POKISEC：一种多架构、容器化的临时恶意软件引爆沙箱

Alejandro Avina[1], **Yashas Hariprasad**[1],**Naveen Kumar Chaudhary**[2]

[1]**加州州立大学东湾分校计算机科学系，美国**[2]**印度国家法证科学大学，甘地讷格尔，印度**

afernandez71@horizon.csueastbay.edu，yashas.hariprasad@csueastbay.edu
naveen.chaudhary@nfsu.ac.in

arXiv:2512.20860v1 [cs.CR] 24 Dec 2025

2025年12月25日

摘要

动态恶意软件分析需要在高度隔离且可快速重置的环境中执行不可信二进制文件。实际上，许多恶意软件引爆工作流仍依赖笨重的虚拟机监控程序或专用裸机实验室，限制了可移植性和自动化。随着 ARM64 开发者硬件（例如 Apple Silicon）的普及，这一挑战愈发突出：常见的开源沙箱配置方案和预构建环境往往默认使用 x86_64 主机，难以在不同架构间顺畅迁移。

本文介绍 pokiSEC，这是一款轻量级的临时恶意软件引爆沙箱，将完整的虚拟化与访问堆栈封装在 Docker 容器中。pokiSEC 集成了 QEMU 和硬件加速（如可用则使用 KVM），并提供基于浏览器的工作流，支持自带 Windows 磁盘映像。其关键贡献是一个通用入口点，可在运行时检测主机架构，并选择经过验证的虚拟机监控程序配置（机器类型、加速模式和设备配置文件），从而让单一容器映像和代码库能够在 ARM64 和 x86_64 主机上启动 Windows 客户机。我们在 Apple Silicon（ARM64）和 Ubuntu（AMD64）上验证了 pokiSEC，结果表明其交互性能适用于分析人员的工作流，并可通过临时容器生命周期实现一致的拆除语义。实现代码见 https://github.com/PanLuvme/pokiSEC。

关键词 恶意软件分析 ·Docker ·KVM · 虚拟化 · 网络安全

1 引言

恶意软件分析日益依赖动态 观察，即在受控环境中执行可疑样本并记录其行为 [1, 2]。与静态逆向工程不同，基于引爆的工作流程能够揭示运行时行为，例如进程创建、注册表修改、投放的有效载荷、提权尝试以及出站网络活动 [3, 4, 5, 6]。因此，核心要求是实现可靠的隔离。分析环境必须与分析人员的主机及周边的任何生产基础设施充分隔离，以防止意外感染或持久化，同时又要足够易用，以支持反复、迭代地检查 [7, 8]。

历来，从业者一直依靠一类虚拟机监控程序、专用的“物理隔离”实验室，或经过精心整理的虚拟机模板来安全地引爆恶意软件 [9, 10, 11, 12]。这些方法依然有效，但在现代工作流程中会带来不容忽视的实际开销。它们在运行之间重置速度缓慢、消耗资源，而且往往难以自动化，也难以集成到 CI/CD 风格的流水线或可重复的实验环境中。与此同时，分析人员社区已转向使用可复现工具和基础设施即代码开展安全研究，但恶意软件引爆环境仍常常需要专门的主机配置、专用的虚拟化基础设施以及手动进行环境准备 [13, 14, 15]。

第二个较新的压力点是架构碎片化 [16]。随着 ARM64 工作站（尤其是 Apple Silicon）日益普及，安全团队和研究人员越来越常在混合设备群中工作，其中开发笔记本电脑

以及部分持续集成运行器与传统的 x86_64 分析工作站有所不同。许多开源安全工具和预构建的沙箱配置方案都针对 x86 假设进行了优化，而跨架构运行通常并非设计中的首要目标[17, 18]。因此，使用 ARM64 笔记本电脑的分析人员可能无法启动标准的 Windows 沙箱，除非承担高昂的仿真开销，或依赖专有的虚拟化技术栈。这种不匹配影响的不只是性能，也会影响可复现性：“同一个沙箱”在两台不同的机器上可能并不完全相同，从而损害实验的一致性和协作分析。
与此同时，容器化已展现出对可复现部署和打包极具吸引力的优势。Docker 风格的工作流程使交付完整的软件技术栈变得简单直接，能够减少“在我的机器上可以运行”的问题，并简化新贡献者的上手过程。然而，恶意软件引爆通常需要完整的客户机操作系统和硬件辅助虚拟化，而不仅仅是进程级隔离。要满足这些要求，就需要一种将虚拟化本身封装在容器内的架构，同时在可用时通过硬件加速保持接近原生的性能 [19, 20]。

本文介绍了 POKISEC，这是一款轻量、临时的恶意软件引爆沙箱，旨在完全运行于 *Docker* 容器内。该系统将基于 QEMU/KVM 的虚拟化和基于 Web 的导入界面封装到一个单一的可移植镜像中。其主要目标是在保留安全分析所需隔离特性的同时，降低引爆环境的运维阻力、安装复杂度、特定架构配置要求，以及手动执行“重置为干净快照”的循环。在 POKISEC 中，用户可以通过网页加载器上传预配置的 Windows 磁盘映像（QCOW2），然后通过浏览器内桌面会话与已启动的客户机交互。该技术栈经过结构化设计，使主机只需具备 Docker（以及虚拟化支持）；其余沙箱依赖项、编排逻辑和访问层均包含在镜像中，并在其中进行版本管理。

一项核心设计挑战，是在不要求用户维护多个独立构建版本和操作手册的前提下，实现多架构可移植性。pokiSEC 通过通用入口点解决这一问题：该入口点会在运行时检测底层主机架构，并选择相应的 QEMU 二进制文件和加速配置。在 ARM64 主机上，系统会启动面向 ARM 的 QEMU 配置并启用 KVM 加速；而在 x86_64 主机上，则使用标准的 x86_64 虚拟机监控程序配置并启用 KVM。此方法将特定于架构的 QEMU 标志隐藏在统一的执行接口之后，使单一代码库和单一容器映像能够支持异构环境。

除了可移植性之外，引爆沙箱还必须确保每次运行都能“重新开始”。因此，pokiSEC 强调临时性：容器应以非持久化语义运行（例如使用 Docker 的 --rm），确保引爆过程中产生的文件系统和系统状态变更仅存在于临时的写时复制层中，并在容器停止时销毁。由此形成的工作流程降低了不同运行之间交叉污染的风险，也减少了反复恢复到已知干净基线的操作成本；这对于迭代式恶意软件分析和受控实验至关重要。

该系统还优先考虑可用性，以便用户快速上手并进行交互式探索。它无需依赖仅通过命令行界面操作的工作流和手动配置网络，而是引入了一种加载器到虚拟机监控器的交接机制：轻量级网页加载器接收磁盘映像后便交出控制权，由虚拟机监控程序在同一端点接管会话，并通过基于浏览器的 VNC 界面呈现客户机桌面。这一设计让用户体验保持简单（“上传后启动”），同时保留底层 QEMU/KVM 的灵活性。
贡献总结：本文介绍了一种容器化恶意软件引爆环境，将虚拟机监控程序、编排逻辑和可通过浏览器访问的界面打包到单个 Docker 镜像中。本文提出了一个通用入口点，可根据运行时架构检测动态重新配置虚拟化后端，从而在 ARM64 和 x86_64 主机上实现统一的沙箱工作流。此外，本文展示了一种用于引爆的临时执行模型，该模型注重快速拆除，并确保每次运行之间可靠地完成清理。最后，本文在具有代表性的 ARM64 和 AMD64 主机上验证了这一方法，说明容器化虚拟化能够在混合硬件机群上支持接近原生的交互式工作流。

论文结构： 第 2 节回顾恶意软件沙箱以及基于虚拟化/容器化的方法。第 3 节介绍 POKISEC 的系统设计和方法，包括高级架构和关键组件。第 4 节结合实现细节与概念验证工作流，描述通用入口点、加载器到虚拟机监控器的交接以及临时持久化模型。第 5 节报告跨架构的评估结果，第 6 节总结局限性并提出未来工作的方向。

2 相关工作

动态恶意软件分析长期以来一直借助沙箱化执行，在受控环境中观察恶意软件的行为。早期广泛使用的开源系统（如 Cuckoo Sandbox）推动了自动化引爆流程的发展

这些流程会在经插桩的虚拟机中执行样本，并收集行为产物供后续分析。虽然这类系统功能强大，但通常需要复杂的主机配置、谨慎的依赖项管理，并且与特定的虚拟化假设紧密耦合。当团队尝试在不同机器上扩展分析、将引爆集成到自动化工作流中，或确保协作者使用一致的环境时，这些成本就会更加明显。

QEMU 和 KVM 等虚拟化技术构成了许多实用沙箱实现的基础。QEMU 提供可移植的机器仿真和虚拟化能力，而 KVM 则能在受支持的主机上实现硬件辅助加速，缩小仿真与近原生执行之间的性能差距。然而在实践中，要配置 QEMU/KVM 以确保客户机可靠执行并非易事：不同架构的机器类型、设备模型和加速标志各不相同，而针对特定主机的“已验证可用”配置通常被写入临时脚本中。在多架构环境中，这一配置负担更加突出，因为相同的客户机工作流必须同时运行于 ARM64 和 x86 64 主机上。_

容器化通过打包依赖项并规范运行时环境，改变了软件部署方式。尤其是 Docker，已成为分发可复现工具和实现轻量级隔离的事实标准机制。尽管如此，对于恶意软件引爆场景，仅靠容器并不足够，因为恶意行为通常是在完整的客户机操作系统中观测的，而非作为单个进程运行。因此，一种日益普遍的设计模式是将沙箱技术栈本身容器化，把编排逻辑、虚拟化后端和访问接口打包到可移植镜像中。pokiSEC 顺应了这一方向，同时着重解决实践中经常未能充分满足的两项需求：一流的跨架构执行能力，以及真正临时性的拆除语义，以减少跨运行污染。

最后，向 Windows on ARM 环境及更广泛的 ARM64 生态系统迈进，为沙箱工程开辟了一个新的维度。虽然 ARM64 主机已支持 Windows 客户机，但对分析人员而言，实际情况是许多成熟的操作方案和工具链都是以 x86_64 默认值为前提编写的。弥合这一差距，不仅需要支持 ARM64 客户机和主机，还需要通过工具隐藏架构复杂性，提供一致的用户体验。pokiSEC 将架构检测与重新配置视为首要设计目标，而非事后补充，并将这种可移植性与网页驱动的引爆工作流相结合，降低交互式分析的入门门槛，从而为这一领域作出贡献。

3 系统概述与跨架构设计方法

3.1 设计目标与运行假设

pokiSEC 围绕四项核心目标进行设计，以满足实际的恶意软件引爆需求和可复现系统研究的要求：

1. **适用于异构主机的可移植性：**单一发行制品应能在 AMD64 和 ARM64 机器上运行，无需针对每台主机进行重新配置。2. **降低运维阻力：**部署只需容器运行时和标准虚拟化支持；所有其他依赖项都必须随附打包。3. **默认采用临时执行：**每次引爆运行都应从已知基线开始，并在结束时不保留任何主机端状态。4. **交互式可用性：**分析人员应能上传预先构建的 Windows 磁盘映像，并通过浏览器会话与客户机交互。

我们假设：(i) 宿主机内核支持硬件虚拟化（例如，x86_64 上的 VT-x/AMD-V，或 ARMv8 上的 EL2）；(ii) 容器运行时获准向容器开放相关设备接口（例如 /dev/kvm）。若无法使用此类加速，QEMU 可以回退到软件仿真；然而，由于交互式恶意软件分析对延迟十分敏感，本系统针对加速执行进行了优化。

3.2 拟议解决方案：将容器化虚拟化作为一等原语

我们介绍 **pokiSEC**，这是一种容器化引爆沙箱，通过将虚拟机监控程序和访问堆栈打包到单个 Docker 镜像中，抽象化了运行操作的复杂性 [21]。其运行时将 QEMU [22] 与 KVM 加速[23] 相结合，并提供基于 Web 的加载器，支持自带映像 工作流（例如 Windows QCOW2 镜像）。这种设计消除了主机级别的依赖项管理，使沙箱能够在不同机器和团队间移植。

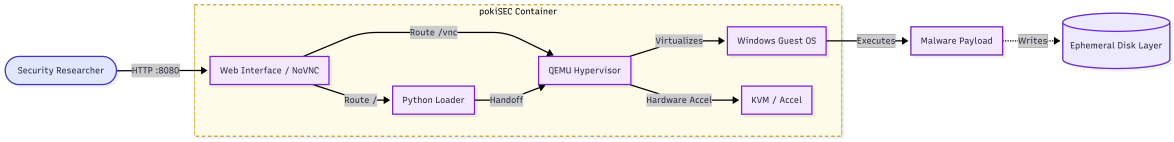


图 1：系统架构。反向代理将初始会话路由至用于镜像导入的加载器，然后在 QEMU/KVM 启动后切换至虚拟机桌面会话。

形式化地，令主机架构为

$$\mathcal{A} = \{\texttt{x86_64}, \texttt{aarch64}\},$$

并令虚拟化可用性为 $\mathcal{V} = \{0, 1\}$，表示硬件加速是否可访问（例如，存在 /dev/kvm 且具有相应权限）。pokiSEC 会选择一个启动配置向量

$$\mathbf{c} = (q, m, \alpha, \rho, \nu) \in \mathcal{C},$$

其中 q 是 QEMU 二进制文件, inary, m 是机器类型， α 是加速模式， ρ 是设备/网络配置，并且 ν 是远程显示协议概述。通用入口点实现

$$f : \mathcal{A} \times \mathcal{V} \to \mathcal{C}, \qquad \mathbf{c} = f(a, v), \tag{1}$$

确保单一代码路径在不同架构下生成经过验证的参数包。

3.3 高级架构与控制平面/数据平面分离

POKISEC 采用微虚拟化架构：主机运行容器引擎，而容器同时封装（i）用于摄取/编排的 控制平面，以及（ii）用于客户机执行和交互的 数据平面。

控制平面（加载器 +编排器）： 轻量级网页加载器（Flask）接受 QCOW2 磁盘镜像，执行输入验证，并将其放置到已知路径。入口点/编排脚本执行架构检测，选择 QEMU/KVM 配置，并管理进程生命周期。

数据平面（虚拟机监控程序 + 远程桌面）： 数据平面启动 QEMU/KVM，并通过 HTML5 VNC 客户端提供客户机桌面，从而支持仅通过浏览器访问以进行交互式分析。

路由与加载器到虚拟机监控器的交接： HTTP 反向代理在加载器用户界面（初始状态）与客户机桌面（活动状态）之间复用单个外部端点。我们将该工作流建模为一个有限状态机：

$$\mathcal{S} = \{\text{LOADER}, \text{VM_RUNNING}, \text{TERMINATED}\},$$

其转换为

$$\text{LOADER} \xrightarrow{e_u} \text{VM_RUNNING} \xrightarrow{e_t} \text{TERMINATED},$$

其中 e_u 表示成功的上传/验证事件，而 e_t 表示终止。路由策略为

$$r : \mathcal{S} \to \{\text{route_loader}, \text{route_vnc}\},$$

使得 r(加载器) = 路由_加载器，并且 r(虚拟机_运行中) = 路由_VNC。

3.4 通用入口点：架构自适应启动策略

在运行时，入口点确定 $a \in \mathcal{A}$和 $v \in \mathcal{V}$，并选择加速作为

$$\alpha(v) = \begin{cases} \texttt{kvm} & \text{if } v = 1, \\ \texttt{tcg} & \text{if } v = 0. \end{cases} \tag{2}$$

机器配置 $m(a)$ 从一个小型的经过验证的集合中选取，该集合中的配置已知能够在该主机指令集架构上可靠地启动目标客户机，因此选择开销为常数时间。

算法 1 跨架构引爆的通用入口**要求：** 磁盘映像 I (QCOW2)，可选支持暴露

KVM 的容器运行时 1： $a \leftarrow$ HOSTARCH() 2： $v \leftarrow$HASKVM() 3： $\mathbf{c} \leftarrow f($ $a,\ v)$ 4：STARTPROXY()

▷ 最初路由至加载器

5: **如果** IMAGEUPLOADED(I)
则6: STOPLOADER() 7:
LAUNCHQEMU(I，**c**) 8:
SWITCHPROXYTOVNC() 9: **结束如**
果10: **当** VMRUNNING()**时**11:
MONITOR() 12: **结束循环**13:
TEARDOWN()

3.5 分析性能模型

为分析用户感知的响应速度，我们将端到端交互时间分解如下：

$$T_{\text{tti}} = T_{\text{up}} + T_{\text{cfg}} + T_{\text{boot}} + T_{\text{handoff}} + T_{\text{vnc}}, \tag{3}$$

其中， T_{up} 为映像上传/放置时间， T_{cfg} 为配置选择开销， T_{boot} 为客户机启动时间，T_{handoff} 和 T_{vnc} 表示代理切换和首帧延迟。

上传时间主要由输入/输出决定，并随映像大小 $|I|$变化：

$$T_{\text{up}} \approx \frac{|I|}{B_{\text{eff}}} + T_{\text{fs}}, \tag{4}$$

其中 B_{eff} 表示有效吞吐量（网络或本地）， T_{fs} 表示文件系统放置开销 .

启动时间在很大程度上取决于架构和加速。设

$$T_{\text{boot}} = g(a, v),$$

并将混合环境下的预期启动时间建模为混合分布：

$$\mathbb{E}[T_{\text{boot}}] = \sum_{a \in \mathcal{A}} \sum_{v \in \mathcal{V}} \Pr[a, v]\ \mathbb{E}[g(a, v)]. \tag{5}$$

一个有用的派生指标是虚拟化效率比：

$$\eta(a, v) = \frac{\mathbb{E}[T_{\text{boot}}(a, v)]}{\mathbb{E}[T_{\text{boot}}(a, 1)]}, \tag{6}$$

用于量化相对于加速基线（$v = 1$）的性能减速。在评估中， η 凸显了工作负载对 KVM 访问的依赖程度。

3.6 隔离与风险形式化

尽管 POKISEC 主要是一项系统贡献，但以概率术语形式化遏制，有助于指导配置决策和未来的加固。令 E 表示恶意软件样本攻陷主机这一事件。标准的分解方式为：

$$\Pr[E] = \Pr[\text{escape}] \cdot \Pr[\text{reach} \mid \text{escape}] \cdot \Pr[\text{persist} \mid \text{escape}, \text{reach}]. \tag{7}$$

其中，逃逸 表示 客户机到主机或客户机到容器的逃逸，影响范围 表示影响主机资产或相邻网络的能力，持久性 捕获拆除后仍然存续的情况。

我们将逃逸概率与攻击面 评分联系起来。令 $\mathcal{K}$ 为暴露的虚拟机监控程序/IO组件集合（例如，设备仿真模块、半虚拟化接口），并定义

$$S(\mathbf{c}) = \sum_{k \in \mathcal{K}} w_k \cdot \mathbb{I}\{k \text{ enabled under } \mathbf{c}\}, \tag{8}$$

其中， w_k 对组件暴露进行加权， $\mathbb{I}\{\cdot\}$ 为指示变量。在有效速率为 λ的简单泊松漏洞模型下，可得界

$$\Pr[\text{escape}] \le 1 - \exp\big(-\lambda\, S(\mathbf{c})\big). \tag{9}$$

这并非声称能给出精确的现实概率；相反，它提供了一种有原则的方法来论证：在广泛的假设下，尽量减少暴露的组件（降低 $S(\mathbf{c})$)应当会单调降低逃逸可能性。

临时性主要针对持久性。如果拆除语义能够确保容器侧的可写状态被销毁，我们就可以建模

$$\Pr[\text{persist}] \approx \Pr[\text{externalized state}] \cdot \Pr[\text{re-attachment}], \tag{10}$$

其中，外部化状态表示恶意软件成功写入临时边界之外（例如，写入挂载卷或主机路径）的运行所占比例。这直接说明了采用严格卷策略和最小权限设备暴露的必要性。

3.7 多目标配置优化

尽管 pokiSEC 在当前实现中使用经过验证的基于规则的选择器（第 3.4 节），但其设计也可以表述为一个有原则的多目标优化问题，在可用性/性能与暴露风险之间进行权衡。这种表述有助于：(i) 从算法层面解释设计原理；(ii) 支持未来扩展，使系统能够学习或自动调整安全默认值。

令 $\mathbf{c} \in \mathcal{C}$ 表示如公式（1）所示的候选启动配置。我们定义两个主要目标：

- **响应性/吞吐量目标：** 最小化用户感知延迟，由公式（3）中的预期交互时间 $\mathbb{E}[T_{\text{tti}}(\mathbf{c})]$表示。
- **暴露度目标：** 最小化攻击面，由公式（8）中的 $S(\mathbf{c})$ 表示。

标准的标量化形式为：

$$\min_{\mathbf{c}\in\mathcal{C}} \; J(\mathbf{c}) \triangleq \omega_1\, \mathbb{E}[T_{\text{tti}}(\mathbf{c})] \;+\; \omega_2\, S(\mathbf{c}), \tag{11}$$

其中 $\omega_1, \omega_2 \ge 0$ 表示部署偏好（例如，交互式分析员工作站可能选择较大的 ω_1，而高风险实验室可能通过较大的 ω_2优先降低暴露度）。可行集受兼容性和资源约束控制：

（C1）架构可行性：（C2）	$\mathbf{c} \in \mathcal{C}(a),$	(12)
加速可行性：（C3）资源	$\alpha(\mathbf{c}) = \texttt{kvm} \Rightarrow v = 1,$	(13)
界限：（C4）输入/输出策	$\text{CPU}(\mathbf{c}) \le C_{\max},\;\; \text{MEM}(\mathbf{c}) \le M_{\max},$	(14)
略：（C5）网络策略：	$\text{VOL}(\mathbf{c}) \in \{\varnothing, \text{read-only}\},$	(15)
	$\rho(\mathbf{c}) \in \{\text{isolated}, \text{NAT}, \text{restricted}\}.$	(16)

这里， $\mathcal{C}(a) \subseteq \mathcal{C}$ 表示架构 a 上有效的配置，而 $C_{\max}$， $M_{\max}$ 表示主机施加的配置限制。约束 (C4)–(C5) 体现了遏制策略的选择，这些选择会直接影响持久性和影响范围（参见式 (7) 和式 (10)）。

帕累托解释： 式 (11) 等价于在 $(\mathbb{E}[T_{\text{tti}}],\;\; S)$ 的帕累托前沿上选择一个点。例如，启用更多设备功能可能会降低 T_{tti} ，但提高 S（$\mathbf{c}$）；反过来，禁用外围设备会降低暴露风险，但可能损害可用性。标量化提供了一种简洁、适合 ACM 的解释，说明 POKISEC 为何强调小型的经过验证的配置文件 $\mathcal{C}(a)$ 和数量有限的已启用设备。

与当前实现的关联： 在当前系统中，通用入口点通过从离散的预先验证的候选集中进行选择，实现了对式 (11) 的受约束常数时间近似：

$$\mathcal{C}_{\text{valid}}(a, v) \subset \mathcal{C}(a),$$

并选择

$$\mathbf{c}^\star = \arg\min_{\mathbf{c}\in\mathcal{C}_{\text{valid}}(a,v)} \Big(\omega_1\, \widehat{T}_{\text{tti}}(\mathbf{c}) + \omega_2\, S(\mathbf{c})\Big), \tag{17}$$

$\widehat{\ }$其中 T_{tti} 是一个经验预测器（例如，按配置文件测得的启动/握手时间小型表）。这使得优化过程显式化，同时将运行时决策开销保持为 $\mathcal{O}(1)$，与第 3.1 节中的运行目标一致。

稳健配置目标：在异构部署中，首选配置除了应快速且低暴露之外，还应稳定；也就是说，它应避免那些性能对主机特定因素（CPU 型号、虚拟机监控程序后端、内核版本以及加速功能可用性）高度敏感的配置文件。因此，我们用一个惩罚跨环境启动时间变异性的稳健性项来扩展多目标表述。令$\mathcal{E}$ 表示执行环境上的分布（例如 $(a,\ v)$ 对以及依赖主机的参数），并将环境 $e \sim \mathcal{E}$ 下的启动时间随机变量定义为 $T_{\mathrm{boot}}(\mathbf{c}; e)$。稳健目标为：

$$\min_{\mathbf{c}\in\mathcal{C}} \ \omega_1 \, \mathbb{E}_{e\sim\mathcal{E}}[T_{\mathrm{tti}}(\mathbf{c}; e)] + \omega_2 \, S(\mathbf{c}) + \omega_3 \, \mathrm{Var}_{e\sim\mathcal{E}}(T_{\mathrm{boot}}(\mathbf{c}; e)) \, , \tag{18}$$

其中，ω_1、ω_2、$\omega_3 \geq 0$ 为部署特定权重。方差惩罚项鼓励选择平均响应性良好且在不同主机间表现稳定的配置，从而提升引爆工作流的可复现性，并减少在 ARM64 与 AMD64 机器之间切换时出现的意外情况。

3.8 容量与并发模型

对于多用户或实验室部署，可将引爆请求建模为队列中的作业。设到达过程遵循速率为 λ 的泊松过程，平均服务率为 $\mu = 1/\mathbb{E}[D]$，其中 D 为虚拟机会议持续时间（启动 + 交互 + 关闭）。在 M/M/1 近似下，利用率为

$$\rho = \frac{\lambda}{\mu}, \qquad \rho < 1, \tag{19}$$

队列中的预期等待时间为

$$\mathbb{E}[W_q] = \frac{\lambda}{\mu(\mu - \lambda)}. \tag{20}$$

这些公式使部署方能够将期望的响应性转化为配置目标（例如，需要多少个并行工作进程才能将 $\mathbb{E}[W_q]$ 控制在阈值以下）。

3.9 关键技术与实现原理

pokiSEC 依赖以下组件：

- **Docker** [21]：封装沙箱技术栈，以实现可复现的部署。
- **QEMU** [22]：为不同架构提供统一的虚拟化/仿真基础设施。
- **KVM** [23]：在可用时提供硬件加速。
- **Flask (Python)**：实现用于镜像导入的加载器/控制平面。
- **反向代理 +HTML5 VNC**：提供可通过浏览器访问的桌面，并实现从加载器到虚拟机监控程序的平滑切换。

4 原型实现与概念验证工作流

本节介绍 POKISEC 的实现细节，以及用于验证跨架构执行、无缝用户交互和临时数据清理的端到端概念验证（PoC）工作流。我们的原型实现为容器化微虚拟化堆栈，其中单个 Docker 镜像封装了 (i) 用于摄取和编排的控制平面，以及 (ii) 用于客户机执行和交互式访问的数据平面。

4.1 实现概述

该原型由四个协同工作的组件构成：

- **通用入口点（编排器）：**一个基于 shell 的控制器，用于检测主机架构和虚拟化能力、选择经过验证的 QEMU/KVM 配置，并管理加载器和虚拟机监控程序进程的生命周期。
- **网页加载器（Flask）：**一个简洁的上传界面，无需命令行工具即可摄取自带映像（QCOW2）。
- **虚拟机监控程序运行时（QEMU/KVM）：**在条件允许时通过加速启动 Windows 客户机。

• **远程桌面 + 路由（NoVNC + 反向代理）：** 在浏览器中呈现客户机桌面，并在加载器状态与活动虚拟机状态之间复用单个端点。

我们将上传的磁盘映像记为 I，将启动配置记为 $\mathbf{c} \in \mathcal{C}$，其中 $\mathbf{c}$ 捆绑了 QEMU 二进制文件、机器类型、加速模式以及设备、网络和显示配置文件。在运行时，系统使用常数时间策略（第 3 节中的公式（1））计算 $\mathbf{c}$，并执行客户机。

4.2 通用入口点与架构自适应启动策略

POKISEC 的核心工程贡献在于通过单个容器映像实现动态架构检测 和 基于策略的启动选择。令 $a \in \mathcal{A} = \{\texttt{x86_64}, \texttt{aarch64}\}$ 表示主机指令集架构，并令 $v \in \{0, 1\}$ 表示硬件加速是否可用（例如，/dev/kvm 存在且具有访问权限）。入口点计算：

$$\mathbf{c} = f(a, v), \tag{21}$$

其中，f 为 a 和 v选择适当的机器类型和加速标志，同时将暴露的设备集保持在最小范围内。

在我们当前的原型中，f 通过针对 uname -m 的分支（用于 a）以及针对 KVM 的能力测试（用于 v）实现。加速选择如下：

$$\alpha(v) = \begin{cases} \texttt{kvm} & \text{if } v = 1, \\ \texttt{tcg} & \text{if } v = 0, \end{cases} \tag{22}$$

其中 tcg 表示 QEMU 的软件翻译回退。

实现细节： 以下代码片段展示了入口点所使用的架构自适应启动逻辑。（对于 ACM 投稿，我们建议使用 listings 渲染代码；此处采用逐字显示以确保可移植性。）

架构=$(uname -m)

如果 ["$ARCH" == "aarch64"]; 则 echo "检测到
ARM64 主机 -> qemu-system-aarch64"
qemu-system-aarch64 \ -M virt,highmem=off \ -cpu host \ -
accel kvm \ -device ramfb\ -drive file=${IMAGE_
PATH},format=qcow2

否则如果 ["$ARCH" ==“x86_64"]; 则 echo "检测到 x
86_64 主机 -> qemu-system-x86_64" qemu-system-x86_64
\ -cpu host \ -enable-kvm \ -drive file=${IMAGE_
PATH},format=qcow2

fi

时间复杂度： 选择过程为 $\mathcal{O}(1)$：检测和查找不会随 $|I|$ 的规模增长而扩展。端到端响应速度主要受上传和客户机启动影响，我们通过以下方式对其建模：

$$T_{\text{tti}} = T_{\text{up}}(I) + T_{\text{cfg}} + T_{\text{boot}}(\mathbf{c}) + T_{\text{handoff}}, \tag{23}$$

其中，T_{cfg} 是恒定时间的配置开销，而 $T_{\text{up}}(I)$随映像大小近似线性增长：

$$T_{\text{up}}(I) \approx \frac{|I|}{B_{\text{eff}}} + T_{\text{fs}}. \tag{24}$$

4.3 加载器到虚拟机监控器的交接与单端点用户体验

为支持非命令行界面的工作流程，POKISEC 提供网页加载器，可通过拖放界面接收 QCOW2 磁盘镜像。I 上传并通过验证后，系统便会切换至交互式桌面会话，无需用户更改端口或 URL。我们通过将自终止加载器 与入口点监控相结合来实现这一功能：

1. 加载器在固定端点（例如 localhost:8080）上运行并接收 I。2. 摄取成功后，加载器会自行终止（例如通过 os.kill）。3. 入口点检测到加载器退出后，会立即启动 QEMU，然后激活远程桌面路由。

我们将控制流形式化为一个有限状态机：

$$\mathcal{S} = \{\mathsf{LOADER}, \mathsf{VM_RUNNING}, \mathsf{TERMINATED}\},$$

其转换如下：

$$\mathsf{LOADER} \xrightarrow{e_u} \mathsf{VM_RUNNING} \xrightarrow{e_t} \mathsf{TERMINATED},$$

其中，e_u 是上传完成事件，e_t 是虚拟机终止。反向代理应用路由策略 $r(\cdot)$，使加载器端点仅在 LOADER 状态下可访问，而 NoVNC 端点仅在 VM_RUNNING 状态下可访问。这种隔离可减少意外的状态重叠，并提供一致的分析人员用户体验。

4.4 临时数据 l 持久化模型与清理保证 证

POKISEC 在容器边界强制实施非持久化。令 B 表示基线制品（容器映像层，以及在视为不可变输入时由分析人员提供的磁盘映像），令 W 表示容器的可写层。一次引爆运行会产生状态差异

$$\Delta \;=\; \Phi(I, X, \mathbf{c}), \tag{25}$$

其中 X 是已执行样本，而 $\Phi(\cdot)$ 记录其引发的更改（释放的文件、注册表修改、加密活动等）。当容器以非持久化语义（例如 –rm）启动时，拆除会强制执行：

$$W \leftarrow \varnothing \quad \text{on stop}, \tag{26}$$

这会移除 Δ的容器侧制品。

为分析残留 风险，我们将客户机内持久性与边界外持久性区分开来。定义事件 Pa s 恶意软件持久化 超出拆除范围：

$$\Pr[P] \;=\; \Pr[\text{externalized state}] \cdot \Pr[\text{re-attachment}], \tag{27}$$

外部化状态会记录逃逸临时边界的写入操作（例如主机挂载卷或配置错误的设备暴露）。这促使我们制定严格的策略：默认不使用主机挂载卷（或使用只读挂载卷），并采用最小权限设备暴露。

4.5 概念验证测试平台与实验流程

我们使用 具有代表性的 ARM64 工作站和 Windows on ARM 客户机映像验证了该原型 e.

测试环境。

- **主机硬件：** Apple MacBook Pro（M3 Pro，ARM64）。
- **客户机操作系统：** Windows 11 ARM64（将 VHDX 转换为 QCOW2）。
- **容器引擎：** Docker Desktop for Mac（已启用 VirtioFS）。

执行工作流程。

1. **部署：** 启动容器并绑定服务端口：
   docker run --rm -p 8080:8080 pann123/pokisec

2. **摄取：** 打开 http://localhost:8080 并上传 QCOW2 Windows 镜像（例如 10GB）。上传时间符合公式 (24)。

3. **启动 + 交接：** 一旦发生 e_u，加载器便会终止，入口点则会启动 QEMU，并使用 $\mathbf{c} = f(a, v)$，将路由切换至 NoVNC 桌面会话。用户仍停留在同一端点。
4. **引爆：** 在客户机中执行无害的验证制品（例如 EICAR 测试字符串），并观察其在标准 Windows 环境下的预期行为（如果策略允许，也包括网络可达性）。
5. **清理：** 停止容器并重新启动。正确的清理定义为先前引入的制品不存在：

$$\mathcal{A}_{t+1} \cap \mathcal{A}_t = \varnothing, \tag{28}$$

其中 $\mathcal{A}_t$ 是在临时边界内运行 t 期间引入的可观测制品集合（文件、注册表项、配置更改）。–rm 生命周期确保满足式（26）。

该概念验证展示的内容。 此概念验证验证了：(i) 通过通用入口点实现跨架构可行性；(ii) 适用于交互式分析人员的单端点加载器到桌面转换；以及 (iii) 通过临时拆除语义消除各次运行间的容器级持久化。更广泛的评估（第 5 节）可将此工作流扩展到多种主机指令集架构，量化 T_{tti} （式（23）），并测量交接路径相较于直接本地 VNC 的开销。

5 结果与评估

本节将从以下三个方面评估 POKISEC：（i）交互性能；（ii）跨架构可移植性；以及（iii）临时执行条件下的运行可重复性。评估重点关注面向用户的主要指标“交互时间”，并验证单一容器化堆栈能否在异构主机上可靠运行。

5.1 评估方法与指标

我们测 量以下指标，这些指标与式（23）中提出的分析分解相一致：

- **交互时间（T_{tti}）：** 从启动一次运行到浏览器中首次出现可用客户机桌面帧的经过时间。该时间包括上传（如适用）、配置选择、客户机启动以及加载器到 NoVNC 的交接。
- **客户机启动时间（T_{boot}）：** 从 QEMU 进程启动到 Windows 登录屏幕（或等效的稳定启动完成标志）的经过时间。
- **交接延迟（T_{handoff}）：** 从上传完成到 NoVNC 会话出现之间的时间，用于体现进程编排和路由切换开销。
- **跨架构成功率：** 在 ARM64 和 AMD64 主机上无需手动干预即可进入稳定 Windows 桌面的运行次数所占比例。

为隔离硬件加速的影响，我们在可用时比较加速虚拟化（启用 KVM）与软件仿真（TCG 回退），并通过效率比来解读结果：

$$\eta(a, v) = \frac{\mathbb{E}[T_{\mathrm{boot}}(a, v)]}{\mathbb{E}[T_{\mathrm{boot}}(a, 1)]}, \tag{29}$$

其中， $\eta > 1$ 表示相对于 KVM 加速基线的减速程度。

5.2 性能结果

在交互式引爆场景中，硬件辅助虚拟化相比纯仿真带来了显著的性能提升。在 ARM64 主机（Apple M3 Pro）上，Windows 11 在 KVM 加速下的启动时间约为**25 秒**，为分析任务提供了媲美实用裸机启动工作流的体验。这一结果符合预期：启用加速后，交互式工作负载可从指令转换开销的降低和 I/O 虚拟化的改进中获得尤为明显的收益。
根据公式 (23) 中的分解，观测到的端到端交互时间可解释为：

$$T_{\mathrm{tti}} \approx T_{\mathrm{up}} + T_{\mathrm{cfg}} + 25\mathrm{s} + T_{\mathrm{handoff}},$$

其中， T_{cfg} 在实际应用中始终为常数且数值较小；当通过网页加载器载入较大的磁盘映像（例如 $\sim$10 GB）时， T_{up} 占据主导。在本地部署中， T_{up} 强烈取决于有效吞吐量 B_{eff} （公式 (24)），并且可通过重复使用预制映像，或在适当情况下通过更快的路径暂存映像来分摊。

5.3 跨架构兼容性与可移植性

我们在现代分析人员环境所针对的两大主流主机架构上验证了 POKISEC：

- **ARM64：** Apple M3 Pro 使用 Docker Desktop for Mac（已启用 VirtioFS）。
- **AMD64：** 使用原生 Docker 的 Ubuntu 22.04 Server。

在这两个平台上，系统均成功完成了端到端集成测试，启动了稳定的 Windows 桌面，并通过基于浏览器的 NoVNC 界面提供了客户机会话。这证明了通用镜像特性：单一容器构建和单一编排路径即可支持混合硬件机群，无需逐主机重新配置。

形式化地，设 $\mathcal{E} = \{e_1, \ldots, e_n\}$ 为经过测试的环境集合，并设 $\mathbb{I}_{\mathrm{ok}}(\mathbf{c};\, e) \in \{0, 1\}$ 表示在环境 e 中，按照配置 **c** 成功启动来宾系统。可移植性目标是实现：

$$\forall e \in \mathcal{E},\ \mathbb{I}_{\mathrm{ok}}\big(f(a_e, v_e); e\big) = 1, \tag{30}$$

我们的集成测试在经过测试的 ARM64 和 AMD64 配置下满足这一要求。

5.4 临时执行下的运行可重复性

我们还通过在客户机内执行无害的验证制品（EICAR 测试字符串）、停止容器并重新启动一个全新实例，验证了其非持久性。与临时拆除模型（公式 (26)）一致，后续运行不会在容器边界内保留先前引入的制品。这支持了预期的引爆工作流：每次运行都从已知基线开始，并在结束时不携带容器范围内的状态。

5.5 研究结果总结

总体而言，评估表明：(i) 启用 KVM 的执行可为 Windows 来宾系统提供实用的交互式性能（例如，在 M3 Pro 上启动耗时 $\sim$25 秒）；(ii) 通用入口点使单一容器化沙箱栈能够在 ARM64 和 AMD64 主机上运行；(iii) 临时生命周期语义可实现可重复的拆除，适用于迭代引爆工作流。

6 结论

本文介绍了 POKISEC，这是一款多架构、容器化的恶意软件引爆沙箱，将虚拟化后端和浏览器访问界面打包到单个可复现的 Docker 镜像中。通过封装 QEMU/KVM 以及加载器到虚拟机监控程序的交接机制，POKISEC 减轻了动态恶意软件分析环境通常带来的运维负担。尤其是，通用入口点将依赖架构的 QEMU 配置抽象为一致的启动流程，使分析人员无需逐台机器重新配置，即可在 ARM64 和 AMD64 主机上运行可比的引爆工作流。由此构建的系统支持快速配置、通过网页桌面进行交互式分析，以及临时数据拆除语义，从而简化每次运行之间的清理工作。

除了可用性之外，POKISEC 还展示了在混合硬件机群中实现“便携式沙箱”的实用方向；在这些环境中，可复现性和低门槛上手正变得越来越重要。我们的评估表明，硬件辅助虚拟化能够提供适用于分析人员工作流程的交互式性能（例如，在 Apple Silicon 上启动 Windows 通常只需数十秒），同时在不同平台间保持统一的操作界面。

通过若干扩展，可以将 POKISEC 从引爆沙箱发展为更全面的威胁情报工作流。首先，我们计划通过对常见的 Windows 监控工具（如 Sysmon/Procmon）进行检测，在客户机内集成结构化遥测数据收集，并在拆除前将制品（进程谱系、注册表差异、文件操作和网络活动）导出为规范化的 JSON 架构。这样既能生成可重复、可供机器处理的输出，又能保留系统的临时执行模型。其次，我们计划加入自动化后处理，将收集到的事件关联起来，归纳出更高层次的行为和指标，以支持分诊和丰富化。最后，我们将探索本地的隐私保护型语言模型，根据捕获的日志生成简洁的事件摘要和建议的后续步骤，供分析人员参考，并重点确保过程透明（例如，引用底层事件），且可部署于物理隔离或受限环境中。

参考文献

[1] Yashas Hariprasad、SS Iyengar 和 Naveen Kumar Chaudhary。《保护未来：面向量子安全视频传输的高级加密》。*IEEE*消费电子学报，2024年。[2] Cliff Wang、Sundararaja S Iyengar 和 Kun Sun。《网络系统人工智能嵌入式保障》。施普林格，2023年。[3] Jerry Miller、Lawrence Egharevba、Yashas Hariprasad、Kumar KJ Latesh 和 Naveen Kumar Chaudhary。物联网网络中针对 DODAG 控制消息泛洪的网络安全攻击检测框架。载于国际信息安全、隐私与数字取证会议，第213–230页。施普林格，2022年。[4] Paul Barham、Boris Dragovic、Keir Fraser、Steven Hand、Tim Harris、Alex Ho、Rolf Neugebauer、Ian Pratt 和 Andrew Warfield。Xen 与虚拟化的艺术。载于第十九届*ACM*操作系统原理研讨会论文集（*SOSP '03*），第164–177页，2003年。

[5] Michael Vrable、Justin Ma、Jay Chen、David Moore、Erik Vandekieft、Alex C. Snoeren、Geoffrey M. Voelker 和 Stefan Savage。Potemkin 虚拟蜜罐农场的可扩展性、保真度与遏制。载于第二十届*ACM*操作系统原理研讨会论文集（*SOSP '05*），第148–162页，2005年。[6] Stephen Soltesz、Herbert Pötzl、Marc E. Fiuczynski、Andy Bavier 和 Larry Peterson。基于容器的操作系统虚拟化：一种可扩展、高性能的虚拟机监控器替代方案。载于第二届 *ACM SIGOPS/EuroSys* 欧洲计算机系统会议（*EuroSys '07*）论文集，第275–287页，2007年。[7] Carsten Willems、Thorsten Holz 和 Felix Freiling。利用 CWSandbox 实现自动化动态恶意软件分析。*IEEE*《安全与隐私》，5(2)：32–39，2007年。[8] Heng Yin、Dawn Xiaodong Song、Manuel Egele、Christopher Kruegel 和 Engin Kirda。Panorama：捕获系统级信息流以检测和分析恶意软件。载于第*14*届 *ACM* 计算机与通信安全会议论文集（*CCS '07*），第116–127页，2007年。[9] SS Iyengar、Seyedsina Nabavirazavi、Yashas Hariprasad、Prasad HB 和 C Krishna Mohan。人工智能增强型恶意软件检测：通过智能威胁识别提升安全性。载于实践中的人工智能：网络安全与取证理论及应用，第227–255页。施普林格，2025年。[10] SS Iyengar、Seyedsina Nabavirazavi、Yashas Hariprasad、Prasad HB 和 C Krishna Mohan。恶意便携式文档格式（PDF 文件）的混合检测：防范嵌入式 JavaScript 攻击。载于实践中的人工智能：网络安全与取证理论及应用，第257–290页。施普林格，2025年。[11] SS Iyengar、Seyedsina Nabavirazavi、Yashas Hariprasad、Prasad HB 和 C Krishna Mohan。用于数字取证的隐私保护人工智能（联邦学习）。载于实践中的人工智能：网络安全与取证理论及应用，第161–176页。施普林格，2025年。[12] SS Iyengar、Seyedsina Nabavirazavi、Yashas Hariprasad、Prasad HB 和 C Krishna Mohan。网络安全基础：理论、技术与应用。载于实践中的人工智能：网络安全与取证理论及应用，第27–87页。施普林格，2025年。[13] SS Iyengar、Seyedsina Nabavirazavi、Yashas Hariprasad、Prasad HB 和 C Krishna Mohan。数字取证中联邦学习的网络威胁情报与安全。载于实践中的人工智能：网络安全与取证理论及应用，第177–199页。施普林格，2025年。[14] SS Iyengar、S Nabavirazavi 和 Y Hariprasad。Hb，p.，和 Mohan，ck（2025）。用于数字取证的隐私保护人工智能（联邦学习）。实践中的人工智能：网络安全与取证理论及应用，第161–176页，2025年。[15] KJ Latesh Kumar、Yashas Hariprasad、KS Ramesh 和 Naveen Kumar Chaudhary。用于检测私有云环境中虚拟机攻击的人工智能驱动关联技术。载于网络系统人工智能嵌入式保障，第183–199页。施普林格，2023年。[16] Christian Rossow、Christian J. Dietrich、Chris Grier、Christian Kreibich、Vern Paxson、Norbert Pohlmann、Herbert Bos 和 Maarten van Steen。恶意软件实验设计的审慎实践：现状与展望。载于*2012*年 *IEEE* 安全与隐私研讨会（*SP*），第65–79页，2012年。[17] Alexandru Agache、Marc Brooker、Alexandra Iordache、Anthony Liguori、Rolf Neugebauer、Phil Piwonka 和 Diana-Maria Popa。Firecracker：面向无服务器应用的轻量级虚拟化。载于第*17*届 *USENIX* 网络系统设计与实现研讨会论文集（*NSDI '20*），第419–434页，2020年。[18] Manuel Egele、Theodoor Scholte、Engin Kirda 和 Christopher Kruegel。自动化动态恶意软件分析技术与工具综述。*ACM*《计算调查》，44(2)，2012年。[19] Ulrich Bayer、Andreas Moser、Christopher Kruegel 和 Engin Kirda。恶意代码的动态分析。《计算机病毒学期刊》，2(1)：67–77，2006年。[20] Artem Dinaburg、Paul Royal、Monirul I. Sharif 和 Wenke Lee。Ether：通过硬件虚拟化扩展进行恶意软件分析。载于第*15*届 *ACM* 计算机与通信安全会议论文集（*CCS '08*），第51–62页，2008年。[21] Dirk Merkel。Docker：用于一致开发与部署的轻量级 Linux 容器。《*Linux* 杂志》，2014(239)：2，2014年。[22] Fabrice Bellard。QEMU：一种快速、可移植的动态翻译器。载于 *USENIX* 年度技术会议，第41–46页，2005年。[23] Avi Kivity、Yaniv Kamay、Dor Laor、Uri Lublin 和 Anthony Liguori。KVM：Linux 内核虚拟机。载于*Linux* 研讨会论文集，第1卷，第225–230页，2007年。