EPISODE 784

[INTRODUCTION]

[0:00:00.3] JM: WebAssembly is a runtime that lets languages beyond JavaScript executes in front-end web applications. WebAssembly is novel, because most modern front-end applications are written entirely in JavaScript. WebAssembly lets us use languages like Rust and C++ after they have been compiled down to a WebAssembly binary module. Of course, language interoperability is only one part of why WebAssembly is exciting.

The execution environment for WebAssembly modules has benefits for security and software distribution and consumption as well. In previous shows, we've given an overview of WebAssembly and explored its future applications, as well as its relationship to the Rust programming language. In today's episode, we explore the packaging and execution path of a WebAssembly module and some other applications of the technology as well.

Syrus Akbary is the CEO and Founder of Wasmer; a company focused on creating universal binaries powered by WebAssembly. Wasmer provides a way to execute WebAssembly files universally. Syrus joins the show to talk about the state of WebAssembly and what his company is building.

Before we get started, I want to mention that there's a new product I'm building. It's called Find Collabs and we are having a hackathon for Find Collabs. You can go to findcollabs.com and findcollabs.com/hackathon to find out more. Find Collabs is a platform for internet collaboration and it's built for people like you, the Software Engineering Daily listener. I really hope you'll have a chance to check it out. With that, let's get on with this episode of Software Engineering Daily.

[SPONSOR MESSAGE]

[0:01:51.2] JM: HPE OneView is a foundation for building a software-defined data center. HPE OneView integrates compute, storage and networking resources across your data center and leverages a unified API to enable IT to manage infrastructure as code. Deploy infrastructure faster. Simplify life cycle maintenance for your servers. Give IT the ability to deliver infrastructure to developers as a service, like the public cloud.

Go to softwareengineeringdaily.com/hpe to learn about how HPE OneView can improve your infrastructure operations. HPE OneView has easy integrations with Terraform, Kubernetes, Docker and more than 30 other infrastructure management tools. HPE OneView was recently named as CRN's enterprise software product of the year.

To learn more about how HPE OneView can help you simplify your hybrid operations, go to softwareengineeringdaily.com/hpe to learn more and support Software Engineering Daily.

Thanks to HPE for being a sponsor of Software Engineering Daily. We appreciate the support.

[INTERVIEW]

[0:03:13.8] JM: Syrus Akbary, you are the Founder of Wasmer. Thanks for coming on Software Engineering Daily.

[0:03:18.5] SA: Thank you for inviting me. I'm super excited to talk today.

[0:03:21.4] JM: Most of our web experiences today are powered by JavaScript. What are the types of applications where JavaScript does not perform well?

[0:03:30.9] SA: Basically, especially when browsers are trying to render certain games, or certain operations where actually the JIT of the browser is able to perform not very well, I will say the cases are games for browsers and some other password, actually like the JIT cannot optimize as much as you can actually with a native application.

[0:03:55.1] JM: JavaScript has a dynamic type system, it has garbage collection. Why did these features of JavaScript result in performance issues?

[0:04:04.0] SA: In general, I think optimizing JavaScript is a little bit hard. Basically, browsers have been trying to optimize as maximum the JavaScript performance over the last few years and they have been doing a very, very great job. At the end, just because of the fact that first, JavaScript is not typed and second, because the JIT have to guess what is the types of the variables that we are using, like this adds a lot of overhead that at the end is very hard to optimize to the max, because you have to assure always backwards compatibility, or some other things that have to work always with JavaScript. Sometimes it's very hard to make a shortcut there, while in native where everything is typed, is much easier to do these optimizations.

[0:04:49.2] JM: We've covered the differences between JavaScript and WebAssembly in some details in past episodes, but I wanted to start with just some discussion of JavaScript and WebAssembly. Can you give a little bit of history as to how the WebAssembly project got started and what its relationship is to JavaScript?

[0:05:09.4] SA: First, it was started by asm.js, which was a very interesting project that actually was trying to do is compiling or transforming LLVM IR to JavaScript. What LLVM IR is basically when we have an 80 project, let's say made in C or C++, rather than compiling it directly to machine code, what happens is this can apply to C or C++ project gets transformed into a level LLVM intermediate representation.

Then from there, it actually got ported to move to real machine code. Actually, what it helps is other an abstraction on top of the chipset. Basically asm.js how it started is by transforming this LLVM IR to JavaScript, it was doing in a way that actually it add fake types to JavaScript. Basically, it was indicating like, “Oh, this variable is an integer, or this variable is binary.” It was having them type some on top of JavaScript.

Basically, you’ve got some inertia from Firefox, especially at the beginning that was adding a special handling for these types to optimize it much more. It started with this asm.js project, which actually eventually become what WebAssembly is today, but with the main difference that asm.js was a subset of JavaScript, a type subset of JavaScript. Basically, WebAssembly was rethinking of this model, that rather than being on top of JavaScript, that the browser have to parse. It was actually a bytecode that was much more performant to read for the browser.

[0:06:51.7] JM: WebAssembly has these applications in the browser, but it also has other applications. Before we dive into the technical details of WebAssembly, let's discuss some of the impact that WebAssembly might have at a high level. If we were to fast forward five years into the future, how will WebAssembly have changed our online experience?

[0:07:13.9] SA: I think it will change a lot of things from online right now. The applications we are seeing on the web are basically made with HTML layout, which at the end is great, but sometimes it's hard to think that we will be able to run Microsoft Office, for example, natively on the browser, or with exactly the same experience of the native application, or even running super performant games on the browser.

I think from five years to now, we'll start seeing a lot of more reach applications on the web and actually, not just in the web. One of the sides of WebAssembly is I think it will improve really the way we experience web in the future, just basically trying to bring the quality of native applications, or the speed of native applications back to a browser. I think of the browser are just one site that is taking that advantage of WebAssembly. There are a few others I'm super excited about.

[0:08:12.7] JM: What does WebAssembly enable for "edge computing?"

[0:08:17.4] SA: Basically with WebAssembly, first what do you have is a very nice abstraction that lets you not worry about what is the chipset, where your application is going to be run. This actually have a lot of advantages for edge computing. The main one is basically you have a binary, or a bytecode that can be actually executed anywhere in any platform.

Second, because of the way WebAssembly is designed, we have first memory solution from each of the process. In general, WebAssembly instance that I cannot jump out of their memory. That's one very important aspect of WebAssembly. The other very important aspect is basically we can run this code, or this bytecode almost, or as close as possible to native speed.

When you mix these two things, you start seeing that we can actually right now stop using services, or programs like Docker that provides all this memory solution and we can just jump directly to execute this native code at native speed, but with all the advantages of memory solution for each process.

[0:09:29.3] JM: Could you explore those differences between packaging our programs in Docker containers, versus packaging our programs in WebAssembly in a little more detail?

[0:09:38.9] SA: Yes. Actually, this part is something that we are very excited about in the company that I am working. Basically right now, the way we deploy services, or we deploy our applications to the cloud just in general like using Docker, which basically how it works is we have a base operating system that could be Ubuntu or could be an Alpine distribution. Basically, on top of this operating system we add our applications and Docker basically provides a container that in general can be 400 megabytes, 500, or few gigabytes, or if you are using Alpine, will be much lighter. In general, we always have to ship the operating system that the application is going to use.

Then we ship the application as well with this operating system. With Docker, we will end having containers that again, are very big. Because we include all the operating system there, right? However with WebAssembly, that's not really necessary. That's because of the ways WebAssembly is architected. We don't need to be emulating our operating system on top of WebAssembly to provide this memory solution safetiness.

Basically, I'm going to give you a very good example right now. For example, if you are using Docker and you have a website which receives let's say 20 bits is enough, right? You will need to run a Docker container, or an instance of Docker container. You cannot for a home off, just being there like listening to any request that gets on for that server on that port, even if this server just receives 20 requests per month.

That means you will end paying for a full month and that's again not super optimal. In the case of WebAssembly for example, what we can do is rather than having an instance that this is running constantly, we can have container that rather than being 400 megabytes, it's maybe 4 megabytes that can actually be spin up and spin down in a very fast way.

Basically right now, we Docker this startup time in general is 1 second, again container size maybe few hundred megabytes. With that, like with this current solution is very hard to think of functions necessary and there are few hacks that are trying to make it work with hot paths for Docker and so on. In general, there is not a very good solution to do it with Docker, just because Docker is not the ideal technology for that.

However with WebAssembly, what we can start seeing is applications that are contained in a 4 megabyte binary, and at the same time that have a startup time of few milliseconds. What this enables is some premise computing. That means spinning up and spinning down instances as we need it. Rather than having an instance that is running for a whole month just for 20 requests, for serving 20 requests, with WebAssembly, what we can do is just spin up this server as we need it. Basically, it will maybe eventually charge just 20 seconds or something like that and this is actually something much more performant and that use much better the resources that we have.

[0:12:57.4] JM: You're describing a use case for WebAssembly for powering functions as a service. The current model as it's commonly understood at least is that if I want to run a function, a “function as a service,” or AWS lambda function, or Google Cloud function is I write my code for the function, I deploy it to the cloud provider. When I want to trigger that function, the function gets spun up in a container on the fly and then it gets executed and then maybe gets spun down after some period of time. You're suggesting that rather than deploying that function to a container, we could just bundle it into a WebAssembly binary and have it execute that way. Do I understand correctly?

[0:13:41.3] SA: Yeah, that's completely right.

[0:13:43.2] JM: Just to reiterate, what would be the benefits of having our function execute in a WebAssembly binary versus a container?

[0:13:49.9] SA: First, it will be much faster. Second, WebAssembly already provides a memory solution, so we will not need to worry about that. Third, part of being faster or being as close as native performance as possible is that we will be able to reproduce that super easily locally as well.

In general, right now, all the solutions that for example like Google Cloud or in general, I will say more Amazon lambda, which is what is closer to the web sampling model, is just useful or it's only available on Amazon Web Services. What we are trying to do is have something open or provide something open that everyone can use not only in Amazon Web services, but almost in any other server. In general, by leveraging on WebAssembly, we can start providing new tools like that and that was very hard, or basically was using the tools that were not ideal for data scenarios.

[0:14:51.3] JM: Let's come back to this question of WebAssembly in the cloud. I want to talk a little bit more about the basic WebAssembly tool chain, or the different tool chains that are out there, because WebAssembly is commonly known as this browser, this thing that you might run in the browser to have modules that you might want to run faster than you would run a JavaScript script, or module. Let's talk about that.

If I have a WebAssembly module that I want to run in the browser, maybe it's going to power my game or render something, what's the interaction between my JavaScript code and my WebAssembly module?

[0:15:34.5] SA: Basically first, what do you do is in general when you are trying to create this WebAssembly module, you either do it in a static language like C or Rust. Then basically, you will compile it from Rust or C to WebAssembly. For example, WebAssembly support in Rust is very well prepared. You see like you will in general need to use Emscripten. Let's say you have this WebAssembly binary that execute this function in a much more performant way, then what you will do is from WebAssembly – of like a binding to use it in JavaScript.

In general, this binding if you are using Emscripten will be already provided for you. If you are using Rust for example, there is a library called wasm-bindgen that will again provide this very easy way to use WebAssembly from JavaScript in icy way. The reason for doing that is for example, sometimes your function will accept the strings, but in WebAssembly in the specifications itself, there are no strings.

Basically what you need to do is transform from this JavaScript string into bytes that WebAssembly can understand. In general for this binding, there are a lot of tools that provide us the help. When we want to use this function from JavaScript, what we will do is create a WebAssembly instance. This WebAssembly instance will have 13 functions exported from the WebAssembly module itself. Let's say we create a function in WebAssembly, which is just for having two numbers.

We will call this function sum. This sum function received two parameters, which are two integers. From JavaScript, we will create WebAssembly instance. Once we have that instance, we will say we will try to call the sum function in this instance and basically, we will retrieve the value. Sometimes in the case of integers, we can just use it directly. In the case of strings, we will need, or strings, or other structures, we will need to use some wrapping or transformation from WebAssembly types to JavaScript. Then we will just use it in JavaScript as normal. Is that clear enough to know basically how JavaScript interoperates with JavaScript?

[0:17:46.5] JM: Absolutely.

[0:17:47.6] SA: Yeah, with WebAssembly. Sorry.

[0:17:48.5] JM: Absolutely. Now let's describe how we would run WebAssembly in environments other than a JavaScript execution runtime.

[0:17:57.1] SA: Yeah. Basically until now, what we have been seeing is WebAssembly is mainly shipping browsers. This is awesome because it provides a way for executing files at native speed or params at native speed in the browser. That's something that is super exciting. However, there are a lot of other use cases of WebAssembly, especially outside of the browser.

One of these use cases is actually trying to bring WebAssembly server-side. Trying to think of the same thing that no JavaScript did for JavaScript. Actually in my company was where we are trying to do something similar, trying to bring WebAssembly to the server-side in a way that is actually very easy to use and at the same time that is not tied to a JavaScript runtime.

Basically, the benefits of that is you will not need to run JavaScript runtime. You will not have any overhead from that. Second, you will be able to run WebAssembly not just from JavaScript, but also from Python, or from Rust, or from C, or for simple C++. When you start putting all these pieces of the puzzle together, you start seeing that WebAssembly can become this new standard for having libraries that are compatible in almost any environment, any language, any platform, any from desktop to mobile, from the browser to a server.

Basically, we will start seeing how libraries are just universal. We can create let's say a library in typescript that actually we can use it from Python. I think that's something super, super attractive. I can explain why if you want there, a little bit after.

[0:19:41.1] JM: I mean, I can why. For today, if you want to have let's say a set of tools for doing natural language processing in Python, if you have that set of tools in Python and you're writing an application in Rust, it's not necessarily straightforward to be able to interoperate between your Rust application and that Python NLP application. You're suggesting WebAssembly as a means to allowing that interoperability.

[0:20:13.9] SA: Completely. WebAssembly basically in there, what it will provide is a way to interoperate between languages very, very easily. Right now again, actually the example that you come into this is super accurate, because we have been seen companies that are they are Python shops and they are using rust, as well for the more critical path of their application. Basically, each time they need to use this Rust code from Python, it's a little bit of a pain and they have to compile it for the architectures where this module is going to be run. Basically, it's super painful. With WebAssembly, basically all these problems are solved.

[SPONSOR MESSAGE]

[0:21:03.9] JM: DigitalOcean is a reliable, easy-to-use cloud provider. I've used DigitalOcean for years, whenever I want to get an application off the ground quickly. I've always loved the focus on user experience, the great documentation and the simple user interface. More and more people are finding out about DigitalOcean and realizing that DigitalOcean is perfect for their application workloads.

This year, DigitalOcean is making that even easier with new node types. A $15 flexible droplet that can mix and match different configurations of CPU and RAM to get the perfect amount of resources for your application. There are also CPU-optimized droplets perfect for highly active frontend servers, or CICD workloads.

Running on the cloud can get expensive, which is why DigitalOcean makes it easy to choose the right size instance. The prices on standard instances have gone down too. You can check out all their new deals by going to do.co/sedaily. As a bonus to our listeners, you will get a $100 in credit to use over 60 days. That's a lot of money to experiment with.

You can make a $100 go pretty far on DigitalOcean. You can use the credit for hosting, or infrastructure and that includes load balancers, object storage, DigitalOcean spaces is a great new product that provides object storage, and of course computation. Get your free $100 credit at do.co/sedaily. Thanks to DigitalOcean for being a sponsor.
The co-founder of DigitalOcean Moisey Uretsky was one of the first people I interviewed and his interview was really inspirational for me, so I've always thought of DigitalOcean as a pretty inspirational company. Thank you, DigitalOcean.

[INTERVIEW CONTINUED]

[0:23:11.3] JM: Let's talk about the current state of affairs. If I want to have my Rust application access a Python NLP application, there are ways for me to do that. What would I be doing and what would be wrong with that?

[0:23:27.2] SA: I mean, right now if you are actually trying to run Python from Rust, you can completely – there are ways we can make it work, but at the end it's not very easy. Actually, I will put it in reverse. If you want to use Rust from Python, again it's something that you can completely do, but first you have to worry about having the rust tool chain whenever you're using this Rust code, then you need to make sure the bridge between the web – between Rust and Python is well done, there are no memory issues in between. Basically, all this process is super painful for the developer. It's basically not ideal.

What we think will be ideal is having some universal binary for each of these libraries that we can use not just from Python, but from any other language and we can interoperate easily. You can think into having – once we transform this library to WebAssembly, it should be super easy to use in one or other language without actually carrying out how to do the bridges between these two.

[0:24:33.5] JM: Of course. That sounds wonderful to everybody. One difference between different languages is different languages have different runtimes. Some of those runtimes have managed memory systems with garbage collection. Some of them have self-managed memory like with C. Let's just take a step back, how is WebAssembly allowing us to have a consistent runtime for these different languages when some of them are garbage collected and some of them are not?

[0:25:04.5] SA: Basically, first regarding garbage collection. Right now, for example Rust projects when we ship or when we transform these or compile these Rust modules to WebAssembly, in general we ship the garbage collector and collection inside of the WebAssembly itself.

Apart from that, from the WebAssembly specification, there are people working on having a Rust collector inside of WebAssembly or bundle into WebAssembly itself. Basically right now when we compile it, the garbage collector will be actually also bundled. The way it will interoperate at least for now is each of these languages will have their own garbage collector. As of right now and there will be not a unified way of collecting garbage for these different languages.

Actually, one thing that we are researching to Wasmer is how we can improve all these strategies to make it much easier and much more performant to do this intercommunication.

[0:26:05.4] JM: If I'm shipping a garbage collector in my WebAssembly binary, why is that more performant or more convenient than just having my Python program run normally and interoperating with my other programs?

[0:26:22.9] SA: I mean, the main advantage that I will say is that the developer experience when you have to interoperate with other languages right now is not ideal. With WebAssembly, what you do is having [inaudible 0:26:34.4] basically ship your library to WebAssembly. It will be super easy to be used in any other language in a very easy and common way across any other language. I will say that's the main advantage of using WebAssembly there. It's not regarding performance. You will actually get almost the same performance, even a little bit worse if you are using WebAssembly, maybe 5%, or 10% of decrease in performance. What you will gain is much better developer experience for using these libraries.

Let me actually put a very concrete example. Before starting Wasmer, I did work on a graphical framework for Python that was actually a part of the JavaScript implementation. Basically, there was the main reference implementation of GraphQL, which was made by Facebook. Each time the reference implementation was changing in JavaScript, I have to basically backport all the changes into Python. This process was super, super, super painful. Why? Because in general, I don't want to embed a JavaScript runtime in my Python applications, and almost no one wants to do that, because they have other implications.
Basically, what I believe or why I actually started Wasmer is because I tried to make a library that will be universal and could be used not just in JavaScript, but also in Python in a way that I don't need to rewrite all these things natively for that language.

[0:28:07.1] JM: What you're saying is the main thing that we're getting out of WebAssembly from this point of view from this conversation is developer experience. You are boiling down a large tool chain into a single executable binary. That binary is a WebAssembly module.

[0:28:25.6] SA: Yes. That's completely right. Then you have a lot of convenience for using it. If you want to even ship this module to the browser and use it is in very performant way, you could. If you want to run it server-side, you could. If you want to run it, like let's say you create that module, which is I don't know, just trying to detect the faces from my image and you do it in Rust.

Basically, what you will do is you compile this Rust code to WebAssembly and ideally, you would be able to use it from Python easily, you will be able to use it from JavaScript easily, or you will be able to use the from Go easily as well. Basically, the goal is to provide only one byte code binary that can be used across any other language very, very easily.

[0:29:11.9] JM: Just to revisit the container discussion earlier, containers give us this nice abstraction for working with a complex application that has been bundled into a container, or if we were looking at from the Kubernetes point of view, maybe we would say a pod which can contain multiple containers and have this bundling abstraction. Why do we need WebAssembly? Just to revisit that discussion of the containers.

[0:29:36.0] SA: Why we need the WebAssembly, first when you are using container, the startup time is not very good. In general, it can be I don't know, in the order of seconds, or at least more than 700 milliseconds. Basically when you have this very long startup time, the way that you have to figure out how to make it work with on-premise computing, it becomes super, super challenging.

First regarding Docker versus WebAssembly, Docker the container images include operating system inside and the containers itself are very heavy, both for a startup time for container size. With WebAssembly, both the time becomes rather than in the order of seconds becomes on the order of milliseconds. Second, the application size decreases from a few hundred megabytes to few megabytes. At the end, it’s much easier to distribute and use.

[0:30:30.9] JM: Got it. Now before we talk about what you're doing at Wasmer, I want to talk about the bundling tool chain and the usage tool chain for Rust, because as you discussed earlier your vision with Wasmer is to make it easy for all kinds of languages to compile down to wasm and be used with wasm. We already have a tool chain for Rust that can be used. Why is the tool chain to Rust so specific? Why isn't that rust tool chain usable for any language?

[0:31:07.0] SA: Basically, this tool chain is very adapted into how Rust is made, or how Rust works. The way this tool chain is specific, I think you're talking about wasm-bindgen, which is very – I mean, first there are two integrations of for being able to use WebAssembly in Rust. One is having a target of WebAssembly into a language. Basically, because of Rust in general uses LLVM under the hood; this is very easy to convert from LLVM to WebAssembly. This the first step, how you are generating the WebAssembly from file from Rust.

This is actually like – this is how you're generating this – you have to be done for certain – for each of the ways that we – or each of the languages that we use. For Rust, it’s targeting to WebAssembly is already bundled into the language. For C, we need to use a side project, such as Emscripten to do it.

Right now, for example for Go, they are trying to have also a way for converting, or for targeting Go, or for targeting WebAssembly in your Go application, so basically you can compile from Go WebAssembly. This first step basically have to be – is very tight – have a very tight integration with the language itself. You cannot create something that actually will work for everyone, a part of the LLVM to work assembly transformation.

On this first side, basically each language needs to provide a way to target WebAssembly and this is the first step. Then the second step is how we can actually make the usage of WebAssembly from this language very easy. In the case of Rust, basically we have the library wasm-bindgen, which is a great library and it help us basically to operate with other structs that are not just integers very easily from the host language that is calling WebAssembly.

In the cases of basically for Python, we will need to create other of wrapper on top of WebAssembly to make very easy to interoperate with it. Basically, from each of these languages you need to do a small transformation from WebAssembly types to the language types. Also these transformations are super tied into the language that we are using, because sometimes for Python, if we want to use a big int, the way they’re like – this big int is implemented in Python is very different than maybe the one that we are using in Rust, or we can be used in other language or targets to WebAssembly.

[0:33:45.8] JM: If I understand correctly, Rust compiles down to the LLVM intermediate representation. That intermediate representation can be translated into WebAssembly and that translation work has been done by different teams that work on WebAssembly. If you wanted to do this with Python, Python doesn't compile down to the LLVM intermediate representation. I think Python compiles down to some different bytecode version. Then so we would have to write some translation system to get the Python bytecode into the WebAssembly structures.

[0:34:29.3] SA: For example, right now in Python does, because python is a very dynamic language. It's going to be very hard to transform this Python code into WebAssembly, or into something that is static. In general, the strategies that people follow, there is do actually port the whole Python runtime, or the Python built on machine into WebAssembly. Basically, you have a interpreter of Python in WebAssembly, but this interpreter is not compile.

However in the case of Python, there is a project super curious called Nuitka that this actually it transforms the Python code into C code. Basically in general, we can gather like increase the performance to from five to 10X. Basically, once you have this Python code converted to C, then just because the C code is completely static and can be compiled to WebAssembly easily, then you can have a static and very performant Python to WebAssembly-like file. In general, what people – the approaches that people take is just porting or using the whole interpreter.

[0:35:36.6] JM: Okay. Well, let's get into your solutions. In order to get into your solutions, we need to talk about a term called ABI, which is application binary interface. What is an application binary interface?

[0:35:50.4] SA: Basically, each time we have a program that – or we compile a program, this program in general is trying to open files in your file system, or open sockets, or things like that. Basically, each time we compile a C file, this C file the way it operates with the whole system, let's say with your operating system is through a set of API calls between your application and the operating system. Until now, there has been interface, especially for unique that this is like – or ABI interface that is very used called POSIX ABI. This POSIX ABI you can think of a set of APIs or function calls, that basically allow us to interoperate with the operating system, in the sense of we want to open a file, or we want to open a socket, or we want to delete a file.

Basically, this set of API calls is called a ABI. It's called ABI, because we don't ship this logic into the application itself. We just assure the bindings are going to be there when we call the application. This application says like, “Oh, I want to open a file.” This open a file is just a link to the open file function for example in your operating system. These set of API calls for interoperating with your operating system, or with another binary application is called ABI.

[0:37:15.0] JM: There's also a term called POSIX. P-O-S-I-X. People listening have probably seen this term but they may not know what it means. What is POSIX?

[0:37:24.1] SA: POSIX is basically the interface that is commonly used for interacting with your operating system in Unix-like environment. This is basically like – this is very commonly used basically from people that are using Mac, Mac OS, or people that are using Ubuntu, basically like all the applications that are compiled to Unix-like systems, they are shipping with POSIX ABI. That's basically – that's the reason in general, these binary applications cannot be used in Windows for example. Because Windows in general doesn't – you have to do other tricks to make them work there basically.

[0:38:06.8] JM: The ABI will be making calls over the POSIX interface to make – is it a syscall, syscalls on the computer?

[0:38:16.3] SA: Yes. Basically, POSIX itself is an API. It's just a set of functions that we can call from our binary application. These set of functions are syscalls that in general, interact with our operating system and let us open files, or do whatever things that we need in the low-level stuff.

[0:38:37.9] JM: What kinds of applications need to make syscalls? Is it every application?

[0:38:41.9] SA: In general, for example if you have a application that is trying to, I don't know, open an image and render it, in general you will need a way to open this file and read this content. For example, for that you will need to interact with a read syscall.

[0:38:59.5] JM: This is basically every application. If I have a text editor, the text editor needs to open files, it needs to save files. If I have Slack, then certainly I'm opening files, I'm opening images, I'm interacting with the network and all of these things are going to require syscalls.

[0:39:16.9] SA: In the case of Slack, for example, Slack ships a electron-like application, Slack run Chromium under the hood. Basically some of these syscalls sometimes are being used not really, but in other wrapper, so sometimes through JavaScript, or through other interfaces. Yeah, each time we are opening a file, or accessing to network, or opening that port for a server, we are interacting with a syscall under the hood.

[0:39:45.9] JM: How does a WebAssembly module make syscalls?

[0:39:49.9] SA: Basically right now, there is not a universal way of doing. There is one project, like basically the first project that was actually been able to generate first JavaScript files, ASM JavaScript file from LLVM and second to generate a WebAssembly files, this project called Emscripten ship with POSIX ABI and it make it available in the browser.

Basically, it define a set of syscalls that are going to be available. Let's say, read a file, or open a file, or close a file, or open in a socket. Basically Emscripten define all this set of ABI calls that are going to be available and make basically a POSIX-like ABI for WebAssembly. However, this is just for projects, or that we have that are compiled to WebAssembly using Emscripten.

However for example, when you use Rust and you want to open a file, there is no standard way of doing it. Basically when you compile this Rust file to WebAssembly, there is no – a set of ABI calls that will be automatically provided. First with Emscripten, there was a set of ABI calls just imitating the POSIX ABI for interacting with the operating system. Emscripten approach was only available for C and C++ projects. For other projects that are targeting WebAssembly, there is no defined set of ABI calls that are basically like they use.

What I think will happen ideally is all companies and all the different languages targeting all the same ABI. That means when you compile a Rust file to WebAssembly, or when you compile a Go file to WebAssembly, or when you compile a C or C++ file to WebAssembly, these ABI calls remain consistent within all these three different WebAssembly files. That means the way for reading a file is exactly the same, is a function that receives the same arguments and outputs the same return type. They're opening a socket is exactly is also a function that is consistent across, can have all these different languages, rather than Go creating their own set of ABI calls, or Rust creating their own set for ABI calls for opening a file, or opening a socket.

Basically, one thing that need to happen is standardize the way the ABI calls are billed for WebAssembly. An interesting thing is actually building this set of ABI calls in a way that actually can make our WebAssembly files work in a lot of different scenarios. One is running in the browser. They might be running the server. They might be running in forms. Basically, there are a lot of use cases that will be ideal to define a standard ABI for WebAssembly.

If you want to dig deeper into how Emscripten does it is when we have a C or C++ project that's compiled to WebAssembly, we basically generate this WebAssembly bytecode and these WebAssembly bytecode interoperate with a host through certain set of syscalls. In the case of Emscripten, they emulated the POSIX ABI in JavaScript itself. When you try to open a file and they just – they create a fake wrapper in JavaScript that try to emulate the same API, or the same responses, which is super funny.

For example, when you – in Emscripten when you have a C and C++ project compiled to WebAssembly using Emscripten, and this project for whatever reason is opening a socket in your computer or is trying to read from a socket, in the case of Emscripten, the wrapper that they created is using rather on sockets, is using web sockets under the hood. Basically, it created or it emulated all these set of ABI calls in JavaScript in a way that plays nicely, or the WebAssembly file receives what it expected to receive for when it calls this syscalls.

[SPONSOR MESSAGE]

[0:44:03.9] JM: GoCD is a continuous delivery tool created by ThoughtWorks. It's open source, it's free to use and GoCD has all the features that you need for continuous delivery. You can model your deployment pipelines without installing any plugins. You can use the value stream map to visualize your end-to-end workflow. If you use Kubernetes, GoCD is a natural fit to add continuous delivery to your cloud native project.

With GoCD on Kubernetes, you define your build workflow, you let GoCD provision and scale your infrastructure on the fly and GoCD agents use Kubernetes to scale as needed. Check out gocd.org/sedaily and learn how you can get started. GoCD was built with the learnings of the ThoughtWorks engineering team, and they have talked in such detail building the product in previous episodes of Software Engineering Daily. ThoughtWorks was very early to the continuous delivery trend and they know about continuous delivery as much as almost anybody in the industry.

It’s great to always see continued progress on GoCD with new features, like Kubernetes integrations, so you know that you’re investing in a continuous delivery tool that is built for the long-term. You can check it out for yourself at gocd.org/sedaily.

[INTERVIEW CONTINUED]

[0:45:34.0] JM: Your vision for making WebAssembly modules interoperable involves this application binary interface. Is that in contrast to the tool chain that has been built with Rust? Are these disjoint strategies, or do we need both of these things?

[0:45:55.6] SA: Regarding defining a common set of ABI calls, I think it's just we need a little bit more time off from the industry to basically between Rust, Go, C and C++ basically defining a uniform set of syscalls, like a uniform ABI and on such that define, this should be super easy to interoperate in certain ways, or making sure these WebAssembly binaries were run.

[0:46:20.2] JM: Sorry to interrupt you, but why does it need to be consistent – I want an ABI for my Rust application, for my rust WebAssembly modules to interact with my system. I want an ABI for my Python applications to interact with my system, but why did those need to be the same thing? Can't my Rust applications just interact with the system and then Python interacts with the system and both those are fine, why does it need to be consistent?

[0:46:45.4] SA: Yeah. It des makes things much easier for first – the persons that are running the runtime and second, for whoever is running it. Let's say for whatever reason in each of the browsers, the API they ship for using JavaScript, or for creating an array is different. Just because of the fact that you want to target one, or the other browser, it will be very painful that the way your code is written, it's only adaptable for one scenario. Ideally, it will be all the browsers, they want to decide what is a good API for using, or for creating an array, for example. Then let everyone use this exact API. It's possible to create different ABIs for completely different context. In general, if the context is the same, I think it's very good for the industry to push for a standard there.

[0:47:40.1] JM: Got it. Is that because on this host machine, let's say I had a WebAssembly module that's written in Python and a WebAssembly module that's written in Rust. If they were using the same application binary interface, then the underlying system only needs to know how to interpret one of those interfaces. If it were two interfaces, then my computer would need to know how to interpret both of those interfaces.

[0:48:07.7] SA: Yeah, they're completely right. At the end, that's a little bit more painful. It's possible. At the end, all is possible. If you want to implement 20 different ABI sets, you could be able to do it. Ideally, when the context is the same, I think that we should push for standards.

[0:48:26.4] JM: Why is it so important? It's not that difficult to be able to run 20 different languages on my machine. Why is it so important to be able to have a consistent ABI?

[0:48:37.6] SA: Basically, the importance of that is because we can make sure your application runs in places where you have haven't thought of, or maybe it will come later on in the line. Let's say you create, I don't know, an application that is high-image viewer that opens an image. Ideally, where we would like to be is in a place where this application can be run in the browser. Not just in the browser. Can be run server-side, or can be run in a desktop, or can be run in a phone application, or in a phone itself. Ideally, when we have this nice abstractions, we can then let your code to be used in places where initially we haven't planned on. That might be super useful to be on.

[0:49:22.0] JM: Wait. Why does the consistency of an ABI have anything to do with where this application is going to run?

[0:49:30.0] SA: Basically, the consistency of the ABI let us target systems, where basically let's think of we are treating a custom – we define everyone a standard ABI interoperating from WebAssembly with the host system. If we are able to do it in a very proper way, we are able basically – all the industry, like the sides and to what's a good ABI system. Then once you compile your application to WebAssembly, then this application can be run in a runtime that is on the server, or it can be run in a runtime that is on the phone without actually having to go to a runtime and implement the ABIs all again. Basically, I think it offers the advantages regarding consistency and usage. I'm not sure if this clarifies completely your question.

[0:50:20.7] JM: Well, what I'm pushing back against is the question of resource consumption. Maybe I don't know about the resource consumption too much, but if I wanted to run a Rust application and a Python application on my phone and I needed two different ways of interpreting their respective application binary interfaces, yeah, that's going to take some extra resources, but is it really that much more to run a Python runtime in addition to a Rust runtime?

[0:50:49.6] SA: No. I mean, I will not say it’s regarding more. What I mean is a little bit ideal. If we define like a – is more ideal. If we define something that everyone agrees on, rather on creating custom things for doing the same thing, like custom APIs for doing the same thing. Basically, if the industry can decide into what's the ideal way of reading files and everyone, we can define what's a good way, then we don't need two different function calls that are doing exactly the same.

It's basically like when the context is the same, I think it is to define a standard. Let's say for example in the case of we want to ship or user WebAssembly and Internet of Things devices, maybe there are various other set of ABI calls that we need to consider. Basically is because the context is completely different. In the case of the functions, or what we expect from doing from a function is the same, across platforms I think pushing for a standard is the obvious approach.

[0:51:48.4] JM: If you contrast an Internet of Things device like my toaster, if I want to deploy applications to my toaster, versus deploying applications to a server, I can certainly see how the quantity of resources would be available, would be different in the toaster versus the server sitting in a data center. Would the syscalls be different? Wouldn't the syscalls for making a file and reading a file and reading an image, wouldn't those basically be the same?

[0:52:16.9] SA: Yeah. I mean, probably in the general case, I will say yes. I think in some other cases, there might be some special usages from my ABI, basically you might need to divert. In general, I think the set of syscalls that you go using in your binary application should be consistent and should be in general the same. In general, I'm pushing forward towards consistency on the industry for WebAssembly and how to create our ABI in WebAssembly, or what is the proper ABI for WebAssembly.

[0:52:54.4] JM: If the syscalls are consistent across the toaster and the data – the server and the data center, I still don't understand why the consistent ABI across different WebAssembly modules is necessary, because couldn't they just define – they could each define their own ABI as long as the syscalls in the respective environments are the same –

[0:53:16.6] SA: Oh. Basically, a set of the ABI – like what is ABI is a set of syscalls. Basically if the syscalls are consistent, then the ABI is going to be consistent. Basically, ABI and syscalls are the same thing.

[0:53:31.5] JM: Okay. I mean, most of these underlying systems are Linux, or UNIX-based systems. I'm having trouble understanding why we need the consistent ABI, I guess. Again, if I have a Rust application and a Python application, what I need the consistent ABI for –

[0:53:49.7] SA: Okay. Basically, what I mean with consistent ABI is a consistent set of syscalls that we can all use for opening a file. That's for me a consistent ABI, this consistent set of syscalls.

[0:54:01.8] JM: I see. That just doesn't exist today. Today, a Rust thing that's been compiled down to WebAssembly is going to have a different set of syscalls than a C application?

[0:54:12.2] SA: Yeah. I mean, if we are doing from Rust, like there will be no a standard there, so basically we'll have to create our syscalls by hand. If we are using C or C++ and Emscripten, there is a set of syscalls based on POSIX that basically we will target to. If we are using Go, again each module that we are using will define their own set of syscalls. Again, that's not ideal. The ideal will be everyone targeting the same set of syscalls.

[0:54:46.4] JM: Okay. Is this unified ABI, this is cloud ABI?

[0:54:51.6] SA: Cloud ABI is something super interesting. Basically, Cloud ABI what it adds is a permission set on top of our ABI calls. Let's say our binary application is trying to open a file in your system. Let's say for whatever reason, we don't want to allow that, or we want to allow that, but we don't want to allow opening sockets. What cloud ABI did is actually wrapping these ABI calls with a permission system.

Basically, it's time we try to open a file, or we try to access or to execute a certain syscall, cloud ABI is making sure we have permission to do that. How it knows we got permission is before executing, or at the same time we execute that file, we establish what is the set of permissions that we are going to give.

For example, we establish like, “Oh, this binary application is going to be able to open a file, or this binary application will eventually be able to open a port.” Basically, that's what cloud ABI is doing. Also, other thing that they’re doing is it had the smallest subset of syscalls that we need to implement. In the case of Emscripten, I think the total number of syscalls are in the order of 200, I think around. In the case of cloud ABI is just 49 syscalls that have a permission mechanism on top of that.

[0:56:13.2] JM: What are you building at Wasmer?

[0:56:15.1] SA: Basically at Wasmer, first what we are seeing is WebAssembly, we believe WebAssembly will become incredibly useful in the future and we believe we'll go outside of the browser environment. First, what we are trying to do is bring WebAssembly server side, so basically anyone that wants to use WebAssembly and don't need the JavaScript runtime, they can do it super easily.

Second, what we are trying to do is move WebAssembly to a server side, we are trying to make very easy to interoperate with WebAssembly from other languages. For example, if you want to use WebAssembly modules inside of Python, you should be able to do it in a very easy way. If you want to use the same WebAssembly module inside of Go, you should be able to, or basically almost any other language that you can think. This is the other thing that we are going to completely focus on.

The other vertical is basically creating something that would power the next generation of cloud platform systems. Basically, because of all the things that I explained before, WebAssembly is ideal for running on the edge, because a startup time is super good and we can run very close to native performance. Basically, what we are trying to do is create this runtime that will be very useful for these cloud environments.

[0:57:37.2] JM: Do you think you'll spin up a cloud provider yourself as a business model, or do you think you will sell this technology to cloud providers? What do you think the business model will look like?

[0:57:47.4] SA: Our business model, it’s first we are more focused on engaging with the community. Basically, we want to make first WebAssembly successful and use across any scenario, or a lot of scenarios. Then what we are going to do is either become a virtual cloud provider. That means we might not care as much about the infrastructure, but we might provide a very easy way for you to execute WebAssembly files from the cloud. That's one of the sides of the business model.

[0:58:15.7] JM: Like the second layer cloud providers, like the Zeit, or –

[0:58:19.3] SA: Yeah, yeah. Completely.

[0:58:20.7] JM: Yeah. Zeit, or Netlify, or Spotinst. These second layer cloud providers are so interesting. I think this is a trend that snuck up on me that the whole cloud providers built on other cloud providers thing.

[0:58:33.5] SA: Yeah. At the end, what I think is important is we don't want to focus right now at least on the short-term on infrastructure, because it's very hard to build a proper infrastructure system, where I don't know, you have a lot of servers running and they are co-located in a lot of different places. Right now, our approach is more regarding software and make it very easy to use and very easy for the developer to ship. That's what matters. The how, or what's happening under the hood, or how many servers, do we have is not as important for the developer.

[0:59:08.4] JM: Well Syrus, I want to thank you for coming on the show and bearing with my insufficient knowledge of this space. I feel how I felt near the beginning – when I started covering Kubernetes and when I started covering cryptocurrency technology, I just felt confused the entire time. Eventually, a switch flipped where I felt a little bit more comfortable, but that's about where I feel with the WebAssembly today. The whole tool chain it feels so foreign to me. I appreciate you bearing with my confusion.

[0:59:39.6] SA: It's been a super, super interesting talk at the same time. One thing that I haven't commented is that maybe interesting too just saying –

[0:59:47.6] JM: Sure. Yeah.

[0:59:49.5] SA: - in few minutes is other use case for WebAssembly for example, for the centralized applications. A lot of crypto companies are right now looking into WebAssembly to execute a smart contract in a very efficient way. Actually, I think basically the cases of WebAssembly in the future grows especially out of the browser to shipping servers as we are doing Wasmer, to ship it to the centralized applications, a script we are doing, or even becoming the next JBM.

[1:00:18.8] JM: Definitely. I think there's also potential for mobile computing, some unified deployment mechanism. I mean, we're seeing such a desire to unify the mobile development paradigms, of course we always have for the last 10 or 15 years, however long mobile computing has been a thing. We'll see about that. I completely agree with your excitement around WebAssembly. There are so many different things that it can accelerate and improve and help the security of, help the isolation of. Thanks for coming on the show, Syrus. Really great talking.

[1:00:49.3] SA: Thank you. I really enjoy our talk and hopefully for people that are looking to WebAssembly, this talk incites them to start trying it. I want to welcome people to try and letting us know what are their use cases of WebAssembly and basically, Wasmer is going to be here for hopefully being able to drive some of these WebAssembly love and WebAssembly new excitement. Hopefully, we'll have more people using Wasmer and using in general WebAssembly in the future.

[1:01:19.5] JM: Awesome. Okay, thanks Syrus.

[1:01:20.3] SA: Thank you.

[END OF INTERVIEW]

[1:01:25.4] JM: This podcast is brought to you by wix.com. Build your website quickly with Wix. Wix code unites design features with advanced code capabilities, so you can data-driven websites and professional web apps very quickly.

You can store and manage unlimited data. You can create hundreds of dynamic pages, you can add repeating layouts, make custom forms, call external APIs and take full control of your site's functionality using Wix code APIs and your own JavaScript. You don't need HTML or CSS.

With Wix code's built-in database and IDE, you've got one-click deployment that instantly updates all the content on your site. Everything is SEO-friendly. What about security and hosting and maintenance? Wix has you covered, so you can spend more time focusing on yourself and your clients.

If you're not a developer, it's not a problem. There is plenty that you can do without writing a line of code, although of course, if you are a developer then you can do much more. You can explore all the resources on the Wix code site to learn more about web development wherever you are in your developer career. You can discover video tutorials, articles, code snippets, API references and a lively forum where you can get advanced tips from Wix code experts.

Check it out for yourself at wix.com/sed. That's wix.com/sed. You can get 10% off your premium plan while developing a website quickly for the web. To get that 10% off the premium plan and support Software Engineering Daily, go to wix.com/sed and see what you could do with Wix code today.

[END]