RedLeaf: Towards An Operating System for Safe and Verified Firmware

Vikram Narayanan
University of California, Irvine

Marek S. Baranowski
University of Utah

Leonid Ryzhyk
VMware Research

Zvonimir Rakamarić
University of Utah

Anton Burtsev
University of California, Irvine

Abstract

RedLeaf is a new operating system being developed from scratch to utilize formal verification for implementing provably secure firmware. RedLeaf is developed in a safe language, Rust, and relies on automated reasoning using satisfiability modulo theories (SMT) solvers for formal verification. RedLeaf builds on two premises: (1) Rust's linear type system enables practical language safety even for systems with tightest performance and resource budgets (e.g., firmware), and (2) a combination of SMT-based reasoning and pointer discipline enforced by linear types provides a unique way to automate and simplify verification effort scaling it to the size of a small OS kernel.

ACM Reference Format:
Vikram Narayanan, Marek S. Baranowski, Leonid Ryzhyk, Zvonimir Rakamarić, and Anton Burtsev. 2019. RedLeaf: Towards An Operating System for Safe and Verified Firmware. In *Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3317550.3321449

1 Introduction

Every modern off-the-shelf system (desktop and server) executes several layers of firmware underneath the most privileged layer of software be it an operating system (OS) or a hypervisor. On an Intel platform, firmware subsystems such as System Management Mode (SMM), Management Engine (ME) [28], Innovation Engine (IE) [27], Baseboard Management Controller (BMC) [43], power-management controllers (P-Unit [29] and PMC [59]), and likely many others, leverage hardware memory isolation and dedicated microcontrollers to run in isolation from the systems software. For example, anecdotal evidence suggests that at least four different firmware subsystems are running on the Intel Silvermont Moorefield system-on-chip along with regular cores [58]. Some firmware subsystems have complete control over the hardware platform. For example, SMM, IE, and ME have unrestricted access to the entire memory of the system. BMC, ME, and IE provide physical-like access to the system over the network, i.e., have the capability to power cycle the machine, boot from a network drive, install system software, intercept keyboard, mouse, and video I/O, and perform direct memory access (DMA) transactions. Moreover, some firmware subsystems [27, 28] are enabled and continue execution as long as the power is connected to the machine, i.e., even if the rest of the system is powered down.

Being the most privileged part of the system, the platform firmware becomes a primary target for security attacks. A single SMM, IE, ME, or BMC vulnerability gives an attacker full control over the machine. Furthermore, the attack is virtually undetectable with existing anti-virus and forensics tools.

To make things worse, modern firmware is fundamentally insecure. The software engineering technology behind the platform firmware has remained unchanged for decades. Modern firmware is developed in a combination of low-level assembly and an unsafe programming language, namely C. (Several rare exceptions demonstrate applications of fuzzing and symbolic execution [24, 40].) Typical firmware both adheres to multiple low-level hardware specifications and implements functionality of a minimal OS, i.e., implements multiple device drivers and sometimes even provides support for file systems and network protocols [27, 28, 43]. Due to such inherent complexity, bugs and vulnerabilities are routinely introduced in the omni-privileged firmware code. Multiple attacks on platform firmware were demonstrated in recent years [25, 39, 50, 51, 72–75], and Intel disclosed 20 firmware vulnerabilities overall in 2018 [1–20]. Likely, this is just a tip of an iceberg. We anticipate that similar to modern operating systems that face hundreds of vulnerabilities a year, modern firmware as it exists today will likely go through a cycle of aggressive vulnerability discovery and exploitation in the next several years. Furthermore, it is unlikely that existing security mechanisms will be able to ensure the security of firmware subsystems.

HotOS '19, May 13–15, 2019, Bertinoro, Italy

ACM ISBN 978-1-4503-6727-1/19/05…$15.00
https://doi.org/10.1145/3317550.3321449

We argue that ensuring the security and correctness of firmware requires a clean-slate approach. Modern attacks utilize automated bug finding tools for discovery of vulnerabilities. A promising approach to win the arms race against such attacks is to utilize a rigorous approach to program correctness.

RedLeaf is a new operating system aimed at leveraging a safe, linear-typed programming language, Rust, for developing safe and provably secure firmware. RedLeaf aims at creating an efficient execution environment that is verifiably end-to-end secure, meaning that: (1) verification covers all code that executes on the machine from the kernel and platform-specific device drivers to applications that implement firmware-specific functionality, (2) it is feasible to demonstrate equivalence to a high-level specification—implementation of the system is indistinguishable from the high-level abstract state machine, and (3) the verified code remains fast.

RedLeaf builds on two premises: (1) Rust's linear type system enables practical language safety even for systems with the tightest performance and resource budgets, e.g., firmware, (2) a combination of SMT-based reasoning and pointer discipline enforced by linear types provides a way to automate and simplify verification effort and scale it to the size of a small operating system kernel that can run firmware subsystems. RedLeaf provides a Floyd-Hoare-style [42, 47] modular verification (i.e., based on pre-conditions, post-conditions, and loop invariants) for low-level systems that are designed to be fast and small. It achieves that by developing a new verification toolchain built on the SMACK verifier [36, 69], Boogie intermediate verification language [31, 54], and Z3 SMT solver [30].

In the past, IronClad [46] achieved whole stack verification of non-trivial systems, but the systems incurred 100–189x slowdown. We target the same level of guarantees of whole-stack verification, but with the performance of our system remaining close to the unverified unmanaged code. Specifically, our verified code will compile into low-level machine code, require no runtime, and avoid garbage collection.

We argue that unique properties of Rust's linear type system, and specifically its ability to lift the burden of resolving memory aliasing from the verifier, open a new page in the domain of practical and scalable verification. While RedLeaf is an early work in progress, we believe that our methodology and approach can be useful to enable new directions in the development of the next generation of secure and reliable systems.

2 Overview of RedLeaf

RedLeaf is a minimal operating system aimed at implementing firmware subsystems. Today, a broad range of firmware requires functionality of an operating system with support for running concurrent activities (i.e., threads of execution), interrupt handling, scheduling, memory management, and even process isolation and inter-process communication. For example, recent analysis of the Intel ME [28] revealed that it runs a small, possibly, Minix-based microkernel that provides support for conventional process isolation and all traditional primitives of a microkernel operating system [71]. Trusty [32] and Little Kernel [37] are minimal kernels used for implementing firmware of a Trusted Execution Environment (TEE) on Android devices. Firmware subsystems of Baseboard Management Controller (BMC) [43], numerous network switches, and recent BIOS implementations rely on the full Linux operating system as their firmware OS [34, 41, 63, 65].

Furthermore, there is a growing trend motivated by both security concerns [32, 41, 60] and economics of open hardware and software development to open up traditionally closed firmware subsystems. EDK II is an open source implementation of the UEFI firmware from Intel [49]. LinuxBoot [34] and OpenBMC [41] are open-sourced versions of BIOS and BMC controller firmware subsystems. Started as part of the Open Compute Project [68], today multiple switch vendors support open bare metal switches, i.e., switches that allow execution of a third-party operating system like Linux on the switch itself [62]. Switch vendors develop the switch hardware, and provide libraries for programming it, but leave the software stack that controls the hardware open. Multiple Linux-based switch operating systems are available [63, 65].

RedLeaf Architecture We develop RedLeaf as a minimal operating system aimed at the needs of a diverse family of firmware subsystems. RedLeaf is designed to run as a small core kernel and a collection of language-enforced protection domains [21, 48, 66] that implement kernel subsystems and specific firmware logic. Isolated kernel subsystems do not share data in RedLeaf. Instead, they rely on the linear type system to implement lightweight zero-copy communication [21]. Whenever an object is passed from one domain to another, e.g., via a function call, the sender loses ownership of the object—the compiler enforces that it cannot reference the object in the future. This allows us to modularize the verification effort, since verifying a Rust function in general does not require reasoning or maintaining complex shape invariants about the objects on the heap not owned by it.

Rust and Linear Types RedLeaf builds on the premise that linear types are critical for creating a scalable automated verification infrastructure. In particular, Rust enforces (using its type system) a rigorous discipline for controlling of sharing and aliasing in the program heap. In a software verifier for a traditional procedural language, developers are required to provide complex program heap non-aliasing and partition invariants by writing cumbersome quantified annotations. Such annotations are complicated to come up with and write, as well as to reason about using SMT solvers. Rust's linear type system and ownership model "forces" developers to

carefully think about and essentially explicitly specify their program heap sharing/non-aliasing properties early on in the development process. Our verification toolchain leverages this information to (1) remove the burden of specifying a large class of program heap invariants from the user and (2) scale up the verification by performing an efficient quantifier- and array-free encoding of the program heap (see section 3).

In contrast to previous efforts aimed at developing scalable automated verification in a verification-friendly language [45, 46], Rust allows RedLeaf to remain fast. Historically, to implement safety, programming languages rely on managed runtime, and specifically garbage collection. Despite many advances in garbage collection, its overhead remains prohibitive for systems with tight time and space budgets. Rust, however, implements safety without garbage collection, and instead relies on a restricted ownership model enforced by its linear type system—there exists a unique reference to each live object in memory. Single ownership allows static tracking of the object lifetime and its deallocation without a garbage collector.

Related Work In the domain of operating systems, several verified kernels have been tried in the past. Unfortunately, with no or minimal automation of proofs the cost of verifying OS code is prohibitively high. It took over 20 person-years to develop the first verified microkernel, seL4 [53], which is only 10,000 lines of code. Concentrating on multicore support, CertiKOS [44] and later Xu et al.'s [76] framework for reasoning about interrupts in the μC/OS-II kernel rely on the Coq interactive theorem prover [33] to construct proofs, yet, still taking 2 and 5.5 person-years, respectively. Both seL4 and CertiKOS verify only the core kernel and leave device drivers unverified. Hyperkernel introduces push button verification, i.e., translates LLVM intermediate representation (IR) into SMT queries, to achieve significant progress in automation of OS proofs [61]. Yet, without explicit support from the programming language that simplifies the specification of loop invariants, only the simplest kernel, i.e., the kernel in which all code paths are finite, is amenable to the suggested verification approach.

3 Leveraging Rust for Verification

Software verification involves reasoning about program state, which typically consists of the program counter, valuations of local and global variables, and program heap that stores dynamically allocated objects and data structures. When running on an actual machine the heap is bounded, however, its size is so large that for the purpose of verification it is customary to assume it is unbounded. Hence, in theorem-prover-based software verifiers it is straightforward to model the program heap using an unbounded array (i.e., map) that maps addresses into values. In practice, however, having just one large array has a detrimental impact on scalability since current state-of-the-art automatic theorem provers, namely SMT solvers, struggle in the presence of numerous array reads and writes. Hence, various approaches have been devised to split the program heap into a collection of arrays, thereby having less read/write pressure on each array. The key idea is to place dynamically allocated objects, or their parts, that cannot alias into separate arrays. To conservatively establish non-aliasing, these approaches typically leverage either type information [26] or results of a conservative pointer analysis [70]. Still, despite significantly improving scalability, even such memory-splitting-based approaches run out of steam when faced with programs with a lot of aliasing and intricate multi-level linked data structures.

Our main goal is to perform full functional verification of RedLeaf. Currently, performing such deep and detailed verification of even a small-sized system can only be achieved using modular reasoning that relies on the *design-by-contract paradigm*. This entails requiring from a developer to manually annotate their programs with procedure contracts in the form of pre- and post-conditions, and loop invariants. In an imperative language designed with verification in mind, such as Dafny [55], the presence of dynamic memory allocation, pointers, side-effects, and unbounded (linked) data structures entails that the most complex and painful aspect of writing annotations is describing memory aliasing and separation constraints.

In Rust, on the other hand, either alias-freedom or immutability comes for free. More specifically, if there is a mutable reference, then there is no aliasing, but a program can have any number of immutable references to an object. We next present several examples that explore to which extent the non-aliasing and immutability rules baked into Rust help with both the specification and verification processes.

Frame Example Figure 1 (top) gives a simple example illustrating how uncontrolled side-effects make modular verification much more complicated and involved. To be able to modularly verify the assertion in the C code on the left, we need to provide the verifier with a postcondition on `foo()` of the following form:

```
ensures forall addr:int ::
  Mem[addr] == old(Mem[addr])
```

We assume that the verifier is using array `Mem`, which maps addresses into values, to model the program heap/memory. The `old` construct denotes that whatever it wraps refers to the pre-state of the procedure. Such post-conditions are called *frame rules*, and when performing modular verification they are used to *frame* what part of the global program state procedure `foo()` can update. In this simple case, the frame rule specifies that `foo()` does not modify any memory/heap locations. In terms of specifying them, frame rules get really complicated really fast, for example in the presence of even

```
void foo() {...}
void main(void) {
  int *x = (int*)malloc(sizeof(int));
  *x = 5;
  foo(x);
  assert(*x == 5);
}
```

```
fn foo(x: &i32) {...}
fn main() {
  let x: Box<i32> = Box::new(5);
  foo(&*x);
  assert!(*x == 5);
}
```

```
void bar(int *x, int *y) {
  *x = 5;
  assert(*y == 10);
}
void main(void) {
  int *x = (int*)malloc(sizeof(int));
  int *y = (int*)malloc(sizeof(int));
  *y = 10;
  bar(x, y);
}
```

```
fn bar(x: &mut i32, y: &i32) {
  *x = 5;
  assert!(*y == 10);
}
fn main() {
  let mut x = Box::new(0);
  let y = Box::new(10);
  bar(&mut *x, &*y);
}
```

Figure 1. Simple examples illustrating how uncontrolled side-effects (top) and unrestricted aliasing (bottom) in a language complicate the verification process. C implementations are on the left, while their Rust counterparts are on the right.

simple unbounded arrays and linked data structures such as lists. In terms of proving them, they present a major burden for SMT solvers since they always involve quantifiers, which SMT solvers often struggle with since the quantifier instantiation approach they implement is very brittle.

If we encode this example in Rust, as shown on the top-right in Figure 1, `x` would be passed as an immutable reference (i.e., `x:&i32` versus `x:&mut i32`), in which case the listed frame rule could be derived automatically from the function signature. In fact, if we push this further, the frame rule (and the quantifier it brings into its SMT encoding) would not actually be needed, as we know from the function signature that the function cannot modify x; in particular, it cannot modify it even via some other alias to the same memory location. Moreover, when verifying modules written in a pure subset of Rust, there is no need to model memory as an array, since the main reason to do so for imperative languages is the possibility of aliasing. Instead, we can model the program heap as a collection of typed variables.

Aliasing Example Figure 1 (bottom) gives a simple example illustrating how unrestricted aliasing make modular verification much more complicated and involved. To be able to modularly verify the assertion in the C code on the left, we need to provide the verifier with two pre-conditions on `bar()`. The first one is obvious: `requires *y == 10`. However, the second one has to ensure that pointers `x` and `y` do not alias: `requires x != y`. Similarly to specifying frame rules, non-aliasing specifications become really complex really fast, again for example in the presence of linked data structures. For instance, if `bar()` would take two linked lists as input, we would often need to specify that the lists are completely disjoint, meaning that no two list elements can alias. This typically requires the introduction of quantified *list reachability* predicates, which are very hard to reason about.

If we encode this example in Rust, as shown on the bottom-right in Figure 1, we can drop the second pre-condition as we know by construction that `x` and `y` cannot alias the same memory location. Also, Rust does not have dynamic memory allocation akin to `malloc`. Instead, we would write `x = Box::new(); y = Box::new()`, which would introduce two new variables to the program state. Their location in memory is irrelevant since they cannot alias each other or any other variables. Hence, in the SMT encoding of Rust, we could use scalar variables to capture `x` and `y`, instead of the more expensive memory maps used in the traditional encodings of imperative languages.

To summarize, it is well-accepted that pointer separation and non-aliasing is really hard to both specify and reason about, especially once one starts introducing linked data structures such as lists and trees. Separation logic helps with that to some extent, since it enables focusing only on the parts of the heap that are being touched by a particular function. Once we ensure that every variable is disjoint from every other variable (i.e., no two variables can alias), as in Rust, we do not need any more to use arrays to model the program heap/memory. In other words, the semantics of pure Rust can be defined without introducing a heap (typed or untyped). This enables us to encode many interesting

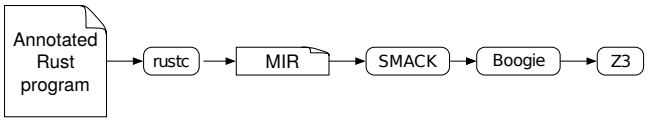

Figure 2. Toolflow of the Rust verifier.

verification problems to SMT without arrays or quantifiers, thereby greatly improving the scalability and performance of the verifier.

In general, clearly there is no way around having an unbounded program heap if we want to model real programs precisely, such as programs that allocate unbounded vectors or linked data structures. However, this program heap can be much better structured within a Rust verifier than in verifiers for conventional languages. For example, instead of using an untyped array of bytes to represent program heap, we can leverage typed lists and maps provided by SMT solvers, which are both unbounded, but are much more efficient. More specifically, the state of a Rust program is a set of variables, where each variable describes a potentially unbounded tree of typed objects. It is a tree because there is no aliasing and it is potentially unbounded due to the existence of recursive data structures.

Rust Verifier As a part of RedLeaf's verification effort, we work on creating a rigorous (i.e., sound) modular Rust verifier based on SMT solving. Figure 2 outlines the toolflow of the verifier. We leverage the *mid-level IR* (MIR) [64] format as the starting point for our translation into Boogie [22]. MIR is a simple core representation of Rust programs that compiles away many of the complexities of Rust. For example, all of the Rust control-flow-related keywords, such as loop, break, and continue, are compiled away into simple goto statements supported by MIR. Internally, MIR is represented as a control-flow graph, which also makes it suitable to be the verification target. Given that MIR contains only a very limited number of language primitives, building a translator into Boogie is a modest development effort. At the same time, MIR makes all types explicit and includes full type information, since it gets constructed after all the Rust type-checking is already done. Hence, the encoding of the program heap can be performed much more efficiently, as we already described. Finally, at the MIR level, program annotations are easier to parse and translate into Boogie compared to translating from LLVM-IR.

4 Verification of the Core Kernel

We approach verification of the RedLeaf kernel and application domains using modular assume-guarantee Floyd-Hoare-style reasoning [42, 47]. Specifically, we annotate Rust programs, in particular functions and loops, with predicates about program state in the form of pre-conditions, post-conditions, and loop invariants. Then, the verification process can modularly discharge proofs for each function in isolation, thereby allowing for the full-functional verification process to scale to operating-system-size programs. For example, the Rust function in Figure 3 is annotated with a post-condition about the program state, encoded as a function attribute. The first conjunct ensures that the vector's length does not change, the second ensures that the vector is sorted in the end of the function, and finally the third ensures that every value in the sorted vector is present in the unsorted input vector.

```
#[ensures="(x.len() == old(x).len())
 && (forall i,j in 0..x.len():
        i<=j ==> x[i] <= x[j])
 && (forall i in 0..x.len():
        exists j in 0..x.len():
           x[i] == old(x)[j])"]
fn sort(x: &mut Vec<i32>) {
  // ... Code to sort the vector ...
}
```

Figure 3. Example of a post-condition encoded as a function attribute.

If this Rust code is appropriately translated into a Boogie program [31, 54], the Boogie verifier [23] can automatically verify that the post-condition holds (for all possible inputs x) as long as the code to sort the vector is correct. Boogie is sound, i.e., it will never prove an incorrect program, but it will often fail to automatically recognize valid programs as correct. Hence, the majority of the verification effort goes into developing proper pre-conditions, post-conditions, and loop invariants inside every subsystem to help Boogie complete the verification.

Specifications We develop a simple language for writing specifications as an abstract description of the desired system behavior, e.g., we allow specifying how the state of the kernel that is modeled as a state machine advances with every system call. The verification process then ensures that the system implementation meets this high-level description. Specifications are a part of the trusted computing base, so it is important to develop them correctly.

Unsafe Rust Extensions While recent work on implementing an embedded operating system in Rust [38, 56] develops techniques for minimizing the amount of unsafe Rust code in the OS kernel, development in a pure safe code is simply impossible, e.g., all code that accesses raw memory-mapped machine state, e.g., DMA ring buffers, requires an unsafe cast [38].

The RustBelt project provides a guide for ensuring that unsafe code is encapsulated within a safe interface [52]. Specifically, by modeling the semantics of the type system, RustBelt's rules can be used to determine if an unsafe program

can have a safe encapsulation. It does so by generating a code contract specifying what conditions must be true for the library to be safely accessed. The rules given in RustBelt only cover reasoning about unsafe pointers generated from within the Rust programming language. In RedLeaf we extend the rule system to model DMA and other hardware level pointers. This allows us to produce the appropriate code contract to ensure safety. Then a verifier can check the code contract, proving the library is safe.

Memory Allocation Rust programs allocate memory on the stack and on the heap. Both allocation mechanisms require extra steps to be safe. The program is "stack safe" if the stack pointer never points outside of allocated memory specifically dedicated to the stack. In Rust the violation of this property is possible through a stack overflow, which happens when the stack does not have enough space to store the new stack frame. Currently, Rust implements a dynamic check that ensures that there is always enough space on the stack for a function to proceed. In RedLeaf we allocate the new stack from the verified dynamic memory allocator (heap allocator).

Heap allocation requires dynamic memory management. In RedLeaf we adapt one of the existing verified memory allocators. Specifically, we adapt the `memb` from Contiki [57], a popular open-source operating system for IoT. Memb's specifications were fully proven automatically using Frama-C [57]. While the existing and verified Memb allocator allows us to get the system running, in the future we plan to investigate how we can develop and verify a more sophisticated memory allocator implemented in Rust.

Threads and Context Switching Executing on bare metal, RedLeaf has to provide support for creating threads of execution and context switching between them. A minimal thread is represented by a thread structure that stores run-time state of the thread (i.e., callee saved registers) and a pointer to the thread's stack. To implement support for creating a new thread we allocate a new thread data structure and stack using the verified memory allocator.

To switch between threads we implement and verify a small assembly routine that saves the runtime state of the process, restores the state of the next process by loading its runtime state in the CPU registers, and switches the stack between the old and the new thread. The entire context switch routine is only 13 machine instructions on 32bit x86. However, the code for performing a context switch is necessarily architecture specific, and is written in assembly. In order to verify the correctness of the context switch code, we need to model the effects of the machine instructions. We rely on McSema [35], which translates native machine code into LLVM bitcode. As the Rust compiler emits LLVM bytecode, we inline the translated context switch routine directly into the emitted bytecode.

Interrupts RedLeaf disables interrupts while running inside the kernel. Interrupt handlers are inherently unsafe as they preempt execution of the code asynchronously leaving memory in a possibly inconsistent state. By disabling the interrupts, we can reason about the code path through the kernel atomically (i.e., we rule out interleaved execution, and allow the verifier to reason about each "system call" from start to finish and independent from the rest of the system). As a consequence, RedLeaf disables preemption of a thread inside the core kernel. The path through the kernel is short, hence executing it with preemption disabled is actually in many cases faster than fine-grained locking [67]. Interrupts are delivered when the kernel returns from the system call. The interrupt handler again runs atomically.

5 Conclusions

RedLeaf brings together results from the domains of verification, programming languages, and systems to enable new methodology for developing verified systems software. To achieve complete verification of a small operating system aimed at the development of firmware subsystems, we propose a set of new tools, a collection of techniques and engineering principles, and a methodology focused on scalable development of verified systems.

Acknowledgments

We thank the anonymous HotOS reviewers. This material is partially based upon work supported by Intel and the National Science Foundation under grants number 1837127 and 1837051.

References

[1] Intel Security Advisory. 2018. Bluetooth pairing vulnerability. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00128.html.

[2] Intel Security Advisory. 2018. BMC Firmware Vulnerability Intel Server Boards, Compute Modules and Systems. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00130.html.

[3] Intel Security Advisory. 2018. DCI Policy Update. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00127.html.

[4] Intel Security Advisory. 2018. EDK II Untested memory not covered by SMM page protection. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00159.html.

[5] Intel Security Advisory. 2018. Firmware Authentication Bypass. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00152.html.

[6] Intel Security Advisory. 2018. Insecure Handling of BIOS and AMT Passwords. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00160.html.

[7] Intel Security Advisory. 2018. Intel 2G Firmware Update for Modems using ETWS. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00116.html.

[8] Intel Security Advisory. 2018. Intel Active Management Technology 9.x/10.x/11.x/12.x Security Review Cumulative Update Advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00141.html.

[9] Intel Security Advisory. 2018. Intel Baseboard Management Controller (BMC) firmware Advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00149.html.

[10] Intel Security Advisory. 2018. Intel Converged Security Management Engine (Intel CSME) 11.x issue. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00118.html.

[11] Intel Security Advisory. 2018. Intel CSME Assets Advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00125.html.

[12] Intel Security Advisory. 2018. Intel NUC BIOS SW SMI Call-Out. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00110.html.

[13] Intel Security Advisory. 2018. Intel NUC Firmware Security Advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00176.html.

[14] Intel Security Advisory. 2018. Intel Platform Trust Technology (PTT) Update Advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00142.html.

[15] Intel Security Advisory. 2018. Intel Q118 Intel Active Management Technology 9.x/10.x/11.x Security Review Cumulative Update. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00112.html.

[16] Intel Security Advisory. 2018. Intel Server Board Firmware Advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00138.html.

[17] Intel Security Advisory. 2018. Intel Server Board TPM Advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00148.html.

[18] Intel Security Advisory. 2018. Intel Server Boards Firmware Advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00179.html.

[19] Intel Security Advisory. 2018. Platform firmware included insecure handling of certain UEFI variables. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00158.html.

[20] Intel Security Advisory. 2018. Power Management Controller (PMC) Security Advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00131.html.

[21] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. 2017. System Programming in Rust: Beyond Safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS)*. 156–161. https://doi.org/10.1145/3102980.3103006

[22] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects (FMCO)*. 364–387. https://doi.org/10.1007/11804192_17

[23] Michael Barnett and K. Rustan M. Leino. 2005. Weakest-Precondition of Unstructured Programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. 82–87. https://doi.org/10.1145/1108792.1108813

[24] Oleksandr Bazhaniuk, John Loucaides, Lee Rosenbaum, Mark R Tuttle, and Vincent Zimmer. 2015. Symbolic Execution for BIOS Security. In *Proceedings of the 9th USENIX Conference on Offensive Technologies (WOOT)*. 8–8.

[25] Catalin Cimpanu. 2018. New Spectre Attack Recovers Data From a CPU's Protected SMM Mode. https://www.bleepingcomputer.com/news/security/new-spectre-attack-recovers-data-from-a-cpus-protected-smm-mode/.

[26] Jeremy Condit, Brian Hackett, Shuvendu K. Lahiri, and Shaz Qadeer. 2009. Unifying Type Checking and Property Checking for Low-level Code. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 302–314. https://doi.org/10.1145/1480881.1480921

[27] Intel Corporation. 2016. Intel Forums: What is Intel Innovation Engine? https://forums.intel.com/s/question/0D50P000049060MSAQ/what-is-innovation-engine.

[28] Intel Corporation. 2017. What is Intel Management Engine? https://www.intel.com/content/www/us/en/support/articles/000008927/software/chipset-software.html.

[29] Intel Corporation. 2018. Intel Pentium Silver and Intel Celeron Processors. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/silver-celeron-datasheet-vol-1.pdf.

[30] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[31] Robert DeLine and K. Rustan M. Leino. 2005. *BoogiePL: A Typed Procedural Language for Checking Object-Oriented Programs*. Technical Report MSR-TR-2005-70. Microsoft Research.

[32] Android Developers. 2018. Trusty TEE. https://source.android.com/security/trusty.

[33] Coq Developers. 2018. The Coq Proof Assistant. https://coq.inria.fr.

[34] LinuxBoot Developers. 2019. Linux as Firmware. https://linuxboot.org.

[35] McSema Developers. 2018. Framework for Lifting x86, AMD64, and AArch64 Program Binaries to LLVM Bitcode. https://github.com/trailofbits/mcsema.

[36] SMACK Developers. 2018. SMACK Software Verifier and Verification Toolchain. http://smackers.github.io.

[37] The LK Developers. 2019. The LK embedded kernel. https://github.com/littlekernel/lk.

[38] The Tock OS Developers. 2018. Tock Embedded Operating system. https://www.tockos.org.

[39] Loïc Duflot, Olivier Levillain, Benjamin Morin, and Olivier Grumelard. 2009. Getting into the SMRAM: SMM Reloaded. https://cansecwest.com/csw09/csw09-duflot.pdf.

[40] Jakob Engblom. 2017. Finding BIOS Vulnerabilities with Symbolic Execution and Virtual Platforms. https://software.intel.com/en-us/blogs/2017/06/06/finding-bios-vulnerabilities-with-excite.

[41] Facebook. 2015. Introducing OpenBMC: an Open Software Framework for Next-Generation System Management. https://code.fb.com/open-source/introducing-openbmc-an-open-software-framework-for-next-generation-system-management/.

[42] Robert W. Floyd. 1967. Assigning Meanings to Programs. In *Proceedings of Symposia in Applied Mathematics, Vol. 19*. 19–32.

[43] Corey Gough, Ian Steiner, and Winston Saunders. 2015. *Energy Efficient Servers: Blueprints for Data Center Optimization*. Apress.

[44] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 653–669.

[45] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*. 1–17. https://doi.org/10.1145/2815400.2815428

[46] Chris Hawblitzel, Jon Howell, Jacob R Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 165–181.

[47] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. https://doi.org/10.1145/363235.363259

[48] Galen C. Hunt and James R. Larus. 2007. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.* 41, 2 (2007), 37–49. https://doi.org/10.1145/1243418.1243424

[49] Intel. [n. d.]. The EFI Development Kit II (EDKII) Project. https://github.com/tianocore/tianocore.github.io/wiki/EDK-II.

[50] Intel Security Advisory. 2017. Intel Active Management Technology, Intel Small Business Technology, and Intel Standard Manageability Escalation of Privilege. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00075.html.

[51] Intel Security Advisory. 2018. Intel Q317 ME 6.x/7.x/8.x/9.x/10.x/11.x, SPS 4.0, and TXE 3.0 Security Review Cumulative Update. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00086.html.

[52] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the Foundations of the Rust Programming Language. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 66:1–66:34. https://doi.org/10.1145/3158154

[53] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*. 207–220. https://doi.org/10.1145/1629575.1629596

[54] K. Rustan M. Leino. 2008. This is Boogie 2. https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2

[55] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

[56] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. The Case for Writing a Kernel in Rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys)*. 1:1–1:7. https://doi.org/10.1145/3124680.3124717

[57] Frédéric Mangano, Simon Duquennoy, and Nikolai Kosmatov. 2017. Formal Verification of a Memory Allocation Module of Contiki with Frama-C: A Case Study. In *Proceedings of the 11th International Conference on Risks and Security of Internet and Systems (CRiSIS)*. 114–120.

[58] Mark Ermolov. 2018. Twitter. https://twitter.com/_markel___/status/982364102449393668.

[59] Ivan Herrera Mejia and Zeev Offen. 2017. Interface for Communication Between Circuit Blocks of an Integrated Circuit and Associated Apparatuses, Systems, and Methods. US Patent 9,594,413.

[60] Ronald Minnich. 2017. Replace Your Exploit-Ridden Firmware with Linux. Open Source Summit Europe + ELC Europe. https://www.youtube.com/watch?v=iffTJ1vPCSo.

[61] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. 252–269. https://doi.org/10.1145/3132747.3132748

[62] Big Switch Networks. 2019. Open Net Linux - Hardware Support and Certification. https://www.opennetlinux.org/hcl.

[63] Big Switch Networks. 2019. Open Network Linux. http://opennetlinux.org/.

[64] Niko Matsakis. 2016. Introducing MIR. https://blog.rust-lang.org/2016/04/19/MIR.html.

[65] OpenSwitch. 2019. OpenSwitch (OPX) Network Operating System. https://www.openswitch.net/.

[66] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V Out of NFV. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 203–216.

[67] Sean Peters, Adrian Danis, Kevin Elphinstone, and Gernot Heiser. 2015. For a Microkernel, a Big Lock Is Fine. In *Proceedings of the 6th Asia-Pacific Workshop on Systems (APSys)*. 3:1–3:7. https://doi.org/10.1145/2797022.2797042

[68] Open Compute Project. 2014. Open Compute Networking Project Workshop. https://www.opencompute.org/wiki/Networking/Workshop-2014-07.

[69] Zvonimir Rakamarić and Michael Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*. 106–113. https://doi.org/10.1007/978-3-319-08867-9_7

[70] Zvonimir Rakamarić and Alan J. Hu. 2009. A Scalable Memory Model for Low-Level Code. In *Proceedings of the 10th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*. 290–304. https://doi.org/10.1007/978-3-540-93900-9_24

[71] Dmitry Sklyarov. 2017. Intel ME: The Way of the Static Analysis. https://www.troopers.de/troopers17/talks/772-intel-me-the-way-of-the-static-analysis/.

[72] Alexander Tereshkin and Rafal Wojtczuk. 2009. Introducing Ring -3 Rootkits. https://invisiblethingslab.com/resources/bh09usa/Ring-3Rootkits.pdf.

[73] Rafal Wojtczuk and Joanna Rutkowska. 2009. Attacking Intel Trusted Execution Technology. https://invisiblethingslab.com/resources/bh09dc/AttackingIntelTXT-paper.pdf.

[74] Rafal Wojtczuk and Joanna Rutkowska. 2009. Attacking SMM Memory via Intel CPU Cache Poisoning. https://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf.

[75] Rafal Wojtczuk and Joanna Rutkowska. 2011. Attacking Intel TXT via SINIT Code Execution Hijacking. https://invisiblethingslab.com/resources/2011/Attacking_Intel_TXT_via_SINIT_hijacking.pdf.

[76] Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. 2016. A Practical Verification Framework for Preemptive OS Kernels. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV)*. 59–79. https://doi.org/10.1007/978-3-319-41540-6_4