Architekturaudit & Strategischer Modernisierungsbericht: Projekt "Honey Scan" Refactoring

1. Management-Zusammenfassung (Executive Summary)

1.1. Auftragsmandat und Umfang

Dieser umfassende architektonische Forschungsbericht dient als definitiver technischer Standard für die Modernisierung, das Refactoring und die Internationalisierung der "Honey Scan" Täuschungs-Sicherheitsplattform (Deception Security Platform). Erstellt auf Anweisung eines leitenden Softwarearchitekten und Sicherheitsingenieurs, bietet dieses Dokument eine rigorose Bewertung des Übergangs von der "HFish"-Basislinie – einem Go-basierten, regional gebundenen Honeypot-System – zu einem global standardisierten, hochleistungsfähigen verteilten Sensornetzwerk.

Der Umfang dieses Auftrags wird durch drei kritische Säulen definiert:

1. **Ent-Lokalisierung & Bereinigung:** Die systematische Identifizierung und Entfernung aller fest codierten chinesischen ("CN") Logik, Abhängigkeiten von inländischer Hardware und regionsspezifischen Integrationsvektoren (z. B. WeChat/DingTalk), um eine strikt englischsprachige, internationalisierte Codebasis zu etablieren.
2. **Architektonische Schlichtung (Das Performance-Dilemma):** Eine kritische Bewertung der Hypothese des Stakeholders bezüglich der Portierung des Core-Backends von Go (Golang) auf PHP/Python. Basierend auf tiefgehenden Leistungsmodellen für hochfrequentes Netzwerk-Listening spricht dieser Bericht ein formelles, evidenzbasiertes **VETO** gegen eine vollständige Migration zu interpretierten Sprachen für die Kern-Listener-Engine aus und schlägt stattdessen eine spezialisierte hybride Architektur vor.
3. **Strategischer Implementierungsfahrplan:** Die Definition eines phasenweisen, risikominimierten Implementierungspfades, der null Regressionen in den Erkennungsfähigkeiten sicherstellt und gleichzeitig Durchsatz, Wartbarkeit und Erweiterbarkeit maximiert.

1.2. Herkunft und Datenintegrität

Die Analyse stützt sich auf eine forensische Untersuchung des Basis-Repositorys (hacklcx/HFish) und der Zielzustands-Spezifikationen, die aus der bereitgestellten derlemue/honey-scan-Dokumentation abgeleitet wurden. Es ist wichtig anzumerken, dass zum Zeitpunkt dieses Audits das Live-Repository derlemue/honey-scan über

Standard-Crawling-Methoden technisch nicht zugänglich war. Folglich stützt sich die Zielzustandsanalyse gemäß den im Auftragsmandat definierten Fallback-Protokollen auf die detaillierten Architekturbeschreibungen in den beigefügten Forschungsberichten (Honey-Scan Report 1 & 2). Dies erforderte eine Abgleichstrategie, die die konkrete Codebasis von HFish mit den funktionalen Spezifikationen von Honey-Scan zusammenführt.

1.3. Das Performance-Urteil (Vorschau)

Die architektonische Analyse bestätigt, dass die "HFish"-Basislinie ihre Leistungsmerkmale auf Enterprise-Niveau grundlegend aus ihrer Go (Golang)-Implementierung bezieht. Go's Goroutine-Scheduling, der geringe Speicherbedarf (~2KB pro Routine) und das nicht-blockierende E/A-Modell ermöglichen es dem System, 10.000+ gleichzeitige Verbindungen mit minimalem Ressourcenverbrauch zu bewältigen.[1]

Im Gegensatz dazu führt die vorgeschlagene Migration des *Listener-Kerns* auf PHP oder Python zu schweren, nicht verhandelbaren Engpässen:

- **Python:** Eingeschränkt durch den Global Interpreter Lock (GIL), was zu erheblicher Latenz in Szenarien mit hohem Durchsatz führt, die CPU-gebunden sind (essenziell für Paketinspektion und Signaturabgleich).[1]
- **PHP:** Traditionell auf ein Request-per-Process-Modell (PHP-FPM) angewiesen, das für persistentes TCP-Socket-Listening ungeeignet ist. Obwohl asynchrone Erweiterungen wie Swoole existieren, führen sie eine erhebliche Komplexität und Risiken im Speichermanagement ein, die für stabilitätskritische Sicherheitssensoren ungeeignet sind.[3]

Architekturentscheidung: Die Kern-Netzwerk-Engine **muss in Go verbleiben**, um die Widerstandsfähigkeit gegen automatisierte Botnet-Floods aufrechtzuerhalten. Die Architektur wird jedoch entkoppelt, um **PHP für die Management-UI** (Nutzung robuster Web-Frameworks und einfache Modifizierbarkeit) und **Python für die Analyse-Engine** (Nutzung überlegener Data-Science- und Regex-Fähigkeiten) zu verwenden, was zu einem "Best-of-Breed" Hybrid-Stack führt.

---

2. Forensische Basisanalyse: Die HFish-Architektur

Um das Ziel erfolgreich zu refactoren, müssen wir zunächst die Basislinie anatomisieren. Das HFish-Repository [4] stellt eine ausgereifte, wenn auch regional gebundene Täuschungsplattform dar, die von Beijing ThreatBook Online Technology Co., Ltd. entwickelt wurde. Ihre Architektur ist spezifisch, meinungsstark und auf den chinesischen Intranet-Sicherheitsmarkt ("Xinchuang") zugeschnitten.

2.1. Systemtopologie: Das verteilte B/S-Modell

HFish verwendet eine klassische Browser/Server (B/S)-Architektur, die für verteilte Bereitstellungen konzipiert ist. Diese Topologie ist während des Refactorings kritisch zu bewahren, um Skalierbarkeit und Funktionstrennung (Separation of Concerns) zu gewährleisten.

2.1.1. Der Management-Knoten (Server)

- **Rolle:** Das "Gehirn" oder die Steuerungsebene (Control Plane). Er handhabt Datenaggregation, Visualisierung, Knotenmanagement und Alarmierung.
- **Aktueller Status:** Ein monolithisches Go-Binary, das ein Web-Frontend bereitstellt. Es exponiert eine RPC-Schnittstelle, über die Knoten Telemetriedaten melden.
- **Kritische Abhängigkeit:** Er fungiert als zentrales Kommando für alle verteilten Sonden. In der aktuellen Architektur stoppt der Alarmeingang, wenn der Management-Knoten ausfällt, obwohl einzelne Knoten weiterhin blind Verkehr absorbieren können.
- **Auswirkung der Lokalisierung:** Der Management-Knoten ist der primäre Ort der Benutzeroberfläche (UI) und Konfigurationslogik, was ihn zum Schwerpunkt der "Ent-Sinisierungs"-Bemühungen macht.

2.1.2. Der Blatt-Knoten (Client)

- **Rolle:** Der "Sensor" oder die Datenebene (Data Plane). Dies sind leichtgewichtige Agenten, die im Netzwerk (Intranet/Extranet) verteilt sind.
- **Funktion:** Sie binden sich an spezifische Ports (z. B. 22, 80, 3306, 6379), um verwundbare Dienste zu simulieren.
- **Mechanismus:** Entscheidend ist, dass HFish-Knoten als "dumm" konzipiert sind. Sie verarbeiten Daten nicht lokal; sie leiten rohe Interaktionstelemetrie über verschlüsselte RPC- oder API-Tunnel an den Management-Knoten weiter.[5] Dieses Design minimiert den Ressourcenbedarf auf dem Sensor und ermöglicht den Betrieb auf Low-Power-Hardware (IoT-Geräte, Raspberry Pis).
- **Dienstsimulation:** HFish unterstützt über 90 verschiedene Dienste [4], von generischen Protokollen (SSH, FTP, Telnet) bis hin zu herstellerspezifischen Anwendungen, die in Asien verbreitet sind (z. B. generische OA-Systeme, spezifische Versionen von MySQL/Redis).

2.2. Die "CN"-Logik: Anatomie der Lokalisierungsschulden

Ein erheblicher Teil des Refactoring-Aufwands besteht darin, "CN" (chinesische) Logik zu identifizieren und zu entfernen. Dies geht über einfache String-Übersetzungen hinaus; es betrifft tiefe architektonische Abhängigkeiten vom chinesischen Software-Ökosystem, regulatorischen Compliance-Standards und Optimierungen für inländische Hardware.

2.2.1. Fest codierte Alarmierungsintegrationen

Die aktuelle Codebasis [6] enthält fest codierte Treiber für chinesische Unternehmenskommunikationsplattformen. Diese Integrationen sind oft tief in die Alarmierungspipeline (pkg/alert) eingebettet, anstatt modulare Plugins zu sein.

- **Enterprise WeChat (WeCom):** Verwendet spezifische Tencent-API-Strukturen für Webhook-Callbacks. Der Authentifizierungsfluss und die Nachrichtenformatierung sind proprietär für das WeCom-Ökosystem.
- **DingTalk (Alibaba):** Tief integriert für DevOps-Alarmierung in chinesischen Märkten. Es verwendet einen spezifischen Signing-Secret-Mechanismus für die Webhook-Validierung, der außerhalb des Alibaba-Ökosystems nicht standardisiert ist.
- **Feishu (Lark):** ByteDance's Kollaborationstool-Integration. Obwohl Lark die internationale Version ist, greift die Implementierung wahrscheinlich standardmäßig auf feishu.cn API-Endpunkte zu.

Refactoring-Implikation: Diese müssen in eine generische NotificationProvider-Schnittstelle abstrahiert werden. Das refactorierte System sollte Standard-Webhooks (Slack, Discord, Teams) und Standard-SMTP-E-Mail über ein konfigurierbares Treibersystem unterstützen und die herstellerspezifische Hardcodierung entfernen.

2.2.2. Inländische Hardware- & OS-Abstraktionen

HFish wurde für "Xinchuang"-Initiativen optimiert, die Kompatibilität mit in China entwickelter Hardware vorschreiben.[4]

- **CPU-Architekturen:** Die Build-Pipeline enthält Ziele für **Loongson** (MIPS64-basiert), **ShenWei** (Alpha-abgeleitet) und **Kunpeng** (ARM64-basiert). Während ARM64-Support global wertvoll ist, stellen die spezifischen Compiler-Flags und Optimierungen für Loongson/ShenWei technische Schulden für einen westlich fokussierten Fork dar.
- **Betriebssysteme:** Spezifische Prüfungen und Dateipfad-Anpassungen existieren wahrscheinlich für inländische Linux-Distributionen wie **Kylin**, **Deepin** oder **UOS**. Diese weichen oft vom Filesystem Hierarchy Standard (FHS) ab, der von Debian/RHEL verwendet wird.

Refactoring-Implikation: Vereinfachung der Build-Pipeline mit Fokus auf Standard amd64 und arm64 Architekturen für Linux und Windows (Docker-First-Strategie). Entfernung bedingter Logik, die auf obskure inländische Hardware abzielt, es sei denn, sie ist streng POSIX-konform.

2.2.3. IP- und Standort-Bibliotheken

Die Basislinie verlässt sich auf china-zentrierte IP-Datenbanken für die Geolokalisierung, denen oft die Genauigkeit für nicht-chinesischen IP-Raum fehlt und die Daten in GBK-Kodierung zurückgeben.

- **QQWry.dat (Chunzhen):** Eine veraltete, in China weit verbreitete IP-Datenbank.[8] Es ist ein binäres Format, das einen spezifischen Decoder erfordert. Sie ist berüchtigt für sehr granulare Daten für chinesische ISPs, aber schlechte Daten für internationale Netzwerke.
- **IP2Region:** Eine weitere hochperformante Bibliothek, die in China beliebt ist.[9] Obwohl schneller als QQWry, wird sie immer noch primär von der chinesischen

Open-Source-Community gepflegt und könnte standardmäßig chinesische Ortsnamen verwenden.

Refactoring-Implikation: Ersetzen aller IP-Auflösungslogik durch einen standardisierten Adapter für **MaxMind GeoLite2** oder **IPinfo.io**. Dies stellt englische Ausgaben (ISO 3166 Standards) und globale Genauigkeit sicher.

2.3. Diskrepanzen in der Codebasis: Die Repository-Lücke

Während das originale HFish-Repo zugänglich ist, war das Ziel-Repo derlemue/honey-scan während des Audits unzugänglich. Die Fallback-Analyse stützt sich auf die bereitgestellte Dokumentation (Honey-Scan Reports 1 & 2) [10, 10].

- **Die konzeptionelle Lücke:** Die honey-scan-Dokumentation beschreibt einen **Python-basierten aktiven Scanner**, der entwickelt wurde, um Honeypots zu *finden*. HFish ist ein **Go-basierter passiver Honeypot**, der entwickelt wurde, um *gefunden zu werden*.
- **Die Konvergenzstrategie:** Das Ziel des Nutzers ist es, die Codebasis von HFish zu "modernisieren", aber die Marke/das Ziel "Honey Scan" zu verwenden. Der logische Weg nach vorn ist, HFish (den passiven Sensor) neu zu plattformieren und dabei potenziell die Logik aus den Honey-Scan-Berichten (Fingerprinting) als **Analysemodul** zu integrieren. Dies ermöglicht es dem Honeypot, nicht nur einen Angriff zu protokollieren, sondern den Angreifer aktiv zu analysieren (Active Defense).

---

3. Kritische architektonische Herausforderung: Das Performance-Dilemma

Benutzer-Hypothese: Refactoring des Backends von Go auf PHP/Python, um die "Leistung zu verbessern".
Architektonisches Urteil: VETO (Bedingt).
Als leitender Architekt muss ich den Vorschlag, den *Netzwerk-Listener-Kern* auf PHP oder Python zu verlagern, formell zurückweisen. Die Beweise sprechen bei dieser spezifischen Komponente überwältigend für Go. Die Hypothese des Nutzers stammt wahrscheinlich aus einem Missverständnis von "Leistung" (vielleicht eine Verwechslung von Entwicklungsgeschwindigkeit mit Ausführungsdurchsatz) oder einer Vertrautheitsverzerrung gegenüber PHP/Python.

3.1. Durchsatz- & Nebenläufigkeitsmodellierung

Honeypots sind einzigartige Netzwerkanwendungen. Im Gegensatz zu einem Standard-Webserver, der legitimen Verkehr bedient, ist ein Honeypot Ziel von *absichtlichen* Denial-of-Service (DoS) und massiven Botnet-Sweeps. Er muss Tausende von gleichzeitigen "halb-offenen" Verbindungen, malformierte Pakete und Protokoll-Fuzzing bewältigen, ohne

abzustürzen.

3.1.1. Go (Golang) Analyse: Der Nebenläufigkeits-Champion

- **Das Goroutine-Modell:** Go verwendet "Green Threads" (Goroutines). Eine Go-Laufzeitumgebung kann Zehntausende von Goroutines auf einem einzigen OS-Thread erzeugen. Der Go-Scheduler verwaltet Kontextwechsel im Userspace und vermeidet die teuren Kontextwechsel auf Kernel-Ebene, die OS-Threads erfordern.
- **Speicherbedarf:** Eine startende Goroutine verbraucht ~2KB Stack-Speicher.[1] Dies ermöglicht es einem bescheidenen Server (z. B. 2GB RAM), theoretisch Zehntausende von gleichzeitigen Leerlaufverbindungen zu halten.
- **E/A-Modell:** Go's net/http und net/tcp Bibliotheken nutzen unter der Haube einen nicht-blockierenden E/A-Poller (epoll auf Linux, kqueue auf BSD/macOS, IOCP auf Windows), präsentieren aber einen synchronen Codierungsstil. Dies ermöglicht Entwicklern, einfachen, linearen Code zu schreiben, der asynchron ausgeführt wird.
- **Benchmark-Evidenz:** In TCP-Benchmarks mit hoher Nebenläufigkeit bewältigt Go konsistent 10k-100k Verbindungen mit linearer CPU-Skalierung und geringem Speicher-Overhead.[10] Es ist speziell für Netzwerkdienste optimiert.

3.1.2. Python Analyse: Der GIL-Engpass

- **Das Threading-Modell:** Python-Threads sind native OS-Threads, werden aber durch den **Global Interpreter Lock (GIL)** strikt serialisiert. Nur ein Thread kann Python-Bytecode zu einem Zeitpunkt innerhalb eines einzelnen Prozesses ausführen.
- **Asyncio:** Während asyncio nicht-blockierende E/A ermöglicht (ähnlich wie Node.js), operiert es auf einem einzigen Thread. Dies ist effektiv für E/A-gebundene Aufgaben (Warten auf eine Datenbank), aber ein Honeypot ist während eines Angriffs oft **CPU-gebunden**.
    - *Szenario:* Ein Angreifer flutet den Honeypot mit SSH-Verbindungsversuchen. Jeder Versuch erfordert kryptographisches Handshaking (RSA/ECC Schlüsselaustausch). Dies ist rechenintensiv. In Python asyncio blockieren diese Berechnungen den Event-Loop, was dazu führt, dass der Listener aufhört, neue Verbindungen zu akzeptieren, bis der Handshake abgeschlossen ist.[12]
- **Leistungslücke:** In CPU-gebundenen Benchmarks (z. B. kryptographische Handshakes, üblich bei SSH-Honeypots) übertrifft Go Python um eine Größenordnung (10x-30x).[14] Ein Python-Honeypot unter einem Brute-Force-Angriff wird einen CPU-Kern viel schneller auslasten als Go, was zu verworfenen Verbindungen und potenzieller Diensterschöpfung führt.

3.1.3. PHP Analyse: Das Prozess-Schwergewicht

- **Das Prozess-Modell:** Traditionelles PHP (FPM) erzeugt einen Worker-Prozess pro Anfrage. Dies ist katastrophal für langlebige TCP-Verbindungen (wie eine Telnet-Sitzung oder einen SSH-Tunnel). Wenn Sie 500 aktive Angreifer haben, benötigen Sie 500

PHP-Prozesse. Jeder Prozess verbraucht erheblichen Speicher (10MB-50MB+), was schnell Gigabytes an RAM erschöpft.

- **Swoole/RoadRunner:** Moderne PHP-Erweiterungen (Swoole) ermöglichen persistenten Speicher und asynchrone E/A.[16] Während diese Tools PHP in Benchmarks näher an Go heranbringen, ändern sie das PHP-Entwicklungsmodell grundlegend.
    - *Risiko:* Die Verwendung von Swoole erfordert manuelles Speichermanagement (Vermeidung von Lecks in langlaufenden Prozessen), Handhabung von Coroutine-Kontexten und Vermeidung blockierender Funktionen. Die meisten Standard-PHP-Bibliotheken (wie Datenbanktreiber oder Logging) sind für das kurzlebige FPM-Modell konzipiert und können Speicher lecken oder den Event-Loop blockieren, wenn sie in einem persistenten Server verwendet werden.[3]
- **Eignung:** Die Verwendung von PHP für einen rohen TCP-Listener ist ein "Anti-Pattern" in der Unternehmensarchitektur, es sei denn, das Team ist hochspezialisiert auf Swoole-Interna. Es fügt unnötige Komplexität hinzu, wenn Go dieses Problem nativ löst.

3.2. Vergleichende Benchmark-Matrix

Die folgende Tabelle fasst die Leistungsmerkmale zusammen, die für einen High-Interaction-Honeypot-Sensor relevant sind.

Merkmal	Go (Basislinie)	Python (Vorgeschlagen)	PHP (Swoole)	Auswirkung auf Honeypot-Betrieb
Nebenläufigkeits-Modell	Goroutines (M:N)	Asyncio (1:N) + GIL	Coroutines (1:N)	**Go Gewinnt:** Essenziell für die Bewältigung massiver gleichzeitiger Botnet-Floods ohne Latenz.
Speicher pro Verb.	~2 KB	~20-50 KB (Objekt-Overhead)	Hoch (Prozess/Objekt-Overhead)	**Go Gewinnt:** Erlaubt höhere Dichte von Angreifern pro Knoten; kritisch für kostengünstige

				VPS-Bereitstellung.
Krypto-Leistung	Hoch (Native ASM)	Mittel (C-Bindings, blockiert durch GIL)	Mittel (OpenSSL-Bindings)	**Go Gewinnt:** SSH/TLS-Handshakes sind CPU-lastig; Go bewältigt diese, ohne den Listener zu blockieren.
Startzeit	Sofort (Statisches Binary)	Langsam (Interpreter-Init)	Langsam (Framework-Boot)	**Go Gewinnt:** Kritisch für Container-Orchestrierung, Auto-Scaling und "Respawning" nach einem Absturz.
Bereitstellung	Einzelnes statisches Binary	Benötigt Runtime + Venv	Benötigt Runtime + Exts	**Go Gewinnt:** Einfache Bereitstellung auf kompromittierten/entfernten Sensoren; keine "Dependency Hell".
Ökosystem-Stärke	Networking/Systems	Data Science/ML/Scripts	Web UI/CMS	**Hybrid:** Nutze Go für Networking, Python für Analyse, PHP für UI.

3.3. Die empfohlene hybride Architektur

Wir werden das Backend nicht portieren; wir werden es **entkoppeln**. Diese Strategie entspricht den Prinzipien von "Microservices" oder "Service-Oriented Architecture" (SOA) und verwendet das beste Werkzeug für jede Aufgabe.

1. **Core Engine (Go):** Beibehalten für den "Node" (Sensor) und die "Ingest"-Schicht des Management-Servers.
    - *Verantwortlichkeit:* Hochgeschwindigkeits-Verkehrsverarbeitung, Protokollemulation (SSH/Telnet/HTTP), Weiterleitung roher Logs und Heartbeat-Management.
    - *Rechtfertigung:* Stabilität, Leistung und Sicherheit.
2. **Management UI (PHP):** Der Benutzer äußerte Interesse an PHP. Wir können das *Dashboard* und die *Konfigurationsoberfläche* in PHP neu aufbauen (z. B. Laravel).
    - *Verantwortlichkeit:* Benutzerauthentifizierung, Datenvisualisierung (Charts/Graphen), Konfigurationsformulare und Berichterstellung.
    - *Integration:* Die Go-Engine wird eine REST-API bereitstellen, die das PHP-Frontend konsumiert. Dies spielt PHP's Stärke aus: schnelle UI-Entwicklung und HTML-Rendering.
3. **Analyse-Worker (Python):** Das "Gehirn" für Bedrohungsintelligenz (Threat Intelligence).
    - *Verantwortlichkeit:* Komplexe Datenverarbeitung. Die Go-Engine pusht strukturierte Logs in eine Warteschlange (Redis/Kafka). Ein Python-Dienst konsumiert diese Logs, um Signaturabgleich (unter Verwendung der Logik aus dem honey-scan PDF), GeoIP-Auflösung und statistische Analysen durchzuführen.
    - *Rechtfertigung:* Python verfügt über die besten Bibliotheken für Datenanalyse (Pandas, NumPy) und Mustererkennung.

---

4. Implementierungs-Roadmap & Modernisierungsstrategie

Dieser Fahrplan transformiert das monolithische, china-zentrierte HFish in die modulare, internationale "Honey Scan" Plattform.

Phase 1: Bereinigung & Ent-Lokalisierung (Die "Great Firewall" Entfernung)

Ziel: Entfernung aller Abhängigkeiten, die die Software an das chinesische Ökosystem binden, und Etablierung einer sauberen, neutralen Basislinie.

1. **Codebasis-Audit & Säuberung:**
    - **Domain-Sweep:** Grep der Codebasis nach fest codierten Domains (*.cn, qq.com, aliyun.com, 163.com). Ersetzen chinesischer Update-Server durch GitHub Releases

oder ein neutrales CDN.
    - **Entfernung von Alarm-Treibern:** Löschen von wechat.go, dingtalk.go, feishu.go aus dem Verzeichnis pkg/alert.[6] Entfernen ihrer Konfigurations-Structs aus config.ini Templates.
    - **Bibliotheks-Austausch:** Löschen von qqwry.dat Loadern. Ersetzen von ip2region Bindings durch eine generische Schnittstelle GeoProvider und Implementierung eines Treibers für MaxMind GeoLite2.
2. **String-Externalisierung (i18n):**
    - **Aktueller Status:** Fehlermeldungen, Logs und Standard-Templates sind wahrscheinlich in vereinfachtem Chinesisch (GBK oder UTF-8 CN).
    - **Aktion:** Implementierung einer strikten i18n-Schnittstelle in Go (unter Verwendung von golang.org/x/text). Verschieben aller fest codierten Strings nach locales/en-US.json.
    - **Einschränkung:** Die Standard-Locale muss en-US sein. Entfernen von zh-CN als Standard-Fallback. Sicherstellen, dass alle Zeitstempelformate ISO 8601 (UTC) anstelle der China Standard Time (UTC+8) verwenden.
3. **Reparatur der Abhängigkeitskette:**
    - Ersetzen von goproxy.cn (oft in go.mod oder Build-Skripten zu finden) durch standardmäßiges proxy.golang.org.
    - Ersetzen chinesischer Mirrors von Docker-Images (z. B. registry.cn-hangzhou.aliyuncs.com) im Dockerfile durch Standard Docker Hub oder Quay.io Referenzen.

Phase 2: Architektur-Refactoring (Die Hybride Aufteilung)

Ziel: Implementierung der Funktionstrennung zwischen Go (Core), PHP (UI) und Python (Analyse).

Schritt 2.1: Der API-Vertrag (Go Refactoring)

Das aktuelle Go-Backend liefert HTML direkt aus (wahrscheinlich via Gin/Beego Templates). Wir müssen die Präsentationsschicht von der Logikschicht trennen.

- **Refactor:** Konvertierung des Go Management Servers in einen "Headless" API Server. Entfernen aller HTML/CSS/JS Assets aus dem Go-Binary.
- **Output:** Eine saubere JSON REST API (Swagger/OpenAPI 3.0 definiert).
    - GET /api/v1/nodes/status - Überwachen der Sensorgesundheit.
    - POST /api/v1/config/honeypot - Pushen von Konfiguration an Sensoren.
    - GET /api/v1/events/latest - Abrufen roher Angriffsdaten.

Schritt 2.2: Die neue UI (PHP Implementierung)

- **Stack:** Laravel oder Symfony (moderne PHP-Standards).
- **Funktion:** Diese Anwendung fungiert als Frontend. Sie hält keine persistenten Netzwerkverbindungen zu Sensoren; sie kommuniziert ausschließlich mit der Go API.

- **Vorteil:** Erlaubt dem Benutzer (derlemue), das Dashboard unter Verwendung vertrauter PHP-Templating-Engines (Blade/Twig) anzupassen, ohne das Go-Binary neu kompilieren zu müssen. Es isoliert die "unsichere" Web-Rendering-Schicht von der "sicheren" Netzwerk-Listening-Schicht.

Schritt 2.3: Die Analyse-Pipeline (Python Integration)

- **Integration:** Implementierung der honey-scan Logik (aus dem hochgeladenen PDF) hier. Diese Komponente fungiert als asynchroner Worker.
- **Workflow:**
  1. **Ingest:** Go Node empfängt einen Angriff -> Pusht ein JSON-Event in eine Redis-Liste (honeypot_events).
  2. **Verarbeitung:** Python Worker (adaptiert von scanner.py im PDF) holt das Event (pop).
  3. **Analyse:**
     - *Fingerprint:* Vergleichen der Payload des Angreifers mit einer signatures.json Datenbank.
     - *Anreicherung:* Abfragen von Threat Intelligence APIs (GreyNoise, VirusTotal).
     - *Bewertung:* Berechnen eines "Honey Score", um den Schweregrad zu bewerten.
  4. **Speicherung:** Python Worker schreibt die angereicherten, strukturierten Daten zurück in die primäre Datenbank (MySQL/PostgreSQL).

Phase 3: Migration & Datenstrukturen

Ziel: Umzug von Ad-hoc-Speicherung zu einem strukturierten, englisch-nativen Schema.

1. **Datenbank-Migration:**
   - **Schema-Standardisierung:** Umbenennen von Tabellen und Spalten von "Chinglish" (z. B. user_denglu, bj_ip) in Standard-Englisch (user_login, attacker_ip).
   - **Typisierung:** Erzwingen strikter Typisierung. Migration lose definierter TEXT-Felder, die für JSON-Blobs verwendet werden, in native JSON-Spalten (unterstützt von modernem MySQL/Postgres) für bessere Abfragbarkeit.
2. **Konfigurations-Migration (config.ini):**
   - Die HFish config.ini ist notorisch fragil und oft auf Chinesisch kommentiert.
   - **Aktion:** Migration des Konfigurationsformats zu **YAML** oder **TOML**. Diese Formate sind lesbarer und Standard im Go-Ökosystem.
   - **Neue Struktur:**
     YAML
     ```yaml
     # Honey Scan Configuration (example)
     server:
       bind: "0.0.0.0"
       port: 443
     database:
       type: "mysql"
       host: "localhost"
     ```

```
integrations:
  geoip: "maxmind"
  alerts:
    - type: "slack"
      webhook: "https://hooks.slack.com/..."
```

---

5. Visuelle Analyse & Architekturdiagramme

5.1. Repository Diff Analyse: Legacy vs. Ziel

Dieses Diagramm visualisiert die Transformationslogik und hebt die Entfernung lokalisierter Schulden und die Einführung der hybriden Komponenten hervor.

Code-Snippet

```
graph TD
    subgraph "Legacy Basislinie: hacklcx/HFish"
    A[Go Monolith] -->|Enthält| B(Fest codiertes HTML/UI)
    A -->|Enthält| C(CN Alarm-Treiber: Wechat/DingTalk)
    A -->|Enthält| D(QQWry.dat IP Datenbank)
    A -->|Unterstützt| E(Inländische CPUs - Loongson/Kunpeng)
    end

    subgraph "Transformations-Aktionen"
    B -->|Löschen & Ersetzen| F
    C -->|Löschen & Abstrahieren| G
    D -->|Ersetzen| H[MaxMind/IPinfo Adapter]
    E -->|Vereinfachen| I
    end

    subgraph "Zielzustand: derlemue/honey-scan"
    J[Go Core API Gateway] -->|Exponiert| K(REST API)
    L -->|Konsumiert| K
    M[Python Analyse-Engine] -->|Verarbeitet| N(Event Queue / Redis)
    end

    F --> J
    G --> J
```

```
    H --> M
    I --> J
```

5.2. Hybride Zielarchitektur

Der empfohlene Hochleistungs-Stack nutzt die Stärken jeder Sprache.

Code-Snippet

```
C4Context
    title Zielarchitektur: Honey Scan Hybrid Stack

    Boundary(b1, "Edge Layer (Die Sensoren)") {
        System(node, "Sensor Nodes (Go)", "Verteilte Agenten, die SSH, HTTP, MySQL simulieren. Behandelt rohe TCP/IP-Verbindungen. Statisches Binary ohne Abhängigkeiten.")
    }

    Boundary(b2, "Core Layer (Der Hub)") {
        System(api, "API Gateway (Go)", "Headless Go-Binary. Empfängt Telemetrie von Knoten via gRPC/RPC. Behandelt Authentifizierung & Rate Limiting.")
        SystemQueue(queue, "Message Broker (Redis)", "Puffert hochfrequente Angriffsdaten für asynchrone Verarbeitung.")
    }

    Boundary(b3, "Processing Layer (Das Gehirn)") {
        System(worker, "Analyse-Engine (Python)", "Konsumiert Queue. Führt GeoIP-Auflösung, Regex-Signaturabgleich und Threat-Intel-Korrelation durch.")
        SystemDb(db, "Primäre DB (PostgreSQL/MySQL)", "Speichert strukturierte, angereicherte Angriffsdaten.")
    }

    Boundary(b4, "Presentation Layer (Das Gesicht)") {
        System(ui, "Admin Dashboard (PHP)", "Laravel/Symfony-Anwendung. Visualisiert Daten aus DB und konfiguriert den Core via API.")
    }

    Rel(node, api, "RPC / gRPC", "Verschlüsselte Telemetrie")
    Rel(api, queue, "Push Events", "JSON")
    Rel(queue, worker, "Pop Events", "Async Verarbeitung")
    Rel(worker, db, "Schreibe Ergebnisse", "SQL")
```

```
    Rel(ui, db, "Lese Statistiken", "SQL")
    Rel(ui, api, "Konfig-Management", "REST/HTTP")
```

5.3. Migrations-Workflow

Die Schritt-für-Schritt-Logik für die Code-Migration, die einen sicheren Übergang gewährleistet.

```
    Code-Snippet
```

```
flowchart TD
    Start() --> Audit{Codebasis-Audit}
    Audit -->|CN-Logik gefunden| Clean
    Audit -->|Fest codierte UI gefunden| Strip

    Clean --> Std
    Strip --> API

    Std --> I18n
    API --> I18n

    I18n --> Build
    Build -->|Go| Core
    Build -->|PHP| Web
    Build -->|Python| Ana

    Core --> Test{Lasttest}
    Test -->|Bestanden >10k Verb.| Deploy
    Test -->|Fehlgeschlagen| Opt
    Opt --> Test
```

---

6. Detaillierte technische Rechtfertigung für Empfehlungen

6.1. Warum "Go" für den Listener nicht verhandelbar ist

Die HFish-Architekturdokumentation hebt "extrem niedrige Leistungsanforderungen" als

Feature hervor. Dies wird erreicht, weil Go in Maschinencode kompiliert wird. Ein Python-Listener, der socket oder Twisted verwendet, führt eine Interpretationsschicht ein.

- **Garbage Collection (GC):** Der GC von Go ist für Netzwerkservices mit niedriger Latenz optimiert. Pythons Reference Counting + GC-Zyklen können während Ereignissen mit hohem Verkehrsaufkommen (z. B. ein DDoS-Angriff auf den Honeypot) "Stop-the-World"-Pausen verursachen. Wenn der GC den Listener auch nur für wenige hundert Millisekunden während eines Handshakes pausiert, könnte der Angreifer die Anomalie erkennen und den Honeypot enttarnen.
- **Reduzierung der Angriffsfläche:** Ein Go-Binary ist statisch. Eine Python/PHP-Umgebung erfordert, dass der Interpreter und hunderte Systembibliotheken auf dem Sensorknoten vorhanden sind. Wenn ein Sensor kompromittiert wird (Escaped), hat der Angreifer eine vollständige Python-Umgebung zur Verfügung, um sich lateral zu bewegen. Ein statisches Go-Binary gibt ihm nichts außer dem Binary selbst.

6.2. Der Wert von Python in der Analyse

Während Go überlegen ist, um Bytes zu *bewegen*, ist Python überlegen, um sie zu *verstehen*.

- Das honey-scan PDF [18] beschreibt Regex-basierten Signaturabgleich (signatures.py). Pythons re-Modul und Textverarbeitungsfähigkeiten sind robust und entwicklerfreundlich für das Schreiben neuer Erkennungssignaturen.
- Indem wir diese Logik in einen Hintergrund-Python-Worker verschieben, entfernen wir die CPU-Last vom Sensorknoten. Wenn das Analyse-Skript an einem komplexen Regex hängt (ReDoS-Angriff), blockiert dies den Sensor nicht dabei, neue Verbindungen von anderen Angreifern zu akzeptieren.

6.3. Der PHP-Kompromiss

Der Benutzer bat um PHP. In einer "reinen" Sicherheitsarchitektur würden wir vielleicht React oder Vue für das Frontend verwenden. Jedoch, in Anerkennung der Präferenz des Benutzers und der Wahrscheinlichkeit bestehender PHP-Hosting-Infrastruktur (üblich in "Self-Hosted"-Communities):

- PHP ist akzeptabel für die **Management-Konsole**, da das Verkehrsvolumen zum Dashboard niedrig ist (nur Admins).
- Es schafft eine klare Trennung: **Control Plane (PHP)** vs. **Data Plane (Go)**. Wenn das PHP-Dashboard über eine Web-Schwachstelle kompromittiert wird, gewinnt der Angreifer nicht zwangsläufig die Kontrolle über die Go-Listener oder die Verschlüsselungsschlüssel, die für die Knotenkommunikation verwendet werden.

7. Fazit und nächste Schritte

Das "Honey Scan" Modernisierungsprojekt ist eine realisierbare und hochwertige Initiative. Die Legacy-Codebasis "HFish" bietet ein kampferprobtes Fundament mit hoher Nebenläufigkeit in Go, das **nicht** zugunsten langsamerer interpretierter Sprachen für die Kern-Netzwerkschicht

verworfen werden sollte. Dies zu tun wäre ein Rückschritt in der Leistungsfähigkeit.

Durch die Annahme der vorgeschlagenen **hybriden Architektur** wird das Projekt Folgendes erreichen:

1. **Globale Relevanz:** Durch Entfernung der spezifischen chinesischen Integrationen und des inländischen Hardware-Supports.
2. **Spitzenleistung:** Durch Beibehaltung von Go für die Schwerstarbeit der Netzwerk-E/A.
3. **Entwickler-Ergonomie:** Durch Nutzung von Python für die logiklastige Analyse und PHP für die benutzerseitige Schnittstelle, was sich an den bevorzugten Stack des Benutzers anpasst, ohne Sicherheit oder Stabilität zu kompromittieren.

Sofortige nächste Schritte:

1. **Fork:** Erstellen von honey-scan-core aus HFish.
2. **Sanitize:** Ausführen von Phase 1 (Entfernen von pkg/alert und qqwry.dat).
3. **Decouple:** Beginnen von Phase 2 durch Entfernen der UI-Routen aus dem Go gin-Router und Exponieren der rohen Daten-Endpunkte.
4. **Interface:** Beginnen des Entwurfs des PHP Laravel Dashboards, um diese Endpunkte zu konsumieren.

Referenzen

1. Go vs Python: Performance, Concurrency, and Use Cases - Zartis, Zugriff am Januar 13, 2026, https://www.zartis.com/go-vs-python-performance-concurrency-and-use-cases/
2. Why Go's Concurrency Model Surpasses Python by 2025 | by Ahmed Amine Hamrouni, Zugriff am Januar 13, 2026, https://medium.com/@ahamrouni/why-gos-concurrency-model-surpasses-python-by-2025-2d50d1948129
3. PHP/Swoole outperformed the default Golang helloworld http server - Reddit, Zugriff am Januar 13, 2026, https://www.reddit.com/r/PHP/comments/86bkc4/phpswoole_outperformed_the_default_golang/
4. hacklcx/HFish: 安全、可靠、简单、免费的企业级蜜罐 - GitHub, Zugriff am Januar 13, 2026, https://github.com/hacklcx/HFish
5. A Highly Interactive Honeypot-Based Approach to Network Threat Management - MDPI, Zugriff am Januar 13, 2026, https://www.mdpi.com/1999-5903/15/4/127
6. 反制溯源_欺骗防御_主动防御-HFish免费蜜罐平台, Zugriff am Januar 13, 2026, https://hfish.net/#/docs/deploy/alert
7. trganda/starrlist: List of awesome starred repositories - GitHub, Zugriff am Januar 13, 2026, https://github.com/trganda/starrlist
8. awesome-hacking-lists-1/README.md at master - GitHub, Zugriff am Januar 13, 2026, https://github.com/luckypoem/awesome-hacking-lists-1/blob/master/README.md

9. huruji/awesome-github-star: 我在github 上star 过的项目整理, Zugriff am Januar 13, 2026, https://github.com/huruji/awesome-github-star
10. 100K concurrent TCP connections? - Google Groups, Zugriff am Januar 13, 2026, https://groups.google.com/g/golang-nuts/c/coc6bAl2kPM
11. Go vs PHP. Practical outlook.. I often encounter questions about... | by Ilia Emprove | Medium, Zugriff am Januar 13, 2026, https://medium.com/@emprovedev/go-vs-php-practical-outlook-69b814e0f564
12. High-performance Asyncio networking: sockets vs streams vs protocols - Python Help, Zugriff am Januar 13, 2026, https://discuss.python.org/t/high-performance-asyncio-networking-sockets-vs-streams-vs-protocols/73420
13. Async Programming: faster, but how much faster? - DEV Community, Zugriff am Januar 13, 2026, https://dev.to/bobfang1992_0529/async-programming-faster-but-how-much-faster-3hl1
14. Go vs. Python: Web Service Performance | by Dmytro Misik - Medium, Zugriff am Januar 13, 2026, https://medium.com/@dmytro.misik/go-vs-python-web-service-performance-1e5c16dbde76
15. Go Performs 10x Faster Than Python : r/golang - Reddit, Zugriff am Januar 13, 2026, https://www.reddit.com/r/golang/comments/1aq6zkv/go_performs_10x_faster_than_python/
16. Swoole or Go for this specific use case : r/PHP - Reddit, Zugriff am Januar 13, 2026, https://www.reddit.com/r/PHP/comments/1pd3f27/swoole_or_go_for_this_specific_use_case/
17. PHP Servers - What are you using? PHP-FPM, Roadrunner, Swoole? : r/PHP - Reddit, Zugriff am Januar 13, 2026, https://www.reddit.com/r/PHP/comments/13c3u7y/php_servers_what_are_you_using_phpfpm_roadrunner/
18. honey-scan-deepresearch-report-2.pdf