論理型言語によるSELinux向け認可判定機構の実装と評価

橋本 正樹 † 滝澤 峰利 † 高山 扶美彦 ‡ 辻 秀典 ‡ 田中 英彦 †

† 情報セキュリティ大学院大学 情報セキュリティ研究科
221-0835 神奈川県横浜市神奈川区鶴屋町 2-14-1
hashimoto@iisec.ac.jp

‡ 株式会社情報技研
103-0024 東京都中央区日本橋小舟町 3-1 クレイドル日本橋 3 階
hide@iit.jp

あらまし SELinux はシステム侵害後の被害拡大を防止できる点で有用であるが，ポリシ記述の簡単化はもとより，ポリシの可読性や保守性，拡張性の向上が課題となっている．本研究は，論理型言語によるポリシ記述・処理系を実装することでそれら課題を解決し，記述範囲の拡張や強制アクセス制御機構の相互運用性向上をはかるものである．本稿では，SELinux の認可判定機構を Datalog で置き換えた上で各種評価を行い，論理型言語によるポリシ処理系が現実的な性能で動作することを実証する．また，強制アクセス制御機構の構成変更に伴う改ざん耐性と迂回困難性の変化を検証し，提案機構の有用性と安全性に対する考察を行った結果を報告する．

Implementation and Evaluation of Logic-based Authorization System for SELinux

Masaki Hashimoto† Minetoshi Takizawa† Fumihiko Takayama‡
Hidenori Tsuji‡ Hidehiko Tanaka†

†Graduate School of Information Security, Institute of Information Security
2-14-1, Tsuruya-cho, Kanagawa-ku, Yokohama-shi, Kanagawa 221-0835, JAPAN
hashimoto@iisec.ac.jp

‡Institute of Information Technology, Inc.
3-1-3 Kohune-cho, Nihonbashi, Chuo-ku, Tokyo 103-0024, JAPAN
hide@iit.jp

Abstract SELinux is an effective MAC system for preventing the damage from spreading after security breaches, and there are many challenges around its policy processing issues such as readability, maintainable and scalability. Our research aims to improve its expressive power of policy description language and interoperability of authorization system, implementing them by logic programming language. In this paper, we replace the authorization system of SELinux with our logic-based authorization system of Datalog, and demonstrate the feasibility based on various performance evaluations. We also report the results of validating the impact of architectural changes and discussion about safety and usefulness of our system.

1 はじめに

1.1 研究の背景と課題

SELinux は強制アクセス制御の Linux に対する実装であり，米国家安全保障局を中心としたオープンソースコミュニティによって活発に開発が進められている．SELinux による強制アクセス制御は，ポリシで定めた権限を，特権プロセスを含む全プロセスにシステムコールレベルの粒度で強制するため，システム管理者は，制御対象のプロセスに細粒度の権限のみを与えることが

可能となり，結果として，システム侵害後の権限昇格等を用いた被害拡大を OS レベルで抑止可能となる．

しかし一方で，細粒度のアクセス制御のためには，その制御規則となるポリシについても細粒度に正しく記述する必要があるが，これは人間にとって非常に困難な仕事であり，強制アクセス制御がその有用性にも関わらず必ずしも効果的に利用されているとは言い難い現状の原因となっている．

1.2 先行研究

著者らは，先行研究 [1] により，論理プログラムとして認可判定規則を表現することで，属性の継承やサブルーチンとして構造化された記述が可能なポリシ記述言語を提案し，1.1 節で示した課題の解決を計った．本言語は，個々の規則を記述するための宣言文と，宣言文の集合として構成されたポリシに問い合わせを行うための認可判定文を，構文規則とその形式的な意味，推論規則を定めることで基本設計を行い，これを Datalog[2] を用いて実装したものである．

この研究では，その有効性を確認するために，本言語を用いて具体的なアクセス制御モデルを構造的に記述する手法を示した上で，SELinux のポリシを実際に本言語で記述した実験システムを構成し，認可判定の妥当性と表現力を評価した．その結果，妥当性の評価実験では，SELinux の認可判定が妥当であることを前提とした時に，本言語による認可判定が論理的に正しい応答を示し，妥当であることを実証した．さらに，表現力の評価実験では，本言語を用いた記述手法が SELinux のポリシを少ない記述量で構成できることを示した．

1.3 本研究の貢献

本研究の貢献は，先行研究で提案したポリシ記述言語を SELinux に接続し，応答アルゴリズムとアクセス制御アーキテクチャの変更によって生じる性能と安全性の変化が，実用上は大きな差異とならないことを実証することである．提案機構は論理型言語による認可判定機構をユーザ空間に構築し，これをカーネル空間にある SELinux と接続する方式であるため，認可判定機構をカーネル空間におき，単純で高速なアルゴリズムで検索して応答する従来の SELinux の方式と比べると，性能と安全性が低下する懸念がある．この懸念を払拭し，先行研究で示したポリシ記述言語の実用性を担保するのが本研究の目的である．

本研究により，ポリシを Datalog による論理プログラムとして構造的に記述し，実際の SELinux に実装可能であることが示され，結果として，数理論理学的な裏付けを自然にポリシ記述に適用できることが期待できるし，加えてその表現力においても，一階述語論理を基礎とした大幅な拡張を期待できる．

1.4 本稿の構成

第 2 章では，本研究の関連研究として，SELinux と，Datalog の拡張実装である XSB について，特にポリシ記述や処理との関係から説明する．次に，第 3 章では，提案機構の基本設計として，アーキテクチャとその構成コンポーネント，認可判定処理の流れについて説明し，提案機構の実装を説明する．第 4 章では，提案機構の評価として，詳細な応答性能を計測した上で改ざん耐性と迂回困難性を検証し，その結果について考察する．最後に，第 5 章で，本稿をまとめる．

2 関連研究

2.1 SELinux と認可判定処理

SELinux は，強制アクセス制御を実現する Flask セキュリティアーキテクチャ[3] の Linux カーネルに対する実装で，セキュリティサーバとオブジェクトマネージャ，アクセスベクタキャッシュ(以降，各々SS/OM/AVC と表記する) から構成される．この実装により，システムコールを横取りすることでアクセス制御に関連するカーネル内制御を奪い，独自のポリシによるアクセス制御を SELinux が強制する．

SELinux の認可判定処理は，アクセス主体のセキュリティ情報，アクセス対象のセキュリティ情報，アクセス対象のセキュリティクラス (以降，これらをまとめてアクセス制御情報と表記する) から，対応する許可操作の集合であるアクセスベクタを計算することで実現される．この計算は，カーネルソース内の SELinux 関連関数である context_struct_compute_av (以降，CAV と表記する) が担っており，この関数は以下のように定義されている．

```
static void context_struct_compute_av(
    struct context *scontext,
    struct context *tcontext,
    u16 tclass,
    struct av_decision *avd)
```

この中の scontext がアクセス主体のセキュリティ情報，tcontext がアクセス対象のセキュリティ情報であ

り，カーネル内で定義されている context 構造体の形式で渡される．tclass はアクセス対象のセキュリティクラスで unsigned short 型である．

CAV は，カーネル空間に保持する連結リストを検索し，アクセス制御情報と一致するエントリのアクセスベクタを av_decision 構造体に格納する．なお，CAV が参照するカーネル空間の連結リストは SELinux におけるポリシの実体であり，ユーザ空間で管理者が記述したポリシソースファイルをコンパイルしてカーネル空間に読み込ませる仕様となっている．従って，SELinux の Trusted Computing Base(以降，TCB と表記する) は，カーネル空間の SELinux 関連コード/データ，ユーザ空間にあるポリシ処理系であり，認可判定処理にかかる計算量は，context の総数 n に対して $O(n)$ である．

2.2 Datalog とその拡張実装 XSB

Datalog は論理プログラミングを基礎とした言語で，特に大規模データベースと情報を交換するために設計されたルールベースの言語である．即ち，データへ直接アクセスするインターフェースを用意してルールベースによる情報の交換をサポートする．また，Datalog は構文的観点から Prolog のサブセットであり，Datalog プログラムは Prolog インタープリタによって構文解析して実行することができる．

XSB[4] は Datalog の一実装で，ISO 標準に準拠した完全な Prolog システムとなっており，その上でさらに，テーブル型述語と非テーブル型述語の統合をサポートする．XSB は Prolog を含む通常の論理プログラミングシステムにはない以下のような特徴を持つ．

- SLG 導出 [5] により整礎的意味論に従った完全な解導出が可能
- 高階論理プログラミング言語 HiLog[6] の実装
- Unification Factoring[7] をはじめとした様々な索引付け制御技術
- 移植性と拡張性のためのソースコード利用可能性

XSB のコンポーネントの多くは SB-Prolog に基づいているが，SLG 導出と HiLog 表現の処理のために Prolog システムと一部異なっている．例えば，Prolog の SLD 導出 [8] は深さ優先探索に基づくため無限ループに陥りやすいが，XSB の SLG 導出はほぼ全ての論理プログラムを正しく評価できる．SLG 導出にかかる計算量は，評価対象のルール数 n と，具体化が必要な項数 k に対して $O(n^k)$ である [9]．

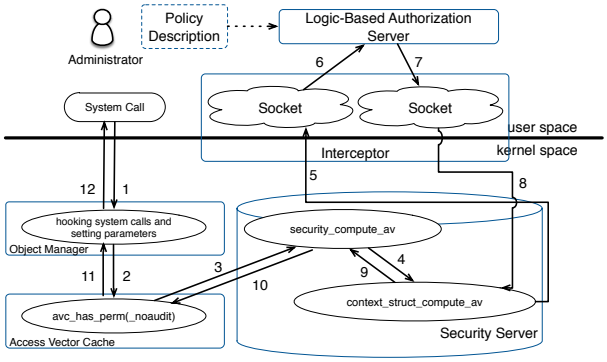

図 1: SELinux と提案機構の構成

これらの特徴により，Datalog やその実装である XSB は，先行研究はもとより SecPAL[10] や Lithium[11] 等をはじめとする，様々なポリシ記述に関する研究の実装プラットフォームとして利用されている．

3 論理型言語によるポリシ処理系

3.1 提案機構の構成

提案機構は，SELinux の認可判定処理を横取りし，ユーザ空間においた認可判定機構で実行するものである．また，これに伴い，ポリシの記述から認可判定機構への実装までを含むポリシ処理系全体を置き換える．

本節では，図 1 に示した提案機構の構成について，コンポーネント間の関係，各コンポーネントの役割/特徴を以下に説明する．図中のオブジェクトマネージャとアクセスベクタキャッシュ，セキュリティサーバは，SELinux オリジナルの構成コンポーネントであり，インターセプタと論理型認可サーバ (以降，各々IC/LAS と表記する) が本研究で追加したコンポーネントである．また，管理者によるポリシ記述とその LAS への実装は，先行研究で提案したものである．

3.1.1 OM/AVC/SS

OM/AVC/SS は，各々，認可判定の強制・執行系，認可判定結果のキャッシュ，認可判定処理系であり，その実体は，Linux のシステムコールに対する横取り処理である．プロセスが強制アクセス制御の対象となる処理を実行すると，呼び出されたシステムコールを OM が横取りし，そのアクセス制御情報を同定して AVC に渡す．AVC は受け取ったアクセス制御情報をキーにキャッシュを検索し，ヒットすれば，対応する認可判定結果を OM に返す．ミスであれば，SS による認可

判定処理が行われ，その結果が AVC にキャッシュされながら OM に渡る．SELinux では，この一連の処理により，ポリシによる認可判定処理を強制する．

3.1.2 IC/LAS

IC は，SELinux の一連の処理から認可判定の部分を横取りし，代わりに認可判定を行う LAS と通信するコンポーネントである．具体的には，AVC から SS にアクセス制御情報が渡った後に実行される SELinux の認可判定用関数を IC の代替関数で置き換え，ソケットを通じてユーザ空間の LAS に渡し，戻ってきた LAS による計算結果を SS による計算結果とする処理を実行する．この際には，必要なデータの整形等も行う．

LAS は，SS の代わりに認可判定処理を行う常駐プロセスで，IC 経由で受け取ったアクセス制御情報を元に実際の認可判定を行うコンポーネントである．受け取るアクセス制御情報は IC によって処理に適した形式に整形済みであり，ポリシは先行研究で提案した方式により，あらかじめ管理者から与えられている．

3.2 認可判定処理の流れ

提案機構における認可判定の流れを以下で説明する．説明中の security identifier(sid) は制御対象の識別子として各システムコールで使う構造体に組み込まれているもので，`ssid`はアクセス主体の sid，`tsid`はアクセス対象の sid を意味する．全ての sid は対応する context に関連付けられており，SELinux の user/role/type/mls range を一意に識別することができる．また，`tclass`は制御内容の識別子としてそれぞれの操作に応じた実装に組み込まれているものである．

STEP 1 OM → AVC → SS SELinux 由来の処理．認可判定に必要な準備をする．
- 1-1 OM が認可判定対象となる `ssid`と `tsid`，`tclass`を同定する．
- 1-2 AVC が `ssid`と `tsid`，`tclass`をキーにキャッシュを検索する．hit した場合は STEP6-2 に分岐する．
- 1-3 SS が `ssid`を `scontext`に，`tsid`を `tcontext`に各々対応付ける．

STEP 2 SS → IC IC がアクセスベクタ計算関数を横取りして LAS の計算に備えたアクセス制御情報を生成し，ソケット経由でユーザ空間に渡す．
- 2-1 SS が SELinux 由来の CAV の代わりに提案システムが用意する代替関数を呼び出す．
- 2-2 IC が `scontext`と `tcontext`，`tclass`を文字列形式に戻し，LAS に渡す．

STEP 3 IC → LAS LAS がアクセス制御情報を元に認可判定処理を実行する．
- 3-1 LAS が SS から渡された `scontext`と `tcontext`，`tclass`を元に認可判定を行い，許可する操作の一覧を文字列形式で決定する．
- 3-2 LAS が許可する操作の一覧を SS 向けのビット形式に変換し．IC に返す．

STEP 4 LAS → IC IC が LAS から受け取った認可判定処理の結果からアクセスベクタを生成する．
- 4-1 IC が LAS から受けとった許可操作一覧のビットをアクセスベクタに代入する．
- 4-2 IC がアクセスベクタを SS に返す．

STEP 5 IC → SS SS が認可判定処理結果を代入したアクセスベクタを受け取り，AVC に返す．

STEP 6 SS → AVC → OM SELinux 由来の処理．AVC がアクセスベクタをキャッシュし，OM が認可判定結果を執行する．

4 評価と考察

4.1 応答性能の評価

第 2 章で説明した通り，SELinux のアクセスベクタ計算は線形時間，XSB による SLG 導出は多項式時間かかることがわかっている．従って，両処理の応答速度には相当な性能差があるはずで，SELinux がシステムコールの粒度で認可判定処理を実行することと合わせて考えると，先行研究で提案した記述言語と，そのSELinux への実装である提案機構が現実的な性能で動作しないことが十分にあり得る．

一方で，SELinux には AVC があり，この有効性次第では，認可判定処理そのものの応答性能差が現実の強制アクセス制御の実行上大きな影響を及ぼさない可能性もあり得るため，本節では，本研究の最初の評価として，これを実際の実験システム上で検証する．

4.1.1 準備

実験システムは，キャッシュサイズ 3072KB の Intel Core 2 Duo CPU P8600(2.40GHz)，667MHz で動作する DDR2 メモリ 1024MB ×2 を実装した計算機上に構築した．OS は，Debian Squeeze(6.0.6, Linux Kernel 2.6.32, i386) に SELinux 関連のパッケージを追加したものを利用する．また，ApacheBench で用いる HTTP Server は，nginx(0.7.67) である．

flush interval	キャッシュをクリアする AVC へのアクセスの回数
try no	計 3 回の測定番号
AVC access nb	AVC にアクセスした回数
miss nb	キャッシュミスの回数 (LAS にアクセスした回数)
miss %	キャッシュミスのパーセンテージ
1t/10t	LAS を動作させるスレッド数
time/access	1 つの AVC へのアクセスの平均時間
penalty/access	理想のアクセス時間 (キャッシュがない場合) との差
total time	全リクエストを処理するためにかかった総時間
penalty/miss	1 つのキャッシュミスにかかった平均時間
total penalty	全キャッシュミスにかかった総時間

表 1: 応答速度計測で使う用語

Process name	Query number	Average speed(μs)
zsh	24	5607
console-kit-dae	18	5419
sshd	232	4183
dbus-daemon	27	5526
nginx	312	7840
memcached	849	9619

表 2: LAS の処理時間

応答性能検証用ソフトウェアとしては，LAS が AVC から認可判定処理要求を受け取る頻度を計測するために，プロセスごとに AVC のキャッシュヒットとミスを測るためのパッチを作成した．また，これを用いた予備実験により，通常の状態ではキャッシュが効き過ぎて，LAS にリクエストを送ることが滅多になく，LAS による処理時間を計測できないことがわかったので，AVC の効果を意図的に低下させるパッチも作成した．ここでは，2^n 回 AVC にアクセスしたらキャッシュをクリアする仕組みとなっている．

4.1.2 LAS 単体での計測結果

4.1.1 で載せたパッチを用いて，LAS 単体での認可判定処理にかかる時間を計測した．その結果を表 2 に示す．Process name は問い合わせ元のプロセス名，Query number は問い合わせ数，Average speed は応答までの平均速度をそれぞれ意味する．

この結果から，実測においては認可判定処理毎に LAS による処理時間が変化するが，4ms から 10ms の間だと見れば良いことがわかった．

Algorithm 1 avc-ab algorithm

```
for n ∈ {0, 8, 9, 10, 11, 12} do          ▷ Cache reset intervals
(n = 0 ⇒ no reset, n > 0 ⇒ reset every 2^n access)
    for thread_num ∈ {1, 10} do
        Turn off cache reset        ▷ echo 0 > /sys/kernel/
security/security_cache_check/cache_reset
        Clear OS cache                                    ▷
echo 3 > /proc/sys/vm/drop_caches
        Clear AVC cache      ▷ echo flush > /sys/kernel/
security/security_cache_check/control
        Warmup               ▷ ab -n 50000 -c 20 HOSTNAME
        Set cache reset interval    ▷ echo n > /sys/kernel/
security/security_cache_check/cache_reset
        loop 3 times
            Reset stats      ▷ echo reset > /sys/kernel/
security/security_cache_check/control
            Run bench with thread_num threads             ▷
ab -n 10000 -c thread_num HOSTNAME
            Collect results
        end loop
    end for
end for
```

	lookups	hits	misses	allocs	reclaims	free
動作前	3147688	3144554	3134	3309	2464	3240
動作後	3689413	3686271	3142	3318	2464	3241
差	541725	541717	8	9	0	1

表 4: avcstat の状況

4.1.3 認可判定処理全体での計測結果

4.1.1 で載せたパッチと ApacheBench を用いて認可判定処理全体の処理時間を計測した．そのために用いた評価アルゴリズムを Algorithm 1 に，応答速度の計測結果を表 3 に示す．また，計測結果で用いる用語は，あらかじめ表 1 に整理する．

この検証は，前述した通り，特定回数 AVC にアクセスした時点でキャッシュをクリアする仕組みとなっており，この結果から，1 つのキャッシュミスが 10ms 程度かかることと，キャッシュの削除頻度がキャッシュミスのコストに影響を与えないことがわかった．

また，AVC によるキャッシュ効果を測るために，キャッシュを削除せずに，`avcstat`コマンドを用いてキャッシュの状況についても調べた．その手順は以下の通りで，表 4 はその結果である．

1. キャッシュをクリアする
2. ApacheBench でウォーミングアップを行う
3. ApacheBench が認可判定処理を 10000 回ずつ 3 セット発行するように動作させる

flush interval	try no	AVC access nb	miss nb	miss %	time/access(ms)		penalty/req(ms)		total time(s)		penalty/miss		total penalty	
					1t	10t	1t	10t	1t	10t	1t	10t	1t	10t
never	try 0	190000	0	0.00	2.41	5.55	0.14	0.42	24.08	5.55	0.000	0.000	1.44	0.42
	try 1	190000	0	0.00	2.26	5.13	0.00	0.00	22.64	5.13	0.000	0.000	0.00	0.00
	try 2	190000	0	0.00	2.67	5.21	0.41	0.08	26.73	5.21	0.000	0.000	4.09	0.08
	avg	190000	0	0.00	2.45	5.29	0.18	0.17	24.48	5.29	0.000	0.000	1.84	0.17
4096	try 0	190000	312	0.16	2.50	8.33	0.24	3.20	25.03	8.33	7.676	10.269	2.39	3.20
	try 1	190000	306	0.16	2.78	8.00	0.52	2.88	27.82	8.00	16.922	9.405	5.18	2.88
	try 2	190000	306	0.16	2.53	8.56	0.27	3.43	25.31	8.56	8.722	11.222	2.67	3.43
	avg	190000	308	0.16	2.61	8.30	0.34	3.17	26.05	8.30	11.084	10.299	3.41	3.17
2048	try 0	190000	672	0.35	3.18	11.93	0.91	6.80	31.76	11.93	13.570	10.122	9.12	6.80
	try 1	190000	672	0.35	3.26	14.88	1.00	9.75	32.60	14.88	14.830	14.510	9.97	9.75
	try 2	190000	672	0.35	3.19	11.90	0.92	6.77	31.87	11.90	13.743	10.076	9.23	6.77
	avg	190000	672	0.35	3.21	12.90	0.94	7.77	32.08	12.90	14.048	11.569	9.44	7.77
1024	try 0	190000	1602	0.84	4.09	20.75	1.83	15.62	40.95	20.75	11.429	9.752	18.31	15.62
	try 1	190000	1602	0.84	4.28	20.24	2.02	15.12	42.79	20.24	12.579	9.436	20.15	15.12
	try 2	190000	1607	0.85	3.92	22.73	1.66	17.60	39.21	22.73	10.312	10.955	16.57	17.60
	avg	190000	1604	0.84	4.10	21.24	1.83	16.11	40.98	21.24	11.436	10.046	18.34	16.11
512	try 0	190000	5408	2.85	9.96	51.35	7.69	46.23	99.58	51.35	14.227	8.548	76.94	46.23
	try 1	190000	5415	2.85	8.30	51.35	6.04	46.22	83.05	51.35	11.156	8.536	60.41	46.22
	try 2	190000	5368	2.83	8.12	53.93	5.86	48.80	81.25	53.93	10.918	9.092	58.61	48.80
	avg	190000	5397	2.84	8.80	52.21	6.53	47.09	87.96	52.21	12.103	8.724	65.32	47.09
256	try 0	190000	48408	25.48	47.31	458.07	45.05	452.95	473.10	458.07	9.305	9.357	450.46	452.95
	try 1	190000	46751	24.61	46.10	474.60	43.84	469.47	461.04	474.60	9.377	10.042	438.40	469.47
	try 2	190000	48835	25.70	47.92	450.84	45.65	445.72	479.16	450.84	9.348	9.127	456.52	445.72
	avg	190000	47998	25.26	47.11	461.17	44.85	456.04	471.10	461.17	9.343	9.501	448.46	456.04

表 3: Apache Bench の結果 (リクエスト回数 = 10000)

この結果を見ると，ミスが 8 回発生したが，reclaims の数が増えていないので，キャッシュにすでに乗っていたエントリを変える必要がなかったことがわかり，結果として，キャッシュにまだ余裕があることがわかった．また，1 回 free が行われたことについては，ミスが 8 回発生し，アロケーション 9 回あったので，その中の 1 つのアロケーションの持っていた結果を保存する必要がなく，解放したものと考えられる．

4.1.4 Memcached による計測結果

最後に，ここまでの結果を確認するために，memcached サーバを立て，LAS の応答速度を計測した．その結果を表 5 に示す．それぞれの結果が 3 回をベンチマークを動かして平均を取った値となる．また，すべてのベンチマークをシングルスレッドで取っている．用語については，表 3 の説明と同様である．

memcached での計測では，計測全体で一度しか認可判定処理が発生しないことがわかった．結果として，memcached ではキャッシュ削除の頻度を上げてもキャッシュミスが非常に少ないため，応答速度への影響を検討しづらい部分もあったが，少なくともキャッシュ削除の頻度が 1024 アクセスに 1 回より頻繁でなければ，LAS の応答速度に影響がほとんど出ないことがわかった．

4.2 改ざん耐性と迂回困難性の検証

本節では，4.1.1 で説明した環境で，オリジナルの SELinux と提案機構それぞれについて，TCB とするコンポーネントを仮定し，各々のサイズを比較することで強制アクセス制御機構の改ざん耐性に提案機構が与える影響を検証する．また同時に，同仮定のもとで，制御対象の捕捉から認可判定の執行にいたる処理経路を比較し，提案機構が強制アクセス制御機構の迂回困難性に与える影響についても検証する．

はじめに，改ざん耐性の検証では，オリジナルの SELinux の TCB を Linux カーネルと SELinux ポリシと仮定し，同時に提案機構の TCB を，提案機構用 Linux カーネルと IC の実体であるカーネルモジュール LKM，LAS の実体で SELinux ポリシに相当する XSB

flush interval	AVC access nb	miss nb	miss %	total time(s)	total penalty(s)	penalty/miss(ms)
never	90003	0	0	31.68	0	–
4096	90003	23	0.0256	31.91	0.230	10
2048	90003	48	0.0533	31.91	0.230	4.79
1024	90003	105	0.117	32.34	0.660	6.29
512	90003	257	0.286	33.30	1.62	6.30
256	90003	846	0.940	36.67	4.99	5.90

表 5: Memcached による計測結果

	Linux Kernel	LKM	Policy	Total
提案機構	2437472	4928	770279	3212679
SELinux	2426656	0	903549	3330205
差	10816	4928	-133270	-117526

表 6: コードとデータサイズの比較 (byte)

のポリシ処理系と仮定した上で，各々のサイズを比較した．その結果を表 6 に示す．

提案機構用 Linux カーネルと LKM は，提案機構で IC を実装するために追加したコードとデータが含まれており，カーネル空間で動作する両者を合わせると，オリジナルの Linux カーネルと比べて 0.6%程度サイズが増加している．また，SELinux ポリシと XSB のポリシ処理系の比較では，14.8%程度サイズが減少しており，Linux カーネルと LKM，ポリシを合わせると 4%程度サイズが減少していることがわかった．

次に，迂回困難性の検証では，OS の正当性と適切なポリシによる制限を前提として，あるアクセス主体が与えられたアクセス権限を越えて以下の 3 点を実行する難しさを定性的に比較した．

1. ポリシを書き換えて再度組み込む．
2. 強制アクセス制御を無効にする．
3. 認可判定結果を書き換える．

はじめに，1 と 2 については，オリジナルの SELinux と提案機構ともに，いくつかの決められたコマンドの実行を適切に制限することで不正な実行を防止可能であり，従って，与えられた前提下での迂回困難性は双方ともに同程度であると考えられる．また，3 については，オリジナルの SELinux はデータフローがカーネル空間で動作する OM/AVC/SS に閉じており，一方の提案機構は IC の一部と LAS がユーザ空間に露出している．一般に，カーネル空間データの不正な書き換えは，ユーザ空間データのものより困難であるため，オリジナルの SELinux の方が提案機構よりも迂回不可能性が高いと考えられる．

4.3 有用性と安全性の考察

評価結果を簡潔にまとめると，以下のようになる．

1. LAS の認可判定処理はかなり遅く，数 μs で済むような処理が数 ms がかかる．
2. ほとんどの認可判定がキャッシュで処理されるので，提案機構でも性能の問題は発生しない．
3. アプリケーションによって, 必要な認可判定処理の数が大きく変わるので，応答時間が多少延びても影響が少ない場合がある．
4. TCB のサイズをその改ざん耐性と見たときに，オリジナルの SELinux と提案機構はそれほど大きな差がない．
5. 提案機構が一部の処理をユーザ空間に依存することから，迂回困難性については提案機構が劣る．

総合すると，提案機構は，先行研究で示したポリシ記述言語を，応答性能と改ざん耐性の面で概ね問題なく SELinux に実装可能とする点で有用であるが，認可判定処理の実行経路の面では安全性に課題があると見ることができる．この対策としては，IC や LAS の書き換えポリシを強化し，強制アクセス制御自身によってこれを保護することで，オリジナルの SELinux と同程度の迂回困難性を得ることができると考えられる．

従って，オリジナルの SELinux の機構と，提案した論理型言語による機構は，双方に異なる特徴があるため，利害得失は利用ケースに依存すると考えられる．例えば，ある程度固定的なポリシを強固に強制するような場合は前者が適しているし，ポリシの柔軟な修正や構造化記述，記述力の拡張が必要な場合には後者の方が適している．ただし，例えば，論理型言語による記述の柔軟性向上を活かして動的なポリシ変更等を行

う場合には，認可判定結果のキャッシュを利用できないケースがあり得るため，その対策を取る必要がある．

5 おわりに

本稿では，著者らが先行研究で提案したポリシ記述言語を，実際の SELinux に接続する実装方式について説明し，応答性能と改ざん耐性，迂回困難性を評価することで，その有用性と安全性を実証した．

提案機構は，オリジナルの SELinux の認可判定処理を横取りし，ユーザ空間においた Datalog によってその処理を代行するもので，横取り処理・カーネル/ユーザ空間の通信を担う IC と，ユーザ空間で認可判定処理を担う LAS を基礎に構成した．

提案機構は，認可判定処理の一部をユーザ空間に実装するため，応答性能と安全性に懸念があったが，本研究において，LAS そのものの応答速度に加えて，ApacheBench と Memcached による応答性能の評価を行い，キャッシュの影響を検証することで，提案機構が実際の認可判定処理を現実的な時間で実行できることを実証した．また，安全性については，複数の仮定をおいた上で改ざん耐性と迂回困難性の評価を行い，改ざん耐性についてはオリジナルの SELinux と大きな差異がなく，迂回困難性については，実行経路にユーザ空間が含まれることで低下していることを示し，その対策を含めた有用性について考察した．

今後は，実行経路の堅牢化方法を検討すると同時に，AVC の利点を受けながら記述の柔軟性をあげる方法を検討する．また，本研究の具体的な応用として，アクセス制御モデルや名前空間の異なる分散環境で，強制アクセス制御を実現するポリシ記述方式やアーキテクチャについても検討していく計画である．

参考文献

[1] 橋本 正樹, 金 美羅, 辻 秀典, 田中 英彦. 論理プログラミングを基礎とした認可ポリシ記述言語. 情報処理学会論文誌, 51(9):1682–1691, Sep 2010.

[2] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, Mar 1989.

[3] Ray Spencer, Stephen Smalley, Peter Loscocco, ,Mike Hibler, and David Andersen. The flask security architecture: System support for diverse security policies. In *in Proceedings of The Eighth USENIX Security Symposium*, pages 123–139, 1999.

[4] Prasad Rao, Konstantinos F. Sagonas, Terrance Swift, David Scott Warren, and Juliana Freire. Xsb: A system for effciently computing wfs. In *LPNMR '97: Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 431–441, London, UK, 1997. Springer-Verlag.

[5] Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1):20–74, 1996.

[6] Weidong Chen, Michael Kifer, and David S. Warren. Hilog as a platform for database languages. In *Proceedings of the second international workshop on Database programming languages*, pages 315–329, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[7] S. Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, K. Sagonas, S. Skiena, T. Swift, and D. S. Warren. Unification factoring for efficient execution of logic programs. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 247–258, New York, NY, USA, 1995. ACM.

[8] Krzysztof R. Apt. Introduction to logic programming. Technical report, Austin, TX, USA, 1988.

[9] David S. Warren. Programming in tabled prolog, 1995.

[10] Moritz Becker, Cedric Fournet, and Andrew Gordon. Design and semantics of a decentralized authorization language. In *CSF '07: Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 3–15, Washington, DC, USA, 2007. IEEE Computer Society.

[11] Joseph Y. Halpern and Vicky Weissman. Using first-order logic to reason about policies. *ACM Trans. Inf. Syst. Secur.*, 11(4):1–41, 2008.