CS131 Homework 6 Report

Jeremy Cristobal
Discussion 1C

Abstract

Exploring the three programming languages of: D, Odin, and Zig, and evaluating their viability and usefulness for secure, camera-based HVAC control. Security features of each language are examined specifically and compared against the security features of the other languages in order to determine which programming language is best suited for this application.

1 Introduction

In this project, we are tasked with assisting in the development of a heating, ventilation, and air conditioning (HVAC) system that utilizes a Human Embodied Autonomous Thermostat (HEAT) framework. In contrast to traditional thermostat systems, HEAT controls temperature by measuring the comfort level of all individuals. This process presents itself in two steps, which correspond two its two components. First, the thermal comfort sensing monitors the involved individuals through the collection of real-time physiological data, such as skin temperature. Then, the thermal comfort optimization determines the optimum temperature given the collected data. For the HEAT system specified in the project, said data is collected through security cameras.

However, the security vulnerabilities of C/C++ concern potential institutions that may wish to use this system. As such, we will investigate the potential for the languages of: D, Odin, and Zig as an alternative to C/C++. Specifically, we want to examine: readability and ease of auditing, ability to create freestanding programs, simplicity of the language, and hardware interface capabilities.

2 Ethical Concerns

Before we begin, it is important to discuss the ethics regarding the usage of HEAT for this HVAC system. Because the system is gathering real world data through the use of its cameras and transmitting said data across a given network, we can easily classify this HEAT system as an Internet of Things (IoT) system. Since its conception, the Internet of Things has faced many ethical challenges. Because our HEAT system is rather limited in scope, however, it will not face all of the challenges posed by most Internet of Things systems, but some are still hardware support via tiers.

2.1 Informed Consent

The first issue that needs addressing is the concept of informed consent. The difference between informed consent and consent is that the stakeholders in informed consent need a decent understanding of what they are consenting to. The word ‘decent’ is intentionally vague because there is no technical point at which consent becomes informed consent, which in and of itself can be problematic.

Regardless of this, informed consent will certainly be a point of concern for companies wishing to implement the HEAT system. Historically, End License User Agreements (ELUA) have been used to ‘get informed consent’ from users, but have been perpetually ineffective for a multitude of factors. The solution, then, is to implement a regulatory framework in lieu of an ELUA. Though this may be more costly to the industry, ultimately its superior effectiveness should outweigh any potential problems it imposes. Additionally, for secure industries such as banks that have a regular flow of clients, this may

ultimately prove to be more useful than forcing each new client that enters the building to agree to an ELUA.

2.2 Physical Safety

The proposed HEAT system is a thermostat controller, which at first glance may seem relatively harmless compared to some other Internet of Things systems such as self-driving cars. Nonetheless, it warrants discussion because there are some potentially dangerous effects that can arise.

The first potential concern is that the system will malfunction or that someone with malicious intent will attack the system, either of which could cause drastic temperature changes. However, HVAC systems are typically limited in the temperature range in which they can operate, so there should be no concerns about reaching a potentially fatal temperature.

Another potential concern is that because the temperature outside is a big factor of inside temperature regulation, doors may be connected to this Internet of Things. In such cases, a malfunction or malicious attacker could potentially lock the doors shut or open locked doors. Certainly, this is a potential issue, and unfortunately the solution is not clear. Liability for harm caused by these systems is a very murky area with no clear-cut answers. Potentially, we have the option to simply not include doors in the HEAT temperature regulation features, but at the cost of effectiveness. This then raises challenging moral dilemmas regarding safety, cost, and efficiency that I do not have the answers to.

3 D

D is a language released in 2001. It was created with the intentions of improving upon certain aspects of C/C++. Consequently, its syntax is very similar to that of C/C++ and it implements many of the same features.

3.1 D Memory and Security

One of the security concerns regarding C/C++ is found within its implementation of arrays, where they are simply contiguous chunks of data that are accessed via pointers. This makes C/C++ susceptible to buffer overflow attacks. As a solution to this problem, D introduces the data type slice. Slices are segments of arrays that track both the pointer and the length of the segment. This additional capability to track the length of the segment is able to prevent many of the memory corruption issues that can manifest in traditional C/C++ arrays.

Additionally, there is a subset within D called SafeD that makes it impossible for the program to corrupt memory. In order to accomplish this feat, pointers and unchecked casts are excluded. These safe functions are marked with the '@safe' attribute, and safe functions cannot exhibit undefined behavior or create unsafe values that other parts of the program would be able to access.

3.2 D Advantages

Because the company in question has traditionally used C/C++ for programs in the past, learning the syntax of D should prove to be rather easy because its syntax is so largely based on C. Furthermore, D has a mode that can successfully compile C, and as such one should be able to import parts of the existing program that are memory safe directly into D.

Additionally, like C/C++, D functions as both a system and application programming language. As such, it is well-suited for embedded software and can function as a freestanding program. Because we are examining languages for the purpose of cameras with embedded CPUs, both of these aspects are immensely important.

3.3 D Disadvantages

D uses a garbage collector to handle much of its memory. This, in and of itself, is by no means always a disadvantage and is, in fact, helpful in many instances. However, it is less than ideal for what we are ultimately trying to accomplish because it directly contradicts the goal of creating stripped-down software. Though there are methods of programming in D that avoid using the garbage collector, the aforementioned type slice, which serves as a security measure, relies heavily upon it. As such, when writing the software, a choice must be made between excluding slice or including the garbage collector.

Another disadvantage is that D, like C/C++, is a statically typed language, meaning all type checking happens during compile time, not run time. Additionally, all variables and functions must be declared with an explicit type in D. Though this may manifest as a benefit for different applications, for this project specifically we are concerned about the complications and cluttering of code that these explicit declarations will ultimately cause.

4 Odin

Odin, like D, has a history closely related to C/C++. Its conception is attributed to the annoyance of a certain individual with C++, and it originally was intended to be a preprocessor for C with additional features. However, eventually this idea was scrapped and the language was built from scratch, drawing inspiration from Pascal, Go, and Oberon in addition to C. Amongst its philosophies are simplicity, minimalism, clarity, orthogonality, speed of compilation, and high performance.

4.1 Odin Memory and Security

Much like D, Odin introduces the slice data type not found within C/C++. As previously stated, the additional capability of being able to track the length of the slice is a powerful tool that can help aid against attacks in ways that pointers alone cannot.

Odin, like C/C++, is a manual memory management language, meaning the programmer must manage memory themselves. However, Odin has built in functionality for custom allocators, which is a feature not native to C/C++. As such, Odin provides the capability to track what the allocator is doing and how it is doing these allocations. A good custom allocator can potentially help circumvent certain security issues.

4.2 Odin Advantages

Odin, like D and C/C++, is a statically typed language that is type checked during compile time. However, unlike its predecessors, Odin does not require its type to be explicitly stated on declaration. It certainly is allowed to be, but if it is not then the type will be inferred by the right side of the declaration. For example, the line <x : = 24;> will default x to being of type int.

Additionally, this contributes to the idea of simplicity that Odin prides itself upon. In order to keep consistent with the theme of simplicity, while loops were removed in favor of extended abilities for for loops, there is no operator overloading, it is not object oriented, and more.

Using the ‘foreign’ keyword, Odin is able to import and utilize C code. This will again be useful for importing parts of the existing code into Odin, though is more difficult than importing code into D. Lastly, Odin is a low-level systems language, which should allow it to interact with the camera hardware and without an operating system without problem.

4.3 Odin Disadvantages

Though the lack of object oriented programming serves to improve the simplicity and minimalism of the language, it also removes some powerful capabilities. Additionally, there is no built in parallelism or concurrency, though there are some Odin library functions that are able to help in that regard.

Additionally, the language is fairly new and not very widespread. Its syntax is unusual in some senses compared to C/C++, and as such it may be more difficult for the current workers to transition to Odin than some other languages. Learning and becoming accustomed to the new syntax may also serve to slow down auditing.

5 Zig

Zig is an imperative programming language first released in 2016 that, like the previously discussed languages, was created with the goal of improving upon C/C++. It was made for the purposes of creating robust, optimal, and reusable software.

5.1 Zig Memory and Security

Zig, like C/C++ and Odin, forces the programmer to manually manage the memory as it has no automatic garbage collector. Furthermore, unlike C/C++

(which by default has malloc, realloc, and free), Zig does not have a default memory allocator. It is thus the job of the programmer to manually create and specify an allocator by which the program will use.

A unique aspect of Zig is that it features four different build modes that the user can choose from. In Figure 1, it is easy to see that each of these modes has its benefits and drawbacks. For the purpose of this project, the reproducibility, medium runtime, and safety check aspects of ReleaseSafe seem to be the best choice. These safety checks serve to prevent the execution of undefined behavior and are thus crucial to fulfill the improved security requirement.

Build Mode	Safety Checks	Runtime Speed	Reproducible
Debug	Yes	Slow	No
ReleaseFast	No	Fast	Yes
ReleaseSafe	Yes	Medium	Yes
ReleaseSmall	No	Medium	Yes

Figure 1: A table comparing each different build mode of Zig against each other for the aspects of: safety, runtime speed, and reproducibility.

5.2 Zig Advantages

Zig is designed to be a small and simple programming language, as demonstrated by the ability of its entire grammar to be represented in a 500 line file. Like the creator of Odin, the creator of Zig decided to not make it an object oriented language in favor of simplicity. Though parallelism and multithreading are not currently possible, Zig has an asyncio library that allows for concurrency.

Zig should always do what it is expected to do because it has no hidden control flow or memory allocation. This makes Zig far easier to debug and audit. Additionally, besides its usage for increased safety and security measures, the four different build modes are incredibly useful for improving the ease of debugging even more.

Similar to D, Zig is able to not only import snippets of C code, but is able to fully compile it as well. As a system programming language, it is well suited to create freestanding programs that can directly interact with low-level hardware. In fact, Zig is very explicit with what hardware it works well and is fully compatible with through its system of ranking hardware support via tiers.

5.3 Zig Disadvantages

Like all the languages discussed, Zig is statically typed. Unlike Odin which can infer types, however, Zig declarations are quite complex and clutter the code. Declarations require: a key word (most commonly var, or fn), the type declaration (such as i32 or u8), and the variable name. Additionally, the types are not in the style of C/C++ and will reduce the readability of those switching from said languages.

6 Conclusion

After researching each of these languages thoroughly, it is clear that one is simply better suited for this job than the others. All three are well suited for system programming and will be able to function as standalone programs, so this is not an advantage for any of these given languages. As far as memory and security go, D has the advantage, as its SafeD subset makes memory corruption nearly impossible. Regarding simplicity and readability, however, Odin takes the cake. Its simple syntax and ability to infer types allow for easily readable code, and its explicit exclusion of object oriented programming and parallelism simplify the language immensely. However, the best language for this particular project is clearly Zig. Though it may be more difficult to read and audit than the other two, its advantages more than make up for this detriment. The security and safety of Zig is a clear upgrade from that of both C/C++ and Odin. Though not as stripped down as Odin, Zig is much more simple than the likes of C/C++ and D. Because it has great marks in both of these categories, whereas the other languages excel in one but fail in the other, it is the best language for this project. Therefore, I highly recommend Zig for the software of HEAT systems.

References

Allhoff, Fritz and Henschke, Adam. "The Internet of Things: Foundational ethical issues". https://www.sciencedirect.com/science/article/pii/S2542660518300532

Byon et al. "HEAT - Human Embodied Autonomous Thermostat". https://ebyon.engin.umich.edu/wp-content/uploads/sites/162/2020/05/2020.HEAT-Human-Embodied-Autonomous-Thermostat.pdf

Eggert, Paul. "Homework 6. Evaluate languages for secure camera-based HVAC control". https://web.cs.ucla.edu/classes/fall20/cs131/hw/hw6.html

"Frequently Asked Questions" (Odin). https://odin-lang.org/docs/faq/

"Garbage Collection" (D). https://dlang.org/spec/garbage.html

Milewski, Bartosz. "SafeD". https://dlang.org/articles/safed.html

"Odin Overview". https://odin-lang.org/docs/overview/

"Odin Wiki". https://github.com/odin-lang/Odin/wiki

"Overview" (D). https://dlang.org/overview.html

Schveighoffer, Steven. "D Slices". https://dlang.org/articles/d-array-article.html

"Zig Introduction". https://ziglang.org/

"Zig Documentation". https://ziglang.org/documentation/master/