A Cybersecurity Moonshot

Hamed Okhravi
MIT Lincoln Laboratory
hamed.okhravi@ll.mit.edu

***Abstract*—Cybersecurity needs radical rethinking to change its current landscape. This article charts a vision for a cybersecurity moonshot based on radical but feasible technologies that can prevent the largest classes of vulnerabilities in modern systems.**

I. Introduction

In 2016, MIT Lincoln Laboratory conducted a study on 'moonshot' efforts in various areas. The author led one of those studies for cybersecurity. The study resulted in a number of follow-on projects to execute its vision and start the journey towards its ultimate goal. This article captures some of the high-level challenges and opportunities identified in the moonshot study as well as the research and development performed in the past five years at MIT to execute its vision.

That cybersecurity needs ambitious and creative rethinking is not only supported by intuition, but also by data. For instance, while more and more funding is being spent for cybersecurity every year, by multiple measures such as the extent of data breaches, the number of vulnerabilities exploited, and the number of attacks, the situation seems to be getting worse [1]. Despite decades of research and practice in cybersecurity, large, traditional classes of vulnerabilities don't seem to be tapering down. A recent study by Microsoft, for example, shows that consistently a large fraction of vulnerabilities are memory safety errors [2], despite the fact that such errors are one of the earliest classes of vulnerabilities to be studied, and decades of research has been dedicated to mitigating them. To make meaningful progress in cybersecurity, we need disruptive technologies and concerted efforts; we need a moonshot.

A moonshot is defined by Donald Boudreaux as "radical but feasible solutions to important problems" [3]. A cybersecurity moonshot, thus should outright prevent some large classes of vulnerabilities (radical) and be based on realistic and practical technologies (feasible). While a moonshot project is often thought of as a consolidated effort, in reality, it is often a process contributed to by many researchers and enabled by some critical technologies that reach a tipping point.

In this article, we chart a vision for the cybersecurity moonshot. We investigate the largest classes of vulnerabilities in modern systems and identify three main legacy design choices that are the root causes of the status quo: the use of unsafe programming languages (like C/C++); the lack of proper semantics in the processor; and the monolithic, highly-privileged design of the operating systems. We discuss how these legacy design choices, inherited from early systems such as the PDP-11 machine and MULTICS (which in turn heavily influenced UNIX), have contributed to some of the largest classes of vulnerabilities in modern systems. We then describe a roadmap for the cybersecurity moonshot based on multiple past and ongoing projects at our laboratory and multiple other research groups.

Unsafe languages delegate most security checks to the developers in the interest of making the compiler as simple as possible. These checks are notoriously hard to get right, and a long history of bugs introduced as a result of this trade-off highlights the need for rethinking here. Processors have traditionally treated all code and data (including all data types) as raw bits[1]. This makes it increasingly difficult to perform meaningful security checks in the processor. Finally, the monolithic and highly-privileged design of modern operating systems often allows a single vulnerability to be exploited to hijack the control of the entire system and perform arbitrary malicious actions.

The first pillar of our vision has to do with enriching the processor with semantic information. Recent advancements in tagged architectures [4], [5] allow the storage and association of extra bits of metadata for each word of memory, and the enforcement of security checks on runtime execution of code in the processor itself. Designing and automatically generating proper security policies that balance the trade-offs appropriately, however, requires additional research that we highlight later in the article

The second pillar focuses on the proliferation of modern, safe, systems programming languages such as Rust that provide native memory safety. Rust does so by virtue of having a strong type system and performing static bounds checks to prevent spatial memory corruption (*e.g.,* buffer overflows) and ownership checks to prevent temporal memory corruption (*e.g.,* use-after-frees). Achieving a fully memory-safe system, however, requires addressing important research questions.

The third pillar focuses on the operating system (OS). Enabled by the safe programming language and the semantically-rich processor, the operating system can be designed in a way that each of its functional modules is in a separate compartment. Each compartment can then be assigned the "least privilege" necessary for it to do its job, but not any more. The privilege enforcement can be done seamlessly in

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited. This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

[1]This is particularly true about the von Neumann architecture commonly used today. The Harvard architecture does separate data and code, but it doesn't distinguish between different types of data.

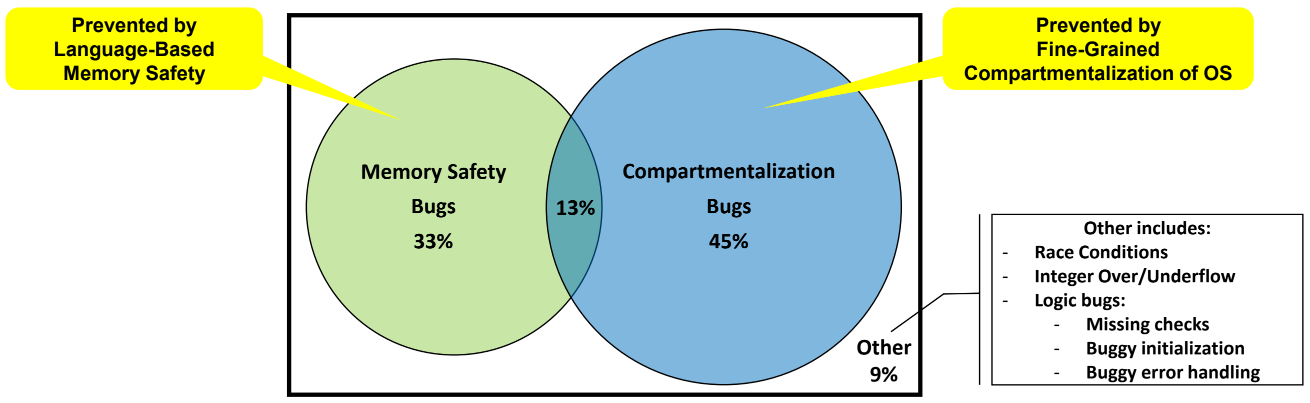

Fig. 1: The past five years of vulnerabilities in Linux categorized by their type.

the tagged architecture, thus removing the need for an omni-privileged software layer. We call this new design a *Zero-Kernel OS (ZKOS)*. Redesigning an OS around this new model in a performant manner, however, requires more research.

Equipped with these pillars, we discuss how this redesigned computer system can inherently prevent the largest classes of vulnerabilities (radical), while relying on near-ready, concrete technologies (feasible). We also highlight the additional research problems that need to be addressed to further mature them for operational deployment.

II. Why Moonshot? What Moonshot?

A moonshot is a risky undertaking. It imposes not only costs in the form of abandoning existing systems (*i.e.,* new hardware and software cost), but also it, by its very nature, has technical risks. It requires time-consuming ventures to address known unknowns during which many unknown unknowns also come to light. It can only be justified if incremental improvements to existing systems are not sufficient to achieve the desired goal.

This is indeed the case for cybersecurity. Multiple decades of research into defenses that are soon followed by novel attacks that bypass them have created an unending arms race. To exemplify this arms race, it is helpful to look at the memory safety domain. To mitigate simple code injection attacks, canaries were developed and deployed [6]. This was soon followed by selective overwrite attacks that could bypass canaries. In response, write XOR execute (W⊕X) was developed [6]. Soon code reuse attacks were devised to bypass W⊕X [7]. Code randomization techniques were matured to combat code reuse [8], just to be bypassed by various forms of information leakage attacks [9]. Control-flow integrity was then developed as a defense that is resilient to information leakage attacks, but it was in-turn bypassed by attacks that respect the control-flow graph [10]. At the risk of laboring the point, the situation is significantly exacerbated by recent microarchitectural attacks [11]. With the wide availability of automated tools that generate exploits against the most sophisticated defenses, little tangible security seems to have been achieved. This anecdotal arms race is also supported by real-world attack data, which is the ultimate measure of success/failure. The attacks, breaches, and malicious activities are only getting more voluminous [1]. The incremental improvements, therefore, don't seem to be working.

To better highlight what needs fixing, it is beneficial to have a more quantitative understanding of the current vulnerabilities. We have analyzed the past five years of vulnerabilities in Linux that have *Critical* or *High* severity ratings, a total of 506 vulnerabilities. This analysis was done manually by inspecting each vulnerability and asking the question of 'what would have prevented this vulnerability?'. Figure 1 illustrates the results. We have categorized these vulnerabilities based on whether they are related to memory safety issues, lack of compartmentalization, or other types of vulnerabilities. For example, a stack-based buffer overflow vulnerability is a memory safety bug (*e.g.,* CVE-2017-12762), while a vulnerability that allows the user ID (`uid`) to be decremented to zero, thus allowing a user from a buggy module to access sensitive kernel functionalities is a compartmentalization bug (*e.g.,* CVE-2016-4997). There are also vulnerabilities that can be mitigated by either memory safety or compartmentalization (a buffer overflow allowing privilege escalation, as in CVE-2019-18675).

As can be observed from Figure 1, the vast majority ($> 90\%$) of vulnerabilities in Linux are either related to lack of memory safety ($\sim 33\%$) or lack of fine-grained compartmentalization ($\sim 45\%$) or both ($\sim 13\%$). The remaining vulnerabilities ($\sim 9\%$) are related to other classes of bugs, namely, race conditions, integer over/underflows, and logic bugs such as missing permission checks or buggy error handling.

This provides us a roadmap for our vision: in order to prevent 90% of current vulnerabilities in an operating system (Linux), memory safety and fine-grained compartmentalization must be implemented. Preventing 90% of vulnerabilities conveniently corresponds to an order of magnitude reduction in vulnerabilities, which is the goal called out in the 2019 Federal Cybersecurity Research and Development Strategic Plan as well. Note that these numbers are based on known vulnerabilities. Of course, we cannot reason about unknown vulnerabilities, but given that these distributions have changed very little over the past couple of decades, they are not expected to change significantly in short- or medium-term.

In the following sections, we discuss how these two ambitious goals (memory safety and fine-grained compartmentalization) can be achieved, empowered by recent advanced in semantically-rich processors (*i.e.,* tagged architectures). We start with the semantically-rich processors as they are an important enabling technology for the other two goals.

III. Semantically-Rich Processors

A foundational challenge in modern computer security is the lack of semantic information in commodity processors. Processors view code, various types of data (including control and non-control data), and even configuration information (e.g., for memory-mapped input/output devices) as raw bits. There is no notion of a `return address` vs. an `integer` in a processor. This is partly the reason why a buffer of integers can overflow into a return address and maliciously alter control flow in what is known as control-flow hijacking attacks. The lack of semantics in the processor allows the overflown data (in this case an integer) to be interpreted as a return address, which diverts control to an injected code (a.k.a. code injection attack) or existing code in the memory space of the application (a.k.a. return-oriented programming attack). If the processor had the semantic information, this attack would not be possible. Even in memory, data and code are stored as raw bits and there is little distinction between various types of data. This is often referred to as the *flatness* of memory space. Recent work has shown that attempts to detect malicious activities using modern hardware features (e.g., Hardware Performance Counters – HPC) fail precisely because of the poor semantics in processors [12].

It is important to note that these design decisions were made in a different era (60's and 70's) when resources were extremely limited and there was little justification for including extra metadata in processor or memory. Simplicity and resource conservation were the two main goals to be able to achieve a functional system.

Unsurprisingly the needs and capabilities of the modern era are vastly different from those early times. We have gigabytes of memory available to an average computer rather than kilobytes; modern processors have billions of transistors in real estate rather than tens of thousands. What is perhaps surprising is that for the most part, these early design decisions have stayed unchanged for multiple decades.

This has changed in recent years, however. Multiple influential efforts have focused on building semantically-rich processors by adding tags that carry metedata to every word in memory (including registers). They are thus called *tagged architectures*. A policy can then be enforced on any code execution in the processor via a policy enforcement logic. This logic can be implemented by adding additional instructions to the processor or by adding a dedicated policy enforcement engine in parallel with the main application processor.

Figure 2 illustrates a simplified view of a tagged architecture. As can be observed in the figure, the memory, the instruction and data caches, the register file, and the instruction pointer are all tagged with extra metadata. Upon execution of each instruction, the policy enforcement engine either allows or denies the execution. If the execution is allowed, the policy engine determines the tag of the output based on the tags of the instruction, the instruction pointer, and the operands. If the execution is denied, however, it raises an exception and goes into exception handling.

A simple policy, for example, can enforce that a `return` instruction can only return to a value tagged with a specific type (e.g., `RET`), and not any other value, thus preventing the control-hijacking attack described above.

Prominent examples of tagged architectures include CHERI [4] and DOVER [5] that provide expressive policies. Even commodity processors have started implementing tagged architectures. ARMv8.5-A architecture has a new Memory Tagging Engine (MTE) that provides four bits of tags and extensions to the ISA to set and check tags during execution.

A semantically-rich processor has two main advantages to a pure software-based enforcement of security checks: it is significantly more efficient and it reduces the trusted computing base (TCB) in software. While software defenses exit that retrofit security checks to unsafe languages (e.g., SoftBound [13]), they impose a large performance overhead because multiple instructions are needed to check the safety of a sensitive instruction (e.g., a pointer dereference). Tagged architectures avoid this by implementing native instructions that can perform such check efficiently, thus adding a modest overhead, or having a parallel engine that introduces little overhead to native execution by trading off silicon area for performance. They also reduce the size of the TCB in software. While hardware is by no means immune to vulnerabilities as evident by recent microarchitectural attacks, reducing the TCB in software is a valuable exercise. Needless to say that accurate representation of semantics in the processor relieves the upper security layers from trying to inaccurately reconstruct the semantics as is the case with many HPC-based defenses.

A. Research Challenges

For all the benefits semantically-rich processors provide, there are still important research challenges to be addressed before they can be used widely.

There is often a strong trade-off between how expressive tagged architectures are and how large the tags are (which in turn creates silicon, power, and memory overheads). DOVER is thus more expressive than ARM MTE because it has much larger tags (configurable tags that can be 265 bits or even larger vs. 4 bits). A research challenge is how to design flexible and meaningful policies in a small tag space. Particularly if MTE achieves widespread adoption due to the vast fabrication capacity behind ARM processors, it becomes imperative to design policies that can leverage a 4-bit tag to its fullest.

Automatically generating complete and effective fine-grained policies for tagged architectures is another important challenge. This is a task best done in the compiler since it has the necessary semantic information. While tag policies have been generated automatically in the related work [5] for some

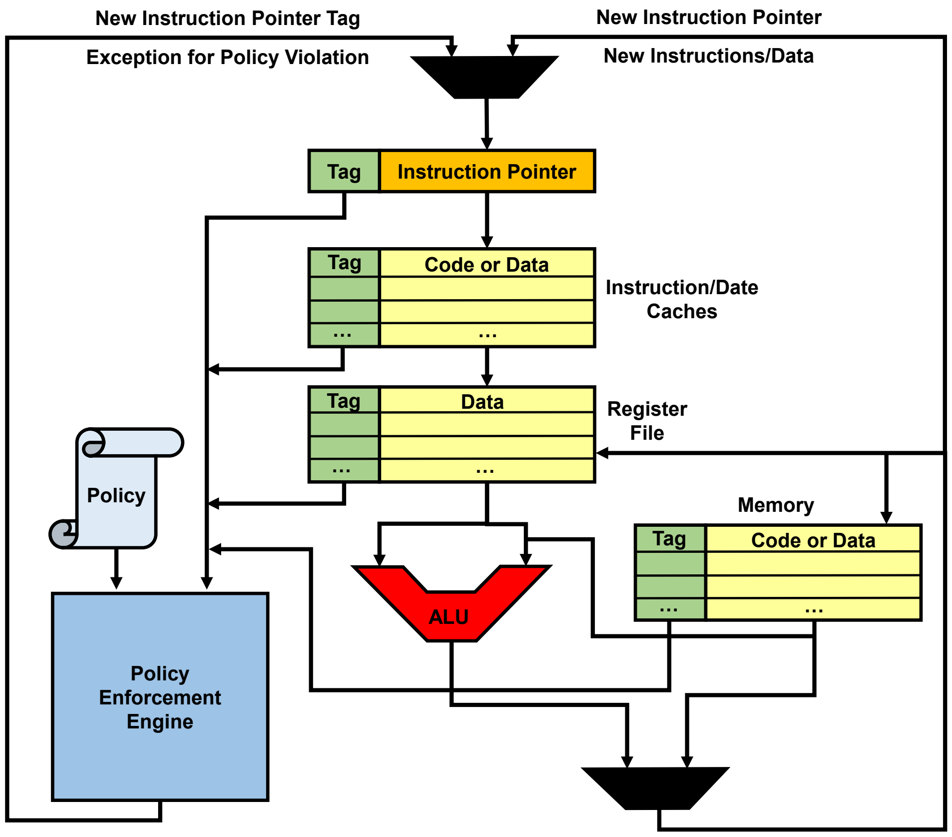

Fig. 2: A simplified view of a tagged architecture.

high-level security goals (*e.g.,* control-flow integrity), reasoning about the completeness and efficacy of the tag policies for an arbitrary goal remains an open research problem.

Another important challenge is understanding the trade-offs involved in policy design. Often a given policy (e.g., preventing buffer overflows) can be designed in multiple different ways that trade-off granularity, overhead, policy complexity, and power consumption, among other dimensions. While how to build tagged architectures is rather well-researched, what policies to enforce with them has comparatively received much less research. Understanding these trade-offs is thus an important open research problem. How to compose various policies and how to enable them in dynamic loading/linking also remain as challenges in this area.

Finally, designing tagged architectures that can work seamlessly with modern hardware features such as multi-core, direct memory access (DMA), and out-of-order and speculative execution, while resisting various side-channel attacks remain open problems.

IV. Language-Based Memory Safety

The second component of our vision is wide-spread usage of safe programming languages, thus achieving language-based memory safety. Unsafe languages like C/C++ attempt to simplify the compiler as much as possible by delegating security checks to the developers. This has created a long history of vulnerabilities introduced by developer mistakes, particularly related to memory safety. Memory safe languages prevent these vulnerabilities by having a strong type system, performing safety checks during compilation, and inserting necessary runtime checks into the compiled code.

While memory safe languages such as Java, Haskell, and Python have long existed in the community, their reliance on a language interpreter prohibits their effective usage for developing systems software (e.g., operating system or middleware). Moreover, the interpreters of such languages are often themselves written in C/C++, which open up the system to memory corruption vulnerabilities, defeating the main security purpose of a safe programming language. It wasn't till recently (2015) that a safe systems programming language was developed: Rust. Another similar safe language is Go (circa 2012), but unlike Go, Rust does not have a language runtime, which makes it appropriate for systems programming. Thus, in this article, we focus on Rust.

Rust has a strong type system and provides native spatial and temporal memory safety at compile time, preventing, for example, buffer overflows and use-after-free vulnerabilities, respectively. Spatial memory safety in Rust is provided by strong typing of data, compile-time bounds checking of static data, and automatically inserted runtime bounds checking instructions for dynamic data. Temporal memory safety, on the other hand, is provided by the notion of ownership, which only allows one mutable reference[2] or multiple immutable references to exist to an object. This ensures that when the object is destructed, all references to it can be zeroized without the need for heavy-weight garbage collection, thus preventing temporal safety bugs such as use-after-free or double-free.

Like C, Rust has no true language runtime and can be compiled directly to processor instructions, making it ideal for implementing systems software. An evidence for this is the fact that Rust has already been used to build embedded operating systems like Tock OS and enterprise operating systems like Redox.

Despite all the benefits a safe programming language provides, a valid question can still be: do we really need it? In other words, can we fix applications written in C/C++, perhaps with additional sanitization (development-time analysis) or automatically inserted runtime checks (as done by SoftBound)? The answer becomes clear when we consider that unsafe languages like C/C++ have ambiguous cases that are impossible to secure with any automated analysis. For example, the C language specification does not distinguish between a pointer to the first element of a structure and a pointer to the entire structure. As such, there is no automated analysis that can establish a bound finer than the entire structure for a pointer than can potentially only intend to point to the first element of the structure. When the first element is a buffer, for example, it can overflow into the rest of the structure, corrupting function pointers in the process and resulting in control-hijacking vulnerabilities. Such vulnerabilities have been shown to be abundant and exploitable in real-world software [14]. As a result, runtime defenses either have false negatives or false positives (crashing benign code). Sanitizers also face coverage challenges; their analyses are incomplete and miss many vulnerabilities. In fact, vulnerabilities in reputable software packages are found everyday despite state-of-the-art sanitization being applied to them. The answer, therefore, is yes; safe programming languages are indeed required to prevent memory safety bugs. Approached that attempt to protect unsafe languages only provide partial protection at best.

A. Research Challenges

There are still important challenges to fully leverage the safety provided by Rust. In some cases, the restrictions enforced by the language are too restrictive for the developer. Two examples of this include complex data structures like a doubly-linked list and low-level interactions with hardware devices. In a doubly-linked list, the restriction that only one mutable reference can exist to an object is problematic; a doubly-linked list needs two for each object. For low-level interactions with hardware such as a memory-mapped input/output device (MMIO), the code needs to manipulate raw bits in configuration registers for which the strong typing of Rust is too restrictive.

Rust itself provides a mechanism to bypass the language's safety restrictions with the `unsafe` keyword. Code enclosed in an `unsafe` section will not be checked for adherence to the restrictions on raw pointers or ownership. This means that a complex data structure or code handling hardware devices can still be developed in Rust, but at a cost to security. Code enclosed in `unsafe` sections can be vulnerable to spatial or temporal memory corruption bugs, just like C/C++. More troublesome is the fact that `unsafe` code not only impacts unsafe sections, but also safe sections. An arbitrary raw pointer in `unsafe` Rust can maliciously modify the entire addressable memory space of the process, thus changing the behavior of even the safe sections of the code.

A major gap in the past and current efforts to built Rust-based OSes and utilities (such as Tock OS, Redox, Mesalock Linux, and uutils) is the widespread usage of `unsafe` sections in their codebase. An important lesson from these efforts is to try to minimize the amount of unsafe code or properly contain their impact.

Therefore, additional research is needed to either alleviate the need for unsafe Rust or properly sandbox/isolate unsafe sections from the rest of the code so that their bugs don't endanger the rest of the application. One important context to address this challenge in for MMIO devices [15]. An unsafe driver for an MMIO device only needs to access certain regions of memory (allocated for MMIO) and only certain parts of that region (allocated for that particular device). As such, it can be isolated to those parts using proper tagging by the tagged processor. A more complex policy can also distinguish between configuration registers and data regions to implement an even finer-grained isolation.

An orthogonal research effort has focused on formally verifying that while certain complex data structures (e.g., doubly-linked lists) are using unsafe code internally, they provide safe interfaces to the rest of the code. This is called interior mutability [16].

V. Fine-Grained Compartmentalization of OS

The third component of our vision is fine-grained compartmentalization of software stack. Nowhere is this more important than in the operating system. There are two main operating system designs in modern systems: monolithic kernels and microkernels[3]. Monolithic kernels (e.g., Linux, UNIX, and Android) include most of the important functionalities (e.g., file system, IPC, device drivers, etc.) in the kernel, while microkernels (e.g., MINIX, L4 and seL4) only provide the bare minimum functionality (e.g., virtual memory, scheduler, and basic IPC) in the kernel and push others (e.g., file system, application IPC, device drivers, etc.) to user space services. Both designs have some weaknesses, however. A monolithic kernel is not secure. A very large code base runs at the highest privilege level (often enforced by the ring model), thus any severe vulnerability in any part of the code often results in a complete takeover of the system. Moreover, there are many different developers and coding standards, and current monolithic kernels are as secure as their least secure component.

[2]A mutable reference is what is traditionally referred to as a pointer in some programming languages like C/C++.

[3]While other OS designs also exist (Unikernels, Exokernels, etc.), they are not as widely used as the two main designs and for the sake of the argument made in this article, they share many of the weaknesses of the two main designs.

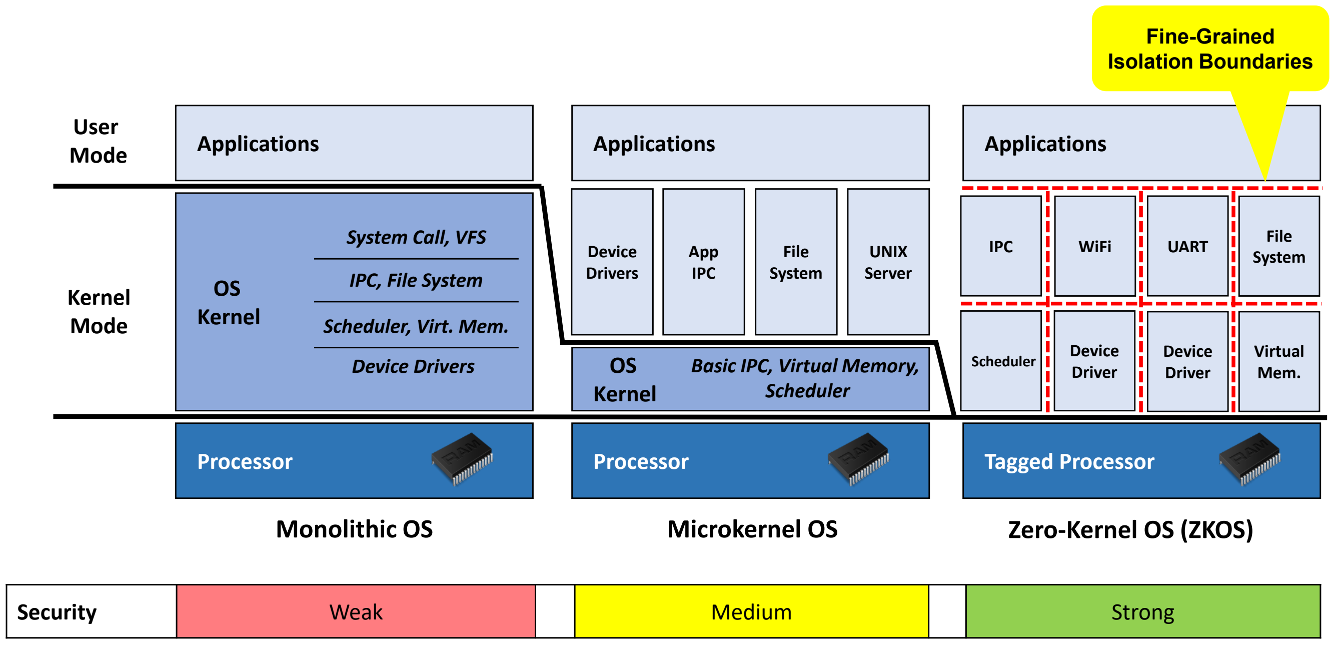

Fig. 3: The Zero-Kernel OS design compared with monolithic and microkernels.

The microkernel design improves upon the OS security by virtue of pushing services into the user space, the security provided by microkernels is not sufficient. First, a vulnerability in the kernel can still compromise the entire system. Second, microkernels do not provide additional protection for the services themselves. A buggy driver can still interfere with other services running on the system. As such, microkernels can only provide coarse-grained isolation. For example, virtual machines (VMs) running on top of seL4 can be isolated from each other, but each VM is subject to compromise internally. Moreover, microkernels also face performance challenges compared to monolithic kernels in general[4]. Most system calls in this model require multiple context switches (e.g., from application to service, from service to kernel, from kernel to devices, and all the way back), which imposes a large overhead.

A new design that can significantly improve security, while not sacrificing performance is what we call the Zero-Kernel Operating System (ZKOS) [17]. In this model, each module or service in the OS is isolated in its own compartment (Figure 3). Each module's privilege is then reduced to the bare minimum necessary for it to perform its job. This design importantly relies on the tagged architecture for seamless enforcement of isolation and privilege. Privilege in this context refers to the read/write/execute permissions to any region in memory as well as being allowed or not allowed to execute specific instructions. Note that because each memory word is tagged, switching between two privilege levels happen transparently and without additional operations by simply executing their corresponding instructions. For example, the instructions associated with the memory manager can be tagged with one type of tag (e.g., `MEMMGR`), which ensures that the memory manager can only modify certain parts of memory (e.g., heap metadata), while a driver can be tagged with another type of tag (e.g., `DRVR1`), which only allows it to access the memory-mapped region associated with its corresponding device. This change of privilege happens automatically when the OS executes the instructions associated with the driver or those associated with the memory manager.

[4]This rule may not apply on a one-to-one basis because individual implementations of microkernels can dedicate significant effort to optimizing performance, but generally speaking, additional context switches cause performance degradation.

This design has a number of important implications. First, the operating system will no longer have an omni-privileged piece of code in it (hence the name "Zero-Kernel"). This is unlike a microkernel design in which the kernel is still running at the highest privilege level. Second, the fine-grained enforcement of isolation (word granularity) obviates the need for page-level permissions (and thus paging altogether) and the coarse-grained ring model that do not provide sufficient protection. In some studies, the overhead of paging alone is estimated to be 20-30%, thus such a design can also save this performance. Third, because of the fine-grained isolation and privilege enforcement, the traditional user space/kernel space division can also be eliminated, thus allowing applications to run in the same nominal space as the OS modules (note that the separation is actually being enforce at a finer granularity, so this does not make the system less secure). This means that system calls can become almost as efficient as function calls because no heavy-weight context switching will be needed, thus making the system more performant.

The ZKOS design, in effect, turns OS security from an all or nothing battle into a series of contests, each of which is easier for the defender to win.

An astute reader might wonder where ZKOS's performance comes from; in other words, what is being traded-off with performance in a ZKOS design? The answer is silicon area in the form of tag space in memory and the policy engine in the processor. The tagged architecture is what makes such a design possible and efficient.

We note here that fine-grained isolation and compartmentalization have been studied rather extensively in various contexts, including in OSes. Separation kernels, unikernels, the least privilege separation kernel (LPSK) [18] are some of the attempts in the community to implement compartmentalization for the OS. What makes ZKOS different from these efforts is the granularity of its compartmentalizaion and its reliance on tagged architectures that eliminates the need for an omni-privileged software layer. Separation kernels provide coarse-grained (VM level) compartmentalization. Unikernels, while they minimize the attack surface of the system by not including unnecessary OS services, they do not provide isolation among the OS services that are included in the image. Finally, the LPSK, while providing fine-grained (service level) compartmentalization, it relies on an omni-privileged kernel for limiting the information flow among various services.

A. *Research Challenges*

Although this compartmented OS design looks promising, there are still important research problems to be solved.

Not all tagged architectures have large tag spaces or an expressive policy language. ARM MTE, for example, only has four bits of tags. How to compartmentalize an entire OS with such a limited tag space remains a research problem.

Moreover, when OS compartments exchange data, that data either needs to be re-tagged, or needs to be assigned a third tag with a dynamic policy that allows the receiver module to access it and denies the sender from accessing it. Neither feature (efficient re-tagging or dynamic policies) is supported or matured in existing tagged architectures. More research is thus needed in this area.

Compartmentalization can also be built in multiple different ways that trade-off policy complexity, performance, granularity of compartmentalization, and tag size. Studying and understanding such trade-offs and picking an optimal design for each use case thus remains an open problem.

Finally, compartmentalizaion can further be extended into the application space, isolating various modules of a complex application such as a web browser or a web server into their own compartments. Additional work is needed in this area to study appropriate designs for such compartmentalizaion.

VI. Where to Start?

As mentioned earlier, a number of past and on-going projects at our laboratory have started executing the vision laid out in this document. We have focused on embedded systems as the first target for implementing the pillars discussed in this article. Embedded systems often are not as feature-rich as enterprise systems, and do not require the same level of generality in the services they provide. As such, they are an easier target for a new design. We have specifically focus on autonomous systems as the application domains of choice for our secure embedded system design. Expanding such a computer system to more feature-rich tactical and enterprise applications remains as future work.

In addition, under each pillar, we have focused on easier-to-solve gaps first. For example, unsafe Rust is easier to contain for MMIO devices because it is already known what memory regions such code should and should not be allowed to access, whereas minimizing the need for unsafe Rust in the general case is much harder to achieve. Part of our effort, therefore, has focused on proper containment of unsafe Rust code that interacts with MMIO devices [15].

VII. Conclusion

In this article, we charted a vision for a cybersecurity moonshot, a radical, but feasible redesign of computer systems in which important security properties are inherent. This vision is not the result of research from one team or one effort; the components of our vision are the culmination of long running projects and research efforts from many independent groups of researchers in academia, research laboratories, and industry. This article described how these components could fit together to achieve ambitious goals in cybersecurity, and why such a moonshot is indeed needed and timely.

VIII. Acknowledgment

We sincerely thank David Martinez, Bob Bond, Marc Zissman, and Roger Khazan whose support and guidance has made this effort possible. In addition, we would like to thank Howie Shrobe, Nathan Burow, and the members of the Resilient Mission Computer (RMC) project who contributed to many of the ideas and research challenges outlined in this article.

References

[1] "Statistics and Market Data on Cyber Crime & Security," https://www.statista.com/markets/424/topic/1065/cyber-crime-security/, 2020.

[2] M. Miller, "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape," 2019.

[3] D. J. Boudreaux, "What's Your Moonshot?" https://www.mercatus.org/videos/whats-your-moonshot, 2020.

[4] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie *et al.*, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 20–37.

[5] G. T. Sullivan, A. DeHon, S. Milburn, E. Boling, M. Ciaffi, J. Rosenberg, and A. Sutherland, "The dover inherently secure processor," in *2017 IEEE International Symposium on Technologies for Homeland Security (HST)*. IEEE, 2017, pp. 1–5.

[6] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 48–62.

[7] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 552–561.

[8] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 276–291.

[9] J. Seibert, H. Okhravi, and E. Söderström, "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 54–65.

[10] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 901–913.

[11] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[12] B. Zhou, A. Gupta, R. Jahanshahi, M. Egele, and A. Joshi, "Hardware performance counters can detect malware: Myth or fact?" in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018, pp. 457–468.

[13] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 245–258.

[14] R. Gil, H. Okhravi, and H. Shrobe, "There's a hole in the bottom of the c: On the effectiveness of allocation protection," in *2018 IEEE Cybersecurity Development (SecDev)*. IEEE, 2018, pp. 102–109.

[15] A. Huang, "Software defined memory ownership system," Master's thesis, MIT, 2020.

[16] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "Rustbelt: Securing the foundations of the rust programming language," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, p. 66, 2017.

[17] J. Restivo, "A zero kernel operating system: Rethinking microkernel design by leveraging tagged architectures and memory-safe languages," Master's thesis, MIT, 2020.

[18] T. E. Levin, C. E. Irvine, and T. D. Nguyen, "A least privilege model for static separation kernels," Naval Postgraduate School Monterey CA Center for Information Systems, Tech. Rep., 2004.