2025年度　卒 業 論 文

Unikernelアプリケーション向けファイアウォール自動設定機構の実装

2026年1月30日

情報理工学科
学生証番号: 2254667

松山 慎

指導教員: 林原 尚浩

京都産業大学 情報理工学部

概 要

近年、クラウド環境におけるアプリケーション実行基盤として、軽量性と高い性能を両立する Unikernel が注目されている。Unikernel は、アプリケーションとその実行に必要な最小限の OS 機能のみを単一の実行イメージとして構成することで、起動時間の短縮やメモリ使用量の削減を実現する。一方で、不要な機能を排除できることから、攻撃面積を低減できるというセキュリティ上の利点も有している。しかし、Unikernel の構成は多数のライブラリ依存関係に基づいて自動的に決定されるため、最終的にどのライブラリや機能が実行イメージに含まれているのかを利用者が把握することは容易ではない。この結果、意図しないネットワーク機能や通信経路が有効化される可能性があり、設計段階での安全性判断が困難になるという課題が存在する。

本研究では、Linux 互換性を重視した Unikernel 開発基盤である Unikraft を対象として、実行イメージに含まれるライブラリ構成および依存関係を自動的に抽出し、可視化する手法を提案する。Unikraft では Kconfig による構成管理が採用されており、各ライブラリや機能の有効化は select、imply、depends on といった依存関係に基づいて決定される。本研究では、これらの依存関係を解析し、構成オプションをノード、依存関係をエッジとする有向グラフとして表現することで、構成決定の過程を視覚的に把握可能とした。これにより、従来はブラックボックス化していた Unikernel の構成要素間の関係を明確に示し、構成理解性の向上を図る。

さらに、本研究では、抽出した構成情報を運用時のセキュリティ制御に活用することを目的として、ファイアウォールと連携した通信制御方式を設計および実装した。具体的には、構成依存関係解析の結果およびアプリケーション設定ファイルを基に、Unikernel が実行時に必要とする通信ポートを特定し、ファイアウォール上で当該ポートが開放されているかを自動的に確認する。必要なポートが開放されていない場合には、iptables を用いて最小限の通信のみを許可するファイアウォールルールを自動的に生成および適用することで、通信を成立させると同時に不要な通信を遮断する。本方式により、ファイアウォール設定を事前に行っていない環境においても、安全に Unikernel を実行可能とすることを目指す。

提案手法の有効性を検証するため、外部公開サービスとして nginx、内部サービスとして redis を対象とした評価を行った。評価の結果、両アプリケーションにおいて、必要な通信ポートを正しく特定できること、ファイアウォール上でのポート開放状況を自動的に判別できること、ならびに必要に応じてファイアウォール設定を自動的に行い、通信を成立させられることを確認した。また、自動設定後において不要な通信ポートが開放されていないことも確認でき、提案方式が攻撃面積の削減に寄与することを示した。

本研究の成果により、Unikernel を「結果として安全」な実行基盤として扱うのではなく、構成情報と通信制御の対応関係を明示することで、「設計として安全」な実行基盤として扱うことが可能となる。本研究は、Unikernel の構成可視化と運用時セキュリティ制御を結び付けた点に新規性を有しており、今後のクラウドネイティブ環境における軽量仮想化技術の安全な運用に貢献する基盤的研究である。

目 次

第1章　序論

1.1　研究背景

1.1.1　クラウド環境における実行基盤の変遷

近年、クラウドコンピューティングの普及に伴い、アプリケーション実行基盤には高い性能、柔軟なスケーラビリティ、ならびにセキュリティが同時に求められている。従来広く利用されてきた仮想マシン（Virtual Machine: VM）は、完全なゲストオペレーティングシステムを含む構成により強固な分離性を実現している一方で、起動時間の長さやメモリ使用量の増大といったオーバーヘッドが課題として指摘されている。

一方、Docker に代表されるコンテナ技術は、ホスト OS のカーネルを共有することで高速な起動と高いリソース効率を実現し、クラウドネイティブ環境において広く普及している。しかし、カーネルを共有する構造上、コンテナ間およびホストへの攻撃影響が及ぶ可能性があり、セキュリティ境界の弱さが課題として挙げられている。

このように、性能とセキュリティを高い水準で両立可能な新たな実行基盤が求められている。

1.1.2　仮想化技術における Unikernel の位置付け

仮想化技術は、抽象化の粒度に応じて大きく三つに分類できる。VMware、VirtualBox、QEMU などはハイパーバイザ型仮想化に分類され、完全なゲスト OS を仮想化することで高い分離性を提供する。この方式では、各仮想マシンが独立した OS カーネルを持つため、セキュリティ境界は強固である一方、OS 起動や管理に伴うオーバーヘッドが大きいという課題がある。

Docker に代表されるコンテナ技術は OS レベル仮想化に分類され、ホスト OS のカーネルを共有することで軽量性と高速な起動を実現している。コンテナはプロセス単位で分離されるため、リソース効率に優れるが、カーネルを共有する構造上、分離性はハイパーバイザ型仮想化に比べて限定的である。

Unikernel はこれら二つの方式の中間的な位置付けにある実行基盤である。Unikernel では、アプリケーションと必要最小限の OS 機能を単一の実行イメージとして構成し、ハイパーバイザ上で直接動作する。このため、完全なゲスト OS を含まないにもかかわらず、VM と同等の分離性を確保できる。

さらに、Unikernel は不要なデーモンや汎用的な OS 機能を含まないため、実行イメージのサイズが小さく、起動時間も短い。このように、Unikernel は VM の持つ高い分離性と、コンテナの持つ軽量性を両立する実行基盤として位置付けられる。

表 1.1 に、VM、コンテナ、および Unikernel のアーキテクチャ的特徴を比較して示す。Unikernel は、ハイパーバイザ上で直接実行される単一イメージという構成により、高い分離性と軽量性を

同時に実現している点が特徴である。

表 1.1: 仮想化技術におけるアーキテクチャ比較

項目	VM	コンテナ	Unikernel
仮想化方式	ハイパーバイザ型	OS レベル仮想化	ハイパーバイザ上で直接実行
ゲスト OS の有無	あり（完全な OS）	なし	なし
カーネルの扱い	VM ごとに独立	ホスト OS と共有	必要最小限の機能のみを内包
実行単位	仮想マシン	プロセス（コンテナ）	単一実行イメージ
起動時間	遅い	速い	非常に速い
リソース使用量	大きい	小さい	非常に小さい
分離性	非常に高い	中程度	高い
攻撃面積	大きい	中程度	非常に小さい
既存アプリの互換性	高い	高い	基盤依存

図 1.1 は、仮想化技術における Virtual Machine（VM）、Container、および Unikernel のアーキテクチャ比較を示す。VM はハイパーバイザ上で完全なゲスト OS を実行するため高い分離性を持つが、OS 起動やカーネル機能に伴うリソースオーバーヘッドが大きい。Container はホスト OS のカーネルを共有し軽量性を高める一方、カーネル分離の弱さがセキュリティ境界に影響する。Unikernel はアプリケーションと必要な OS 機能を単一の実行イメージとして統合し、ハイパーバイザ上で直接動作するため、VM と同等の分離性を保ちながら、VM や Container よりも軽量な実行基盤となっている。図はこれらの違いをレイヤ構造として視覚的に示している。

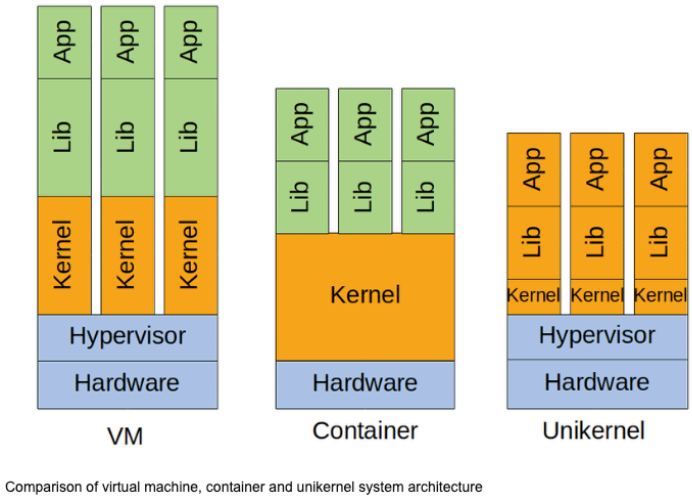

図 1.1: VM、コンテナ、Unikernel のアーキテクチャ比較（出典: Goethals et al. 2018）

1.2 Unikernel の概要と実装

1.2.1 Unikernel の基本概念

Unikernel は、アプリケーションコードと OS 機能を分離せず、単一の実行イメージとして構成する点に特徴がある。従来の OS が提供するプロセス管理やユーザ権限管理といった汎用的な機構を持たない代わりに、アプリケーションの実行に必要な最小限の機能のみを選択的に組み込むことで、構成の最小化を実現する。

従来の汎用 OS では、複数アプリケーションの同時実行を前提として、多数のデーモンやサービスが常駐する構成となっている。一方、Unikernel は単一アプリケーション専用の実行環境として設計されており、不要なサービスやバックグラウンド処理を一切含まない。その結果、実行イメージのサイズ削減、起動時間の短縮、メモリ使用量の低減といった利点が得られる。

また、Unikernel は単一アドレス空間で動作するため、ユーザ空間とカーネル空間の切り替えが存在しない。この構造により、システムコールやコンテキストスイッチに伴うオーバーヘッドが削減され、高い実行性能が期待できる。一方で、アプリケーションと OS 機能が同一アドレス空間に配置されるため、構成設計の妥当性が安全性に直結するという特徴を持つ。

セキュリティの観点では、攻撃対象となり得る領域を限定できるという利点を有する。不要な機能を含まない最小構成により、脆弱性が存在する可能性そのものを低減できる点は、従来の汎用 OS やコンテナ実行環境と比較して大きな特徴である。

一方で、Unikernel は従来の OS が備えるユーザ分離機構や動的な権限制御を持たないため、構成の誤りや不要な機能の混入は、そのままセキュリティリスクにつながる。このため、Unikernel を安全に運用するためには、実行イメージに含まれる機能を正確に把握し、意図しない機能が含まれていないことを確認することが重要となる。

1.2.2 Unikernel の代表的な実装

Unikernel は単一の実装に限られた技術ではなく、これまでに複数の開発基盤が提案されている。

MirageOS は OCaml 言語を用いた Unikernel 実装であり、高い型安全性と信頼性を特徴とする。一方で、対応言語が限定されており、既存の Linux アプリケーションを利用することは困難である。

IncludeOS は C++ を用いた Unikernel 実装であり、高速なネットワーク処理を特徴とするが、対応アプリケーションや用途は限定的である。

OSv は POSIX 互換性を重視した Unikernel 実装であり、既存アプリケーションの移植性に優れる一方で、構成が比較的大規模になりやすいという課題がある。

Unikraft は、既存の Linux アプリケーション資産の活用を重視した Unikernel 開発基盤である。Linux 互換インタフェースを提供することで、nginx や redis などの既存アプリケーションを比較的容易に Unikernel として実行できる点を特徴とする。また、OS 機能をライブラリ単位で構成可能とするライブラリ OS 型の設計を採用しており、必要最小限の機能のみを選択的に組み込む柔軟な構成管理を実現している。一方で、この柔軟性の高さにより、ライブラリ間の依存関係が複雑化し、最終的な実行イメージの構成把握が困難になるという課題を有している。

1.3 Unikraft と研究課題

1.3.1 Unikraft の特徴

Unikraft は Linux 互換性を重視した Unikernel 開発基盤であり、既存の Linux アプリケーションを比較的容易に Unikernel として実行できる点が特徴である。POSIX 互換インタフェースを提供することで、nginx や redis などの既存アプリケーションを大きな改変なしに利用でき、従来の Unikernel が抱えていたアプリケーション移植性の問題を大きく緩和している。

また、Unikraft はライブラリ OS 型の設計を採用しており、スケジューリング、メモリ管理、ネットワークスタック、ファイルシステムといった OS 機能をライブラリとして分離して提供する。この構成により、アプリケーションの実行に必要な機能のみを選択的に組み込むことが可能となり、Unikernel の利点である軽量性と最小構成を柔軟に実現できる。

さらに、Unikraft では Kconfig を用いた構成管理機構が採用されており、各ライブラリや機能の依存関係が select、imply、depends on によって記述されている。これにより、複雑な依存関係を持つ構成であっても自動的に整合性を保ったビルドが可能となり、利用者は比較的容易に Unikernel イメージを生成できる。

一方で、この自動化された構成管理は、最終的にどのライブラリや機能が実行イメージに含まれているのかを利用者が直感的に把握しにくいという側面も持つ。特に、select や imply によって暗黙的に有効化される構成オプションは、利用者の明示的な指定とは独立して組み込まれるため、実行イメージの内部構成がブラックボックス化しやすい。

このように Unikraft は、高い実用性と柔軟な構成管理を両立した Unikernel 開発基盤である一方で、構成理解性やセキュリティ評価の観点において新たな課題を内包している。

1.3.2 構成管理に関する課題

Unikraft では Kconfig を用いた構成管理機構が採用されており、ライブラリや機能の依存関係は select、imply、depends on によって記述される。しかし、これらの依存関係は自動的に解決されるため、最終的な実行イメージに含まれる構成要素を利用者が把握することは容易ではない。

特にネットワーク関連ライブラリは外部通信を可能にするため、構成把握が不十分な場合、Unikernel の攻撃面積が意図せず拡大する可能性がある。

1.4 研究目的と貢献

1.4.1 研究目的

本研究の目的は、Unikraft を対象として、Unikernel 実行イメージに含まれるライブラリ構成および依存関係を自動的に抽出・解析し、それらを可視化することで構成理解性を向上させることである。さらに、抽出した構成情報を用いて、実行時に必要な通信ポートが適切に開放されているかを確認し、必要に応じてファイアウォールを自動的に設定する機構を実現することを目的とする。

1.4.2 本研究の貢献

本研究の主な貢献は以下の三点である。

1. Unikraft における Kconfig 依存関係を対象とした、構成情報の自動抽出および可視化手法を提案した点。
2. 構成情報に基づいてネットワーク関連ライブラリの利用状況を判別し、ファイアウォールと連携した通信制御方式を設計・実装した点。
3. 実アプリケーションを用いた評価により、提案手法の有効性を示した点。

これらの貢献により、従来は利用者が把握することが困難であった Unikernel 実行イメージの内部構成について、「どのライブラリが、どの依存関係に基づいて有効化されたのか」を説明可能となった。特に、Kconfig による自動的な構成決定の結果を可視化することで、構成情報のブラックボックス化を解消し、Unikernel が最小構成であるかどうかを客観的に評価できるようになった。

また、可視化された構成情報を基にネットワーク関連ライブラリの利用有無を判別し、実際に必要な通信ポートのみをファイアウォール上で許可する仕組みを導入したことで、Unikernel 実行時の通信要件を事前に検証し、意図しない通信を防止する運用が可能となった。これにより、ファイアウォール設定を人手で行うことなく、構成に即した通信制御を自動的に適用できるようになった。

さらに、本手法は特定のアプリケーションに依存せず、Unikraft の構成管理機構に基づいて動作するため、今後さまざまな Unikernel アプリケーションに対して同様の安全性評価および通信制御を適用できる基盤を提供するものである。

1.5 本論文の構成

1.5.1 論文全体の構成

本論文は全 7 章から構成される。第 2 章では Unikernel および関連技術に関する研究を概観する。第 3 章では Unikraft の構成管理機構と依存関係の課題を整理する。第 4 章では構成依存関係の抽出および可視化手法を提案する。第 5 章では構成情報に基づくファイアウォール自動設定機構について述べる。第 6 章では nginx および redis を用いた評価を行い、第 7 章で結論と今後の課題を述べる。

第2章　関連研究

2.1　Unikernel に関する研究

2.1.1　Unikernel の概念と特徴

Unikernel は、アプリケーションとその実行に必要な最小限の OS 機能のみを単一アドレス空間に統合した実行イメージとして構成される仮想化技術である [1]。従来の汎用オペレーティングシステムでは、多数のサービスやデバイスドライバが常駐して動作するのに対し、Unikernel では用途に不要な機能を排除した特化型構成を採用する点に特徴がある。

これにより、起動時間の短縮やメモリ使用量の削減が可能となり、クラウド環境における軽量な実行基盤として注目されてきた。また、Unikernel は仮想マシン上で動作するため、ハイパーバイザによる分離性を維持しつつ、コンテナ技術に近い軽量性を実現できる点が評価されている [5][6]。

一方で、Unikernel は単一アドレス空間で動作するため、従来 OS が備えるプロセス分離やユーザ権限管理といった防御機構を持たない。そのため、構成設計や含まれる機能の妥当性が、実行時の安全性に直接影響を与えるという特徴を有する。

2.1.2　クラウド環境における Unikernel の利用

クラウド環境では、高速なスケールアウトや高密度なサービス配置が求められる。このような要求に対し、Unikernel の高速起動性や低リソース消費特性は大きな利点となる。先行研究では、Web サーバやマイクロサービスを Unikernel として実装し、仮想マシンやコンテナと比較した性能評価が行われている [2][5]。

これらの研究において、Unikernel は起動時間やメモリ使用量の面で優位性を示すことが報告されている。一方で、デバッグ手法や運用管理手法が十分に成熟していない点や、構成変更の柔軟性に欠ける点が課題として指摘されている。特に、構成が静的に決定されることにより、実行イメージに含まれる機能を運用者が把握しにくいという問題が存在する。

2.2　Unikernel のセキュリティに関する研究

2.2.1　攻撃面積削減に関する研究

Unikernel のセキュリティ上の利点として、攻撃面積（Attack Surface）の削減が挙げられる。従来の OS では、多数のサービスやドライバが動作しており、それらが攻撃対象となる可能性がある。一方、Unikernel では不要な機能を排除した最小構成を採用することで、脆弱性が存在する可能性のあるコード量を削減できる [1][6]。

このような特性から、Unikernel は理論的には高い安全性を有する実行基盤であると評価されてきた。また、VM 上で動作する点から、ハイパーバイザによる分離性を活用できる点も利点として挙げられている。

2.2.2 Unikernel におけるセキュリティ課題

一方で、Unikernel には特有のセキュリティ課題も存在する。単一アドレス空間で動作する構造上、従来 OS のようなプロセス分離が行えず、一度脆弱性が悪用された場合の影響が大きくなる可能性がある [7]。

また、Unikernel は構成が静的に決定されるため、実行イメージに含まれる機能やライブラリの把握が不十分な場合、意図しない機能が有効化される可能性がある。特に、ネットワーク機能に関連するライブラリは外部通信を可能にするため、構成管理が不十分な場合、Unikernel の攻撃面積が想定以上に拡大する恐れがある。このため、構成情報を正確に把握し、安全性を説明可能とする手法の重要性が指摘されている。

2.3 構成管理および依存関係解析に関する研究

2.3.1 Kconfig による構成管理

Kconfig は、Linux カーネルをはじめとする多くのソフトウェアで利用されている構成管理機構であり、構成オプションとその依存関係を記述するために用いられる。Kconfig では、depends on による前提条件の指定や、select、imply による構成オプション間の有効化関係を定義できる [4]。

この仕組みにより、構成の整合性を自動的に保つことが可能となる一方、依存関係が複雑化しやすく、最終的にどの構成オプションが有効化されたのかを利用者が直感的に理解することは困難であることが報告されている。

2.3.2 構成依存関係の可視化に関する研究

構成依存関係の理解を支援するため、依存関係をグラフ構造として可視化する研究がこれまでに提案されている。これらの研究は、ソフトウェア構成の理解性向上や保守性改善を目的としており、大規模ソフトウェアを対象とした事例が多い。しかし、Unikernel を対象とした構成依存関係の可視化研究は限定的であり、特に構成情報を運用時のセキュリティ制御へ直接活用する研究は十分に行われていない。

2.4 ネットワーク通信制御とファイアウォール

2.4.1 ファイアウォールを用いた通信制御

ファイアウォールは、ネットワーク通信を制御する代表的なセキュリティ機構であり、iptables などのツールを用いて通信の許可および遮断を制御できる。従来のサーバ環境やコンテナ環境においては、最小権限通信を実現するためのファイアウォール設定が広く用いられている。

2.4.2　ファイアウォール自動化の課題

ファイアウォールの設定は手動で行われることが多く、設定ミスや過剰な許可ルールがセキュリティリスクとなることが指摘されている。近年では、アプリケーションの構成情報やポリシーに基づいて ファイアウォールを自動設定する研究も進められているが、Unikernel の構成情報と直接連携した手法は限定的である。

2.5　本研究の位置づけ

本研究は、Unikraft を対象として Unikernel の構成情報を自動的に抽出し、Kconfig に基づく依存関係を可視化する点に特徴がある。さらに、可視化によって得られた構成情報を用いてネットワーク関連ライブラリの利用状況を判別し、ファイアウォールと連携した通信制御を自動化する点に新規性がある。

既存研究が Unikernel の軽量性や理論的な安全性に着目しているのに対し、本研究は「構成の可視化」と「運用時の通信制御」を結び付けることで、実用的なセキュリティ向上手法を提案するものである。

第3章　Unikraftの構成と依存関係

3.1　Unikraft の全体構成

3.1.1　Unikraft のアーキテクチャ概要

Unikraft は、アプリケーションと OS 機能をライブラリとして提供するライブラリ OS 型の Unikernel 開発基盤である。Unikraft では、アプリケーション本体、共通ライブラリ、プラットフォーム依存コードが明確に分離されており、必要な機能のみを選択的に組み込むことが可能である。

実行イメージは、アプリケーションコードと選択されたライブラリ群を静的にリンクすることで生成され、単一のアドレス空間で動作する。この構成により、不要な機能を排除した軽量な実行環境を実現している。

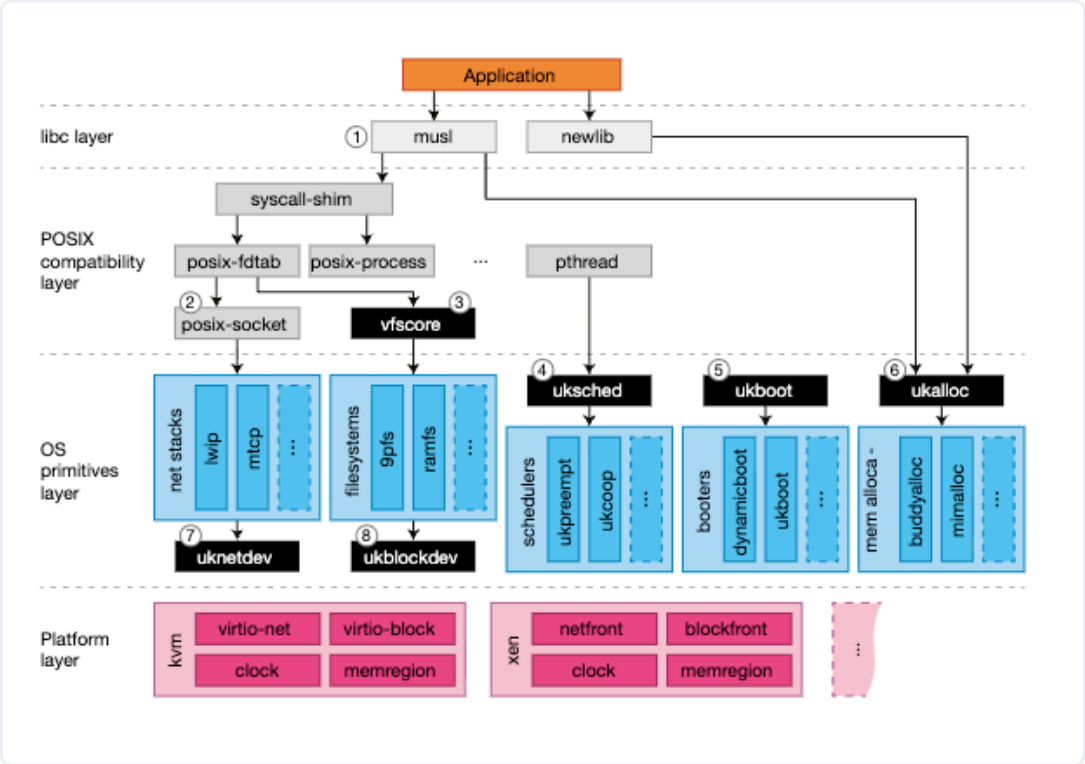

図 3.1: Unikraft におけるライブラリ OS 型アーキテクチャ

図 3.1 は、Unikraft の全体アーキテクチャを示す。Unikraft はライブラリ OS 型の Unikernel 開発基盤であり、従来の汎用 OS のように固定的なカーネル構成を持つのではなく、OS 機能を細粒度なライブラリとして提供する点に特徴がある。

図に示すように、Unikraft の実行イメージは、アプリケーション本体と、コア機能を提供する共通ライブラリ、ハイパーバイザや仮想化環境に依存するプラットフォームライブラリ、および CPU アーキテクチャに依存するライブラリから構成される。これらのライブラリは、Kconfig に基づく構成管理機構によって選択され、ビルド時に静的にリンクされることで、単一の実行イメージが生成される。

このような構成により、Unikraft では不要な OS 機能を実行イメージから排除でき、軽量かつ高速な起動を実現できる。一方で、ライブラリ間の依存関係は Kconfig の select、imply、depends on によって自動的に解決されるため、最終的にどのライブラリが実行イメージに含まれているかを利用者が把握することは容易ではない。

本研究では、この図に示されるライブラリ構成と依存関係に着目し、Unikraft の構成情報を機械的に抽出・可視化することで、構成決定過程の透明化を図る。

3.1.2 Unikraft におけるライブラリ構成

Unikraft では，メモリ管理，スケジューリング，割り込み処理，ファイルシステム，ネットワーク機能などの OS 機能が，それぞれ独立したライブラリとして提供されている。アプリケーションは，必要な機能に対応するライブラリを選択することで，実行に必要な OS 機能を獲得する。

しかし，これらのライブラリは互いに独立して存在するわけではなく，多くの場合，複数の下位ライブラリへの依存関係を持つ。例えば，ネットワーク機能を利用する場合，TCP/IP スタック，ネットワークデバイス抽象化層，ソケットインタフェースなど，複数のライブラリが連鎖的に有効化される。

3.2 Kconfig による構成管理

3.2.1 Kconfig の基本概念

Unikraft では，Linux カーネルと同様に Kconfig を用いた構成管理機構が採用されている。Kconfig は、ソフトウェアの構成オプションとその依存関係を記述するための仕組みであり、Linux カーネルをはじめとする多くのソフトウェアで利用されている。Unikraft においても、各ライブラリや機能は Kconfig によって定義されており、構成オプションとして管理されている。

構成オプションは真偽値や選択肢として定義され、ユーザの選択や他のオプションとの依存関係に基づいて有効・無効が決定される。

3.2.2 select、imply、depends on の役割

Kconfig における依存関係は、主に select、imply、depends on の三種類に分類される。depends on は、ある構成オプションが有効化されるための前提条件を表し、条件を満たさない場合、当該オプションは選択できない。

select は、ある構成オプションが有効化された際に、別の構成オプションを強制的に有効化する依存関係を表す。一方、imply は select と類似するが、強制ではなく推奨的に有効化を行う点で異なる。

これらの依存関係は構成の整合性を保つ上で有用である一方、構成決定の過程を利用者が直感的に理解することを困難にする要因ともなっている。

3.3 構成依存関係の複雑性

3.3.1 依存関係の連鎖と構成決定

Unikraft においては、一つのライブラリを有効化することで、複数の下位ライブラリが連鎖的に有効化される場合がある。特に select による依存関係は強制的であるため、利用者が意図しない構成オプションが自動的に有効化される可能性がある。

このような依存関係の連鎖により、最終的な実行イメージに含まれるライブラリ構成は、利用者が明示的に指定した内容と一致しない場合がある。

3.3.2 構成情報のブラックボックス化

Kconfig による構成管理は自動化に優れる一方で、最終的にどの構成オプションが有効化されたのかを一覧的に把握することは容易ではない。特に、select や imply によって有効化されたオプションは、利用者の明示的な操作とは独立して決定されるため、構成情報のブラックボックス化が生じやすい。

このような状況では、Unikernel が最小構成であるかどうかを判断することが困難となり、セキュリティ上の懸念につながる。

3.4 ネットワーク関連ライブラリと依存関係

3.4.1 Unikraft におけるネットワークスタック

Unikraft では、ネットワーク機能は複数のライブラリによって構成されている。例えば、ネットワークデバイス抽象化層、TCP/IP スタック、ソケットインタフェースなどがそれぞれ独立したライブラリとして提供されている。

これらのライブラリは相互に依存関係を持ち、上位のネットワーク機能を有効化することで、下位のライブラリが自動的に組み込まれる。

3.4.2 セキュリティ観点からの問題点

ネットワーク機能は外部との通信を可能にするため、攻撃対象となりやすい要素である。構成依存関係を十分に把握しないままネットワーク関連ライブラリを有効化した場合、不要な通信機能が実行イメージに含まれる可能性がある。

このような不要な通信機能の存在は、攻撃面積の拡大につながり、Unikernel の安全性を低下させる要因となる。そのため、ネットワーク関連ライブラリの構成把握は、Unikernel を安全に運用する上で重要な課題である。

3.5　本章のまとめ

3.5.1　課題の整理

本章では、Unikraft の構成管理機構とライブラリ依存関係の特徴について述べた。Unikraft は柔軟な構成管理を可能とする一方で、Kconfig に基づく依存関係の複雑性により、最終的な構成を利用者が把握することが困難であるという課題を有している。

特に、ネットワーク関連ライブラリは外部通信に直結するため、構成把握が不十分な場合、意図しない攻撃面積の拡大を招く可能性がある。

3.5.2　次章へのつながり

次章では、本章で整理した課題を踏まえ、Unikraft の構成情報を自動的に抽出し、依存関係を可視化する手法について詳述する。これにより、構成の理解性向上とセキュリティ評価への応用を目指す。

第4章　構成依存関係の抽出と可視化手法

4.1　本章の目的

4.1.1　解決すべき課題

第3章で述べたように、Unikraft では Kconfig に基づく多数のライブラリ依存関係により、最終的な Unikernel 実行イメージの構成を利用者が把握することが困難である。特に、select や imply によって暗黙的に有効化される構成オプションは、構成決定の過程を不透明にし、構成情報のブラックボックス化を引き起こす要因となっている。

このような状況では、実行イメージに含まれる機能が「なぜ有効化されたのか」を説明することが難しく、最小構成であるかどうかの判断や、セキュリティ観点での評価を行うことが困難となる。したがって、構成情報を人手で解析することなく、機械的に抽出・整理し、依存関係を可視化する手法が求められる。

4.1.2　提案手法の概要

本章では、Unikraft により生成される構成情報および各ライブラリに含まれる Kconfig ファイルを解析対象とし、構成オプション間の依存関係を自動的に抽出・可視化する手法を提案する。

提案手法では、depends on、select、imply といった依存関係を区別して抽出し、構成オプションをノード、依存関係をエッジとする有向グラフとして表現する。これにより、構成要素間の関係を一覧的に把握可能とし、後続章で述べる通信制御への応用を可能とする。

4.2　構成情報の抽出

4.2.1　解析対象となる構成情報

Unikraft における構成管理では、ビルド時に各ライブラリが提供する Kconfig ファイルおよび、最終的な構成結果を含む設定情報が生成される。本研究では、これらのうち、各ライブラリに含まれる Kconfig ファイルを主な解析対象とする。

Kconfig ファイルには、構成オプションの定義とともに、depends on、select、imply といった依存関係が明示的に記述されている。これらの記述を機械的に解析することで、構成オプション間の関係を抽出できると考える。

4.2.2 構成オプションの抽出方法

構成オプションは、Kconfig ファイル中の config エントリとして定義されている。本研究では、Kconfig ファイルを走査し、各 config エントリの名称を構成オプションとして抽出する。

さらに、各構成オプションに対して、依存関係として記述されている depends on、select、imply の内容を解析し、オプション間の関係を整理した。抽出結果は、構成オプション名と依存関係の組を基本単位とする内部データ構造として保持し、後続の解析および可視化処理に利用する。

4.3 依存関係の解析

4.3.1 depends on の解析

depends on は、構成オプションが有効化されるための前提条件を示す依存関係である。本研究では、depends on に記述された条件式を解析し、どの構成オプションに依存しているかを抽出する。

抽出結果は、「ある構成オプションが有効でなければ当該オプションは有効化されない」という制約関係として扱う。

4.3.2 select および imply の解析

select は、ある構成オプションが有効化された際に、別のオプションを強制的に有効化する依存関係である。一方、imply は select と類似するが、強制性を持たない点で異なる。

本研究では、select および imply を「構成オプション間の有効化関係」として抽象化し、依存関係の種類を区別した上で解析を行う。これにより、構成要素がどのような理由で有効化されたのかを識別可能とする。

4.4 依存関係グラフの生成

4.4.1 グラフモデル

抽出した構成オプションおよび依存関係は、有向グラフとして表現する。各ノードは構成オプションを表し、エッジは依存関係を表す。

エッジの向きは、「依存元から依存先」へ向かうものとし、depends on、select、imply の種類に応じて区別する。このようなグラフモデルを用いることで、構成全体の構造を直感的に把握できるだけでなく、特定の構成オプションを起点とした依存関係の追跡が可能となる。

4.4.2 依存関係の種類の可視化

依存関係の可視化においては、depends on、select、imply を異なる表現で描画することで、構成の決定要因を視覚的に識別できるようにする。

例えば、強制的な有効化を伴う select と、条件付きの依存関係である depends on を区別することで、構成に与える影響の大きさを直感的に理解可能とする。

4.5　可視化結果の活用

4.5.1　構成理解の支援

生成された依存関係グラフにより、Unikraft の構成要素間の関係を一覧的に把握することが可能となる。これにより、どの構成オプションがどの機能に起因して有効化されたのかを追跡できる。

特に、ネットワーク関連ライブラリに着目することで、外部通信に関与する構成要素を特定しやすくなる。

4.5.2　セキュリティ評価への応用

可視化された依存関係は、Unikernel の攻撃面積評価にも利用可能である。不要なネットワーク関連構成要素が含まれている場合、それらを特定し、構成の見直しや通信制御の検討につなげることができる。

このように、本手法は構成の理解性向上だけでなく、セキュリティ向上のための基盤情報を提供する。

4.6　本章のまとめ

4.6.1　提案手法の有効性

本章では、Unikraft の構成情報を自動的に抽出し、Kconfig に基づく依存関係を可視化する手法を提案した。本手法により、従来把握が困難であった構成要素間の関係を明確に示すことが可能となる。

4.6.2　次章へのつながり

次章では、本章で得られた構成情報を用いて、ネットワーク関連ライブラリの利用状況を判別し、ファイアウォールと連携した通信制御方式の設計および実装について述べる。

第5章　構成情報に基づくファイアウォール自動設定

5.1　本章の目的

5.1.1　研究背景との関係

第 4 章では、Unikraft における構成依存関係を自動的に抽出し、可視化する手法について述べた。本章では、その可視化によって得られた構成情報を、実際の運用時セキュリティ制御に活用することを目的とする。

Unikernel は最小構成を志向する実行基盤であるが、構成が自動生成されるという特性上、利用者が意図しないネットワーク機能が有効化される可能性がある。このような状況では、Unikernel の安全性を「結果として」評価することはできても、「設計として」説明することは困難である。本研究では、構成情報に基づいて通信制御を自動化することで、この問題の解決を図る。

5.1.2　本章で扱う範囲

本章では、Unikraft により生成された Unikernel 実行イメージを対象として、(1) ネットワーク関連ライブラリの利用状況の判別 (2) 利用ポートの特定 (3) iptables を用いたファイアウォール制御の自動化について述べる

5.2　ネットワーク関連ライブラリの判別

5.2.1　ネットワーク機能と構成情報

Unikraft におけるネットワーク機能は、複数のライブラリによって構成されている。具体的には、ネットワークデバイス抽象化層、TCP/IP スタック、ソケットインタフェースなどが独立した構成要素として提供される。

これらのライブラリが有効化されることで、Unikernel は外部との通信能力を獲得する。したがって、ネットワーク関連ライブラリの有無は、Unikernel が外部通信を行うかどうかを判断する重要な指標となる。

5.2.2　ネットワーク関連ライブラリの特定方法

本研究では、第 4 章で生成した構成依存関係グラフを用いて、ネットワーク機能に関与する構成オプションを特定する。具体的には、TCP/IP スタックやソケット API に対応する構成オプションを起点とし、それらに依存する下位ライブラリを含めて抽出する。

この手法により、実行イメージにネットワーク機能が含まれているかどうかを、構成情報に基づいて判別可能となる。さらに、ネットワーク機能が不要な構成に対しては、外部通信を完全に遮断する ファイアウォール設定を適用できる。

5.3 利用ポートの抽出

5.3.1 アプリケーションと通信ポート

ネットワーク通信の制御においては、単にネットワーク機能の有無を判別するだけでなく、アプリケーションが実際に使用する通信ポートを特定することが重要である。不要なポートを許可した場合、攻撃対象が増加する可能性がある。

本研究では、アプリケーション設定および構成情報を基に、使用される通信ポートを抽出する。

5.3.2 ポート情報の取得方法

通信ポートの取得には、アプリケーションの設定ファイルおよび実行時情報を用いる。例えば、Web サーバの場合には、設定ファイルに記述された待受ポート番号を解析対象とする。

これにより、実行イメージが使用する通信ポートを事前に把握し、ファイアウォール設定に反映することが可能となる。

5.4 Linux におけるファイアウォールの実装

5.4.1 iptables を用いた通信制御

Linux 環境におけるファイアウォール機構として、iptables は長年にわたり広く利用されてきた代表的なパケットフィルタリング方式である。iptables は Linux カーネルに組み込まれた Netfilter フレームワークを操作するためのユーザ空間ツールであり、カーネルレベルで通信パケットを検査し、許可または遮断を行う仕組みを提供する。

Netfilter は、パケットの送受信経路上に複数のフックポイントを設け、それぞれの段階でパケット処理を行う構造を持つ。iptables は、これらのフックポイントに対してルールを登録することで、通信制御を実現している。主なフックポイントとして、INPUT、OUTPUT、FORWARD が存在し、それぞれ受信パケット、送信パケット、転送パケットに対する制御を担当する。

iptables では、ルールはテーブル、チェイン、ルールという三層構造で管理される。テーブルはルールの目的に応じた分類単位であり、filter、nat、mangle などが存在する。本研究で主に利用する filter テーブルは、通信の許可・遮断といった基本的なアクセス制御を目的としたものである。

チェインは、パケットが通過する処理経路を表し、filter テーブルでは INPUT、OUTPUT、FORWARD が標準的に用意されている。各チェインには複数のルールが定義され、パケットは上から順にルールと照合される。条件に一致したルールが適用されると、その時点で処理が確定する。

iptables のルールは、送信元アドレス、宛先アドレス、通信プロトコル、ポート番号、接続状態など、さまざまな条件を組み合わせて記述できる。特に、conntrack モジュールを用いたステートフルフィルタリングにより、確立済み通信や関連通信のみを許可する制御が可能である。これにより、必要最小限の通信のみを許可しつつ、利便性と安全性を両立できる。

本研究では、Unikernel 実行前に iptables の filter テーブルを対象としてルールを自動的に設定し、デフォルトで通信を遮断した上で、必要な通信ポートのみを明示的に許可する方式を採用する。このようなホワイトリスト型の制御により、Unikernel の攻撃面積を最小化し、安全な実行環境を構築することを目的とする。

また、iptables はシステム全体に影響を及ぼすため、既存の通信やサービスに対する影響を考慮する必要がある。そのため、本研究では設定適用前に既存ルールを保存し、実験終了後に元の状態へ復元する仕組みを導入することで、実験環境の安全性と再現性を確保している。

5.4.2 自動設定方式の概要

提案方式では、以下の手順でファイアウォール制御を行う。

1. 構成依存関係解析結果からネットワーク機能の有無を判別
2. アプリケーション設定から利用ポートを抽出
3. 必要な通信のみを許可する iptables ルールを生成
4. Unikernel 実行前にファイアウォールルールを適用

この一連の処理を自動化することで、人手による設定ミスを防止する。

5.5 実装

5.5.1 実装環境

本研究で提案する方式は、Unikraft により生成された Unikernel 実行イメージを対象として実装した。Unikernel は QEMU 上で起動し、通信制御は Unikernel 内部ではなく、実行ホストである Linux OS 上のファイアウォール機構を用いて行う。

実験環境は以下の構成とした。

- ホスト OS：Linux（Ubuntu 系ディストリビューション）
- Unikernel 実行基盤：Unikraft + QEMU
- ファイアウォール機構：iptables
- 実装言語：Python 3

提案方式の実装は、Python により記述した単一のスクリプトとして構成されており、Unikernel のビルド、構成情報の解析、通信ポートの特定、iptables ルールの生成および適用を一連の処理として自動化している。

5.5.2 実装全体の処理フロー

実装したスクリプトは、以下の手順で処理を行う。

1. Unikraft による Unikernel のビルドを実行し、構成情報を生成する。
2. ビルド結果からネットワーク関連ライブラリの使用有無を判別する。
3. アプリケーション設定ファイルを解析し、Unikernel 内部で使用される待受ポートを抽出する。
4. Unikernel 実行時にホスト側で公開される通信ポートを特定する。
5. 現在の iptables ルールを保存する。
6. 必要最小限の通信のみを許可する iptables ルールを生成し、適用する。
7. Unikernel を QEMU 上で起動する。
8. 実験終了後、iptables ルールを元の状態へ復元する。

以下では、各処理について詳細に述べる。

5.5.3 Unikernel のビルドおよび構成情報の取得

スクリプトの最初の処理として、`kraft build` コマンドを実行し、Unikraft による Unikernel のビルドを行う。この処理により、`.unikraft/` および `.unikraft/build/` 以下に構成情報およびビルド成果物が生成される。

本研究では、これらのディレクトリに含まれる `Config.uk` ファイルを解析対象とし、Unikernel 実行イメージに含まれるライブラリ構成を取得する。

5.5.4 ネットワーク関連ライブラリの判別

Unikernel が外部通信を行うかどうかを判別するため、`.unikraft/` 以下に存在する `Config.uk` ファイルを再帰的に走査する。

各 `Config.uk` に対して、TCP/IP スタック、ネットワークデバイス、ソケット API に関連する構成オプションが含まれているかを文字列検索により判別する。

ネットワーク関連ライブラリが一つも検出されなかった場合、当該 Unikernel は外部通信を行わないと判断し、ファイアウォール制御処理を行わずに終了する。

5.5.5 アプリケーション設定ファイルからのポート抽出

ネットワーク関連ライブラリが検出された場合、次にアプリケーション設定ファイルを解析し、Unikernel 内部で使用される通信ポートを抽出する。

例えば、nginx を対象とした場合、`nginx.conf` に記述された `listen` ディレクティブを正規表現により解析し、待受ポート番号を抽出する。

この処理により、Unikernel 内部でアプリケーションが待ち受けるポート番号を特定する。

5.5.6 ホスト側公開ポートの特定

Unikernel は QEMU 上で実行されるため、外部からの通信はホスト側のポートフォワーディングを介して行われる。

本研究では、Unikernel 起動時に使用する `kraft run` コマンド中の `-p ホスト側:ゲスト側` オプションを解析し、ホスト側で公開されるポート番号を特定する。

ファイアウォール制御の対象となるのは、Unikernel 内部の待受ポートではなく、ホスト側で外部に公開されるポートであるため、本処理は通信制御において重要な役割を果たす。

5.5.7 iptables ルールの保存と自動生成

iptables の設定変更が既存の通信へ影響を与えないよう、スクリプトは処理開始時に `iptables-save` を用いて現在のファイアウォールルールを一時ファイルへ保存する。

その後、以下の方針に基づいて新たな iptables ルールを生成する。

- デフォルトポリシーは DROP とする。
- ループバック通信は許可する。
- 既存の管理用通信（SSH 等）は遮断しない。
- 外部通信に必要な最小限のポートのみを許可する。

生成されたルールは `iptables` コマンドを用いて順次適用される。

5.5.8 Unikernel の起動とファイアウォール復元

ファイアウォールルール適用後、`kraft run` コマンドを用いて Unikernel を QEMU 上で起動する。

実験終了後、もしくは異常終了が発生した場合でも、`iptables-restore` を用いて保存済みのルールを復元し、ホスト OS のファイアウォール状態を元に戻す。

この仕組みにより、実験環境の安全性および再現性を確保している。

5.6 本章のまとめ

5.6.1 得られた知見

本章では、Unikraft の構成情報を基に、ファイアウォールと連携した通信制御方式を設計・実装した。提案方式により、Unikernel 実行時に必要最小限の通信のみを許可することが可能となる。

5.6.2 次章へのつながり

次章では、本章で提案したファイアウォール自動設定方式を適用し、攻撃面積削減効果および実行への影響を評価する。

第6章　評価

6.1　評価の目的と方針

6.1.1　評価目的

本章では、第5章で提案した構成情報に基づくファイアウォール自動設定方式について評価を行う。本研究における評価の目的は、アプリケーションの実行に必要な通信ポートがファイアウォール上で適切に開放されているかを確認し、必要なポートが開放されていない場合に、自動的にファイアウォール 設定を行えるかを検証することである。

具体的には、以下の点を評価観点とする。

1. Unikraft により生成された Unikernel 実行イメージに対して、アプリケーションが必要とする通信ポートを正しく特定できるか。
2. Unikernel 起動前のファイアウォール状態において、必要な通信ポートが既に開放されているかどうかを自動的に判別できるか。
3. 必要なポートが開放されていない場合に、ファイアウォール設定を自動的に追加・更新し、通信を成立させられるか。
4. 自動設定後において、不要な通信ポートが開放されていないことを確認できるか。

これにより、提案方式がファイアウォール設定を事前に行わずとも、Unikernel を安全に実行できるかを明らかにする。

6.1.2　評価方針

評価には、性質の異なる二種類のアプリケーションを用いた。

1. nginx：外部からのアクセスを想定した Web サーバ。
2. redis：外部公開を想定しない内部サービス。

これらを Unikraft 上で Unikernel として実行し、ファイアウォール設定が存在しない、または不十分な状態を初期状態として、

1. 必要なポートが検出されるか
2. ファイアウォールが自動的に設定されるか
3. 実際に通信が成立するか

を確認することで、提案手法の有効性を評価する。

6.2　評価環境

6.2.1　ハードウェアおよびソフトウェア環境

評価は以下の環境で実施する。

1. ホスト OS：Linux
2. 仮想環境：QEMU
3. Unikernel 開発基盤：Unikraft
4. ファイアウォール制御機構：iptables
5. 通信確認ツール：nmap

本評価では、Unikernel は QEMU 上で起動し、ファイアウォール制御はホスト OS 上で適用する構成とした。

6.2.2　評価手順の概要

評価は以下の手順で実施する。

1. 対象アプリケーションを Unikraft により Unikernel としてビルド
2. 構成依存関係を抽出・可視化
3. アプリケーションが必要とする通信ポートを構成情報および設定ファイルから抽出
4. Unikernel 起動前にファイアウォールの状態を確認
5. 必要なポートが開放されていない場合、ファイアウォールを自動設定
6. Unikernel を起動し、通信が成立するかを確認

6.3　ケース 1：nginx Unikernel

6.3.1　対象アプリケーションの概要

nginx は代表的な Web サーバであり、HTTP 通信を通じて外部からのアクセスを受け付ける。本評価では、nginx を Unikernel として実行し、外部公開サービスを想定した通信制御が適切に行えるかを検証する。

6.3.2 構成依存関係の可視化結果

第 4 章で提案した手法を用いて、nginx Unikernel の構成依存関係を可視化した。図 6.1 に、ビルド成果物（`.unikraft/build/` 以下の `.o`）に実際に含まれたライブラリ集合を起点として、`Config.uk` から抽出した依存関係（`select`、`imply`、`depends on`）を有向グラフとして表現した結果を示す。

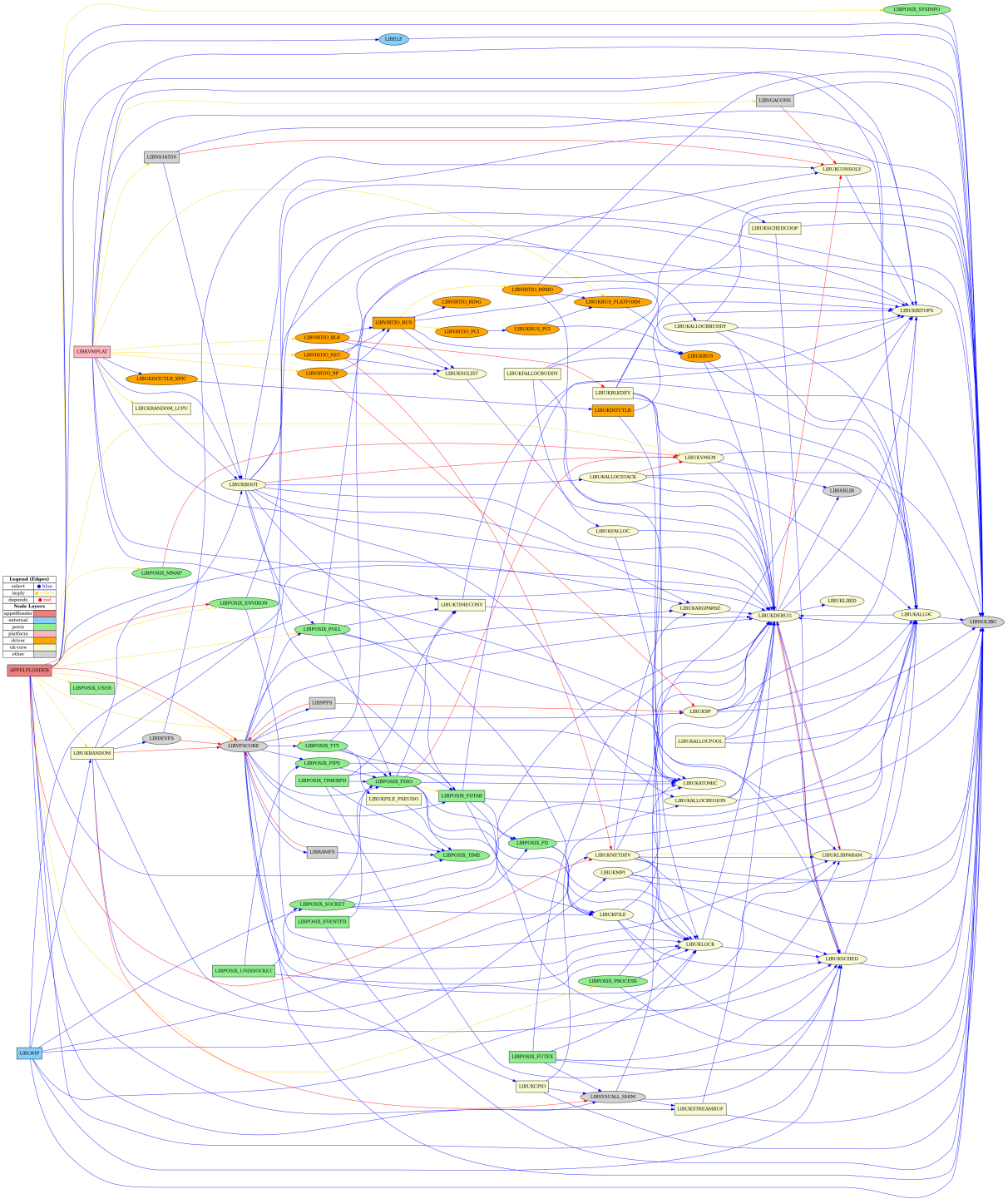

図 6.1: nginx Unikernel におけるライブラリ依存関係の可視化結果

図 6.1 では、ノードはライブラリを表し、エッジは構成上の依存関係を表す。具体的には、`select` は「あるオプションを有効化すると別のオプションが自動的に有効化される関係」であり、`imply` は「推奨として有効化を促す関係」、`depends on` は「成立に必要な前提条件を要求する関係」である。また、ノードは層構造（app、external、posix、platform、driver、uk-core）に基づいて色分けしており、上位の機能が下位層へどのように依存しているかを一目で追えるようにしている。

まず、nginx は外部からの HTTP 要求を受け付けるため、TCP/IP 通信とソケット API を必要とする。図中では、external 層として `lwIP` が含まれており、これが TCP/IP スタックとして動作することを示す。さらに、posix 層として `LIBPOSIX_SOCKET` や `LIBPOSIX_UNIXSOCKET` などのノードが存在しており、アプリケーションが POSIX 互換 API を通して通信機能を利用する構成になっていることが分かる。このように、アプリケーション側の要求が、posix 層を介して lwIP に接続される形で表現されており、nginx がネットワーク機能を必要とすることが構成の観点から確認できる。

次に、ネットワークスタックが実際にパケットを送受信するためには、デバイスドライバおよびプラットフォーム依存処理が必要となる。図中では driver 層として `VIRTIO` 関連ノードや `UKNETDEV` に相当するノードが含まれており、仮想 NIC を用いて通信を実現する構成になっていることが読み取れる。さらに、platform 層として `KVMPLAT` が含まれていることから、本評価環境（QEMU/KVM）上で動作させるための依存関係が有効化されていることが分かる。この一連の経路により、アプリケーションが要求するネットワーク機能が、最終的にプラットフォームとドライバ層へ落ちていく構造として可視化されている。

加えて、本図の重要な点は、依存関係が「最終的に含まれたライブラリ一覧」だけではなく、「なぜそのライブラリが含まれたのか」を追跡できることである。例えば、あるノードが `select` により暗黙的に有効化されている場合、ユーザが直接そのオプションを選択していなくても、別の選択を起点として有効化されたことがエッジとして残る。これにより、構成変更を行う際に、影響範囲がどこまで波及するかを設計段階で検討できる。

以上より、図 6.1 は、nginx Unikernel が POSIX 互換 API と TCP/IP スタック、さらに仮想 NIC ドライバとプラットフォーム層へ連鎖的に依存して動作していることを示している。この結果は、第 5 章で述べる「ネットワーク関連構成を検出し、ファイアウォール制御へ接続する」という設計方針に対して、構成情報の観点から根拠を与えるものである。

6.3.3 必要ポートの判別

構成依存関係解析およびアプリケーション設定の解析により、nginx Unikernel が外部との通信に TCP ポート（HTTP 通信用）を必要とすることを特定した。

6.3.4 ファイアウォール状態の確認

Unikernel 起動前のファイアウォール状態を確認したところ、nginx が利用する通信ポートは開放されていない状態であった。

この結果から、事前にファイアウォール設定を行わない場合、nginx Unikernel の通信は成立しないことが確認された。

```
Host is up (0.0000030s latency).
Not shown: 65534 closed tcp ports (reset)
PORT   STATE SERVICE
22/tcp open  ssh

Nmap done: 1 IP address (1 host up) scanned in 0.68 seconds
```

図 6.2: iptabels 設定前のポート開放状態

6.3.5 ファイアウォール自動設定結果

構成情報およびアプリケーション設定から、nginx が外部との HTTP 通信を必要とすることを判別し、該当ポートのみを許可するファイアウォールルールを自動生成した。

```
[+] detected network libs: ['LIBLWIP', 'LIBPOSIX_SOCKET', 'LIBPOSIX_UNIXSOCKET', 'LIBUKNETDEV', 'LIBVIRTIO_NET']
[+] nginx listen ports (guest): [80]
[+] published ports (host): [8080]
[+] allow ports total: [22, 8080]
[*] backing up iptables
[cmd] command -v iptables-save >/dev/null 2>&1 && echo OK || echo NG
[cmd] iptables-save > /tmp/iptables.unikraft.bak
[*] applying firewall rules
[cmd] command -v iptables >/dev/null 2>&1 && echo OK || echo NG
[cmd] iptables -F
[cmd] iptables -P INPUT DROP
[cmd] iptables -P OUTPUT DROP
[cmd] iptables -P FORWARD DROP
[cmd] iptables -A INPUT -i lo -j ACCEPT
[cmd] iptables -A OUTPUT -o lo -j ACCEPT
[cmd] iptables -A INPUT -i docker0 -j ACCEPT
[cmd] iptables -A OUTPUT -o docker0 -j ACCEPT
[cmd] iptables -A FORWARD -i docker0 -j ACCEPT
[cmd] iptables -A FORWARD -o docker0 -j ACCEPT
[cmd] iptables -A INPUT -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
[cmd] iptables -A OUTPUT -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
[cmd] iptables -A FORWARD -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
[cmd] iptables -A OUTPUT -p udp --dport 53 -j ACCEPT
[cmd] iptables -A OUTPUT -p tcp --dport 53 -j ACCEPT
[cmd] iptables -A OUTPUT -p tcp --dport 443 -j ACCEPT
[cmd] iptables -A INPUT -p tcp --dport 22 -j ACCEPT
[cmd] iptables -A OUTPUT -p tcp --sport 22 -j ACCEPT
[cmd] iptables -A INPUT -p tcp --dport 8080 -j ACCEPT
[cmd] iptables -A OUTPUT -p tcp --sport 8080 -j ACCEPT
[*] running unikraft
[cmd] kraft run -W -p 8080:80
```

図 6.3: iptables 設定結果（nginx）

生成されたルールでは、HTTP 通信に必要な TCP ポートのみが許可され、それ以外の外部通信は遮断される構成となっている。

6.3.6 通信確認結果

ファイアウォール設定後、外部ホストから nmap を用いてポートスキャンを実施した。

```
Host is up (0.0000060s latency).
Not shown: 65532 closed tcp ports (reset), 1 filtered tcp port (no-response)
Some closed ports may be reported as filtered due to --defeat-rst-ratelimit
PORT     STATE SERVICE
22/tcp   open  ssh
8080/tcp open  http-proxy

Nmap done: 1 IP address (1 host up) scanned in 2.00 seconds
```

図 6.4: iptables 設定後のポート開放状態（nginx）

結果として、Web サーバが待ち受けるポートのみが open として検出され、それ以外のポートは遮断されていることが確認できた。また、ブラウザによるアクセスにより、nginx が正常に応答することを確認した。

6.4 ケース 2：redis Unikernel

6.4.1 対象アプリケーション概要

redis はインメモリ型データストアであり、本評価では外部公開を想定しない内部サービスとして扱う。外部からの接続を不要とする構成において、ファイアウォール制御が適切に行えるかを検証する。

6.4.2 構成依存関係の可視化結果

redis Unikernel に対しても，第 4 章で提案した手法を用いて構成依存関係の解析を行った。

.unikraft/build 以下に含まれるライブラリを確認した結果，nginx Unikernel の場合と同一のネットワーク関連ライブラリが含まれていることが分かった。このため，構成依存関係グラフの構造は，図 6.1 に示した nginx Unikernel の可視化結果と同一となった。

これは，Unikraft においてはアプリケーションの種類に関わらず，ネットワーク機能を有効化した時点で，TCP/IP スタックやソケット API などの共通ライブラリが一括して有効化される設計となっているためである。

一方で，アプリケーションレベルでは，nginx は外部公開を前提とした HTTP サーバであるのに対し，redis は外部公開を想定しない内部サービスとして利用される。このように，構成依存関係の観点では両者は同一であっても，要求される通信ポリシーは大きく異なる。

本結果は，構成依存関係の可視化により，アプリケーション種別に依存しない基盤側の構成要素を把握できること，および，後続の Firewall 制御においてはアプリケーションの利用形態を考慮した制御が必要であることを示している。

6.4.3 必要ポートの判別

構成情報の解析により、redis Unikernel はネットワーク機能を利用するものの、外部からの接続を必要としない構成であることを確認した。

6.4.4　ファイアウォール状態の確認

Unikernel 起動前のファイアウォール状態を確認したところ、redis に関連する通信ポートは開放されていない状態であった。図 6.2 に，iptables 設定前におけるポート開放状態を示す。

6.4.5　ファイアウォール自動設定結果

redis は外部からの接続を必要としないため、外部通信を遮断するファイアウォールルールを自動生成した。

```
[+] detected network libs: ['LIBLWIP', 'LIBPOSIX_SOCKET', 'LIBPOSIX_UNIXSOCKET', 'LIBUKNETDEV', 'LIBVIRTIO_NET']
[+] allow ports total: [22, 6379]
[*] backing up iptables
[cmd] command -v iptables-save >/dev/null 2>&1 && echo OK || echo NG
[cmd] iptables-save > /tmp/iptables.unikraft.bak
[*] applying firewall rules
[cmd] command -v iptables >/dev/null 2>&1 && echo OK || echo NG
[cmd] iptables -F
[cmd] iptables -P INPUT DROP
[cmd] iptables -P OUTPUT DROP
[cmd] iptables -P FORWARD DROP
[cmd] iptables -A INPUT -i lo -j ACCEPT
[cmd] iptables -A OUTPUT -o lo -j ACCEPT
[cmd] iptables -A INPUT -i docker0 -j ACCEPT
[cmd] iptables -A OUTPUT -o docker0 -j ACCEPT
[cmd] iptables -A FORWARD -i docker0 -j ACCEPT
[cmd] iptables -A FORWARD -o docker0 -j ACCEPT
[cmd] iptables -A INPUT -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
[cmd] iptables -A OUTPUT -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
[cmd] iptables -A FORWARD -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
[cmd] iptables -A OUTPUT -p udp --dport 53 -j ACCEPT
[cmd] iptables -A OUTPUT -p tcp --dport 53 -j ACCEPT
[cmd] iptables -A OUTPUT -p tcp --dport 443 -j ACCEPT
[cmd] iptables -A INPUT -p tcp --dport 22 -j ACCEPT
[cmd] iptables -A OUTPUT -p tcp --sport 22 -j ACCEPT
[cmd] iptables -A INPUT -p tcp --dport 6379 -j ACCEPT
[cmd] iptables -A OUTPUT -p tcp --sport 6379 -j ACCEPT
[*] running redis unikernel
[cmd] kraft run --rm -M 256M -p 6379:6379
```

図 6.5: iptables 設定結果（redis）

この設定により、外部からの TCP 接続はすべて遮断される一方で、必要な内部通信は許可される構成となっている。

6.4.6　通信確認結果

外部ホストからのポートスキャンを実施した結果、redis が使用するポートは open として検出されなかった。

```
Host is up (0.0000060s latency).
Not shown: 65533 closed tcp ports (reset)
PORT     STATE SERVICE
22/tcp   open  ssh
6379/tcp open  redis

Nmap done: 1 IP address (1 host up) scanned in 0.84 seconds
```

図 6.6: iptables 設定後のポート開放状態（redis）

一方で、内部からのアクセスにより redis が正常に動作することを確認した。

6.5 評価結果の考察

6.5.1 構成依存関係可視化の評価

まず、Unikernel の構成依存関係可視化について評価を行った。

.unikraft/build 以下の生成物を確認することで、最終的に含まれるライブラリを把握することは可能である。しかし、この方法では、なぜそのライブラリが含まれたのか、どの構成オプションが起点となったのか、といった 構成生成の因果関係を把握することは困難である。

第 4 章の手法では Kconfig に記述された select や depends on といった依存関係を解析し、ライブラリ間の関係をグラフとして可視化した。その結果、lwIP を含むネットワーク関連ライブラリがどの構成条件によって有効化されたのかを明確に確認できることを確認した。このことから、第 4 章の手法によりネットワーク機能が構成上「入り得るかどうか」を設計段階で判断可能であることが示されたと考えられる。

6.5.2 必要ポート確認と自動設定の有効性

nginx および redis の両ケースにおいて、アプリケーションが必要とする通信ポートを正しく判別し、ファイアウォール上での開放状況を確認した上で、自動設定を行えることを確認した。これにより、提案方式が「必要なポートが閉じている状態からでも、自動的に通信可能な状態へ遷移できる」ことを示したと考えらる。

6.5.3 不要な通信の抑制

不要な通信ポートを遮断することで、外部からの攻撃対象となり得る範囲を限定できた。これにより，Unikernel の最小構成という設計思想を、通信制御の面からも補強できたと考えられる。

6.6　本章のまとめ

6.6.1　評価結果のまとめ

本章では、構成情報に基づくファイアウォール自動設定方式について、必要な通信ポートがファイアウォール上で開放されているかを確認し、開放されていない場合に自動設定できるかという観点から評価を行った。

その結果、提案方式により、ファイアウォール設定を事前に行わない場合であっても、Unikernel実行に必要な通信を確保できることを確認した。

6.6.2　次章へのつながり

次章では、本研究全体の成果をまとめるとともに、今後の課題について述べる。

第7章　結論

7.1　本研究の成果

7.1.1　研究のまとめ

本研究では、本研究では、Unikraft を用いて構築される Unikernel を対象として、ライブラリ構成および依存関係の可視化と、構成情報に基づくファイアウォール自動設定を組み合わせた手法を提案した。

Unikernel は最小構成を志向する実行基盤である一方、Kconfig による構成管理の自動化により、利用者が意図していない機能や通信経路が含まれる可能性がある。本研究はこの課題に対し、構成情報を明示的に扱い、実行前に通信要件を確認・制御する仕組みを導入することで対応した。

7.1.2　構成依存関係の可視化

第 4 章では、Unikraft における Kconfig を解析し、構成オプション間の依存関係を自動的に抽出・可視化する手法を示した。これにより、どの構成要素がどの理由で有効化されているのかを追跡可能とし、従来は把握が困難であった Unikernel 実行イメージの構成を利用者が理解しやすい形で提示できることを示した。

この可視化は、構成設計の妥当性確認や、後述する通信制御の根拠情報として有効である。

7.1.3　ファイアウォール自動設定の実現

第 5 章では、可視化によって得られた構成情報を用いて、アプリケーションが必要とする通信ポートを特定し、ファイアウォール上での開放状況を確認した上で、必要に応じて自動的にファイアウォール設定を行う方式を提案・実装した。

この方式により、ファイアウォール設定を事前に行っていない環境においても、Unikernel の実行に必要な通信を確保しつつ、不要な通信を遮断する制御が可能となった。

7.1.4　評価結果のまとめ

第 6 章では、nginx および redis を対象として評価を行い、以下の点を確認した。

1. アプリケーションが必要とする通信ポートを正しく特定できること
2. ファイアウォール上で必要なポートが開放されているかを自動的に確認できること

3. 必要なポートが閉じている場合に、自動的にファイアウォール設定を行い、通信を成立させられること

4. 自動設定後において、不要な通信ポートが開放されていないこと

これらの結果から、提案手法が 「ファイアウォール設定を前提としない Unikernel 実行」を可能とし、構成情報に基づいた安全な通信制御を実現できることを示した。

7.2 研究の意義

7.2.1 設計としての安全性の提示

本研究の意義は、Unikernel を「結果として安全」な実行基盤として扱うのではなく、構成情報と通信制御の対応関係を明示することで、設計段階から安全性を説明可能な実行基盤として扱える点にある。

構成依存関係の可視化とファイアウォール自動設定を組み合わせることで、Unikernel に含まれる機能と許可される通信の関係を明確に示すことができ、安全性に関する説明責任を果たしやすくなる。

7.2.2 運用負荷の軽減

また、本研究で提案した方式は、ファイアウォール設定を自動化することで、利用者による手動設定や設定ミスを減らし、Unikernel の運用負荷を軽減する効果も期待できる。

7.3 今後の課題

7.3.1 動的な通信要件への対応

本研究では、Unikernel 起動時点での構成情報および設定ファイルを基に、通信要件を判別している。今後は、実行時に通信要件が変化するアプリケーションへの対応や、動的な通信制御への拡張が課題として挙げられる。

7.3.2 他アプリケーションおよび環境への適用

評価では nginx および redis を対象としたが、今後は他のアプリケーションや、異なる実行環境への適用を通じて、提案手法の汎用性を検証する必要がある。

7.3.3 ファイアウォール制御方式の拡張

本研究では iptables を用いたファイアウォール制御を行ったが、他の通信制御機構との連携や、より高水準なポリシー記述方式への拡張も今後の課題である。

7.4　全体のまとめ

本研究では、Unikraft を用いた Unikernel を対象として、構成依存関係の可視化と、ファイアウォール上で必要なポートの確認および自動設定を組み合わせた手法を提案した。

提案手法により、Unikernel 実行前に通信要件を確認し、必要な通信のみを安全に許可する運用が可能となった。本研究の成果は、Unikernel をより安全かつ実用的に運用するための基盤技術として、今後の展開が期待される。

参 考 文 献

[1] Wikipedia, “ Unikernel, ” Wikipedia,
https://en.wikipedia.org/wiki/Unikernel/, 2026.

[2] Simon Kuenzer *et al.*,
“ Unikraft: Fast, Specialized Unikernels the Easy Way ”,arXiv, Apr. 26, 2021.

[3] Unikraft Project, Unikraft: A Fast, Secure and Open-Source Unikernel Development Kit,
http://unikraft.org/, 2019.

[4] Simon Kuenzer,
“ Building a Linux-compatible Unikernel, ” FOSDEM 2023, DOI:10.5446/61677.

[5] Maxime Letemple *et al.*,
“ uTNT: Unikernels for Efficient and Flexible Internet Probing, ” arXiv, May 07, 2024.

[6] Shahidullah Kaiser *et al.*,
“ Exploring the Viability of Unikernels for ARM-powered Edge Computing, ” arXiv, Dec. 04, 2024.

[7] Matthew Leon,
“ The Dark Side of Unikernels for Machine Learning, ” arXiv, Apr. 27, 2020.

[8] Qiita, Unikernel な情報,
https://qiita.com/t-imada/items/ed6a76f5b257f5608ad0, 2021.

[9] Zenn, Unikraft を試してみる,
https://zenn.dev/zenogawa/articles/try_unikraft/, 2025.

[10] Linux Kernel Documentation, Kconfig Language,
https://docs.kernel.org/kbuild/kconfig-language.html, 2023.

[11] Qiita, iptables の基本的な使い方,
https://qiita.com/dtanimoto00/items/1d0a9b02867add646ea5/, 2020.

[12] Nmap Project, Nmap マニュアル（日本語),
https://nmap.org/man/ja/index.html, 2025.

[13] MITRE Corporation, CVE (共通脆弱性識別子) 一覧,
https://www.cve.org/, 2025.

付 録A　実装プログラム

本付録では、本研究で提案した、Unikraft におけるライブラリ依存関係の可視化手法、および構成情報に基づくファイアウォール自動設定方式を実現するために実装したプログラムについて示す。本研究で用いた実装は、特定のアプリケーションに依存しない設計とし、Unikraft が生成する構成情報およびビルド成果物を入力として自動処理を行う点に特徴がある。

A.1　構成依存関係可視化プログラム

A.1.1　概要

本節では、第 4 章で述べた Unikraft のライブラリ依存関係（select、imply、depends on）を自動的に抽出し、可視化するプログラムについて示す。本プログラムは、アプリケーション固有の実装に依存せず、Unikraft が生成する `.unikraft/` および `.unikraft/build/` を入力として利用する。`.unikraft/build` に含まれる `.o` ファイルを基に最終的にビルドへ含まれたライブラリ集合を抽出し、対応する `Config.uk` を解析することで、依存関係を有向グラフとして可視化する。

A.1.2　実装方針

本プログラムは、以下の手順により構成される。

1. `.unikraft/` 以下を探索し、すべての `Config.uk` を収集する。
2. 各 `Config.uk` から `config SYMBOL` を抽出し、シンボル名とファイルの対応関係を構築する。
3. `.unikraft/build/` 以下の `.o` ファイルを解析し、ビルドに含まれるライブラリ集合を抽出する。
4. 各 `Config.uk` を解析し、`select`、`imply`、`depends on` を抽出する。
5. Graphviz を用いて依存関係を有向グラフとして可視化する。

A.1.3　Config.uk の収集とシンボル逆引き

Unikraft では、各ライブラリおよびプラットフォーム単位で `Config.uk` が配置されている。本実装では `.unikraft/` 以下を再帰的に探索し、すべての `Config.uk` を収集するとともに、`config SYMBOL` に記述された構成シンボルと対応するファイルとの対応関係を構築する。これにより、ビルド成果物に含まれるライブラリ名から対応する `Config.uk` を逆引き可能とした。

Listing A.1: Config.uk の収集とシンボル逆引き

```
def build_config_map():
    config_map_short = {}
    config_map_full = {}
    symbol_map = {}

    for root, _dirs, files in os.walk(BASE):
        if "Config.uk" not in files:
            continue

        full = os.path.join(root, "Config.uk")
        dirname = os.path.basename(root)
        config_map_short[dirname] = full

        rel = os.path.relpath(root, BASE)
        rel_norm = rel.replace("/", "_")
        config_map_full[rel_norm] = full

        try:
            with open(full, encoding="utf-8", errors="ignore") as f:
                for line in f:
                    m = re.match(r"config\s+([A-Z0-9_]+)", line.strip())
                    if m:
                        symbol_map[m.group(1)] = full
        except Exception:
            pass

    return config_map_short, config_map_full, symbol_map
```

A.1.4 ビルド生成物からのライブラリ抽出

Unikraft では、ビルド後に `.unikraft/build/` 以下へ各ライブラリに対応するオブジェクトファイルが生成される。本実装では `.o` ファイルを走査し、`.ld.o` を除外することで、最終的にビルドへ含まれたライブラリ集合を抽出する。この手法により、構成結果として実際に使用されたライブラリのみを可視化対象とすることが可能となる。

Listing A.2: ビルドに含まれるライブラリ集合の抽出

```
def get_build_objects_and_libs():
    objs = []
    libs = set()

    for root, _dirs, files in os.walk(BUILD_DIR):
        for f in files:
            if f.endswith(".o") and not f.endswith(".ld.o"):
                objs.append(f)
                libs.add(f.replace(".o", "").upper())

    return objs, libs
```

A.1.5　依存関係の抽出

Config.uk に記述された select、imply、depends on を解析し、ライブラリ間の依存関係を抽出する。特に depends on は論理式として記述される場合があるため、本実装では式中に含まれる構成シンボルを抽出し、依存関係として扱う。これにより、構成生成の因果関係を網羅的に把握できる。

Listing A.3: select、imply、depends on の抽出

```
def parse_config_uk(path, cache):
    if path in cache:
        return cache[path]

    selects, implies, depends = [], [], []

    try:
        with open(path, encoding="utf-8", errors="ignore") as f:
            for line in f:
                s = line.strip()

                m = re.match(r"select\s+([A-Z0-9_]+)", s)
                if m:
                    selects.append(m.group(1))

                m = re.match(r"imply\s+([A-Z0-9_]+)", s)
                if m:
                    implies.append(m.group(1))

                m = re.match(r"depends␣on\s+(.+)", s)
                if m:
                    expr = m.group(1)
                    syms = re.findall(r"\b[A-Z0-9_]+\b", expr)
                    depends.extend(syms)
    except Exception:
        pass

    cache[path] = (selects, implies, depends)
    return selects, implies, depends
```

A.1.6　可視化に関する補足

依存関係の種類はエッジの色によって区別し、select、imply、depends on を視覚的に識別可能とした。また、ノードは app、external、posix、platform、driver、uk-core といった層構造に基づいて色分けを行い、Unikraft の内部構成を直感的に理解できるよう工夫した。

A.2　ファイアウォール自動設定プログラム

A.2.1　概要

本節では、第 5 章で述べた、ファイアウォール上で必要なポートが開放されているかを確認し、開放されていない場合に自動的に設定するプログラムについて示す。iptables はホスト OS 上で動作し、Unikernel は QEMU 上で実行される。そのため、本研究ではホスト側の公開ポートをファイアウォール制御の対象とした。

A.2.2　設計方針

本プログラムは、既存サービスの通信を遮断しないこと、Unikernel 実行に必要な最小限の通信のみを許可すること、実験終了後に元のファイアウォール状態へ復元できることを満たすよう設計した。このため、iptables の適用前に現在の設定を保存し、実行後に必ず復元する仕組みを導入した。

A.2.3　外部から到達可能な open ポートの取得

既存のサービスが利用しているポートを保護するため、iptables 適用前に nmap を用いて外部から到達可能な open TCP ポートを取得し、これらは遮断しないよう許可対象に含める。これにより、SSH などの既存通信を維持したまま、追加のファイアウォール制御を行うことが可能となる。

A.2.4　iptables のバックアップと復元

iptables の変更は既存のシステム設定へ影響する可能性があるため、適用前に `iptables-save` により設定を保存し、終了時に `iptables-restore` により元の状態へ復元する。

Listing A.4: iptables の保存と復元

```
def iptables_backup():
    run(f"iptables-save␣>␣{IPTABLES_BACKUP}")

def iptables_restore():
    run(f"iptables-restore␣<␣{IPTABLES_BACKUP}")
```

A.2.5　最小許可ポリシーの適用

本研究では、デフォルトポリシーを DROP とし、必要な通信のみを明示的に許可する最小許可方式を採用する。許可対象は、既存の外部公開ポート、および Unikernel 実行に必要なホスト側公開ポートとする。

Listing A.5: iptables 最小許可ポリシー

```
def iptables_apply(allow_ports):
    run("iptables␣-F")
```

```
    run("iptables␣-P␣INPUT␣DROP")
    run("iptables␣-P␣OUTPUT␣DROP")
    run("iptables␣-P␣FORWARD␣DROP")

    run("iptables␣-A␣INPUT␣-i␣lo␣-j␣ACCEPT")
    run("iptables␣-A␣OUTPUT␣-o␣lo␣-j␣ACCEPT")

    run("iptables␣-A␣INPUT␣-m␣conntrack␣--ctstate␣ESTABLISHED,RELATED␣-j
        ␣ACCEPT")
    run("iptables␣-A␣OUTPUT␣-m␣conntrack␣--ctstate␣ESTABLISHED,RELATED␣-
        j␣ACCEPT")
    run("iptables␣-A␣FORWARD␣-m␣conntrack␣--ctstate␣ESTABLISHED,RELATED␣
        -j␣ACCEPT")

    for p in sorted(allow_ports):
        run(f"iptables␣-A␣INPUT␣-p␣tcp␣--dport␣{p}␣-j␣ACCEPT")
        run(f"iptables␣-A␣OUTPUT␣-p␣tcp␣--sport␣{p}␣-j␣ACCEPT")
```

A.2.6 nginx に対する適用

nginx Unikernel では、`kraft run -p HOST:GUEST` によりホスト側ポートが外部公開される。本研究では、この HOST 側ポートをファイアウォール制御の対象とし、必要な通信のみを許可するルールを自動生成した。

Listing A.6: nginx の公開ポート抽出

```
def parse_published_ports_from_kraft(cmd):
    ports = set()
    for m in re.finditer(r"-p\s+(\d+)\s*:\s*(\d+)", cmd):
        ports.add(int(m.group(1)))
    return ports
```

A.2.7 redis に対する適用

redis Unikernel では、`redis.conf` の `port` ディレクティブから待受ポートを抽出し、`kraft run -p PORT:PORT` を動的に生成する。これにより、アプリケーション設定に基づいた一貫性のあるファイアウォール制御を実現した。

Listing A.7: redis.conf からのポート抽出

```
def parse_redis_port(conf):
    DEFAULT_PORT = 6379
    if not conf.exists():
        return DEFAULT_PORT

    for line in conf.read_text(encoding="utf-8", errors="ignore").
        splitlines():
        s = line.strip()
```

```
        if not s or s.startswith("#"):
            continue
        m = re.match(r"port\s+(\d+)\b", s)
        if m:
            port = int(m.group(1))
            if port == 0:
                raise RuntimeError("redis.conf␣の␣port␣が␣0␣であるため
                    ␣TCP␣が無効である")
            return port
    return DEFAULT_PORT
```

A.2.8 評価との対応

本付録で示したファイアウォール自動設定プログラムは、第 6 章の評価目的である「ファイアウォール上で必要なポートが開放されているかを確認し、開放されていなければ自動設定する」という要件を満たすよう設計されている。これにより、Unikernel 実行前に通信要件を満たす設定へ自動的に遷移し、実験終了後に元の状態へ復元することで、安全性と再現性を両立できることを示した。