BedRock Systems Inc

Developing With Formal Methods at BedRock Systems, Inc.

(c) 2022 BedRock Systems, Inc.

This paper incorporates material from:
G. Malecha, G. Stewart, F. Farka, J. Haag and Y. Hirai, “Developing With Formal Methods at BedRock Systems, Inc.,” in IEEE Security & Privacy, vol. 20, no. 3, pp. 33-42, May-June 2022, doi: 10.1109/MSEC.2022.3158196.

URL:https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9760701&isnumber=9782817

The BedRock HyperVisor™ is a commercial, highly concurrent, verified virtualization platform that employs formal methods to enable proofs of complex, lock-free concurrent code; support automating proofs of large programs; and integrate with "informal" parts of the software lifecycle.

Building on academic research but with feet firmly planted in industrial applications, BedRock Systems™ is in the process of building the BedRock HyperVisor (BHV™), a formally verified, highly concurrent, microkernel-based commercial hypervisor. By formally verified, we mean that the C++ and assembly-code implementation of the operating system is proved correct in the Coq proof assistant.[1] By highly concurrent, we mean that we use, and verify, lock-free data structures in core parts of the implementation. That the BHV is microkernel based means that most of the system runs in user mode on top of a small, kernel-level program, which provides bare-bones abstractions such as address spaces and interprocess communication (IPC).

To verify the BHV, BedRock Systems is pioneering the use of formal methods (FM) at scale. This experience report explains how we use FM and why. Our experience shows that advances in FM techniques finally enable them to integrate well in the standard software development process. In essence, FM-based software development is "just" mathematically rigorous software engineering. Additionally, the design aims of FM align with software engineering best practices. Further, our experiences suggest that FM techniques are increasingly able to directly address (and in some cases, improve upon) current best practices in software engineering.

Despite the alignment of aims, the road is not always an easy one. Pioneering FM at scale means that we must build many tools and libraries ourselves. Although FM tools still lag behind more mainstream tools, we believe they have matured to the point of being usable in an industrial context. Further, development and adoption of these tools is growing, and we anticipate the situation will continue to improve.

Beyond the tools themselves, writing verified software, even in mainstream languages, still suffers from a dearth of libraries. One of the main benefits of mature ecosystems such as C++ is the availability of libraries, but very few of these libraries are formally specified, let alone verified. Our own work has already begun to address this problem internally, and we believe that it is just a matter of time before developers are able to use libraries that are formally verified.

Outline

We report on BedRock System's work to build a verified, performant hypervisor. Achieving this goal requires system-, as opposed to program-level, FM. These requirements inform our FM process, which starts in the design phase. Here, the ability to consider problems abstractly enables us to rapidly explore the design space. This process often produces replicable patterns that can be precisely documented, informing future design and development.

We connect this design-level formalization to the running code through the rest of the software engineering process. Making this connection formal ultimately delivers the correctness guarantees that FM advertises, e.g., eliminating bugs, but it also brings two challenges: the development of a program logic for a mainstream programming language, C++, and extensible automation to translate high-level reasoning principles into robust proofs about low-level code.

Scaling FM requires solving nontechnical problems as well. We explain how we manage our FM development process for both predictability and reliability. Our approach is agile based and centers on daily standups and frequent code reviews. We focus our efforts on concentrated verticals, which we call spikes, to ensure that specifications are both usable by clients and provable for implementers. Beyond delivering better code, focused group work also improves onboarding and on-the-job training, which are crucial when working on the bleeding edge. Our experiences suggest that driving otherwise open-ended research from concrete "in the code" problems is crucial to predictable execution, but can be a difficult shift from a more theoretical academic mindset.

The BedRock HyperVisor™ (BHV™)

The aim of FM at BedRock Systems is to develop a flexible and ultrasecure compute platform. For flexibility, the BHV follows a microkernel architecture (see Figure 1). Independent applications provide decentralized services for features such as virtual compute and networking. This modular architecture enables us to customize the BHV's feature set by selecting different applications in different contexts. However, it also requires us to establish the system's correctness in a similarly modular fashion.

Virtual Compute

The BHV's core use case is as a virtualization platform. This functionality is provided by the BedRock Virtual Machine Monitor (VMM), which virtualizes a single unmodified computer. Although the code is complex and tied to the BHV's application programming interface (API), the top-level specification is simple: the BedRock VMM is the correct implementation of a bare-metal computer. We call this property the BedRock Bare-Metal Property™

Formally, the Bare-Metal Property™ states that the BedRock VMM is a timing-insensitive refinement of the hardware specification. (Note that the FM work on BHV currently focuses on the ARM architecture, but other architectures will be supported in the future.) Informally, this means that anything that happens when running a guest on the BedRock VMM could happen on a compliant system. Guests act as if they are running on isolated bare metal and achieve a similar level of security guarantee. This enables consolidation without the added risk of lateral (cross-VM) attacks on the virtualization platform.

The bare-metal property is enhanced by BedRock Active Security®, which acts as a runtime monitor for virtual machines. In terms of a bare-metal guarantee, an Active Security-enabled guest runs on a standard processor with (potentially guest-specific) security extensions, e.g., register protection or write-execute mutual exclusion. With this specification, security-compliant guests cannot distinguish an Active Security-enabled VMM from hardware. Guests that violate the policy, however, see a machine that includes the runtime security monitor.

Virtual Networking/Communication

Leveraging the modularity of the BHV architecture, we extend the single-computer correctness condition to multiple computers using a virtual network switch. The virtual switch (vSwitch) enables guests to communicate with one another using the virtual I/O device net-work protocol.[2] As in the VMM, the software implementation enables expressive and dynamic policies to be enforced in the vSwitch. The architecture of the vSwitch occurs repeatedly throughout the BHV, e.g., in the console multiplexer (UMX) and other services. The commonality enables abstraction and reuse within our implementation.

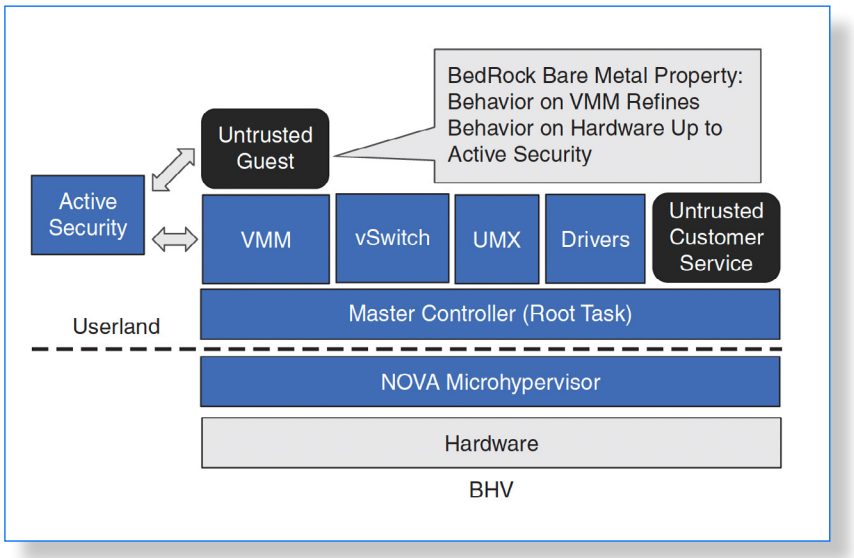

Figure 1. The BHV. Top-level applications such as the Virtual Machine Monitor (VMM) and virtual switch (vSwitch) sit atop the master controller. The master controller provides support for userland services, which are used by independent programs to provide application-specific services. The NOVA microhypervisor provides minimal primitives that enable these features. The Active Security module enforces security policies on guests at runtime. UMX is the system console multiplexer.

An Extensible, Distrusting Platform

The BHV supports running unverified applications side by side with verified ones, without compromising the integrity of the verified applications. Without this requirement, we could verify a weaker specification that requires a disciplined use of the API. However, these weaker specifications are insufficient when adversarial code might be running on the platform. To address this issue, the BHV's top-level specification uses an operational semantics based on process calculi in which untrusted processes, such as guests, are modeled by their machine-level behavior. Although verbose in some cases, operational semantics enables us to model untrusted code as simply "what the bits say."

Supporting unverified applications is crucial in practice because it enables a path to verified systems rather than mandating an all-or-nothing mentality. This enables both "preview" releases, which may contain unverified functionality, and the ability to support customer applications, which have not been verified.

Formal Methods at BedRock Systems

Achieving formal guarantees at the level of the BHV places demanding requirements on our FM techniques. In contrast to many code-based verifications, which focus on verifying a single application, establishing the correctness of the BHV requires reasoning across multiple programs coordinating through low-level mechanisms, such as shared memory. The need to interoperate with unverified applications limits the assumptions that we can make across these boundaries, which sometimes requires us to return to first principles when developing a verification strategy. Consider, for example, shared memory queues. When both parties are trusted, a server can assume that clients follow the protocol; however, when the client could be malicious, the server is still obligated to behave correctly to ensure that the bad actor cannot compromise the guarantees of well-behaved clients.

These constraints require our techniques to apply at the machine-code level while still enabling the use of language-specific reasoning principles in contexts that we prove are well behaved. The presence of unverified components adds additional complexity to an already challenging problem, but is needed to enable the transition to a verified software stack.

Language-agnostic, system-level software verification demands a unifying formalism, one that can uniformly talk about the values of program variables, state of hardware devices, application-level protocols, and more. Further, verification techniques must support open-world reasoning, which means that the proofs of individual threads can be combined with the proofs of other threads (and arbitrary contexts) to establish a whole-system guarantee.

Separation logic[3] satisfies these exacting requirements. Modern separation logic, as embodied by the Iris library,[4] is built around first-class resources that can be owned by executing entities, e.g., threads, and invariants, which provide a mechanism for atomically sharing these resources between threads. At the heart of separation logic is the separating conjunction (written P * Q), which implicitly expresses disjointness of the resources in P and Q. Disjointness is the right default for building compositional systems, and the succinct manner of expressing it in separation logic leads to elegant and modular specifications. Although using resources to reason about memory is standard practice, the generality of resources means that we can use them to track other kinds of system state, such as kernel objects and the state of hardware devices. Because separation logic treats resources uniformly, it is also possible to define and compose abstractions that encapsulate resources of distinct types, e.g., those that bundle program variables and kernel resources.

BRiCk—A Program Logic for C++

To verify C++ programs, we need formal reasoning principles for the language fragment that the programs use. These rules are codified in BRiCk,5 the BedRock C++ program logic. BRiCk builds off of the Clang/LLVM[6] compiler front end and uses its source-level abstract syntax to represent C++ programs.

BRiCk axiomatizes reasoning principles for each type of node. We justify these reasoning principles informally by appealing to the C++ standard[7] and academic work-formalizing aspects of both C[8] and C++.[9] The choice to formalize C++ axiomatically, rather than operationally, is primarily a pragmatic one: it enables us to easily underspecify language constructs, grow the supported feature set over time and, we believe, accurately model the standard.

Automation for BRiCk

Automation is crucial to scaling program verification to large and evolving code bases. BedRock System's automation for BRiCk is built around the mental model of symbolic debugging, where the current state is expressed formally in separation logic and the core automation interprets program fragments against this state. To be understandable, the automation must preserve program-specific abstractions as much as possible. Reaching into a complex invariant to justify a read may enable the verification to make progress, but the resulting state is often incomprehensible to clients who wish to remain insulated from the definition of the invariant. To achieve this, library developers write, and prove, "hints" encapsulating common reasoning patterns that are not immediately obvious from the code. These hints cover coding patterns sanctioned by the library writer and are applied automatically when clients follow these guidelines. Deviating from these coding patterns leads the automation to get stuck, but in an understandable state that facilitates debugging.

BedRock System's automation also provides more manual tactics for reasoning about language constructs that deserve special attention. For example, loop invariants are notoriously difficult to find automatically, so we provide tactics to specify loop invariants manually. Our collection of tactics also includes some more specialized tactics for reasoning about common coding patterns such as cas-loops and foreach-loops. Beyond their general usefulness, these tactics also document reusable reasoning patterns. We have also experimented with tactics that apply more aggressive heuristics. However, we tend to prefer slightly more verbose (but maintainable) proof scripts as these more aggressive tactics can sometimes fail in unpredictable ways.

Design-Time FM

Although it is tempting to think of verification as another step added to the end of the software development process, we have found that this approach misses out on much of the value that FM can provide. Undoubtedly there is value in the final proof, but FM insights can dramatically improve code quality even before we verify the code. Beyond improved code quality, design-level FM often produce easier-to-verify code that is more future proof. This is especially important because it avoids expensive refactorings of both the code and the proof. Although FM reviews are an important input to the final verified product, systems engineers generally find that such reviews help to clarify high-level design thinking.

We consider two instances, among many, where design-time interaction with systems engineers clarified an existing design and influenced a refactoring. In the design phase, many of the insights of FM could have come equally from architectural and design reviews. The added value of FM lies in the ability to carry the insights forward, from high-level discussion down to code-level specification and proof artifacts. Many insights of FM are inspired by abstractions that work well in functional programming, e.g., higher-order functions such as folds. The expressivity of higher-order separation logic allows us to use these abstractions in low-level C++ code, e.g., modeling a C++ class as an effectful function, rather than its first-order representation to insulate developers from later extensions.

Locking Protocols in the VMM

Hardware virtualization is a complex task with a demanding specification. In the VMM, the problem is exacerbated by the fact that much of the "code" that we are working with is not under our control. Specifically, our verification cannot make any assumptions about the guest code, its memory-access patterns, or its use of synchronization.

When virtualizing a multicore machine, proper synchronization is crucial. Even though our implementation seeks to be as efficient as possible, it is sometimes necessary to pause the virtual CPUs to give the virtualization layer a snapshot of the system that it can inspect in a stable way. We refer to this pausing as round up; one module, typically Brass, requests that the virtual CPUs of a guest synchronize to prevent memory access during a certain period of time.

Specifying round up informally (even with a working implementation) turned out to be difficult for technical reasons: first, virtual CPUs can run either on the hardware or in our software instruction emulator; second, both virtual CPUs and virtual devices can access the memory.

Ultimately, separation logic provided a precise and extensional explanation that was understandable by both systems and FM engineers. The central idea was to model a memory "lock" token, mediating access to the guest's memory. The full ownership of this token guarantees exclusive access to the guest's memory regions. A partial ownership allows shared access which, while still safe, does not provide atomicity guarantees. When leveraging hardware virtualization, the program has no fine-grained control over the guest's actions, so the hardware thread owns a portion of the lock token (recall that the guest manages its own synchronization). The semantic condition of taking a step on behalf of the guest shows that the memory token is also needed by the instruction emulator. When a guest virtual CPU is not running on the hardware or being emulated in software, it can relinquish its portion of the token, thus ceding its ability to access memory. The notion of delegatable ownership is a central tenet of separation logic and is a useful strategy when modeling systems such as this one. By expressing the ability to do something as a first-class resource, we avoid the need to think about all the entities that could do it.

Architecting Services

IPC is a staple of interprocess coordination in microkernel- based architectures. NOVA[10] provides a fast, intra-core IPC mechanism, and the BHV builds higher-level concepts of services and sessions to enable sharing kernel resources such as memory and semaphores. The stateful, reactive nature of services is a fundamentally different development paradigm than the "direct" programming model; care must be taken when developing applications like these, especially when clients should not always be trusted. Code reviews revealed that many early services suffered from the same mishandling of subtle situations such as session lifetime. Further, the ad hoc development of many services meant that we lost opportunities to share code.

To support the reactive programming model, we developed a code- and specification-level template for writing and specifying verifiable services. The abstraction encapsulates session management by decoupling connect and disconnect and requiring developers to think about disconnect logic under arbitrary states of the protocol. Code that fits into the template is highly regular and has a much higher chance of being specifiable than more ad hoc code. Further, the library completely encapsulates the subtle logic around session lifetime that plagued earlier code. Specifying this abstraction turned out to be quite difficult due to the fact that certain operations such as connect and disconnect are silent in the BHV. Our approach uses purely logical "callbacks" that enable servers to logically set up services before the application is notified. These callbacks can transition the service's specification state, and the code can "catch up" when it learns of the new connection in the first message. This approach enables us to provide an abstraction that is much easier to reason about, and to hide the implementation and proof details from users.

The specification here relies on sophisticated features of concurrent separation logic but, in the end, the abstraction is fairly intuitive and it enables a simple programming model. Situations such as these, which have occurred several times to date across our codebase, highlight the value of separation logic as a means to simplify code and proofs.

Verification Engineering

Although it is common for design reviews to uncover misunderstandings and bugs, being certain the code is bug free requires verifying that it satisfies its specification. At BedRock Systems, we have successfully verified production code at all levels of our virtualization stack. Beyond pure C++ libraries, our verification also covers user-space device drivers (for a serial driver and a direct memory access driver), a concurrent terminal mutiplexer, and portions of other applications and NOVA.

Developing Proofs

After the specification is developed and the code is written (not always in that order), BedRock Systems' process prescribes a detailed code review with an FM engineer. Together, the two (or more) develop an informal proof that the code satisfies the specification. In most cases, this involves a sketch of the "class invariant" and a Hoare outline for the nontrivial functions provided by the class. These outlines directly connect the specification and the code—there is no additional layer of modeling.

This review is often an iterative process, and it is essential that we can get through a cycle rapidly. A common approach is to formulate a class invariant and then expand and refine it as we incorporate new bits of functionality in successively greater detail. In practice, much of this iteration is carried out over a less formal medium, e.g., a (virtual) whiteboard. Separation logic resources are often nicely conceptualized graphically, and we find that sketching boxes and moving them around can be very helpful to explore abstractions and implementations.

Once we have covered the core functionality of the class, we formalize the definitions. Generally, this process is rather straightforward, but it relies on good working knowledge of the verification tool (Coq, in our case). At this level, we are choosing specific data representations, such as whether to use a list or a finite map, how to express the relationship between an array in C++ and its length, and so on. Many of these problems can be solved prescriptively, e.g., "always use arrayR to represent an array"; however, we do not claim to have all of right answers yet. But as we verify more code, we refine patterns and create new ones. Beyond providing a codified best practice, patterns such as these also enable us to narrow the focus of the automation's development.

When code does not fit within our existing abstractions, we look to expand them, develop new ones, or rework the code to fit within them. Although rewriting code may seem to indicate that our techniques are not up to the challenge, we note that developers often prefer simplified code, and on many occasions, very subtle bugs have been found around these points. Mathematically, separation logic can scale to arbitrarily complex code, but keeping reasoning simple is often the better path in the long term. When proposing code changes, we always consider runtime costs, readability considerations, and limitations (or enablements) of the new code.

Bugs

Throughout verification we found and fixed a number of bugs across all parts of our stack. Although many bugs are found during testing, we have determined that concurrency bugs, resource leaks, and error-handling logic are especially difficult to test and are therefore often caught by code reviews or during proofs. For example, in developing our shared-pointer library, we ran a significant number of randomized tests without uncovering several bugs caught during the FM code review. Another instance of a subtle logic bug was a synchronization issue within NOVA, which could cause incorrect continuation to be used when switching threads (execution contexts in NOVA). In these instances, and many others involving concurrency, we found that state-based reasoning, which focuses on what is true in a particular state, is more useful than trace-based reasoning, which describes the operations that took place to arrive at a given state, for zeroing in on problems.

Although the previous two bugs were found during the FM review phase, in other instances, our reviews missed subtle bugs that were ultimately uncovered as we formalized the proof. In the UMX, we uncovered a synchronization issue that would occur if a client disconnected at precisely the right time during data forwarding. Ultimately, this bug could cause data loss, but reliably triggering it in a testing scenario would be extremely difficult.

Beyond logical bugs, FM code reviews and proofs uncovered portability and standards compliance issues within our code. Portability bugs often arise in code that implements low-level data marshaling and might rely, e.g., on the endianness of the system or the ability to perform unaligned reads and writes. Although strict adherence to the C++ standard may seem overly pedantic, we believe that it is the only viable path forward in the long term. Optimizing compilers crucially rely on undefined behavior to enable optimizations, and non-compliant code can result in bugs at higher optimization levels that are difficult to track down because they do not exist in debug builds. The C++ standard is the contract between developers and compiler writers; if developers need something that the standard does not provide, the standard needs to be expanded to provide it.

Proof Maintenance. Keeping proofs in sync with code is essential to maintaining high quality through refactorings. At BedRock Systems, our continuous integration (CI) checks that all proofs succeed before any merge to the main branch. Overall, we have not found this to be particularly burdensome as well-designed verified code tends to be fairly stable. When code changes are small (and correct), our automation is often able to discharge new obligations automatically, and no changes to the proof scripts are necessary. Inevitably though, more complex changes, especially those that affect class invariants and concurrency protocols, require manual proof maintenance. Robust proof automation and appropriate abstractions mitigate the burden to some degree, but do not scale to all interface- and specification-level refactorings. In these situations, the cost is unavoidable: significant algorithmic changes will require fundamentally different proofs.

Beyond guaranteeing bug freedom, one benefit of proofs over testing is that they are hyperlocal. Proof failures tell you exactly the point where the code may be broken as well as giving you the symbolic state that is problematic. This information is especially useful in tracking down concurrency "Heisenbugs" that occur infrequently and are therefore difficult to test and reproduce.

When working on improving automation, checking proofs in CI is crucial as proofs of code double as test cases for automation. Timing statistics from CI runs provide useful information around automation performance. Line-count statistics of new proofs are a good first-order signal of the effectiveness of the automation because verbose proofs often point to shortcomings in automation, although it is important to factor in complexity of the underlying code as well.

Extensible Automation

The need to understand and maintain proofs requires that we express them at a high level. Making this possible for larger C++ programs that evolve over time requires that we keep the proofs small by automating the administrative reasoning necessary to complete a proof. To facilitate high-level reasoning, our automation is post-facto extensible using stylized reasoning principles that we call hints. Hints are justified once and applied auto-matically by automation whenever the situation merits it. These hints enable us to reason at a natural level of abstraction while also insulating clients from some of the more technical details of specifications and code.

We contrast this semiautomatic reasoning with more manual reasoning traditionally provided by interactive theorem provers and the Iris Proof Mode (IPM).[11] The IPM provides fine-grained context management and low-level primitive tactics for reasoning about separation logic formulas. This sort of reasoning is ideal for subtle proofs that require tricky resource management; our metatheory leverages this verification approach extensively. When verifying C++ programs, however, we find that large parts of the proof are "follow-your-nose" proofs. Indeed, in many instances, the program is effectively the proof, and the proof is merely bookkeeping. In these circumstances, it is ideal to teach the automation to follow its own nose so that the verification engineer can focus on subtle aspects of the verification.

We offer two instances where custom, but reusable, hints accelerated proof development. The first arose when verifying higher-level specifications on top of lower-level ones for the microkernel. When verifying the microkernel, we need to provide specifications that are maximally distrusting of applications running atop it. We achieve this by providing low-level, "undisciplined" specifications that can support arbitrary, especially concurrent, usage. In practice, however, these specifications can be both difficult to read and program against. On top of these specifications, we can prove simpler specifications that are able to hide details when using the API in a more restricted setting. As a simple example, if we know that a capability must refer to a semaphore object, then we do not need to consider error codes from the microkernel that correspond to capability mismatches. Proving the well-behaved specifications from the unsafe ones can be onerous, but is generally not complicated; however, the tedium of this task can be alleviated by a small number of generic insights, which are easily expressed within our hint infrastructure. Using these hints, we reduced the size of some proofs by more than half, which greatly increased the readability and maintainability of the proofs as the specifications evolved. Ultimately, the proofs became fairly close to the proof outlines written by experts because the automation was extended with the expert's strategy for reasoning.

A second instance where hints were able to abstract low-level details arose when working with arrays, and especially with array initialization and destruction. For modularity purposes, BRiCk's semantics describes array initialization compositionally by initializing each array cell sequentially. Although this provides a clear specification, using this approach becomes quite costly when working with large arrays. Providing special hints for default-initializing primitive arrays enabled us to fuse many reasoning steps together, resulting in more natural descriptions of the program state. Our hints can also codify patterns for reasoning about array accesses in a natural way by decomposing (and recomposing) a large array into locations of interest and the rest of the array. These sorts of "borrows" constitute a large part of the administrative reasoning necessary in low-level C++ programs, and our ability to express these borrowing patterns generically generally means that the automation can churn through array reasoning with relatively little manual intervention.

Industrial Programming Languages

One of the largest sources of complexity in verification is not the code we write but the language in which we write it. C++, and modern programming languages in general, are both large and complex and provide formal reasoning challenges in and of themselves. Although necessary to address, we note that these challenges arise on a per-language rather than a per-program basis, so most FM practitioners need not worry about these issues. Further, BRiCk already addresses these issues for the fragment of C++ that it supports.

Supporting Large Languages. The size of modern languages means that we must formalize them incrementally. To facilitate this, BRiCk's semantics is directly expressed as a program logic, rather than as a derived logic on top of an underlying operational semantics. This approach allows us to leverage the built-in modularity of separation logic to modularize our semantics. It also makes it natural to underspecify the semantics of particular language constructs, a property that is essential early on and still useful when working with multiple related languages, e.g., C++14 and C++17.

Although many language features can be desugared to simpler primitives, we avoid this when possible. This is partly for soundness as some transformations only refine the high-level specification; however, there are also reasons to support the sugar natively. Consider virtual functions in C++.
In theory, we could desugar these to tables of function pointers, but doing so (even abstractly) would expose reasoning principles not justified by the C++ standard. Further, desugaring the construct would require all developers using the construct to reason about the desugaring, something which is clearly undesirable. Supporting the feature directly not only keeps us closer to the standard but also enables us to build opinionated abstractions and automation for the use cases of virtual functions.

Supporting Sophisticated Language Features. Industrial languages also have features that are difficult to reason about. The archetypical difficult language feature in C++ is the object model, which is front and center in C++ semantics. Keep in mind that the concurrent memory model is another cross-cutting feature, albeit one more limited in scope, because regular C++ variable accesses must be data-race free. Although developers often think of C++ pointers as virtual addresses, the C++ language puts significantly more structure on them. This additional structure gives optimizers the ability to reason more aggressively about programs but comes at the cost of more bookkeeping in formal proofs. For example, our semantics tracks pointer provenance to rule out undefined behavior that arises from low-level pointer manipulation.

Although not pervasive, the interoperation between C++ and assembly is another necessary part of low-level programming. Beyond just giving a semantics for assembly, we are forced to answer questions such as, "What is the effect of sharing memory between multiple programs?" Empirically, we know what compilers do, but the standard text is silent on many of these low-level questions.

To avoid these issues in the short term, we make judicious, simplifying assumptions that seem to hold in practice. For example, BRiCk assumes that deallocating memory does not invalidate the pointer, an assumption not sanctioned by the standard. This choice requires that we add liveness side conditions to certain operations to avoid obvious unsoundnesses. Researchers suggest[12] that this seems necessary (in C) to support common, low-level programming idioms and is therefore likely justified by compilers in practice.

Managing FM Teams

In the past few years at BedRock Systems, we have experimented with a few different approaches for managing FM work. In this section, we discuss some of the lessons that we learned. We underscore that the value of FM is directly correlated with its pervasiveness. When FM are involved early and regularly throughout development, things tend to go smoothly. Delaying FM involvement until the end makes them more difficult to accurately plan and ultimately prove the code correct.

At a high level, our experience suggests that managing FM teams is not fundamentally different from managing "normal" development teams. At a lower level, we found that focused efforts exercising specifications from both the client and implementation sides are highly effective at delivering high-quality, reusable, and verified code. We refer to this approach as spike-based verification because it is built around focused verticals.

In the next section, we focus on our spike-based verification efforts on the UMX, a multithreaded service that implements a console multiplexer.

The UMX Spike

The UMX spike was planned from the top down, from a top-level specification to the implementation, but generally completed from the bottom up, from application dependencies to the top-level specification. During the initial planning, we identified two high-level components: the control plane, which interacts with clients, and data plane, which forwards data.

Rather than splitting the team equally between these components, we opted to focus first on the data plane and then the control plane. Consolidating resources improved collaboration and resulted in timely and constructive feedback. For example, the team was able to identify a key missing abstraction around string literals early on and developed preliminary automation for them that was immediately used by the rest of the team.

It is useful to note that the separation of clients and implementation via a formal specification generally enables a greater degree of parallelism than is possible in traditional software engineering. Client verification can start even before an implementation exists and certainly before a proof is completed. We find that our goal of automatable abstractions tends to insulate client code from shallow, specification-level changes, allowing them to (relatively quickly) adapt proofs when underlying interfaces change.

The control plane verification did not proceed as smoothly as the data plane verification due to unforeseen complexities in two components: the service library and use of shared memory. The underlying issue in both of these stemmed from subtle complexities and insufficient expert bandwidth. This is especially problematic at external interfaces, where the behavior of clients is largely unconstrained and therefore sometimes difficult to conceptualize. In these circumstances hav-ing experts on hand is essential, and even with them, it is sometimes challenging to estimate the difficulty of a task before you are already deep in it.

Although experiences like these do arise, they occur less frequently than one might think. Bleeding edge work often comes with risks and slowdowns, but in many cases, this is not fundamentally different than state-of-the-art software engineering. As in that context, it is crucial to avoid early overgeneralization and scope creep. We have found that the combination of spike-based verification and agile-style sprints helps with this. Rather than solving problems in a vacuum, engineers focus on real use cases attached to real code. With a specific use case in mind, a solution can often take the place of the "perfect" solution and enable for-ward progress (see Figure 2). New use cases (and fresh eyes) often inspire new insights that can be used to generalize existing specifications. In practice, the proof burden introduced by generalizing an interface is often significantly reduced by "adapter" hints provided to our extensible automation.

The scrum approach to FM is highly effective at transferring skills among developers. Short, daily sync meetings helped to connect more and less experienced developers. The focused scope of the group also greatly reduced the context switching overhead in collaboration and resulted in both faster and better feedback across the board.

Hiring and Training for FM

Spike-based formal verification improves onboarding, but hiring for FM is still difficult. There are few candidates with general FM expertise and even fewer with expertise in specialized areas such as concurrent separation logic. The universities teaching FM to undergraduates help mitigate the gap, but we have found that teaching FM "on the job" is necessary. Even Ph.D.s with deep experience in separation logic require training to transition from academic FM (which often focus on smaller programs and meta-level issues) to industrial program verification. Over time, this transition period at BedRock Systems has shortened, and we believe that good training material and exposure through spike-based verification will further reduce the overhead.

When hiring general software engineers, we have found that candidates with functional programming background are able to pick up FM much more readily than those without such exposure. In part, this is attributable to the fact that Coq is built around a functional programming language (Gallina), but we believe that it is more than that. Functional programming languages tend to focus on minimalism, which seems to train developers to more quickly separate the core problem from the noise surrounding it. This skill is transferable not only to specification writing but also to the design of good interfaces.

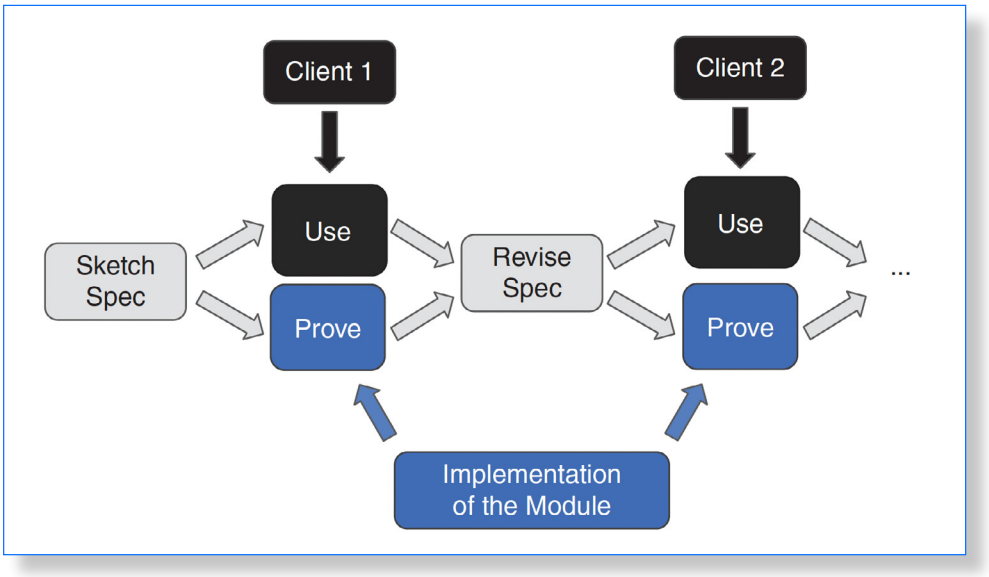

Figure 2. The lifecycle of a specification (spec). We verify code in spikes that address both users and implementers of specified code, iteratively refining specs until they are both realizable and useful. Later clients benefit from usability improvements that occur in previous verification cycles.

Incorporating FM into all aspects of the software development lifecycle, from software system design to implementation to code maintenance, has the potential to revolutionize the software industry. But making pervasive FM a reality requires solving deep technical and nontechnical challenges, many of which we have begun to address at BedRock Systems.

On the technical side, we see refining language standards and improving automation as crucial barriers that are beginning to fall. Reasoning about industrial languages such as C++ is necessary but raises difficult problems in semantics, especially around complicated corners of standards. This is an active area of research, and increasingly, standards committees (especially the C standard committee) are seeing the value in it. Engineering verification to scale to complex industrial code bases means building automatable, but also highly expressive, program logics. BedRock System's automation for C++ supports a hybrid of highly automated reasoning when possible and deliberate reasoning when necessary. The need for both is essential as programs grow not only in size but also complexity.

On the nontechnical side, it is crucial to expand proficiency in FM across the board. Specialized FM experts will still be necessary, but we believe that much of the knowledge required for verification could be made accessible to undergraduates. Mainstream interest in Rust and the growing popularity of functional programming languages both suggest that attitudes toward new technologies are changing in a positive way for FM. In the meantime, on-the-job training can compensate for a lack of general knowledge.

References

1. Y. Bertot and P. Castran, Interactive Theorem Proving and Program Development: Coq'Art the Calculus of Inductive Constructions. New York, NY, USA: Springer-Verlag, 2010.
2. M. S. Tsirkin and C. Huck, "Virtual I/O Device (VIRTIO)," OASIS Virtual I/O Device (VIRTIO) Technical Committee, Version 1.1, [Online]. Avail-able: https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.html
3. J. Reynolds, "Separation logic: A logic for shared mutable data structures," in Proc. 17th Annu. IEEE Symp. Logic Comput. Sci., 2002, pp. 55–74, doi: 10.1109/LICS.2002.1029817.
4. R. Jung et al., "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning," ACM SIG-PLAN Notices, vol. 50, no. 1, pp. 637–650, 2015, doi: 10.1145/2775051.2676980.
5. "BedRock systems/BRiCk." GitHub. https://github.com/bedrocksystems/BRiCk (Accessed: Nov. 30, 2021). C.
6. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in Proc. Int. Symp. Code Generation Optimization, San Jose, CA, USA, Mar. 2004, pp. 75–86, doi: 10.1109/CGO.2004.1281665.
7. Information Technology—Programming Languages—C++, ISO/IEC 14882: 2011.
8. R. J. Krebbers, "The C standard formalized in Coq," Ph.D. dissertation, Radboud University Nijmegen, Nijmegen, The Netherlands, 2015.
9. T. Ramananandro, G. Dos Reis, and X. Leroy, "Formal verification of object layout for C++ multiple inheritance," SIGPLAN Notices, vol. 46, no. 1, pp. 67–80, Jan. 2011, doi: 10.1145/1925844.1926395.
10. U. Steinberg and B. Kauer, "Nova: A microhypervisor- based secure virtualization architecture," in Proc. 5th Eur. Conf. Comput. Syst., Association for Computing Machinery, 2010, pp. 209–222, doi: 10.1145/1755913.1755935.
11. R. Krebbers, A. Timany, and L. Birkedal, "Interactive proofs in higher-order concurrent separation logic," SIG-PLAN Notices, vol. 52, no. 1, pp. 205–217, Jan. 2017, doi: 10.1145/3093333.3009855.
12. P. E. McKenney et al., Pointer Lifetime-End Zap, ISO/IEC JTC1/SC22/WG21 P1726R0, 2019. [Online]. Avail-able: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1726r0.pdf

Gregory Malecha is the director of formal methods at BedRock Systems, Inc., San Mateo, California, 94401, USA. His research interests include formal verification, automation, and programming languages. Malecha received a Ph.D. in computer science from Harvard University. Contact him at gregory@bed-rocksystems.com.

Gordon Stewart is a formal methods lead at BedRock Systems, Inc., San Mateo, California, 94401, USA. His research interests include formal verification and compiler correctness. Stewart received a Ph.D. in computer science from Princeton University. Contact him at gordon@bedrocksystems.com.

František Farka is a senior formal methods engineer at BedRock Systems, Inc., San Mateo, California, 94401, USA. His research interests include logic in computer science, type theory, and proof search. Farka received a Ph.D. in computer science from the University of St Andrews and Heriot-Watt University. Contact him at frantisek@bedrocksys tems.com.

Jasper Haag is a formal methods engineer at BedRock Systems, Inc., San Mateo, California, 94401, USA. His research interests include formal verification. Haag received a B.S. in computer science from the Massachusetts Institute of Technology. Contact him at jasper@bedrocksystems.com.

Yoichi Hirai is a senior software engineer BedRock Systems, Inc., San Mateo, California, 94401, USA. His research interests include modal logics for knowledge and concurrency. Hirai received a Ph.D. in computer science from the University of Tokyo. Contact him at yoichi@bedrocksystems.com.

(c) 2022 BedRock Systems, Inc.

This paper incorporates material from:
G. Malecha, G. Stewart, F. Farka, J. Haag and Y. Hirai, "Developing With Formal Methods at BedRock Systems, Inc.," in IEEE Security & Privacy, vol. 20, no. 3, pp. 33-42, May-June 2022, doi: 10.1109/MSEC.2022.3158196.

URL:https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9760701&isnumber=9782817