SOSP’05 報告

金田憲二

平成 17 年 11 月 1 日

1 概要

日時・場所

2005 年 10 月 23 日から 26 日まで，英国ブライトンで開催されました．

採択論文

155 本の論文の投稿があり，その中から 20 本の論文が採択されました（採択率は約 13 %）．

セッションは，(1) Integrity and Isolation，(2) Distributed Systems，(3) History and Context，(4) Containment，(5) Filesystems，(6) Bugs，(7) Optimization の 7 つから成ります．

本会議に採択された論文中で，以下の 5 本が TOCS への投稿へと進みました．

- Labels and Event Processes in the Asbestos Operating System *(Integrity and Isolation)*
- BAR Fault Tolerance for Cooperative Services *(Distributed Systems)*
- Vigliante: End-to-End Containment of Internet Worms *(Containment)*
- Speculative Execution in a Distributed File System *(Filesystems)*
- Rx: Treating Bugs As Allergies — A Safe Method to Survive Software Failures *(Bugs)*

Peer-to-Peer 関連の論文は以下の 2 本でした．

- BAR Fault Tolerance for Cooperative Services
- Implementing Declarative Overlays

VM 関連の論文は以下の 4 本でした．

- Detecting Past and Present Intrusions through Vulnerability-Specific Predicates
- Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm
- THINC: A Virtual Display Architecture for Thin-Client Computing

- Vigliante: End-to-End Containment of Internet Worms

採択論文全体の傾向としては，大山さんもおっしゃっていましたが，何か特定の流行りがあるというわけではなく，何らかの意味でシステムの安全性に関わる論文が多いという程度のものでした．

ちなみに，採択された論文は，ほとんど全てが USA の大学・企業のものでした．USA 以外では，ヨーロッパからは University of Cambridge と NEC Europe Ltd. の論文が 2 本あり，カナダからは University of Tronto の論文が 1 本ありました．日本からは，ポスターでの発表が 2 件と，WIPs での発表が 1 件でした．

出席者

会議への出席者は 400 名弱で，ヨーロッパからの参加者は 100 名程度だそうです．僕が気づいた有名人としては，Marc shapiro 先生などがいらっしゃいました．

会議参加者全体の中で，学生の数は 100 名程度だそうです．企業から参加している方は，Microsoft，IBM，HP，Google，Amazon，VMware などでした．

日本からの参加者は，中島達夫先生 (早稲田大)，佐藤一郎先生 (NII)，光来先生 (東工大)，吉瀬先生 (電通大) などと学生を含めて，約 10 名程度でした．

その他

- 会議の雰囲気は全体的に良かったと思いました．多くの人が真面目に発表を聴いており，質問も多く出ていました．その理由としては，会議中のラップトップ PC の利用が禁止であったり，プレゼンテーションがうまく，理論系ほど話が難しくないので，発表についていきやすかったことが挙げられるのではないかと思いました．

  人数の割には会場が狭いせいもあって，顔を知らない人同士でも話をする機会が比較的多くありました（学生やポスドクの間なら）．

- 今回でちょうど 20 回目の開催で，しかもヨーロッパでの 2 回目の開催ということもあってか，SOSP の運営を今後どうしていくのかという話が色々と挙がっていました．例えば，blind submission (査読の際に，著者や査読者の名前を隠すこと) に意味があるのかとか，OSDI と合同してしまって 3 年に 1 度はヨーロッパで開催するようにしたらどうか，といった議論がありました．

- ヨーロッパから採択された論文が少ないのは問題だという話が何度か出ていました．とくに今回の会議では，論文の投稿それ自体も，ヨーロッパからのものは少なかったようです．どうせ採択されないからということで，投稿が敬遠される動きがあるのかも知れません．

- InfoSys というインドのソフトウェアサービスの会社がスポンサーに加わっていました．

Keynote Address:

Andy Tanenbaum (Vrije Universiteit)

近年ソフトウェアの大規模化・複雑化が進む中で，どうやってソフトウェアの安全性を保証していくのかという話．

話のおおまかな流れとしては，以下のようになっていました．

1. 近年，ソフトウェアが大規模化が進んでいる（ムーアの法則に従うかのように）．例えば，Windows のソースコードを考えてみると，何百万行となってしまっている．
2. その結果，コンピュータは不安定で，非常に扱いにくいものとなってしまっている．ブートにもやたらと時間がかかる．実際，何かの調査の結果によると，多くの PC の購入者は（Computer Science で学位をとった人でさえ），自分の思ったとおりに PC を動かすことができず，イライラした経験があるそうです．
3. 今後は，PC もテレビぐらいに楽に操作できて安定していることが重要．
4. そこで MINIX-3．MINIX-3 は，マイクロカーネルである MINIX の最新バージョン．カーネルはプロセススケジューリングなどのコアな機能だけを扱っており．ファイルシステムやデバイスドライバはユーザプロセスとなっている．

   カーネルのコードは数千行におさまっている．

質問では，cyclone のような安全な言語を使って OS を記述するのはどうかというのがあり，興味があるとか何とか答えていました．

Session 1: Integrity and Isolation

Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution

A. Seshadri, M. Luk, E. Shi, A. Perrig (Carnegie Mellon University), L. van Doorn (IBM), P. Khosla (Carnegie Mellon University)

信頼できないホスト（実行コードやレジスタ・メモリの状態が改竄される恐れのあるホスト）上で，コードが改竄されずに実行されることを保証する枠組み Pioneer の提案．Terra などと異なり，ハードウェアの支援無しに実現を目指す software-based attestation というアプローチをとる．この Pioneer 上で，実際に kernel rootkit detector を実装した．

Pioneer の動作について概説する．Pioneer は，dispatcher という external trusted entity と，untrusted platform との間の challenge-response protocol に基づく．まず，dispatcher は untrusted platform 上に，dynamic root of trust と呼ばれる TCB を確立する．そして，その dynamic root of trust の元で実行コードの integrity を測り，その後，実際にコードを実行する．

より詳細には，Pioneer は以下の手順で動作する（図 1 と図 2 参照）．

1. dispatcher は，random nonce を含んだ challenge を untrusted platform に送信する．

2. Untrusted platform 上で，verification function と呼ばれる，自分自身の checksum を計算する関数を実行する．それと同時に，コードを改竄されることなく実行できる環境を構築する．例えば，マスカブル割り込みが無効にされ，割り込みハンドラも独自のものに置き換えられた，最も高い特権レベルでプログラムを実行できる環境を構築する．

3. Untrusted platform 上で計算した checksum を dispatcher に返信する．

4. dispatcher は，verifiable function の copy を元に checksum を計算し，untrusted platform から送信されてきたものと比較する．

   checksum の値が異なる場合には，verifiable function が改竄されたとみなす．また，untrusted platform が checksum の計算にかかった時間が，ある定められた範囲を超えた場合も，verifiable function が改竄されたとみなす（詳しくは後述）．

5. Untrusted platform では，checksum 送信後，実行コードのハッシュ値を SHA1 で計算し，その後，実際にコードを実行する．

6. dispatcher も実行コードの copy を元にハッシュ値を計算し，コードの integrity を調べる．

7. もし必要であれば，untrusted platform 上で得られた実行結果を dispatcher に返信する．

上述の説明では，checksum の計算が一定時間内に完了する場合には verifiable function が改竄されていないとみなし，checksum の計算に一定時間以上かかる場合には，verifiable function が改竄されているとみなした．これが正しく動作することを保証するために，Pioneer では，まず，dispatcher が untrusted platform のハードウェア設定について正確に把握していること（checksum の計算にかかる時間を知っている）と仮定している．また，アタッカーが checksum の計算を操作しようとすると，その計算にかかる時間が増大するように色々と細かな工夫をしている．

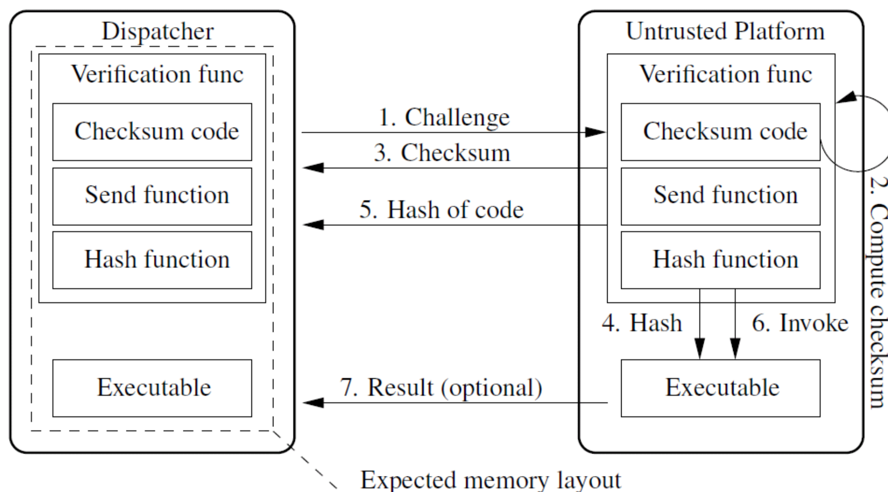

図 1: Overview of Pioneer. The numbers represent the temporal ordering of events.

1.	$D:$	$t_1 \leftarrow$ current time, $nonce \xleftarrow{R} \{0,1\}^n$
	$D \rightarrow P:$	$\langle nonce \rangle$
2.	$P:$	$c \leftarrow \text{Checksum}(nonce, P)$
3.	$P \rightarrow D:$	$\langle c \rangle$
	$D:$	$t_2 \leftarrow$ current time
		if $(t_2 - t_1 > \Delta t)$ then exit with failure
		else verify checksum c
4.	$P:$	$h \leftarrow \text{Hash}(nonce, E)$
5.	$P \rightarrow D:$	$\langle h \rangle$
	$D:$	verify measurement result h
6.	$P:$	transfer control to E
7.	$E \rightarrow D:$	$\langle$result (optional)$\rangle$

図 2: The Pioneer protocol. The numbering of events is the same as Figure 1. D is the dispatcher, P the verification function, and E is the executable

.

Labels and Event Processes in the Asbestos Operating System

P. Efstathopoulos (University of California at Los Angeles), M. Krohn (Massachusetts Institute of Technology), S. VanDeBogart (University of California at Los Angeles), C. Frey, D.Ziegler (Massachusetts Institute of Technology), D. Mazieres (New York University at Stanford), F. Kaashoek, R. Morris (Massachusetts Institute of Technology)

プロセス間通信や情報流を制御することのできるラベル機能をもつ Asbestos OS の提案．この Asbestos は，既存のラベル機能をもつ OS と比較して，decentralized compartments [1] をサポートしているなど，より柔軟にラベルを制御できる点が新しい（論文中には，mandatory access control と discretionary access control を組み合わせと記述されていた）．実際にこの Asbestos 上で，ユーザごとにデータを隔離する Web サーバを記述した．

まず Asbestos の概要について説明する．Asbestos の IPC は，Mach などのマイクロカーネルに似たもの．プロセスは，**ポート**を介してメッセージの送受信をすることで，通信を行なう．ただし，それぞれのポートに対してメッセージを送受信できる権利というものが定められている．例えば，初期状態では，ポートを生成したプロセスのみ，そのポートからメッセージを受信する権利を持つ．送信する権利は，後述するラベルチェックによって定まる．

次に，Asbestos のラベル機能について説明する．情報流のラベルは lattice を構成する．$\sqsubseteq$ はラベルの半順序関係を，$\sqcup$ は least-upper-bound operator を，$\sqcap$ は greatest-lower-bound operator を表す．

個々のプロセス P は，以下の 2 つのラベルを持つ．

送信ラベル P_s で表される．P の現在の contamination（P から送信されるメッセージの機密度の高さ）を表す．

受信ラベル P_r で表される．他のプロセスから受信可能な最大 contamination （どれだけ機密度の高いメッセージを P が受信できるか）を表す．

以下の条件が満たされるとき，P は Q へとメッセージを送信可能である．

$$P_s \sqsubseteq Q_r$$

Q にメッセージが配送されると，Q の送信ラベルは P の送信ラベルに汚染される（P から受信した結果を，Q は送信する可能性があるので）．

$$Q_s \leftarrow Q_s \sqcup P_s$$

このラベルは，より詳細には，ハンドラ (compartment を名前付ける 61-bit number) から特権レベル (ordered set $[*, \mathbf{1}, \mathbf{2}, \mathbf{3}]$) [2]への関数として定義される．例えば，ハンドラ h_1 の特権レベルが $\mathbf{0}$，ハンドラ h_2 の特権レベルが $\mathbf{1}$，それ以外のハンドラのデフォルトの特権レベルが $\mathbf{2}$ であるラベルは，$\{h_1\mathbf{0}, h_2\mathbf{1}, \mathbf{2}\}$ と表される．そして，$\sqsubseteq$，$\sqcup$，$\sqcap$ は，ぞれぞれ以下のように定義される．

$$\begin{aligned} L_1 \sqsubseteq L_2 \quad &\text{iff} \quad L_1(h) \leq L_2(h) \text{ for all } h. \\ (L_1 \sqcup L_2)(h) \quad &= \quad \max(L_1(h), L_2(h)) \\ (L_1 \sqcap L_2)(h) \quad &= \quad \min(L_1(h), L_2(h)) \end{aligned}$$

[1]各プロセスが compartment と呼ばれるラベルを管理する単位を動的に生成できること．compartment は例えばユーザに対応し，各ユーザごとに別々の compartment が用意される．

[2]$*$ が最も低い特権レベルで，$\mathbf{3}$ が最も高い特権レベル

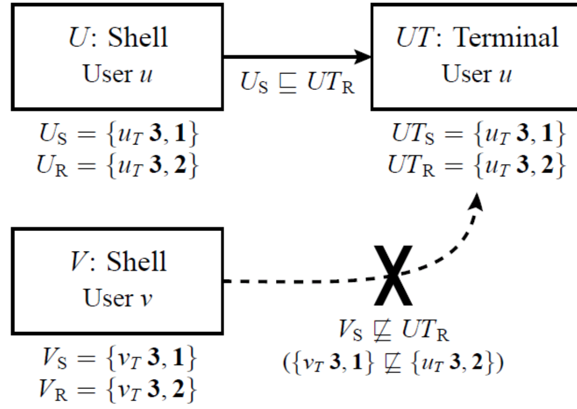

図 3: Simplified process communication with labels.

P, Q	Processes
p, h	Ports, handles
$\star, \mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}$	Label levels, in increasing order
L, C, D, V, E	Labels (functions from handles to levels)
P_S	Process P's send label
P_R	Process P's receive label
p_R	Port p's receive label
$L_1 \sqsubseteq L_2$	Label comparison: true iff $\forall h, L_1(h) \leq L_2(h)$
$L_1 \sqcup L_2$	Least-upper-bound label: $(L_1 \sqcup L_2)(h) = \max(L_1(h), L_2(h))$
$L_1 \sqcap L_2$	Greatest-lower-bound label: $(L_1 \sqcap L_2)(h) = \min(L_1(h), L_2(h))$
$L^\star$	Stars-only label: $L^\star(h) = \begin{cases} \star & \text{if } L(h) = \star, \\ \mathbf{3} & \text{otherwise} \end{cases}$

図 4: Notation.

具体例として，図 3 に示されるプロセス間通信を考える．

ユーザ u と v のそれぞれのシェル U，V と，ユーザ u のログインしている端末 UT がある．各ユーザ u と v に，taint handler u_T と v_T が割り当てられている．$U_s \sqsubseteq UT_r$ が成り立つので，U は UT にメッセージを送信できるが，$V_s(v_T) > UT_r(v_T)$ より，V は UT にメッセージを送信できない．

ただし，これまでに述べてきたラベル機能だけ扱えるものには限界があり，例えば，複数のユーザの情報を管理するファイルサーバを実装する際に，各ユーザごとにデータを隔離することができない．そこで，実際の論文では，これ以降にも，integrity の保証などのために，さらにラベルの操作が拡張され，複雑なものになっている（例えば，メッセージごとに別々の送信ラベルを付けることを可能にしたり，declassification privilege を導入したり）．最終的なラベルに関するオペレーションをまとめたものを，図 5 に示す．

send$(p, \text{data}, C_S, D_S, V, D_R)$ // Send message to port p
Let Q be the process with receive rights for p
Let $E_S = P_S \sqcup C_S$
Requirements:
(1) $E_S \sqsubseteq (Q_R \sqcup D_R) \sqcap V \sqcap p_R$
(2) If $D_S(h) < \mathbf{3}$, then $P_S(h) = \star$
(3) If $D_R(h) > \star$, then $P_S(h) = \star$
(4) $D_R \sqsubseteq p_R$
Effects:
Grant D_S and contaminate with E_S, but preserve Q_S's $\star$ handles
$Q_S \leftarrow (Q_S \sqcap D_S) \sqcup (E_S \sqcap Q_S^{\star})$
$Q_R \leftarrow Q_R \sqcup D_R$

new_port(L)
Let p be an unused port
Effects:
$p_R \leftarrow L$
$p_R(p) \leftarrow \mathbf{0}$
$P_S(p) \leftarrow \star$
Return p

set_port_label(p, L)
Requirement:
P has receive rights for p
Effect:
$p_R \leftarrow L$

図 5: Label operations associated with three Asbestos system calls. P is the calling process.

こういった研究でよく言われることですが，ユーザが適切にラベルを設定するのは，やはり難しい気がしました．

Mondrix: Memory Isolation for Linux using Mondriaan Memory Protection

E. Witchel (University of Texas at Austin), J. Rhee (Purdue University), K. Asanovic (Massachusetts Institute of Technology)

Mondriaan Memory Protection を Linux kernel へ応用したという研究．

発表者が,「Modriaan というと，また昔の論文の焼き直しかという反応があるが，毎回きちんと長い時間をかけて差分を作っている」と主張していました．

Session 2: Distributed Systems

BAR Fault Tolerance for Cooperative Services

A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, C. Porth (University of Texas at Austin),

故障をモデル化するにあたって，プロセスが Byzantine behavior（仕様からそれた任意の行動）をとるだけでなく，rational behavior (仕様からそれた自己中心的な行動) をとったりすることも考慮することが，重要になってくる．

そこで，BAR (Byzantine, Altruistic，Rational) モデルというのを導入し，そのモデルの元で耐故障性を実現するバックアップサービスを実装した．このサービスのアーキテクチャーは 3 層から成り，最下層で，state machine の複製や terminating reliable broadcast を実現する．中間層で，各ノードへの仕事の割り振りを実現する．最上層で，特定のアプリケーションに応じた機能を実現する．

Fault-Scalable Byzantine Fault-Tolerance Services

M. Abd-El-Malek, G. R. Ganger (Carnegie Mellon University), G. R. Goodson (Network Appliance, Inc.), M. K. Reiter, J. J. Wylie (Carnegie Mellon University)

(TODO)

Implementing Declarative Overlays

B. T. Loo, T. Condie (University of California at Berkley), J. M. Hellerstein (Intel Research Berkley and University of California at Berkley), P. Maniatis, T. Roscoe (Intel Research Berkley), I. Stoica (University of California at Berkley)

Peer-to-Peer システムを記述するための汎用的な枠組み P2 の提案．Overlog と呼ばれる， Prolog や Datalog に似た declarative logical language を用いて，プロトコルを記述する．実際にこのシステム上で，Chord などの Peer-to-Peer システムを記述した．

Overlog は基本的には，table declaration statements と rules からなる．

いくつか例を挙げると，例えば，

```
materialize(neighbor, 120, infinity, keys(2))
```

という宣言で，`neighbor` という名前の，120 秒間保持される，無限サイズ長の tuple からなるテーブルを指定する．`keys(...)` construct は，tuple の field で primary key となるものを指定する．

また，membership 情報を隣人に定期的にブロードキャストするプロトコルは，以下のようなルール `R1` によって記述することができる．

```
R1 refreshEvent(X) :- periodic(X, E, 3).
```

`periodic` というのは built-in term で，定期的に（この場合 3 秒ごとに）ノード `X` において，一意に定まる識別子 `E` をもつ tuple を生成する．

同様に，自分の sequence number を増やしながら隣人を refresh するプロトコルは，以下のように記述することができる．

```
materialize(sequence, infinity, 1, keys(2)).
R2 refreshSeq(X, NewSeq) :- refreshEvent(X),
                            sequence(X, Seq),
                            NewSeq := Seq +1.
R3 sequence(X, NewS) :- refreshSeq(X, NewS).
```

`sequence` は，expire しない single entry からなる表．ノード `X` において `refreshEvent` が発行される度に，現在 表 `sequence` に格納されている sequence number `Seq` を 1 増やした値が， `newSeq` に代入される．

そして，*location specifier* という，ある特定ノードにおいて tuple が存在するかを指定する annotation を導入する．`pred@loc(...)` というシンタックスになっており，例えば以下のように記述する．

```
materialize(member, 120, infinity, keys(2))
R4 member@Y(Y, A, ASeqX, TimeY, ALiveX) :- refreshSeq@X(X, S),
                                           member@X(X, A, ASeqX, _, AliveX),
                                           neighbor@X(X, Y),
                                           not member@Y(Y, A, _, _, _)
                                           TimeY := f_now@Y()
```

Session 3: History and Context

Detecting Past and Present Intrusions through Vulnerability-Specific Predicates

A. Joshi, S. T. King, G. W. Dunlap, P. M. Chen (University of Michigan)

現実世界では，security flow が発生してから 実際が patch が適応されるまでに長い時間を必要とし，その間にアタックを受ける危険がある．そこで，IntroVirt というシステムを提案する．このシステムは，仮想マシンを用いることで過去の実行を再現し，この再実行中にユーザから与えられた predicate を実行することで，脆弱性のあるコードが過去に実行されたかどうかを検査する．

predicate 中で，メモリのアドレスなどの代わりに，プログラム中の変数を直接指定できるようにするなどの工夫をしている．

Capturing, Indexing, Clustering, and Retrieving System History

L. Cohen (Hewlett-Packard Laboratories), S. Zhang (Stanford University), M. Goldszmidt, J. Symons, T. Kelly (Hewlett-Packard Laboratories), A. Fox (Stanford University)

(TODO)

異常時発生時のログを元に学習を行ない，同様の異常が発生した他の箇所を認識する．

Connections: Using Context to Enhance File Search

C. A. N. Soules, G. R. Ganger (Carnegie Mellon University)

Google Desktop のような，デスクトップ検索の話．Web 検索と違って，デスクトップ上のファイルはリンク構造をもたないため，アクセス履歴をもとにファイル間の関係を類推する．

他の研究と比較して，1 人で作れるくらいの小規模なシステム．SOSP とあまり relevant でない気もしました．

Poster Session

全部で 31 件のポスタープレゼンテーションがあった．中身は，ピンからキリまでといった感じで，結構怪しい話も多かった．THINC や Nexus OS はデモをしていた．

ポスター発表のタイトルの一部を挙げると以下の通り．

- Recovery Oritented Programming
- Operating System Construction in Haskell
- Towards Scalable and Simple Software-DSM Systems
- Proactive Operating System Recovery
- Using Model Checker and Replay Facility to Debug Complex Distributed System
- Pre-virtualization: Uniting two Worlds

Session 4: Containment

Vigliante: End-to-End Containment of Internet Worms

M. Costa (University of Cambridge and Microsoft Research), J. Crowcroft (University of Cambridge), M. Castro, A. Rowstron, L. Zhou, L. Zhang, P. Barham (Microsoft Research)

ホスト上で実際にプログラムを走らせることで脆弱性に関する情報を取得し，その情報をブロードキャストすることで worm の封じ込めを行なう Vigliante というシステムを提案している．

より詳細には，Vigliante は，脆弱性に関する情報を生成する detector host と，その情報を元にメッセージをフィルタリングする vulnerable host から構成され，以下のように動作する（図 6 参照）．

1. detection engine が，受信メッセージを元に worm の侵入を検知する．
2. 侵入検知時に得られた情報から，self-certifying alerts (SCAs) と呼ばれる，脆弱性に関する情報を自動生成する．SCA 中には，どんなサービスに対して，どんなメッセージを送ると，どんな危険があるかといった情報が含まれる（図 7 参照）．
3. SCA が本当に正しいものかどうかを検証する．サンドボックス上で，SCA に含まれた情報を元に侵入動作を再現することで検証を行う．
4. detector host が SCA をブロードキャストする．
5. SCA を受け取ったホストは，実際にそれが正しいものか検証する．
6. SCA を元に，メッセージをフィルタリングする．

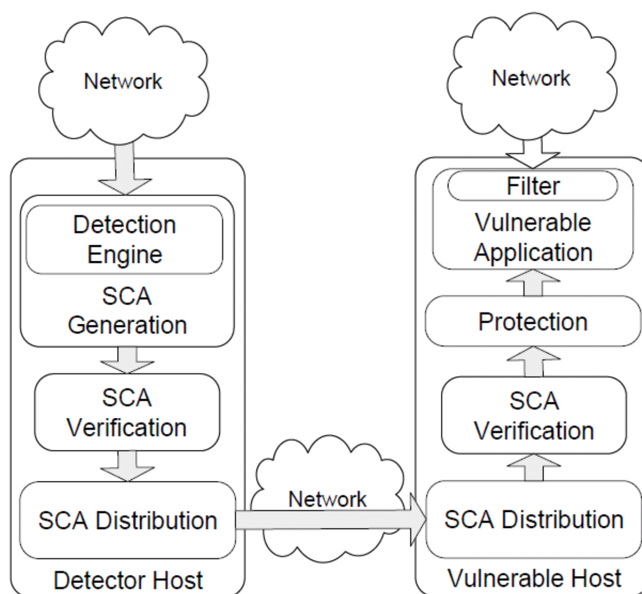

図 6: Automatic worm containment in Vigilante.

Service: Microsoft SQL Server 8.00.194
Alert type: Arbitrary Execution Control
Verification Information: Address offset 97 of message 0
Number messages: 1
Message: 0 to endpoint UDP:1434
Message data: 04,41,41,41,41,42,42,42,42,43,43,43,43,44,44,44,44,45,45,45,
45,46,46,46,46,47,47,47,47,48,48,48,48,49,49,49,49,4A,4A,4A,4A,4B,4B,4B,4B,
4C,4C,4C,4C,4D,4D,4D,4D,4E,4E,4E,4E,4F,4F,4F,4F,50,50,50,50,51,51,51,51,
52,52,52,52,53,53,53,53,54,54,54,54,55,55,55,55,56,56,56,56,57,57,57,57,58,58,
58,58,0A,10,11,61,EB,0E,41,42,43,44,45,46,01,70,AE,42,01,70,AE,42,......

図 7: An example arbitrary execution control SCA for the Slammer worm. The alert is 457-bytes long and has been reformed to make it human readable. The enclosed message is 376-bytes long and has been truncated.

以下では，それぞれのフェーズについてより詳細に説明する．

SCA をどう生成するかについて説明する．まず，detector host は受信したメッセージのログをとっておく．そして，侵入を検知したら，それの原因となっているメッセージを特定し，そのメッセージを含む SCA の候補を生成する．侵入の原因となっているメッセージを特定する方法としては，以下の 2 つを提案している．

- ページを実行禁止にする方式．オーバヘッドは小さいが，精度が悪い．まず，実行禁止ページを実行しようとする際に発生する例外を捕捉する．そして，メッセージログを最近受信したものから順にたどっていき，実行しようとしていたコードを含むメッセージを探し，そのメッセージを元に SCA を生成する．
- データフローを解析する方式．オーバヘッドは大きいが，精度は良い．まず，ネットワークから受信したメッセージ（もしくは，それに依存したデータ）を実行しようとしていた際には，脆弱性があるとみなす．そして，実行しようとしていたコードを含むメッセージを元に SCA を生成する．

例えば，図 8 は，スタックオーバーフローを引き起こすコードを表し，図 9 は，コード実行前と実行後のスタックの状態を表す．この場合，detector engine は，`ret` 命令が実行された時点で侵入されたとみなす．

得られた SCA の候補を，サンドボックス上で侵入検知を再現することで検証する．実行コードをバイナリ変換することで，サンドボックスは実現されている．

得られた SCA をどうブロードキャストするかについて説明する．worm が広がるまえに各ホストに SCA を届けるために，階層的 Peer-to-Peer オーバレイネットワーク上で SCA をブロードキャストする．

SCA を元にメッセージのフィルタをどう生成するかについて説明する．SCA 中に含まれる情報を元に侵入を再び実行し，その時にどういった条件分岐をたどったかを記録する．そして，外部からメッセージを受信した際には，同じ条件分岐をたどるかによって，worm かどうかを判定する．

```
mov al,byte ptr [msg]          //move first byte to AL
add al,0x10                    //add 0x10 to AL
mov cl,0x31                    //move 0x31 into CL
cmp al,cl                      //compare AL to CL
jne out                        //jump if not equal
mov cl,byte ptr [msg]          //move first byte to CL
xor eax,eax                    //move 0x0 into EAX
loop:
mov byte ptr [esp+eax+4],cl    //move byte into buffer
mov cl,byte ptr [eax+msg+1]    //move next byte to CL
inc eax                        //increment EAX
test cl,cl                     //test if CL equals 0x0
jne loop                       //jump if not equal
out:
mov esp,ebp
ret
```

図 8: Vulnerable code.

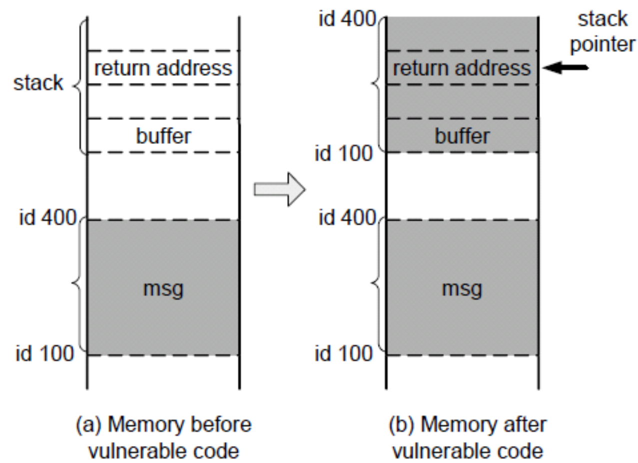

図 9: Example of SCA generation with dynamic dataflow analysis. The figure shows the memory when (a) a message is received and the vulnerable code is about to execute, and (b) after the vulnerable code executes and overwrites the return address in the stack. Grayed areas indicate dirty memory regions and the identifiers of dirty data are shown on the left.

Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm

M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, S. Savage (University of California at San Diego)

数十万個の IP をもち，かつ native code を実行できる honeyfarm (a large network of honeypot systems) を実現するシステム．実際に，小数の物理マシン上で 64,000 台の honeypot をエミュレーションした．

この研究のとるアプローチとしては，リクエストを処理するのに十分な短期間のみ，仮想マシンを各物理マシンに割り当てるようにすることで，スケーラビリティを確保している．より具体的には，ネットワークパケットが到達したときに，特殊なゲートウェイルーターが，IP アドレスを動的に他の物理マシンに割り当てる．そして，その物理マシン上で生成された新たな仮想マシンが，パケットを受信する．

仮想マシンモニタには Xen を利用しており，多くの仮想マシンを生成・管理することのオーバヘッドを削減するために，flash cloning (host referencing image を複製・改変することで新しい VM を高速に生成する技術) と delta virtualization (copy-on-write を利用した，高速に VM を複製する技術) を導入している．

質問としては，Potemkin むけの対策をした新しいアタック（例えば，長時間メッセージを送り続けるアタック）を作成できるのではないかというものが挙がっていました．また，大量のマシンから取得したログを，効率良く解析できるのかという意見もありました．

The Taser Intrusion Recovery System

A. Goel, K. Po, K. Farhadi, Z. Li, E. de Lara (University of Tronto)

ウィルスなどに侵されたコンピュータ上で，ウィルスに侵されていないファイルのみ復旧させるシステム．非常に簡単にシステムの動作の流れについて説明すると，以下の通り．

1. ファイルへの読み書きをログにとっておく．
2. 何らかの intrusion detection system を使って，あるファイル (X) がウィルスに侵されたことを検出する．
3. ログを元に，プロセスとファイル（ソケット）との間の依存関係を計算する．
4. X と依存関係の**ない**ファイルのみ復旧する．

質問としては，ファイルへの読み書きだけを元に依存関係を計算しているので，色々と捉えきれていない依存関係があるのではないかという話でした（例えば共有ライブラリのメモリへのマップなどとか）．

Pannel: Peer-to-Peer: Still Useless?

これまでの Peer-to-Peer の研究に意味があったのかとか，どんなキラーアプリケーションがあるのかなどについて議論していた．

いくつか記憶に残ったものを挙げると，以下の通り．

- キラーアプリケーションとしては，分散 DNS などが挙がっていた．
- 新しいルーティングアルゴリズムを提案して，既存のものと性能比較をするという研究は多いが，Akamai などの実際に動いているシステムと比較をしているものは少ないなどという意見もあった．
- 実際に利用者が多いと言う点では useful という意見もあった．
- 学生の教育という観点からは意味があったという意見もあった．

Session 5: Filesystems

Hibernator: Helping Disk Arrays Sleep through the Winter

Q. Zhu, Z. Chen, L. Tan, Y. Zhou (University of Illinois at Urbana-Champaign), K. Keeton, J. Wilers (Hewlett-Packard Laboratories)

(TODO)

Speculative Execution in a Distributed File System

E. B. Nightingale, P. M. Chen, J. Flinn (University of Michigan)

NFS などの分散ファイルシステムで，ファイルアクセスを投機的に行なうことで性能向上を図るという研究．投機実行をする前にはチェックポイントを取得し，予測した結果と実際に得られた結果が異なる場合には，roll-back する．一つのファイルに対して複数のプロセスが投機実行をしている際には，そのプロセス間の依存関係も考慮して，roll-back を行う．

実際にベンチマークプログラムを走らせたところ，2 倍近くの性能向上が見られた．

IRON File Systems

V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau (University of Wisconsin at Madison)

(TODO)

Work-in-Progress Session

各5分の発表が15件あった．応募自体は25件ぐらいあったと言っていたと思う．
そのうちのいくつかを挙げると，以下の通り．

- Improving Dynamic Update for Operating Systems
- INSIGIT: A Distributed Monitoring System for Tracking Continuous Queries
- Nexus: a New Operating System for Trustworthy Computing
- ExtraVirt: Detecting and Recovering from Transient Processor Faults

最後の論文は，University of Michigan のもので，耐故障性機能をもつ VM を提案していた．動作の概略を述べると以下の通り．まず，マルチプロセッサマシン上で，同じ VM を複数同時に走らせておく．そして，その VM のうち一つだけ異なる状態になった場合，ハードウェアが一時的に故障をしたとみなし，故障から復旧をする．

Session 6: Bugs

RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking

Y. Yu, T. Rodeheffer (Microsoft Research), W. Chen (University of California at Barkeley)

CLR (Common Language Runtime) 上で動作するプログラム中に race condition があるかどうかを動的に判定する手法について．特徴としては，lockset と threadset というものを動的に求めることで race condition を検出している点と，race condition 検査の精度を adaptive に変化させることで高速化を図っている点.

まず，どういったアクセスを，並行に行なわれたものとみなすかについて説明する．あるスレッド T_1 のロケーション x へのアクセスが，別のスレッド T_2 のロケーション x へのアクセスと並行に行なわれたとみなされるのは，以下の 2 つの条件が共に満たされる場合.

- T_1 と T_2 が共通に保持しているロックが存在しない.
- T_1 と T_2 のアクセスがお互いに happens-before 関係[3]になっていない.

そこで，この 2 つの条件が成り立つかどうかを動的に計算することで，race condition の検出を行う．この 2 つの条件が成り立つかどうかを検査するために，それぞれ lockset と threadset という集合を（各変数ごとに）求める．簡単に言ってしまうと，ある変数の lockset は現在その変数にアクセスしているスレッドが保持している共通のロックの集合を表し，threadset は現在その変数にアクセスしているスレッドの集合を表す.「現在」というのは，各スレッドが vector clock を利用して happens-before 関係を計算することで求める.

そして，ある変数に対して以下の条件が共に満たされた場合に，race condition が発生したとみなす.

- threadset の要素数が 1 より大きい（複数のスレッドが同時にアクセスしている）.
- lockset が空.

また adaptive に検査を変化させることで，高速化を実現している．例えば，

- 毎回 threadset を計算しているとオーバヘッドが大きいので，最初は lockset のみを使って race condition を計算し，それによって race condition の発生する可能性があった場合のみ，threadset を計算する.
- 最初はオブジェクト単位で race condition を計算し，それによって race condition の発生する可能性があった場合のみ，そのオブジェクトのフィールド単位で race condition を計算する.

[3]transitive に定義される，イベント間の順序関係．例えば，親スレッドの `fork` と，子スレッドのメモリアクセスとの間には， happens-before 関係が成り立つ．逆に，親スレッドの `fork` 後に行なったメモリアクセスと，子スレッドのメモリアクセスとの間には，happens-before 関係は成り立たない.

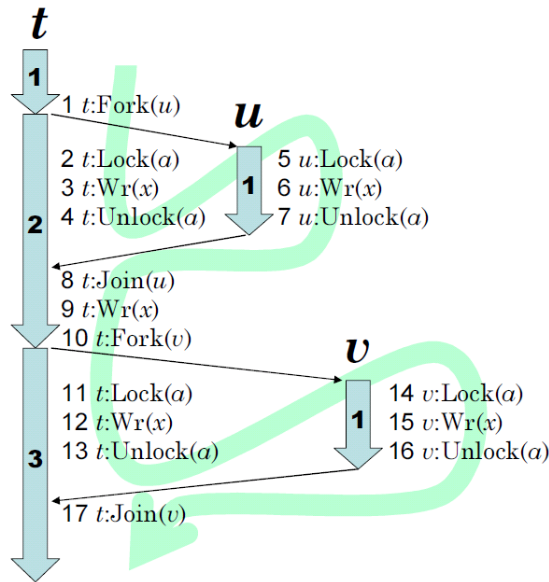

図 10: An example for the basic algorithm. Fat arrows show thread execution and skinny arrows show synchronization. Thread clock is indicated inside the fat arrow. Instruction numbers indicate sequential order of execution of an example interleaving.

Rx: Treating Bugs As Allergies — A Safe Method to Survive Software Failures

F. Qin, J. Tucek, J. Sundaresan, Y. Zhou (University of Illinois at Urbana Champaign)

バグが発生した場合に，実行環境を変えて際実行するという研究．実行環境を変えることで，deterministic に発生するバグを減らす．ただ，単純に reply してしまうと，signature とか replay 時に保存しておかなければいけないものがあるとき大変だと言っていた．

Session 7: Optimization

Idletime Scheduling with Preemption Intervals

L. Eggert (NEC Europe Ltd.), J. D. Touch (University of Southern California)

(TODO)

FS2: Dynamic Data Replication in Free Disk Space for Improving Disk Performance and Energy Consumption

H. Huang, W. Hung, K. G. Shin (The University of Michigan)

使われていないディスクスペースにファイルの複製などを置くことによって，ハードディスクの seek time や rotational time を削減しようという論文.

怪しいといえば怪しいけれど，新規性があるといえばあるという論文.

THINC: A Virtual Display Architecture for Thin-Client Computing

R. A. Baratto, L. N. Kim, J. Nieh (Columbia University)

VM を使って rdesktop や X window のようなリモートデスクトップ環境を実現しようという話. データの転送は，デバイスドライバのレベルで行なっている．動画とか音とかがきちんと配送されるように，色々と最適化を行なっている.

実際に会場でデモをしていた.