One Giant Leap for Computer Security

Hamed Okhravi, Nathan Burow, Richard Skowyra, Bryan Ward, Samuel Jero, Roger Khazan, and Howard Shrobe

***Abstract*—Today's computer systems trace their roots to an era of trusted users and highly constrained hardware. Consequently, their design's fundamentally emphasize performance and discount security. The seemingly endless war in memory between attackers and defenders, and its collateral damage to users in the form of data breaches, ransomeware, and other malware infecting systems, will continue to rage until we adopt a fundamentally new system architecture that emphasizes security as a first class citizen, and not just performance. The research community has already developed many of the advanced technologies required create such a secure and performant system architecture. Here, we present a vision for how tomorrow's technologies can co-operate across all layers of modern systems — from hardware through the operating system to user applications — in order to enable performant systems that are immune to the underlying causes of today's exploits. Put another way, we show how the small steps towards security represented by existing technologies can be combined into one giant leap for the security of computer systems. We are not so naïve as to think that this will stop all cyber attacks, however it can dramatically shift the landscape in the defender's favor. In order to jump-start the shift to secure-by-design systems, we highlight both foundational technologies for such systems that require further research, and the research challenges in composing existing security technologies to create a secure and performant system architectures.**

I. Introduction

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, and programming languages that have continued largely unchallenged in commodity systems since ~1970, a fact which is particularly ironic in light of the rapid pace of innovation in early systems such as Project MAC in 1963, Multics circa 1969, and 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; an era 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. Operating Systems (OS) remain monolithic, with no isolation or separation of privileges within the OS. Indeed, privilege separation is only possible through the hierarchical ring-oriented privilege model which results in over-privileged code in the ring-0 OS. Today's systems programming languages, C and C++, came of age in an era when compilers were rudimentary, and deliberately provide only minor abstractions over 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, over-privileged 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 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's 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, operating system, 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, and uninhibited by compatibility or optimality constraints. While some previous efforts such as DARPA's CRASH and MRC projects were pioneers in clean-slate redesign of a computer system, due to their constraints, they still heavily rely on existing system components with legacy designs (*e.g.,* UNIX-like operating systems and C/C++ languages). Here, we envision the next natural step and think beyond legacy designs — envisioning instead a moonshot that

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.

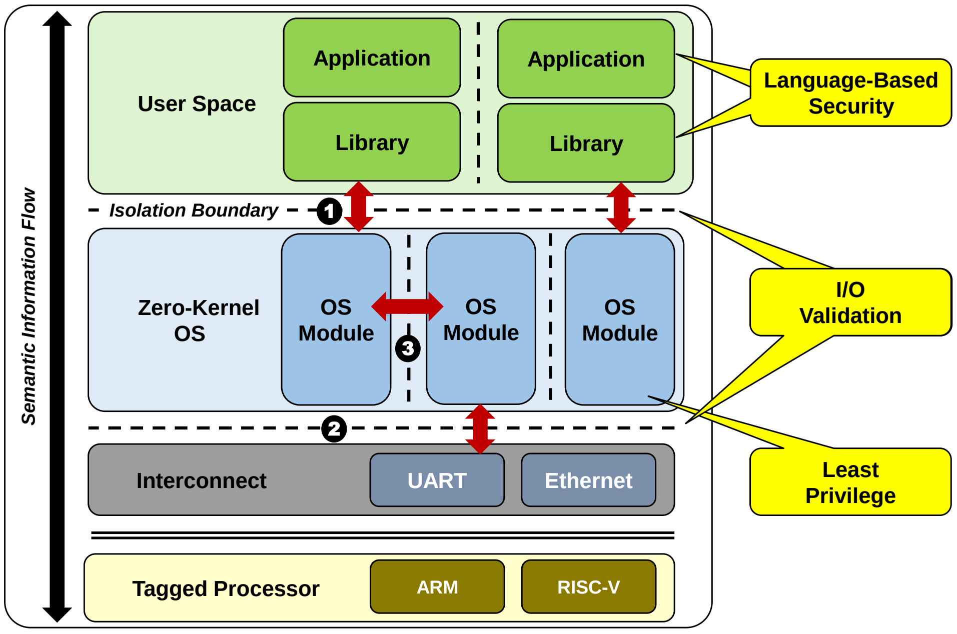

Fig. 1: Proposed secure-by-design architecture.

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 which cut across all layers of the system design: (i) safety within code modules, i.e., language based safety; (ii) safety between code modules, i.e., I/O Validation; and (iii) 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 Figure 1 Label 1, or hardware to the OS as at Label 2. Least privilege is achieved by isolating both applications from each other, and 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 aware of more than raw bits.

- **Language Based Security**. The fundamental properties of language based security are memory/type safety, which at a high level require that memory only be used 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 too prominent examples that are in widespread use. section III 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 networking. 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 like Heartbleed, which exploited a failure of the system to accurately track bounds for a buffer passed over the network. Further, as discussed in section IV, semantically aware hardware can solve I/O validation for all communication within the system, most notably IPC.
- **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 allows us to enforce that each component has access to the minimum possible set of resources for its function, i.e., Least Privileges. Current systems only provide isolation at the process level through virtual memory. Further, commodity OSs and even research micro-kernels 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. section V presents our vision, which replaces this model with a single flat memory space where isolation is provided at arbitrary

granularities by hardware, building off our existing work on “zero-kernel” operating systems.

Before discussing each component of our vision in turn, we first introduce foundational concepts we build upon throughout our vision in section II. Once we have presented our vision, we discuss how it is uniquely enabled by recent advances in semantically aware hardware, namely tagged architectures, in section VI. In particular, we show how tagged architectures can address the challenges we identify in language based security, I/O validation, and least privileges, and how existing techniques in these areas make the application of tags feasible.

II. 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 section VI

A. 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 allows any mapped area of memory to be corrupted by a bug. Microsoft recently disclosed that around 70% of attacks against their software are rooted in such vulnerabilities [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 allow an attacker to manipulate the program’s state, building and executing so called “weird machines” that, for instance, give a remote attacker control over the compromised system (*e.g.,* by spawning a command shell).

a) Memory Safety: Memory Safety violations arise when a pointer is misused to access non-programmer-intended object. For instance, a write overflows an array into an adjacent object, or a pointer to a free’d object is used to write to memory that has been reallocated to a new object. Such buffer overflows and use-after-free (UaF) vulnerabilities violate the two tenets of memory safety: (i) Spatial or Bounds safety, which requires that all pointer dereferences are in bounds of the referenced object, and (ii) Temporal or Lifetime safety, which requires that the referenced object be allocated. Formal models of memory safety assign pointers `capabilities` to access memory, which 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 which 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 subsection III-A.

b) 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 vs 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 either 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: (i) time of enforcement, and (ii) 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 in order 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 are either at compile time, or at runtime. Rust and Go are more efficient than the Runtime Time Information (RTTI) 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.

B. Tagged Architectures

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 memory, which enables the enforcement of arbitrary policies over 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 which grant ability to perform operations. The notion of tags/capabilities has seen significant academic (CHERI) as well as government (DARPA CRASH/SSITH) interest.

CHERI (Capability Hardware Enhanced RISC Instructions) [4] is a tagged architecture capability system which extends a 64-bit MIPS 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 256-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 their 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 [5] 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 decision to separate it into two cores. The first core is the primary application CPU that operates on the data itself. The second is the Policy EXecution (PEX) core that computes policy over that metadata. This decision allows 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 section VI after fully laying out our vision.

III. 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) checks guarantee that a program will halt upon encountering that class of bugs. Of course, neither kind of check is free. Static guarantees from type checking 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: (i) Language Design and (ii) 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 provides an escape from the languages 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]).

A. 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 allows 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 allow 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 can do, rather than a binary safe/unsafe tradeoff where a single line of unsafe code could corrupt the entire otherwise-safe process.

a) Complex Datastructures: A common example of the need to violate language abstractions is when implementing complex (*e.g.,* graph-like) datastructures in a language like 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, or a single read/write reference. This does not allow data structures like 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 APIs to unsafe code. The intent is to encapsulate necessarily-unsafe code with a safe interface, such that external code can only use the unsafe code in the way intended by its developer. Rust uses this approach to provide an automatically reference-counted pointer (called an `Rc`), for example. Unsafe code is needed to manage (among other things) raw memory for object wrapping and unwrapping. Its API only allows 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 their own. That said, this approach comes with its own challenges. First, the API must indeed be 'safe'. This means that safety guarantees which hold over a program written only in the core Rust language still hold over 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 over 20 person-years to fully verify. The Rustbelt [6] 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 over 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 above. 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.

b) Separate Compilation: 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: inter-process 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 section IV, including how hardware can be leverage to preserve semantic information across these boundaries.

c) Low-Level Operations: Operating systems and many embedded applications cannot operate over an abstraction of a computer, because they are responsible for actually interfacing 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 10s 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 may be able to be unmapped or rendered inaccessible during execution of that code fragment.

B. Language Implementation Challenges

Implementing the runtime required for a safe language presents two challenges: (i) insuring that runtime itself is not buggy / vulnerable, and (ii) that the language meets the performance demands of "real-world" users.

a) 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 run-time enforcement of both spatial safety (*e.g.,* array bounds checking) and 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.

b) Performance: An oft-cited reason for why legacy languages persist to this day despite their known security shortcomings is that they allow developers to eek out the highest possible performance with the smallest possible footprint. This is especially relevant in the embedded and realtime domains, where platforms are highly resource-constrained and may need to process inputs in a time-deterministic manner to meet realtime constraints. The performance argument is valid for languages which rely on heavyweight runtime enforcement of checks to provide memory or thread-safety. Garbage collectors, for example, are leveraged by Java (and JVM languages in general), Go, and most interpreted languages. They provide automatic enforcement of temporal memory safety by tracking object references and deallocating objects which 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 over time. This causes sudden slowdowns when the batch is processed, as well as sudden reductions a 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 which allows static analysis of object references. The compiler can thus identify when an object should be deallocated, whether there exist any bugs which could cause invalid pointer dereferences at runtime, and if so emit an error. Barring use of the `unsafe` keyword, any Rust program which 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 [7], for example, is an embedded embedded operating system written entirely in Rust and designed to run on resource-constrained IoT devices.

IV. Input/Output Validation

Regardless of how expressive and powerful a language's type system is, any non-trivial 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 is 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 is 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's populated based on input data (*e.g.,* in a loader preparing a program on disk for execution). If the data is encoded differently than what the program expects (*e.g.,* in little- vs. 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 operating system using its system call interface, or against other processes via inter-process communication. This problem will exist as long as incoming data is 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 pre-existing information to automate input validation. Even for external, *e.g.,* network or peripheral device inputs, the compiler has access to the intended type of input data, for example. 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 is parsed successfully (Fig. 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 intra-system communication (including system calls and inter-process 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.

V. Least Privileges

Least privileges are a well known security principle, which 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 operating system. Adopting memory safety, see section III, moves applications closer to the least privilege ideal as access now requires a programmer created reference. Operating systems, however, pose additional challenges as they have direct access to, and control over, sensitive machine state. To the best of our knowledge, developing a performant and secure application of least privileges to operating systems by compartmentalizing them is an open research challenge.

Operating systems 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 also the natural repository for "privileged" operations that change 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 micro-kernels that attempt to split traditional OS services into unprivileged, user level processes have over-privileged 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.

Operating systems frequently reach down into the gory details of the hardware architecture, directly manipulating registers, using memory mapped I/O, handling context switching and interrupts, etc. 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.

A. Operating System 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 Zero Kernels [1] that respects the least privilege principle. Zero Kernels provide isolation of code and data regions, thereby enabling compartmentalization and least privileges. Building on this foundation, Zero Kernels eliminate the kernel/userspace divide, and all processes operate in a single, flattened address space. Doing so is possible because tagged architectures allow isolation/privilege control 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.

As the security of Zero Kernels 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 divided 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 (Fig. 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 in order 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 the Zero Kernels as we envision them fundamentally rely on tagged architectures. Consequently, they can be implemented in tomorrow's more secure languages that have been designed with hardware support in mind, see subsection VI-A. 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.

B. Operating System 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 operating system 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 [8], [9]. The shim is solely responsible for managing memory permissions can be used to isolate different parts of the system. Notably, this approach allows the enforcement of least privileges even on impoverished, *i.e.,* embedded, systems that lack 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 operating system into smaller components wherein the principal of least privilege can be applied, similar to the analysis required to partition Zero Kernels into minimally privileged components.

VI. Support from Hardware

Existing work on hardware defenses for software security has two major thrusts: (i) ISA extensions by commodity hardware companies, and (ii) tagged architectures by the research community. ISA extensions take the form of dedicated hardware that can enforce a *single* security property, but have low costs and 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 [4], [5]. Tagged architectures are more expensive than ISA extensions, coming with potentially high memory overhead for the tags, and in some cases a dedicated security co-processor. The higher costs of tagged architectures allow 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, or privilege mode swaps, replacing these coarse mechanisms with finer grained tags, see section V. Consequently, the hidden cost of existing hardware security mechanisms, up to 30% performance overhead [10], 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.

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 co-designed 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.

A. 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, and runtime overhead of dynamic enforcement, *e.g.,* garbage collection, along with static requirements that are over-strict necessitating *unsafe* code 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 the use of dynamic information in computing policy results at 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 remove the runtime cost of security policies.

a) 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 where hardware/software codesign is helpful is inline assembly and other low level operations. As such code is outside of the semantics of safe languages, they can provide no guarantees about its behavior, or how it alters 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, and provide some coarse guarantees about memory and type safety for objects it creates – or full guarantees with programmer annotations.

b) 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 insuring that the correct tags are generated by the compiler, and the policy enforced by the architecture over those tags is correct. The power of tagged architectures is that they have effectively zero performance overhead (their downsides in terms of memory overhead *etc.* 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 INRIA's CompCert. Ultimately, once a language and architecture for security have been standardized this effort should be undertaken.

c) 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 life time 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. Similar, albeit less dramatic, 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.

B. Hardware Support for IO Validation

For the intra-system IO 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/userspace boundary, full type/bound/lifetime information can be preserved throughout the system. Indeed, this is another case where hardware/software co-design 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.

C. Hardware Support for Isolation in Operating Systems

In co-designing hardware and software, we should rethink many of the hardware defenses we use today. These defenses, like memory isolation and hardware privilege switching, incur measurable, though often forgotten, overhead. The Singularity Project [10] 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.,* syscalls). 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.

D. Hardware Limitations

Hardware support for security policies is a powerful tool, but is not a panacea. Proposals for new hardware mechanisms show the trade-off between the immediate deployability (low memory, power, 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.

VII. Conclusion

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 operating systems, 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 towards the ultimate goal of a world with fully secure computer systems.

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., 2009.

[2] 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.

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

[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] 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.

[7] A. Levy, B. Campbell, B. Ghena, P. Pannuto, P. Dutta, and P. Levis, "The case for writing a kernel in rust," in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, ser. APSys '17, 2017, pp. 1:1–1:7.

[8] T. C.-H. Hsu, K. Hoffman, P. Eugster, and M. Payer, "Enforcing least privilege memory views for multithreaded applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 393–405.

[9] A. A. Clements, N. S. Almakhdhub, S. Bagchi, and M. Payer, "Aces: Automatic compartments for embedded systems," in *Proceedings of the 27th USENIX Conference on Security Symposium*, ser. SEC'18, 2018, pp. 65–82.

[10] G. C. Hunt and J. R. Larus, "Singularity: rethinking the software stack," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 2, pp. 37–49, 2007.