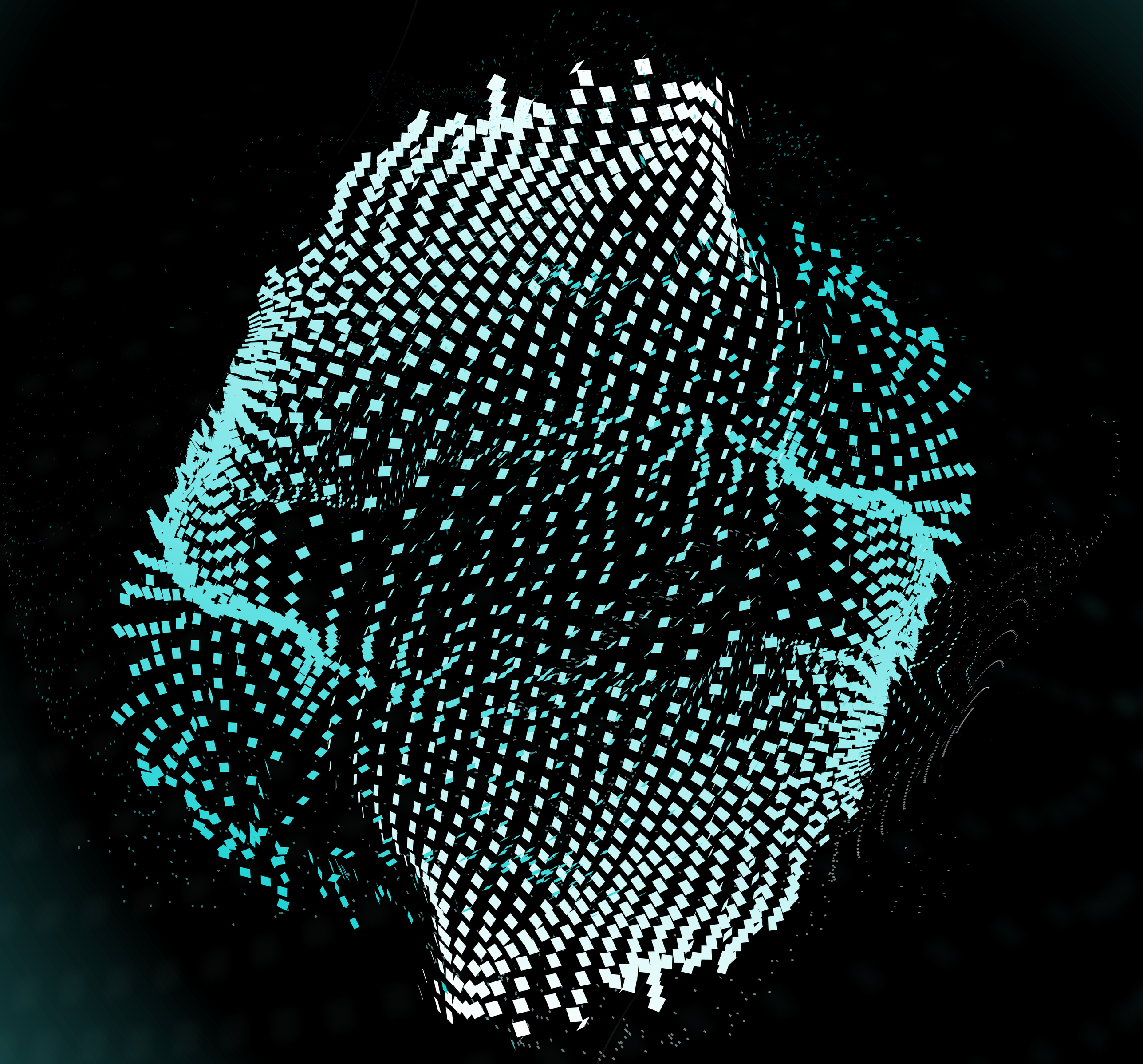
شمارهٔ یازدهم | آبان ۱۴۰۵ | دانشگاه صنعتی شـــریف

byte Magazine



انجمن علمی دانشکدهٔ مهندسی کامپیوتر

نشریهٔ عـــــلمی فرهنگی بایـــت

۰۳ ۳، ۲، ۱؛ ادامهٔ حرکت
سورنا کیا

۰۴ زحمت رندرش با شما!
محمدحسین اسلامی

۰۴ ماجرای مهندسی معکوس گرافیک Apple M1 در پروژهٔ Asahi Linux
آریان افضل‌زاده

۰۶ از کدنویسی تا اثبات: نیم‌نگاهی به زبان Lean
فرزام کوهی

۱۳ وقتی ابزار دفاع، ابزار حمله می‌شود
محمدامین کوهی

۱۷ فقط یک GPU، احساس ناکافی بودن
امیرحسین صوری

۲۱ امنیت وب در عصر پساکوانتوم
علی الماسی

۲۵ از LXC و داکر تا Firecracker
امیرحسین محمدزاده

۳۱ عشق، دروغ و سیلیکون؛ بحران شخصی در قلب صنعت تراشه
امیرحسین شهیدی

مدیر مسئول
معین آعلی

سردبیر
سورنا کیا

صاحب امتیاز
انجمن علمی دانشکدهٔ مهندسی کامپیوتر دانشگاه صنعتی شریف

صفحه‌آرایی و گرافیک
آرش شاه‌حسینی - امیررضا جعفری - محمدپارسا آرانی - امیررضا اینانلو - نرگس کاری - امیرحسین محمدزاده - فاطمه نیلفروشان

تحریریه
معین آعلی - آرمان طهماسبی‌زاده - امیرحسین صوری - فرزام کوهی - مهدی بهرامیان - نیما شیرزادی - امیرمهدی نامجو

نویسندگان
سورنا کیا - محمدحسین اسلامی - آریان افضل‌زاده - فرزام کوهی - محمدامین کوهی - امیرحسین صوری - علی الماسی - امیرحسین محمدزاده - امیرحسین شهیدی

ویراستاری
امیرحسین صوری - امیرحسین شهیدی

فنی
امیرحسین محمدزاده

۳، ۲، ۱؛ ادامهٔ حرکـــت

خانوادهٔ بایت، مسیری که هنوز ادامه دارد.



سورنا کیا
کارشناسی ۱۴۰۳
دانشگاه صنعتی شریف

۳ دقیقه

حدود یک سال پیش بود که یکی از دوستان سال‌بالایی‌ام، بایت را به من معرفی کرد و از شکل‌گیری بخش جدیدی در نشریه گفت؛ مجموعه‌ای که برای آغاز فعالیتش به‌دنبال اعضای تازه‌ای بود. آن روز شاید تصور نمی‌کردم پیوستن به این جمع، به یکی از تجربه‌های مهم دوران دانشجویی‌ام تبدیل شود. شناخت چندانی از فضای نشریه نداشتم و تصویر روشنی از مسیر پیش رویمان برایم شکل نگرفته بود. با این‌حال، پیشنهاد همکاری را پذیرفتم و به جمعی پیوستم که قرار بود سالی پرفرازونشیب را در کنار یک‌دیگر پشت سر بگذاریم.

سال گذشته، سال آسانی نبود؛ نه برای کشور و نه برای بایت. اعتراضات خونین، سایهٔ جنگ و در نهایت وقوع آن، روزهایی دشوار را رقم زدند. دانشگاه نیز برای نزدیک به ۹ ماه، از ۹ دی سال گذشته تا اول مهر امسال، تعطیل بود. با این‌حال، بایت در این روزها برخلاف باقی نشریات و کانون‌های فرهنگی، متوقف نشد. امیرحسین شهیدی، مدیرمسئول وقت نشریه، در شرایطی دشوار امور بیرونی را پیش برد و یکی از سخت‌ترین شماره‌های تاریخ دانشگاه را به چاپ رساند. آرش شاه‌حسینی، سردبیر وقت، نیز در دوران تعطیلی دانشگاه ارتباط اعضا را حفظ کرد و توانست در فرصتی یک‌هفته‌ای، دوباره جمع را کنار هم بیاورد و شماره‌ای تازه را آمادهٔ انتشار کند.

در میانهٔ اعتراضات و روزهایی که فضای کشور ملتهب بود، تصمیم گرفتم متنی برای بایت بنویسم. راستش انتظار نداشتم پیشنهادم چندان جدی گرفته شود؛ اما با استقبال امیرحسین روبه‌رو شدم و وقتی موضوع را با آرش مطرح کردیم، او هم با اشتیاق همراه شد.

همین اتفاق ساده باعث شد نگاهم به حضورم در بایت تغییر کند. آن شب، با استرس فراوان اما قلب‌هایی مطمئن، کنار هم نشستیم و سرمقاله‌ای نوشتیم. همان شب بود که «شام آخر» شکل گرفت؛ شامی که واقعاً شام آخر شد!

شاید این خاطره در نگاه اول، اتفاق کوچکی در میان تمام آن روزهای دشوار به نظر برسد؛ اما برای من، تصویری از چیزی است که بایت را ساخته است: فضایی که در آن ایده‌ها فرصت مطرح شدن پیدا می‌کنند، اعضا می‌توانند فراتر از مسئولیت اولیه‌شان نقش بگیرند و اعتماد، زمینه‌ساز مشارکت و همراهی می‌شود. آن‌چه بایت را بایت کرده، صرفاً انتشار چند شماره نیست؛ آدم‌هایی‌اند که در روزهای آسان و دشوار، تصمیم گرفته‌اند کنار یک‌دیگر بمانند و چیزی را با هم بسازند. شاید در اولین سرمقاله به‌عنوان سردبیر باید از موجی تازه، ایده‌هایی نو و مسیری متفاوت حرف بزنم. اما نه! پیش از هر چیز، باید به یاد داشته باشیم که این مسیر از پیش آغاز شده است. آرش و امیرحسین، در کنار دیگر اعضای نشریه، بخشی از راه را ساخته‌اند و آن‌چه امروز در اختیار ماست، حاصل تلاش و تجربهٔ جمعی آن‌هاست. تغییر مسئولیت‌ها به‌معنای شروع از نقطهٔ صفر نیست؛ هر شماره، هر ایده و حتی هر تجربهٔ ناموفق، بخشی از مسیری است که بایت تا امروز پیموده است. بایت قرار است نشریهٔ همهٔ دانشجویان دانشکدهٔ مهندسی کامپیوتر شریف باشد؛ جمعی که در آن تفاوت سلیقه‌ها و دغدغه‌ها نه مانع، بلکه فرصتی برای بهتر دیدن و بهتر ساختن است. امیدوارم در فصل تازهٔ فعالیت نشریه، صداهای بیشتری شنیده شوند و دانشجویان بیشتری فرصت پیدا کنند تا ایده‌ها و نگاهشان را با دیگران به اشتراک بگذارند. این مسیر، بی‌تردید دشواری‌های خودش را خواهد داشت؛ اما آن‌چه می‌تواند بایت را زنده و پویا نگه دارد، حضور آدم‌هایی است که برای ساختنش وقت می‌گذارند. پس به احترام تمام کسانی که تا امروز برای بایت نوشتند، ساختند و ماندند؛ آنان که بخشی از وقت و وجود خود را در این نشریه گذاشتند و از ادامه دادن نهراسیدند؛ و به امید تمام کسانی که قرار است از این پس عضوی از این خانواده شوند:

۳، ۲، ۱؛ ادامهٔ حرکت!



زحمت رندرش با شما!



محمدحسین اسلامی
کارشناسی ۱۴۰۱
دانشگاه صنعتی شریف

۷ دقیقه

در سال‌های نه‌چندان دور، ایجاد جلوه‌های بصری چشم‌گیر در بازی‌های ویدیویی بیش از هر چیز به یک عامل وابسته بود: قدرت خام پردازشی کارت‌های گرافیک. برای سال‌ها، مسیر پیشرفت صنعت بر افزایش تعداد ترانزیستورها، بزرگ‌تر شدن معماری‌های پردازشی و تولید سخت‌افزارهای قدرتمندتر استوار بود. این رویکرد تا زمانی که محدودیت‌های فیزیکی و اقتصادی اجازه می‌داد، پاسخ‌گوی نیازهای روزافزون بازی‌ها بود؛ اما پرسش اساسی این بود که تا چه زمانی می‌توان این مسیر را ادامه داد؟ با پیچیده‌تر شدن محیط‌های بازی، افزایش رزولوشن‌ها و ورود فناوری‌هایی مانند رهگیری پرتو[1]، دیگر فقط افزایش قدرت سخت‌افزاری نمی‌توانست راه‌حل تمام چالش‌ها باشد. همین محدودیت‌ها زمینه را برای ظهور روش‌های جدیدی فراهم کردند که به‌جای تکیهٔ صرف بر قدرت پردازشی، از هوش مصنوعی برای دستیابی به کیفیت بصری بالاتر استفاده می‌کنند.

در همین راستا، شرکت NVIDIA در سال ۲۰۱۸ فناوری جدیدی با نام DLSS[2] معرفی کرد. DLSS راهکاری مبتنی‌بر هوش مصنوعی است که برای افزایش کیفیت تصویر و بهبود عملکرد بازی‌های ویدیویی توسعه داده شده است. ایدهٔ اصلی این فناوری بر یک تغییر رویکرد اساسی استوار است؛ به‌جای این‌که کارت گرافیک تمام تصویر را با بالاترین رزولوشن ممکن تولید کند، بازی می‌تواند تصویر را در رزولوشنی پایین‌تر رندر کند و سپس با کمک مدل‌های یادگیری عمیق، نسخه‌ای با جزئیات بیش‌تر و کیفیت بالاتر از آن بازسازی شود. این روش باعث می‌شود فشار پردازشی روی GPU کاهش پیدا کند و منابع آزادشده برای بخش‌های سنگین‌تر -مانند رهگیری پرتو یا جلوه‌های بصری پیشرفته- مورد استفاده قرار گیرند.

در پشت این فرایند، یک شبکهٔ عصبی قرار دارد که با استفاده از حجم عظیمی از تصاویر با کیفیت بالا آموزش دیده است تا بتواند جزئیات از دست‌رفته در تصاویر کم‌رزولوشن را پیش‌بینی و بازسازی کند. برخلاف روش‌های سنتی افزایش رزولوشن که تنها بر بزرگ‌تر کردن تصویر و تخمین سادهٔ پیکسل‌ها تکیه دارند، DLSS از اطلاعات بیش‌تری مانند فریم‌های قبلی و نحوهٔ حرکت اجسام در صحنه استفاده می‌کند تا تصویری پایدارتر و نزدیک‌تر به خروجی رندر اصلی تولید کند. به همین دلیل، DLSS تنها یک ابزار افزایش رزولوشن نیست، بلکه نمونه‌ای از حرکت صنعت بازی به سمت استفاده از هوش مصنوعی برای بازسازی و تولید محتوای گرافیکی در زمان واقعی است.

تاریخچهٔ DLSS و نسخه‌های آن

همان‌طور که اشاره شد، اولین نسخهٔ DLSS در سال ۲۰۱۸ معرفی شد. ایدهٔ اولیهٔ این فناوری ساده بود: به‌جای رندر مستقیم بازی در رزولوشن بالا، تصویر در رزولوشنی پایین‌تر تولید می‌شود و سپس یک شبکهٔ عصبی آموزش‌دیده، جزئیات از دست‌رفته را بازسازی می‌کند. با وجود ایدهٔ نوآورانه، DLSS 1 نتوانست انتظارات را به‌طور کامل برآورده کند. مدل هوش مصنوعی این نسل برای هر بازی به‌صورت جداگانه آموزش می‌دید و در بسیاری از موارد خروجی نهایی با مشکلاتی مانند تاری تصویر و ایجاد ناهماهنگی‌های بصری همراه بود؛ به گونه‌ای که در برخی شرایط، تصویر رندرشده در رزولوشن اصلی کیفیت بهتری ارائه می‌داد.

با معرفی DLSS 2 در سال ۲۰۲۰، این فناوری وارد مرحله‌ای جدید شد. NVIDIA در این نسخه رویکرد آموزش مدل را تغییر داد و به‌جای استفاده از مدل‌های اختصاصی برای هر بازی، از یک مدل عمومی‌تر استفاده کرد که می‌توانست در بازی‌های مختلف عملکرد بهتری داشته باشد. همچنین، DLSS 2 با استفاده از اطلاعات زمانی، یعنی داده‌های مربوط به فریم‌های قبلی، توانست تصمیم‌های دقیق‌تری برای بازسازی تصویر بگیرد. این تغییر باعث افزایش چشمگیر کیفیت تصویر شد و DLSS را به یکی از فناوری‌های مهم در بهینه‌سازی گرافیک بازی‌ها تبدیل کرد. این فرایند با استفاده از هسته‌های Tensor -سخت‌افزارهایی اختصاصی برای انجام محاسبات هوش مصنوعی- انجام می‌شد.

در سال ۲۰۲۲، DLSS 3 با معرفی قابلیت Frame Generation گام متفاوتی برداشت. در نسخه‌های قبلی، هوش مصنوعی وظیفهٔ بازسازی یک تصویر با رزولوشن بالاتر را بر عهده داشت، اما در DLSS 3 سیستم قادر شد فریم‌های کاملاً جدیدی بین فریم‌های واقعی تولید کند.

این کار با کمک اطلاعاتی مانند بردارهای حرکت و داده‌های مربوط به تغییرات صحنه انجام می‌شد و می‌توانست نرخ فریم قابل‌مشاهده را به میزان قابل‌توجهی افزایش دهد. البته این روش در کنار روان‌تر کردن تصاویر، محدودیت‌هایی مانند افزایش تأخیر ورودی داشت و به سخت‌افزارهای جدیدتر سری RTX 40 نیاز داشت.

DLSS 4 که همراه با معرفی معماری Blackwell و کارت‌های RTX 50 در سال ۲۰۲۵ معرفی شد، دو تغییر مهم را به‌همراه داشت. نخستین قابلیت، Multi Frame Generation بود که امکان تولید چندین فریم مصنوعی میان فریم‌های رندرشده را فراهم کرد.

دومین تغییر، جایگزینی معماری سنتی‌تر شبکه‌های عصبی کانولوشنی با معماری Transformer بود؛ معماری‌ای که در سال‌های اخیر در بسیاری از مدل‌های پیشرفتهٔ هوش مصنوعی مورد استفاده قرار گرفته است. این تغییر به مدل اجازه می‌دهد ارتباطات پیچیده‌تر میان بخش‌های مختلف تصویر و تغییرات زمانی صحنه را بهتر درک کند و در نتیجه، پایداری و کیفیت بازسازی تصویر بهبود پیدا کند.

با وجود پیشرفت‌های هر نسل، DLSS تا این مرحله بر دو هدف اصلی تمرکز داشت: بازسازی تصویر با کیفیت بالاتر و افزایش نرخ فریم. اما DLSS 5 رویکرد متفاوتی را دنبال کرد؛ رویکردی که بیش‌تر به سمت استفاده از هوش مصنوعی برای درک و بازسازی هوشمندانه‌تر صحنه‌های گرافیکی حرکت می‌کند.

گام بعدی، DLSS 5

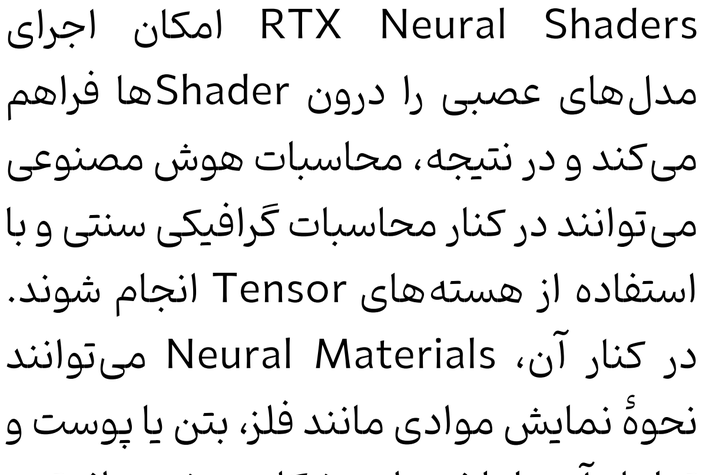
در مارس ۲۰۲۶، NVIDIA از نسل جدید فناوری DLSS با نام DLSS 5 رونمایی کرد؛ نسلی که برخلاف تمرکز عمدهٔ نسخه‌های قبلی بر بازسازی تصویر و افزایش نرخ فریم، قدم دیگری به سمت استفادهٔ مستقیم از هوش مصنوعی

۱. Ray Tracing
۲. Deep Learning Super Sampling

در فرایند تولید گرافیک برداشت. DLSS 5 با معرفی فناوری 3D-Guided Neural Rendering، از مدل‌های هوش مصنوعی برای ایجاد نورپردازی و موارد واقع‌گرایانه‌تر در صحنه‌های بازی استفاده می‌کند. این فناوری از سپتامبر ۲۰۲۶ به‌صورت رسمی در دسترس قرار گرفت و نخستین بار با بازی NBA 2K27 برای کارت‌های گرافیک سری GeForce RTX 50 ارائه شد.

DLSS 5 را نمی‌توان صرفاً نسخه‌ای سریع‌تر یا دقیق‌تر از فناوری‌های قبلی DLSS دانست. مهم‌ترین تغییر در این نسل، حرکت از «کمک گرفتن از هوش مصنوعی پس از رندر» به «استفادهٔ مستقیم از هوش مصنوعی در فرایند رندر» است؛ رویکردی که NVIDIA از آن با عنوان Neural Rendering یاد می‌کند. در نسل‌های پیشین، GPU ابتدا صحنه را با روش‌های مرسوم گرافیکی -مانند Rasterization، نورپردازی و Shading- پردازش می‌کرد و پس از تولید تصویر، مدل هوش مصنوعی وارد عمل می‌شد تا آن را بازسازی یا بهبود دهد. در DLSS 5، شبکه‌های عصبی به بخش‌هایی از خود فرایند رندر وارد می‌شوند و می‌توانند وظایفی را بر عهده بگیرند که پیش از این به‌صورت کامل با روش‌های سنتی گرافیکی انجام می‌شدند.





این تغییر را می‌توان در قالب مفهوم Neural Rendering بهتر درک کرد. در این رویکرد، هوش مصنوعی دیگر صرفاً وظیفهٔ اصلاح تصویر نهایی را ندارد، بلکه می‌تواند در تولید ظاهر و رفتار بخش‌هایی از صحنه نیز مشارکت کند. برای مثال، فناوری RTX Neural Shaders امکان اجرای مدل‌های عصبی را درون Shaderها فراهم می‌کند و در نتیجه، محاسبات هوش مصنوعی می‌توانند در کنار محاسبات گرافیکی سنتی و با استفاده از هسته‌های Tensor انجام شوند. در کنار آن، Neural Materials می‌توانند نحوهٔ نمایش موادی مانند فلز، بتن یا پوست و تعامل آن‌ها با نور را به شکل هوشمندانه‌تری مدل کنند. در حوزهٔ نورپردازی نیز، شبکه‌های عصبی می‌توانند با استفاده از تعداد محدودی از مسیرهای نور، اطلاعات ناقص را بازسازی کنند و به تولید تصویری که کیفیتی نزدیک به روشی سنتی مانند Path Tracing دارد، کمک کنند. حتی فشرده‌سازی بافت‌ها نیز می‌تواند با کمک مدل های عصبی انجام شود تا حجم دادهٔ مورد نیاز کاهش یابد، بدون آن‌که افت کیفیت روش‌های سنتی فشرده‌سازی به همان اندازه محسوس باشد.

نخستین تلاش‌ها برای بازسازی یک تصویر کم‌رزولوشن در DLSS 1 تا ورود Neural Rendering در DLSS 5 را می‌توان بررسی کرد و دید که مرز میان «رندر کردن» و «بازسازی کردن» به‌تدریج کم‌رنگ‌تر شده است. در رویکرد جدید، بخشی از آن‌چه پیش‌تر نیازمند محاسبات سنگین و مستقیم GPU بود، به مدل‌های هوش مصنوعی سپرده می‌شود؛ تغییری که می‌تواند مسیر توسعهٔ گرافیک بی‌درنگ را در سال‌های آینده تحت‌تأثیر قرار دهد.

اگر این روند ادامه پیدا کند، شاید در آینده دیگر پرسش اصلی این نباشد که GPU چقدر قدرتمند است، بلکه این باشد که GPU چقدر خوب می‌داند چه چیزی را لازم نیست خودش رندر کند و این شاید یکی از جالب‌ترین تغییراتی باشد که هوش مصنوعی تا کنون به دنیای گرافیک آورده است.

مسیر تکامل DLSS در نهایت نشان می‌دهد که نقش هوش مصنوعی در گرافیک رایانه‌ای دیگر به بهبود یک تصویر آماده محدود نمی‌شود.

ماجرای مهندسی معکوس گرافیک Apple M1 در پروژهٔ Asahi Linux



آریان افضل‌زاده
کارشناسی ۱۴۰۱
دانشگاه صنعتی شریف

۱۸ | ۶ دقیقه

وقتی اپل در سال ۲۰۲۰ نخستین مک‌های مجهز به تراشهٔ M1 را عرضه کرد، از CPU[1] و NPU[2] گرفته تا GPU[3] آن، طراحی اختصاصی اپل بود. برای این GPU، مستندات فنی عمومی و درایور متن‌بازی وجود نداشت. بنابراین، هر کسی که می‌خواست محیط دسکتاپ و برنامه‌های سه‌بعدی در سیستم‌عامل‌هایی به‌جز macOS روی این مک‌ها اجرا کند، باید به پردازش نرم‌افزاری در CPU متکی می‌ماند، یا برای فهمیدن سازوکار این سخت‌افزار و ساختن استک گرافیکی از صفر، تلاش می‌کرد. این مقاله، روایت همین فرایند است که در پروژهٔ Asahi Linux انجام شد.

آغاز کار آلیسا روزنتسوایگ

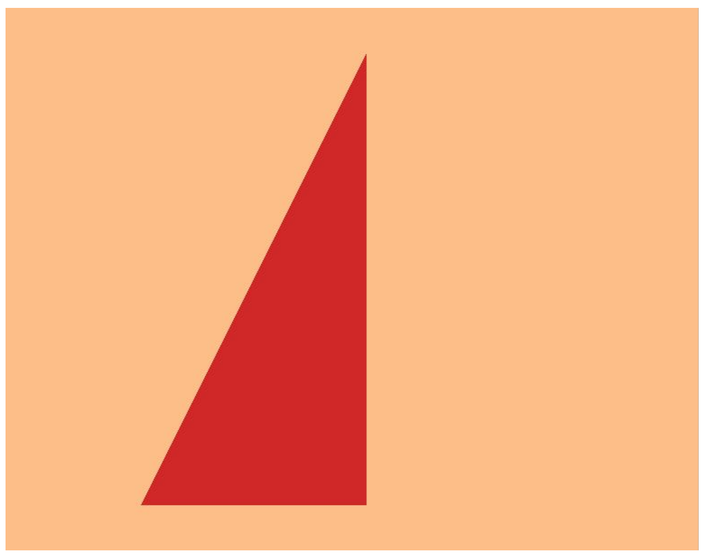
پروژهٔ Asahi Linux در دسامبر ۲۰۲۰ به ابتکار هکتور مارتین آغاز شد. هدف آن، اجرای مستقیم لینوکس روی مک‌های مبتنی بر تراشه‌های اپل بود. آلیسا روزنتسوایگ، دانشجوی علوم کامپیوتر از دانشگاه تورنتو، که روی Panfrost (درایور آزاد پردازنده‌های گرافیکی Mali) کار کرده بود، یکی از افرادی بود که به درخواست مارتین از توسعه‌دهندگان پروژه‌های متن‌باز برای همکاری، پاسخ داد. روزنتسوایگ ابتدا قصد داشت فقط چند توصیهٔ کوتاه بدهد، اما یک Mac mini خرید و خودش کار را در فضای کاربر[4] به دست گرفت.

اما چگونه این کار را در لینوکس، بدون وابستگی به درایور هسته انجام داد؟ پاسخ این است که به‌جای وابستگی به هستهٔ لینوکس[5]، کار در فضای کاربر را داخل macOS و با مهندسی معکوس درایور گرافیکی اپل انجام می‌داد. در روش او، کتابخانه‌ای کوچک به یک برنامهٔ آزمایشی تزریق می‌شود و فراخوان‌های میان برنامه و هسته را رهگیری می‌کند. هنگام ارسال دستورها به GPU، این کتاب‌خانه محتوای حافظهٔ مشترک را برای تحلیل ذخیره می‌کند. در macOS به‌جای LD_PRELOAD، باید از DYLD_INSERT_LIBRARIES و به جای ioctl، از چارچوب IOKit استفاده می‌کرد. فراخوانی مهم IOConnectCallMethod مسیر ارتباط برنامه‌های گرافیکی با درایور هسته بود. با دنبال کردن تخصیص حافظه، ایجاد بافر دستور و ارسال، می‌توانست محتوای حافظه را ثبت و سپس جداگانه بررسی کند. البته یافتن این فراخوانی‌ها نیز به‌دلیل نبود مستندات، ساده نبود.

نخستین هدف، شناخت مجموعه‌دستورهای Shader بود، یعنی کدی که روی GPU اجرا می‌شود. روزنتسوایگ Shaderهای بسیار ساده‌ای را با ابزار رسمی اپل کامپایل می‌کرد، تغییر کوچکی در کد می‌داد و خروجی دودویی دو نسخه را مقایسه می‌کرد. با تکرار این کار، بخش‌هایی از قالب دستورها، عملگرها و رجیسترها آشکار شد. او در ژانویهٔ ۲۰۲۱، ابزاری متن‌باز برای Disassemble کردن Shaderهای ساده منتشر کرد که کد دودویی اجرایی را به‌صورت تقریبی، به دستورات اسمبلی برمی‌گرداند تا مهندسی معکوس تسهیل شود. این عمل، با Decompile کردن که تبدیل کد دودویی به زبان سطح بالاست، تفاوت دارد.

رسم نخستین مثلث

در گرافیک سه‌بعدی، رسم نخستین مثلث نقطهٔ عطفی است، زیرا هر صحنه‌ای در نهایت از مثلث‌ها ساخته می‌شود. کم‌تر از یک ماه پس از شروع کار، نخستین مثلث با کد روزنتسوایگ روی M1 رسم شد، هرچند Shader رأس و Shader قطعه هنوز به زبان ماشین نوشته شده بودند.



او برای رسیدن به این نتیجه، گام به گام پیش رفت. نخست، رنگ پس‌زمینه را در بافرهایی که Metal (رابط گرافیکی اپل) ساخته بود تغییر داد. بعد، Shaderها را با نسخه‌های دست‌نویس جایگزین کرد. پس از آن، ساختارها را از نو ساخت تا هر خطا قابل‌ردیابی باشد. در این مسیر به مشکلی عجیب برخورد: حافظه‌ای که خودش با پارامترهای Metal تخصیص می‌داد، کار نمی‌کرد. سرانجام، معلوم شد جدولی در حافظهٔ GPU، فهرست تخصیص‌های مجاز را نگه می‌دارد و آخرین خانهٔ این جدول همیشه صفر است. رنگ موردنظر فقط زمانی ظاهر شد که شناسهٔ حافظهٔ جدید پیش از این خانهٔ پایانی قرار گرفت.

از زبان ماشین تا Mesa و باگ ناممکن

رسم مثلث فقط اثبات امکان‌پذیری روش بود. یک درایور واقعی، به کامپایلری نیاز داشت که زبان‌هایی مانند GLSL[6] را به دستورهای بومی GPU تبدیل کند. در ماه‌های بعد، روزنتسوایگ نوشتن چنین کامپایلری را برای Mesa (مجموعهٔ نرم‌افزاری گرافیک لینوکس) در macOS آغاز کرد و پس از یک هفته توانست با آن، Shaderهای پایهٔ رأس و قطعه را تولید کرده و یک مکعب سه‌بعدی سایه‌زده را رندر کند. تا مه ۲۰۲۱، درایور حدود ۷۵ درصد آزمون‌های OpenGL ES 2 را پشت سر می‌گذاشت و کد کامپایلر و درایور نیز به Mesa راه یافته بود.



یکی از خواندنی‌ترین بخش‌های این ماجرا مشکلی است که روزنتسوایگ آن را «باگ ناممکن» نامید. درایور، مدل‌های ساده را درست رسم می‌کرد، اما در مدل‌های پرجزئیات فقط بخشی از تصویر را می‌کشید و خطا می‌داد. پاسخ در معماری GPU یافت شد. با توجه به این‌که M1 یک طراحی بزرگ‌تر مبتنی‌بر تراشهٔ A14 است (Apple Family 7) که ریشه در گوشی‌های آیفون دارد، معماری گرافیکی آن «کاشی‌محور[7]» است. شایعات بر این بودند که طراحی‌های پیشین GPU های موبایلی اپل -از Apple Family 4

۱. Central Processing Unit
۲. Neural Processing Unit
۳. Graphics Processing Unit
۴. User Space
۵. Linux Kernel
۶. OpenGL Shading Language
۷. Tile-based
۸. Immediate-mode
۹. Middleware

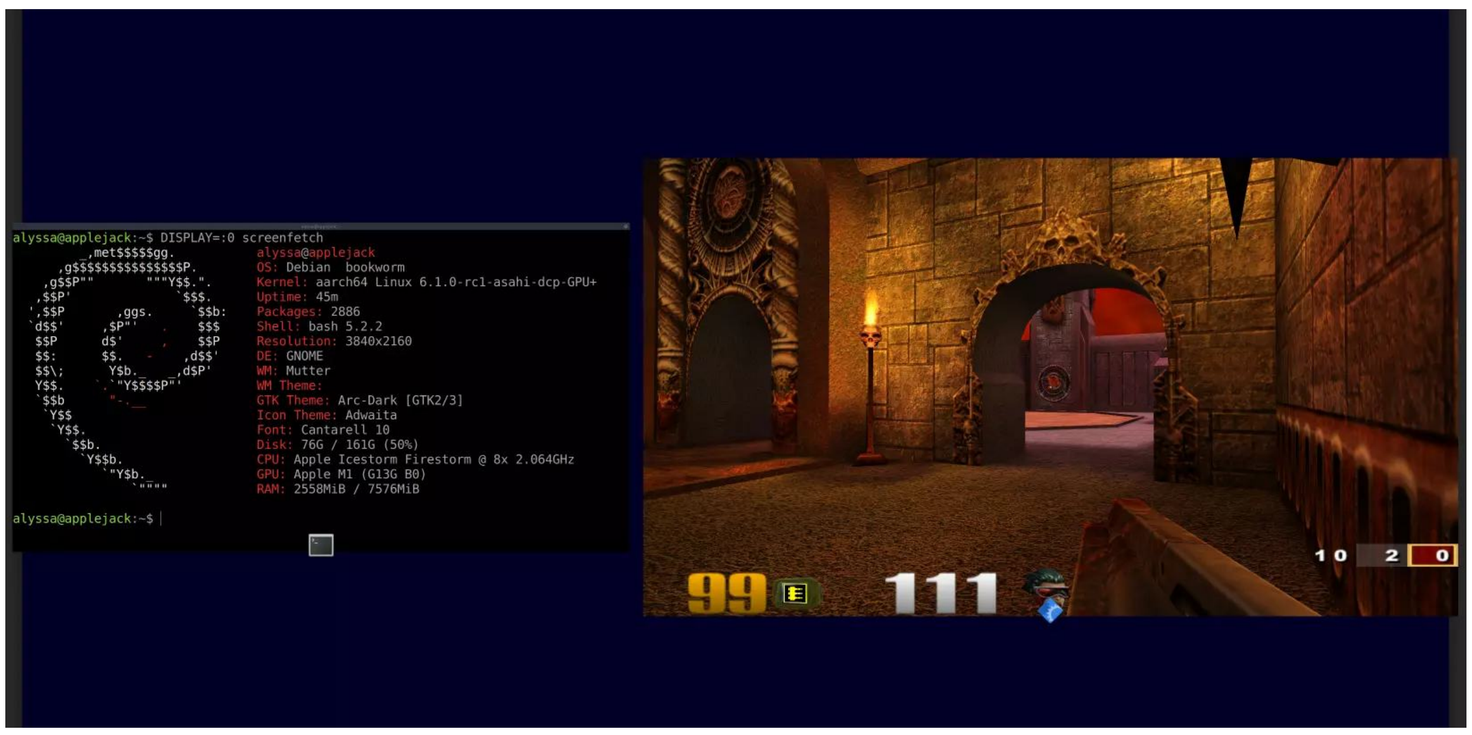
یا تراشهٔ A۱۱ به بعد- همین معماری را با الهام از PowerVR از شرکت Imagination داشتند، برخلاف معماری‌های متداول دسکتاپی که حالت‌فوری۸ هستند. با در نظر گرفتن تشابه احتمالی با معماری PowerVR، چنین پردازنده‌ای با استفاده از مفهوم «Tiled Vertex Buffer» (یا Parameter Buffer در PowerVR) ابتدا داده‌های همهٔ رأس‌ها را در بافری در حافظه ذخیره می‌کند و سپس تصویر را کاشی به کاشی می‌سازد. پس از پر شدن این بافر، رندر جزئی انجام می‌شود. درایور متن‌باز، کد لازم برای ادامهٔ کار پس از رندر جزئی را فراهم نکرده بود. با افزودن آن و تنظیم درست بافر عمق، مشکل حل شد.

درایور هسته و میان‌افزار[۹]

بخش بزرگی از کار اولیه در فضای کاربر و حتی روی macOS انجام شده بود و برای لینوکس، هنوز درایور هسته نیاز بود. «Asahi Lina» در سال ۲۰۲۲ این بخش را بر عهده گرفت و با ابرناظر[۱] m۱n۱ ارتباط macOS با GPU را در سطحی پایین‌تر رهگیری کرد. او دریافت که GPU اپل، یک پردازندهٔ کمکی از نوع ARM64 دارد که سیستم‌عامل بی‌درنگ یا RTOS[۲] اختصاصی اپل را اجرا می‌کند. تمام وظایف مدیریت GPU از جمله زمان‌بندی کارها، مدیریت توان و بازیابی از خطا بر عهدهٔ ثابت‌افزار[۳] این پردازندهٔ کمکی است، بدون دخالت هستهٔ macOS! بنابراین، نوشتن درایور هسته، مستلزم بازسازی پروتکلی بین هسته، میان‌افزار و واحدهای گرافیکی بود. لینا این درایور را به Rust نوشت؛ زبانی که در حال ورود رسمی به هستهٔ لینوکس بود و این درایور را به نخستین درایور گرافیکی به زبان Rust بدل می‌کرد. سرانجام، درایور هسته در کنار درایور فضای کاربر و کامپایلر روزنتسوایگ، سه بخش اصلی استک گرافیکی Asahi را تشکیل داد.

انتشار عمومی و پایان کار

تا اوت ۲۰۲۲، درایور Asahi علاوه‌بر رسم مثلث و مکعب، بازی‌های واقعی روی M۱ اجرا می‌کرد. در دسامبر همان سال، نخستین نسخهٔ عمومی درایور GPU برای Asahi Linux منتشر شد، که OpenGL ۲.۱ و OpenGL ES ۲.۰ را در حد اجرای محیط‌های دسکتاپ GNOME، KDE و بازی‌هایی مانند Quake III و Neverball روی GPU پشتیبانی می‌کرد.



از سال ۲۰۲۳ به بعد، پس از پایان تحصیل روزنتسوایگ، تلاش‌ها برای سازگاری درایور Asahi با OpenGL ۴.۶ و OpenGL ES ۳.۲ شروع شد، در حالی که macOS سال‌ها فقط از OpenGL ۴.۱ پشتیبانی کرده بود. برخی قابلیت‌هایی که M۱ پشتیبانی سخت‌افزاری مستقیم برای آن‌ها نداشت، با کمک Compute Shaderها پیاده‌سازی شدند. پس از آن، درایور Vulkan ۱.۳ به نام Honeykrisp نوشته شد. سپس، ترکیب Vulkan با DXVK، نرم‌افزار Proton و شبیه‌سازی x۸۶، راه اجرای بازی‌های ویندوزی را هموار کرد. روزنتسوایگ در جمع‌بندی سال ۲۰۲۵ نوشت که با سازگاری استک Asahi با OpenGL ۴.۶ و OpenGL ES ۳.۲، OpenCL ۳.۰ و Vulkan ۱.۳ و upstream شدن درایورها به Mesa، به اهدافش رسیده است. او با پیوستن به شرکت اینتل، از سازوکار اپل کناره گرفت و ادامهٔ کار را به دیگر اعضای Asahi سپرد. ماجرای GPU در Asahi Linux نمونه‌ای کم‌نظیر از مهندسی معکوس سخت‌افزار مدرن است. گروه پروژه، نه به مستندات داخلی اپل دسترسی داشت و نه می‌توانست از کد درایور رسمی استفاده کند. پیشرفت با مشاهده، مقایسه، آزمایش و ساخت فرضیه انجام شد: از تغییر یک رنگ در بافر فرمان و نوشتن Shader با زبان ماشین تا شناخت میان‌افزار و اجرای بازی‌های مدرن، همگی با اتکا بر اکوسیستم متن‌باز لینوکس و Mesa.

۱. Hypervisor
۲. Real-Time Operating System
۳. Firmware

از کدنویسی تا اثبات: نیم‌نگاهی به زبان Lean



فرزام کوهی
کارشناسی ۱۴۰۱
دانشگاه صنعتی شریف

۱۵ دقیقه

در برنامه‌نویسی، معمولاً از کامپیوتر می‌خواهیم که یک برنامه را اجرا کند و نتیجه‌ای را به‌عنوان خروجی به ما برگرداند. اما پرسش عمیق‌تری که مطرح می‌شود این است که آیا می‌توانیم علاوه‌بر اجرای یک برنامه، درستی آن را نیز بررسی کنیم؟

برای مثال، فرض کنید کد یک مسئلهٔ ساده، مانند مرتب‌سازی یک آرایه را با یک الگوریتم دلخواه نوشته‌ایم. با اجرای چندین آزمون، می‌توانیم تا حدودی مطمئن شویم که برای تعدادی از ورودی‌ها، عملکرد برنامه و الگوریتم درست است. با این‌حال، حتی هزاران آزمون هم ثابت نمی‌کنند که الگوریتم برای تمامی ورودی‌های ممکن درست است.

در این‌جا مفهومی به نام روش‌های صوری یا Formal Methods مطرح می‌شود. در روش‌های صوری، هدف این است که رفتار سامانه‌های نرم‌افزاری و سخت‌افزاری با استفاده از منطق و ریاضیات به شکلی مدل شود که بتوان برخی ویژگی‌های آن‌ها را به‌صورت دقیق بررسی و در مواردی اثبات کرد.

یکی از زبان‌ها و ابزارهایی که امروزه برای چنین کاری استفاده می‌شود، Lean است.

درستی یعنی چه؟ مفاهیم Specification و Verification

برای درک بهتر نقش Lean، ابتدا باید کمی دقیق‌تر مشخص کنیم که منظور از «درستی» یک برنامه چیست.

در مهندسی نرم‌افزار، درست بودن یک برنامه لزوماً به این معنا نیست که برنامه صرفاً بدون خطا اجرا شود. ممکن است برنامه اجرا شود، اما نتیجه‌ای تولید کند که با نیاز یا مشخصات مورد انتظار ما مطابقت نداشته باشد.

به همین دلیل، پیش از صحبت دربارهٔ اثبات درستی یک برنامه، باید مشخص کنیم که از آن برنامه چه انتظاری داریم. این انتظار به‌صورت یک مشخصه یا Specification بیان می‌شود.

مشخصه توصیف می‌کند که یک سامانه یا برنامه تحت چه شرایطی باید چه رفتاری داشته باشد.

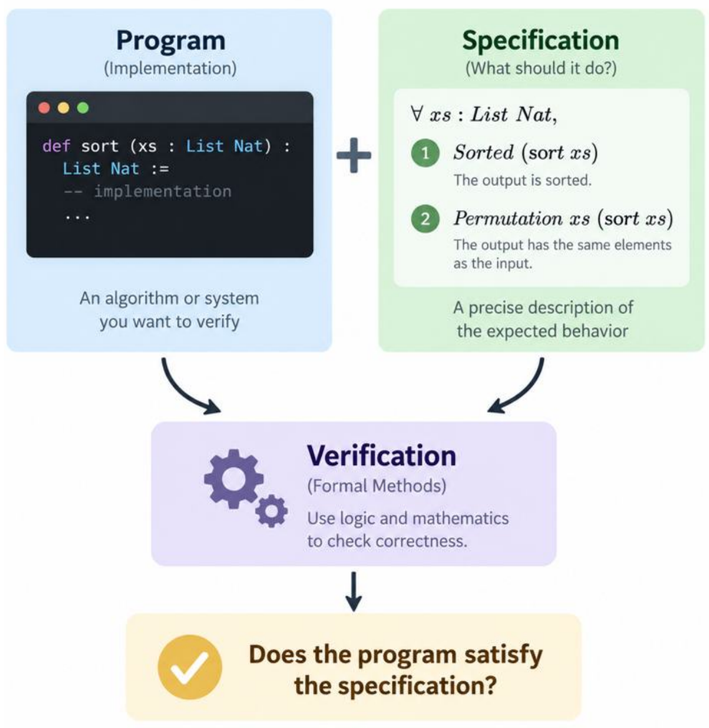
برای مثال در مسئلهٔ مرتب‌سازی، تنها این‌که تابع مرتب‌سازی یک فهرست خروجی تولید کند، کافی نیست. ممکن است از آن انتظار داشته باشیم که دست‌کم دو ویژگی زیر را برآورده کند.

۱. عناصر خروجی واقعاً مرتب باشند.

۲. خروجی دقیقاً شامل همان عناصری باشد که در ورودی وجود داشته‌اند و هیچ عنصری حذف یا اضافه نشده باشد.

در نتیجه، چیزی که در Formal Verification اثبات می‌کنیم، معمولاً این نیست که «برنامه به‌طور کلی درست است»، بلکه اثبات می‌کنیم که برنامه با یک Specification مشخص سازگار است.

به‌طور ساده می‌توان این رابطه را به‌صورت زیر نشان داد.



فرایند بررسی این‌که آیا یک سامانه، برنامه یا الگوریتم ویژگی‌های مورد انتظار خود را برآورده می‌کند یا نه، صحت‌سنجی یا Verification نام دارد.

صحت‌سنجی می‌تواند شکل‌های مختلفی داشته باشد. در روش‌های معمول مهندسی نرم‌افزار، این بررسی ممکن است از طریق آزمون، تحلیل ایستا یا بررسی دستی کد انجام شود. اما در Formal Verification، سیستم و ویژگی‌های مورد انتظار آن با استفاده از زبان‌های دقیق منطقی و ریاضی مدل می‌شوند و سپس تلاش می‌شود درستی آن‌ها به صورت ریاضی اثبات شود.

برای مثال، فرض کنید تابعی با نام sort داریم که یک لیست را مرتب می‌کند. می‌توانیم دو ویژگی مورد انتظار برای آن تعریف کنیم:

«برای هر فهرست xs، خروجی sort(xs) مرتب باشد.»

و همچنین:

«برای هر لیست xs، اعضای sort(xs) همان اعضای xs باشند.»

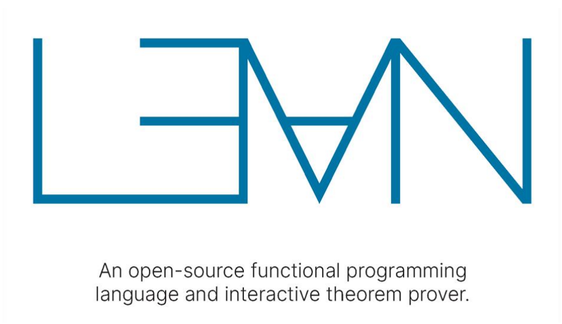
در Formal Verification هدف این است که چنین گزاره‌هایی نه فقط برای چند ورودی آزمایشی، بلکه برای تمام ورودی‌هایی که شرایط مشخص‌شده را دارند اثبات شوند.

در همین نقطه، تفاوت مهم میان Testing و Formal Verification مشخص می‌شود. تست نشان می‌دهد که برنامه برای تعدادی ورودی مشخص رفتار درستی داشته است، در حالی که اثبات صوری می‌تواند نشان دهد یک ویژگی مشخص، تحت فرض‌های معین، برای تمام حالت‌های مورد نظر برقرار است.

البته Formal Verification به این معنا نیست که بتوان هر ویژگی دلخواهی را دربارهٔ هر برنامه‌ای به‌سادگی اثبات کرد. نخست باید Specification به‌درستی نوشته شود و سپس اثبات تنها نسبت به همان Specification معنا دارد. اگر مشخصات ناقص یا اشتباه باشند، حتی یک اثبات کاملاً درست نیز لزوماً به‌معنای درست بودن سیستم از دید کاربر نخواهد بود.

اگر بخواهیم به شکل کلی‌تر تعریف کنیم، Formal Methods نام مجموعه‌ای از روش‌ها و ابزارهایی است که برای توصیف، مدل‌سازی، تحلیل و اثبات ویژگی‌های سیستم‌ها با استفاده از منطق و ریاضیات به‌کار می‌روند. Formal Verification یکی از مهم‌ترین کاربردهای این روش‌هاست.

ابزارهایی مانند Lean دقیقاً در همین چارچوب قرار می‌گیرند. Lean به ما اجازه می‌دهد ویژگی‌های مورد انتظار یک سیستم را به‌صورت گزاره‌های دقیق بیان کنیم و سپس برای آن‌ها اثبات‌هایی بسازیم که خود کامپیوتر قادر به بررسی صحت آن‌ها باشد.



Lean چیست؟

Lean در عین حال یک زبان برنامه‌نویسی و یک اثبات‌گر تعاملی قضایا[1] است.
این زبان بر پایهٔ نوعی از نظریهٔ انواع وابسته[2] طراحی شده و می‌توان از آن، هم برای تعریف داده‌ها و نوشتن برنامه‌ها و هم برای بیان و اثبات گزاره‌های ریاضی استفاده کرد.
از این نظر، Lean در نقطه‌ای جالب میان چند حوزه قرار می‌گیرد: برنامه‌نویسی تابعی، منطق ریاضی، نظریهٔ نوع‌ها و Formal Verification.
برای شروع، Lean آن‌قدرها هم با زبان‌های برنامه‌نویسی دیگر غریبه نیست. برای مثال، می‌توان تابعی تعریف کرد که یک عدد طبیعی را دو برابر کند:

```
def double (n : Nat):Nat :=
   n + n
```

در این کد، Nat نوع اعداد طبیعی است. بنابراین تابع double، یک مقدار از نوع Nat دریافت می‌کند و پس از دو برابر کردن آن، مقداری از همان نوع برمی‌گرداند.
برای اجرای این تابع نیز می‌توان نوشت:

```
#eval double 5
```

و نتیجه برابر خواهد بود با:

```
10
```

تا این‌جا، Lean شبیه یک زبان برنامه‌نویسی تابعی به نظر می‌رسد. اما ویژگی جالب آن، زمانی ظاهر می‌شود که بخواهیم دربارهٔ برنامه‌هایی که نوشته‌ایم استدلال کنیم.

از نوشتن برنامه تا اثبات ویژگی‌های آن

فرض کنید تابع سادهٔ زیر را نوشته‌ایم.

```
def addTwo (n : Nat) : Nat :=
  n + 2
```

اکنون می‌توانیم علاوه‌بر تعریف تابع، دربارهٔ آن یک قضیه نیز بیان کنیم.

```
theorem example (n : Nat) :
    addTwo n = n + 2 := by
  rfl
```

در این‌جا، دیگر تنها در حال تعریف یک برنامه نیستیم. داریم گزاره‌ای را بیان می‌کنیم که می‌گوید:
«برای هر عدد طبیعی n، خروجی addTwo n برابر با ۲ + n است.»
قسمت بعد از by =: اثبات این قضیه است.
دستور rfl از مفهوم reflexivity می‌آید و در حالتی استفاده می‌شود که دو طرف یک تساوی، پس از باز شدن تعریف‌ها عملاً یکسان باشند.
این مثال ساده است، اما یک ایدهٔ مهم را نشان می‌دهد: در Lean می‌توان یک گزاره را نوشت و سپس از سیستم خواست که بررسی کند آیا اثبات ارائه‌شده برای آن معتبر است یا خیر.

گزاره و Type؛ یکی از ایده‌های اصلی Lean

برای درک بهتر Lean، باید به یکی از ایده‌های مهم آن اشاره کنیم: در این زبان، ارتباط نزدیکی میان انواع و گزاره‌های منطقی وجود دارد.
در Lean، گزاره‌های منطقی در نوعی با نام Prop قرار می‌گیرند.
برای مثال، عبارت زیر یک گزاره است.

```
2 + 2 = 4
```

Lean می‌تواند نوع این عبارت را بررسی کند.

```
#check 2 + 2 = 4
```

و نتیجه نشان می‌دهد که این عبارت از نوع Prop است.
نکتهٔ جالب‌تر این است که در منطق Lean، اثبات یک گزاره نیز به‌نوعی یک مقدار محسوب می‌شود.
به‌صورت ساده می‌توان این ارتباط را چنین در نظر گرفت.

```
Type        ↔ Proposition
Program     ↔ Proof
```

این ایده بخشی از مفهومی معروف به Curry-Howard Correspondence است. در این نگاه، ساختن یک proof برای یک proposition شباهت زیادی به ساختن یک مقدار برای یک type دارد. به همین دلیل است که Lean می‌تواند اثبات‌ها را همانند اشیاء قابل‌بررسی در سیستم نوع خود پردازش کند.

Dependent Types؛ وقتی نوع به مقدار وابسته می‌شود

یکی دیگر از مفاهیم مهم Lean ، مفهومی به نام Dependent Types یا انواع وابسته است.
در زبان‌های برنامه‌نویسی رایج، نوع خروجی یک تابع معمولاً مستقل از مقدار ورودی است. برای مثال، ممکن است تابعی داشته باشیم که یک عدد صحیح دریافت کند و یک مقدار Boolean برگرداند.

```
Integer → Boolean
```

اما در یک dependent type، نوع خروجی می‌تواند به مقدار ورودی وابسته باشد.
برای درک ساده‌تر این موضوع، می‌توان یک بردار را تصور کرد که طول آن بخشی از نوع آن باشد.
مثلاً به‌جای این‌که فقط بگوییم:

```
List Integer
```

می‌توان چیزی مشابه این تصور کرد:

```
Vector Integer 5
```

یعنی برداری از اعداد صحیح که دقیقاً پنج عضو دارد.
در چنین سیستمی، تابعی که فقط روی بردارهای پنج‌عضوی کار می‌کند، می‌تواند این شرط را به‌طور مستقیم در type خود بیان کند.
این موضوع باعث می‌شود بخشی از خطاهایی که در زبان‌های معمولی هنگام اجرای برنامه دیده می‌شوند، پیش از اجرا و در مرحلهٔ بررسی typeها مشخص شوند.

Tactic چیست؟

اگر قرار بود برای هر قضیه، تمام جزئیات اثبات را به‌طور مستقیم به شکل proof term بنویسیم، کار بسیار دشوار می‌شد.
Lean برای ساده‌تر کردن فرایند اثبات، ابزارهایی به نام Tactic در اختیار کاربر قرار می‌دهد.
برای مثال:

```
theorem and_swap (P Q : Prop) :
    P ∧ Q → Q ∧ P := by
  intro h
  constructor
  · exact h.right
  · exact h.left
```

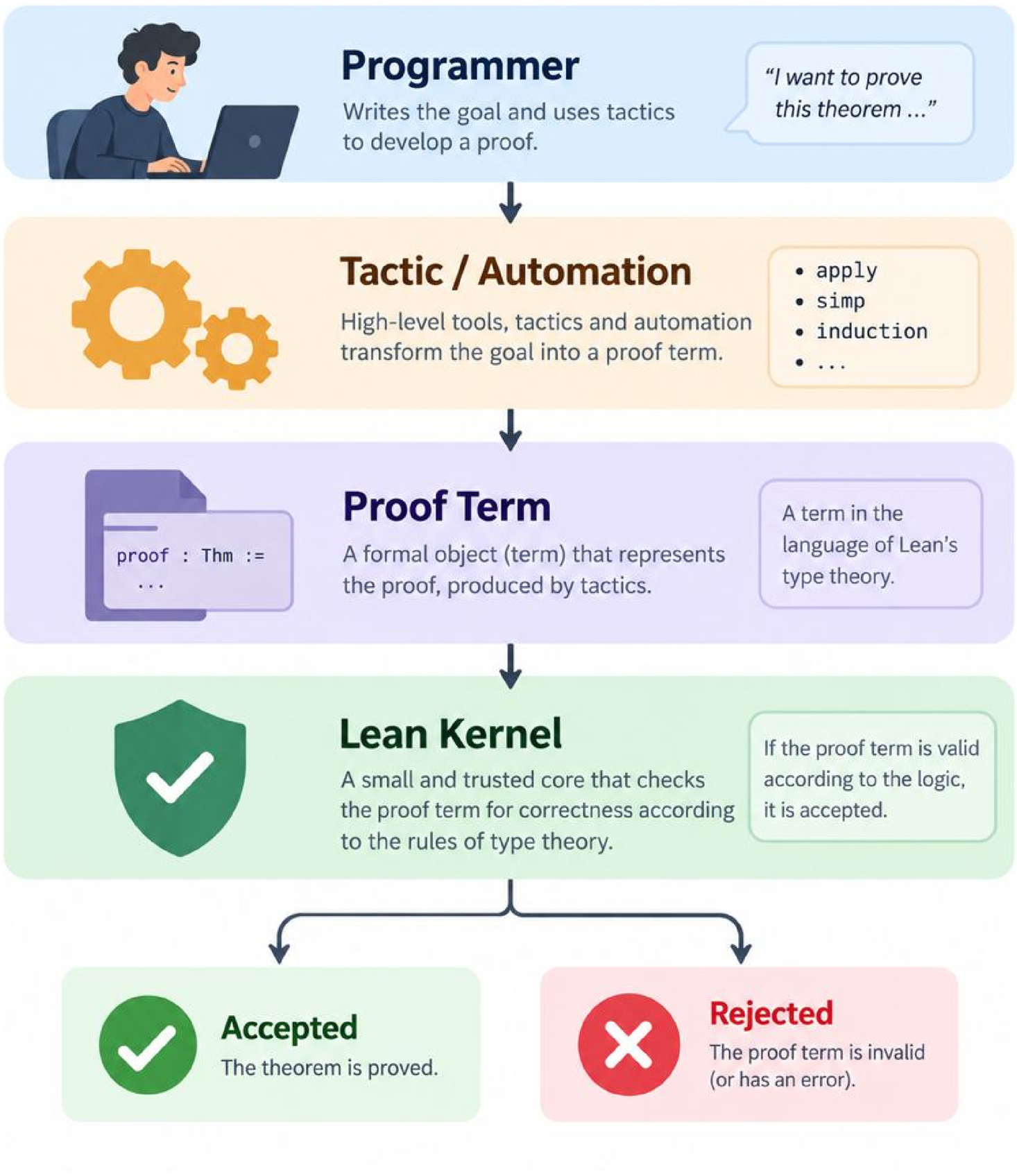
این قضیه می‌گوید:

```
P ∧ Q → Q ∧ P
```

یعنی اگر P و Q هر دو درست باشند، آن‌گاه Q و P نیز هر دو درست هستند.
در خط اول داریم:

```
intro h
```

۱. Interactive Theorem Prover
۲. Dependent Type Theory



یعنی فرض P ∧ Q را با نام h وارد محیط اثبات کن.

سپس:

```
constructor
```

از آن‌جا که باید Q ∧ P را ثابت کنیم، Lean مسئله را به دو هدف کوچک‌تر تقسیم می‌کند:

```
Q
P
```

سپس با:

```
exact h.right
```

قسمت دوم فرض h را برای اثبات Q استفاده می‌کنیم و با:

```
exact h.left
```

قسمت اول آن را برای اثبات P به کار می‌بریم.

Tacticهای متنوعی در زبان Lean وجود دارند، از جمله:

- `intro`
- `exact`
- `apply`
- `constructor`
- `rfl`
- `simp`
- `rw`
- `cases`
- `induction`

بعضی از این Tacticها عملیات بسیار ساده‌ای انجام می‌دهند و بعضی دیگر می‌توانند بخش قابل‌توجهی از روند اثبات را خودکار کنند.

اگر Tactic اشتباه کند، چه می‌شود؟

در این‌جا یکی از ویژگی‌های مهم معماری Lean مطرح می‌شود. فرض کنید یک Tactic بسیار پیچیده نوشته شده باشد که هزاران خط کد دارد و به‌طور خودکار برای یک قضیه proof تولید می‌کند. حالا اگر خود این Tactic دارای bug باشد، چگونه می‌توان به نتیجهٔ آن اعتماد کرد؟ Lean برای این مسئله از یک کرنل استفاده می‌کند. در نهایت، Tactic باید یک proof term تولید کند و کرنل بررسی می‌کند که این proof با قواعد منطقی سیستم سازگار است یا خیر. به‌صورت ساده، روند کار را می‌توان این‌طور نمایش داد.

اگر Tactic یک proof نادرست تولید کند، کرنل آن را نمی‌پذیرد.

این معماری باعث می‌شود برای اعتماد به صحت نهایی اثبات، لازم نباشد به تمام ابزارهای جانبی و Tacticها اعتماد کامل داشته باشیم. بخش اصلی اعتماد روی کرنل متمرکز می‌شود.

Lean فقط برای اثبات قضایای ریاضی نیست

وقتی نام theorem prover مطرح می‌شود، ممکن است در نگاه اول تصور کنیم چنین ابزارهایی تنها برای ریاضیات کاربرد دارند؛ اما Lean در مسائل مهندسی کامپیوتر نیز قابل‌استفاده است.

برای مثال فرض کنید تابعی برای دسترسی به عضو یک آرایه نوشته‌ایم.

در زبان‌های معمولی، ممکن است در زمان اجرا با خطایی مانند index out of bounds مواجه شویم. در یک سیستم نوع پیشرفته، می‌توان شرط معتبر بودن index را به بخشی از Specification تبدیل کرد. مثلاً از نظر مفهومی، تابع فقط زمانی اجازهٔ دسترسی به عضو آرایه را بدهد که بتوان ثابت کرد:

```
i < length array
```

در این حالت، بخشی از درستی عملیات وارد type و interface برنامه شده است.

این ایده را می‌توان به مسائل مهم‌تری نیز تعمیم داد که در ادامه آمده‌اند.

- اثبات صحت الگوریتم‌ها
- بررسی ساختارهای داده
- verification کامپایلرها
- بررسی پروتکل‌های ارتباطی
- سیستم‌های توزیع‌شده
- verification نرم‌افزار
- verification سخت‌افزار

به همین دلیل، Lean فقط ابزاری برای ریاضی‌دانان نیست و از دید مهندسی نرم‌افزار نیز اهمیت دارد.

Mathlib؛ کتاب‌خانهٔ بزرگ ریاضیات Lean

یکی از مهم‌ترین اجزای اکوسیستم Lean، کتابخانه‌ای به نام Mathlib است.
همان‌طور که در برنامه‌نویسی معمولاً کتابخانه‌های آماده را به‌جای پیاده‌سازی مجدد تمام الگوریتم‌ها و ساختارهای داده استفاده می‌کنیم، در theorem proving نیز منطقی نیست که هر قضیه و تعریف از ابتدا ساخته شود.
Mathlib مجموعهٔ بزرگی از تعریف‌ها، قضایا و اثبات‌های رسمی است که کاربران می‌توانند هنگام نوشتن اثبات‌های جدید از آن‌ها استفاده کنند.

برای مثال، اگر بخواهیم دربارهٔ اعداد، جبر، آنالیز، نظریهٔ مجموعه‌ها یا ساختارهای ریاضی دیگر کار کنیم، در بسیاری از موارد بخش مهمی از تعاریف و نتایج پایه از قبل در Mathlib وجود دارد.
در نتیجه، اثبات قضیهٔ جدید می‌تواند بر پایهٔ نتایج قبلی ساخته شود؛ درست همان‌طور که یک برنامهٔ جدید از کتابخانه‌های موجود استفاده می‌کند.

Lean به‌عنوان زبان برنامه‌نویسی

گاهی Lean تنها به‌عنوان یک ابزار اثبات معرفی می‌شود، اما Lean 4 در عین حال یک زبان برنامه‌نویسی تابعی نیز هست.
در Lean می‌توان از امکاناتی مانند موارد زیر استفاده کرد:

- functions
- pattern matching
- inductive data types
- type classes
- higher-order functions
- metaprogramming
- macros

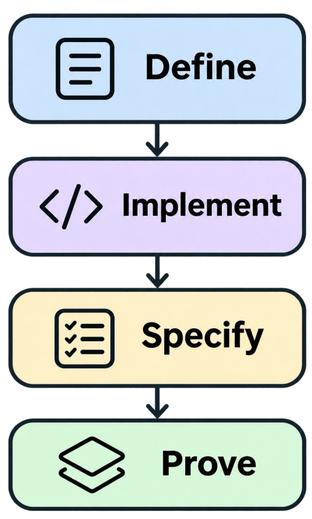
برای مثال، می‌توان یک نوع دادهٔ بازگشتی تعریف کرد.

```
inductive MyList (α : Type) where
  | nil
  | cons (head : α) (tail : MyList α)
```

یک نوع دادهٔ جدید برای لیست تعریف می‌کند. این نوع دو حالت دارد: nil که نشان‌دهندهٔ لیست خالی است و cons که یک عنصر (head) را به همراه ادامهٔ لیست (tail) نگه می‌دارد. با تکرار cons می‌توان لیست‌هایی با هر تعداد عنصر ساخت. سپس روی آن تابعی نوشت.

```
def MyList.length : MyList α → Nat
  | .nil ⇒ 0
  | .cons _ xs ⇒ 1 + length xs
```

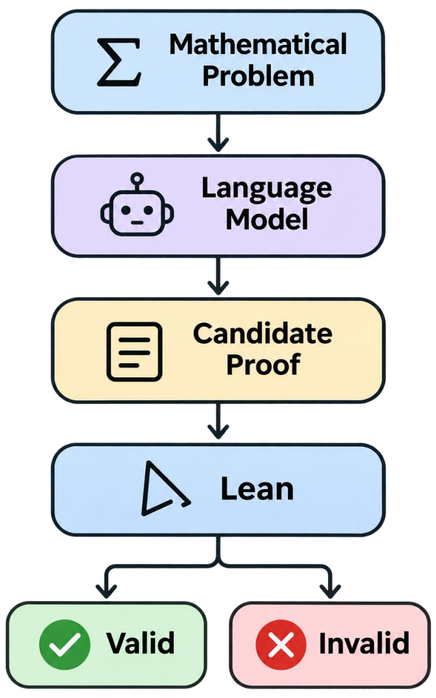
و پس از آن دربارهٔ همین تابع قضیه تعریف کرد و ویژگی‌های آن را اثبات کرد. تابع MyList.length برای محاسبهٔ تعداد عناصر موجود در یک لیست، از نوع MyList استفاده می‌شود. اگر لیست خالی (nil) باشد، مقدار صفر برگردانده می‌شود. در غیر این صورت، برای هر عنصر موجود در لیست، یک واحد به طول بخش باقی‌مانده اضافه می‌شود تا در نهایت تعداد کل عناصر به‌دست آید.
بنابراین در Lean می‌توان این مسیر را در یک محیط واحد طی کرد:



همین ترکیب برنامه‌نویسی و اثبات، یکی از ویژگی‌هایی است که Lean را از بسیاری از زبان‌های برنامه‌نویسی معمولی متفاوت می‌کند.

Lean و هوش مصنوعی

در سال‌های اخیر، theorem proving بار دیگر به یکی از موضوعات مهم در پژوهش‌های هوش مصنوعی تبدیل شده است. مدل‌های زبانی بزرگ می‌توانند متن، کد و حتی استدلال‌های ریاضی تولید کنند. اما تولید یک پاسخ با اثبات درستی آن پاسخ تفاوت دارد. ممکن است یک مدل زبانی پاسخی تولید کند که از نظر ظاهری منطقی باشد، اما در میان استدلال خود یک خطای کوچک داشته باشد. در یک theorem prover مانند Lean، شرایط متفاوت است. یک مدل هوش مصنوعی می‌تواند یک proof یا Tactic پیشنهاد دهد، اما در نهایت این Lean است که بررسی می‌کند آیا proof تولیدشده معتبر است یا خیر. به‌صورت ساده داریم:



اگر proof اشتباه باشد، Lean آن را رد می‌کند. حتی می‌توان این فرایند را به شکل یک حلقه در نظر گرفت. مدل یک Tactic پیشنهاد می‌دهد، Lean آن را اجرا می‌کند، اگر خطایی وجود داشته باشد آن خطا به مدل بازگردانده می‌شود و مدل تلاش دیگری انجام می‌دهد.
برای مثال، فرض کنید وضعیت اثبات در Lean به شکل زیر باشد.

```
P Q : Prop
h : P ∧ Q
⊢ Q
```

این وضعیت، یعنی Lean از ما می‌خواهد Q را اثبات کنیم و فرض h : P ∧ Q نیز در اختیار ماست.
یک مدل زبانی می‌تواند پیشنهاد دهد:

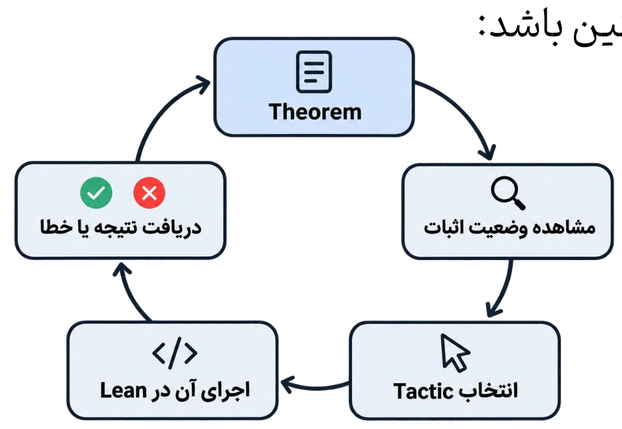
```
exact h.right
```

Lean این دستور را بررسی می‌کند. اگر معتبر باشد، هدف حل می‌شود؛ در غیر این صورت، خطا تولید خواهد شد.
در مسائل واقعی، مدل ممکن است Tacticهایی مانند rw، apply، cases یا induction را پیشنهاد دهد و حتی چندین مرحله از یک proof را بسازد.

Lean در ایجنت‌های هوش مصنوعی

تا این‌جا دیدیم که یک مدل زبانی می‌تواند برای Lean، Tactic یا proof پیشنهاد دهد و Lean صحت آن را بررسی کند. اما در سیستم‌های جدیدتر، هوش مصنوعی

می‌تواند نقش فعال‌تری داشته باشد و به شکل یک ایجنت عمل کند. یک ایجنت معمولاً به این صورت کار می‌کند: وضعیت فعلی را مشاهده می‌کند، تصمیم می‌گیرد قدم بعدی چه باشد، یک عمل انجام می‌دهد و سپس بر اساس نتیجه، تصمیم بعدی خود را می‌گیرد.
در theorem proving این چرخه می‌تواند چنین باشد:



در این حالت، Lean فقط یک زبان نیست؛ بلکه به نوعی محیط تعامل ایجنت هم محسوب می‌شود. ایجنت یک Tactic پیشنهاد می‌دهد و Lean وضعیت جدید اثبات یا پیام خطا را برمی‌گرداند.
برای مثال اگر Lean چنین هدفی داشته باشد:

```
n : Nat
⊢ n + 0 = n
```

ایجنت ممکن است یک Tactic مانند simp پیشنهاد دهد. اگر Tactic درست باشد، Lean هدف را حل می‌کند و اگر اشتباه باشد، ایجنت می‌تواند با استفاده از پیام خطا مسیر دیگری را امتحان کند.

ایجنت فقط Tactic تولید نمی‌کند

یک ایجنت پیشرفته می‌تواند چند کار مختلف انجام دهد:

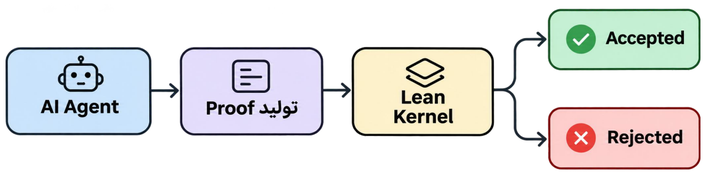
- وضعیت فعلی اثبات را تحلیل کند.
- در Mathlib به دنبال قضایا و لم‌های مرتبط بگردد.
- چند Tactic مختلف پیشنهاد دهد.
- مسیرهای ناموفق را کنار بگذارد.
- در صورت خطا، استراتژی خود را تغییر دهد.
- مسئلهٔ بزرگ را به چند زیرمسئلهٔ کوچک‌تر تقسیم کند.

به همین دلیل، theorem proving را می‌توان نوعی مسئلهٔ جست‌وجو نیز در نظر گرفت؛ جایی که ایجنت میان چند مسیر مختلف حرکت می‌کند تا در نهایت به یک اثبات کامل برسد.

ابزارهایی مانند LeanDojo نیز برای ایجاد چنین تعاملی میان مدل‌های هوش مصنوعی و Lean طراحی شده‌اند.

Lean به‌عنوان Verifier

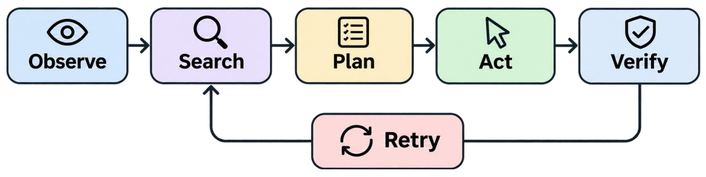
یکی از مهم‌ترین مزیت‌های استفاده از Lean در ایجنت‌ها این است که ایجنت مجبور نیست خودش دربارهٔ درست بودن پاسخ نهایی تصمیم بگیرد. می‌توان این فرایند را به شکل زیر خلاصه کرد:



در بسیاری از سیستم‌های مبتنی بر مدل زبانی، همان مدلی که پاسخ را تولید کرده باید دربارهٔ صحت آن نیز قضاوت کند. اما در این‌جا Lean به‌عنوان یک Verifier مستقل عمل می‌کند. اگر proof اشتباه باشد، حتی اگر از نظر ظاهری قانع‌کننده به نظر برسد، Lean آن را نمی‌پذیرد.

جست‌وجو، برنامه‌ریزی و بازخورد

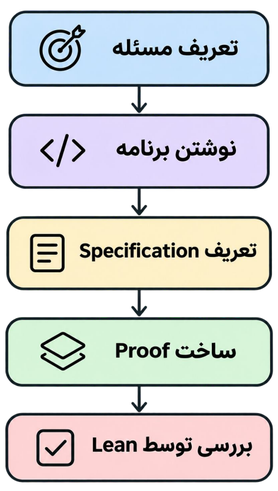
در سیستم‌های Agentic جدید، مدل می‌تواند علاوه‌بر تولید Tactic، از قابلیت‌های دیگری مانند جست‌وجو، بازیابی و برنامه‌ریزی نیز استفاده کند.
برای مثال، ممکن است ابتدا در Mathlib به‌دنبال قضایای مرتبط بگردد، سپس یک راهبرد برای اثبات طراحی کند و بعد آن را مرحله‌به‌مرحله در Lean اجرا کند.
اگر در میانهٔ مسیر به خطا برسد، می‌تواند به مرحلهٔ قبلی برگردد و مسیر دیگری را امتحان کند. بنابراین فرایند کلی می‌تواند چنین باشد:



همین چرخه باعث می‌شود استفاده از Lean در ایجنت‌های هوش مصنوعی جذاب باشد.
در واقع Lean محیطی فراهم می‌کند که ایجنت در آن می‌تواند حدس بزند، آزمایش کند، اشتباه کند و دوباره تلاش کند؛ اما در نهایت فقط پاسخی پذیرفته می‌شود که از بررسی رسمی Lean عبور کرده باشد.
به بیان ساده می‌توان گفت:
«AI می‌تواند استدلال و جستجو کند، اما Lean بررسی می‌کند که نتیجه واقعاً معتبر باشد.»

جمع‌بندی

Lean یکی از ابزارهایی است که مرز میان برنامه‌نویسی و منطق ریاضی را کم‌رنگ‌تر می‌کند.
در یک زبان برنامه‌نویسی معمولی، برنامه را می‌نویسیم و از کامپیوتر می‌خواهیم آن را اجرا کند.
اما در Lean می‌توان علاوه‌بر نوشتن برنامه، ویژگی‌هایی را که از آن انتظار داریم نیز به‌صورت دقیق بیان کرد و سپس برای آن‌ها proof ساخت.
به‌صورت خلاصه می‌توان مسیر را چنین دید:

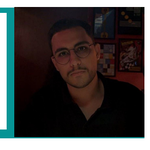


به همین دلیل، Lean تنها یک زبان برنامه‌نویسی یا صرفاً یک ابزار ریاضی نیست. این زبان در نقطه‌ای قرار دارد که برنامه‌نویسی، منطق، نظریهٔ نوع‌ها، Formal Verification و حتی هوش مصنوعی به یک‌دیگر می‌رسند.
شاید بتوان ایدهٔ اصلی Lean را در یک جمله خلاصه کرد:
«کامپیوتر فقط ابزاری برای اجرای برنامه نیست؛ می‌تواند ابزاری برای بررسی استدلال ما دربارهٔ درستی آن برنامه نیز باشد.»



وقتی ابزار دفـــــاع، ابزار حـــــمله می‌شود

داســـتان eBPF و روت‌کیتـــی کـه یـاد گـــرفت از دیـد لینوکـــس پنهـــان شـود



محمدامین کوهی
کارشناسی ۱۴۰۲
دانشگاه صنعتی شریف

۱۳ دقیقه

فرض کنید به یک سرور لینوکس SSH می‌زنید و حس می‌کنید یک جای کار می‌لنگد. ps را اجـرا می‌کنید؛ پردازهٔ عجیبـی دیده نمـی‌شود. اتصال‌های شبکه را بررسی می‌کنید و آن‌جا هم همه‌چیز عادی است. سراغ فایـل‌ها می‌رویـد و باز چیـز خاصـی پیدا نمی‌کنید.

احتمالاً در این مرحله با خودتان می‌گویید: «خب، پس مشکلی نیست.»

اما اگر مشکل دقیقاً همین باشد، چه؟ اگر مهاجم هنوز روی سیستم باشد، ولی کاری کرده باشد که ابزارهای خود سیستم نتوانند او را ببینند؟

این‌جا داستان eBPF شروع می‌شود.

eBPF؛ پنجره‌ای به داخل کرنل

اگر با لینوکس کار کرده باشید، احتمالاً می‌دانید بخش زیادی از اتفاقات مهم سیستم در کرنل (هستهٔ سیستم‌عامل که مستقیم با سخت‌افزار و منابع سیستم کار می‌کند) می‌افتد؛ از مدیریت process و حافظه گرفته تا فایل‌سیستم و شبکه.

برنامه‌های عادی ما در user space اجرا می‌شوند و وقتی می‌خواهند کاری مثل باز کردن یک فایل، ساختن process یا ارسال packet انجام دهند، معمولاً از طریق system call ها با کرنل ارتباط برقرار می‌کنند.

حالا فرض کنید بخواهیم دقیق‌تر ببینیم داخل کرنل چه اتفاقی می‌افتد. مثلاً بفهمیم کدام process فایل خاصی را باز کرده، چه system callهایی اجرا شده‌اند، یک packet از چه مسیری گذشته یا اجرای یک تابع در کرنل چقدر طول کشیده است.

این‌جاست که eBPF وارد می‌شود.

eBPF اجازه می‌دهد برنامه‌های نسبتاً کوچکی را به نقاط مشخصی از Kernel متصل کنیم. می‌توان آن را شبیـه قـرار دادن چند سنسور در بخش‌های مختلف سیستم تصور کرد؛ با این تفاوت که بعضی از این سنسورها فقط تماشاچی نیستند و می‌توانند روی جریان اجرای سیستم هم اثر بگذارند.

به همین دلیل، امروز eBPF در ابزارهای مانیتورینگ، شبکه، performance profiling و امنیت کاربرد زیادی دارد.

برنامه‌ها می‌توانند به هوک‌هایی مثل system callها، tracepointها، ورود و خروج تابع‌های کرنل و رویدادهای شبکه متصل شوند.

در واقع eBPF دست ما را برای دیدن اتفاقاتی که در لایه‌های پایین سیستم می‌افتند بسیار بازتر کرده است.

اما هرچه به کرنل نزدیک‌تر شویم، قدرت بیش‌تری هم داریم و دقیقاً همین‌جا پرسش اصلی مطرح می‌شود:

«اگر مدافع بتواند با eBPF اتفاقات کرنل را ببیند، چرا مهاجم نتواند با همان eBPF بعضی از آن‌ها را مخفی کند؟»

Verifier؛ ایست بازرسـی قبـل از ورود

لینوکس اجازه نمی‌دهد هر کدی همین‌طور وارد کرنل شود.

وقتی یک برنامهٔ eBPF قرار است لود شود، ابتدا از بخشی به نام verifier عبور می‌کند. verifier مسیرهای مختلف اجرای برنامه را بررسی می‌کند تا مثلاً برنامه به حافظهٔ خارج از محدوده دسترسی نداشته باشد، از متغیر مقداردهی‌نشده استفاده نکند یا در مسیری قرار نگیـرد که اجرای آن هیـچ‌وقت تـمام نشود.

بعد از تأیید نیز برنامه می‌تواند با JIT Compiler (کامپایلری که کد را به‌جای پیش از اجرا، هنگام اجرا به کد ماشین تبدیل می‌کند) به کد ماشین مخصوص همان پردازنـده تـبدیـل شـود تا با سـربار کمی اجـرا شود.

تا این‌جا همه‌چیز خوب به نظر می‌رسد، اما یک نکته وجود دارد:

verifier بررسی می‌کند که برنامه امن اجرا شود، نه این‌که هدف برنامه چیست.

ممکن است برنامه‌ای از نظر حافظه و ساختار کاملاً معتبر باشد، ولی هدفش مخفی‌کردن process یا سوکت متعلق به یک بدافزار باشد.

خود مستندات eBPF هم دقیقاً روی همین تفاوت تأکید می‌کنند: verifier ابزار بررسی ایمنی برنامه است، نه ابزاری برای تشخیص اینکه برنامه کار مخربی انجام می‌دهد یا نه.

یعنی از دید verifier تقریباً این اتفاق می‌افتد:

«کدت از نظر ساختاری سالمه، بفرما تو کرنل!»

البتـه هـــر user عادی هم نمی‌تواند چنین کاری کند

این‌جا شاید یک تصور اشتباه ایجاد شود: پس هر کسی که یک shell روی لینوکس داشته باشد، می‌تواند یک eBPF Rootkit بنویسد و آن را داخل کرنل بارگذاری کند؟ نه.

بارگذاری برنامه‌های eBPF قدرتمند، نیازمند privilege است. در کرنل‌های جدید، CAP_BPF برای انجام عملیات privileged مربوط به BPF تعریف شده است و بسته به نوع برنامه و محلی که قرار است به آن ضمیمه شود، قابلیت‌هایی مثل CAP_PERFMON یا CAP_NET_ADMIN هم ممکن است لازم باشند.

قابلیت‌ها را می‌شود به نوعی «تکه‌ای از قدرت root» در نظر گرفت.

یک process لازم نیست تمام اختیارات root را داشته باشد، لینوکس می‌تواند فقط اختیار مشخصی را به آن بدهد.

بعضی سیستم‌ها امکان اجرای محدودی از eBPF را برای userهای معمولی هم فراهم می‌کنند، اما این حالت محدودتر است و بسیاری از توزیع‌ها آن را غیرفعال می‌کنند.

برای قابلیت‌هایی در سطح چیزی که LinkPro انجام می‌داد، مهاجم باید قبل از آن به سطح دسترسی بالایی رسیده باشد.

پس eBPF وسیله‌ای جادویی برای تبدیل یک user معمولی به root نیست.

مهاجم اول باید جای پای مناسبی در سیستم پیدا کند؛ بعد eBPF می‌تواند به ابزاری برای مخفی‌شدن و ادامهٔ عملیات تبدیل شود.

و اتفاقاً داستان ما دقیقاً همین را نشان می‌دهد.

قبل از eBPF، روت‌کیت‌ها چه می‌کردند؟

روت‌کیت چیز جدیدی نیست. هدف آن معمولاً این است که مهاجم بعد از نفوذ بتواند حضور خودش را در سیستم پنهان کند؛ مثلاً process اجرا شود، ولی در ابزارهای معمول دیده نشود یا فایلی روی دیسک وجود داشته باشد، اما ls آن را نمایش ندهد.

در لینوکس، یکی از روش‌های کلاسیک ساخت روت‌کیت استفاده از ماژول کرنل بود.

ماژول کرنل کدی است که هنگام اجرای سیستم می‌تواند به کرنل اضافه شود. بسیاری از درایورهای لینوکس هم از همین مکانیزم استفاده می‌کنند.

برای مهاجم این روش فوق‌العاده قدرتمند است، ولی یک مشکل دارد: دارید مستقیم با کرنل بازی می‌کنید. یک پوینتر اشتباه، یک باگ کوچک یا ناسازگاری با نسخهٔ کرنل ممکن است سیستم را crash کند و کل عملیات مخفی را با یک پنیک کرنل بسیار غیرمخفی تمام کند!

eBPF در بعضی سناریوها راه جذاب‌تری فراهم می‌کند. مهاجم می‌تواند بدون نوشتن ماژول کرنل سنتی، کد خودش را به نقاط حساسی از کرنل متصل کند.

نقطهٔ شروع همه‌چیز؛ یک Jenkins آسیب‌پذیر

در سال ۲۰۲۵، تیم CSIRT[1] شرکت Synacktiv هنگام بررسی یک زیرساخت میزبانی‌شده روی AWS به زنجیرهٔ نفوذی رسید که در نهایت به بک‌دوری به نام LinkPro ختم شده بود.

نقطهٔ شروع حمله، یک Jenkins آسیب‌پذیر و در معرض اینترنت بود. آسیب‌پذیری مورد استفاده CVE-2024-23897[2] گزارش شد؛ مشاوره‌نامهٔ امنیتی Jenkins[3] آن را با عنوان «خواندن فایل دلخواه از طریق CLI که می‌تواند به اجرای کد از راه دور برسد» معرفی کرده است. Jenkins یکی از ابزارهای شناخته‌شدهٔ CI/CD است؛ یعنی معمولاً کارهایی مثل بیلد، تست و دیپلوی پروژه‌ها را خودکار می‌کند.

به همین دلیل، Jenkins ممکن است به مخزن کد، secretها، اطلاعات حساس، مخزن کانتینر و حتی سرورهای deployment دسترسی داشته باشد. پس اگر مهاجم Jenkins را در اختیار بگیرد، احتمالاً فقط یک web server ساده را نگرفته است؛ ممکن است بخشی از زنجیرهٔ توسعه و استقرار نرم‌افزار را در دست داشته باشد.

از همین نقطه، مهاجم توانست به محیط کوبرنتیز قربانی برسد. بخشی از زیرساخت روی Amazon EKS قرار داشت؛ سرویس AWS برای اجرای کوبرنتیز.

در ادامه یک ایمیج داکری مخرب روی چند کلاستر اجرا شد. قسمت خطرناک ماجرا این بود که Root File System ماشین میزبان با دسترسی خواندن/نوشتن داخل کانتینر روی /mnt ماونت شده بود. یعنی کانتینر عملاً می‌توانست به کل فایل‌سیستم هاست دسترسی داشته باشد. Synacktiv مشاهده کرد که مهاجم از همین دسترسی برای جست‌وجوی توکن‌ها، API Keyها، گواهی‌های امنیتی و اطلاعات حساس موجود روی هاست و پادهای دیگر استفاده کرده است.

چند مرحله بعد، LinkPro روی تعدادی از سیستم‌ها ظاهر شد.

و تازه این‌جا قسمت جذاب ماجرا شروع می‌شود.

Hide؛ اگر ls چیزی نشان ندهد، یعنی فایلی وجود ندارد؟

یکی از بخش‌های LinkPro ماژولی به نام Hide بود. کار آن تقریباً همان چیزی است که از اسمش انتظار داریم: مخفی‌کردن آثار بدافزار.

فرض کنید دستور ls را اجرا می‌کنیم.

در ظاهر، ls فقط محتویات یک دایرکتوری را نشان می‌دهد، اما در پشت صحنه کرنل باید لیست فایل‌ها را در اختیار برنامه قرار دهد. در لینوکس system callهایی از خانوادهٔ getdents و getdents64 برای خواندن directory entryها استفاده می‌شوند.

فرض کنیم نتیجهٔ واقعی این باشد:

```
normal.txt
.tmp~data
config.json
```

حالا اگر روت‌کیت بتواند هنگام برگشت نتیجهٔ getdents دخالت کند، کافی است entry مربوط به خودش را طوری تغییر دهد که برنامه از روی آن عبور کند:

```
normal.txt
config.json
```

فایل linkpro هنوز روی سیستم وجود دارد. کرنل هم آن را می‌شناسد. فقط شما آن را نمی‌بینید.

LinkPro برای چنین کارهایی از هوک‌های eBPF مانند tracepoint و Kretprobe استفاده می‌کند که دربارهٔ آن‌ها می‌توانید مقالهٔ «کالبدشکافی سیستم‌عامل در لحظه»[4] را بخوانید که نیما شیرزادی در شمارهٔ نهم نشریه نوشته است.

tracepoint را می‌توان نقطه‌ای از پیش مشخص‌شده در کرنل دانست که برای مشاهدهٔ رویدادهای مهم در نظر گرفته شده است.

ماژول Hide روی getdents و getdents64 هوک قرار می‌داد و می‌توانست فایل‌هایی با نام‌های مشخص و PIDهای ذخیره‌شده در یک BPF Map را از directory listing حذف کند.

همین ایده برای processها هم جواب می‌دهد.

در لینوکس اطلاعات processها در فایل‌سیستم مجازی /proc هم قابل‌مشاهده‌اند. هر process معمولاً یک دایرکتوری با PID خودش دارد:

```
/proc/1
/proc/432
/proc/912
```

اگر هنگام لیست کردن /proc، entry مربوط به PID بدافزار مخفی شود، process هنوز در حال اجراست، اما ابزارهایی که به این اطلاعات تکیه دارند، ممکن است آن را نبینند.

این‌جا دیگر پرسش «چه چیزی روی سیستم اجرا می‌شود؟» جواب ساده‌ای ندارد.

حتی bpftool هم قرار نبود چیزی ببیند

ممکن است مدیر سیستم شک کند که یک eBPF Rootkit روی سیستم فعال است و

۱. Computer Security Incident Response Team
۲. مطالعه
۳. منبع
۴. مطالعه

سراغ bpftool برود؛ ابزاری برای مشاهده و مدیریت programها، mapها و linkهای BPF. البته LinkPro برای این حالت هم برنامه داشت. ماژول Hide روی sys_bpf هم هوک قرار می‌داد. وقتی ابزاری مثل bpftool می‌خواست ID برنامه‌های BPF را enumerate کند، LinkPro می‌توانست برای آبجکت‌های خودش خطای ENOENT یا همان «وجود ندارد» برگرداند.

یعنی تقریباً:

```
bpftool: برنامهٔ BPF دیگری هست؟
کرنل: بله، ID بعدی ...
LinkPro: نه! چیزی نیست.
bpftool: بسیار خب.
```

اما این‌جا یک پرسش طبیعی پیش می‌آید.

اگر LinkPro تقریباً همه‌چیز را برای مخفی‌شدن در نظر گرفته بود، اصلاً چطور پیدایش کردند؟

روت‌کیت واقعیت را پاک نمی‌کند؛ زاویهٔ دید شما را خراب می‌کند

نکتهٔ اصلی همین است: LinkPro همهٔ واقعیت سیستم را حذف نمی‌کرد؛ فقط بعضی مسیرهایی را که برای دیدن آن واقعیت استفاده می‌کنیم، دست‌کاری می‌کرد.

یک نمونهٔ بسیار خوب netstat و ss است.

ماژول user space مربوط به LinkPro اطلاعات موجود در /proc/net را تغییر می‌داد تا پورت خودش را از دید ابزارهایی مثل netstat مخفی کند. اما ss برای گرفتن اطلاعات سوکت از مسیر دیگری -یعنی Netlink- با کرنل ارتباط برقرار می‌کند. نتیجه چه شد؟ پورت شمارهٔ ۲۲۳۳ با netstat مخفی بود، ولی با ss قابل‌مشاهده بود.

یعنی:

```
/proc/net ← netstat ← اطلاعات دست‌کاری‌شده ← چیزی دیده نمی‌شود
Netlink ← ss ← مسیر متفاوت ← Port دیده می‌شود
```

اشتباه جالب‌تری هم در بخش BPF وجود داشت.

LinkPro تلاش می‌کرد program های eBPF خودش را از bpftool پنهان کند، اما توسعه‌دهندهٔ آن file descriptorهای مربوط به BPF Mapها و BPF Linkها را درست وارد مکانیزم مخفی‌سازی نکرده بود. در نتیجه programها مخفی می‌شدند، اما mapها و linkهای مرتبط هنوز امکان enumerate شدن داشتند و حتی می‌شد PID مرتبط با LinkPro را پیدا کرد. خود Synacktiv صراحتاً این مورد را یک اشتباه در پیاده‌سازی LinkPro می‌داند.

ضمن این‌که تیم Synacktiv از ابتدا هم در حال بررسی یک سیستم «ظاهراً سالم» نبود. آن‌ها در جریان تحقیقات فارنزیک یک زیرساخت از قبل آلوده بودند و زنجیرهٔ حمله را از Jenkins تا ایمیج داکری مخرب و EKS دنبال کرده بودند. در چنین شرایطی اگر ps بگوید همه‌چیز عادی است، تحقیق تمام نمی‌شود.

این شاید یکی از مهم‌ترین درس‌های کل داستان باشد:

«اگر احتمال می‌دهید سیستم به خطر افتاده، یک پرسش را فقط از یک مسیر نپرسید.»

اگر netstat چیزی نشان نمی‌دهد، مسیر دیگری را امتحان کنید. اگر BPF programها دیده نمی‌شوند، map و linkها را بررسی کنید. اگر به کرنل اعتماد ندارید، از telemetry بیرون هاست، شبکه یا زیرساخت ابری هم کمک بگیرید.

تناقض بین چند منبع اطلاعاتی برای یک تحلیل‌گر فارنزیک اتفاق بدی نیست؛ اتفاقاً ممکن است همان چیزی باشد که روت‌کیت را لو می‌دهد.

بخش دیگر LinkPro، Knock نام داشت. این‌جا eBPF دیگر برای مخفی‌کردن فایل یا process استفاده نمی‌شد؛ سراغ شبکه می‌رویم.

در حالت Passive یا Reverse، LinkPro دو برنامهٔ eBPF از نوع XDP و Traffic Control یا TC نصب می‌کرد. هدف این بود که کانال ارتباطی C2 فقط بعد از دریافت یک TCP Packet خاص فعال شود.

XDP یا eXpress Data Path یکی از اولین نقاطی است که packet ورودی در شبکهٔ لینوکس می‌تواند در آن پردازش شود. برنامهٔ XDP تقریباً در سطح درایور و قبل از عبور packet از استک معمول شبکهٔ لینوکس اجرا می‌شود.

برای همین XDP در کارهایی مثل DDoS Filtering بسیار مفید است؛ چون می‌توان packet نامطلوب را قبل از طی‌کردن بخش بزرگی از استک با XDP_DROP حذف کرد.

ولی حالا از زاویهٔ مهاجم به آن نگاه کنیم.

Rootkit می‌تواند همان‌جا بنشیند و packet های ورودی را نگاه کند.

نیازی نیست یک اتصال عجیب همیشه باز باشد یا بک‌دور همواره با C2 Server حرف بزند. برنامه فقط منتظر packet مشخصی می‌ماند؛ چیزی که معمولاً Magic Packet نامیده می‌شود.

تا قبل از رسیدن آن Packet، بک‌دور ساکت است.

مهاجم علامت را می‌فرستد، XDP آن را می‌بیند و مسیر ارتباطی فعال می‌شود.

این مدل برای مهاجم یک مزیت واضح دارد: ابزار دفاعی‌ای که فقط دنبال اتصال دائمی یا پورت فعال بگردد، ممکن است مدت زیادی چیز خاصی نبیند.

نقشهٔ دوم؛ اگر eBPF کار نکرد، چه؟

یکی از نکات جالب LinkPro این بود که همهٔ تخم‌مرغ‌هایش را در سبد eBPF نگذاشته بود.

بخشی از مخفی‌سازی آن به helper ای با نام bpf_override_return وابسته بود که فقط وقتی در دسترس است که کرنل با گزینهٔ CONFIG_BPF_KPROBE_OVERRIDE کامپایل شده باشد.

اگر این قابلیت وجود نداشت، verifier اجازهٔ لود شدن ماژول Hide را نمی‌داد.

در این حالت، LinkPro به پلن B می‌رفت: LD_PRELOAD.

در لینوکس، می‌توان از /etc/ld.so.preload استفاده کرد تا یک لایبرری shared قبل از لایبرری‌های معمول برنامه لود شود.

LinkPro یک لایبرری با نام libld.so را وارد این مسیر می‌کرد و توابعی مثل readdir، getdents، open و بعضی عملیات‌های شبکه را هوک می‌کرد.

مسیر تقریباً این شکلی می‌شد:

```
Application
↓
Malicious Function
↓
Real libc Function
```

تابع مخرب می‌توانست تابع اصلی را اجرا کند، نتیجهٔ واقعی را بگیرد، چند مورد را حذف کند و بعد نسخهٔ دست‌کاری‌شده را به برنامه تحویل دهد.
پس اگر مخفی‌سازی در کرنل جواب نمی‌داد، هنوز یک روش user space وجود داشت؛ البته روشی محدودتر.

پس باید eBPF را خاموش کنیم؟

نه؛ و اتفاقاً همین بخش داستان کمی طعنه‌آمیز است.
بسیاری از ابزارهای امنیتی مدرن خودشان برای شکار بدافزار از eBPF استفاده می‌کنند. مدافع می‌تواند system callها، ساختن process، دسترسی به فایل و ترافیک شبکه را با سربار کم و تقریباً بی‌درنگ ببیند.
اما مهاجم هم همان نقطه را نگاه می‌کند و احتمالاً با خودش می‌گوید: «جای خوبی برای نشستن است!»
این موضوع فقط مخصوص eBPF نیست. از رمزگذاری برای امن کردن ارتباط استفاده می‌کنیم و بدافزار هم ترافیک C2 خودش را رمزگذاری می‌کند. از کانتینر برای isolation استفاده می‌کنیم و یک کانتینر با دسترسی اشتباه ممکن است تبدیل به راهی برای رسیدن به هاست شود.
از eBPF برای دیدن کرنل استفاده می‌کنیم و روت‌کیت هم می‌تواند از همان نقطه برای تغییر چیزی که ما می‌بینیم استفاده کند.
تکنولوژی عوض نشده است؛ فقط کسی که از آن استفاده می‌کند عوض شده.

روت‌کیت را پیدا کردیم؛ حالا پاکش کنیم؟

فرض کنیم بالاخره مطمئن شدیم یک هاست به LinkPro یا روت‌کیت مشابهی آلوده شده است.
اولین واکنش ممکن است این باشد: فایلش را پاک کنیم، process را kill کنیم، eBPF programها را unload کنیم و تمام.
اما وقتی مهاجم تا سطح root یا اختیارات مشابه آن رسیده، دیگر نمی‌توانیم مطمئن باشیم چیزی که پیدا کرده‌ایم، تمام چیزی است که تغییر کرده. شاید کلید SSH دیگری اضافه شده باشد. شاید یک user جدید ساخته شده باشد. شاید service، cron job یا بک‌دور دوم وجود داشته باشد. شاید اطلاعات حساسی که روی این ماشین بوده‌اند دزدیده شده باشند و مهاجم از آن‌ها برای رفتن به سیستم دیگری استفاده کرده باشد.
مشکل اصلی این است:
روت‌کیتی که پیدا کرده‌ایم، الزاماً تنها چیزی نیست که مهاجم روی سیستم گذاشته است.
به همین دلیل در پاسخ جدی به رخداد امنیتی، سیستمی که مهاجم روی آن دسترسی root گرفته یا روت‌کیت نصب کرده، معمولاً قابل‌اعتماد فرض نمی‌شود. راه مطمئن‌تر در بسیاری از موارد این است که اول شواهد لازم برای فارنزیک جمع‌آوری شود، هاست آلوده isolate شود و بعد سیستم از یک ایمیج یا بکاپ شناخته‌شده و سالم دوباره ساخته شود.
راهنمای NIST نیز برای چنین شرایطی بازیابی از بکاپ سالم یا نصب دوباره و کامل سیستم‌عامل و برنامه‌ها را توصیه می‌کند. در محیط ابری این مدل حتی منطقی‌تر است. به‌جای این‌که ساعت‌ها تلاش کنیم یک node مشکوک کوبرنتیز را درمان کنیم، معمولاً می‌توان آن node را کنار گذاشت و یک نسخهٔ سالم از ایمیج مورد اعتماد ساخت.

البته بیلد دوباره پایان کار نیست.
انواع اطلاعات حساس، توکن‌ها، API Keyها و secretهایی که مهاجم ممکن است دیده باشد، باید rotate شوند و سایر سیستم‌های مرتبط هم از نظر حرکت جانبی بررسی شوند.
حتی بکاپ هم به خودی خود قابل‌اعتماد نیست. اگر بکاپ زمانی گرفته شده باشد که سیستم قبلاً آلوده بوده، بازیابی آن ممکن است روت‌کیت یا بک‌دور را دوباره وارد سیستم کند. NIST هم دقیقاً به همین خطر در مورد بکاپ‌های آلوده اشاره می‌کند. بنابراین بعد از چنین نفوذی، پرسش اصلی دیگر این نیست که:
«چطور این روت‌کیت را پاک کنم؟»
پرسش بهتر این است:
«هنوز به کدام قسمت این سیستم می‌توانم اعتماد کنم؟»

یک Jenkins، یک کانتینر و یک روت‌کیت

شاید جذاب‌ترین نکتهٔ حملهٔ LinkPro این باشد که هیچ مرحله‌ای از آن به‌تنهایی شبیه جادو نیست.
حمله با یک Jenkins آسیب‌پذیر شروع شد، به محیط deployment رسید، یک کانتینر با دسترسی بیش از حد اجرا شد، مهاجم به هاست و اطلاعات حساس رسید و در نهایت روت‌کیتی وارد ماجرا شد که از eBPF برای مخفی‌شدن و ایجاد کانال ارتباطی مخفی استفاده می‌کرد.
وقتی این مراحل کنار هم قرار می‌گیرند، نتیجه حمله‌ای می‌شود که شناسایی و پاک‌سازی آن به‌مراتب سخت‌تر است.
و این ما را دوباره به همان سروری برمی‌گرداند که ابتدای مقاله به آن SSH زده بودیم.
ps چیز عجیبی نشان نمی‌دهد، ls هم عادی است و netstat می‌گوید پورت مشکوکی وجود ندارد.
پس سیستم سالم است؟
ممکن است باشد؛ همین‌طور ممکن است نباشد.
LinkPro یک نکتهٔ مهم را یادآوری می‌کند: خروجی ابزارهای سیستم، خود واقعیت نیست؛ فقط تصویری است که از یک مسیر مشخص از واقعیت گرفته‌ایم.
اگر مهاجم بتواند خودش را بین ما و همان مسیر قرار دهد، برای فهمیدن حقیقت باید از چند زاویه نگاه کنیم؛ از user space، کرنل، شبکه، کانتینر ران‌تایم و حتی از بیرون خود هاست. eBPF یکی از جذاب‌ترین قابلیت‌های لینوکس مدرن است و هر روز بیش‌تر در محیط‌های ابری، شبکه و امنیت استفاده می‌شود. قرار نیست از آن بترسیم. اما هر بار که ابزاری دسترسی عمیق‌تری به سیستم به ما می‌دهد، بد نیست یک پرسش را هم از خودمان بپرسیم:
«اگر این ابزار دست طرف مقابل بیفتد، چه؟»



فقط یک GPU، احساس ناکافی بودن

مروری بر روش‌های Multi-GPU در محاسبات مدل‌های یادگیری ماشین



امیرحسین صوری
کارشناسی ۱۴۰۱
دانشگاه صنعتی شریف

۱۲ دقیقه

امروزه می‌بینیم که در دنیای LLMها[1] و LMMها[2]، در میان مدل‌های وزن‌باز، گاهی مدل‌های غول‌پیکری با چند صد میلیارد یا حتی چند تریلیون پارامتر نیز منتشر می‌شوند؛ شاید نتوانیم با یک واحد از منابع محاسباتی موجودمان -مثل GPU-، آن‌ها را لود کرده و fine-tune شان کنیم یا برای inference از آن‌ها استفاده کنیم.

این‌جاست که استفاده از چندین GPU ضروری می‌شود. حتی گاهی اندازهٔ مدل به این بزرگی نیست، اما می‌خواهیم برای تجربهٔ بهتر کاربری، نرخ پردازش درخواست‌ها را افزایش دهیم یا برای batch inference در یک پروژه، سرعت را افزایش دهیم.

در این متن می‌خواهم به برخی از مهم‌ترین روش‌های محاسبات Multi-GPU اشاره کنم. برای درک بهتر تفاوت این روش‌ها، ابتدا باید به این پرسش پاسخ بدهیم: «در روش x، دقیقاً چه چیزی را بین GPUها تقسیم می‌کنیم؟»

متداول‌ترین پاسخ‌هایی که می‌توان داد، این‌ها هستند: خود داده‌ها، مجموعهٔ لایه‌های مدل، تنسورهای هر لایه از مدل و یا بخش‌های مختلف یک دنباله مثل متن (دنباله‌ای از توکن‌ها).

پیش از شروع

در محاسبات Multi-GPU، معمولاً هر GPU توسط یک فرایند[3] مدیریت می‌شود و هر فرایند هم یک شناسه به نام Rank دارد. مثلاً در یک اجرا با چهار GPU، معمولاً Rankهای صفر تا سه داریم.

این فرایندها لازم دارند در طول آموزش و inference، مرتب با یک‌دیگر داده ردوبدل کنند. پنج عملیات ارتباطی مهمی را که در روش‌ها هم خواهیم دید، در ادامه آورده‌ام.

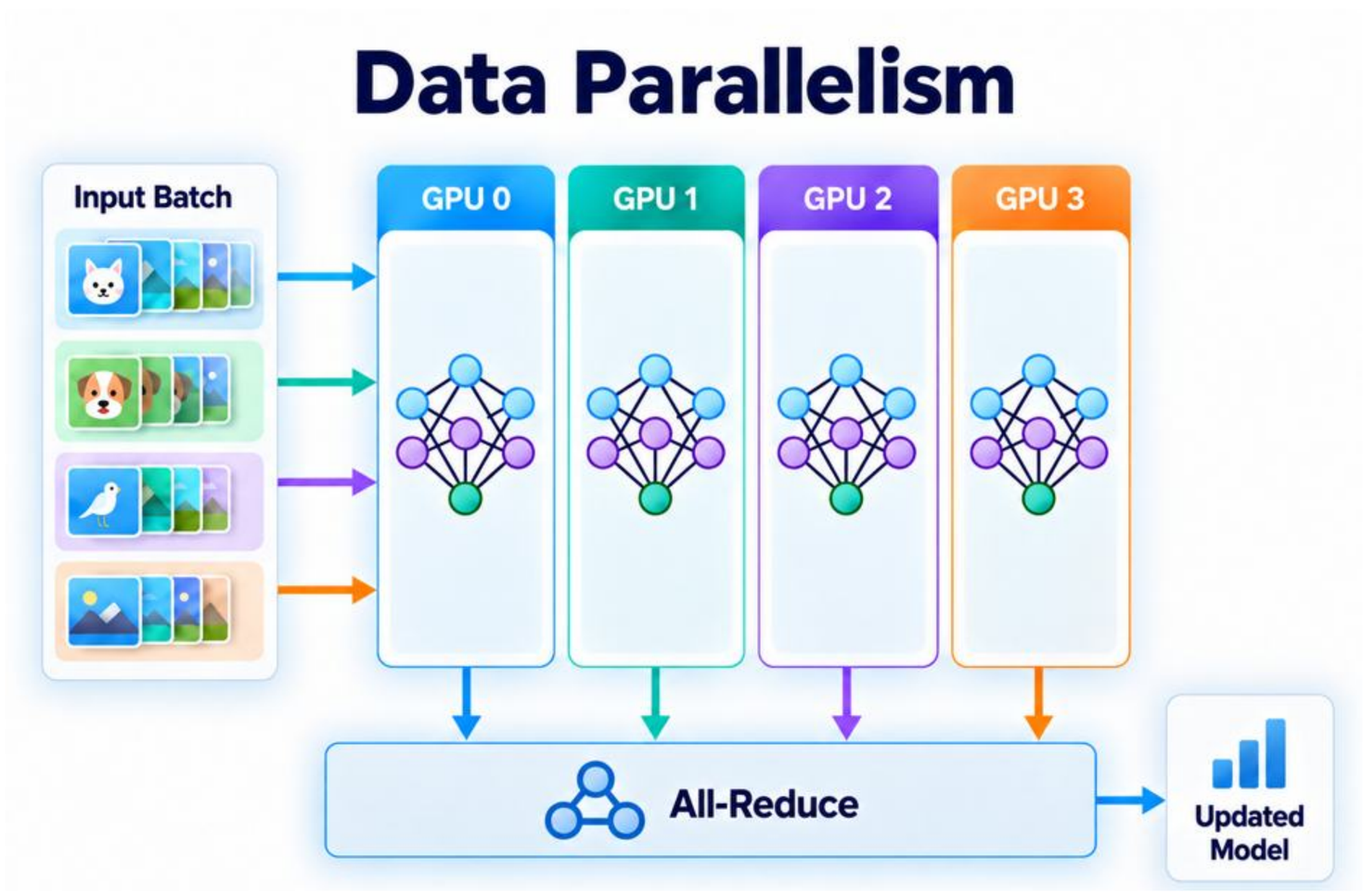
- **All-Reduce:** هر GPU مقدارهایی مثل گرادیان‌های محاسبه‌شده روی داده‌های خودش را دارد؛ این مقادیر با عملیات جمع یا میانگین، ترکیب می‌شوند و نتیجهٔ نهایی دوباره به همهٔ GPUها می‌رسد.
- **All-Gather:** هر GPU فقط بخشی از داده را نگه داشته‌است و با این عملیات، همهٔ بخش‌ها تجمیع می‌شوند تا هر GPU به‌طور موقت یک نسخهٔ کامل از دادهٔ همهٔ GPUها را داشته باشد.
- **Reduce-Scatter:** ابتدا داده‌های GPUها را با هم ترکیب می‌کند و سپس نتیجه را دوباره به بخش‌هایی تقسیم می‌کند و هر بخش را فقط به یک GPU ارسال می‌کند.
- **Point-to-Point:** خروجی هر بخش از یک GPU به یکی از GPUهای دیگر فرستاده می‌شود.
- **All-to-All:** هر GPU دادهٔ خود را به چند بخش تقسیم می‌کند و هر بخش را برای GPU مقصد مربوطه می‌فرستد؛ در پایان نیز، هر GPU مجموعه‌ای از بخش‌ها را از همهٔ GPUهای دیگر دریافت می‌کند.

خالی از لطف نیست بگوییم که این ارتباط فرایندها هزینه دارد و به همین دلیل، نوع اتصال GPUها -مثلاً PCIe در برابر NVLink یا NVSwitch- می‌تواند تأثیر زیادی بر کارایی واقعی هر یک از این روش‌ها داشته باشد.

۱. موازی‌سازی داده (DP[4])

فرض کنید چهار GPU داریم و مدل ما به‌راحتی روی VRAM[5] هر GPU جا می‌شود یا به‌اصطلاح لود می‌شود. در این روش، ما یک نسخهٔ کامل از مدل را روی هر GPU قرار می‌دهیم و batchهای دادهٔ خود را میان آن‌ها تقسیم می‌کنیم تا عملیات forward به‌طور جداگانه روی آن‌ها اجرا شود.

اگر در حال آموزش یا fine-tune مدل باشیم، مرحلهٔ backward هر کدام نیز جداگانه انجام می‌شود تا گرادیان‌ها مشخص شوند و سپس با استفاده از All-Reduce، گرادیان‌ها با یک‌دیگر تجمیع می‌شوند و حاصل به هر GPU داده می‌شود تا نسخه‌های مدل با گرادیان یکسان به‌روز شوند و عملاً با یک نسخهٔ واحد، امکان ادامهٔ آموزش فراهم می‌شود.



خب، کاربرد این روش چیست؟ درست حدس زدید، این روش مشکل کمبود حافظه را حل نمی‌کند، یعنی اگر کل مدل روی هر GPU جا نشود، کاری از دست این روش ساخته نیست؛ اما می‌تواند توان پردازشی بیش‌تری در اختیارمان بگذارد تا نمونه‌های بیش‌تری را در واحد زمان پردازش کنیم.

۱. Large Language Models
۲. Large Multimodal Models
۳. Process
۴. Data Parallelism
۵. Video Random Access Memory

۲. بهبود موازی‌سازی داده

یکی از مشکل‌های روش قبلی این است که همه‌چیز مدل -پارامترها، گرادیان‌ها و وضعیت‌های بهینه‌ساز[۱]- روی تمام GPUها تکرار می‌شود و این می‌تواند موجب هدررفت بخش زیادی از حافظه شود.

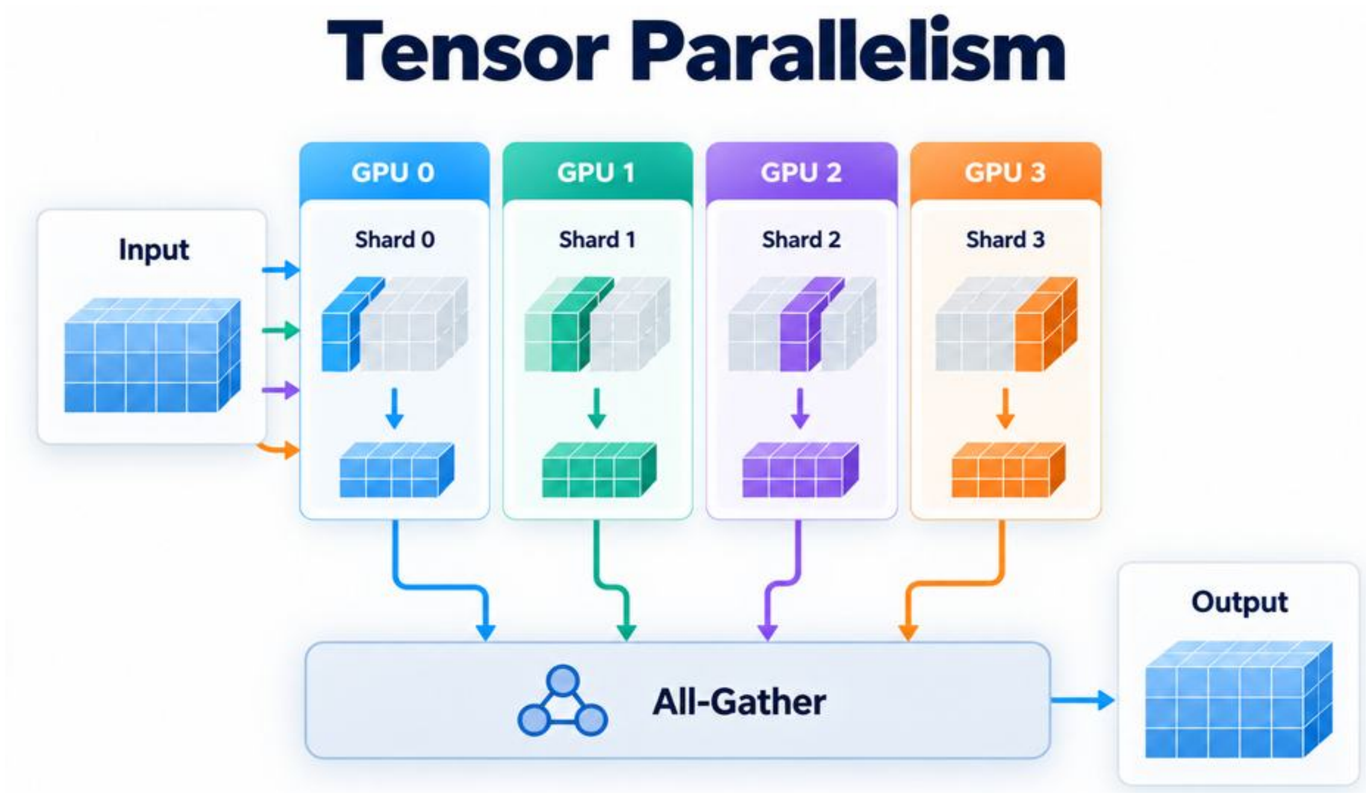
یک دسته از راهکارهای موجود برای این چالش، روش‌هایی مثل ZeRO[۲] (راهکار لایبرری DeepSpeed) و FSDP[۳] (راهکار خود PyTorch) هستند. ایدهٔ کلی‌شان این است که به‌جای ذخیرهٔ کامل موارد بالا روی هر GPU، آن‌ها را اصطلاحاً بین GPUها shard یا تقسیم کنیم و هر GPU فقط بخشی از آن‌ها را نگه دارد؛ هنگام اجرای هر بخش از مدل، پارامترهای لازم به‌طور موقت با All-Gather بازسازی می‌شوند و پس از انجام محاسبات، پارامترها با Reduce-Scatter دوباره میان GPUها تقسیم می‌شوند.

با این وجود، این روش‌ها همچنان نوعی موازی‌سازی داده محسوب می‌شوند؛ چون هر GPU بخش متفاوتی از batch را پردازش می‌کند و از نظر منطقی همان مدل را روی داده‌های خودش اجرا می‌کند. تقسیم‌بندی فقط نحوهٔ نگه‌داری و تبادل وضعیت مدل را تغییر می‌دهد.

در مورد ZeRO، سه حالت کلی قابل‌استفاده هستند: ZeRO-1 فقط وضعیت بهینه‌ساز را تقسیم می‌کند، ZeRO-2 گرادیان‌ها را هم تقسیم می‌کند و ZeRO-3 علاوه‌بر آن‌ها پارامترها را هم تقسیم می‌کند. FSDP نیز چند حالت مختلف دارد که در حالت fully sharded، شباهت زیادی به ZeRO-3 دارد.

۳. موازی‌سازی تنسور (TP[۴])

فرض کنید در یک مدل با معماری ترنسفورمر، یک ماتریس وزن بسیار بزرگ داریم و محاسبه‌ای به‌صورت Y=XW وجود دارد. ایدهٔ این روش این است که خود ماتریس وزن W را بین چند GPU تقسیم کنیم. مثلاً هر GPU بخشی از ستون‌ها یا سطرهای ماتریس را نگه می‌دارد و فقط قسمت مربوط به خودش از ضرب ماتریسی را انجام می‌دهد؛ سپس خروجی‌های حاصل با عملیات ارتباطی مناسب کنار هم قرار می‌گیرند یا تجمیع می‌شوند تا Y نهایی را تشکیل دهند.



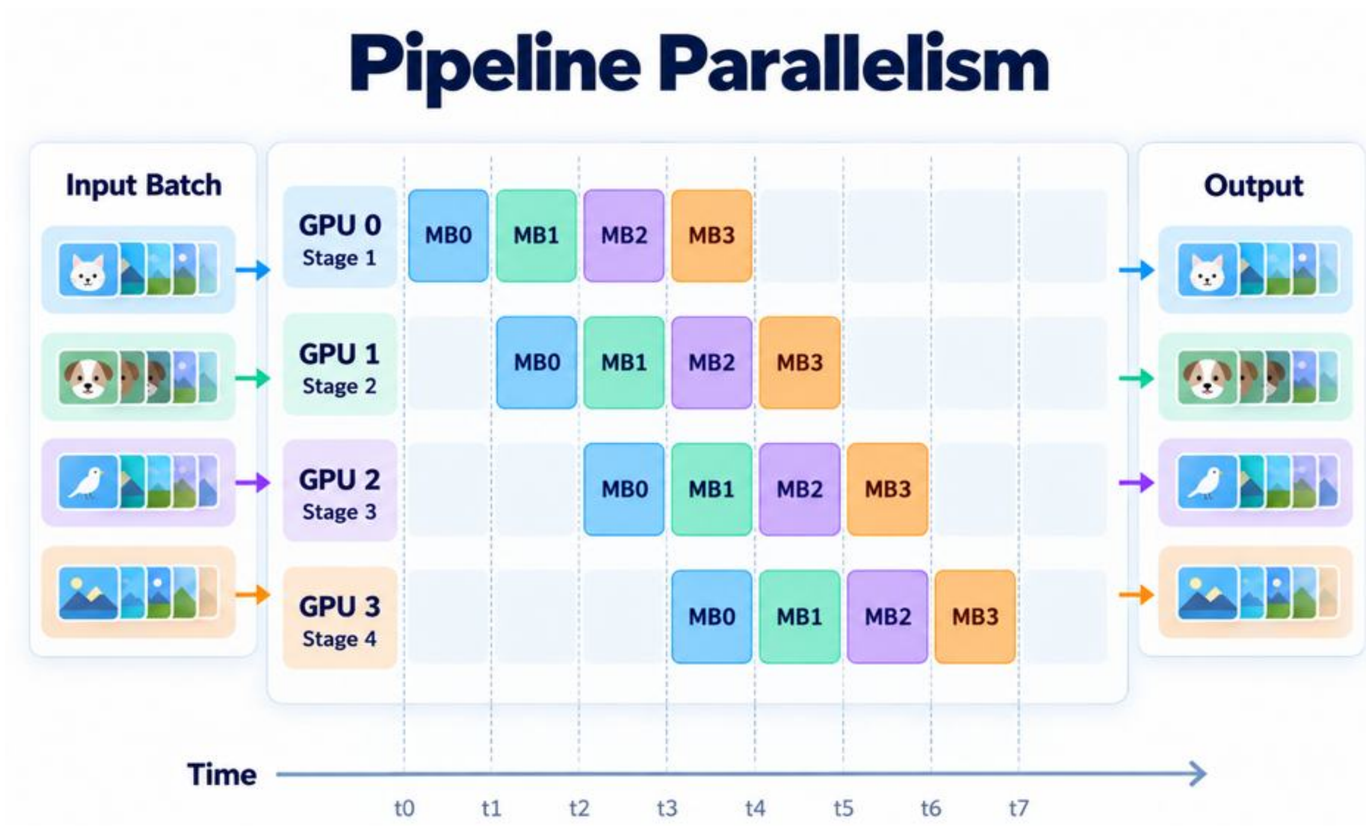
در مدل‌های ترنسفورمری، پروجکشن‌های کوئری، کلید و مقدار[۵] در مکانیزم توجه[۶] و همچنین لایه‌های MLP[۷] می‌توانند میان چند GPU تقسیم شوند.

همان‌طور که گفته شد، موازی‌سازی داده به‌تنهایی برای مدل‌هایی که روی یک GPU جا نمی‌شوند، کارساز نبود. اما مزیت اصلی موازی‌سازی تنسور این است که می‌توان مدلی با لایه‌های بسیار بزرگ را روی چند GPU پخش کرد. البته هزینهٔ این قابلیت، ارتباط زیاد بین GPUهاست؛ چون تقریباً در هر لایه نیاز به تبادل داده وجود دارد. به همین دلیل، این روش روی GPUهایی که اتصال سریع مثل NVLink دارند، بسیار بهتر عمل می‌کند.

۴. موازی‌سازی پایپ‌لاین (PP[۸])

شیوه‌ای دیگر از محاسبات Multi-GPU که می‌تواند برای مدل‌های عظیم مفید واقع شود، موازی‌سازی پایپ‌لاین است. در این روش، به‌جای شکستن ماتریس وزن هر لایه، مجموعهٔ لایه‌ها را بین GPUها تقسیم می‌کنیم. مثلاً در یک مدل ترنسفورمری چهل‌لایه‌ای، GPU اول می‌تواند لایه‌های صفر تا ۹ را اجرا کند، GPU دوم لایه‌های ۱۰ تا ۱۹ و همین‌طور تا آخر. خروجی هر بخش معمولاً با یک ارتباط Point-to-Point به GPU مسئول بخش بعدی فرستاده می‌شود؛ در مرحلهٔ backward هم گرادیان‌ها در جهت عکس بین بخش‌ها منتقل می‌شوند.

چالش این روش این است که وقتی می‌خواهیم یک batch را یک‌جا از این پایپ‌لاین عبور دهیم، داده‌ها باید به ترتیب مشخص‌شده از همهٔ GPUها عبور کنند و طبیعتاً وقتی داده‌ها در حال پردازش در یکی از GPUها هستند، GPUهای دیگر بی‌کارند. برای حل این مشکل می‌توان یک batch را به چند micro-batch تقسیم کرد و به‌ترتیب، به مدل ورودی داد؛ حالا وقتی GPU دوم مشغول پردازش micro-batch اول است، GPU اول می‌تواند کار روی micro-batch دوم را شروع کند و بی‌کار نماند؛ درست مثل ایدهٔ پایپ‌لاین دستورات در معماری کامپیوتر!



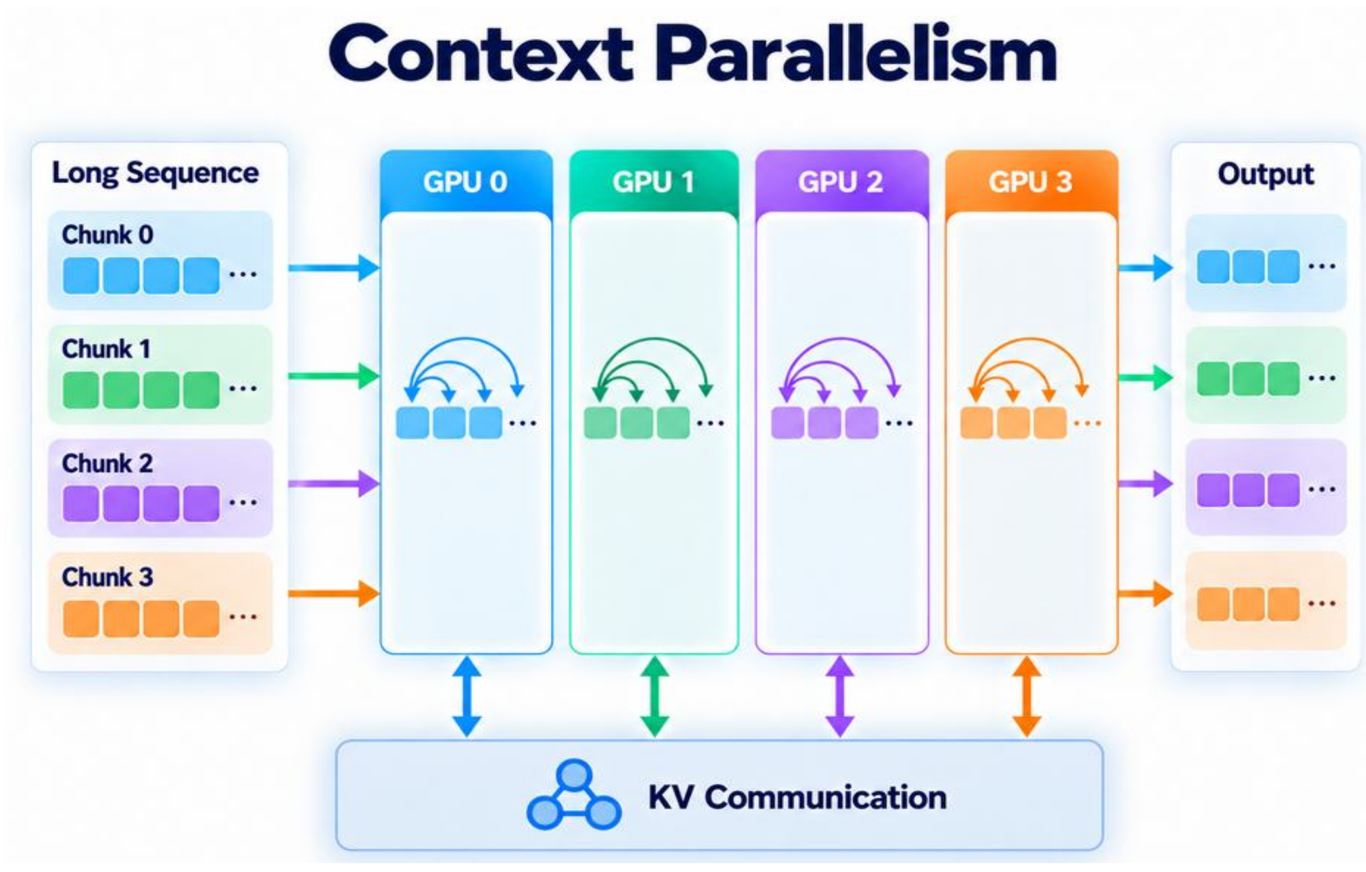
۱. Optimizer
۲. Zero Redundancy Optimizer
۳. Fully Shared Data Parallel
۴. Tensor Parallelism
۵. Query/Key/Value
۶. Attention
۷. Multi-Layer Perceptron
۸. Pipeline Parallelism

۵. موازی‌سازی دنباله[1] و کانتکست[2]

گاهی مشکل اصلی اندازهٔ مدل نیست، بلکه طول ورودی است. در مدل‌های ترنسفورمری هرچه دنباله (مثلاً متن) طولانی‌تر شود، حافظهٔ مورد نیاز برای فعال‌سازی‌ها[3] افزایش می‌یابد و هزینهٔ محاسبات توجه نیز به‌شدت رشد می‌کند.

در موازی‌سازی دنباله، بعضی محاسباتی که روی هر توکن تقریباً مستقل انجام می‌شوند -مثل LayerNorm و Dropout- در امتداد بُعد دنباله تقسیم می‌شوند. مثلاً اگر ۸۰۰۰ توکن و چهار GPU داشته باشیم، هر GPU می‌تواند فعال‌سازی مربوط به حدود ۲۰۰۰ توکن را نگه دارد و همان بخش را پردازش کند. هدف اصلی این روش، کاهش حافظهٔ فعال‌سازی است؛ در این حالت، خودِ توجه، لزوماً به‌طور کامل بین GPUها تقسیم نشده است.

در موازی‌سازی کانتکست، تقسیم عمیق‌تر است: خود توکن‌های کانتکست و معمولاً کلیدها و مقادیر توجه بین GPUها پخش می‌شوند. در همان مثال بند قبل، هر GPU دو هزار توکن را در اختیار دارد، اما کوئری‌های آن باید بتوانند به توکن‌های سه GPU دیگر هم توجه کنند. بنابراین GPUها باید اطلاعات کلید و مقدار را با یک‌دیگر مبادله کنند. اگر علاقه‌مندید، می‌توانید روش‌هایی مثل Ring Attention و DeepSpeed-Ulysses را برای توزیع توجه روی دنباله‌های بسیار بلند مطالعه کنید. البته دقت کنید که نام‌گذاری SP و CP در همهٔ منابع کاملاً یکسان نیست.

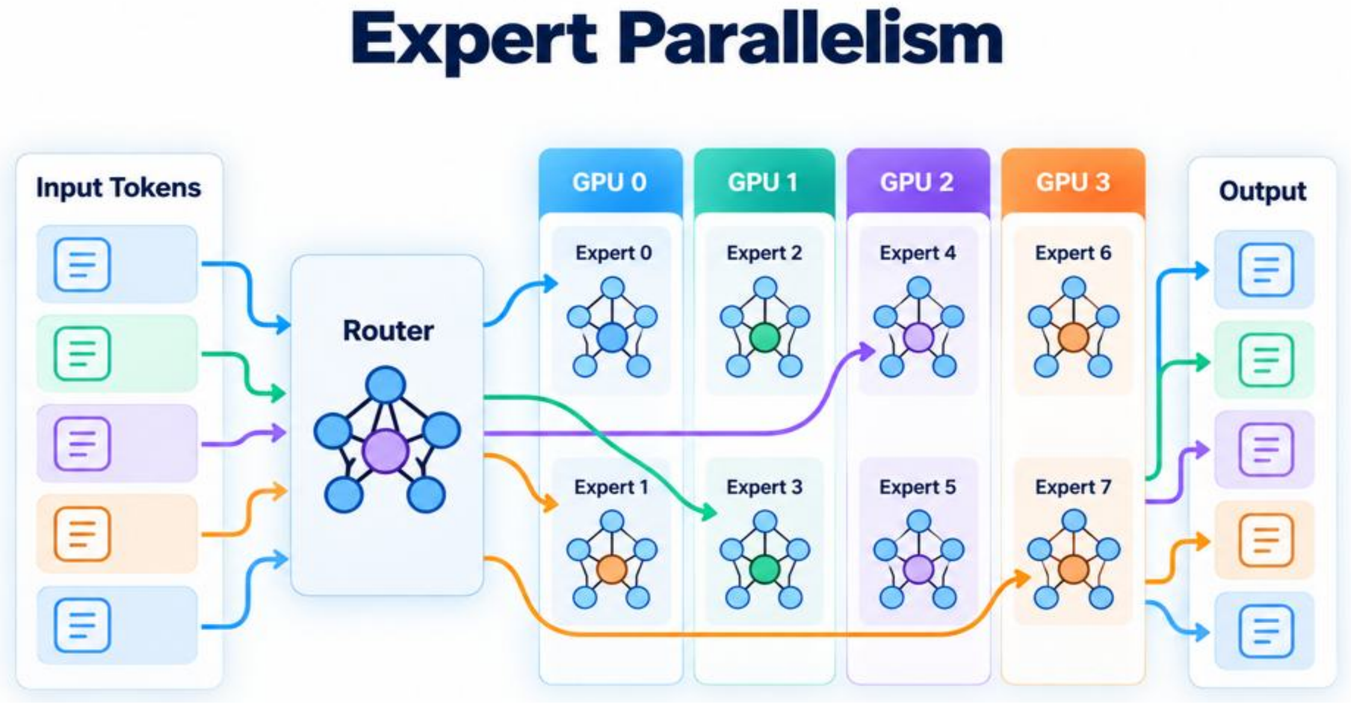


به‌طور خلاصه، SP بیشتر برای کم‌کردن حافظهٔ فعال‌سازی است، در حالی که CP خودِ توجه روی کانتکست‌های بسیار بلند را بین چند GPU توزیع می‌کند.

۶. موازی‌سازی متخصص (EP[4])

ابتدا بیایید نگاهی به تعریف مدل‌های MoE[5] بیندازیم. در معماری MoE، به‌جای این‌که همهٔ ورودی‌ها از یک مسیر محاسباتی یکسان عبور کنند، چندین شبکهٔ کوچک‌تر یا «متخصص» داریم و یک روتر مشخص می‌کند هر توکن توسط کدام متخصص یا متخصص‌ها پردازش شود. به این ترتیب، مدل می‌تواند تعداد پارامتر بسیار زیادی داشته باشد، بدون این‌که برای هر توکن لازم باشد همهٔ آن‌ها فعال باشند. این معماری اجازه می‌دهد متخصص‌ها را بین GPUهای مختلف تقسیم کنیم.

فرض کنید ۸ متخصص و چهار GPU داریم؛ هر GPU دو متخصص را نگه می‌دارد. روتر برای هر توکن امتیاز متخصص‌ها را محاسبه می‌کند. مثلاً ممکن است توکن اول به متخصص‌های ۲ و ۱۰ و توکن دوم به متخصص‌های ۵ و ۶ فرستاده شوند. چون این متخصص‌ها ممکن است روی GPUهای دیگری باشند، توکن‌ها با عملیاتی مثل All-to-All به GPU مالک متخصص فرستاده می‌شوند. آن GPU محاسبهٔ متخصص را انجام می‌دهد و خروجی دوباره به محل اصلی توکن بازگردانده می‌شود.



چالش مهم این روش توازن بار[6] است. اگر روتر تعداد زیادی توکن را به یک متخصص خاص بفرستد، GPU آن تبدیل به گلوگاه می‌شود، در حالی که GPUهای دیگر ممکن است ظرفیت خالی داشته باشند.

پیشنهاد سرآشپز: ترکیب روش‌ها

در مدل‌های بسیار بزرگ معمولاً یک نوع موازی‌سازی به‌تنهایی کافی نیست و چند روش روی ابعاد مختلف با هم ترکیب می‌شوند. مثلاً اگر تنظیمات [DP=2, TP=4, PP=3] را داشته باشیم، در مجموع به ۲×۴×۳=۲۴ تا GPU نیاز داریم. مدل ابتدا از نظر عمق به سه بخش برای پایپ‌لاین تقسیم می‌شود؛ داخل هر بخش، محاسبات هر لایه بین چهار GPU با TP پخش می‌شود؛ سپس کل این ساختار یک‌بار دیگر تکرار می‌شود تا دو رپلیکای موازی برای داده‌ها داشته باشیم که batchهای متفاوتی را پردازش می‌کنند.

در واقع، یک GPU می‌تواند هم‌زمان عضو چند گروه ارتباطی باشد: مثلاً با سه GPU دیگر برای TP همکاری کند، با بخش‌های مجاورش در پایپ‌لاین داده ردوبدل کند و در عین حال با رپلیکای متناظر خودش در گروه DPها، گرادیان‌ها را هماهنگ کند.

حتی در مدل‌های بزرگ‌تر، CP و EP هم می‌توانند به این ترکیب اضافه شوند. بنابراین طراحی یک سیستم Multi-GPU فقط انتخاب

۱. Sequence Parallelism
۲. Context Parallelism
۳. Activations
۴. Expert Parallelism
۵. Mixture of Experts
۶. Load Balancing

یک روش نیست؛ مسئلهٔ اصلی این است که GPUها را چگونه میان این ابعاد تقسیم کنیم و کدام ارتباط‌ها را روی سریع‌ترین لینک‌های سخت‌افزاری قرار دهیم.

تا این‌جا چند روش Multi-GPU را دیدیم؛ حالا می‌خواهم دو روش برای افزایش کارایی inference معرفی کنم که در کنار این روش‌ها می‌آیند و امکان استفادهٔ مؤثرتر از ظرفیت GPUها را برایمان فراهم می‌کنند.

بهینه‌سازی inference با PagedAttention

مدل هنگام تولید توکن‌ها، کلیدها و مقادیر، توجه مربوط به توکن‌های قبلی را در KV Cache نگه می‌دارد تا مجبور نباشد در هر مرحله دوباره آن‌ها را محاسبه کند. مشکل این است که با افزایش تعداد درخواست‌ها و طول متن، این cache می‌تواند بخش بزرگی از VRAM را اشغال کند.

مشکل دیگر این است که در روش‌های معمول، برای هر درخواست باید فضای حافظهٔ نسبتاً بزرگ و پیوسته‌ای در نظر گرفته شود، در حالی که از قبل دقیقاً نمی‌دانیم هر درخواست در نهایت به چه مقدار حافظه نیاز خواهد داشت؛ در نتیجه بخشی از حافظه ممکن است بی‌استفاده بماند یا تکه‌تکه شود.

PagedAttention به‌جای تخصیص یک فضای بزرگ و پیوسته برای KV Cache هر درخواست، آن را به بلوک‌های کوچک‌تر تقسیم می‌کند که می‌توانند در نقاط غیرپیوستهٔ حافظه قرار بگیرند؛ ایده‌ای شبیه Paging در حافظهٔ مجازی.

این کار استفاده از حافظه را بهینه می‌کند و اجازه می‌دهد درخواست‌های بیش‌تری هم‌زمان در حافظه نگه داشته شوند.

این روش که بیش‌تر به‌واسطهٔ لایبرری vLLM شناخته‌شده است، جایگزین روش‌هایی مثل TP یا DP نیست، بلکه در کنار آن‌ها استفاده می‌شود.

مثلاً ممکن است یک مدل با موازی‌سازی تنسور روی چند GPU اجرا شود و در عین حال، هر بخش از سیستم KV Cache مربوط به خودش را به‌صورت بلوکی مدیریت کند. vLLM نیز علاوه‌بر این مدیریت حافظه، از روش‌های مختلف Multi-GPU پشتیبانی می‌کند.

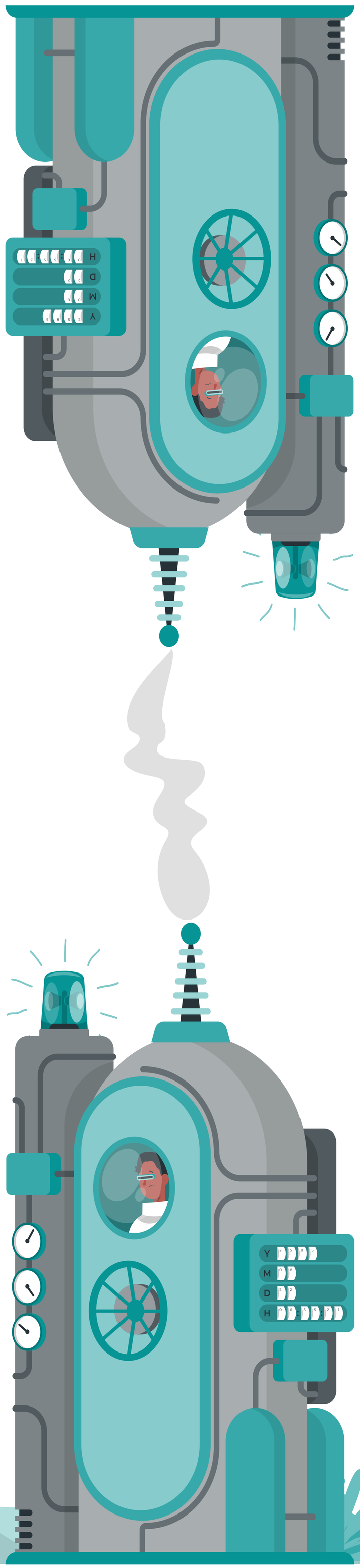
بهینه‌سازی inference با Continuous Batching

یکی دیگر از مشکلات سرویس‌دهی LLMها این است که درخواست‌ها طول یکسانی ندارند. اگر چند درخواست را در یک batch ثابت قرار دهیم، ممکن است یک درخواست بعد از ۲۰ توکن تمام شود، در حالی که درخواست دیگری هنوز صدها توکن برای تولید داشته باشد. اگر تا پایان کل batch صبر کنیم، بخشی از ظرفیت GPU عملاً هدر می‌رود.

در این روش، batch ثابت نیست؛ درخواست‌های تمام‌شده از batch خارج می‌شوند و درخواست‌های جدید می‌توانند در حین اجرا جای آن‌ها را بگیرند. در نتیجه GPU به‌جای انتظار برای تمام‌شدن طولانی‌ترین درخواست، همواره مجموعه‌ای از درخواست‌های فعال را پردازش می‌کند. هدف اصلی این روش افزایش بهره‌برداری و نرخ پردازش در سرویس‌دهی است و vLLM آن را در کنار بقیهٔ روش‌های بهینه‌سازی inference به کار می‌گیرد.

این روش هم اساساً Multi-GPU نیست. می‌توان آن را روی یک GPU نیز استفاده کرد، اما در یک سیستم Multi-GPU هم مکمل روش‌های موازی‌سازی است؛ مثلاً چند رپلیکای مدل می‌توانند هرکدام مجموعه‌ای از درخواست‌ها را با این روش پردازش کنند، یا یک مدل با TP روی چند GPU، درخواست‌های موجود در batch پویا را به‌صورت مشترک اجرا کند.

در نهایت، در روش‌های Multi-GPU پرسش اصلی این است که کدام بخش از داده، مدل یا محاسبه را میان GPUها تقسیم کنیم و در عین حال، هزینهٔ ارتباط بین آن‌ها را تا حد ممکن پایین نگه داریم. اما باید توجه کرد که استفادهٔ بهتر از GPUها فقط به موازی‌سازی محدود نمی‌شود؛ مدیریت بهتر حافظه و زمان‌بندی محاسبات هم نقش مهمی در استفادهٔ مؤثرتر از منابع دارند.



امنیت وب در عصر پساکوانتوم



علی الماسی
کارشناسی ۱۴۰۳
دانشگاه صنعتی شریف

۱۴ دقیقه | ۸

فرض کنید برای اتصال به اینترنت از یک VPN استفاده می‌کنید؛ در این حالت، تمامی ترافیک ورودی و خروجی دستگاه شما از سرور ارائه‌دهندهٔ آن عبور می‌کند. حال در نظر بگیرید که می‌خواهید وارد حساب کاربری خود در یک وب‌سایت شوید.

از آن‌جا که مدیر این سرور واسط به تک‌تک بسته‌های ارسالی شما دسترسی دارد، دو پرسش مطرح می‌شود.

- چگونه مطمئن می‌شوید که نام کاربری و رمز عبور شما در میانهٔ مسیر شنود و سرقت نمی‌شود؟
- چه تضمینی وجود دارد که سرور VPN با جعل هویت آن وب‌سایت، نسخه‌ای ساختگی از صفحه را به شما تحویل ندهد؟

برای حل این دو مشکل به دو چیز نیاز داریم و هیچ‌کدام بدون دیگری کافی نیستند.

اولی پروتکل TLS[1] است که ارتباط میان مرورگر و سرور مقصد را رمزگذاری می‌کند تا کسی -از جمله مدیر سرور VPN- در میانهٔ مسیر، نتواند محتوای آن را بخواند یا تغییر دهد.

اما رمزگذاری به‌تنهایی کافی نیست؛ اگر مرورگر نداند طرف مقابلش کیست، ممکن است ارتباط رمزگذاری‌شده‌اش را با خود VPN برقرار کند. این‌جا دومی لازم می‌شود: «زیرساخت کلید عمومی وب» یا Web PKI[2]؛ مجموعه‌ای از سازمان‌ها، قواعد و پروتکل‌ها که مشخص می‌کنند کدام کلید عمومی متعلق به کدام دامنه است. مرورگر با اتکا به آن درمی‌یابد کلیدی که سرور ارائه کرده است، واقعاً متعلق به همان دامنه‌ای است که به‌دنبالش بوده است.

رمزنگاری

TLS در دو مرحله کار می‌کند. اول، مرورگر و سرور با رمزنگاری نامتقارن به یک کلید مشترک می‌رسند و هویت سرور بررسی می‌شود؛ بعد با همان کلید و رمزنگاری متقارن، تمام داده‌ها را ردوبدل می‌کنند. از بخش متقارن شروع می‌کنیم تا ببینیم چرا نمی‌توان فقط به آن اکتفا کرد.

رمزنگاری نامتقارن

در رمزنگاری نامتقارن، نه یک کلید مشترک، بلکه یک زوج‌کلید استفاده می‌شود. کلید عمومی را می‌توان بی‌خطر در اختیار همه گذاشت و کلید خصوصی باید نزد صاحبش بماند، چون از روی کلید عمومی عملاً نمی‌توان به کلید خصوصی رسید. کلید خصوصی کاربرد دیگری هم دارد؛ می‌توان با آن پیامی را «امضاء» کرد و هرکس کلید عمومی را داشته باشد، می‌تواند بررسی کند که امضا واقعاً با همان کلید خصوصی ساخته شده است. همین دو ویژگی کارهایی را ممکن می‌سازد که با رمزنگاری متقارن به‌تنهایی شدنی نیست؛ برای نمونه، دو طرفی که هیچ اطلاعات محرمانه‌ای از پیش ندارند، می‌توانند روی شبکه‌ای عمومی به یک کلید مشترک برسند و یک طرف می‌تواند هویت خود را اثبات کند.

در TLS، این سازوکار دو کار جدا انجام می‌دهد.

کار اول، توافق بر سر کلید مشترک است. مرورگر و سرور هرکدام یک زوج‌کلید موقت[4] می‌سازند. «موقت» یعنی این زوج‌کلید فقط برای همین یک ارتباط تولید می‌شود و بعد از پایانش دور ریخته می‌شود؛ برعکس کلید بلندمدت سرور که ماه‌ها و سال‌ها ثابت می‌ماند و در کار دوم مورد استفاده است. هر کدام، بخش عمومی زوج‌کلید خودش را برای طرف مقابل می‌فرستد و با ترکیب کلید خصوصی موقت خودش و کلید عمومی طرف مقابل به یک مقدار مشترک می‌رسد. کلید نشست از همین مقدار مشترک ساخته می‌شود. شنودکننده همهٔ مقدارهای عمومی را می‌بیند، ولی حساب کردن مقدار مشترک از روی آن‌ها از نظر محاسباتی بسیار دشوار است.

کار دوم، احراز هویت است. توافق کلید به‌تنهایی کافی نیست. مهاجمی که وسط مسیر قرار بگیرد، می‌تواند با هر طرف جداگانه توافق کلید انجام دهد و خود را طرف دیگر معرفی کند. پس مرورگر باید مطمئن شود طرف مقابل واقعاً همان سروری است که می‌خواهد. سرور پیام‌های ردوبدل‌شده تا آن لحظه -از جمله کلید عمومی موقت خودش- را با کلید خصوصی بلندمدتش امضاء می‌کند. مهاجم کلید خصوصی سرور را ندارد، پس نمی‌تواند این امضاء را بسازد؛ و اگر کلید موقت را در مسیر عوض کرده باشد، امضاء با آن‌چه مرورگر دریافت کرده، نمی‌خواند.

مرورگر امضاء را با کلید عمومی سرور بررسی می‌کند. این کلید در یک گواهی دیجیتال است که سرور برای مرورگر می‌فرستد؛ گواهی سندی است که می‌گوید «این کلید عمومی به این دامنه تعلق دارد». آن را یک مرجع صدور

رمزنگاری متقارن

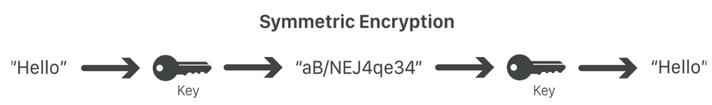
در رمزنگاری متقارن یک کلید محرمانهٔ مشترک هم برای رمزگذاری به کار می‌رود و هم برای رمزگشایی. بزرگ‌ترین مزیت این روش سرعت است: پردازنده‌های امروزی دستورهای سخت‌افزاری مخصوصی مثل AES-NI دارند که رمزگذاری حجم زیادی از داده را با سربار کم ممکن می‌کند.

در TLS، برای هر نشست یک کلید تازه ساخته می‌شود و پس از پایان نشست کنار گذاشته می‌شود. اما این کلید باید پیش از شروع تبادل داده نزد هر دو طرف باشد و همین‌جا مشکل شروع می‌شود: مرورگر و سرور تا آن لحظه هیچ مقدار محرمانهٔ مشترکی[3] ندارند و شبکه‌ای که از آن استفاده می‌کنند، عمومی است و هر کسی می‌تواند به آن گوش دهد.

برای حل این مسئله دو راه ساده وجود دارد، ولی هیچ‌کدام امن نیستند. فرستادن مستقیم کلید فایده‌ای ندارد، چون شنودکننده هم آن را می‌بیند و می‌تواند کل ترافیک نشست را رمزگشایی کند. رمزکردن کلید با کلیدی دیگر هم مشکل را حل نمی‌کند، چون آن کلید دوم نیز باید به شکلی امن به طرف مقابل برسد و مسئله فقط یک قدم عقب می‌رود.

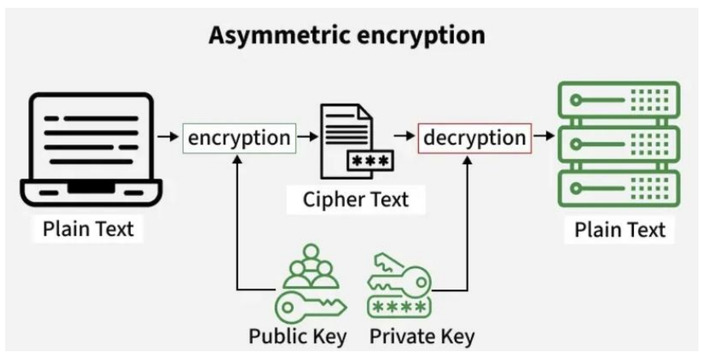
راه‌حل این مشکل در رمزنگاری نامتقارن است. با روشی به نام توافق کلید، دو طرف می‌توانند روی شبکهٔ عمومی به یک کلید مشترک برسند، بی‌آن‌که خود کلید را بفرستند.

رمزنگاری نامتقارن برای رمزکردن حجم زیاد داده بسیار کند است؛ برای همین فقط برای ساختن همین کلید به کار می‌رود و بعد از آن همهٔ داده‌ها با رمزنگاری متقارن ردوبدل می‌شوند.



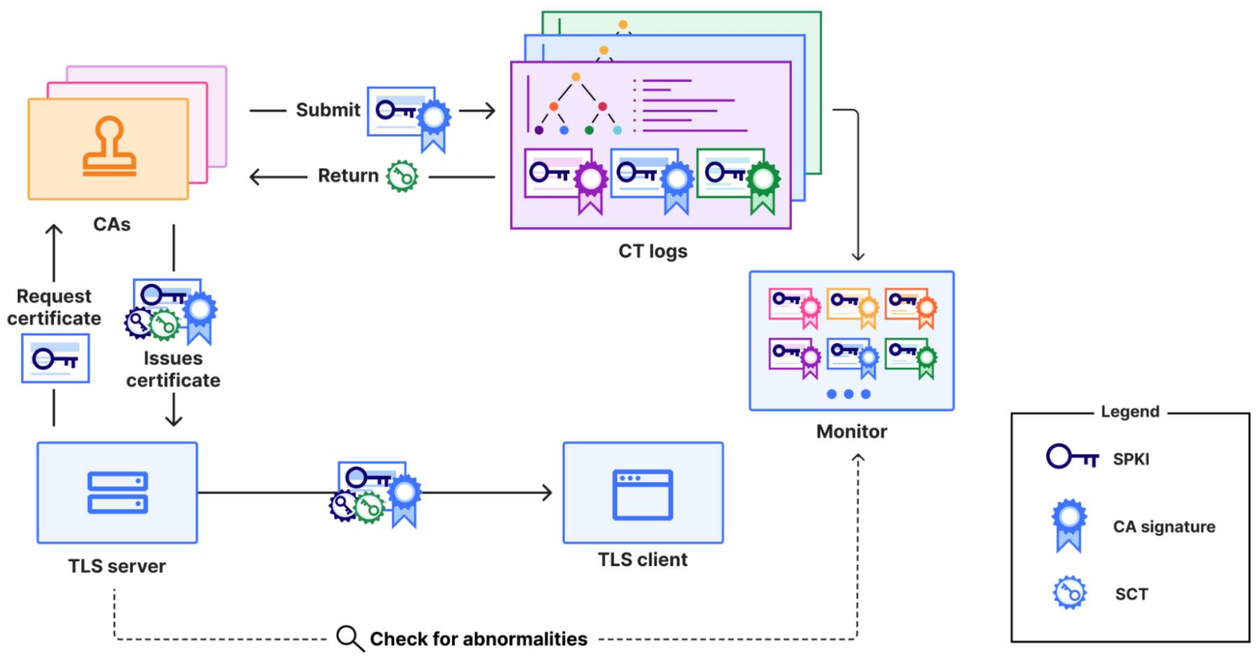
۱. Transport Layer Security
۲. Web Public Key Infrastructure
۳. Shared Secret
۴. Ephemeral

گواهی یا CA[1] معتبر امضاء کرده است و مرورگر همچنین بررسی می‌کند که گواهی منقضی نشده و برای همین دامنه صادر شده باشد. پس از پایان فرایند دست‌تکانی[2]، کلید موقت حذف می‌شود و تمام داده‌های نشست با کلید متقارنی که از shared secret مشتق شده است، رمزگذاری می‌شوند.



زنجیرهٔ اعتماد و مراجع صدور گواهی

مرورگر چطور تشخیص می‌دهد که گواهی ارائه‌شده از سمت سرور جعلی نیست؟ مرورگرها و سیستم‌عامل‌ها همراه خود فهرستی از مراجع صدور گواهی معتمد دارند که در «مخزن ریشه» ذخیره شده است. مرورگر به هر گواهی‌ای اعتماد می‌کند که زنجیرهٔ امضاءهایش به یکی از این ریشه‌ها برسد. مرجع ریشه معمولاً به‌طور مستقیم برای وب‌سایت‌ها گواهی صادر نمی‌کند. لو رفتن کلید خصوصی ریشه کل اعتماد را از بین می‌برد، به همین دلیل آن را آفلاین و دور از دسترس نگه می‌دارند. ریشه فقط گواهی یک مرجع میانی را امضاء می‌کند و آن مرجع میانی گواهی دامنهٔ مقصد را. نتیجه، یک زنجیرهٔ گواهی است. سرور در دست‌تکانی گواهی خودش و گواهی‌های میانی را می‌فرستد، ولی ریشه را نمی‌فرستد؛ چون مرورگر از قبل آن را دارد. مرورگر زنجیره را از پایین به بالا بررسی می‌کند: امضای هر گواهی را با کلید عمومی گواهی بالاتر تأیید می‌کند، تاریخ اعتبار را می‌سنجد و مطمئن می‌شود زنجیره به یکی از ریشه‌های مخزن ختم می‌شود. با این حال، اعتماد به CA به خودی خود یک ریسک است. اگر یک CA مورد نفوذ قرار گیرد یا دچار خطا شود، می‌تواند بی‌سروصدا برای هر دامنه‌ای گواهی معتبر صادر کند، بی‌آن‌که مالک دامنه متوجه شود. برای حل این مشکل، سازوکار شفافیت گواهی یا CT[3] به وب اضافه شد. هر گواهی باید در چند فهرست عمومی به‌عنوان لاگ ثبت شود. هر کس می‌تواند این لاگ‌ها را بخواند و مثلاً ببیند برای دامنه‌اش چه گواهی‌هایی صادر شده است. لاگ‌ها فقط افزودنی‌اند: فقط می‌توان ورودی تازه به آن‌ها اضافه کرد و ساختارشان طوری است که تغییر یا حذف ورودی‌های قبلی قابل‌تشخیص است. صدور گواهی برای sharif.edu را در نظر بگیرید. CA نمی‌تواند ابتدا گواهی را صادر کند و سپس آن را ثبت کند، چرا که مرورگر باید از ثبت شدن آن اطمینان داشته باشد. به همین دلیل، ابتدا نسخه‌ای از گواهیِ بدون تأییدیه را با نام «پیش‌گواهی» برای چند سرور ثبت وقایع (CT Log) می‌فرستد. هر لاگ آن را ثبت می‌کند و یک برچسب زمانی امضاءشدهٔ گواهی یا SCT[4] بازمی‌گرداند؛ سندی که تضمین می‌دهد این گواهی در لاگ درج خواهد شد. در نهایت، CA این SCTها را درون گواهی نهایی قرار می‌دهد و آن را به مالک دامنه تحویل می‌دهد. هنگام اتصال، مرورگر علاوه‌بر زنجیرهٔ سنتی اعتبارسنجی، SCTها را هم بررسی می‌کند؛ به این صورت که باید از لاگ‌هایی باشند که خودش می‌شناسد و تعدادشان نیز کافی باشد. مرورگرهایی مثل کروم و سافاری گواهی‌ای را که SCT معتبر کافی نداشته باشد، رد می‌کنند. یعنی عملاً هر گواهی‌ای که این مرورگرها در وب عمومی بپذیرند، پیش‌تر در لاگ‌های عمومی ثبت شده است. همین ثبت اجباری است که CT را از نظر امنیتی مهم می‌کند. فرض کنید کنترل یک مرجع صدور به دست مهاجم بیفتد و او برای sharif.edu گواهی جعلی صادر کند. برای این‌که این گواهی در این مرورگرها کار کند، ناچار است آن را در لاگ‌های عمومی ثبت کند و ثبت شدنش یعنی قابل‌کشف‌شدنش: مالک دامنه و سرویس‌های پایش می‌توانند آن را ببینند و پیگیری کنند. افزون بر این، ثبت عمومی، مدرکی علیه آن CA می‌سازد که نمی‌تواند انکارش کند و می‌تواند به سلب اعتماد مرورگرها از آن منجر شود.



تهدید کوانتومی برای رمزنگاری نامتقارن

امنیت همهٔ سازوکارهای نامتقارنی که تا این‌جا دیدیم روی دو مسئلهٔ ریاضی بنا شده‌اند. RSA به سختی تجزیهٔ عددهای بزرگ به عوامل اول تکیه دارد و رمزنگاری خم بیضوی (که در توافق کلید ECDHE و امضای ECDSA به کار می‌رود) به‌سختی «مسئلهٔ لگاریتم گسسته». برای کامپیوترهای کلاسیک، هیچ الگوریتم کارآمدی برای حل این دو شناخته نشده و شکستن‌شان زمان و منابعی نجومی می‌طلبد. کامپیوتر کوانتومی این وضع را عوض می‌کند. الگوریتم شور[5] هر دو مسئله را در زمانی معقول حل می‌کند؛ یعنی از روی کلید عمومی می‌توان کلید خصوصی را محاسبه کرد. البته شور فقط روی کامپیوتر کوانتومی مقاوم در برابر خطا با تعداد کافی کیوبیت منطقی اجرا می‌شود (کیوبیت منطقی از ترکیب تعداد زیادی کیوبیت فیزیکی پرخطا ساخته می‌شود). شکسته شدن این دو مسئلهٔ ریاضی، هر دو پایه‌ای را که در ابتدا برای امنیت برشمردیم متزلزل می‌کند. نخست، محرمانگی ارتباط، یعنی همان پرسش اول: مهاجم نیازی ندارد رمزگشایی را در همان لحظه انجام دهد؛ کافی است ترافیک رمزگذاری‌شدهٔ شما را امروز ذخیره کند. سال‌ها بعد، با در دست داشتن یک کامپیوتر کوانتومی می‌تواند کلید موقت هر نشست را از روی همان مقدارهای عمومی ردوبدل‌شده بازسازی کند و تمام ترافیک گذشته را بخواند. در این‌جا موقت‌بودن کلیدها کمکی نمی‌کند، چون تمام داده‌های لازم برای محاسبهٔ دوبارهٔ آن‌ها از همان ابتدا آشکارا روی شبکه مخابره شده است. دوم، احراز هویت، یعنی همان پرسش دوم: اگر مهاجم بتواند از روی کلید عمومی یک CA به کلید خصوصی آن برسد، می‌تواند برای هر دامنه‌ای گواهی‌ای بسازد که مرورگر آن را معتبر بداند. در این وضعیت، Web PKI و زنجیرهٔ اعتمادی که

۱. Certificate Authority
۲. Handshake
۳. Certificate Transparency
۴. Signed Certificate Timestamp
۵. Shor's Algorithm

برای شناسایی سرور ساخته بودیم بی‌معنا می‌شود و دیگر هیچ تضمینی نیست که طرف مقابل، نسخه‌ای ساختگی از وب‌سایت را پیش روی شما نگذاشته باشد.

بحران مقیاس‌پذیری

دو الگوریتم پساکوانتومی پس از سال‌ها بررسی عمومی، استاندارد شده‌اند؛ ML-KEM برای توافق کلید و ML-DSA برای امضاء. تا کنون هیچ الگوریتم کوانتومی شناخته‌شده‌ای برای شکستن آن‌ها ارائه نشده است. توافق کلید همین حالا در حال رفتن به ML-KEM است، چون خطر محرمانگی فوری است. Cloudflare می‌گوید بخش بزرگی از ترافیکش همین حالا از ترکیب الگوریتم قدیمی و جدید استفاده می‌کند. پس مشکل اصلی که می‌ماند احراز هویت است و علتش اندازهٔ امضاءهاست. طول امضاء ML-DSA-44 برابر با ۲۴۲۰ بایت و امضاء ECDSA فقط ۶۴ بایت است؛ یعنی افزایشی نزدیک به ۳۸ برابر. کلید عمومی نیز از ۶۴ بایت به ۱۳۱۲ بایت می‌رسد. در هر اتصال، چندین امضاء و کلید عمومی مبادله می‌شود (امضاء سرور، امضاءهای زنجیره و SCTها). اگر همگی با ML-DSA جایگزین شوند، حجم آن‌ها از حدود ۴ کیلوبایت به حدود ۱۵ کیلوبایت می‌رسد. این موضوع از نظر عملکرد شبکه تعیین‌کننده است؛ اگر حجم از سقف پنجرهٔ اولیهٔ ارسال در TCP (حدود ۱۴٫۶ کیلوبایت) فراتر برود، یک زمان رفت‌وبرگشت یا RTT[1] اضافه به اتصال تحمیل می‌شود و صفحه دیرتر باز خواهد شد. افزون بر این، لاگ‌های CT نیز موظف به ذخیرهٔ این امضاها هستند و طبق برآوردهای Cloudflare، حجم ذخیره‌سازی آن‌ها حدود ۴۰ برابر خواهد شد.

گواهی‌های درخت مرکل یا MTC[2]

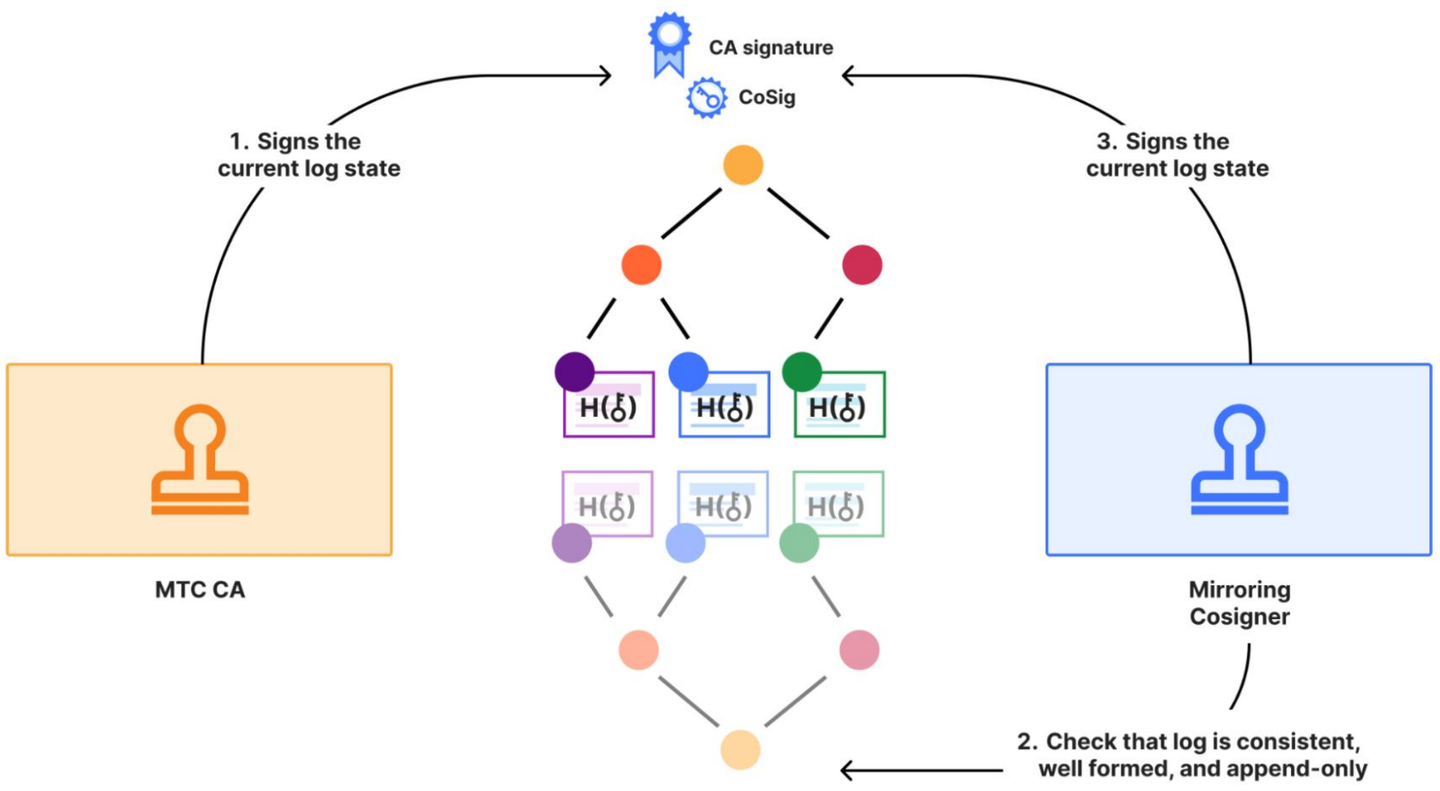
در معماری فعلی، صدور و ثبت دو کار جداگانه‌اند و هر کدام امضاء خودش را می‌خواهد. CA گواهی را امضاء می‌کند، لاگ‌ها برای آن SCT صادر می‌کنند و همین گواهی در دو شکل (پیش‌گواهی و گواهی نهایی) در چند لاگ تکرار می‌شود. دو هزینه‌ای که در بخش قبل دیدیم، یعنی امضاءهای سنگین در دست‌تکانی و حجم لاگ‌ها، از همین تکرار می‌آید.

ایدهٔ «گواهی‌های درخت مرکل» این تفکیک را از میان برمی‌دارد و رویکرد صدور از طریق ثبت در لاگ را دنبال می‌کند؛ به‌گونه‌ای که گواهی تنها زمانی صادرشده تلقی می‌شود که در لاگ به ثبت رسیده باشد. در این الگو، ثبت دیگر مرحله‌ای مجزا پس از صدور نیست، بلکه شفافیت به بخشی ذاتی از خود سازوکار صدور تبدیل می‌شود. در عمل، CA لاگ خودش را نگه می‌دارد و این لاگ یک درخت مرکل فقط‌افزودنی است. وقتی درخواست صدور می‌رسد (معمولاً از طریق پروتکل ACME)، مرجع صدور گواهی کنترل متقاضی بر دامنه را راستی‌آزمایی می‌کند و ورودی جدید را -شامل نام دامنه و کلید عمومی- به انتهای لاگ اضافه می‌کند. بعد تک‌تک گواهی‌ها را امضاء نمی‌کند و فقط وضعیت کل درخت را در قالب یک checkpoint یا tree head امضاء می‌کند. این امضاء تأیید می‌کند که CA همهٔ ورودی‌های درخت تا آن لحظه را صادر کرده است. گواهی دیگر امضاء مستقلی ندارد و در عوض، اثبات این‌که در درخت هست همراهش می‌آید.

درخت مرکل یک درخت هش است. هر ورودی یک برگ است، هر گره والد هش پیوند دو فرزندش است و ریشه، چکیدهٔ رمزنگاری‌شدهٔ کل درخت است. برای اثبات این‌که یک ورودی در درخت هست، لازم نیست کل درخت فرستاده شود. کافی است هش گره برادر هر گره در مسیر برگ تا ریشه را بفرستیم، که به آن اثبات عضویت می‌گویند. گیرنده با ترکیب پله‌به‌پلهٔ این هش‌ها ریشه را بازسازی می‌کند و اگر با ریشهٔ امضاءشده یکی بود، عضویت ثابت می‌شود. در درختی با ۸ برگ، اثبات عضویت فقط ۳ هش دارد. اندازهٔ اثبات با لگاریتم تعداد برگ‌ها رشد می‌کند، پس حتی برای یک میلیون گواهی حدود ۲۰ هش کافی است؛ یعنی با هش ۳۲ بایتی، اثبات حدود ۶۴۰ بایت و کمتر از ۱ کیلوبایت می‌شود. به همین دلیل، یک امضاء می‌تواند تعداد بسیار زیادی گواهی را پوشش دهد.

لاگ صدور CA، مرجع انحصاری تمامی گواهی‌های آن CA به‌شمار می‌رود؛ بنابراین، هر گواهی تنها یک‌بار ذخیره می‌شود و پایش‌گرها نیز تنها کافی است همین یک لاگ را بررسی کنند. علاوه‌بر این، ورودی‌ها امضاء مجزا ندارند و تنها یک امضاء بر فراز رأس درخت، کل لاگ را پوشش می‌دهد؛ لاگ هم نه خود کلیدها، بلکه صرفاً هش کلیدهای عمومی را نگه می‌دارد. این موارد دقیقاً همان عواملی هستند که در سازوکار فعلی CT، با ورود امضاءهای پساکوانتومی حجم ذخیره‌سازی را تا ۴۰ برابر افزایش می‌دادند.

اگر فقط خود CA لاگ را امضا کند، هنوز می‌تواند به طرف‌های مختلف، نسخه‌های متفاوتی از لاگ نشان دهد. برای همین، وضعیت لاگ به یک co-signer مستقل هم فرستاده می‌شود. این co-signer یک نسخه از لاگ را نگه می‌دارد، بررسی می‌کند که هر وضعیت جدید با وضعیت قبلی سازگار و فقط‌افزودنی باشد و آن را هم امضاء می‌کند. پیش‌نویس سیاست کروم دست‌کم دو co-signature می‌خواهد: یکی از یک Mirroring Co-signer با سازمانی غیر از CA و یکی از خود CA. نتیجه این است که گواهی‌ها حتی اگر لاگ خود CA از دسترس خارج شود، برای پایش در دسترس می‌مانند.



۱. Round-Trip Time
۲. Merkle Tree Certificates
۳. Signed Certificate Timestamp

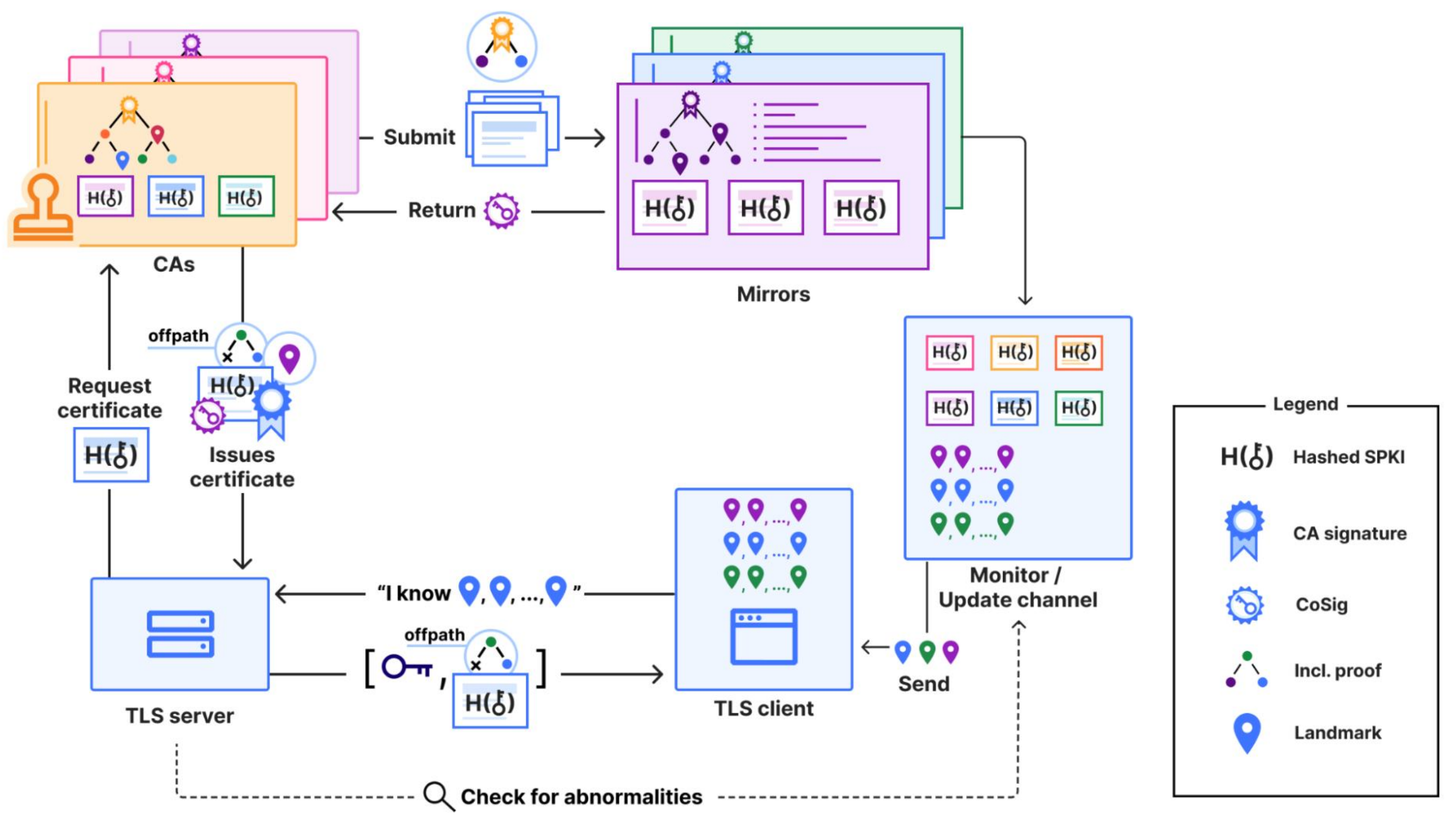
چالش دوم، مسئلهٔ پذیرش و هم‌گرایی در اکوسیستم وب است. MTC در حال حاضر در سطح پیش‌نویس اینترنتی IETF قرار دارد و هنوز به استاندارد رسمی (RFC) تبدیل نشده است. شرکت گوگل اعلام کرده برنامه‌ای برای افزودن گواهی‌های سنتی مبتنی‌بر الگوریتم‌های پساکوانتومی به مخزن ریشهٔ فعلی کروم ندارد و مخزن ریشهٔ مجزایی را برای MTC پیاده‌سازی می‌کند؛ ضوابط نهایی ورود مراجع صدور به این مخزن برای سه‌ماههٔ سوم ۲۰۲۷ زمان‌بندی شده است. Cloudflare نیز متقاضی ورود به این سازوکار است، هرچند تا کنون صدور عملیاتی را آغاز نکرده است. در نهایت، هم‌سویی شرکت‌های اپل، مایکروسافت و موزیلا تعیین خواهد کرد که آیا MTC به استانداردی فراگیر در پهنهٔ اینترنت بدل خواهد شد یا تنها در حد بهینه‌سازی اختصاصی در اکوسیستم کروم باقی خواهد ماند.

حالت لندمارک[۱]: حذف امضاء CA از شبکه

گواهی درخت مرکل دو حالت دارد.

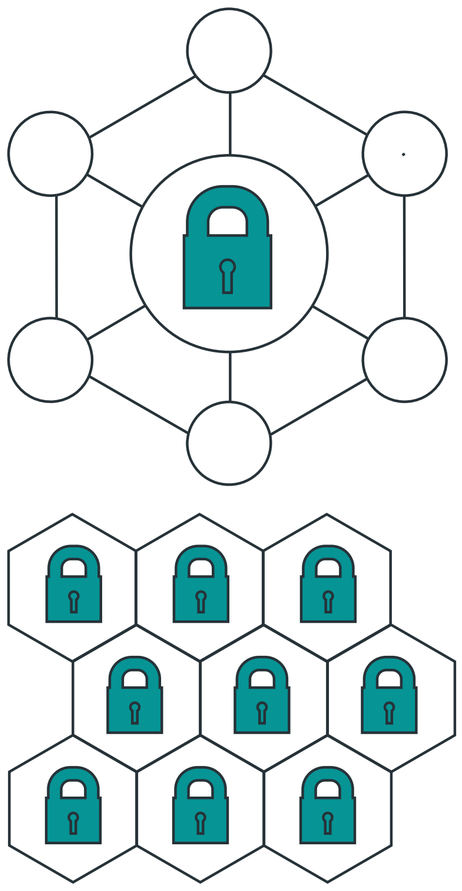
- **حالت مستقل:** سرور اثبات عضویت را همراه با امضاء پساکوانتومی ریشهٔ درخت ارسال می‌کند. این حالت مختص کلاینت‌هایی است که داده‌های لندمارک به‌روز در اختیار ندارند.
- **حالت لندمارک:** به‌صورت دوره‌ای یک ریشه به‌عنوان «لندمارک» (نشانهٔ مرجع) انتخاب می‌شود و هش آن از طریق به‌روزرسانی‌های مرورگر به کلاینت‌ها می‌رسد. هنگام اتصال، سرور فقط گواهی و اثبات عضویت را می‌فرستد. مرورگر ریشه را از روی اثبات بازسازی می‌کند و با هش لندمارکی که از پیش دارد، تطبیق می‌دهد. بدین ترتیب، خود گواهی هیچ امضایی ندارد و امضاء CA و تأییدیه‌های CT از شبکه حذف می‌شوند. فقط امضاء دست‌تکانی سرور می‌ماند، چون اثبات عضویت نشان می‌دهد گواهی در درخت هست، نه این‌که فرستنده کلید خصوصی را دارد. با ML-DSA-44، این یعنی حدود ۳۷۰۰ بایت (کلید عمومی سرور و همین امضاء)، که تقریباً هم‌اندازهٔ زنجیرهٔ کلاسیک امروز است.

در آزمایش مشترک Cloudflare و کروم، میانهٔ زمان دست‌تکانی با حالت لندمارک حدود ۹ درصد کم‌تر از زنجیره‌های کلاسیک بود.



چالش‌ها و چشم‌انداز پیاده‌سازی

اثربخشی حالت لندمارک کاملاً به دسترسی کلاینت به داده‌های به‌روز وابسته است. ابزارهای خط فرمان نظیر curl، اسکریپت‌های خودکار و سامانه‌های اینترنت اشیاء، فاقد چرخهٔ به‌روزرسانی پیوسته، مشابه مرورگرها هستند؛ از این رو، سرورها همچنان باید برای کلاینت‌هایی که داده‌های لندمارک به‌روز ندارند یا از MTC پشتیبانی نمی‌کنند، گواهی کلاسیک پشتیبان نگه دارند.



منابع

- Building a post-quantum certificate authority with Merkle Tree Certificates
- How does SSL work? | SSL certificates and TLS
- GeeksForGeeks: Asymmetric Encryption
- Keeping the Internet fast and secure: introducing Merkle Tree Certificates

۱. Landmark

از LXC و داکر تا Firecracker

بررسی سیر تکامل کانتینرها و ایزوله‌سازی


امیرحسین محمدزاده
کارشناسی ۱۴۰۲
دانشگاه صنعتی شریف

۱ | ۲۱ دقیقه

اگر تا به حال با ابزارهای کانتینری مانند داکر و کوبرنتیز کار کرده‌اید، ولی دقیقاً نمی‌دانید که چه اتفاقی در لایه‌های زیرین آن‌ها می‌افتد تا یک کانتینر را مدیریت کنند، شاید خواندن این متن برای شما خالی از لطف نباشد.

شاید نام‌هایی مانند runc، conmon، dockerd، shim، containerd، podman و اصطلاحاتی مثل کانتینرهای Rootless به گوش‌تان خورده باشد، اما مرز دقیق بین آن‌ها و نقش هرکدام در این زیست‌بوم برایتان شفاف نباشد.

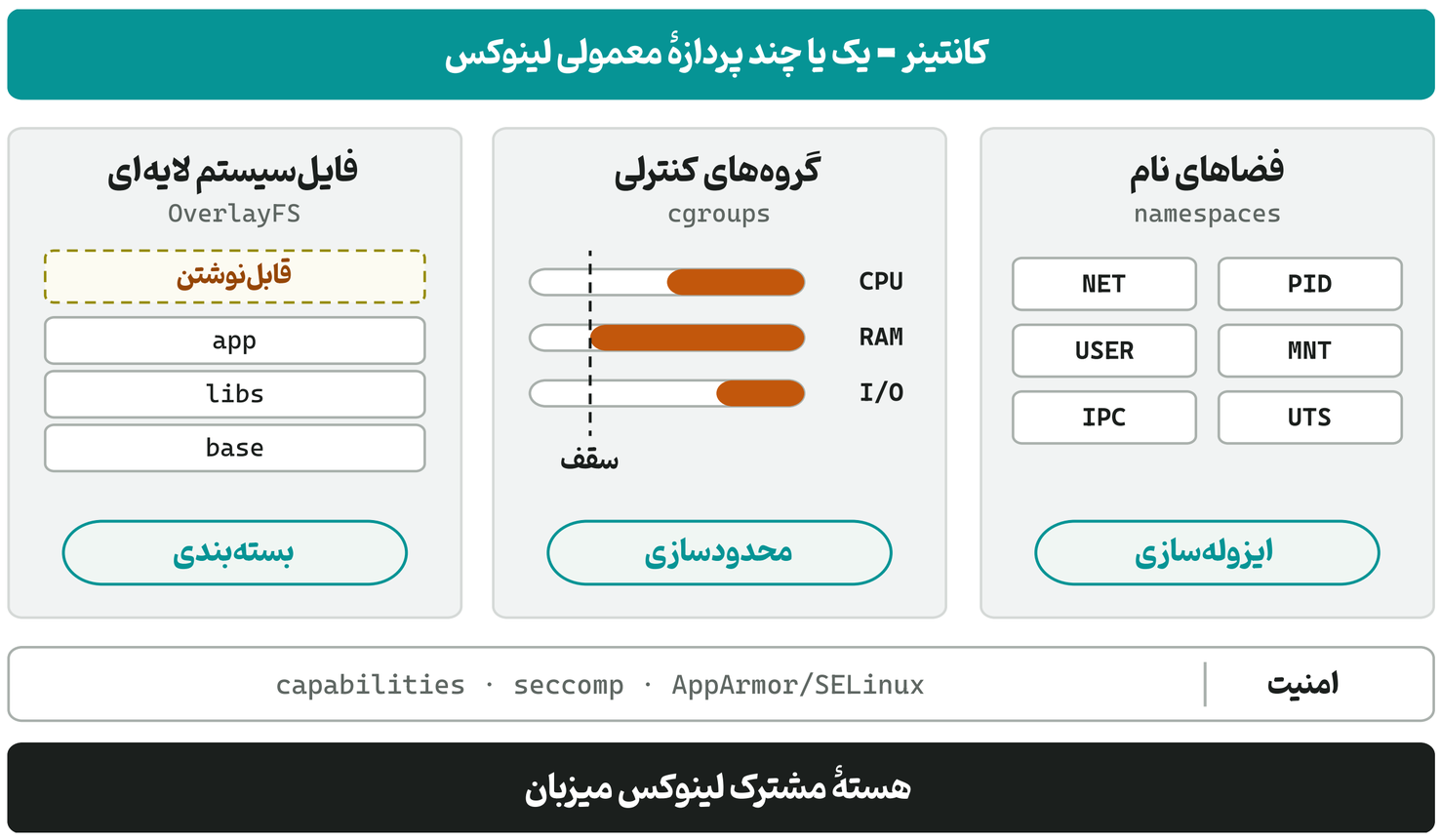
در این متن نمی‌خواهیم صرفاً یک مستند خشک از معرفی ابزارها ارائه دهیم؛ بلکه می‌خواهیم سیر تکامل مهندسی کانتینرها را دنبال کنیم. خواهیم دید که چطور داکر از یک برنامهٔ یکپارچه[1]، تکه‌تکه شد؛ چرا ورود کوبرنتیز و استاندارد CRI[2] معادلات را تغییر داد و ماجرای حذف dockershim چه بود. در ادامه بررسی می‌کنیم که چرا معماری سنتی داکر به‌دلیل نیاز به دسترسی سطح بالا، امروزه یک پاشنهٔ آشیل امنیتی محسوب می‌شود و چطور ابزارهای نسل جدید و Rootless این مشکل را حل کرده‌اند. در نهایت، به زمان حال و عصر عامل‌های هوش مصنوعی[3] می‌رسیم؛ جایی که حتی ایزوله‌سازی هستهٔ لینوکس هم کم می‌آورد و پای MicroVMها به میان کشیده می‌شود.

کانتینر چیست و داکر دقیقاً چه چیزی را اختراع کرد؟

با یک جملهٔ غافل‌گیرکننده شروع می‌کنیم:

چیزی به اسم «کانتینر» در سطح هستهٔ لینوکس وجود خارجی ندارد!

کانتینر سنتی در واقع از یک یا چند پردازهٔ معمولی لینوکسی تشکیل شده و بر سه ستون استوار است: **ایزوله‌سازی، محدودسازی و بسته‌بندی** که در ادامه به آن‌ها می‌رسیم؛



همه‌چیز از سال ۱۹۷۹ و فراخوانی سیستمی سادهٔ chroot در یونیکس شروع شد که اجازه می‌داد پوشهٔ ریشه (/) را برای یک پردازه تغییر دهیم تا نتواند بقیهٔ فایل‌های سیستم را ببیند. این ایده بعدها در دههٔ ۲۰۰۰ میلادی، در هستهٔ لینوکس به تکامل رسید.

- **ستون اول، فضاهای نام[4]:** این مکانیزم دید پردازه را نسبت به محیط اطرافش ایزوله می‌کند؛ به طوری‌که پردازه فکر می‌کند شبکه، فایل‌سیستم، لیست پردازه‌ها (PID) و کاربران مخصوص به خودش را دارد.
- **ستون دوم، گروه‌های کنترلی[5]:** در سال ۲۰۰۶، مهندسان گوگل -که روی زیرساخت داخلی خودشان یعنی Borg، پدربزرگ کوبرنتیز، کار می‌کردند- متوجه شدند که ایزوله کردن دید پردازه کافی نیست؛ اگر یک پردازه کل CPU یا حافظهٔ میزبان را ببلعد، چه؟ این‌جا cgroups متولد شد تا مصرف منابع سخت‌افزاری هر پردازه را محدود کند.

در کنار این دو، لایهٔ امنیتی شامل capabilities، seccomp و AppArmor/SELinux هم مشخص می‌کند پردازه مجاز به انجام چه کارهایی است؛ بعدتر دوباره به seccomp برمی‌گردیم.

در سال ۲۰۰۸، یعنی ۵ سال قبل از تولد داکر، پروژهٔ Linux Containers (به اختصار LXC) این دو قابلیت هستهٔ لینوکس را کنار هم گذاشت و نخستین ابزار کامل کانتینر لینوکسی بر پایهٔ هستهٔ اصلی (بدون نیاز به وصله‌های جانبی) را ساخت.

چرا LXC فراگیر نشد، اما داکر در سال ۲۰۱۳ انقلابی به پا کرد؟

مشکل LXC این بود که کار کردن با آن به‌شدت سخت بود و کانتینرها را شبیه ماشین‌های مجازی و صرفاً نسخهٔ سبک‌تر آن‌ها می‌دید. یعنی اگر روی لپ‌تاپ خودتان یک کانتینر LXC می‌ساختید، انتقال دقیق آن به میزبان طاقت‌فرسا بود. شرکت dotCloud -که بعداً نامش را به Docker تغییر داد- تکهٔ سوم پازل، یعنی **«فایل‌سیستم لایه‌ای (UnionFS / OverlayFS)»** را اضافه کرد و مفهوم Image را وارد بازی کرد. این تکه دیگر ایزوله‌سازی نیست، بسته‌بندی است.

داکر، فایل‌سیستم کانتینر را به لایه‌های فقط‌خواندنی[6] تبدیل کرد. حالا می‌شد کل محیط اجرای برنامه را در قالب یک بستهٔ استاندارد (Docker Image) بسته‌بندی کرد و دقیقاً مثل کدهای گیت، آن را روی هر بستر دلخواه، Push و Pull کرد. عملاً داکر تمام پیچیدگی‌های هسته و LXC را پشت یک ابزار ترمینالی ساده (docker run) پنهان می‌کرد.

البته در بخش پایانی این متن، نسل جدیدی از کانتینرها را معرفی خواهیم کرد که این تعریف سه‌گانه را هم به چالش می‌کشند. اما فعلاً می‌خواهیم بررسی کنیم بعد از این، چه بر سر معماری داکر آمد.



۱. Monolithic
۲. Container Runtime Interface
۳. AI Agents
۴. Namespaces (man-page)
۵. cgroups
۶. Read-only

کالبدشکافی یک غول: چرا داکر تکه‌تکه شد؟

در سال‌های ابتدایی (۲۰۱۳ و ۲۰۱۴)، داکر یک برنامهٔ یکپارچه بود. یک سرویس واحد و غول‌پیکر به نام dockerd[1] روی میزبان اجرا می‌شد و صفر تا صد کارها را یک‌تنه انجام می‌داد؛ از گرفتن دستورات کاربر[2] و دانلود ایمیج گرفته، تا تنظیم شبکه و ساختن خود کانتینر در هسته که ابتدا با کمک همان LXC و کمی بعدتر با کتابخانهٔ اختصاصی داکر به نام libcontainer انجام می‌شد. اما هرچقدر داکر محبوب‌تر شد، این معماری یکپارچه دو بحران بزرگ به بار آورد.

بحران اول: انحصار!

شاید تصور می‌کردید این بحران‌ها هر دو فنی هستند! اما بحران اول که غیرفنی بود، انحصار بود. داکر با سرعتی سرسام‌آور در حال بلعیدن بازار بود و قابلیت‌های بیشتری را مستقیم داخل همان daemon مرکزی اضافه می‌کرد (که بعدتر در ۲۰۱۶، Docker Swarm هم به آن‌ها اضافه شد). غول‌های دیگر تکنولوژی مثل گوگل، RedHat و شرکت CoreOS (که یک موتور کانتینری رقیب به نام rkt ساخته بود) احساس خطر کردند؛ کل دنیا به ابزاری وابسته شده بود که تحت کنترل یک شرکت خاص قرار داشت و هیچ استاندارد رسمی‌ای برای این‌که «کانتینر و ایمیج دقیقاً چه ساختاری دارند» وجود نداشت.

در سال ۲۰۱۵، برای جلوگیری از واگرا شدن زیست‌بوم، کنسرسیومی به نام OCI[3] تحت نظر بنیاد لینوکس شکل گرفت. استاندارد OCI ابتدا دو چیز را مشخص کرد: **یکی Image Spec، یعنی ساختار لایه‌های ایمیج و دیگری Runtime Spec که نحوهٔ اجرای یک باندل[4] را تعریف می‌کند.** باندل پوشه‌ای است که فایل تنظیمات استاندارد `config.json` و فایل‌سیستم ریشهٔ کانتینر را در بر می‌گیرد. چند سال بعد، Distribution Spec هم برای توزیع ایمیج‌ها در رجیستری‌ها اضافه شد.

در این راستا، داکر هم قدم بزرگی برداشت؛ بخش زیرین کدهای خودش، یعنی libcontainer را که مسئول مستقیم تعامل با هستهٔ لینوکس بود، از دل daemon یکپارچه بیرون کشید و در قالب یک ابزار کوچک و مستقل به نام runc به جامعهٔ متن‌باز اهدا کرد.

این‌جا بود که مفهوم **Low-level Runtime** شکل گرفت. کار یک Low-level Runtime (مثل runc که به زبان Go است یا جایگزین‌های سریع‌ترش مثل crun که توسط RedHat و به زبان C نوشته شده است) فقط این است: یک OCI Runtime Spec تحویل بگیرد، فراخوانی‌های سیستمی لازم برای ساختن فضاهای نام و گروه‌های کنترلی را صدا بزند و پردازه را راه‌اندازی کند!

بحران دوم: یک نقطهٔ شکست واحد[5]

برخلاف بحران اول، بحران دوم کاملاً فنی و یک مشکل عملیاتی بود. در معماری اولیه، تمام کانتینرهای در حال اجرا، پردازهٔ فرزند[6] مستقیم daemon داکر بودند. این یعنی اگر dockerd به‌دلیل یک باگ دچار مشکل[7] می‌شد یا اگر مدیر میزبان می‌خواست نسخهٔ داکر را به‌روزرسانی کند، تمام کانتینرهای در حال اجرا روی میزبان هم کشته می‌شدند!

شاید برایتان سؤال شود که چرا این اتفاق می‌افتد؟ مگر در لینوکس وقتی والد یک پردازه می‌میرد، آن پردازه «یتیم»[8] نمی‌شود و پردازهٔ ریشه[9] سرپرستی‌اش را به عهده نمی‌گیرد تا به کارش ادامه دهد؟

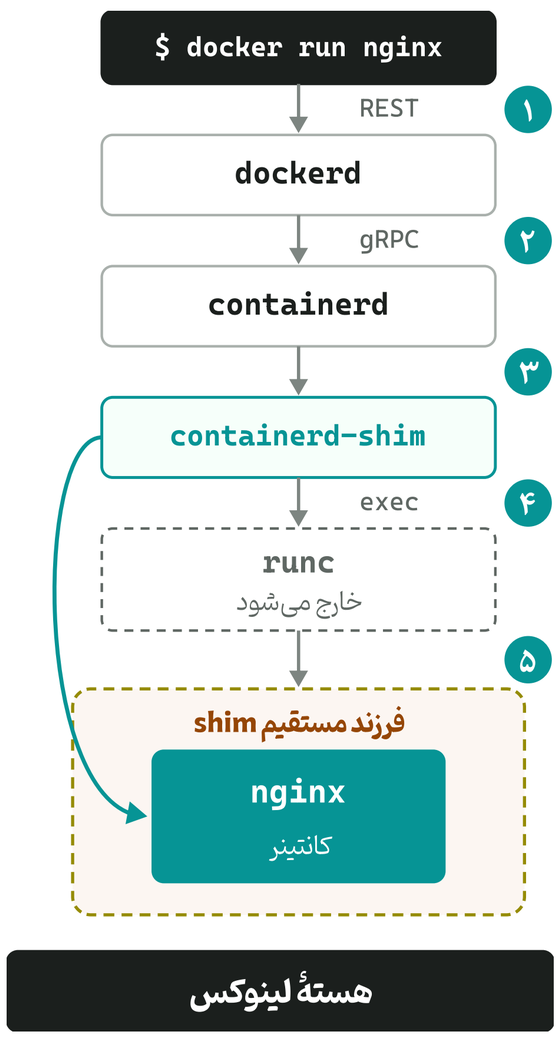
چرا، در حالت عادی همین‌طور است! اما مشکل از دو جا ناشی می‌شد: اول این‌که dockerd برای جمع‌آوری گزارش‌ها[10]، جریان‌های ورودی و خروجی استاندارد[11] کانتینر را از طریق لوله‌های[12] ارتباطی به خودش وصل می‌کرد. با مردن dockerd، سر دیگر این لوله‌ها بسته می‌شد و به محض این‌که کانتینر می‌خواست گزارش جدیدی چاپ کند، هستهٔ لینوکس سیگنال SIGPIPE را به کانتینر می‌فرستاد و آن را می‌کشت! دوم این‌که اگر کانتینر واقعاً به پردازهٔ ریشهٔ سیستم‌عامل به ارث می‌رسید، dockerd پس از بالا آمدن مجدد، دیگر نمی‌توانست کانتینر یتیم‌شده را پیدا کند و کد خروج[13] آن را از هسته بگیرد.

برای حل این مشکل در سال ۲۰۱۶، مهندسان داکر تصمیم گرفتند daemon خود را باز هم کوچک‌تر کنند. آن‌ها تمام منطق مربوط به مدیریت چرخهٔ حیات کانتینر (از دانلود ایمیج و مدیریت فضای ذخیره‌سازی گرفته تا صدا زدن runc) را جدا کردند و به سرویس مستقلی به نام containerd سپردند (که آن هم کمی بعد در ۲۰۱۷ به بنیاد CNCF اهدا شد). این‌جا بود که مفهوم **High-level Runtime** پدیدار شد؛ مؤلفه‌ای که چرخهٔ حیات کانتینر را مدیریت می‌کند؛ یعنی ایمیج‌ها را دریافت و ذخیره می‌کند، باندل را آماده می‌کند و برای اجرا به یک Low-level Runtime تحویل می‌دهد.

اما برای این‌که خود containerd هم به سرنوشت dockerd دچار نشود و بتواند بدون کشتن کانتینرها بازراه‌اندازی[14] شود، قطعهٔ ظریفی به نام containerd-shim به معماری اضافه شد. در معماری جدید، به‌ازای هر کانتینر، یک پردازهٔ سبک به‌عنوان والد مستقیم آن کانتینر بالا می‌آید و تا لحظهٔ مرگ کانتینر بالای سرش می‌ماند. این پردازه، لوله‌های ورودی و خروجی کانتینر را باز نگه می‌دارد تا با بازراه‌اندازی containerd، کانتینر به‌دلیل سیگنال SIGPIPE نمیرد.

این shim یک وظیفهٔ دیگر هم دارد. runc بعد از راه‌اندازی کانتینر خارج می‌شود و پردازهٔ اصلی کانتینر یتیم می‌ماند. shim با ثبت خود به‌عنوان subreaper در هستهٔ لینوکس، والد جدید این پردازه می‌شود تا به پردازهٔ ریشهٔ میزبان به ارث نرسد و کد خروج آن را نگه می‌دارد تا هر وقت containerd بالا آمد و به آن متصل[15] شد، گزارش را تحویلش دهد.

امروزه شما وقتی دستور docker run را می‌زنید، این زنجیره از اتفاقات رخ می‌دهد.



۱. Docker daemon
۲. قابل توجه است که دستورات خط فرمان docker با REST API به dockerd رسیده و توسط آن اجرا می‌شوند.
۳. Open Container Initiative
۴. Bundle
۵. Single point of failure
۶. Child process
۷. Crash
۸. Orphan
۹. Root process (PID=۱)
۱۰. Logs
۱۱. Standard I/O streams (stdin/stdout/stderr)
۱۲. Pipes
۱۳. Exit code / Error code
۱۴. Restart
۱۵. Re-attach

۱. ابزار خط فرمان داکر، درخواست شما را با REST API به dockerd می‌فرستد.
۲. dockerd daemon که دیگر خودش درگیر جزئیات اجرا نمی‌شود، از طریق پروتکل gRPC به containerd می‌گوید: «این کانتینر را برایم بالا بیاور».
۳. سرویس containerd به‌ازای آن کانتینر شما، یک پردازهٔ واسط و سبک به نام containerd-shim ایجاد می‌کند.
۴. حالا این shim می‌رود و runc را صدا می‌زند.
۵. ابزار runc فضاهای نام و گروه‌های کنترلی را در هسته می‌سازد، برنامهٔ داخل کانتینر (مثلاً nginx) را راه‌اندازی می‌کند و بلافاصله خودش خارج می‌شود!

دقت کنید که اگر قرار بود به‌ازای هر کانتینر، یک پردازهٔ runc (که به زبان Go نوشته شده است) در حافظه بماند، مصرف حافظهٔ میزبان سر به فلک می‌کشید.

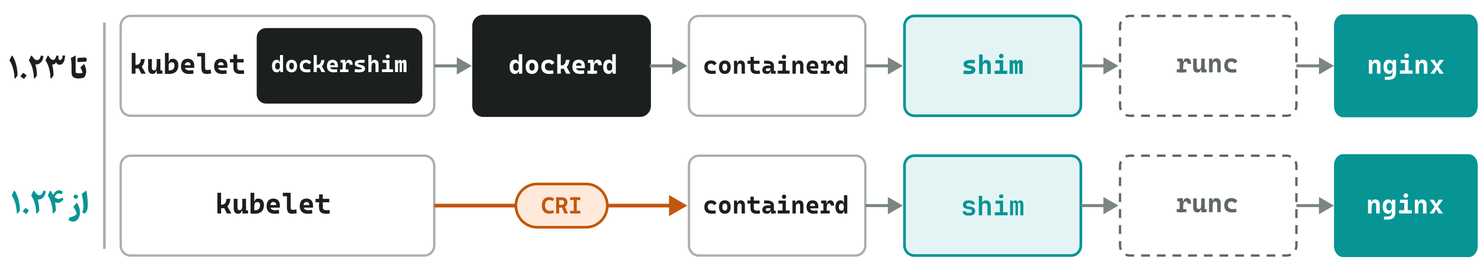
به لطف این معماری چندلایه، امروز می‌توان daemon داکر یا containerd را روی میزبان بازراه‌اندازی یا به‌روزرسانی کرد، در حالی‌که کانتینرها زیر سایهٔ containerd-shim بدون هیچ وقفه‌ای به کارشان ادامه می‌دهند.
درست زمانی که به نظر می‌رسید این معماری به کمال رسیده است، غول جدیدی به نام کوبرنتیز وارد میدان شد و معادلات کمی تغییر کرد.

ورود کوبرنتیز به میدان: استاندارد CRI

وقتی گوگل کوبرنتیز را در میانهٔ ۲۰۱۵ معرفی کرد، داکر پادشاه بی‌رقیب دنیای کانتینرها بود. به همین دلیل، مؤلفهٔ[1] kubelet (مأمور کوبرنتیز روی هر node که وظیفه دارد کانتینرها را بالا بیاورد) طوری نوشته شده بود که مستقیماً با dockerd صحبت می‌کرد. اما خیلی زود دو مشکل اساسی سر راه کوبرنتیز سبز شد.
اول این‌که موتورهای کانتینری دیگری (مثل rkt) هم وارد بازار شده بودند و توسعه‌دهندگان می‌خواستند از آن‌ها در کوبرنتیز استفاده کنند؛ این موضوع کدهای داخل kubelet را به کلاف سردرگمی از شرط‌های if/else برای پشتیبانی از ابزارهای مختلف تبدیل کرده بود.
دوم این‌که، داکر در اصل برای **انسان‌ها** طراحی شده بود و امکانات بی‌شماری مثل ابزار CLI، قابلیت ساخت ایمیج، شبکهٔ اختصاصی و Docker Swarm داشت؛ در حالی‌که کوبرنتیز یک هماهنگ‌کنندهٔ[2] ماشینی بود و به هیچ‌کدام از این امکانات اضافه نیاز نداشت.
برای حل این تنگنا، تیم کوبرنتیز در سال ۲۰۱۶ یک استاندارد ارتباطی به نام CRI معرفی کرد که شامل دو سرویس اصلی بود: ImageService برای مدیریت ایمیج‌ها و RuntimeService برای مدیریت چرخهٔ حیات پادها و کانتینرها. در واقع کوبرنتیز با تعریف CRI (که یک پروتکل مبتنی بر gRPC است) عملاً اعلام کرد که کدهایش را برای هیچ ابزار خاصی تغییر نمی‌دهد و از آن پس kubelet فقط به زبان استاندارد CRI صحبت می‌کند. هر ابزاری که می‌خواهد زیر دست کوبرنتیز کار کند، باید این زبان را بفهمد.

تولد و مرگ dockershim

معرفی CRI یک ایدهٔ مهندسی درخشان بود، اما یک مشکل این میان وجود داشت: تمام دنیا در زیر کوبرنتیز از داکر استفاده می‌کردند، ولی dockerd زبان CRI را نمی‌فهمید! برای این‌که کلاسترهای کاربران از کار نیفتند، تیم کوبرنتیز مجبور شد یک ماژول مترجم به نام dockershim داخل خود kubelet بنویسد تا زبان CRI را به زبان dockerd ترجمه کند.
نتیجهٔ این کار، یک زنجیرهٔ طولانی برای بالا آوردن یک کانتینر ساده در کوبرنتیز بود که امروزه حتی مضحک به نظر می‌رسد!
کم‌کم نگهداری از کدهای dockershim برای تیم کوبرنتیز به یک بار اضافی تبدیل شده بود. آن‌ها تصمیم گرفتند نگهداری از این کدها را کنار بگذارند و به‌طور مستقیم با خود containerd ارتباط برقرار کنند؛ در همین راستا افزونهٔ پشتیبانی از CRI به‌طور مستقیم به containerd اضافه شد و dockershim حذف شد.



دقت کنید که این dockershim هیچ ربطی به containerd-shim که قبل‌تر دیدیم ندارد! containerd-shim یک پردازهٔ سطح پایین و والد کانتینر است، اما dockershim صرفاً یک مترجم نرم‌افزاری بین کوبرنتیز (kubelet) و داکر (dockerd) بود.

واکنش RedHat: تولد CRI-O و conmon

هم‌زمان با این تحولات، شرکت RedHat ایدهٔ رادیکال‌تری را مطرح کرد: اصلاً چرا از containerd استفاده کنیم که هم daemon زیرین داکر است و هم برای کوبرنتیز کار می‌کند؟ یک High-level Runtime می‌سازیم که از همان ابتدا، منحصراً برای استاندارد CRI کوبرنتیز طراحی شده باشد و هیچ قابلیت اضافه‌ای نداشته باشد.
نتیجهٔ این تفکر، تولد پروژه‌ای به نام CRI-O (ترکیب دو استاندارد CRI و OCI) بود.
همچنین مهندسان RedHat از containerd-shim (که به زبان Go نوشته شده است) استفاده نکردند؛ آن‌ها یک برنامهٔ بسیار سبک به زبان C به نام conmon (مخفف Container Monitor) نوشتند که دقیقاً همان نقش والد کانتینر، مدیریت گزارش‌ها و ثبت کد خروج را با مصرف حافظهٔ بسیار کم‌تر ایفا می‌کند.



۱. Component
۲. Orchestrator

پاشنهٔ آشیل امنیتی: چرا داشتن یک daemon با دسترسی ریشه ترسناک است؟

در بخش قبل دیدیم که چطور شرکت RedHat برای محیط کوبرنتیز، CRI-O و conmon را معرفی کرد؛ اما داستان به این‌جا ختم نشد. مهندسان RedHat نگاهشان را به سیستم‌های شخصی توسعه‌دهندگان و میزبان‌های تکی برگرداندند؛ جایی که همه از دستورات docker استفاده می‌کردند. آن‌ها یک پرسش بنیادی مطرح کردند:

«اصلاً چرا برای اجرای یک کانتینر، به یک سرویس پس‌زمینهٔ دائمی نیاز داریم؟»

بیایید نگاه عمیق‌تری به معماری کلاینت-سروری داکر بیندازیم. وقتی شما در ترمینال دستور docker run را می‌زنید، ابزار خط فرمان داکر خودش هیچ کاری نمی‌کند؛ بلکه صرفاً یک درخواست از طریق سوکت یونیکس (/var/run/docker.sock) به daemon مرکزی می‌فرستد. این معماری دو مشکل بزرگ امنیتی و ساختاری دارد.

۱. دسترسی مطلق و خطرناک: دیمن داکر با دسترسی کامل ریشه روی میزبان اجرا می‌شود. وقتی شما برای راحتی کار، یک کاربر را به گروه docker اضافه می‌کنید تا بدون زدن sudo دستورات داکر را اجرا کند، در واقع عملاً به آن کاربر دسترسی ریشه بدون نیاز به رمز عبور داده‌اید! کافی است دستور زیر در shell آن کاربر اجرا شود تا کل فایل‌سیستم میزبان را با دسترسی ریشه در اختیار بگیرد:

```
$ docker run -it          \
    -v /:/host alpine     \
    chroot /host
```

۲. پاک شدن ردپا در سیستم‌عامل: در لینوکس سامانه ممیزی[1] وجود دارد که ثبت می‌کند کدام کاربر چه دستوری را اجرا کرده است. وقتی کاربر ali دستور docker run را می‌زند، صرفاً همین دستور برای او ثبت می‌شود؛ ولی اصل کار که ساختن کانتینر و بالا آمدن آن است، توسط dockerd انجام می‌شود؛ پس در گزارش امنیتی مربوط به آن، کاربر root ثبت می‌شود و نه ali؛ پس ردپای کاربر واقعی در گزارش‌های امنیتی هسته گم می‌شود.

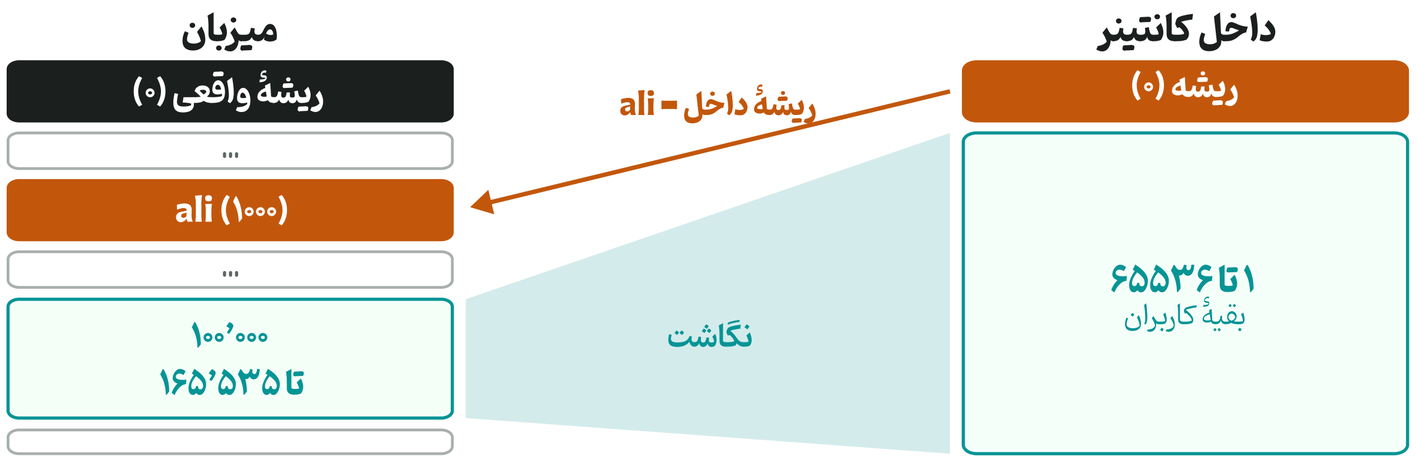
تولد Podman: بازگشت به فلسفهٔ اصیل یونیکس

برای حل این مشکلات، RedHat ابزار Podman را معرفی کرد که کاملاً سازگار با دستورات داکر است (تا جایی که می‌توانید بنویسید alias docker=podman)، اما هیچ daemon ای در پس‌زمینه ندارد!

Podman از معماری کلاینت-سرور استفاده نمی‌کند؛ این ابزار مدل سنتی Fork/Exec در لینوکس را به‌کار می‌گیرد. وقتی دستور podman run را می‌زنید، خود پردازهٔ Podman به‌طور مستقیم ابزارهای زیرین (conmon و crun) را به‌عنوان فرزند خودش ایجاد می‌کند. به محض این‌که کانتینر ساخته شد، پردازهٔ اصلی Podman و crun خارج می‌شوند و فقط conmon و کانتینر شما در حافظه با هویت همان کاربری که دستور را اجرا کرده است، باقی می‌مانند. در نتیجه هم ردپای کاربر در هسته حفظ می‌شود و هم هیچ daemon مرکزی و دائمی‌ای وجود ندارد که باعث ایجاد سطح حملهٔ بیشتر یا بروز خطا شود.

جادوی کانتینرهای Rootless: وقتی کاربر ریشه، واقعاً ریشه نیست!

حذف daemon در Podman، دروازه را برای یکی از مهم‌ترین دستاوردهای امنیتی سال‌های اخیر باز کرد؛ کانتینرهای Rootless؛ قابلیتی که در Podman به‌صورت پیش‌فرض و ذاتی وجود دارد و بعدها به داکر و containerd هم اضافه شد. ما می‌دانیم که در هستهٔ لینوکس، ساختن فضای نام شبکه یا سوار کردن[2] فایل‌سیستم نیازمند دسترسی ریشه است؛ پس چطور یک کاربر عادی و بدون دسترسی سطح بالا می‌تواند کانتینر بسازد؟ پاسخ در مکانیزم هوشمندانهٔ **فضای نام کاربرها**[3] نهفته است. این تنها فضای نامی در لینوکس است که یک کاربر عادی هم اجازه دارد آن را بسازد. وقتی یک کانتینر به صورت Rootless اجرا می‌شود، این اتفاقات می‌افتد:



بازه‌ای از شناسه‌های کاربری فرعی (مثلاً از ۱۰۰٬۰۰۰ تا ۱۶۵٬۵۳۵) که در فایل‌های /etc/subuid و /etc/subgid برای کاربر عادی تعریف شده است، توسط ابزارهای newuidmap و newgidmap خوانده می‌شود و این ابزارها یک جدول نگاشت[4] برای فضای نام کاربر می‌سازند؛ هسته فقط همین جدول را اعمال می‌کند.

از دید داخل کانتینر: پردازه فکر می‌کند کاربر ریشه (UID=0) است و تمام مجوزهای لازم برای نصب بسته یا تغییر فایل‌های داخل کانتینر را دارد.

از دید هستهٔ میزبان: همان پردازهٔ ریشهٔ داخل کانتینر، در واقعیت یک پردازهٔ معمولی با شناسهٔ خود کاربر عادی شما (مثلاً ۱۰۰۰) است؛ و بقیهٔ شناسه‌های داخل کانتینر (۱ به بعد) روی بازهٔ فرعی (۱۰۰٬۰۰۰ به بعد) نگاشت می‌شوند.

ارزش این کار زمانی مشخص می‌شود که یک آسیب‌پذیری امنیتی رخ دهد. در کانتینرهای سنتی که با daemon راه‌اندازی شده‌اند، اگر مهاجم موفق می‌شد با دور زدن ایزوله‌سازی، از کانتینر فرار کند[5]، روی میزبان هم دسترسی ریشه داشت و کل میزبان را تسخیر می‌کرد؛ اما در یک کانتینر Rootless، مهاجم روی میزبان فقط همان دسترسی‌های کاربری را دارد که کانتینر را اجرا کرده است و به فایل‌های سیستمی و سایر کاربران دسترسی ندارد. این خیلی بهتر از دسترسی ریشه است.

البته دنیای Rootless بدون چالش هم نیست. برای مثال، از آن‌جا که یک کاربر عادی اجازه ندارد در فضای نام اصلی میزبان کارت شبکهٔ مجازی بسازد یا قوانین فایروال بنویسد، کانتینرهای Rootless برای اتصال به شبکه از ابزارهای خلاقانه‌ای در فضای کاربری مثل slirp4netns یا ابزار جدیدتر و سریع‌تر pasta استفاده می‌کنند که بسته‌های شبکهٔ کانتینر را در فضای کاربری می‌گیرند و به سوکت‌های شبکهٔ عادی روی میزبان ترجمه می‌کنند. توضیح دربارهٔ این ابزارها فعلاً در حوصلهٔ این متن نمی‌گنجد.

۱. Linux Audit (auditd)
۲. Mount
۳. Users Namespace
۴. Mapping
۵. Container Escape

وقتی هستهٔ لینوکس کم می‌آورد؛ انفجار CVEها، عصر عامل‌های هوشمند و ظهور MicroVM

بگذارید یک بازگشت به اوایل متن داشته باشیم؛ اگر خاطرتان باشد، گفتیم کانتینر در معنای سنتی خود، یک پردازهٔ معمولی است که با مکانیزم‌هایی نظیر فضاهای نام و گروه‌های کنترلی ایزوله شده است؛ و نیز گفتیم که در ادامه نسل جدیدی از کانتینرها را معرفی می‌کنیم که این تعریف را به چالش می‌کشند. در این بخش می‌خواهیم به این موضوع بپردازیم.

در تمام ابزارهایی که معرفی کردیم و پیشرفت‌هایی که از داکر تا Podman و کانتینرهای Rootless از آن‌ها نام بردیم، یک چیز مشترک بود؛ و آن هم **استفاده از هستهٔ لینوکس مشترک روی میزبان** توسط کانتینرها بود. هستهٔ لینوکس حدود ۳۰۰ تا ۴۰۰ فراخوانی سیستمی دارد؛ وقتی برنامه‌ای در داخل یک کانتینر بخواهد فایلی را بخواند، بسته‌ای را در شبکه جابه‌جا کند یا...، به‌طور مستقیم از طریق فراخوانی‌های سیستمی با هستهٔ اصلی میزبان در تعامل است. با وجود این‌که می‌توان با مکانیزم‌هایی مانند seccomp برخی از فراخوانی‌ها را مسدود کرد، ولی باز هم این سطح تعامل وسیع با یک هستهٔ مشترک، یک سطح حملهٔ بسیار قابل‌توجه است.

انفجار نرخ آسیب‌پذیری‌ها در هستهٔ لینوکس

شاید تا چند سال پیش، تکیه بر امنیت کانتینرهای سنتی، کافی به نظر می‌رسید؛ ولی در یکی دو سال اخیر، به‌دلیل ورود هوش مصنوعی به مباحث امنیتی و رشد نرخ کشف آسیب‌پذیری‌های جدید در هستهٔ لینوکس و ابزارهای دیگر، معادلات عوض شده است. امروزه با پیچیده‌تر شدن کدهای هسته (به‌خصوص در زیرسیستم‌هایی مانند eBPF و io_uring) و استفاده از هوش مصنوعی و ابزارهای خودکار کشف باگ و نیز تبدیل شدن تیم توسعه‌دهندگان هستهٔ لینوکس به مرجع رسمی ثبت آسیب‌پذیری[1]، شاهد یک انفجار واقعی در تعداد آسیب‌پذیری‌های ثبت‌شده برای هستهٔ لینوکس هستیم.

در چنین شرایطی، یک آسیب‌پذیری روز صفر، یا حتی باگی که وصله[2] شده ولی هنوز روی میزبان اعمال نشده است، کافی است تا پردازهٔ مهاجم با ارسال یک فراخوانی سیستمی معیوب، از باگ هسته بهره‌برداری[3] کند و از فضای ایزولهٔ کانتینر و حتی محدودیت‌های Rootless فرار کند و دسترسی خود را ارتقا[4] دهد. به‌ویژه آن‌که خود مکانیزم فضای نام کاربران در کانتینرهای Rootless، با اعطای مجوزهای مجازی خاصی به کاربر عادی، بخش‌های پیچیده‌تری از کدهای هسته را در دسترس او قرار می‌دهد و عملاً سطح حملهٔ هسته را وسیع‌تر می‌کند.

نیازمندی عامل‌های هوش مصنوعی و عوض شدن زمین بازی

فارغ از این‌که خود هسته می‌تواند آسیب‌پذیر باشد، الگوی استفاده از کانتینرها هم تغییر کرده است. در گذشتهٔ نزدیک، تقریباً همهٔ کدهایی که در کانتینر اجرا می‌شدند، عملاً قابل‌اعتماد بودند؛ زیرا یا توسط برنامه‌نویسان خود سازمان نوشته شده بودند یا توسط یک مرجع معتبر تأیید شده بودند و در نهایت با بررسی امنیت خود کد، می‌توانستیم اطمینان داشته باشیم که فراخوانی سیستمی نابه‌جایی انجام نمی‌دهد؛ ولی امروزه در عصر عامل‌های هوش مصنوعی، با استفادهٔ جدیدی از کانتینرها روبه‌رو هستیم. کاربری از هوش مصنوعی می‌خواهد مسئله‌ای را حل کند یا داده‌ای را تحلیل کند؛ عامل در لحظه یک تکه کد تولید کرده و در محیط ایزوله اجرا می‌کند و خروجی را برمی‌گرداند.

اجرای کدی که توسط هوش مصنوعی تولید شده -و ممکن است دارای خطا باشد یا در اثر حملهٔ تزریق پرامپت، حاوی دستورات مخرب باشد- عملاً معادل این است که به یک مهاجم ناشناس اجازه دهیم هر کدی که می‌خواهد روی میزبان اجرا کند.

در این شرایط، امنیت کانتینرهای سنتی دیگر کافی نیست! از طرفی استفاده از ماشین‌های مجازی سنتی که هستهٔ مجزا دارند هم به‌دلیل سنگین بودن و زمان بالا آمدن طولانی، برای این کاربردها مناسب نیست.

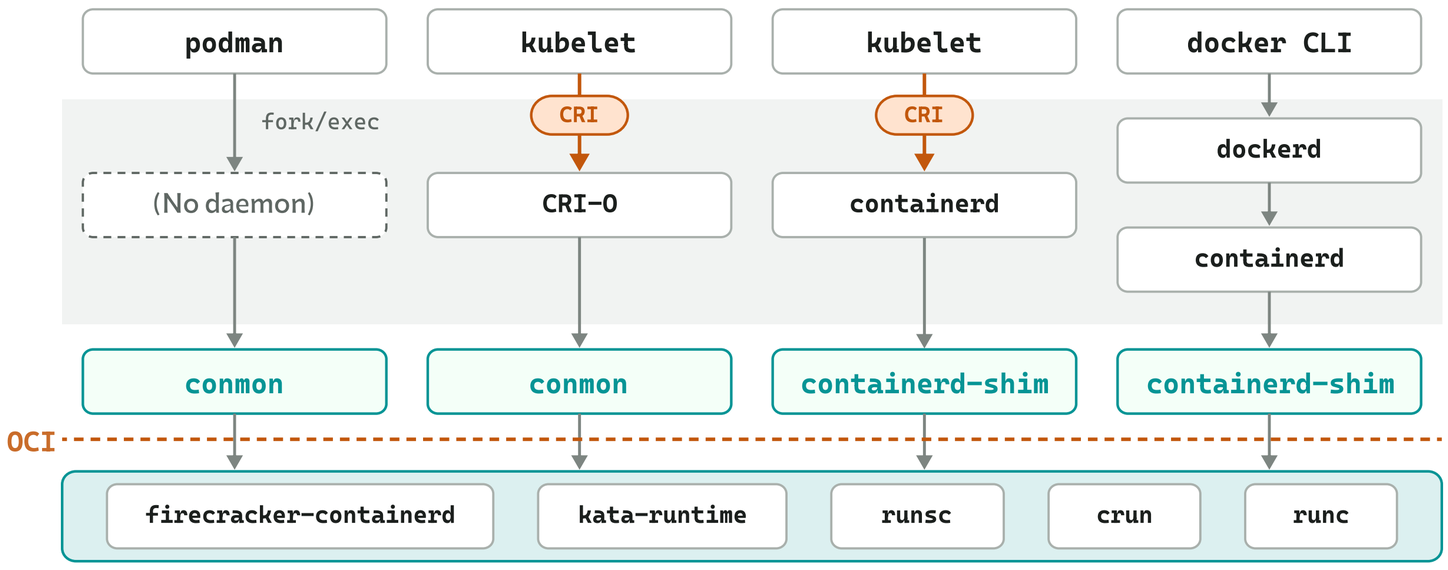
زیبایی استانداردها؛ تعویض موتور در حین پرواز

این‌جاست که ارزش استانداردهایی که در بخش‌های قبلی به آن‌ها اشاره کردیم خودش را نشان می‌دهد. از آن‌جا که امروز همهٔ ابزارهای سطح بالا مانند کوبرنتیز و containerd با زبان استاندارد صحبت می‌کنند، ما می‌توانیم بدون این‌که نحوهٔ ساخت ایمیج یا دستورات‌مان را تغییر دهیم، **صرفاً در پایین‌ترین لایه، Low-level Runtime را تغییر دهیم؛ یعنی به containerd بگوییم runc را با اجراکنندهٔ دیگری جایگزین کند که از هستهٔ میزبان استفاده نمی‌کند.**

امروزه چند رویکرد مهندسی برای جایگزینی این لایهٔ آخر و ساخت کانتینرهای فوق‌ایزوله وجود دارد. البته این مبحث بسیار جدیدی در دنیای مهندسی نرم‌افزار و سیستم‌عامل است و پژوهش‌ها و ابزارها در این زمینه بسیار جدید و با سرعت زیادی در حال توسعه‌اند.

رویکرد اول، هسته در فضای کاربری

مهندسان گوگل برای حل مشکل هستهٔ مشترک، پروژه‌ای به نام gVisor (با اجراکنندهٔ استاندارد runsc) را خلق کردند. در این معماری، کانتینر اجازه ندارد به‌طور مستقیم با هستهٔ لینوکس میزبان صحبت کند؛ بلکه برنامه‌ای به نام Sentry (که به زبان امن Go نوشته شده است) مثل یک دیوار بین کانتینر و هستهٔ میزبان می‌نشیند. این برنامه خودش نقش یک هستهٔ شبیه‌سازی‌شده در فضای کاربری را بازی



۱. CVE Numbering Authority (CNA)
۲. Patch
۳. Exploit
۴. Privilege Escalation

می‌کند و تمام فراخوانی‌های سیستمی کانتینر را جواب می‌دهد و خودش برای کارهای ضروری فقط از تعداد بسیار محدودی فراخوانی امن هستهٔ میزبان استفاده می‌کند. در نتیجه، حتی اگر کانتینر حاوی بدافزار باشد، سطح حمله به هستهٔ میزبان به‌شدت کوچک می‌شود.

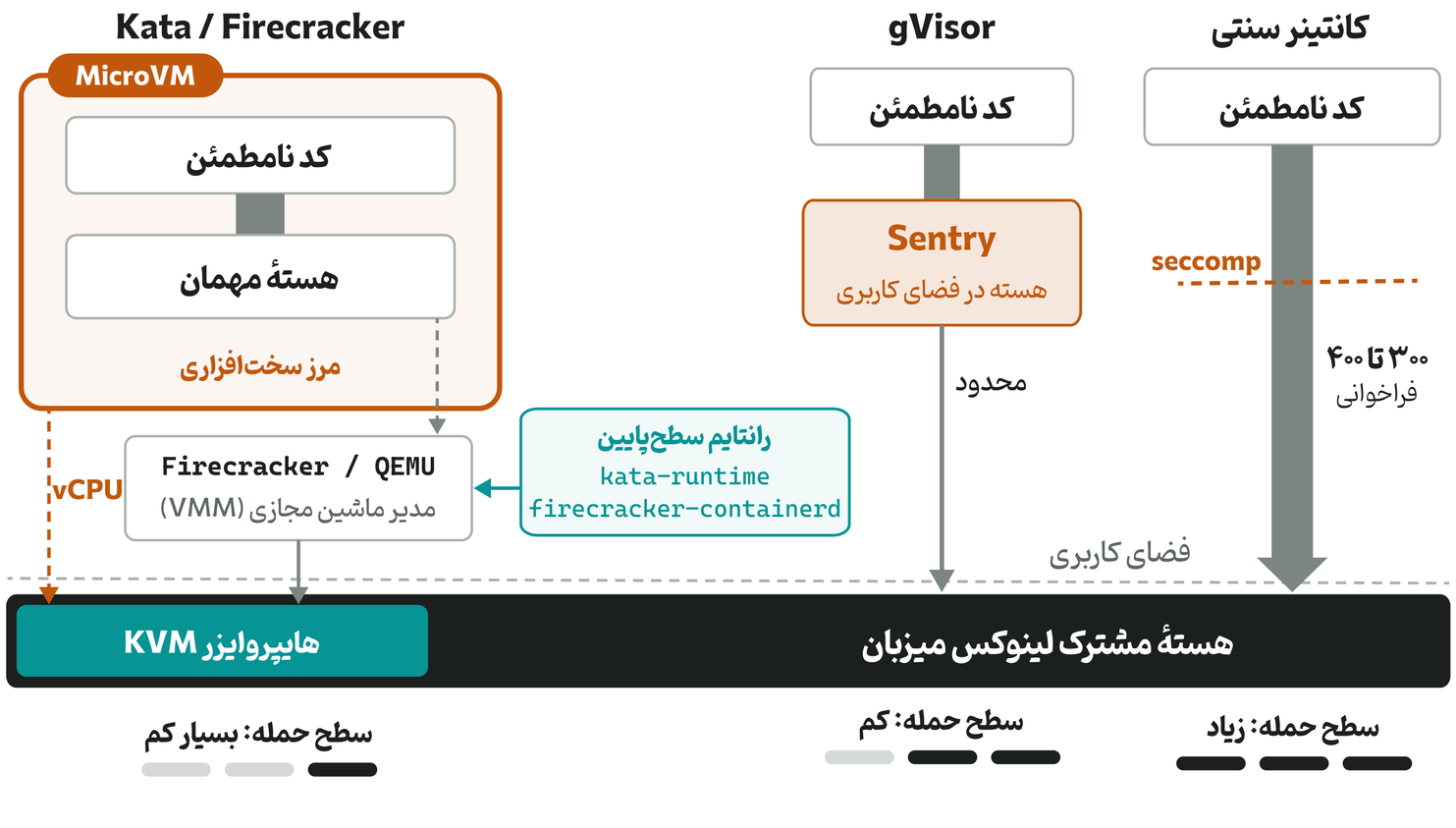
رویکرد دوم، استفاده از ماشین مجازی در لباس کانتینر

ایدهٔ Kata Containers ساده اما رادیکال است: وقتی دستور اجرای کانتینر را می‌دهید، Kata فضای نامی روی هستهٔ میزبان نمی‌سازد؛ در عوض، به کمک قابلیت مجازی‌سازی سخت‌افزاری لینوکس (KVM)[1]، در کسری از ثانیه یک ماشین مجازی فوق‌سبک بالا می‌آورد که هستهٔ لینوکس مجزا و اختصاصی خودش را دارد و کانتینر شما را درون آن اجرا می‌کند. از بیرون، شما و کوبرنتیز فکر می‌کنید با یک کانتینر معمولی طرف هستید، اما در باطن، یک دیوار سخت‌افزاری بسیار محکم‌تر بین کانتینر و میزبان کشیده شده است.

مکانیزم Kata برای بالا آوردن ماشین مجازی، نیاز به یک مدیر ماشین مجازی[2] دارد که در فضای کاربری از KVM استفاده می‌کند؛ برای این کار می‌توان از QEMU بهره برد. اما وقتی شرکت آمازون (AWS) می‌خواست زیرساخت سرویس‌های بدون سرور[3] خود را بسازد و زمانی که سکوهای اجرای کد برای هوش مصنوعی به دنبال بستری برای اجرای کد تولیدشدهٔ آن‌ها بودند، به چیزی سریع‌تر نیاز داشتند. ماشین‌های مجازی سنتی (که از ابزاری مثل QEMU استفاده می‌کنند) کند هستند، چون کدهای مربوط به شبیه‌سازی صدها قطعهٔ سخت‌افزاری قدیمی (از کارت صدا و گرافیک گرفته تا فلاپی‌دیسک!) را در خود دارند.

این‌جا بود که مهندسان آمازون دست‌به‌کار شدند و با زبان Rust، یک مدیر ماشین مجازی حداقلی به نام Firecracker نوشتند که تمام قطعات اضافه را دور ریخته و فقط چند قطعهٔ حیاتی (شبکه، دیسک، vsock، کنسول سریال و یک کنترلر کیبورد حداقلی) را شبیه‌سازی می‌کند. این‌جا بود که MicroVM متولد شد؛ یک ماشین مجازی واقعی با هستهٔ مجزا که طبق ادعای مستندات رسمی آن، در کمتر از ۱۲۵ میلی‌ثانیه بوت می‌شود و تنها ۵ مگابایت سربار حافظه دارد! این مدیر ماشین مجازی را می‌توان هم در لایهٔ زیرین Kata Containers جایگزین QEMU کرد و هم از طریق پروژهٔ firecracker-containerd به‌طور مستقیم به containerd متصل نمود.

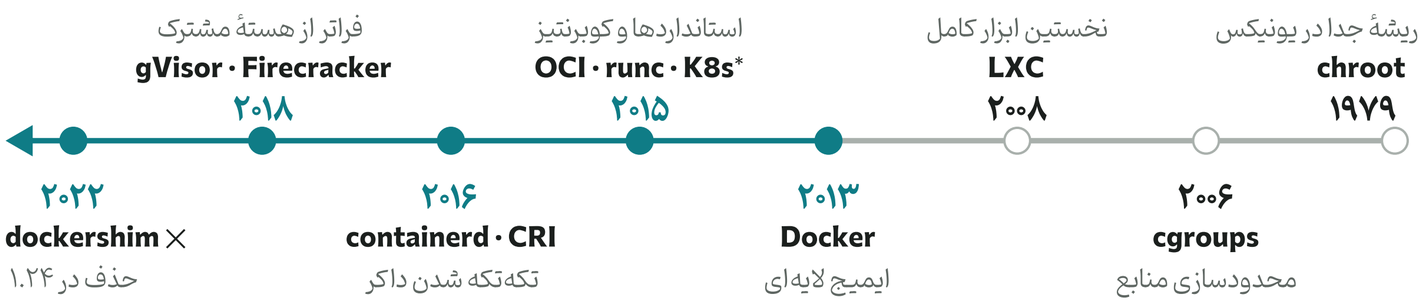
امروزه تقریباً تمام پلتفرم‌های مدرنی که محیط اجرای کد برای هوش مصنوعی فراهم می‌کنند (مانند E2B)، در لایهٔ زیرین خود به‌طور مستقیم از Firecracker بهره می‌برند.



سخن پایانی

سفر ما از یک فراخوانی سیستمی سادهٔ chroot در سال ۱۹۷۹ شروع شد، به غول یکپارچه‌ای به نام داکر در سال ۲۰۱۳ رسید و دیدیم که چطور نیازهای عملیاتی و استانداردهای باز (OCI و CRI)، این غول را به قطعات کوچک، تخصصی و ماژولاری مثل containerd، shim، runc و conmon تقسیم کردند.

در نهایت، همین معماری لایه‌لایه به ما اجازه داد در مواجهه با انفجار آسیب‌پذیری‌های امنیتی، ابتدا به سراغ کانتینرهای Rootless برویم و امروز در عصر عامل‌های هوشمند، مرز بین «کانتینر» و «ماشین مجازی» را با MicroVMها بسیار کم‌رنگ کنیم؛ جایی که سرعت کانتینر و امنیت ماشین مجازی دیگر تا این اندازه در برابر یک‌دیگر قرار ندارند و تا حد زیادی در هم ادغام شده‌اند.

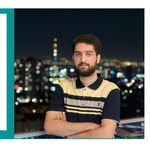


* برخی مواقع برای اختصار، Kubernetes (۸ حرف) را K8s می‌نویسند.



۱. Kernel-based Virtual Machine
۲. VMM: Virtual Machine Manager
۳. AWS Serverless Services e.g. AWS Lambda

عشق، دروغ و سیلیکون؛ بحران شخصی در قلب صنعت تراشه



امیرحسین شهیدی
کارشناسی ۱۴۰۱
دانشگاه صنعتی شریف

۴ دقیقه

در نوامبر ۲۰۲۶، افشای رابطهٔ خارج از ازدواج جنسن هوانگ[۱]، بنیان‌گذار و مدیرعامل انویدیا، به بحرانی فراتر از زندگی شخصی او تبدیل شد. ماجرا پای احتمال جاسوسی، رقابت آمریکا و چین و قاچاق احتمالی تراشه‌های پیشرفته را به میان کشید و نگرانی‌هایی دربارهٔ امنیت اقتصادی یکی از ارزشمندترین شرکت‌های جهان پدید آورد. آلیس هوانگ[۲] -معروف به Huang Le- فعال اقتصادی چینی و از مدیران سابق شرکت سنگاپوری Megaspeed International بود. این شرکت در حوزهٔ زیرساخت پردازش ابری و تجهیزات هوش مصنوعی فعالیت داشت و با انویدیا ارتباط تجاری برقرار کرده بود. گزارش‌ها از ارتباط آلیس و جنسن هوانگ از سال ۲۰۲۴ حکایت داشتند و تصاویر حضور مشترک آن‌ها در تایوان نیز توجه رسانه‌ها را جلب کرد.

نیویورک‌تایمز مدعی شد نهادهای اطلاعاتی آمریکا آلیس را مرتبط با سرویس اطلاعاتی چین می‌دانند و معتقدند او اطلاعاتی به پکن منتقل کرده است. هوانگ در ۸ نوامبر ۲۰۲۶ به خیانت نسبت به همسرش اعتراف کرد و از خانواده و همکارانش عذرخواهی کرد. او از تحقیقات مستقل هیئت‌مدیره حمایت کرد؛ هیئت‌مدیره نیز کمیته‌ای متشکل از مدیران مستقل و مشاوران حقوقی خارجی برای بررسی احتمال نفوذ تشکیل داد. هوانگ اعلام کرد با این تحقیقات همکاری خواهد کرد. انویدیا تأکید کرد در آن زمان شواهدی که نشان دهد اسرار تجاری یا امنیت ملی آمریکا به خطر افتاده است، در اختیار ندارد.

Megaspeed در زنجیرهٔ تأمین تراشه‌های انویدیا و خرید تجهیزات پیشرفته‌ای مانند H۱۰۰ و H۲۰۰ نقش داشت. با وجود محدودیت صادرات تراشه به چین، احتمال انتقال تجهیزات از طریق شرکت‌های واسطه مطرح شد و آمریکا و سنگاپور فعالیت‌های شرکت را بررسی کردند. بر اساس گزارش The Information که رویترز نیز آن را بازتاب داد، شبکهٔ مرتبط با آلیس ممکن است در انتقال حدود ۲۴۰ هزار تراشهٔ انویدیا به ارزش دست‌کم ۷٫۵ میلیارد دلار به شرکت‌های چینی نقش داشته باشد. این ادعا هنوز نتیجهٔ قطعی تحقیقات نیست.

در صورت اثبات نفوذ اطلاعاتی از مسیر رابطهٔ شخصی، این پرونده می‌تواند نمونه‌ای از Honey Trap یا «تلهٔ عاطفی» باشد.

در این روش، فرد با ایجاد رابطهٔ عاشقانه، جنسی یا عاطفی اعتماد هدف را جلب می‌کند تا اطلاعات حساس بگیرد یا از نقاط ضعف او برای اعمال فشار استفاده کند.

این روش الزاماً به‌معنای اغواگری ساده نیست.

گاهی رابطه‌ای طولانی و ظاهراً طبیعی، بستر جمع‌آوری تدریجی اطلاعات می‌شود.

این روش سابقهٔ تاریخی دارد.

در جنگ سرد، اشتازی[۳]، سازمان اطلاعاتی آلمان شرقی، از مأمورانی موسوم به Romeo Spies برای برقراری رابطه با زنان دارای دسترسی به اطلاعات محرمانه استفاده می‌کرد. رابطهٔ برنارد بورسی‌کو[۴]، دیپلمات فرانسوی، با هنرمند چینی شی پی‌پو[۵] نیز به انتقال اسناد محرمانه به چین منجر شد.

پروندهٔ کریستین فنگ[۶] نمونهٔ دیگری از تلاش برای ارتباط با سیاست‌مداران آمریکایی بود.

جاسوسی صنعتی از مسیر کارکنان نیز رخ می‌دهد.

در پروندهٔ Micron، وزارت دادگستری آمریکا افراد و شرکت‌هایی را به سرقت اسرار فناوری حافظهٔ DRAM و انتقال آن به فوجیان جین‌هوا[۷] متهم کرد. لین‌وی دینگ[۸]، مهندس سابق گوگل، نیز در ژانویهٔ ۲۰۲۶ به اتهام سرقت اطلاعات محرمانهٔ زیرساخت هوش مصنوعی و انتقال آن‌ها به‌نفع فعالیت‌های تجاری مرتبط با چین محکوم شد.

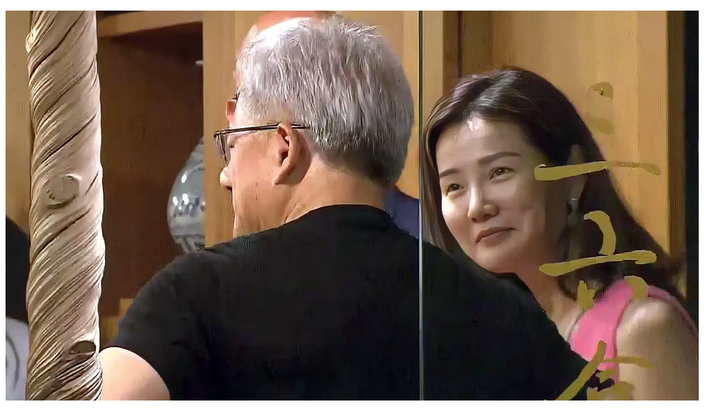
این اطلاعات شامل معماری پردازش، تراشه‌های اختصاصی و سامانه‌های آموزش مدل‌های هوش مصنوعی بود.

پیامدهای بازار نیز چشمگیر بود.

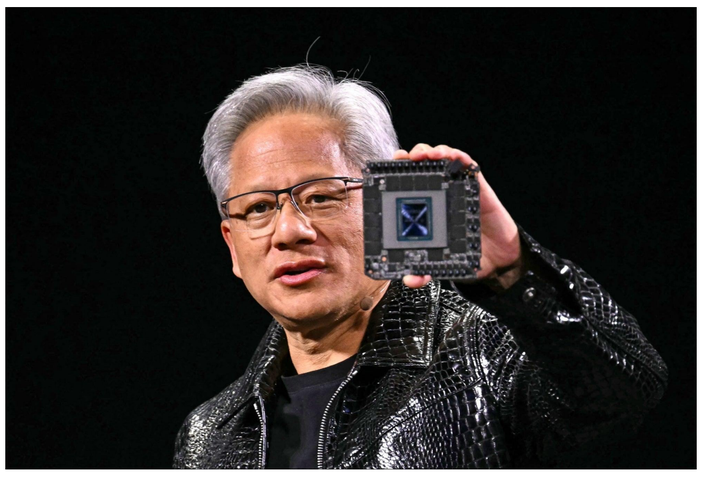
اعتماد به مدیریت و امنیت اطلاعات بخشی از ارزش اقتصادی شرکت است و فقط درآمد، فناوری و سهم بازار اهمیت ندارند. هرگونه تردید در این حوزه‌ها می‌تواند اعتماد سرمایه‌گذاران را تضعیف کرده و پیامدهای مالی گسترده‌ای به همراه داشته باشد.

پس از انتشار خبر تحقیقات، سهام انویدیا در ۹ نوامبر حدود ۴٫۳ درصد افت کرد و نزدیک به ۱۹۰ میلیارد دلار از ارزش بازار آن کاسته شد. در روز بعد نیز فشار فروش ادامه یافت. در ۱۲ نوامبر، پس از گزارش‌هایی درباره آیندهٔ مدیریتی هوانگ و ادعاهای رابطه‌ای دیگر، سهام شرکت در معاملات خارج از ساعت رسمی حدود ۸٫۶ درصد افت کرد.

این افت‌ها را نمی‌توان صرفاً به رابطه‌ای شخصی نسبت داد؛ نگرانی دربارهٔ مدیریت، محدودیت‌های تجاری، دعاوی سهام‌داران و ارزش‌گذاری شرکت‌های هوش مصنوعی نیز نقش داشتند. هوانگ معمار استراتژی انویدیا است و تردید دربارهٔ موقعیت او می‌تواند بر انتظارات سرمایه‌گذاران اثر بگذارد. هنوز میزان انتقال اطلاعات یا وقوع عملیات جاسوسی سازمان‌یافته روشن نیست؛ اما پرونده نشان می‌دهد در رقابت فناوری، روابط انسانی و زنجیرهٔ تأمین نیز در کنار سرورها و کدها اهمیت امنیتی دارند. همین احتمال نیز برای شکل‌گیری بحرانی هم‌زمان در مدیریت، امنیت و بازار سرمایه کافی بوده است.







۱. Jensen Huang
۲. Alice Huang
۳. Stasi
۴. Bernard Boursicot
۵. Shi Pei Pu
۶. Christine Fang
۷. Fujian Jinhua
۸. Linwei Ding

با این سرعت پیشرفت هوش مصنوعی، هنوز یاد گرفتن برنامه‌نویسی ارزش دارد؟ یا کافی است یاد بگیریم چطور با AI حرف بزنیم؟

Claude Anthropic

بله، هنوز ارزش دارد، ولی هدف یادگیری عوض شده.
AI کد خوب می‌نویسد، اما کسی باید بفهمد چه چیزی را می‌خواهد، خروجی درست است یا نه، و چرا خراب شده. بدون دانش برنامه‌نویسی نمی‌توانی ایرادهای ظریف را ببینی (مثل همان مثال خسارت و علامت مثبت/منفی که بی‌صدا گزارش را خراب می‌کند)، و در پروژه‌های جدی به بن‌بست می‌خوری.
پیشنهاد من: اصول را یاد بگیر (ساختار داده، منطق، دیتابیس، دیباگ، خواندن کد) و AI را مثل یک همکار سریع به کار بگیر. «حرف زدن با AI» به‌تنهایی مهارت پایداری نیست، چون ابزارها هر چند ماه عوض می‌شوند، اما درک مبانی می‌ماند.

GPT OpenAI

قطعاً هنوز ارزش دارد؛ اما روش یادگیری برنامه‌نویسی باید تغییر کند.
کسی که فقط کد می‌نویسد، به‌راحتی با AI جایگزین می‌شود؛ اما کسی که مسئله را می‌فهمد، راه‌حل طراحی می‌کند و خروجی AI را ارزیابی می‌کند، ارزشمندتر خواهد شد.
آینده متعلق به کسی نیست که فقط برنامه‌نویسی بلد است یا فقط پرامپت‌نویسی؛ متعلق به کسی است که هر دو را بلد باشد.

Gemini Google

بله، هنوز کاملاً ارزش دارد.
هوش مصنوعی نوشتنِ کدهای تکراری و ساده را بر عهده گرفته، اما همچنان به یک مهندس نیاز دارد تا سیستم را طراحی کند، باگ‌های پیچیده را پیدا کند و تشخیص دهد کدی که AI نوشته اصلاً درست و امن است یا نه.
یاد گرفتنِ «صرفاً حرف زدن با AI» (پرامپت‌نویسی) مثل یاد گرفتن رانندگی با ماشین اتوماتیک است؛ کار را راه می‌اندازد، اما بدون دانستنِ اصول مهندسی و برنامه‌نویسی، در پروژه‌های واقعی و پیچیده خیلی زود به بن‌بست می‌خورید. برنامه‌نویسانی که با AI کار می‌کنند، جایگزین برنامه‌نویسانی می‌شوند که با AI کار نمی‌کنند؛ نه خودِ AI به تنهایی.

DeepSeek DeepSeek

بله، هنوز ارزش دارد؛ ولی شکلش عوض شده.

- فقط «حرف زدن با AI» کافی نیست؛ چون باید بتوانی خروجی را بفهمی، تست کنی، دیباگ کنی و مسئولیتش را بپذیری.
- برنامه‌نویسیِ ارزشمندتر می‌شود: تفکر سیستمی، حل مسئله، طراحی، خواندن کد، امنیت، داده و تست.
- بهترین مسیر: یادگیری اصول برنامه‌نویسی + مهارت کار مؤثر با AI.

خلاصه: AI جای برنامه‌نویس را نمی‌گیرد؛ برنامه‌نویسیِ بدون AI را ضعیف می‌کند. هر دو را یاد بگیر.

بایت را ورق بزنید؛
این بار در فضای دیجیتال.

byte-mag.ir

هر سؤالی داری بپرس...

هوش مصنوعی هم ممکن است اشتباه کند. فکر کردن را به آن نسپارید.

STUDENTS' SCIENTIFIC CHAPTER
انجمن علمی دانشکده مهندسی کامپیوتر
دانشگاه صنعتی شریف

byte