード・ストア，他の VM の停止・再開などをユーザプロセスで実行することが可能になる．これらの，通常は VMM 層でしか行なえなかった処理をユーザプロセスで行えるため，HyperAttacker の実現において VMM への変更点は一般保護例外の処理部分と，`shared_info` 構造体の拡張のみとなっている．

3.2 テスト VM の資源操作

テスト VM のメモリを操作するために，HyperAttacker では共有メモリを使用している．Xen では，各 VM は Pseudo Physical Address を持つ．仮想アドレスから Pseudo Physical Address への変換は各 VM 上の OS が行なう．ハイパーバイザは Machine Address と呼ばれる実メモリのアドレスによる物理メモリ空間を管理している．ハイパーバイザは Pseudo Physical Address を Machine Address に対応付けることによりメモリ管理を行なっている．HyperAttacker ではテスト対象内の書き込みを行なう仮想アドレスに割り当てられた Machine Address を，異常注入プログラムの仮想アドレス空間にも割り当てている．これにより，異常注入プログラムが自身のプロセス空間のメモリに書き込んだ内容が，テスト対象に書き込まれる．

同一の Machine Address への仮想アドレスのマッピングは Xen API が行う．Xen API で提供されているマッピング API では，マッピング先の Machine Address が必要になる．Xen API には，カーネル領域の仮想アドレスから Machine Address に変換する API は用意されているが，ユーザ領域の仮想アドレスを Machine Address に変換する API は用意されていない．そのため，異常注入プログラムはテスト対象の仮想アドレスから Machine Address への変換を行っている．異常注入プログラムは，テスト対象のページテーブルを参照することで Machine Address を取得する．

テスト対象のページテーブルは，以下の手順で取得する．まずテスト対象のプロセスのタスク構造体を参照するため，`init` プロセスのタスク構造体の参照を行う．`init` プロセスのタスク構造体の仮想アドレスをテスト VM のシステムの System.map ファイルから求める．その後，Xen API を用いてその仮想アドレスを Machine Address に変換する．取得した Machine Address を利用し，テスト VM 内の `init` プロセスのタスク構造体を異常注入プログラムから参照する．参照した `init` プロセスのタスク構造体からテスト対象のタスク構造体を取得し，それからページテーブルを得る．

テスト VM のレジスタ操作は，Xen API を利用して異常注入プログラムが行う．異常注入プログラムがテスト VM の仮想 CPU の各レジスタに改竄した値を書き込むことでレジスタ操作を実現している．

3.3 シンボル情報

シナリオに記述された大域変数や関数の名前からそれらが配置されているアドレスを得るため，テスト対象のシンボル情報が必要になる．現在の実装では，objdump コマンドによってテスト対象のバイナリイメージから取得したシンボル情報や，テスト VM 内で動作しているゲスト OS の System.map ファイルを HyperAttacker に与えることとしている．

3.4 ブレークポイント

HyperAttacker では，テスト対象の挙動を解析するため，ブレークポイントを用いている．ブレークポイントをテスト対象のコードに挿入する方法は，尾上らの研究[18] の手法を

Vol.2009-OS-112 No.20
2009/8/6

踏襲している．

HyperAttacker はブレークポイントが挿入される箇所にあるテスト対象中の命令を HLT 命令で上書きする．テスト VM でその HLT 命令が実行されると，一般保護例外が発生し，VMM 内の一般保護例外を処理する関数が呼び出される．その関数は HyperAttacker により，元々のものから挙動解析部の処理に置き換えられている．その関数内では，上書きした元々の命令をエミュレートする．テスト VM には HLT 命令が VMM によって処理されたことを通知する．これら一連の操作により，VMM に処理を移すことが可能となる．

HLT 命令により上書きされる命令について説明する．例えば，異常注入の条件が関数の先頭への到達である場合，関数先頭の `push %ebp` 命令を HLT 命令で上書きすることで，対象の関数に突入し先頭の命令を実行する直前に VMM に処理が移る．同様にシステムコールの発行が条件に指定されていた場合，システムコール割り込みの割り込みハンドラである `system_call` 関数先頭の `push %eax` 命令を HLT 命令で上書きすることで，システムコールの発行を VMM が捕捉することができる．

特定の関数内でのみ異常を注入したいとき (シナリオ文法での position が記述されているとき) は，2 箇所にブレークポイントを挿入している．例えば，`strcpy in parse_data` のような条件が記述されている場合，関数 `parse_data` の先頭と末尾にブレークポイントを挿入し，関数 `parse_data` に入ったことと出たことを把握している．

3.5 異常注入プログラムと挙動解析部間の通信

異常注入プログラムと挙動解析部間の通信には，イベントチャネル通信を用いている．異常注入プログラムは，イベントチャネルをオープンし，仮想 IRQ7 番にバインドする．その後，異常注入プログラムをイベントチャネルが来るまで停止した状態にしておく．挙動解析部はブレークポイントが挿入された命令が実行されると，異常注入プログラムに送信する情報を，異常注入プログラムと挙動解析部で共有されたメモリに書き込み，Dom0 の仮想 CPU に仮想 IRQ7 番を発行する．仮想 IRQ7 番が発行されたことで，異常注入プログラムが待っているイベントチャネルが送信される．処理を再開した異常注入プログラムが，共有メモリに書き込まれた情報を読み込むことで通信を実現している．

4. 実　　験

HyperAttacker を用いて，ソフトウェアに異常を注入する実験を行った．この実験では，既知の脆弱性がないソフトウェアに対して攻撃を模倣するような異常注入を行い，各種セキュリティシステムの動作を確認した．使用した計算機と VMM，OS の仕様は，Intel Pentium 4 2.80 GHz，512 MB Memory，Xen 3.0.4-1 上で Dom0，DomU ともに Linux 2.6.16.33 を動作させている．

```
static void
send_response_tail( httpd_conn* hc )
{
  char buf[1000]

  (void) my_snprintf( buf, sizeof(buf), "\
  <HR>\n\<ADDRESS><A HREF=\"\"></A>
  </ADDRESS>\n\</BODY>\n\</HTML>\n",
  SERVER_ADDRESS, EXPOSED_SERVER_SOFTWARE );
  add_response( hc, buf );
}
```

図 4　thttpd の異常が注入される関数
Fig. 4　Function in thttpd to which a fault is injected

```
target:
  name=thttpd

condition:
  after my_snprintf in send_response_tail
  numoccurred=1

action:
  sp <- randbytes 2000
```

図 5　thttpd に異常注入を行うシナリオ
Fig. 5　Scenario for injecting faults to thttpd

4.1 Stack-Smashing Protector

Web サーバの thttpd[13] に GCC の Stack-Smashing Protector(SSP)[7] による保護を入れ，SSP の動作を確認した．thttpd のバージョンは 2.25b である．SSP はバッファオーバーフロー攻撃の検知機構であり，StackGuard[5] と同種のものである．SSP の保護を有効にしたプログラムでは，関数の先頭で関数フレーム内の特定の場所に特殊なデータ (カナリア) を書き込んでおき，関数末尾でその値を検査する．バッファオーバーフロー攻撃を受けたときはカナリアが破壊されているため，プログラムの実行を停止する．

本実験では，thttpd の libhttpd.c というファイルの，**図 4** に示す箇所に異常を注入するためのシナリオを記述した．そのシナリオを**図 5** に示す．このシナリオでは，プログラム名 `thttpd` のプロセスが，`send_response_tail` 関数内の `my_snprintf` を 1 回実行すると，現在のスタックポインタが指すアドレスから 2000 バイトのランダムなバイト列を書き込むという異常注入が指定されている．`my_snprintf` 関数の脆弱性を利用した，(失敗に終わる) バッファオーバーフロー攻撃を模倣するシナリオである．このシナリオに沿って異常を注入すると，`my_snprintf` 関数の実行直後にスタックが破壊される．それに伴いカナリアも破壊されるため，`send_response_tail` 関数から復帰する時に SSP がバッファオーバーフローを検知する．本実験では thttpd が，カナリアが破壊された時に実行される `__stack_chk_fail` 関数を実行し，プログラムの実行を停止することを確認した．

Vol.2009-OS-112 No.20
2009/8/6

```
void show_config_file ( void )
{
  int fd;

  fd = open( "config_file", O_RDONLY );
  while ( read( fd, buf, 16 ) > 0 ) {
    write( 1, buf, 16 );
  }
}
```

図 6 Systrace が監視するプログラムを簡略化したもの
Fig. 6 Simplified program monitored by Systrace

```
target:
  name=test_open

condition:
  head open in show_config_file
  numoccurred=1

action:
  *ebx <- bytes 0x70 0x61 0x73 0x73 0x77 0x64 0x00
```

図 7 図 6 のプログラムに異常注入を行うシナリオ
Fig. 7 Scenario for injecting faults to program in Fig.6

4.2 Systrace

システムコールを監視するサンドボックスである，Systrace[14] により監視されているプログラムに異常を注入し，Systrace の動作を確認した．Systrace のバージョンは 1.6e である．Systrace はシステムコールの実行を監視し，ソフトウェアが使用できるシステムコールを制御する．システムコール制御はポリシーに従って行なわれ，ポリシーで許可されていないシステムコールが呼び出されたときは，ユーザにその是非を問い合わせる．ポリシーには許可および禁止するシステムコールの種別と，その引数を指定することができる．

本実験では，`test_open` という名前のプログラムの**図 6** が示す部分に，**図 7** のシナリオを用いて異常を注入した．このシナリオでは，プログラム名 `test_open` のプロセスが `open` システムコールを実行する際に，`open` システムコールの引数 `config_file` が改竄され，`passwd` になるという異常を注入する．`bytes` に続く文字列は，`passwd` のバイト表現である．このシナリオに沿って異常を注入したところ，`open` システムコールの引数が改竄されポリシーで許可されていないシステムコールが実行されたことにより，Systrace がユーザに是非を問い合わせたことを確認した．

4.3 カーネルメモリの改竄検出システム

カーネルメモリのメモリ改竄を検出するセキュリティシステムにより監視されている OS のカーネルに異常を注入し，セキュリティシステムの動作を確認した．前述のセキュリティシステムは，カーネルモジュールとして実装されている．モジュールをカーネルに組み込むと，その時点のカーネルメモリの指定した部分 (システムコールテーブル，割り込みテーブルなど) を正しい状態のメモリ内容とみなし，保存する．その後，プロセスがシステムコールを発行するたびに，上記の部分のメモリ内容を保存したメモリ内容と比較する．メモリが書き換えられていれば，攻撃が行なわれているとみなし，管理者に通知する．

```
target:
  name=kernel

condition:
  head init_module
  numoccurred=1

action:
  randaddr (varval sys_call_table, varval sys_call_table + 300) <- randbytes 4
```

図 8 カーネルに異常注入を行うシナリオ
Fig. 8 Scenario for injecting faults to kernel

上記のシステムを組み込んだカーネルに対し，**図 8** のシナリオを用いて異常を注入した．これは任意のプロセスが `init_module` システムコールを実行する際に，システムコールテーブルが改竄されるという異常を注入するものである．このシナリオに沿って異常を注入したところ，システムコールテーブルが改竄され，セキュリティシステムが警告を出すことを確認した．

5. 関連研究

HyperAttacker と同様に，プログラムに意図的に異常を注入し，テストを行うシステムについては，多くの研究がなされている[9]．

Xception[3] は，PowerPC アーキテクチャの環境でカーネルモジュールの形で実現された SWIFI である．Xception が注入する障害はアドレスバスや浮動小数ユニットなどのハードウェア障害である．また，本研究のようにユーザが関数名など言語レベルの情報を用いてシナリオを記述することには言及されていない．

文献 10) の研究では，VMM を用いて VM のメモリ操作を行う SWIFI を提案している．文献 10) で述べられている SWIFI が注入する異常は，メモリを 1 ビット反転させるなど単純なものであり，HyperAttacker のように特定のプロセスに特定の異常を注入することは考えられていない．また，異常を注入する条件がランダムなものとなっているため，特定の条件でテストを行うことは考慮されていない．

文献 11) の研究では，SWIFI (Xception) を用いてソフトウェアに異常を注入する手法を提案している．文献 11) で述べられているシステムは，異常を注入する箇所を指定する際

Vol.2009-OS-112 No.20
2009/8/6

に，メモリアドレスなどで指定する必要があり，本研究のように関数名など言語レベルの情報を用いて条件を記述することができない点で HyperAttacker と異なる．

FIG[2] は，glibc など動的ライブラリの関数呼び出しをフックすることにより返り値を操作し，異常を注入する SWIFI である．FIG は，ユーザの記述する設定ファイルにより注入する異常を設定する点で本研究と共通点を持つ．しかし，FIG は異常注入する箇所がライブラリ関数の返り値のみであるという点で HyperAttacker と異なる．

XenAccess[12] は，Xen のドメイン間をまたがったメモリ操作を可能にするシステムである．本研究のドメイン間メモリ操作は XenAccess を参考とした．XenAccess は他のドメインのメモリ操作を提供することを目的としたシステムであり，本研究のような異常注入を目的としたシステムではない．

黒田らのシステム[19] は，システムコールの動作を改変することによってアプリケーションの実行に異常を注入するものである．HyperAttacker と同じく，各アプリケーションのプログラム情報を用いた異常を注入することができる．しかし，彼らのシステムでは，注入できる異常がシステムコールの動作の改変に限られる上，利用にあたりアプリケーションのソースコードを必要とする．また，カーネルへの異常注入も対象としていない．

KLASY[16] は，C 言語で書かれた OS カーネル用の動的アスペクト指向システムである．また，TOSKANA[6] は NetBSD のカーネルにコードを挿入することを可能にする動的アスペクト指向システムである．これらのシステムを用いると，指定された関数の実行などのタイミングで，望みのコードを実行させることができる．これらのシステムでカーネルに異常を注入し，セキュリティシステムをテストすることは可能であると考えられる．しかし，これらのシステムの研究では fault injection への応用はほとんど考えられていない．本研究は，fault injection によるテストを対象とし，テストを支援するためのシステムの構築に焦点を当てている．また，KLASY や TOSKANA は，仮想マシンモニタを用いていないので，異常を注入したカーネルがクラッシュした時にテストプログラムの動作も止まるという欠点がある．

XenLASY[15] は KLASY を拡張し，Xen 上の複数のドメインにコードを挿入することを可能にしたシステムである．XenLASY は I/O 処理を追跡するためのプロファイラとしての利用に特化しており，同様の技術を fault injection に適用するための方法は明らかにされていない．

6. まとめと今後の課題

ユーザの与えるシナリオに沿って Software Fault Injection を行うことで，ソフトウェアに対する攻撃によって作られる効果を模倣する SWIFI，HyperAttacker を提案した．シナリオは言語レベルの情報を使用して記述することができ，特定のプロセスに対しての異常注入を行なうことができる．実験ではいくつかの異常をソフトウェアに注入し，セキュリティシステムがそれを検出することを確認した．

今後の課題として，まず，シナリオの記述性向上が挙げられる．現在のシナリオでは，異常を注入できるメモリアドレスがレジスタや大域変数が持つ値などに限られている．関数の局所変数が指しているアドレスに異常を注入できるようにするなどの拡張が考えられる．記述できる条件に関しては，ブレークポイントを設定できる箇所を増やすことで，さらに様々な条件の元でテストを行うことができるようになる．別の課題として，複数プロセスにまたがった挙動解析，異常注入を行えるようにすることが挙げられる．広く使われているソフトウェアには，複数のプロセスが協調して動作するものも多い．そういったソフトウェアに対してより多くの条件の元でのテストが行えるようにすることは重要である．また，HyperAttcker の有効性をさらに詳しく評価することも今後の課題である．現在のシステムでも単純なバッファオーバーフロー攻撃やシステムコールテーブル改竄攻撃を模倣することは可能であることは確認したが，セキュリティシステムのテストにどれくらい役立つかは，まだ十分に評価されていない．今後，さらに多くの実験と，それを踏まえたシステムの拡張が必要である．

謝辞　本研究の実施にあたり，東京大学の尾上浩一氏の助言が有益であった．また，本研究の一部は科研費 (19700024) の助成を受けたものである．ここに謝辞を表する．

参　考　文　献

1) Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the Art of Virtualization, *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pp.164–177 (2003).
2) Broadwell, P., Sastry, N. and Traupman, J.: FIG: A Prototype Tool for Online Verification of Recovery Mechanisms, *Proceedings of the Workshop on Self-Healing, Adaptive, and Self-Managed Systems* (2002).
3) Carreira, J., Madeira, H. and Silva, J.G.: Xception: A Technique for the Experi-

mental Evaluation of Dependability in Modern Computers, *IEEE Transactions on Software Engineering*, Vol.24, No.2, pp.125–136 (1998).
4) Chisnall, D.: *The Definitive Guide to the Xen Hypervisor*, Prentice Hall (2007).
5) Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P. and Zhang, Q.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, *Proceedings of the 7th USENIX Security Symposium*, San Antonio, pp.63–78 (1998).
6) Engel, M. and Freisleben, B.: Supporting Autonomic Computing Functionality via Dynamic Operating System Kernel Aspects, *Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pp.51–62 (2005).
7) Etoh, H.: GCC extentions for protecting applications from stack-smashing attacks. http://www.research.ibm.com/trl/projects/security/ssp/.
8) Hofmeyr, S., Forrest, S. and Somayaji, A.: Intrusion Detection Using Sequences of System Calls, *Journal of Computer Security*, Vol.6, No.3, pp.151–180 (1998).
9) Hsueh, M.-C., Tsai, T. and Iyer, R.: Fault Injection Techniques and Tools, *IEEE Computer*, Vol.30, No.4, pp.75–82 (1997).
10) Le, M., Gallagher, A. and Tamir, Y.: Challenges and Opportunities with Fault Injection in Virtualized Systems, *Proceedings of the 1st International Workshop on Virtualization Performance: Analysis, Characterization, and Tools* (2008).
11) Madeira, H., Costa, D. and Vieira, M.: On the Emulation of Software Faults by Software Fault Injection, *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, pp.417–426 (2000).
12) Payne, B.D., Carbone, M. and Lee, W.: Secure and Flexible Monitoring of Virtual Machines, *Proceedings of the 23rd Annual Computer Security Applications Conference* (2007).
13) Poskanzer, J.: thttpd - tiny/turbo/throttling HTTP server. http://www.acme.com/software/thttpd.
14) Provos, N.: Improving Host Security with System Call Policies, *Proceedings of the 12th USENIX Security Symposium*, Washington DC, pp.257–272 (2003).
15) 柳澤佳里，光来健一，千葉滋：XenLASY: Xen の I/O 処理を追跡するためのアスペクト指向プロファイラ，情報処理学会論文誌: プログラミング， Vol.49, No.SIG 1 (PRO35), pp.51–62 (2008).
16) 柳澤佳里，光来健一，千葉滋，石川零：OS カーネル用アスペクト指向システム KLASY，情報処理学会論文誌: プログラミング， Vol.48, No.SIG 10 (PRO33), pp.176–188 (2007).
17) 大山恵弘：セキュリティ機構の開発と評価のための異常注入システム，日本ソフトウェア科学会第 25 回大会論文集 (2008).
18) 尾上浩一，大山恵弘，米澤明憲：システムコール制御に基づく仮想マシン間サンドボックスシステム，情報処理学会論文誌， Vol.2, No.1, pp.33–52 (2009).
19) 黒田滋樹，柴山悦哉：Software Fault Injection を用いた開発時テスト支援環境，コンピュータソフトウェア， Vol.23, No.4, pp.82–86 (2006).