ソケット通信によるデータフローを考慮した IoT デバイス向け動的テイント解析システムの開発

東 政澄[1,a)]　掛井 将平[1]　齋藤 彰一[1]

概要：IoT デバイスは通常のデバイスと異なり，複数のデバイスとプロセス間通信を頻繁に実施する傾向にある．しかし，IoT ファームウェアの自動解析では，単一プロセスの監視を行うものが多く，複数プロセスを総合的に監視するものは少ない． 単一プロセスのみの追跡の場合は複数のアラートが検出されて偽陽性の増加につながるため，複数プロセスを考慮した解析が望まれる．本研究では，複数プロセス考慮の第一歩として，プロセス間通信の 1 つであるソケット通信に着目する．ユーザーの入力を起点とするデータフローを解析する動的テイント解析を拡張し，対象プロセス間でテイント情報を低コストで共有する手法を提案する．本手法は，既存の動的テイントエンジンやカーネルに手を加えることなく導入可能である．評価では，ソケット通信を行うバイナリに対する本手法のシミュレートを行い，実現可能性を評価する．また，基本的なソケット通信を行うバイナリに対して本手法を適用した際の実行時間やオーバーヘッドの計測も評価する．

キーワード：プロセス間データフロー，動的テイント解析，IoT ファームウェア

Development of a Dynamic Taint Analysis System for IoT Devices Considering Socket Communication Data Flow

Masazumi Azuma[1,a)]　Shohei Kakei[1]　Shoichi Saito[1]

Abstract: IoT devices differ from normal devices in that they often communicate with multiple devices and processes. However, most automated IoT firmware analyses monitor only a single process, with few addressing multi-process behavior. Monitoring only a single process can result in numerous alerts and an increase in false positives, making it desirable to analyze multiple processes in tandem. The first step in multi-process analysis, this study focuses on socket communication, a common form of inter-process communication. We propose a low-overhead method for sharing taint information between target processes by extending dynamic taint analysis, which tracks data flow originating from user input. Our method requires no modification to existing dynamic taint engines or the operating system kernel. To evaluate its feasibility, we simulate the proposed method on binaries that use socket communication. We also measure the execution time and overhead when applying it to binaries with basic socket communication functionality.

Keywords: Inter-Process Communication, Dynamic Taint Analysis, IoT Firmware

1. はじめに

モノのインターネット (IoT) はスマートホームやスマートファクトリーのようなビジネスへの利用から，個人利用の生活に根ざした利用まで，多岐に渡る用途で急速に普及している．しかし，普及に伴い，IoT 機器に対する攻撃が増加している．例として 2017 年に，Mirai ボットネットが数百万の IoT 機器を乗っ取り，それらを用いて DNS サーバーに DDoS 攻撃を仕掛けた事例が発生した [1]．また，2015 年には BlackEnergy マルウェアによる持続的な攻撃により，産業制御システムの制御が奪われ，長期の停電が

[1] 名古屋工業大学
Nagoya Institute of Technology
[a)] m.azuma.818@nitech.jp

© 2025 Information Processing Society of Japan

This paper is work in progress and not peer-reviewed.

引き起こされた [2].

IoT 機器が攻撃のターゲットに選ばれる理由として，多くのセキュリティ課題が未解決であり，攻撃に利用できる脆弱性が残りやすいことが挙げられる [3]．セキュリティ課題の例として，まずセキュリティ機構が乏しいことが挙げられる．IoT 機器は計算資源が乏しく，十分なセキュリティ機構を設けることが難しいため，攻撃に対する防御が不十分になる．次に，攻撃に対応するためのセキュリティアップデートが遅延する傾向にあることが挙げられる．IoT 機器を動作させるためのファームウェアはベンダー独自の設計になっていることが多く，更新プログラムもベンダーが作成する．そのため，脆弱性が見つかった場合でも対処に時間を要することが多い．このようなセキュリティ課題が原因で IoT 機器が攻撃の対象に選ばれることが多い．

ファームウェアの脆弱性を特定するために，IoT 機器の徹底的なセキュリティ検査が必要である．セキュリティ検査は，セキュリティの専門家や研究者によって行われる．しかし，これらの検査は手動が多く，解析処理の複雑さや膨大な量のバイナリが原因で解析コストが高い．そのため，自動解析によるファームウェアの脆弱性特定に関する研究が盛んに行われている．

自動解析を目的とした既存研究 [4], [5], [6] では，プロセス間の依存関係を考慮せず，解析可能な単一のプログラムやモジュールに焦点を当てている．しかし，IoT 機器はソケット通信や共有メモリなどのプロセス間通信を用いてデータ共有を行っているため，プロセス間の依存関係を考慮しない解析は IoT 機器の特性を十分に反映していない．また，プロセス毎に脆弱性と思われる箇所を示すアラートを出した場合，一連の処理の結果で起きるアラートが個別に報告されることで，アラート数が増加する．その結果，アラート間の関連性を無視した情報をユーザーに報告するために偽陽性が増加する．以上より，IoT 機器のファームウェアに対する自動解析には，プロセス間の依存関係を考慮することが必要である．

本研究は，プロセス間の依存関係を考慮することで，ファームウェア解析の精度を向上させると共に，IoT 機器に対するより効果的なセキュリティ検査手法の提供を目的とする．これらの目的に対する課題として，まず，解析情報の共有が挙げられる．複数プロセスを同時に解析する場合，各プロセスの解析情報を全体に共有する必要がある．そのため，解析プログラムが各プロセスの解析情報を把握しながら，解析情報を適切に共有する必要がある．次に，導入コストの高さが挙げられる．既に，単一プロセスでのデータフローを追跡する自動解析システムは存在する．それらに代わるシステムを作成しても，変更するためのコストが高くなる．そこで，目的達成の第一歩としてソケット通信に着目し，ソケット通信によるデータフローを考慮した自動解析システムを提案する．本手法は既存のシステムを拡張する形で実装しているため，低コストで導入可能である．

以下，2 章でファームウェアの自動解析に関する研究を説明する．3 章では，本研究のアプローチと提案に使用するテイントエンジンのアーキテクチャについて説明する．4 章では，提案手法について説明し，5 章でその評価を行う．最後に，7 章でまとめを述べる．

2. 関連研究

IoT ファームウェアの脆弱性を特定するために，自動解析に関する研究が行われている．それらの研究は，単一プロセスに着目した解析と複数プロセスに着目した解析に大別される．

2.1 単一プロセスに着目した解析

ファームウェア解析の自動化に向けて，従来の解析技術をファームウェアに適用する研究が行われてきた．SURROGATES[4] と Firmadyne[5] は，IoT デバイスに関する動的解析に関する課題を解決している．また，Costin ら [6] は，静的解析と動的解析を組み合わせることで，IoT デバイスの Web インターフェースを解析するフレームワークを提供している．しかし，これらの解析手法では，IoT のプロセス間通信を考慮していないため，十分な解析を行うことが出来ない．

IoT デバイスのファームウェアに含まれる様々なバイナリは，センサー情報やユーザーの入力などのデータを他のデバイスやプロセスと共有する．単一プロセスに着目した解析では，データの入力地点に近いプロセスに絞って解析が行われる．しかし，バグの発生が，データの入力地点に近いプロセスではなく，データが渡された先のプロセスで起こった場合，そのバグを検知することが出来ず，見逃しの原因となる．また，そのバグの原因となるプロセスを追跡しようとしても，プロセス間のデータフローを解析していないため，特定することが出来ない [7]．IoT 機器のファームウェアに含まれる全バイナリを個別に解析した場合も，バグやアラートが各プロセス毎に検知されるものの，それらの関連性を無視した情報を提供するため，多数の偽陽性が発生する．よって，単一プロセスに着目した解析では，他プロセスに含まれるバグの見逃しや偽陽性増加につながるため，プロセス間の関係を考慮した解析が望まれる．

2.2 複数プロセスに着目した解析

複数プロセスに着目する際に利用される技術として，テイント解析が挙げられる．これは，外部から入るデータのデータフローを追跡する手法である．この技術により，外部データからのデータフローを追跡することで，バグの発生源とその原因箇所を特定することが可能である．テイント解析を用いた自動解析に関する研究として，静的解析を

© 2025 Information Processing Society of Japan

主軸とした解析と動的解析を主軸とした解析の 2 種が存在する．

2.2.1　静的解析を主軸とした解析

静的解析はファームウェアイメージの可読な部分を解析し，脆弱性を特定する方法である．可読な部分を網羅的に解析するため多くの脆弱性を発見することが出来る．その一方で，実動作に基づいていないため，外部データの入り口であるバイナリかの判別が難しい．そのため，バイナリにソケット通信の処理が入っているかや，入力文字列をパースする処理が入っているかを調査することで判定を行う．しかし，判定は絶対的なものではなく，経験則に基づく推定により行われる．そのため，閾値や計算によって誤判定が起き，多くの偽陽性を引き起こす．

Karonte[8] と SaTC[9] は，あらゆるバイナリ形式のファームウェアサービスに適用可能という汎用性が特徴の，静的テイント解析フレームワークである．一方で，汎用性を優先したことにより，解析時間の増加やカバレッジの低さ，偽陰性の高さを引き起こしている．そのため，軽量な解析手法や解析精度の向上を達成するために様々な研究が行われている [10], [11].

2.2.2　動的解析を主軸とした解析

動的解析はファームウェアイメージを QEMU などのエミュレータを用いて動作させることで，実動作に基づく解析を行う方法である．実際の動作を基に解析するため，複雑な近似を必要としないことが特徴である．その一方で，ファームウェアを動作する環境は十分に整備されていない．そのため，特定のケースのみ有効であるツールが多く，スケーラビリティに欠ける．

Dytan[12] や Straighttaint[13] は動的テイント解析のオーバーヘッドの削減や対応データフローの増加を達成している動的テイント解析フレームワークである．一方で，ファームウェアリホスティングの成功率の低さや，効率的なコードカバレッジの探索が出来ていないことが重なり，実用的ではない．そのため，動的テイント解析自体の性能向上に関する研究 [14], [15], [16] や，ファームウェアリホスティングの成功率向上に関する研究 [17] など，動的テイント解析の実用化に向けた研究が行われている．

3.　本研究のアプローチとテイントエンジン

複数プロセスを考慮した多くの研究は静的解析を主軸に行われている．これは，動的解析の制約が厳しく，実用段階に至っていないためである．しかし，動的解析のオーバヘッドの問題やスケーラビリティの課題は解決されつつある．そのため，静的解析と動的解析の両方に着目し，それぞれの欠点を補い合った解析を行うべきである．よって，本研究では，動的テイント解析に着目する．

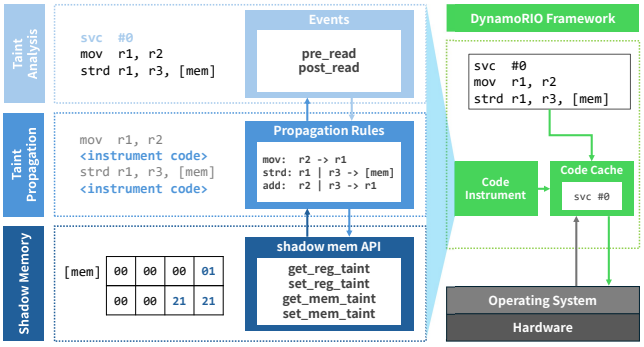

図 1　drtaint-arm32 の全体像
Fig. 1　Overview of drtaint-arm32

3.1　テイントエンジンのアーキテクチャ

本提案は既存の動的テイントエンジンである drtaint-arm32[18] をベースに構築する．drtaint-arm32 は DynamoRIO[19] 上に作成されている．DynamoRIO は x86 や Arm などのアーキテクチャに対応しており，Armv7l アーキテクチャでも安定して動作することが特徴である．

図 1 に drtaint-arm32 のアーキテクチャを示す．DynamoRIO は Code Cache と呼ばれる機構に一時的に命令を保持し，その命令に合った計装コードを実行する．drtaint-arm32 では，計装コードとしてテイント処理を記述することで，動的テイント解析を実現する．テイント処理はシャドウメモリ，テイント伝播処理，テイント解析の 3 つの層から成る．

3.1.1　シャドウメモリ

シャドウメモリとは，解析対象プログラムからは隠匿されているメモリを表す．drtaint-arm32 では，シャドウメモリをテイントの記録のために利用する．レジスタやメモリアドレスのテイント値をバイト粒度で記録する．例えば，あるレジスタの下位 14bit がテイントタグ `0x21` によりテイントされた場合，そのレジスタのシャドウメモリには `0x00002121` が記録される．

3.1.2　テイント伝播

テイント伝播では，事前に定めたルールの下でテイントをどのように他のレジスタやメモリに伝播するかを定義する．例えば，`mov r1, r2` という命令では，`r2` がテイントされていた場合は `r1` に伝播するため，`r2 -> r1` というルールを定義する．そして，伝播処理やシャドウメモリへの記録処理を元のコードに計装する．drtaint-arm32 では，SIMD 命令を除いた約 300 命令に関する伝播ルールを定義している．

3.1.3　テイント解析

テイント解析では，テイントの入口であるソースと出口であるシンクを決め，その間の命令の情報を解析する．例えば，read システムコールの実行後にテイントをバッファにセットし，write システムコールの実行前にテイント値をチェックする．これにより，read というソースから write

© 2025 Information Processing Society of Japan

というシンクまでのデータフローを解析することが出来る．

4. 提案手法

本章では，動的テイント解析における複数プロセス監視のための提案手法について述べる．本提案では，研究の第一歩として，対象となるプロセス間通信をソケット通信にのみ絞る．これは，ソケット通信によるデータフローがシステムコールの追跡のみで比較的容易に監視することが出来るためである．また，入力となるバイナリは Arm アーキテクチャとする．これは，IoT 機器に頻繁に用いられるアーキテクチャが Arm であるためである．その中でも，アーキテクチャの複雑さが比較的少ない Armv7l を対象とする．これに伴い，提案手法では以下の制約がある．

- 入力はファームウェアではなく，Armv7l バイナリとする
- 一度に追跡可能なバイナリは 2 つ以下とする
- 1 組の 2 つのバイナリを入力とする場合は，両バイナリ間でソケット通信を行う

以上を踏まえて，提案手法の全体像を図 2 に示す．提案手法は，動的テイント解析による基本的な解析に加えて，2 つのプロセス間でのタグ情報を共有するタグ共有テーブル，その出力の整形を行う出力フォーマッタによって構成される．タグ共有テーブルにより，ソケット通信によるデータフローを既存のテイントエンジンでも追跡可能となる．また，共有メモリにより実装されており，テイントエンジンやカーネルに変更を加える必要が無いため，テイントエンジンに低コストで組み込み可能である．

4.1 タグ共有テーブル

タグ共有テーブルは，テイント解析で用いられるテイントタグを共有するための補助機構である．テーブルは，drtaint-arm32 を計装したプロセスがタグの情報を読み書きする際に利用される．計装プロセスやそれらを動作させる QEMU 上のプロセスが共有できるようにするために，テーブルは OS 上の共有メモリに実装する．テーブルへの読み取りや書き込みはライブラリとして実装し，計装プロセスでも他のプロセスでもテーブルにアクセス可能とする．テイントタグ情報やフロー情報をテーブルに書き込み，他の計装プロセスに共有することで，一意にフローを識別できる．以降，フロー識別のためのアイデアであるデータキーと詳細なテーブル情報を述べる．

4.1.1 プロセス間通信とデータキー

データキーとは，プロセス間通信に関わる全プロセスが認識する一意のキーを表す [8]．逆に言えば，プロセス間通信を行う際にデータキーを知らなければ通信を行うことが出来ない．以下に IoT ファームウェアが用いる一般的なプロセス間通信とデータキー，IoT 機器での利用用途を示す．

- ファイル
  - データキー：ファイル名
  - プロセスがファイルを読み書きすることで共有
- 共有メモリ
  - データキー：バックアップファイル名，共有メモリページの仮想アドレス
  - バックアップに利用したり共有資源として利用
- 環境変数
  - データキー：環境変数名
  - プロセスが環境変数を読み書きすることで共有
- ソケット
  - データキー：ソケットのエンドポイント
  - 同じホスト上のプロセスや他ホストのプロセスと共有
- コマンドライン引数
  - データキー：呼び出しプログラム名
  - 引数を通じてデータ共有

本手法ではソケット通信によるデータフローに着目するため，タグ共有テーブルではソケットのエンドポイントに着目する．これらの情報は，システムコールフック時のファイルディスクリプタの情報を解析することで比較的容易に取得可能である．

4.1.2 共有メモリによるタグ共有テーブル

タグ共有テーブルは 2 つのテーブルとそれらにアクセスするための API を持つ．図 3 に 2 つのテーブルの内容について示す．

図 3 の左のテーブルは，テイントタグとファイルディスクリプタ，プロセス ID を保存するフロー識別用テーブルである．これは，テイントタグがどのプロセスのどんな種類のフローを追跡しているかを保存するために用いられる．テイントタグ，ファイルディスクリプタ，プロセス ID のカラムでできており，テイントタグの種類分のレコードがある．drtaint-arm32 のテイントタグは 8 種類存在するため，フロー識別用テーブルは 8 レコードの情報を持つ．8 種のテイントタグは共有テーブル初期化時にテーブルに書き込まれる．

図 3 の右のテーブルは，ファイルディスクリプタとプロセス ID を主キーとし，ソケットのエンドポイントに関する情報を保存するソケット通信識別用テーブルである．これにより，どのフローがどのソケットのエンドポイントの情報を取得しているかを他プロセスからも閲覧可能とする．

解析の例として，`send`，`recv` で通信を行うプロセスのデータフローを追跡することを考える．このプロセスは，プロセス 1 とプロセス 2 に分かれており，プロセス 1 には `read`，`send` システムコール，プロセス 2 には `recv` システムコールが含まれているものとする．そして，プロセス 1 の `read` システムコールによる標準入力受信から始まり，`send` による送信，プロセス 2 の `recv` 受信という処理を行うものとする．この時，プロセス 1 と 2 の間でソケット通信によるプロセス間データフローが存在する．このデータ

© 2025 Information Processing Society of Japan

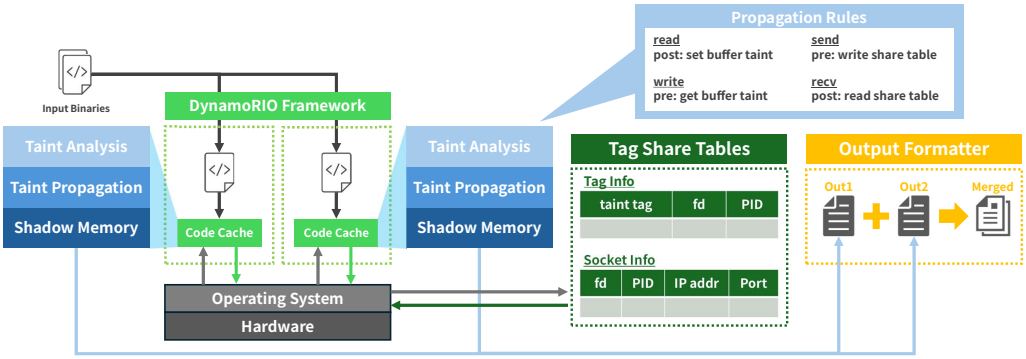

図 2 提案手法の全体像
Fig. 2 Overview of Proposal Method

フロー識別用テーブル

taint tag	fd	PID
00000001	1	1000
00000010	4	2222
00000100	-	-
00001000	-	-

ソケット通信識別用テーブル

fd	PID	IP addr	Port
3	1111	127.0.0.1	1234
4	2222	192.168.0.1	4321
-	-	-	-
-	-	-	-

図 3 タグ共有テーブルの持つ 2 つの共有テーブル
Fig. 3 Two Shared Tables in Tag Sharing Table

フローをタグ共有テーブルにより検出することを考える．

read 命令実行時，引数の情報からファイルディスクリプタとプロセス ID を取得する．これらの情報をフロー識別用テーブルの空いているレコードに書き込み，当該レコードのテイントタグを取得する．その後，取得したテイントタグを利用してテイント解析を実施する．今回の例では，ファイルディスクリプタを 4，プロセス ID を 2222，テイントタグを 0x02 とする (図 3 のフロー識別用テーブルの 2 行目)．

send 命令実行時，送信データのテイントタグを確認し，フロー識別用テーブルからテイントタグをキーとして，ファイルディスクリプタとプロセス ID を参照する．具体的には，送信データにテイントタグ 0x02 が付いていた場合，フロー識別用テーブルのタグが 0x02 のレコードを参照し，ファイルディスクリプタ 4 とプロセス ID 2222 を取得する．その後，ソケット通信識別用テーブルにファイルディスクリプタとプロセス ID，自 IP アドレス，ポート番号を書き込む (図 3 のソケット識別用テーブルの 2 行目)．

recv 命令実行時，通信相手の IP アドレスとポート番号を取得し，ソケット通信識別用テーブルを参照する (図 3 の 2 行目)．これにより，send 命令実行時に書き込んだファイルディスクリプタとプロセス ID を取得できる．そして，ファイルディスクリプタとプロセス ID をキーとして，フロー識別用テーブルからテイントタグを取得 (図 3 の 2 行目) し，そのテイントタグによってテイント解析を行う．このように，プロセス 2 の recv 命令実行時にプロセス 1 で書き込んだファイルディスクリプタ 4 とプロセス ID 2222，テイントタグ 0x02 を取得できる．プロセス ID からソースとなるプロセスを，ファイルディスクリプタから標準入力を含むファイルの種類を一意に特定することができる．以上より，複数のプロセスを介したデータフローをタグ共有テーブルによって一意に識別可能である．

4.2 追加する伝播ルール

タグ共有テーブルを用いて複数プロセスによるデータフローを識別可能であることを示した．識別のために，read，send，recv システムコールフック時に，テイント操作に関する処理であるテイント伝播ルールを追加する．追加する伝播ルールは以下の通りである．

- read
  - 自身の PID とファイルディスクリプタを取得
  - フロー識別用テーブルに，取得した PID とファイルディスクリプタを記入
  - 書き込んだレコードのテイントタグを利用しテイント挿入
- send
  - バッファのテイントを取得し，フロー識別用テーブルから PID とファイルディスクリプタを取得
  - ソケット通信識別用テーブルに，取得した PID とファイルディスクリプタを記入
  - IP アドレスとポート番号を取得し，前操作で記入したレコードと同じレコードに記入
- recv
  - 接続相手の IP アドレスとポート番号を取得し，ソケット通信識別用テーブルを参照
  - 参照したレコードの PID とファイルディスクリプタを取得
  - フロー識別用テーブルを参照し，テイントタグを取得
  - 取得したテイントタグを利用しテイント挿入

4.3 出力フォーマッタ

出力フォーマッタは，動的テイント解析で得られた解析データを 1 つにまとめる機構である．図 4 に出力フォー

© 2025 Information Processing Society of Japan

```
{
    "address": "B6E0EDEC",
    "bytes": "1F002DE9",
    "operands": [
        {
            "name": "r2",
            "taint": "01010101",
            "type": "register",
            "value": "00000006"
        }
    ],
    "program": "target_app3_client",
    "time": "1754898106737137"
},
{
    "address": "B2E65B04",
    "bytes": "0030D3E5",
    "operands": [
        {
            "name": "3202563296",
            "taint": "01",
            "type": "address",
            "value": "68"
        }
    ],
    "program": "target_app3_server",
    "time": "1754898106829601"
}
```


図 4 出力フォーマッタが出力する解析結果
Fig. 4 Analysis Results by Output Formatter

マッタが出力する解析結果の抜粋を示す．解析結果には，テイント伝播に関与した命令に関する情報が記録される．命令に関するデータとして，アドレスや命令のバイト，オペランドを出力する．また，複数プロセス解析固有の情報として，プロセス名やタイムスタンプも出力する．タイムスタンプを基準にしたソートによって，プロセスを介したデータフローに関するデータを実行順に出力することが出来る．

タイムスタンプにはマイクロ秒精度の UNIX 時間を採用する．通常実行の場合，各命令の実行速度が速く，UNIX 時間をタイムスタンプとして利用することは難しい．しかし，本提案のシステムでは，命令の前にテイント処理を計装するため，マイクロ秒精度の UNIX 時間であればタイムスタンプとして機能する．

5. 評価

提案手法の評価を行うために，実現可能性のためのシミュレーションとオーバーヘッドの評価を行う．図 5 に評価に用いるプログラムの略図を示す．評価プログラム 1 は，ソケット通信を行わないプログラムである．`read` システムコールで標準入力を取得し，そのデータを `write` システムコールで標準出力に書き込む．評価プログラム 2 は，ソケット通信を 1 回実施するプログラムである．評価プログラム 1 に，`send` システムコールと `recv` システムコールを用いたソケット通信を組み込んだ処理を行う．評価プログラム 3 は，クライアントとサーバーの関係を模したプログラムである．クライアントから送信したデータをサーバーが加工し送り返す処理を行う．サーバーでの加工処理は入力を大文字に書き換える処理である．

5.1 実現可能性評価のためのシミュレーション

提案手法の実現可能性を評価するため，評価プログラム 2 を用いたシミュレーションを行う．シミュレーションでは，タグ共有テーブルの様子やアクセスの詳細を追う．図 6 に評価プログラム 2 によるシミュレーションを示す．

プロセス 1 の `read` 命令実行時，引数のファイルディスクリプタ `0` と自身のプロセス ID `1` をフロー識別用テーブルに書き込む．そして，同じレコードのテイントタグ `0x01` を利用し，テイント解析を実施する．

プロセス 1 の `send` 命令実行時，引数のバッファのテイントタグを取得する．この時，プロセス 1 の `read` のバッファと同じバッファを使用しているため，テイントタグは `0x01` が得られる．その後，フロー識別用テーブルからテイントタグをキーとして，ファイルディスクリプタ `0` とプロセス ID `1` を取得し，ソケット通信識別用テーブルに書き込む．また，通信相手の IP アドレス `10.0.0.1` とポート番号 `8080` も同レコードに書き込む．

プロセス 2 の `recv` 命令実行時，通信相手の IP アドレスとポート番号を調べる．ここでは，`10.0.0.1` と `8080` が得られる．次に，ソケット通信識別用テーブルから IP アドレスとポート番号をキーとして，ファイルディスクリプタ `0` とプロセス ID `1` を取得する．最後に，フロー識別用テーブルからファイルディスクリプタとプロセス ID をキーとして，テイントタグ `0x01` を取得し，テイント解析を実施する．

以上の流れにより，プロセス 1 の `read` からプロセス 2 の `write` までのデータフローを追跡することが出来る．実際に提案手法を動作させた結果，ソケット通信によるデータフローを追跡することが出来た．また，図 4 のような解析結果も取得することが出来た．

5.2 オーバーヘッド評価

提案手法のオーバーヘッドを評価するため，全評価プログラムを用いた比較実験を行う．評価指標として，実行時間と最大メモリ使用量の 10 回平均を計測する．評価プログラム 1 では，drtaint-arm32 と提案手法を実行することで，タグ共有テーブルとシステムコールフック時の追加処理に関するオーバーヘッドを比較する．評価プログラム 2，3 では `send`，`recv` システムコールが増えた際のオーバーヘッドの変化を計測する．評価結果を表 1 に示す．drtaint-arm32 は複数プロセスによる解析を行えない

© 2025 Information Processing Society of Japan

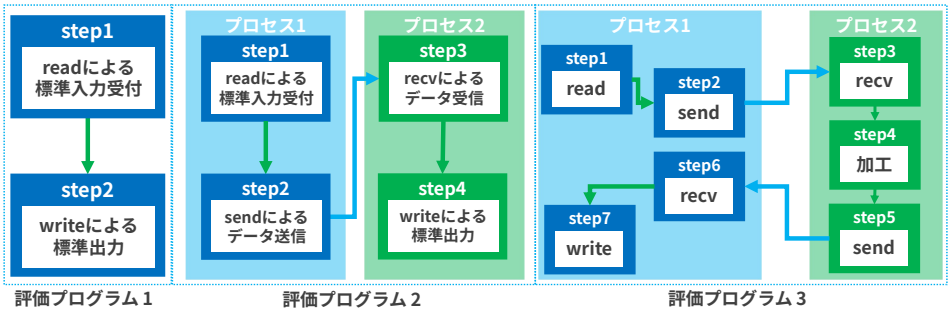

図 5 評価対象プログラム
Fig. 5 Eval Program

表 1 オーバーヘッドの評価結果
Table 1 Overhead Evaluation Results

評価プログラム	提案手法		drtaint-arm32	
	平均実行時間 (s)	最大メモリ使用量 (KB)	平均実行時間 (s)	最大メモリ使用量 (KB)
評価プログラム 1	7.1092	9035.6	6.9367	9090.4
評価プログラム 2	8.6658	10102.4	-	-
評価プログラム 3	9.0065	10100.4	-	-

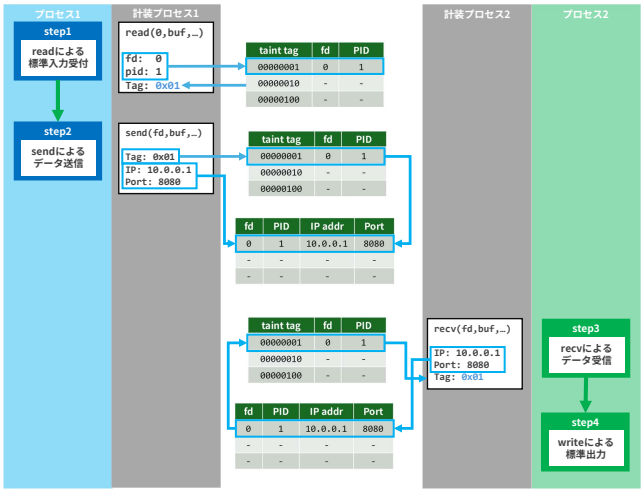

図 6 評価プログラム 2 によるシミュレーション
Fig. 6 Simulation using Evaluation Program 2

ため，評価プログラム 2，3 による計測は未実施である．

評価プログラム 1 による計測結果を比較すると，提案手法のオーバーヘッドは drtaint-arm32 と比べて，平均実行時間が約 0.17 秒増加し，最大メモリ使用量が 54.8KB 減少した．いずれのオーバーヘッドも drtaint-arm32 の計測結果より大きな差が無いことが分かる．

また，評価プログラム 2，3 による計測結果を比較すると，評価プログラム 3 によるオーバーヘッドは，平均実行時間が約 0.34 秒増加し，最大メモリ使用量が 2KB 減少した．ソケット通信の回数が増えるほど平均実行時間は増加するが，これは実行命令数の増加に起因するものと考えられる．解析対象のプログラムが 1 命令増加すると，その命令にテイント処理が軽装されるため，元の命令よりも数十命令増加する．そのため，実行時間の増加の主要因はテイント処理の計装によるものであり，提案手法による影響は小さいと考える．

以上の結果より，提案手法は元にしたエンジンによる解析システムと比べて，低コストで導入可能であることが分かる．また，ソケット通信の回数の増加によるオーバーヘッドへの影響も小さいことが分かる．

6. 今後の課題

本稿では複数のプロセス間通信考慮の第一歩として，ソケット通信に関するデータフローの追跡方法について述べた．今後の課題として，以下の 4 点が挙げられる．

まず，実世界のファームウェアに基づく評価が未実施である．実世界のファームウェアでは，ソケット通信のみに絞ったとしても，様々なエンドポイントとの通信を行っていると考える．そのため，ファームウェアのソケット通信の使用状況を鑑みた評価を行う必要がある．

次に，共有メモリによるデータフローを考慮できていない．IoT 機器で頻繁に用いられるプロセス間通信の一つに，共有メモリが挙げられる．本提案では，単純化のために省略したが，より効果的な解析のためには共有メモリによるデータフローを考慮する必要がある．共有メモリのデータキーは 4.1.1 節で述べたように，バックアップファイル名や共有メモリページの仮想アドレスを考慮する必要がある．

次に，解析前の事前処理を追加できていない．本提案の制約を軽減するために，事前処理を検討している．ファームウェアのバイナリの通信を行うバイナリの特定や，動的テイント解析のコスト削減を目的として行う．これは，解析に補助的に必要な情報を必要とするため，静的解析によって文字列などの情報を元に行う．

最後に，タグ共有テーブルの整理をすべきである．現在

© 2025 Information Processing Society of Japan

の提案手法では，2 つのテーブルによってデータフローを管理しているが，ファイルシステムやプロセス間通信の性質をくみ取れば，1 つのテーブルで管理可能であると考える．そのため，ファームウェアへの拡張前にテーブルを整理することで，解析によるオーバーヘッドを削減する必要がある．

7. まとめ

本論文では，IoT ファームウェアの解析において複数プロセスを考慮する必要性について述べた．IoT デバイスは複数プロセスでデータを共有しているため，効果的な解析を行うためにはそれらを考慮する必要がある．提案手法では，動的テイント解析に着目し，共有メモリによるタグ共有テーブル，テイント解析にて出力された解析結果を整形する出力フォーマッタにより，プロセス間通信を考慮した解析を実施可能であることを示した．

提案手法の有効性を示すために，実現可能性の評価を目的としたシミュレーションやオーバーヘッドの評価を行った．シミュレーションでは，タグ共有テーブルによって提供されるテーブル情報により，プロセス間データフローを一意に識別可能であることを示した．また，オーバーヘッドの評価では，従来の動的テイント解析と比べて，提案手法は複数プロセスを考慮した場合でも低コストで解析可能であることを示した．

今後は，より複雑なケースでの評価や事前処理の追加により，実世界のファームウェアを解析可能かを評価する．また，ソケット通信に加えて，共有メモリによるデータフローも考慮することで，より一層 IoT ファームウェアに効果的な解析を目指す．

参考文献

[1] Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J. A., Invernizzi, L., Kallitsis, M. et al.: Understanding the mirai botnet, *26th USENIX security symposium (USENIX Security 17)*, pp. 1093–1110 (2017).

[2] Kovanen, T., Nuojua, V. and Lehto, M.: Cyber threat landscape in energy sector, *ICCWS 2018 13th international conference on cyber warfare and security*, p. 353 (2018).

[3] Ul Haq, S., Singh, Y., Sharma, A., Gupta, R. and Gupta, D.: A survey on IoT & embedded device firmware security: architecture, extraction techniques, and vulnerability analysis frameworks, *Discover Internet of Things*, Vol. 3, No. 1, p. 17 (2023).

[4] Koscher, K., Kohno, T. and Molnar, D.: {SURROGATES}: Enabling {Near-Real-Time} dynamic analyses of embedded systems, *9th USENIX Workshop on Offensive Technologies (WOOT 15)* (2015).

[5] Costin, A., Zaddach, J., Francillon, A. and Balzarotti, D.: A {Large-scale} analysis of the security of embedded firmwares, *23rd USENIX security symposium (USENIX Security 14)*, pp. 95–110 (2014).

[6] Costin, A., Zarras, A. and Francillon, A.: Automated dynamic firmware analysis at scale: a case study on embedded web interfaces, *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pp. 437–448 (2016).

[7] Chen, J., Diao, W., Zhao, Q., Zuo, C., Lin, Z., Wang, X., Lau, W. C., Sun, M., Yang, R. and Zhang, K.: IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing., *NDSS*, pp. 1–15 (2018).

[8] Redini, N., Machiry, A., Wang, R., Spensky, C., Continella, A., Shoshitaishvili, Y., Kruegel, C. and Vigna, G.: Karonte: Detecting insecure multi-binary interactions in embedded firmware, *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1544–1561 (2020).

[9] Chen, L., Wang, Y., Cai, Q., Zhan, Y., Hu, H., Linghu, J., Hou, Q., Zhang, C., Duan, H. and Xue, Z.: Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems, *30th USENIX Security Symposium (USENIX Security 21)*, pp. 303–319 (2021).

[10] Liu, P., Zheng, Y., Sun, C., Qin, C., Fang, D., Liu, M. and Sun, L.: Fits: Inferring intermediate taint sources for effective vulnerability analysis of iot device firmware, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, pp. 138–152 (2023).

[11] Gibbs, W., Raj, A. S., Vadayath, J. M., Tay, H. J., Miller, J., Ajayan, A., Basque, Z. L., Dutcher, A., Dong, F., Maso, X. et al.: Operation mango: Scalable discovery of {Taint-Style} vulnerabilities in binary firmware services, *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 7123–7139 (2024).

[12] Clause, J., Li, W. and Orso, A.: Dytan: a generic dynamic taint analysis framework, *Proceedings of the 2007 international symposium on Software testing and analysis*, pp. 196–206 (2007).

[13] Ming, J., Wu, D., Wang, J., Xiao, G. and Liu, P.: Straighttaint: Decoupled offline symbolic taint analysis, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 308–319 (2016).

[14] Sang, Q., Wang, Y., Liu, Y., Jia, X., Bao, T. and Su, P.: Airtaint: Making dynamic taint analysis faster and easier, *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 3998–4014 (2024).

[15] Zhang, Y., Liu, T., Wang, Y., Qi, Y., Ji, K., Tang, J., Wang, X., Li, X. and Zuo, Z.: HardTaint: Production-Run Dynamic Taint Analysis via Selective Hardware Tracing, *Proceedings of the ACM on Programming Languages*, Vol. 8, No. OOPSLA2, pp. 1615–1640 (2024).

[16] Chen, S., Lin, Z. and Zhang, Y.: {SelectiveTaint}: Efficient data flow tracking with static binary rewriting, *30th USENIX Security Symposium (USENIX Security 21)*, pp. 1665–1682 (2021).

[17] Angelakopoulos, I., Stringhini, G. and Egele, M.: Pandawan: quantifying progress in linux-based firmware rehosting, *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 5859–5876 (2024).

[18] (poul1x), P. K.: drtaint-arm32: Very WIP taint analysis for DynamoRIO (ARM), `https://github.com/poul1x/drtaint-arm32`.

[19] Bruening, D. and Amarasinghe, S.: Efficient, transparent, and comprehensive runtime code manipulation (2004).

© 2025 Information Processing Society of Japan