One Giant Leap for Computer Security

Hamed Okhravi, Nathan Burow, Richard Skowyra, Bryan C. Ward, Samuel Jero, Roger Khazan, and Howard Shrobe | Massachusetts Institute of Technology

Today's computer systems trace their roots to an era of trusted users and highly constrained hardware; thus, their designs fundamentally emphasize performance and discount security. This article presents a vision for how small steps using existing technologies can be combined into one giant leap for computer security.

Computer security currently consists of bringing order out of an infinite sea of "raw seething bits,"[1] a Sisyphean task that is doomed to failure without a fundamental shift in the way we approach security. The current state of security is rooted in legacy design decisions in computer architecture, operating systems (OSs), and programming languages that have continued largely unchallenged in commodity systems since ~1970, a fact that is particularly ironic in light of the rapid pace of innovation in early systems, such as the Project on Mathematics and Computation (Project MAC) in 1963, Multics circa 1969, and Programmed Data Processor-11 (PDP-11) in 1970. While these and other systems contributed many novel design ideas that we rely upon today, e.g., time-sharing, virtual memory, dynamic linking, and hierarchical file systems, they were also built in a different era, a time when networking was just coming into existence and all users were trusted researchers. With security threats not yet on the horizon and a trusted user base, system designers emphasized performance to maximize the computation possible on the highly constrained processors and memory of the day.

As a direct consequence of the paramount importance of performance in early systems, certain design decisions were made that directly result in security vulnerabilities today. Processors manipulate raw bits without any metadata about the objects they represent and lack any core security features other than virtual memory, which was originally added to mask limitations in the memory available to processes. Consequently, there is no concept of a buffer overflow at the instruction set architecture (ISA) level; such semantics are imposed by programmers and are not fundamental to the machine. All bits are the same to the processor, whether they represent code, data, or pointers to the programmer. OSs remain monolithic, with no isolation or separation of privileges within the OS. Indeed, privilege separation is possible only through the hierarchical ring-oriented privilege model, which results in overprivileged code in the ring-zero OS. Today's systems programming languages, C and C++, came of age in an era when compilers were rudimentary and deliberately provided only minor abstractions across assembly code. As a result, they provide little static verification and have no runtime system to verify security properties. Indeed, these languages are notorious for leaving responsibility for security solely to the (highly error-prone) programmer.

The combination of processors that are only aware of raw bits, overprivileged monolithic OSs, and simplistic, low-abstraction programming languages has created highly vulnerable systems that are prone to exploitation. The past 25 years have seen an arms race

Digital Object Identifier 10.1109/MSEC.2020.2977586
Date of current version: 22 April 2020

 Copublished by the IEEE Computer and Reliability Societies

between attackers punching new holes in software and defenders desperately plugging these holes. Indeed, the vicious cycle of conflict between attackers and defenders in system memory has been termed an *eternal war*.[2] While principled solutions to many of the fundamental flaws in modern systems are known, realizing these solutions in practice has, to date, proven impractical. The time has come to end the eternal war in memory by embracing a new systems architecture where security is a first-class citizen alongside performance, with the processor, OS, and programming languages all cooperating to guarantee security.

In this article, we present a vision for a more secure computer design. Coming from multiple past and on-going projects in our research team, this vision outlines what a computer system design would look like if we were not bound and constrained by legacy decisions. In other words, we try to imagine a computer system with security as its core design principle, uninhibited by compatibility and optimality constraints. While some previous efforts, such as DARPA's Clean-Slate Design of Resilient, Adaptive, Secure Hosts (CRASH) and Mission-Oriented Resilient Clouds (MRC) projects, were pioneers in the clean-slate redesign of a computer system, due to their constraints, they still heavily relied on existing system components with legacy designs (e.g., UNIX-like OSs and C/C++ languages). Here, we envision the next natural step and think beyond legacy designs, envisioning, instead, a moonshot that results in the next giant leap for computer security. The key to our approach is focusing holistically on the entire software stack instead of on the small steps that can be taken to secure individual software layers.

We envision a future where security is fundamental to computer design instead of being imposed from above on top of an unstable foundation of raw bits. We identify three key pillars of computer security that should be inherent to the architecture of any secure system and cut across all layers of the system design: 1) safety within code modules, i.e., language-based safety; 2) safety between code modules, i.e., input–output (I/O) validation; and 3) isolation of code modules, i.e., least privileges for each code module. Figure 1 illustrates such a secure architecture and highlights our cross-cutting design principles. Language-based security applies to all code modules, be they applications, libraries, or OS modules. I/O validation occurs anytime data crosses a logical boundary in the system, either from the OS to an application, as at label 1 in Figure 1, or hardware to the OS, as at label 2. Least privilege is achieved by isolating both applications from each other and the functionality within the OS, as seen at label 3 (dotted lines denote isolation boundaries). As shown in Figure 1, these pillars are all enabled by semantically aware processors that are cognizant of more than raw bits.

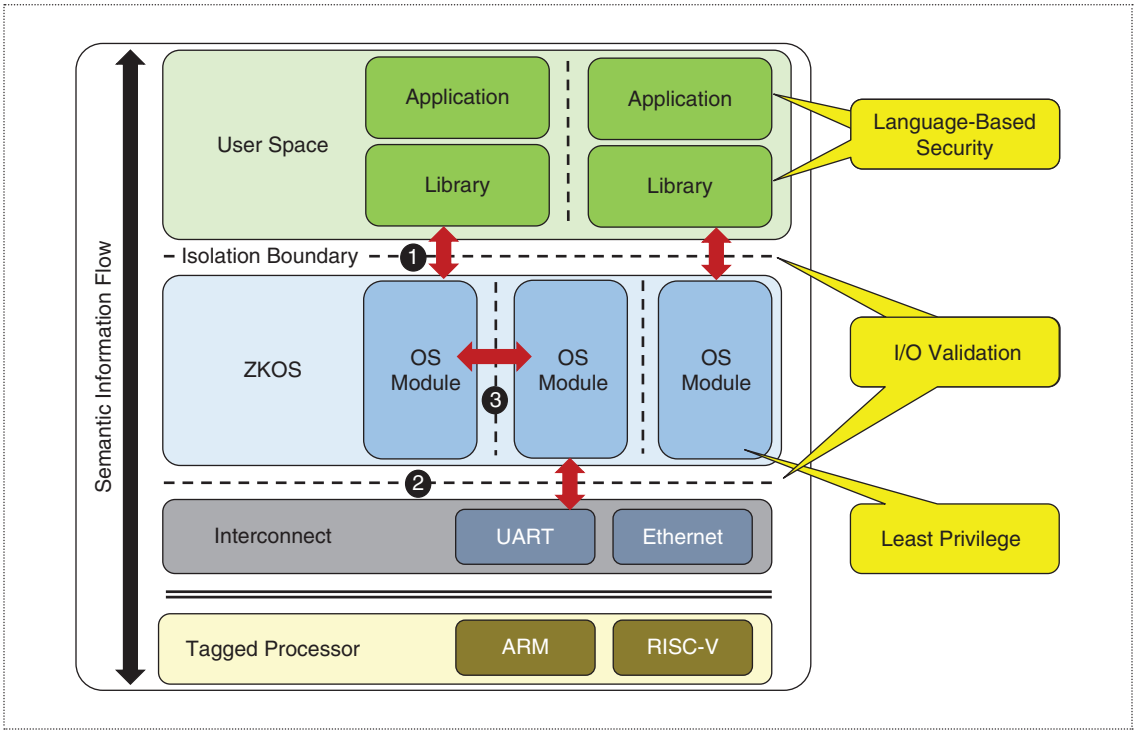

Figure 1. The proposed secure-by-design architecture. ZKOS: zero-kernel OS; UART: universal asynchronous receiver/transmitter.

- *Language-based security*: The fundamental properties of language-based security are memory/type safety, which at a high level require that memory be used only as the programmer intended. Memory/type safety violations arise out of the semantic gap between existing ISAs, which regard bits as bits, and programming languages, which organize memory in terms of typed objects. Research in this area has focused on creating safe languages, with Rust and Go being two prominent examples that are in widespread use. The "Language-Based Security" section discusses the features of these languages and challenges in their design and implementation.
- *I/O validation*: Bits of information within a system can be generated programmatically or enter from external sources, such as user input or a network. While memory/type safety handles the first case, I/O validation is required to ensure that bits from external sources are correctly typed and have valid bounds. Further, I/O is the traditional entry point for attackers and so is a natural location for additional defenses. We imagine new defenses at the system boundary that will help prevent attacks such as Heartbleed, which exploited a failure of the system to accurately track bounds for a buffer passed across the network. Further, as discussed in the "I/O Validation" section, semantically aware hardware can solve I/O validation for all communication within the system, most notably interprocess communication.
- *Least privileges*: Isolating code modules, i.e., reducing each to its set of least privileges to accomplish its job, is fundamental to providing compartmentalization for computer systems. By isolating components, a compromise of any given component does not compromise the system as a whole. Further, isolation enables us to enforce that each component has access to the minimum possible set of resources for its function, i.e., least privileges. Current systems provide isolation only at the process level through virtual memory. Further, commodity OSs and even research microkernels are built on top of a privileged component with access to the entire system. The privileged component is an artifact of the ring model of isolation at the hardware level. The "Least Privileges" section presents our vision, which replaces this model with a single flat memory space where isolation is provided at arbitrary granularities by hardware, building on our existing work on zero-kernel OSs (ZKOSs).

Foundational Concepts

Here, we define memory and type safety, the provision of which is a key component of our vision. Further, we introduce tagged architectures, which can be used to eliminate the semantic gap between source code and machine instructions. We later show how tagged architectures uniquely enable our vision in the "I/O Validation" section.

Memory and Type Safety

Early programming languages, such as C and C++, are designed to enable the highest possible performance of an application. Generally, this is accomplished by minimizing the gap between the operational semantics of the source language and the actual semantics of the underlying machine architecture. This gives developers substantial power to take advantage of low-level program operations (e.g., pointer arithmetic to access struct fields) but also fails to enforce two critical security properties: memory and type safety. C permits the creation and dereferencing of pointers to arbitrary memory addresses, for example, which potentially enables any mapped area of memory to be corrupted by a bug. Microsoft recently disclosed that roughly 70% of the vulnerabilities in its software are rooted in such bugs.[3] C++ is vulnerable to type confusion, where, e.g., an object of an unrelated class is used for a virtual dispatch. Both memory and type safety violations enable an attacker to manipulate the program's state, building and executing so-called weird machines[4] that enable adversaries to treat the program as a computer, which, in turn, executes programs in the form of exploits that, e.g., spawn a command shell.

Memory safety. Memory safety violations arise when a pointer is misused to access a nonprogrammer-intended object. For instance, a write overflows an array into an adjacent object, or a pointer to a freed object is used to write to memory that has been reallocated to a new object. Such buffer overflows and use-after-free vulnerabilities violate the two tenets of memory safety: 1) spatial or bounds safety, which requires that all pointer dereferences are in bounds of the referenced object and 2) temporal or lifetime safety, which requires that the referenced object be allocated. Formal models of memory safety assign pointers capabilities to access memory to encode the bounds and lifetime of the underlying object. A dereference is only valid if a pointer currently holds the capability for the underlying memory object.

Languages that provide memory safety inherently require some runtime checks. Object bounds, particularly for heap objects, can be determined at runtime based on, e.g., user input. Optimizing the required set of checks has seen significant research interest, as it directly impacts performance. Temporal safety frequently depends on garbage collection, which is a research field in its own right. Alternately, languages such as Rust enforce temporal safety at compile time

through their type system, at some loss of expressiveness; see the "Memory and Type Safety" section.

Type safety. While memory safety concerns itself with when and where bits are written in memory, type safety concerns itself with how those bits are interpreted. For instance, is a word of memory a float, an int, or a pointer? Type safety also encompasses the size of the data, e.g., a 32- versus a 64-bit integer. Indeed, the programming language community has developed strongly typed languages that provide spatial safety through their type system. Object-oriented language features, such as inheritance and virtual dispatch, provide additional challenges for type safety, as they introduce an extra layer of abstraction and type information on top of the program's data. C++ is particularly vulnerable to type confusion attacks wherein an attacker causes an illegal downcast, uses a memory corruption to change the type of an object, or causes an arbitrary region of memory to be interpreted as an object of some class.

To prevent type confusion attacks, programming languages have adopted type systems, which can be classified along two different dimensions: 1) time of enforcement and 2) rigor of the type system. Type safety can be enforced statically or dynamically. Static enforcement requires the types of all objects be verified at compile time and not change during execution. In contrast, dynamic enforcement allows variables to change types at runtime (as is common in, e.g., Python), at the expense of performance overhead. Static enforcement can be overly strict and limit program expressiveness to provide guarantees. For instance, downcasts in object-oriented languages are frequently useful but impossible to verify statically without strong assumptions about aliasing.

Regardless of how the type system is enforced, it can be either strong or weak. Modern languages, such as Rust and Go, are strongly typed, whereas C/C++ are only weakly typed. Strongly typed languages check type casts either at compile time or at runtime. Rust and Go are more efficient than the runtime information that is available in C++ and perform most of their checks at compile time. Modern compilers have made strongly typed languages significantly easier to use for developers by including strong type inference algorithms, which enable the compiler to learn the correct type for most variables without the programmer specifying the type. As strongly typed languages are more secure and increasingly more user friendly, we believe that they are here to stay.

Tagged Architectures

Tagged architectures provide hardware support for software security policies. Mondrian[5] is an early example that focused on providing read, write, and execute (RWX) permissions at the word level and allowing different threads to have different permissions on the same word of memory. Modern tagged architectures allow the enforcement of general-purpose security policies, such as memory and type safety. These systems extend hardware with additional metadata "tags" about memory and instructions at the granularity of individual words in the memory, which enables the enforcement of arbitrary policies across instructions and data. In essence, rich semantic information about the expected behavior of the code can be encoded in the processor via tags and validated at runtime. A variant on this notion is capabilities, or unforgeable, immutable tokens that grant the ability to perform operations. The notion of tags/capabilities has seen significant academic [Capability Hardware-Enhanced RISC Instructions (CHERI)] as well as government [DARPA CRASH/System Security Integration Through Hardware and Firmware (SSITH)] interest.

CHERI[6] is a tagged architecture capability system that extends a 64-bit, million instructions per second ISA to provide capability-based memory safety by essentially replacing pointers with capabilities. A capability in CHERI looks a lot like a fat pointer; it is a 265-bit entity providing base, length, and offset fields as well as permission bits and object type information. However, CHERI assigns capabilities to entities other than pointers, such as processes and file permissions, making the system robust and general purpose. CHERI capabilities can be used for memory and type safety, isolation, and access control, among others.

The Dover inherently secure processor[7] is a tagged architecture that grew out of the DARPA CRASH and SSITH programs. Like most tagged architectures, Dover extends each word in memory with metadata. One interesting aspect of Dover's design is the decision to separate it into two cores. The first core is the primary application CPU that operates on the data themselves. The second is the policy execution core that computes the policy across that metadata. This decision enables any CPU to be used as the application CPU with minimal modification; Dover is currently using RISC-V. Dover supports the enforcement of general policies in hardware, including in particular memory/type safety and isolation, which are of critical importance to our vision. We explore how this architecture can be leveraged to create secure systems in the "Support From Hardware" section after fully laying out our vision.

Language-Based Security

Modern languages provide substantially more powerful abstractions than C/C++ do, including static features, such as strong type systems, and runtime features, such as garbage collection, which can collectively be thought

of as language-based security. Such abstractions remove responsibility for low-level details from the programmer and seek to guarantee that if the code compiles and executes, it will be memory and type safe, preventing attackers from being able to hijack the application to execute their weird machines. Static checking guarantees that a program does not suffer from a certain class of bugs, while dynamic (runtime) checking guarantees that a program will halt upon encountering that class of bugs. Of course, neither kind of check is free. Static checks limit functionality in Rust, forcing reliance on unsafe code for common data structures, such as doubly linked lists. Dynamic checks, such as Go's garbage collector, add performance overhead.

Language-based security is a powerful tool for securing software because it removes all reliance on developer-inserted manual checks. As long as the compiler/runtime are correct, any code that compiles and runs is free of some of the most dangerous classes of bug. Regrettably, however, the previous statement is not entirely true on two fronts: 1) language design and 2) language implementation. Language design provides safety by making developers rely on abstractions when writing their code. Such abstractions can limit program functionality, particularly for low-level systems code that implements custom data structures or accesses low-level hardware details. To mitigate these issues, many languages provide an escape from the abstractions so developers can write arbitrary code. Such "unsafe" code regions break the security guarantees of the language. The implementation of "safe" languages can also expose flaws, particularly in the correctness of the runtime system, which is often written in an unsafe language. An additional challenge when implementing a safe language is performance: Developers and users demand code that runs as fast as existing languages (plus or minus 5%[2]).

Language Design Challenges

Some software cannot provide its intended functionality while remaining within the constraints of language abstractions. To address this, many memory and type safe languages provide an "escape hatch" out of the abstractions that gives the developer increased control of the process but at the cost of weakened or eliminated guarantees with respect to bug-freeness. Rust, for example, provides the unsafe keyword. Within a block labeled *unsafe*, C-style pointers can be used to manipulate memory in arbitrary ways. Note that this doesn't mean that unsafe code necessarily has, for example, memory corruption vulnerabilities. Rather, the compiler or runtime can no longer guarantee the absence of such vulnerabilities, and the onus is placed back on the developer to write correct code. If a bug is present, its effects are not necessarily restricted to the unsafe region and data it was directly handed. The bug may enable the corruption of data or code throughout the entire process image, even if all other memory accesses are verified safe by the compiler. An open challenge in language safety research is how to provide a spectrum of security/control options that bound the damage bugs in unsafe code rather than a binary safe/unsafe tradeoff where a single line of unsafe code could corrupt the entire otherwise-safe process.

Complex data structures. A common example of the need to violate language abstractions is when implementing complex (e.g., graph-like) data structures in a language such as Rust, which provides compile-time enforcement of temporal memory safety. Rust's ownership model permits a memory object to have either an arbitrary number of read-only references (also called *pointers* in other languages) or a single read/write reference. This does not allow data structures such as double-linked lists or graphs with cycles, as both require multiple read/write references to the same object. Rust's philosophy when addressing this kind of problem is the notion of safe application programming interfaces (APIs) to unsafe code. The intent is to encapsulate necessarily unsafe code with a safe interface such that external code can use the unsafe code only in the way intended by its developer. Rust uses this approach to provide an automatically reference-counted (RC) pointer, for example. Unsafe code is needed to manage (among other things) raw memory for object wrapping and unwrapping. Its API allows only safe usages of the RC, however, such as attempting to unwrap an object from the RC container and succeeding only if there is a single reference remaining.

Safe interfaces to unsafe code are one mechanism for providing a spectrum of security/control options to developers. RC objects are not as secure as standard references, as they are not thread safe. They are, however, much more secure than relying on the developer to manually track references and handle deallocation on his or her own. That said, this approach comes with its own challenges. First, the API must, indeed, be safe. This means that safety guarantees that hold across a program written in only the core Rust language still hold across code using the language extension defined by the new API. Second, the unsafe code behind that API must actually be bug-free. If either of these properties fails to hold, the application may be at risk of memory corruption vulnerabilities or other dangerous bugs.

One approach to ensuring that safe interfaces to unsafe code are actually safe is to leverage formal verification of the API and unsafe code. This technique mathematically proves that the implementation of a specification has the same semantics that the

specification has. This enables provable bug-freeness but, in general, is very time and labor intensive. The seL4 formally verified microkernel, for example, consists of approximately 10,000 lines of code and took more than 20 person-years to fully verify. The Rustbelt[8] project has made promising inroads on verified Rust APIs, however. Rustbelt is an ongoing effort to formally verify that the APIs encapsulating unsafe code in libraries are, indeed, safe (i.e., that all of Rust's guarantees hold across the compiled software despite the presence of unsafe blocks). Several key APIs from the Rust standard library have already been verified, including the RC object discussed previously. Note that Rustbelt does not verify that the developer has written bug-free code. It verifies that the API semantics are safe when composed with the larger program's semantics, not that the implementation of a specific API is bug-free.

Separate complication. For languages that statically guarantee memory and type safety, such as Rust, separate compilation poses a fundamental design challenge, as their guarantees require whole program analysis. Such analysis is impossible, even assuming that all applications are statically linked, in at least two cases: interprocess communication and system calls to the OS (Figure 1, label 1). Even languages with runtimes that guarantee safety fail to transmit the necessary information across the process–kernel boundary. We discuss this issue in depth in the "I/O Validation" section, including how hardware can be leveraged to preserve semantic information across these boundaries.

Low-level operations. OSs and many embedded applications interface directly with the hardware. Context switching, interrupt handling, memory-mapped I/O, and register operations are all architecture-specific algorithms that require inline assembly instructions in the source code. This is obviously unsafe and can lead to memory corruption. Unfortunately, since architecture-specific instructions are outside of a source language's semantics, language safety mechanisms cannot be used to protect inline assembly. Worse, source code that would otherwise be protected may become corrupted due to a vulnerability in the inline low-level operations.

Formal verification may be an option for protecting inline assembly code. The assembly is designed to accomplish a single well-defined task, such as context switching. It is also generally fairly short, on the order of 10 s of lines of code. While formal verification is time-consuming, it may be tractable for such small snippets. Another approach is to limit the damage that bugs in inline assembly can cause. Memory that is not intended to be modified by the low-level operations can be unmapped or rendered inaccessible during execution of that code fragment.

Language Implementation Challenges

Implementing the runtime required for a safe language presents two challenges: 1) ensuring that runtime itself is not buggy and 2) ensuring that the language meets the performance demands of commodity software vendors.

Runtime vulnerabilities. Languages that provide both spatial and temporal safety are, in an ideal world, immune to memory corruption attacks. In reality, the quality of the memory safety guarantee depends on the quality of the language's compiler and runtime. Examples of memory safe languages include Java, Rust, Go, and nearly all interpreted languages, such as Python and JavaScript. Not all implement it in the same way, however. Java relies primarily on runtime enforcement of both the spatial safety (e.g., array bounds checking) and the temporal safety (via garbage collection). Rust, conversely, relies almost entirely on compile-time temporal safety checks via its ownership model, with some instrumentation inserted at runtime for spatial checks. Attacks on memory safe languages, e.g., Java, do not exploit memory corruption bugs in the programmer-developed source code. Instead they exploit the implementation of the Java runtime, which is written in the memory unsafe C language. Languages, such as Rust, with minimal runtimes suffer from design limitations, as previously discussed.

Performance. An oft-cited reason for why legacy languages persist to this day despite their known security shortcomings is that they enable developers to eke out the highest possible performance with the smallest possible footprint. This is especially relevant in the embedded and real-time domains, where platforms are highly resource constrained and may need to process inputs in a time deterministic manner to meet real-time constraints. The performance argument is valid for languages that rely on heavy-weight runtime enforcement of checks to provide memory or thread safety. Garbage collectors, for example, are leveraged by Java [and Java virtual machine (JVM) languages, in general], Go, and most interpreted languages. They provide automatic enforcement of temporal memory safety by tracking object references and de-allocating objects that are no longer in use. This comes at the cost of unpredictable timing and memory usage properties of the application, as most garbage collectors will batch operations through time. This causes sudden slowdowns when the batch is processed as well as sudden reductions of the memory footprint that will start increasing again after collection.

Garbage collection is not the only way to provide temporal memory safety, however. Rust's ownership model provides a type system that enables the static analysis of object references. The compiler can thus identify when an object should be deallocated as well as whether there exist any bugs that could cause invalid pointer dereferences at runtime, and, if so, emit an error. Barring use of the unsafe keyword, any Rust program that compiles will be free of temporal memory safety violations. These compile-time checks negate the need for runtime enforcement. Rust has performance comparable to legacy C code and security properties far exceeding legacy C and is thus a promising language for embedded systems programming. Tock OS,[9] for example, is an embedded OS written entirely in Rust and designed to run on resource-constrained Internet of Things devices.

I/O Validation

Regardless of how expressive and powerful a language's type system is, any nontrivial program will, at some point, need to process untyped, untrusted input from the external environment or across process boundaries. Soundly assigning types to data of unknown provenance is an open question at best and undecidable at worst. Thus, developers cannot rely on the language to assign types to inputs. They are confronted with the challenge of deciding how to manually verify that incoming data are syntactically and semantically valid. Once those checks succeed, the developers must still use a potentially unsafe casting operation to assign the data a hopefully correct type and then perform a potentially unsafe copy to a hopefully large enough buffer. If those checks are insufficient and malicious data are successfully cast into a language's type system, many security properties may no longer hold.

To illustrate the difficulty of typing incoming data, consider a common encoding for network packet fields: the type-length-value format. The type and length fields provide instructions to the receiver as to how the value field should be treated. If these are maliciously chosen, the attacker can induce a variety of effects. In memory unsafe languages, an inaccurate length field could cause adjacent memory to be read from or written to. This was the mechanism exploited by the well-known Heartbleed attack. This problem is not limited to external hardware interfaces. Consider, for example, a structure containing a memory reference that is populated based on input data (e.g., in a loader preparing a program on disk for execution). If the data are encoded differently than what the program expects (e.g., in little versus big endian), the memory reference will point to a potentially attacker-chosen location and can be leveraged to violate memory safety. Malicious processes can use similar type confusion attacks against a host OS using its system call interface and against other processes via interprocess communication. This problem will exist as long as incoming data are untyped and cannot be validated short of developer-inserted checks.

While the problem of typing unknown data is very difficult to address in the general case, there are opportunities to leverage preexisting information to automate the input validation. Even for external, e.g., network or peripheral device inputs, the compiler has access to the intended type of input data. In many cases, it may be possible to automatically generate a parser for that type and instrument the emitted binary such that a cast succeeds only if the input data are parsed successfully (Figure 1, label 3). This does not mitigate attacks on the semantic validity of the data, but it eliminates the low-hanging fruit of malformed input. In the case of intrasystem communication (including system calls and interprocess communication), the issue is not generating type information but retaining it across user-defined barriers in the system. In such cases, redesigning the hardware and OS to support preserving type information will be effective.

Least Privileges

Least privilege is a well-known security principle that is commonly applied to file-access permissions and other role-based access-control schemes. Unfortunately, this principle is not commonly applied to code and data within applications or modules within the OS. Adopting memory safety (see the "Language-Based Security" section) moves applications closer to the least privilege ideal as access now requires a programmer-created reference. OSs, however, pose additional challenges, as they have direct access to and control over the sensitive machine state. To the best of our knowledge, developing a performant and secure application of least privileges to OSs by compartmentalizing them is an open research challenge.

OSs present both a design and an implementation challenge for enforcing least privileges. The design challenge comes from the OS being the abstraction layer across hardware that presents a uniform interface to applications across many different architectures. Consequently, the OS has visibility into the state of the entire machine and is responsible for managing memory, peripherals, and other low-level details. As such, it is also the natural repository for "privileged" operations that change the hardware state or otherwise affect the correct operation of the entire system. Many of these privileged operations are independent of each other, meaning that today's monolithic kernels do not respect the principle of least privileges. Even research microkernels that attempt to split traditional OS services into unprivileged user-level processes have overprivileged

cores at their heart. Providing isolation and minimizing the privileges of any region of code is the core design challenge for tomorrow's secure OSs.

Even with a sound OS design, a secure implementation is beyond today's technologies and will require advances in hardware and programming language design. Despite advances in memory safe programming languages, C remains the language of choice for OSs. This is partly the result of legacy OSs being written in C for historical reasons. Beyond that, however, there are valid technical reasons for using C. OSs frequently reach down into the gory details of the hardware architecture, directly manipulating registers, using memory-mapped I/O, handling context switching and interrupts, and so forth. Such tasks require unsafe code regions, at a minimum, in today's safe languages. Given that porting to a safe language would still require constructs whose safety cannot be guaranteed, developers have largely stuck with C.

OS Design

The landscape in which OSs are designed has fundamentally changed. With 64-bit address spaces and ample memory in enterprise systems, it is feasible to imagine a radically different system architecture, which we call *ZKOS*,[1] that respects the least privilege principle. ZKOS provides the isolation of code and data regions, thereby enabling compartmentalization and least privileges. Building on this foundation, ZKOS eliminates the kernel/user space divide, and all processes operate in a single flattened address space. Doing so is possible because modern tagged architectures[6,7] enable isolation/privilege control, with significantly richer semantics than the RWX provided by prior work,[5] at the byte level. Consequently, memory- and process-management data structures, for example, no longer need to be hidden in the kernel but can be individually protected. This eliminates the hidden 33% performance overhead cost of virtual memory and context switching identified by singularity.[13]

Because the security of the ZKOS lies in the cooperation of the OS with the underlying tagged architecture, correctly modeling privileges and isolation in the OS code and enabling it to interact with other processes to provide isolation as a service is a key challenge. Significant analysis work is required to divide kernels into minimal regions of functionality while still providing the expected performance and functionality. Similar to seL4, the tag policies themselves should then be formally verified to ensure they are providing the desired privilege separation and isolation of OS components or modules (Figure 1, label 2). Providing isolation as a service is a challenge because it requires the OS to be able to actively mutate tags on memory to change permissions. Suddenly, the security policy is no longer static and determined by the compiler tool chain but dynamic and thus exposed to malicious actors. Consequently, securely providing isolation as a service remains an important challenge in developing OSs that work with tagged architectures to maximize security.

Note that the ZKOS as we envision it fundamentally relies on tagged architectures. Consequently, ZKOS can be implemented in tomorrow's more secure languages that have been designed with hardware support in mind (see the "Hardware Support for Language-Based Security" section). Such languages will eliminate the implementation difficulties faced by today's secure languages around interacting with hardware. However, such technologies require significant research and development before they are ready to deploy, whereas memory and type safe languages, such as Rust, are already here and can be used to improve OS security while more advanced technologies mature.

OS Implementation

We should aspire to have language safety guarantees extend throughout as much of the system as possible. Given this observation, implementing more secure OSs in today's memory safe languages, warts and all, is just as important as designing new OS approaches to work with tomorrow's languages and architectures.

The approach espoused by the Tock OS, implemented in Rust, is to decompose the OS into a kernel, where all unsafe functionality must reside, and many capsules, which implement specific functionality entirely in safe code. The interface between the kernel and the capsules is well defined, and this architecture seeks to limit the amount of unsafe code in the system. However, the kernel and capsules operate in the same address space, and therefore vulnerabilities in the unsafe code of the kernel can corrupt memory in the capsules, thereby violating the assumptions upon which the memory safety guarantees are built.

An alternate approach to rewriting the OS in a safe language is to insert a small, trusted shim below the OS.[10,11] The shim is solely responsible for managing memory permissions that can be used to isolate different parts of the system. Notably, this approach enables the enforcement of least privileges, even on impoverished, i.e., embedded, systems that lack memory management unit (MMU) support. The shim can provide isolation at the granularity of processes or even compartments that are a subset of a process. Techniques are needed that can consolidate the OS into smaller components wherein the principal of least privilege can be applied, similar to the analysis required to partition ZKOSs into minimally privileged components.

Support From Hardware

Existing work on hardware defenses for software security has two major thrusts: 1) ISA extensions by commodity hardware companies and 2) tagged architectures by the research community. ISA extensions take the form of dedicated instructions that can enforce a single security property at a low cost, and they are available for use now. In contrast, tagged architectures remain the focus of DARPA programs and other academic work, though they are beginning to transition to practice.[6,7,12] Tagged architectures are more expensive than ISA extensions, coming with potentially high memory overhead for the tags and, in some cases, a dedicated security coprocessor. The higher costs of tagged architectures enable them to support effectively any security policy, making them a general-purpose solution. Additionally, tags remove the need for virtual memory/MMU as an isolation mechanism and privilege-mode swaps, replacing these coarse mechanisms with finer-grained tags; see the "Least Privileges" section. Consequently, the hidden cost of existing hardware security mechanisms, up to 30% performance overhead,[13] is removed. We believe that the generality of tagged architectures, and the fundamental redesign of secure systems that they enable, make them better suited to our vision of secure-by-design systems than one-off ISA extensions are.

Our vision is for new, secure software stack enabled by a tagged architecture. Consequently, we seek to push hardware-enabled software security to the next step by asking how hardware and software can be codesigned for security, i.e., what checks belong in software and which belong in hardware and how the two can coordinate to enforce security at the least possible cost.

Hardware Support for Language-Based Security

By designing hardware and software security policies together, we can use their strengths to offset their corresponding weaknesses and create a memory and type safe machine with strong isolation primitives. Static guarantees through type systems are the greatest strength of language-based security policies, while the runtime overhead of dynamic enforcement, e.g., garbage collection, along with static requirements that are overly strict and necessitate unsafe code, are their greatest weakness. Correspondingly, the greatest weakness of tagged architectures is that the number/size of tags (and corresponding memory/processing overhead) is prohibitive for certain policies, while their greatest strength is that the use of dynamic information in computing policy results in lower performance (runtime) overhead. Consequently, the goal of hardware/software codesign is to use static guarantees from, e.g., language-type systems to shrink the policy that must be enforced by the hardware as much as possible while using the dynamic nature of the hardware to prevent the need for unsafe code regions and removing the runtime cost of security policies.

Solving language design challenges with tags. Consider unsafe code sections within otherwise safe languages. As previously discussed, these sections exist to provide an escape hatch from the constraints of the language, enabling correct constructs that would otherwise be impossible to write. As a result, the language now provides no guarantees about this unsafe code. Worse, this unsafe code can impact the safety of the entire system. With tags, the static-type systems in the language can be relaxed to apply tags in situations where static guarantees are impossible. The tagged architecture's runtime policy can then be leveraged to extend guarantees to these corner cases by verifying behavior at runtime.

Another situation where hardware/software codesign is helpful is inline assembly and other low-level operations. Since such code is outside the semantics of safe languages, it provides no guarantees about its behavior or how it alters the memory state. With tags, it is possible to guarantee that the inline assembly preserves the memory and type safety of all existing objects through the policy that the architecture enforces as well as to provide some coarse guarantees about memory and type safety for objects it creates or full guarantees with programmer annotations.

Solving language implementation challenges with tags. Language implementation challenges, such as performance overhead from bounds checks in Rust/garbage collection in Go or the correctness of the JVM/language interpreter, are replaced by the challenge of ensuring that the correct tags are generated by the compiler and that the policy enforced by the architecture across those tags is correct. The power of tagged architectures is that they have effectively zero performance overhead (their downsides in terms of memory overhead and so forth have already been discussed). Consequently, any required runtime check is effectively free; the difficulty is in ensuring that the correct set of checks is performed. Making such guarantees, particularly for the relatively small policies, is best done by formal verification. Verifying the compiler tool chain, which generates the tags, is a significantly more difficult task, though not completely intractable, as shown by the National Institute for Research in Computer Science and Automation's CompCert. Ultimately, once a language and architecture for security have been standardized, this effort should be undertaken.

Solving tag size with languages. Just as tagged architectures can address design and implementation challenges for

safe programming languages, safe languages can address tagged architecture design challenges, specifically around the size of tags and complexity of policies. For instance, using coloring to track temporal (lifetime) memory safety for objects creates an explosion in the number of required tags and thus their memory overhead. By using, e.g., Rust's lifetime checker, almost all such tags can be eliminated without resorting to, e.g., garbage-collecting tags. Tags are only required for situations, such as circular references in data structures, where safety cannot be proven statically. Similarly, albeit less dramatically, the effects hold for spatial safety checks and type safety. By requiring tags only for edge cases that are tricky for static guarantees to support, the work required by the tagged architecture can be drastically reduced and along with it, the resources required by the tagging system.

Hardware Support for I/O Validation

For the intrasystem I/O case, where two separately compiled processes are communicating on the same machine, tagged architectures can effectively preserve the required security information, i.e., object bounds and type. By persisting tags on memory that is passed across user-defined barriers within the system, e.g., processes and the OS/user-space boundary, full type/bound/lifetime information can be preserved throughout the system. Indeed, this is another case where hardware/software codesign can eliminate information loss across boundaries in the code, a la the solution for unsafe code regions. Our vision is to have all components of the system, across hardware and software layers, cooperate to preserve critical security information throughout the system, without losing it across barriers in the system architecture. Indeed, as we discuss next, many of the existing barriers are not needed under our proposed system design.

Hardware Support for Isolation in OSs

In codesigning hardware and software, we should rethink many of the hardware defenses we use today. These defenses, such as memory isolation and hardware privilege switching, incur measurable, though often forgotten, overhead. The Singularity Project[13] at Microsoft Research presented a redesigned OS with significant security guarantees from formal models. Singularity does not require memory-based isolation or privilege levels, as these are inherently provided by its design. The authors identify 18% performance overhead due to memory-based isolation between processes (i.e., virtual memory), which jumps to 33% when utilizing privilege levels to isolate user and kernel code (i.e., system calls). Such overheads would never be accepted for a runtime software security solution and can be eliminated by rethinking the way software and hardware work together, including device drivers and the OS.

Hardware Limitations

Hardware support for security policies is a powerful tool but is not a panacea. Proposals for new hardware mechanisms show the tradeoff between the immediate deployability (low memory, power, and silicon overhead) of ISA extensions and generality of tagged architectures. In the end, however, generality is paramount. Security is an evolving arms race between attackers and defenders; fixed defenses that cannot adopt to future threat environments are undesirable. The generality of tagged architectures comes at a cost, though. Memory is required to store the tags and policies, and computational resources are required to propagate the tags and evaluate policies.

A less obvious limitation of hardware is that it cannot generate the tags/policies by itself; these must be given as input. Consequently, the compiler tool chain must be modified to create the tags and a mechanism created to specify the policy for the hardware, e.g., Dover's policy language. Dedicated hardware can process security policies more efficiently than general-purpose CPUs but still needs the appropriate programming and inputs. Guaranteeing that the correct tags are generated and verifying the correctness of policies is thus required for tags to provide the expected guarantees.

Numerous classes of vulnerabilities in modern computer systems are a direct result of decades-old design decisions. By leveraging advances in the fields of safe programming languages, security-aware hardware, and OSs, we envision a new security-centric computer design that can prevent many classes of vulnerabilities by design. Further, our approach, with its emphasis on maintaining and sharing information across both software and hardware, is capable of evolving to mitigate new attacks launched by sophisticated adversaries. We hope that the ideas and directions laid out in this article spark future research toward the ultimate goal of a world with fully secure computer systems. ■

Acknowledgments

We sincerely thank David Martinez, whose vision and advocacy were a major force behind this project, and Bob Bond, whose continued support and guidance made this work possible. This material is based on work supported by the Under Secretary of Defense for Research and Engineering under U.S. Air Force contract 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.

References

1. H. Shrobe, A. DeHon, and T. Knight, "Trust-management, intrusion-tolerance, accountability, and reconstitution architecture (TIARA)," Massachusetts Inst. of Tech., Cambridge, Tech. Rep. MIT-CSAIL-TR-2007-028, May 30, 2007. [Online]. Available: https://core.ac.uk/download/pdf/4402404.pdf
2. L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proc. 2013 IEEE Symp. Security and Privacy*, pp. 48–62. doi: 10.1109/SP.2013.13.
3. M. Miller, "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape," presented at BlueHat IL, Feb. 7, 2019. [Online]. Available: https://www.youtube.com/watch?v=PjbGojjnBZQ
4. S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina, "Exploit programming: From buffer overflows to weird machines and theory of computation," *; login, The USENIX Mag.,* vol. 36, no. 6, pp. 13–21, 2011. [Online]. Available: https://www.usenix.org/system/files/login/articles/105516-Bratus.pdf
5. E. Witchel, J. Cates, and, and K. Asanović, "Mondrian memory protection," in *Proc. 10th Int. Conf. Architectural support programming languages and operating systems (ASPLOS)*, 2002, pp. 304–316. doi: 10.1145/605397.605429.
6. R. N. Watson et al., "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *Proc. 2015 IEEE Symp. Security and Privacy*, pp. 20–37. doi: 10.1109/SP.2015.9.
7. G. T. Sullivan, A. DeHon, S. Milburn, E. Boling, M. Ciaffi, J. Rosenberg, and A. Sutherland, "The dover inherently secure processor," in *Proc. 2017 IEEE Int. Symp. Technologies Homeland Security (HST)*, pp. 1–5. doi: 10.1109/THS.2017.7943502.
8. R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "Rustbelt: Securing the foundations of the rust programming language," *Proc. ACM Program. Lang.,* vol. 2, no. POPL, 2017, Art. no. 66. doi: 10.1145/3158154.
9. A. Levy, B. Campbell, B. Ghena, P. Pannuto, P. Dutta, and P. Levis, "The case for writing a kernel in rust," in *Proc. 8th Asia-Pacific Workshop on Systems (APSys '17)*, 2017, pp. 1:1–1:7. doi: 10.1145/3124680.3124717.
10. T. C.-H. Hsu, K. Hoffman, P. Eugster, and M. Payer, "Enforcing least privilege memory views for multithreaded applications," in *Proc. 2016 ACM SIGSAC Conf. Computer and Communications Security*, pp. 393–405. doi: 10.1145/2976749.2978327.
11. A. A. Clements, N. S. Almakhdhub, S. Bagchi, and M. Payer, "Aces: Automatic compartments for embedded systems," in *Proc. 27th USENIX Conf. Security Symp., (SEC'18)*, 2018, pp. 65–82. doi: 10.5555/3277203.3277210.
12. "Arm to deliver CHERI-based prototype to tackle security threats," *EE Times*, Cambridge, MA. Accessed on: Apr. 2020. [Online]. Available: https://www.eetimes.com/arm-to-deliver-cheri-based-prototype-to-tackle-security-threats/
13. G. C. Hunt and J. R. Larus, "Singularity: Rethinking the software stack," *ACM SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, pp. 37–49, 2007. doi: 10.1145/1243418.1243424.

Hamed Okhravi is a senior staff member of the Secure Resilient Systems and Technology Group at the Massachusetts Institute of Technology (MIT) Lincoln Laboratory, Lexington, where he leads programs and conducts research in the area of systems security. His research interests include cybersecurity, the science of security, and security evaluation. Okhravi received a Ph.D. in electrical and computer engineering from the University of Illinois at Urbana–Champaign in 2010. He received the 2019 MIT Lincoln Laboratory's Best Invention Award, 2018 R&D 100 Award, 2015 Team Award, and 2014 MIT Lincoln Laboratory Early Career Technical Achievement Award. He is a Member of the IEEE. Contact him at Hamed.Okhravill.mit.edu.

Nathan Burow is a technical staff member of the Secure Resilient Systems and Technology Group at the Massachusetts Institute of Technology Lincoln Laboratory, Lexington. His research interests include language-based security, software sanitizers, and tagged architectures to support software security. Burow received a Ph.D. in computer science from Purdue University, West Lafayette, Indiana. He is a Member of the IEEE. Contact him at Nathan.Burow@ll.mit.edu.

Richard Skowyra is a technical staff member of the Secure Resilient Systems and Technology Group at the Massachusetts Institute of Technology Lincoln Laboratory, Lexington, where he leads the effort to design cyber-resilient satellite systems. Skowyra received a Ph.D. in computer science from Boston University in 2014. He received an R&D 100 award in 2018 for the Dynamic Flow Isolation network access-control system. He is a Member of the IEEE. Contact him at Richard.Skowyra@ll.mit.edu.

Bryan C. Ward is a technical staff member of the Secure Resilient Systems and Technology Group at the Massachusetts Institute of Technology Lincoln Laboratory, Lexington. His research interests include systems security, real-time and cyberphysical systems, distributed and concurrent systems, and operating systems. Ward received a Ph.D. in real-time computing from the University of North Carolina at Chapel Hill. Contact him at Bryan.Ward@ll.mit.edu.

Samuel Jero is a technical staff member of the Secure Resilient Systems and Technology Group at the

Massachusetts Institute of Technology Lincoln Laboratory, Lexington. His research interests include network security, transport protocols, embedded systems, and automated testing. Jero received a Ph.D. in computer science from Purdue University, West Lafayette, Indiana in 2018. Contact him at samuel.jero@ll.mit.edu.

Roger Khazan is the leader of the Secure Resilient Systems and Technology Group at the Massachusetts Institute of Technology (MIT) Lincoln Laboratory, Lexington. His research interests include software and hardware systems security, secure communications, and systems resiliency engineering. Khazan received a Ph.D. in electrical engineering and computer science from MIT. He is Senior Member of the IEEE and the Association for Computing Machinery and a member of the Armed Forces Communications and Electronics Association International. Contact him at rkh@ll.mit.edu.

Howard Shrobe is a principal research scientist at the Massachusetts Institute of Technology (MIT) Computer Science and Artificial Intelligence Laboratory, Cambridge. His research interests include computer architectures for inherently secure computing as well as artificial intelligence. He received a Ph.D. from MIT in 1978. He is a Member of the IEEE and a fellow of the American Association for the Advancement of Science and the Association for the Advancement of Artificial Intelligence. Contact him at hes@csail.mit.edu.

Digital Object Identifier 10.1109/MSEC.2020.3002062