Research Statement

Evan Johnson

Promises are cheap. Software vendors routinely describe their offerings as "secure", but few are based on designs that can guarantee even the most basic security properties. Indeed, even guaranteeing memory safety can be inherently challenging as code written in C and C++ requires developers to manually enforce safety, and if *one* developer misses *one* safety check, the system's guarantees vanish. Languages like Rust and WebAssembly (Wasm) promise to address these shortcomings by design, at the language level. By building a system using these languages, you completely eliminate whole classes of vulnerabilities like memory unsafety by construction. This is a fundamental advance in how we build systems, and everyone from Google [24] and Cloudflare [27] to the White House [21] extoll the promises of Rust and Wasm. But these promises too can fall short: any vulnerability in the execution stack—compiler, runtime, and OS—can undermine language-level security [2, 3, 5, 6].

My research bridges this gap to deliver on the promise of language-level security by combining principled system design and practical formal methods. I do this by building systems that prove key (vulnerable) components of the execution stack are bug-free, removing the need to trust them. For instance, three systems I built, VeriWasm [9], WaVe [8], and VTock [26] remove the need to trust the compiler, language runtime, and OS respectively. What these three systems have in common is that they all adhere to two key design concepts that are central to my system-building philosophy. First, systems should be *principled*: they should not rely on developers to correctly implement ad-hoc safety checks—they should instead be built on well-defined security properties that can be automatically checked and enforced. Second, they should be *practical*: to see adoption, systems need to be built them with real production constraints in mind. My Wasm binary verifier VeriWasm [9], for example, was deployed on Fastly's production serverless platform exactly because I designed the system with deployment constraints from the start—that the verifier be (1) fully automatic to avoid manual proof burden, (2) fast enough to run in continuous integration pipelines, and (3) precise enough to handle the output of existing production compilers.

My research combines practical system design and principled formal techniques to prevent vulnerabilities in real systems. As such, I work at the intersection of security [7–9, 18], systems [1, 19, 20], and programming languages [11, 13] and actively collaborate with researchers from all three fields. This practical and interdisciplinary approach has made it unusually amenable to industry adoption: my tools have been deployed in production to millions of users at companies like Fastly and Mozilla, adopted by open source communities [4], and have found bugs in critical systems like commercial airplane firmware [7].

Preserving Language-level Security in Real Systems

Languages like Rust and Wasm promise to prevent whole classes of vulnerabilities like memory corruption and have the potential to change how we build systems. However, to make this a reality, they need to provide strong guarantees—both safety and performance—for system designers to fully embrace them. In this section, I describe my efforts to work with real system designers to deploy Rust and Wasm code with formally-proven safety and production-ready performance.

VeriWasm: Preserving language-level safety at the binary level. Wasm compilers enforce memory isolation by inserting safety checks at memory reads, memory writes, and control flow transfers to prevent sandboxed code from interfering with the rest of the application. For performance reasons, the compilers do this before optimizing the code, allowing checks to be moved or (when safe) removed. In practice, this can introduce subtle bugs in the compiled code and has caused high severity CVEs in industrial toolchains [2, 6].

In response to this problem, I developed VeriWasm, a binary verifier for WebAssembly code. VeriWasm's design is driven by the needs of system designers using Wasm in production. For example, system designers turn on Wasm compiler optimizations because they need their code to be fast and therefore can't afford the overhead of using a verified compiler, so VeriWasm statically checks compiled code instead. Similarly, the natural way to use a verifier like VeriWasm is to integrate it into a continuous integration (CI) pipeline, so VeriWasm is designed to only use lightweight analysis that can complete in seconds so as not to slow down CI. Addressing these two design constraints led me to a key insight: by exploiting the structure of the Wasm language, VeriWasm can turn a hard-to-check global security property—software isolation—into multiple easy-to-automatically-check local security properties. This approach made it fully automatable and highly performant, making it well-suited for use in production systems.

In collaboration with Fastly, we deployed VeriWasm on their production cloud platform to verify that client code is correctly sandboxed and cannot access memory that belongs to another client or the platform itself. This integration led to official adoption of VeriWasm by the Bytecode Alliance, and it was upstreamed into the Lucet sandboxing compiler [4]. Finally, I worked with ARM to integrate VeriWasm into Veracruz—ARM's trusted execution platform—to enable verifiably safe multitenancy on TEEs like TrustZone and AWS Nitro [1].

WaVe: Validating isolation guarantees across the userspace-kernel boundary. While VeriWasm guarantees that sandboxed code cannot read or write outside its designated memory region, this, in itself, does not provide end-to-end isolation. This is because real code sometimes needs to access resources outside the sandbox, e.g., using the system call interface to read or write to files. Mediating access to these sorts of resources outside the sandbox is the responsibility of the sandboxing runtime. Doing this correctly requires implementing an interface from informally-specified POSIX calls to different OS platforms—each with slightly different, similarly informal specifications [23]. In practice this is tricky to get right and unsurprisingly, sandboxing runtimes are notoriously buggy [3, 5].

To address this problem, I developed WaVe [8], a verifiably secure sandboxing runtime. WaVe uses automated verification to ensure that the runtime code preserves Wasm's memory isolation guarantees and correctly restricts each sandbox's access to OS resources like the filesystem and network. What makes this work is the unique design of WaVe's specifications. WaVe only models system call's userspace-observable effects. For example, WaVe's model of the POSIX `read(fd,buf,count)` syscall does not model kernel data structures (e.g., file descriptor tables, inodes, or buffer caches); instead it only models what userspace sees—`read` may write `count` bytes starting at `buf` from file (descriptor) `fd`. This makes it possible to "pay as you go"—you don't need to specify details about file descriptors if you only care about memory isolation. It also allows security policies to be explicit and decoupled from enforcement. This not only makes it clear (and easy to audit) which policies WaVe enforces but also ensures that WaVe enforces a uniform policy across all WASI hostcalls. This collaboration with Fastly helped them identify dangerous ambiguities in the WASI specification and simplify the safety model, leading to the removal of a redundant security mechanism, WASI capabilities.

Verified Tock: Retrofitting automatic verification to a production OS. VeriWasm and WaVe show how system designers can remove the userspace execution stack from the trusted computing base, but developers are also using memory-safe languages like Rust in the kernel [12, 25]. While this is an easy way to provide memory safety for much of the kernel, in some ways it makes preserving language-level safety guarantees even more tricky. This is because low-level OS functionality like interrupt handling and context switching simply cannot be written as memory-safe code, and therefore must be must be written as unsafe code (e.g., inline assembly or unsafe Rust) that then interfaces with safe code. In this unsafe code, the protections that were previously provided by the compiler must be manually enforced by the developer. To address this problem, I developed new techniques to formally reason about unsafe low-level code in Rust systems.

This led to my collaborators and I retrofitting lightweight verification to Tock [12]—a production OS written in Rust and used by companies like Microsoft [30] and OxidOS [22]. One technique we developed is to lift low-level operations like interrupt handling into Rust emulations of these low-level operations, so that the verifier can reason about their control and data-flow. This lets us do two things: prove safety properties about these low-level operations (e.g., that interrupts don't clobber general-purpose registers), and reason about how these concurrent operations affect the rest of the kernel's execution. For instance, if an interrupt modifies a register, the verifier ensures that the kernel treats the register as volatile (and therefore does not rely on a stored value in that register), since it will be changing concurrently with kernel execution. This prevents subtle bugs like time-of-check-to-time-of-use that would otherwise be impossible for the verifier to reason about. While this is still work in progress, it has already found dangerous bugs, including an easily exploitable privilege escalation bug in the ARMv6 interrupt handling code that allows userspace processes to access privileged memory and read and write other user's sensitive data.

VeriZero & Colorguard: Scaling safe code with provably correct optimizations. After building a verified Wasm stack, my collaborators and I spoke to security engineers at Mozilla about using Wasm to sandbox third-party C libraries in Firefox. What we found is that although they recognized that sandboxing can provide strong security guarantees, its performance overhead makes it an infeasible replacement for C in performance-sensitive production systems like browsers. Upon further investigation, we found that the cause of the slowdown was not, as expected, the dynamic checks that the sandboxing compiler uses to enforce isolation, but the the assembly-level context switching code. Like OS context-switching, this code saves, clears, and restores every register so that untrusted code cannot read or modify the state of the trusted code. Even in industrial compilers, this code is slow and error-prone [14–17], but was accepted because it was seen as necessary to uphold Wasm's sandboxing guarantees.

Our insight was that this code could be optimized exactly because we weren't context switching between arbitrary code: we were context switching between sandboxed code and sandboxed code has structure that we can exploit. For example, if we can guarantee that the sandboxed code never reads uninitilized variables and has forward control-flow integrity, we can elide saving/restoring callee-saved registers when we context switch. We formalized five conditions on the structure of sandboxed code that together, can guarantee that we can safely elide *all* context-switch code. Furthermore, these conditions are simple enough that we can prove that they provide the security guarantees we need, and statically check that binary code preserves them. These lightweight transitions incur no more performance overhead than a standard function call, improving performance and making it possible to deploy SFI in production. Our optimized sandbox transition design was integrated into the wasm2c sandbox compiler and used to speed up library sandboxing in Firefox, where it is still deployed today.

Continuing my work on verifiably safe optimizations for sandboxing, I collaborated with Intel to develop Colorguard, an optimization for classic software fault isolation that reuses an existing hardware mechanism, memory protection keys (MPK), for new purposes. Colorguard allows serverless platforms like Fastly to safely pack 16x as many sandboxes in the virtual address space, and scale much more efficiently. Unfortunately, in practice, it is complex to implement, as it requires a deep understanding of Intel hardware, and careful implementation that ensures this hardware is never put into an insecure state, which can lead to a full-system compromise. My solution was to verify the integration of the optimization into the broader system. This entails explicitly describing hardware behavior and the security policy for the system and proving that regardless of how the system uses the hardware management code, that the security policy never breaks. Because of this formal rigor, Colorguard was embraced by the wider community and we were able to upstream it into three industrial compilers, Wasm2c (Google), WAMR (Intel), and Wasmtime (Fastly). Furthermore, we were able to formally verify Wasmtime's integration of Colorguard into their sandbox memory allocator.

Future Work

My previous research demonstrates how combining lightweight formal methods with practical system design can effectively mitigate vulnerabilities in real-world systems. My future research will expand on this theme in four directions:

Retrofitting embedded systems with automatic verification. Expanding on my work on VTock and bug-finding in critical systems [7], I am interested in developing more lightweight and practical means of verifying critical embedded systems. While we would like these systems—e.g., medical devices like insulin pumps—to be fully verified, this is not the current reality and unlikely to come to be as it is simply too burdensome for developers. Given this practical reality, can we design systems that provide strong safety guarantees that system designers most care about (e.g., that a sane insulin dosage is injected, regardless of the current state of the system) without investing in a years-long verification process? Can we take advantage of embedded development best-practice like hardware abstraction layers to build a library of reusable formally verified components? I plan to start to answer these questions by building tools that can provide incremental, "pay-for-what-you-use" safety guarantees using a combination of lightweight formal methods and compiler-based methods like software sandboxing. By combining these techniques these tools will allow developers to retrofit security guarantees to existing code bases without all the boilerplate that comes with standard verification projects.

Clean-slate OS design with lightweight isolation. Wasm has made lightweight isolation a practical reality, but its potential is still largely unexplored. I am particularly interested in building an OS that replaces (or complements) existing hardware-based memory isolation with lightweight software-based isolation. This design choice would have several interesting consequences. Due to the portability of Wasm, it would be trivial to run the exact same OS on a laptop, microcontroller, or any number of trusted execution environments like ARM TrustZone, SGX, or AWS Nitro. Using Wasm would also have very different performance characteristics than using hardware-based isolation: Wasm's extra runtime checks would slow down CPU-bound workloads but its zero-cost context switches [11] would make IO-bound and short-lived processes very efficient. But perhaps the most interesting advantage of using a Wasm-based OS design is that we can take advantage of all the tools and techniques to provide formal safety guarantees for Wasm code [8, 9, 11], and apply them totally out-of-the-box with no additional verification effort.

Building reliable scientific simulations. While my main research focus is to build systems that provide strong security guarantees, I am also interested in collaborating with colleagues to build principled systems in other domains. I am particularly interested in building high-performance scientific simulations with correctness guarantees and have already begun working with computational chemists and physicists towards this goal. I'm currently working with scientists from UCSD and Lawrence Livermore National Lab to redesign their material discovery framework using type-level programming—e.g., unit-of-measure types [10]—to provide simple safety guarantees. During this work, I have discovered two unfortunate facts about high-performance scientific code: 1) it regularly contains memory corruption, undefined behavior, and easily-avoidable configuration errors that meaningfully change the output of simulations, and 2) simulation validity depends on implementations upholding complex domain-specific properties like detailed balance [29] that may not be preserved in real code. This is concerning as scientific simulations are used to influence government policy and to prescribe personalized medicine. How can we build systems that don't rely on scientists, who may or may not be experienced developers, to uphold properties in these codebases? How can we formally reason about simulation validity invariants like detailed balance [29] in real codebases?

Principled safety for modern foreign function interfaces. Transitioning existing code bases from memory-unsafe to memory-safe languages (like Wasm and Rust) has garnered significant support in industry [24] and government [21, 28] alike. While transitioning these codebases is a worthy and valuable goal, it also has the potential to introduce subtle bugs into systems that undergo this process. This is because this transitioning a codebase is necessarily incremental, so the Rust code must interact with C/C++ components via the foreign-function interface (FFI), where data is reduced to bytes and pointers, reinforcing the very problems that Rust was designed to solve. Can we replace the current status quo of developers writing ad hoc glue code with more principled methods that preserve the strong safety that Rust's type system and borrow checker offer? Can we design type systems (and generate glue code) for foreign function interfaces that provide strong safety without burdensome static or dynamic analysis? What tradeoffs (e.g., performance vs. safety vs. simplicity) can be made to ensure these methods are applicable to a variety of production systems?

Ultimately, I will continue to design principled and practical solutions to software security's thorniest problems and ensure the software of the future keeps its promises.

References

[1] M. Brossard, G. Bryant, B. E. Gaabouri, X. Fan, A. Ferreira, E. G. Evans, C. Haster, E. Johnson, D. Miller, F. Mo, D. P. Mulligan, N. Spinale, E. van Hensbergen, H. J. M. Vincent, and S. Xiong. Private delegated computations using strong isolation. *IEEE Transactions on Emerging Topics in Computing*, 12(1):386–398, 2024.

[2] A. Crichton. Guest-controlled out-of-bounds read/write on x86_64. `https://github.com/bytecodealliance/wasmtime/security/advisories/GHSA-ff4p-7xrq-q5r8`, 2023.

[3] A. Crichton. Data leakage between instances in the pooling allocator. `https://github.com/bytecodealliance/wasmtime/security/advisories/GHSA-wh6w-3828-g9qf`, 2024.

[4] C. Fallin. Integrate VeriWasm. `https://github.com/bytecodealliance/lucet/pull/658`, 2021.

[5] D. Gohman. Wasmtime doesn't fully sandbox all the windows device filenames. `https://github.com/bytecodealliance/wasmtime/security/advisories/GHSA-c2f5-jxjv-2hh8`, 2024.

[6] L. Hansen. Mark the jump_table_entry instruction as loading. `https://github.com/bytecodealliance/cranelift/pull/805`, 2019.

[7] E. Johnson, M. Bland, Y. Zhu, J. Mason, S. Checkoway, S. Savage, and K. Levchenko. Jetset: Targeted firmware rehosting for embedded systems. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 321–338, 2021.

[8] E. Johnson, E. Laufer, Z. Zhao, D. Gohman, S. Narayan, S. Savage, D. Stefan, and F. Brown. Wave: a verifiably secure WebAssembly sandboxing runtime. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, May 2023.

[9] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan. Trust but verify: SFI safety for native-compiled Wasm. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society, February 2021.

[10] A. Kennedy. Types for units-of-measure: Theory and practice. In *Central European Functional Programming School*, pages 268–305. Springer, 2009.

[11] M. Kolosick, S. Narayan, C. Watt, M. LeMay, D. Garg, R. Jhala, and D. Stefan. Isolation without taxation: Near zero cost transitions for sfi. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, January 2022.

[12] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 234–251, 2017.

[13] A. E. Michael, A. Gollamudi, J. Bosamiya, E. Johnson, A. Denlinger, C. Disselkoen, C. Watt, B. Parno, M. Patrignani, M. Vassena, and D. Stefan. Mswasm: Soundly enforcing memory-safe execution of unsafe code. *Proc. ACM Program. Lang.*, 7(POPL), Jan. 2023.

[14] Issue 1607: Signal handling change allows inner sandbox escape on x86-32 linux in chrome. `https://bugs.chromium.org/p/nativeclient/issues/detail?id=1607`, 2011.

[15] Issue 1633: Inner sandbox escape on 64-bit windows via kiuserexceptiondispatcher. `https://bugs.chromium.org/p/nativeclient/issues/detail?id=1633`, 2011.

[16] Issue 2919: Security: Naclswitch() leaks naclthreadcontext pointer to x86-32 untrusted code. `https://bugs.chromium.org/p/nativeclient/issues/detail?id=2919`, 2012.

[17] Issue 775: Uninitialized sendmsg syscall arguments in sel_ldr. `https://bugs.chromium.org/p/nativeclient/issues/detail?id=775`, 2010.

[18] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, et al. Swivel: Hardening {WebAssembly} against spectre. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1433–1450, 2021.

[19] S. Narayan, T. Garfinkel, E. Johnson, D. Thien, J. Rudek, M. LeMay, A. Vahldiek-Oberwagner, D. Tullsen, and D. Stefan. Segue & colorguard: Optimizing sfi performance and scalability on modern x86. In *The 17th Workshop on Programming Languages and Analysis for Security*, 2022.

[20] S. Narayan, T. Garfinkel, M. Taram, J. Rudek, D. Moghimi, E. Johnson, C. Fallin, A. Vahldiek-Oberwagner, M. LeMay, R. Sahita, D. Tullsen, and D. Stefan. Going beyond the limits of sfi: Flexible and secure hardware-assisted in-process isolation with hfi. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, pages 266–281, New York, NY, USA, 2023. Association for Computing Machinery.

[21] Office of the National Cyber Director. Press release: Future software should be memory safe. `https://www.whitehouse.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/`, 2024.

[22] A. Radovici. OxidOS Automotive. `https://oxidos.io/`, 2024.

[23] T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell. Sibylfs: formal specification and oracle-based testing for posix and real-world file systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 38–53, 2015.

[24] J. V. Stoep and S. Hines. Rust in the Android Platform. `https://security.googleblog.com/2021/04/rust-in-android-platform.html`, 2021.

[25] The Linux Foundation. Rust for linux. `https://rust-for-linux.com/`, 2024.

[26] The Vtock Team. Vtock. `git@github.com:PLSysSec/tock.git`, 2024.

[27] K. Varda. WebAssembly on Cloudflare workers. `https://blog.cloudflare.com/webassembly-on-cloudflare-workers/`, 2018.

[28] D. Wallach. Translating All C to Rust (TRACTOR). `https://www.darpa.mil/program/translating-all-c-to-rust`, 2024.

[29] Wikipedia. Detailed balance. `https://en.wikipedia.org/wiki/Detailed_balance`, 2024.

[30] C. Windeck. Microsoft security controller Pluton is also coming to Intel Core. `https://www.heise.de/en/news/Microsoft-security-controller-Pluton-is-also-coming-to-Intel-Core-9833954.html`, 2024.