V8 Isolates and Isolated-vm

Linnea Dierksheide, Claire Furtick, Genevieve Flynn

First, it's helpful to explain what Node and V8 are. V8 is Google's Javascript "engine," according to its own definition. An engine in this context is a system that runs other code. As a Javascript engine, V8 takes in JS code and compiles and executes it. This involves handling the order of execution, managing memory, and providing the data types and objects necessary. The execution order is controlled by the call stack and memory is allocated on the heap, which is managed by a garbage collector. Importantly, it provides the Javascript flow of a single threaded application, though this doesn't mean the C++ implementation of V8 is single threaded.

Next, Node is self-defined as "a Javascript runtime which runs on the V8 engine." The runtime is the environment your code runs on while code is running. Javascript is a language designed to be run on the browser -- on the client-side. Node.js allows you to write Javascript code and run it on the server side instead. Node also provides some tools and libraries to make this easier. Essentially, to run server-side JS code, you'd first need to run Node, which uses V8 to execute your code.

The most common users are web servers. Millions of sites want to run their server-side JS code to service user requests, and these need to also have low latency. In the advent of serverless computing, there has been a need to service many tenants on the same hardware. Since these tenants don't trust each other, they need to be isolated. This led to innovations like Virtual Machines and containers. However, running an entire VM, which involves a virtual kernel and userspace specific to each tenant can be slow and consume a lot of space. Containers can be faster as they share the kernel, but both these options can still be slow compared to Isolates, which share both the kernel and components in userspace.

This is where Node Isolates come in. Isolates facilitate isolation within a single process. They could be described almost as "sub-processes," but each isolate has separated memory from the others. Because they don't require starting a new process, isolates can be created and disposed of extremely quickly. Cloudflare, which uses Isolates as its preferred Serverless technology, measures that an Isolate can start up within 5 milliseconds. And, since switching between them doesn't require the system calls to context switch to a different process, that can also increase throughput. The memory isolation essentially gives each Isolate its own heap space in memory, and is prohibited from accessing the heaps of other isolates.

Essentially, Node Isolates exist to run server-side Javascript code with heightened security and speed.

Modules and Abstractions:

V8 isolates are the main modules of this system. There can be multiple isolates within a process. A thread can execute multiple, one at a time, and an isolate can only be executed by one thread at a time, that is, if one thread is executing an isolate, no other thread can execute this isolate until the current thread's execution of it is done. Once it is done, that thread can move on and execute another isolate, and some other thread can start a new execution within the initial isolate. The user of v8 isolates must ensure this synchronization of isolate execution by multiple threads, and they can do so by using the Locker and Unlocker APIs also provided by v8. As the documentation states, each isolate contains per-thread data (line 419 of isolate.cc finds or allocates this data structure). Each isolate has its own chunk of memory within the process's memory space that cannot be accessed directly by another isolate. Each isolate is allocated by a new unique IsolateAllocator. Isolate::New creates a unique pointer to an isolate allocator, and then sets an isolate pointer equal to the isolate allocator pointer cast as an isolate pointer. Within the IsolateAllocator() function, it then calls InitReservation(), which actually finds and reserves the memory for an isolate. These reserved pages are then committed and the isolate's isolate_memory_ pointer is set to point to this memory chunk. These functions are the main aspects of creating the main isolate module.

Other modules of the system are the heap and the read only heap. Isolates have both a heap and a read only heap (line 1725). The isolate's heap is used for the dynamic execution of javascript code within the isolate, whereas the read-only heap contains read-only references to objects/artifacts that an isolate may require in order to execute. Multiple isolates can have access to the same read only heap, also known as a shared read only heap. Therefore, this is a way for isolates to share artifacts, which could include common javascript libraries and APIs (such as the HTTP implementation). This would also make the system faster because instead of compiling all javascript libraries within each individual instance of the v8 engine (each isolate), isolates can share a single instance of the compiled libraries. This concept of isolates sharing read only spaces of memory does not jeopardize the security because each isolate can only read from this shared read only heap.

The system also contains Contexts. Contexts are sandboxed execution environments for running Javascript code. Therefore, without a context, no code can be run within an isolate. Because there can be more than one context per isolate, each isolate contains an unordered list of contexts that it has access to. A Script, which is the object for a

script of Javascript code, can only be compiled and run within a Context; either the currently executing context or another context that is not currently running can be specified.

When learning how isolates interact with each other, we studied the methods provided to isolates in isolate.cc that may have to do with interacting with other isolates. There is also a method in the v8 isolate class called Isolate::RequestInterrupt that takes in a callback() and data() and pushes it onto an api interrupt queue. There is also the Isolate::InvokeApiInterruptCallbacks() method which continuously pops interrupts off of the api interrupt queue. Therefore, we can see that if an isolate can have a reference to another isolate, then an isolate could add an interrupt to another isolate's api interrupt queue or invoke another isolate to execute the interrupts within its queue.

Outside of the specific v8 APIs, we also studied isolated-vm, which is a Javascript library for isolates. Isolated-vm provides an IsolateEnvironment API which packages together the isolate, the scheduler, and the isolateHolder, which helps with managing the tasks. The isolated-vm scheduler class allows an isolate to have a unique scheduler that tracks its tasks and the isolates' state. Overall, the IsolateEnvironment class/API facilitates interaction between multiple isolates via locking, scheduling and executing tasks.

Target Domain:

Node isolates target the domain of running server-side Javascript code, which is generally web servers. Companies that provide serverless infrastructure would find it valuable, because they need to service many independent tenants with as little latency as possible. Being able to run server-side Javascript via Node.js is already valuable as most web developers can easily apply their knowledge from client-side programming to work on the server-side as well. The enhanced security of knowing that code is being run in an Isolate makes an even better case for Node.js as a server option.

To start, Isolates are of course valuable over no sandboxing at all, because that would either mean that there wouldn't be any isolation between operations within a process or that each tenant would need its own process. Being able to house multiple tenants in one process leads to lower latency. Starting up a new Isolate is much faster than starting up a new process. This is crucial for systems with frequently idle tenants as it solves the issue of "cold starts." However, in systems that need to service a few tenants with high and constant resource requirements, Node Isolates wouldn't be as innovative.

Therefore, this system is viewed more favorably by those looking to create serverless “at the edge” technologies. Compared to containers and virtual machines, which are favored by the current leaders like AWS, Isolates are much lighter, especially in terms of memory. Since isolates don’t require nearly as much memory (no virtualized hardware, split the per-process costs such as the virtual address space), a single computer can handle many more tenants running isolates. Another upside is that because many Isolates are running within a single process, the system spends much less time context switching between processes -- which requires system calls down to the kernel -- in order to switch to servicing a different tenant.

Cloudflare, one of the biggest advocates for Isolates, found that it was able to run code for all of its tenants in every one of its locations, rather than just in one or two locations, which is the option offered by AWS. Cloudflare is differentiating its product for edge computing applications and systems, which aim to bring processing close to the location in which it is needed in order to achieve the quickest response times. Thus, although some of Cloudflare’s locations don’t have as many hardware resources as a typical datacenter, they can still service all of their tenants because Isolates are so lightweight. Most systems today prefer process isolation between tenants, but Cloudflare trusts the safety of v8’s internal memory management support.

However, being built on a system made for Javascript means that Isolates can’t run just anything. They can only efficiently compile and run Javascript (there is support for WebAssembly languages, but it comes with more downsides), so a system that needs anything else wouldn’t see Isolates as a worthy replacement for a general purpose VM. A system that needs to directly access system resources like files and devices likely wouldn’t find Isolates particularly useful either. Also, Isolates share both processes and all underlying system resources, which could lead to security breaches between tenants that may not happen with VMs, which only access the shared hardware via the hypervisor.

Overall, systems that are extremely sensitive to latency, want to save memory, and run Javascript could benefit, but it is far from a replacement for the general use case provided by virtual machines.

Performance:

One of the biggest factors that make iIsolates appealing is that they and v8 in general perform very well when compared to similar technology, Compared to traditional containerized processes that are popular among serverless technologies like AWS Lambda, isolates are able to spin up hundreds of times faster due to their design.

Isolates perform incredibly fast compared to more traditional tech like VMs and containers because they need to bring less to the table and can share more resources. VM's are going to be slow because they need to boot up their own kernel and operating system which take up a lot of space and time to configure. Containers are a step more optimal because now individual containers can share the operating system kernel but still provide everything else including code, libraries, environments, etc. Isolates are optimized even further because they can share the web platform APIs, the JS Runtime, the Operating System, hardware, and all they need to bring is the application code and uncommon libraries. This makes their boot up time a lot quicker.

Isolates are designed to run within a process which allow them to avoid the overhead associated with context switching which can take up to 100 microseconds. When added onto Node running on something like an AWS Lambda, this is only amplified. Isolates are also designed to run in one thread at a time, which make Isolates ideal for when running completely separate applications. This design suffers when the client wants to use isolates for performing operations similar to multithreaded applications. For example, a client could want one isolate thread and multiple worker thread isolates. This setup would not be feasible because they cannot access each other's data easily. There are workarounds by making data transferable, but they would be less resource efficient and would be difficult for complex data types. Contexts would then be a viable alternative in this case as isolates can have multiple contexts per isolate, but they would still be limited by the single-threaded execution design of JS.

Isolates are designed with asynchronous and synchronous operations in mind. Because of the asynchronous design, isolates see good performance when running idle connections in the background. On the other hand, if one process takes up too many resources with many operations it can crash the whole system.

Isolates have good baseline memory usage because they tend to take up less than a megabyte of size. This allows many isolates to be run on a single server with very low latency. Bad performance can come from having a lot of dependencies needing to be shipped with the program as this can take up more resources. Also, there is the potential for DDOS attacks from one client spamming isolates. This strain on resources will slow the whole system.

Focusing on v8 in general as opposed to isolates, the performance of the compiler is notable as well. It has improved markedly since its original design. This is important because fast compilation is crucial to web servers. How v8's compiler architecture will work is it will first compile code and then put this code through an optimizer. The design of v8's original optimizing compiler worked well for long running programs, as it would

stop once every millisecond to perform optimizations. This was not a big deal over a long period of time but for short running programs this does not yield much benefit. This algorithm was altered to use counters instead of time based optimization, and optimizing more frequently used functions which lead to a 25% increase in speed.

Optimizations:

Isolates and v8 as a system use optimizations in a clever way to make code execute smoother. Isolates as an abstraction is already a form of an optimization in comparison to VM and containers because of low process overhead (avoiding context switches, etc). Because an isolate is an instance of the v8 engine, optimizations will focus on the core aspects of that system which include the compiler architecture, inline caching, pointer compression, and memory structures.

The v8 compiler architecture has certain optimizations designed to make code execute faster (Just in Time compilation). It is split into three different layers; frontend, optimization layer, and backend. The frontend is responsible for the generation of bytecode. The optimization layer will improve code performance and the backend layer performs lower level tasks like machine-level optimization, scheduling, etc. The *Ignition* interpreter for the frontend will generate the bytecode and then *Turbofan* is the optimizing compiler which will try to make the code run as fast as possible.

v8 as a compiler tries to guess which functions will benefit from optimization, and uses this to inform on when to optimize and oftentimes v8 can make different decisions in each runtime. While the optimizing compiler is normally able to guess correctly, occasionally it is misinformed and has to deoptimize. This is an example of an isolate deoptimizing a specific frame's chain of events. Speculative optimizations are very costly and deoptimization results in terrible performance implications. To fix this, v8 improved the interpreter of bytecode to be better for performance and the new compiler would focus on inline-optimization stages for bytecode. Inline-caches, another optimization to be touched upon, were part of the reason this was successful.

The two compilers have different use cases where they would have good performance. Ignition is ideal for code executed at page load and rarely used application code. This is because it is running byte code and will not be as fast. On the other hand, TurboFan will optimize the compiler code. This is better suited for more frequently used code as time is crucial and for code after page load as response rate is important.

Optimizations which pair well with the compiler is inline caching. Inline cache is used to memorize information on where to find properties of objects to reduce lookups. More

specifically, when loading scripts into isolates it can now check to see if the script is already stored in the isolates hash table and returns the compiled bytecode if so saving compilation time. This is effective as it has a nearly 80% hit rate. Looking into the implementation, IC's have states (IC::State) which keep track of the state of that data in the cache. Then depending on this, the isolate can do a lookup on that data.

Another technique specific to reducing memory usage is pointer compression.There are two aspects to pointer compression, making sure objects are allocated within a 4 GB range and representing pointers as offsets within this range. With the use of pointer compression, v8 can operate like a 32 bit application with the performance of a 64 bit and this is also able to reduce heap size within isolates by up to 48%. When initializing an isolate, you can enable pointer compression through the kPointerCompression variable. Throughout v8, it will then make decisions based on the pointer mode with better performance for less memory usage.

Optimizations include isolates having references to shared data, for example the heap constants. The root table within the heap of the isolate, will point to the root objects. This way, separate contexts within the same isolate can still access the same roots. Contexts are restricted to their isolate and cannot access contexts from other isolates. This makes isolates ideal for operations where you need separate contexts referencing shared data and possibly communicating.

In general, there are also tips for getting isolates to run more optimized and these have to do with understanding how Javascript functions. First, the less data types the compiler has to deal with, the faster the code will be. Javascript classifies functions as monomorphic vs polymorphic, etc. depending on the number of different types and less is better in this case as the compiler does not have to worry about complex variables. Also, JS runs better with more modularized code as it is easier for the compiler to optimize. Clients should avoid large functions if possible.

Isolates are limited in that they can only deal with either Javascript or languages that target Web Assembly. Even still, when shipping languages other than JS you have to add back in some of those resources you optimized to share, so language runtimes, etc which can start to use up a lot of resources and make the execution slower. "If users cannot recompile their processes, they will not be able to run them in an isolate".

To understand why web assembly is slower than Javascript, it's because v8 ships a JS to WASM converter and this running adds a lot more complexity and execution time than a normal isolate. Isolates when dealing with scripts will now have to check if the script is running JS or some other assembly language. The function isJavascript()

conveniently does exactly as described. Stack frames within the isolate will keep track of the state of the code being compiled and if it is wasm. These types of isolates will not be as optimized and highlights some of the limitations associated with less resource heavy isolation.

Security:

The main security property is that V8 isolates have memory isolation via separation of the heap. Within the isolate class, there is the pointer to the heap and to a ReadOnlyHeap, a separate data structure. When the isolate is initialized via Isolate::Init(), both heaps are explicitly set up. Setting up the heaps shows an important difference between the types: read only heaps can be shared between Isolates (which is a feature associated with pointer compression?). However, we don't see this as a significant security risk as they are exclusively ready-only -- if an Isolate wanted to compromise another via this heap, it wouldn't be able to write in any changes that could affect it and the read only heap doesn't point back to an isolate's memory.

In the Heap::SetUp function, a new memory allocator object is created for this specific isolate. The MemoryAllocator may be considered the reference monitor for the system, since it takes memory from the Operating System (as a Space object?) and coordinates allocation and deallocation of pages to specific heaps. Hence, it makes sense that when a heap is set up, it needs an associated memory allocator so that its memory can be tracked. The memory allocator API brings together the heap, MemoryChunks, SpaceTypes, Executability, ReadOnlySpaces to bridge the gap between the backing data stores and the isolates and their memory. And because V8 provides garbage collection as well, this becomes complicated.

This relates to isolates in a bit of a roundabout way. An isolate becomes "activated" when a thread "switches" to it, although because this isn't a traditional context switch, it's called entering. In the Isolate::Enter function, it gets the data that essentially ties together the currently executing thread and the isolate it's executing in. It first checks if the thread is already in the isolate to enter -- in which case nothing changes -- but if not, we either get or allocate the PerThreadIsolateData for this new combination of the two data structures and push the new information onto the EntryStack that tracks the movement of entered/exited isolates. Then we set the ThreadLocals to update the Isolate pointer within the thread local variables. Through this, we can access the isolate_key and the heap_ pointer in the isolate, and the list of allocated pages within the heap.

In the initialization of an Isolate, the CreateParams type argument allows specification of resource constraints, including a maximum amount of memory that can be allocated. Then when making allocations, V8 keeps track of the allocations made to ensure that the isolate is disposed of if it exceeds the limit. In isolated-vm, a Javascript wrapper built around V8 Isolates, the allocator first checks that it won't exceed the maximum memory before doing the allocation, via use of the GetHeapStatistics() V8 function. Within the isolated-vm library, the IsolateEnvironment class could be considered a reference monitor, since it tracks the isolates, their memory, and the ownership of references and values. Isolated-vm has a focus on transferring data between isolates, and this requires explicit ownership checking; for example, it allows a reference to a value to be shared between isolates, but only the isolate that owns the reference can dereference it -- it uses the GetCurrentHolder() function from IsolateEnvironment to properly restrict this access. Isolated-vm also provides locking in order to restrict concurrent access to data when necessary, which enhances security between isolates as well.

Beyond the specific V8 code associated with Isolates and the isolated-vm library, we would be remiss to not discuss the overall security principles in the system, both as they apply specifically to isolates and to V8 and Node.js in general. V8 has over two million lines of code, which likely violates the Economy of Mechanism principle that values simplicity, given that V8 often adds complexity to marginally improve performance. However, memory isolation within a single process adds security in terms of Least Privilege and Defence in Depth -- separation even inside a process makes it harder for an attacker to reach the process layer or below from an Isolate. However, because there isn't process isolation between tenants, there are more underlying shared resources between Isolates (Least Common Mechanism).

Given V8's complexity and wide use, Google offers high rewards (up to $150,000) to incentivize finding and reporting security exploits in its products. It has invested a lot of resources and time into finding ways to minimize Spectre attacks in the last year, mainly by increasing depth and making it harder to exploit their system rather than eliminating the possibility of attacks.

A 2012 paper which assessed Node.js security, when the platform was still quite young, points to the lack of sandboxing within server processes and the many "gotcha" language features of Javascript as downsides to the platform. These language features in particular, can confuse developers and make code difficult to easily verify, which can inadvertently lead to vulnerabilities. On the other hand, some of the issues described have since been addressed, like Promises to solve the "callback soup" described and Isolates as a sandboxing mechanism within processes.

Subjective Thoughts:

Overall it's an interesting innovation. V8 is practical because most of its "users" don't have to know anything about it when writing and running their javascript code, but it has incredible power and complexity. Javascript and serverless tech are both very popular among developers, so the way that Cloudflare has combined both using Isolates as the core technology is really interesting. I think it has potential if edge computing becomes more viable, but at the same time, I doubt it will catch on as a primary serverless option. This is because resources are still heavily shared, security is becoming more important to individual tenants, and it can limit an application's memory. And despite Javascript's popularity, it's limited by the fact that it primarily targets JS. It has potential as an option for lightweight sandboxing for companies that want to run a lot of code from many users -- I think platforms like Leetcode or Screeps could (continue to) utilize isolated-vm to improve their isolation. It would be interesting to consider whether the concept of isolates could be applied to another scenario that is not necessarily instances of v8, but some other service that requires simultaneous execution of multiple isolated computations.
To be terse, it's really cool for the niche it serves.

Who did what:

Linnea: Summary, Security, Target Domain
Claire: Modules/Abstractions
Genny: Optimizations, Performance
All: Code analysis, editing/helping on report, presentation