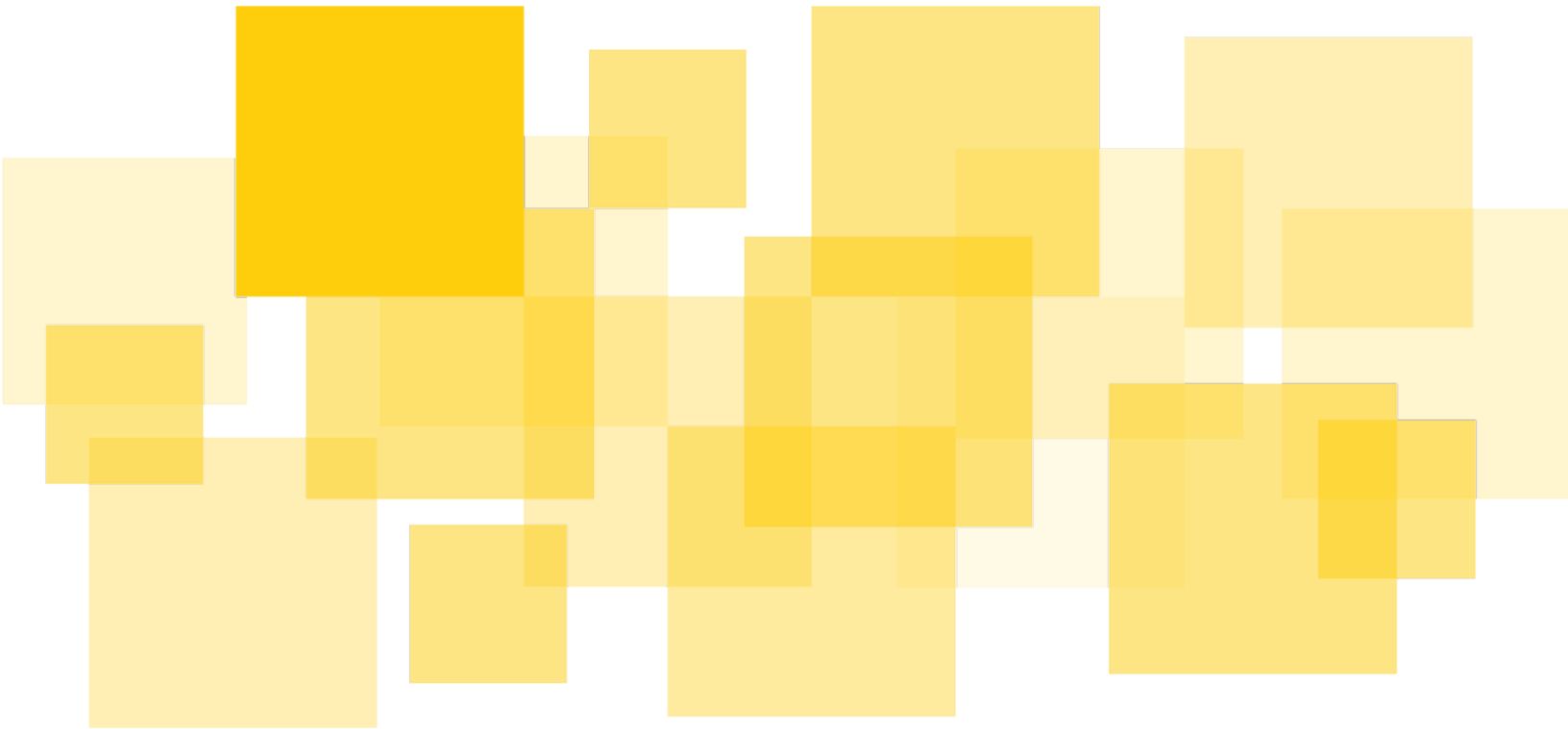


Universal Truth Framework

Runtime Verification Vision (2023-2025)

May 2023



Universal Truth Framework

— Runtime Verification Vision (2023-2025) —

Grigore Rosu — RV & UIUC

This is a long article discussing our company's overall vision for the next 2-3 years. This includes our recent Proof Chain proposal, but also other RV products that we plan to develop and launch. If you are mainly interested in the Proof Chain, then you may want to start with its [litepaper](#) or with the [video](#) in which I present it. Then you can come back here for more details on how the RV infrastructure and products will complement and service the Proof Chain.

Runtime Verification, Inc. (RV)

[Runtime Verification, Inc. \(RV\)](#) is a technology company, with focus on proof generation and verification. Its core technology, the [K framework](#), enables general purpose, versatile tools and solutions for secure, scalable and efficient products and infrastructures.

We are likely known as one of the major blockchain security auditing companies, with emphasis on formal specification and verification of protocols and smart contracts. The first who [formalized the ERC20](#) token standard, the first who formally verified or even audited [Uniswap](#). The first who formalized and verified [Ethereum's proof of stake](#) protocol and the [Ethereum 2 deposit contract](#). These among [many other examples](#).

However, if you think of us, RV, as the best security auditors, who use their own formal verification technology, K, which many say is the best programming language semantics framework, invented and perfected by ourselves, then ... Well, you would be correct. But you would not think big enough. We are way more than that. Security auditing and formal verification are only a fraction of what we are. We are here to revolutionize the very concept of trust in machines, trust in computers, trust in programs, trust in blockchains. Both in terms of what trust means, and how to achieve it. This goes way beyond formal verification, as you'll see in this article presenting our mission and vision.

Who I am

A few words about myself. I am [Grigore Rosu](#), the founder and CEO of RV, also a professor of computer science at the University of Illinois Urbana-Champaign (UIUC). Previously, I was a research scientist at NASA Ames, California. My specialty is programming languages, with emphasis on formal methods, formal verification, and automated reasoning. The K framework took birth in my [Formal Systems Laboratory](#) at UIUC, back in 2003. I coined the term [Runtime Verification](#) back in 2001, together with my NASA colleagues, as the name of a symposium that is now an annual international conference. [Runtime Verification](#) has also become a scientific field of study in itself, with an increasingly broader scope and community.

Why Writing This

I am writing this article in early 2023, to remind our existing RV team why we are all here. But also, importantly, to explain what we do and what we aim at as a company, to our potential new employees and investors who may have not got a chance to dive deep into our technology and research. What I present here is the result of more than 20 years of persistent work, which has been disseminated through peer reviewed publications in top scientific avenues ([Google Scholar](#), [DBLP](#)). You can find all the relevant publications, as well as detailed explanations and videos on our [RV Research](#), [RV Mission and Vision](#), [RV Publications](#), [RV Presentations](#), and [RV Videos](#) webpages. Here, I keep things high level for brevity.

Terminology, Notations, Abbreviations

This article refers to concepts in both mathematical logic and cryptography. There is, unfortunately, some degree of terminology overloading and confusion at the intersection between these two domains, especially surrounding the term “proof”, which plays a central role in our setting. “Proof” has a clear meaning in mathematical logic, and also a clear but completely different meaning in cryptography. Similarly, both “proof checker” and “proof system” are well-established notions in each of the two domains, but with different meanings in each. Consequently, we need to clarify these terms in order to avoid confusion. For the sake of clarity, in this section I establish the terminology, notations and abbreviations that I will strive to consistently use throughout the article.

Mathematical (or Matching Logic) Proof = proof in the usual mathematical sense in some logical formalism of choice – in this article, that logical formalism is **Matching Logic**. A proof is a sequence of steps that are either **Axioms** in a **Theory**, or deduction / inference steps (e.g., **modus ponens**) using **Rules of Inference**, or **Proof Rules**, that are part of a **Proof System**, ending with a statement which is hereby proved. Such statements which are proved this way are called **Theorems**. When the proof of a theorem is formalized as a certificate of correctness of that theorem that can be passed to other parties or to programs, we call it a **Proof Object**. Computer programs which help us search for proofs of theorems are called **Provers**, or **Theorem Provers**, and those which check proof objects for correctness are called **Verifiers**, or **Checkers**, or **Proof Checkers**. Proof objects tend to be very large in practice, because they contain all the proof details, so that the proof checkers can stay very simple, to be trusted.

Claim = a statement that is mathematically provable, that is, a theorem. We chose to call it “claim” instead of “theorem” to make this paper more accessible to non-experts. Indeed, most readers may not immediately realize that everything that is computable, that is, everything that a computer or a machine does when following some prescribed rules (a program) is in fact a theorem, in the theory defining the execution environment (programming language, virtual machine, machine language, etc), and that the actual computation or execution itself is in fact a proof of the theorem. However, even experts may still think more naturally of “foo(17) returns 42” as a “claim” instead of a “theorem”.

Matching Logic = logical formalism of choice in this project, a **second-order logic** fragment including **first-order logic** and a **least-fixed point** construct denoted μ (useful for iteration,

recursion, induction, etc.), which was created to be minimal but expressive enough to capture both program execution and formal verification as theorem proofs.

Zero Knowledge Proof = proof of knowledge, which is usually under the form of a succinct cryptographic artifact, here called a **ZK Certificate**, attesting knowledge that some statement is true. We try to avoid the ZK dedicated terminology of “ZK Proof”, and use instead **ZK Certificate**, to avoid terminology confusion with mathematical proofs. In this cryptographic context of zero knowledge, a **Proof System** is a protocol between a **Prover** and a **Verifier**, where the prover’s goal is to convince the verifier that a statement is true. We will prepend all these cryptographic concepts with **ZK** or, sometimes, **Cryptographic**, to avoid confusion with the homonymous mathematical concepts, i.e, **ZK Proof / Certificate**, **Cryptographic Proof / Certificate**, **ZK Proof System**, **ZK Prover**, and **ZK Verifier**. If we don’t use Cryptographic or ZK with any of these, then we mean the version coming from mathematical logic.

Zero Knowledge Technology = generic umbrella for techniques, methods, algorithms, protocols, circuits, etc., that were developed in the context of zero knowledge research. These include SNARKS, STARKs, zkSNARKs, zkSTARKs, etc., recursive or not. We take the freedom to informally say that a program was ZK-ed, or SNARK-ed, etc., when it was modified into a circuit capable of producing ZK certificates.

ZK(-based) Proof Checker = ZK-ed Matching Logic Proof Checker. In our project, we need only one program to be ZK-ed, namely the (Matching Logic) Proof Checker. Thus, we take the liberty to call it the **Proof Checker** when we are in the mathematical, logical setting, and the **ZK Proof Checker** when we talk about its ZK-ed variant, that is, its implementation as a ZK circuit. The ZK Proof Checker consists of two components: the **ZK Certificate Generator**, which is the component that produces the ZK Certificate that a given proof object has been checked by the Proof Checker, and the **ZK Certificate Checker**, which checks the ZK Certificate produced by the ZK certificate generator. We deliberately avoid the dedicated terminology in the ZK domain, to avoid confusion. For example, we say ZK Certificate Generator instead of ZK Prover, etc.

Proof of Proof = ZK proof of mathematical proof. We sometimes use this terminology to succinctly convey the essence of our approach: the ZK proof checker produces a ZK proof/certificate of knowledge that a mathematical proof for the claim has been checked.

We try to not use mathematical notation and greek letters in this article, but sometimes they are unavoidable. When that happens, we use \mathcal{T} to range over theories, φ and ψ to range over claims or theorems, Π to range over mathematical proofs, and π to range over ZK certificates. These may have intuitive subscripts or superscripts.

UTF = Universal Truth Framework

ZK = Zero Knowledge

SNARK = Succinct Non-Interactive Argument of Knowledge

STARK = Scalable Transparent Argument of Knowledge

DSL = Domain Specific Language

PL = Programming Language

VM = Virtual Machine

EVM = Ethereum Virtual Machine

FOL = First-Order Logic

RV = Runtime Verification

AI = Artificial Intelligence

GPT = Generative Pre-trained Transformer. Instances include ChatGPT, GPT-4, etc.

K = K Framework

MEV = Maximum (or Miner) Extracting Value

CFG = Control Flow Graph

SMT = Satisfiability Modulo Theories

Universal Truth Framework — What?

Let me start with what we mean by the Universal Truth Framework (UTF), what it is. Well, it is a framework in which every claim that is made, by this framework, is verifiably true. Specifically, claims come with independent, succinct, third party checkable proof certificates. A claim in UTF is anything that is computable or provable. In particular the execution of a program, some work that has been done or an action that has been performed, the formal correctness or security of some code, or any mathematical theorem. And indeed, in our framework that we present here, all the above can be formalized as mathematical theorems. Let's take a look at this picture:



Any true claim φ will have a succinct (cryptographic) proof π_φ . Anybody can check the proof π_φ mechanically and efficiently, practically instantaneously, and thus they will know that φ is true. UTF will facilitate producers of claims to also produce proofs for their claims, and consumers of claims to assess the validity of said claims by checking the proofs that come with those claims.

Universal Truth Framework — Why?

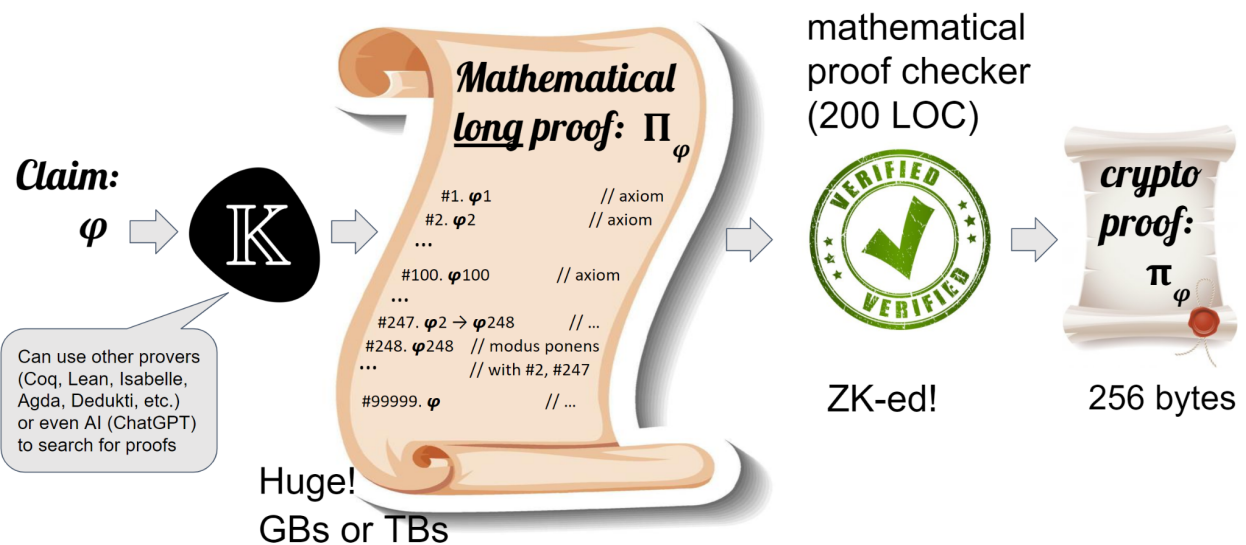
Recent advances in formal proof generation and verification have led to fragmentation, to a point where instances of essentially the same main idea appear and are viewed as different challenges, using different instruments to solve. We propose the UTF as a unifying framework, purposely very general in scope, yet achievable, encompassing everything that can be expressed as a mathematical statement that admits a proof in a mathematical theory. This includes the entire field of computing, as well as that of formal methods, semantics and verification. The UTF thus captures several major existing approaches, but will have many applications beyond those. For example:

- [Verifiable Computing](#) would now work for all programming languages. Basically, you execute your code securely in untrusted environments, say in the cloud, get back a proof certificate, verify it. Now you know your computation was correct.

- [Zero Knowledge](#) capability available for all languages, correct by construction! You can have, for example, zkEVM, or Cairo, or zkVM of RiskZero, or zkLLVM of the Nul Foundation, etc. All of these simply because semantically correct program execution is a claim in the UTF, in the corresponding language theory.
- Formal verification claims, correctness claims, security audits, as well as other program analysis claims, become checkable certificates (vs. PDF reports). So you don't have to trust the developers of the smart contracts, or the auditors of those smart contracts, or anybody else! You simply check the claim certificate.
- Critical procedures or devices in hospitals, aviation, automotive, robotics, etc., yield checkable certificates for their correct application. This will increase our confidence in complex systems, in complex processes, in machines, even in AI, because we don't have to trust them, we check their claim certificates.

Universal Truth Framework — How?

Our proposal for the UTF is to combine proof generators, like K, with ZK technology:



You start with the claim φ and you pass it, generically speaking, to the K framework. Think of the K framework as a searcher in a huge space of possibilities for a mathematical proof of your claim. Your claim can be that this program execution is correct, or that this protocol is correct, or anything that can be formally stated. Then K, with its suite of tools, will search for a mathematical proof for your claim. K can use many other tools as helpers. For example, K has backends for Coq, for Lean, for Dedukti. All these help you to search for a mathematical proof for your claim φ . You can even use AI, like chatGPT, to find helping lemmas and invariants. You can truly think of K as a mighty searcher of a mathematically rigorous proof Π_φ of your claim φ . We may, interchangeably, call the mathematically rigorous proofs Π_φ *proof objects*.

The problem with mathematically rigorous proofs, or proof objects, is that they can be very long. When we go down all the way to the axioms of mathematics and the basic logic reasoning,

mathematical proofs can be huge. Nevertheless, they are the ultimate correctness certificates for claims. Unfortunately, their size is a major problem for us, both in terms of space and checking time, because we want to ship them with the claims they certify and the consumers of the said claims to check them.

Can we reduce the size of proofs significantly, maybe even to a constant size, in a way that makes their checking, or verification, very fast, maybe even constant?

Yes, we can. With a trick that requires two steps.

First, instead of shipping the huge mathematical proof, Π_φ , check it *locally*, with a proof checker! The advantage of having a very detailed mathematical proof is that you can rigorously check it with a very small and simple proof checker. The proof checker will check every single step in the large mathematical proof. Moreover, proof generation and proof checking can be piped, so the proof is checked as it is being generated, and make use of massively parallel architectures, because proof checking is embarrassingly parallel: the entire proof is correct if and only if each step is correct, a perfect application for map-reduce architectures. We implemented a naive proof checker for the logic underlying the K framework, Matching Logic, which makes no use of piping or parallelism, but which was surprisingly small: 200 lines of code! I'll get back to this later, but for now just think about its impact and consequences: with only 200 lines of code, we can check any claim ... made by any program ... in any programming language!

Second, implement this proof checker as a zero knowledge (ZK) circuit! That is, make it produce a cryptographic certificate, or a cryptographic proof, π_φ , that a mathematical proof (Π_φ) for the public claim φ has been presented to it and checked and passed.

Therefore, our proposal as a first solution to implement the UTF is to use K and its arsenal of tools and backends (including Coq and Lean) to produce a matching logic proof (Π_φ), followed by a (small) ZK-ed proof checker to yield a succinct ZK proof (π_φ). We call it *Proof of Proof*: ZK proof of mathematical proof. Importantly, note that now you don't have to trust K or any tool that can generate mathematical proofs! You only have to trust the proof checker, which is public, small and the same for all languages and claims (execution, formal verification, etc.).

Of course, users also have to trust that their claim φ is *indeed* what they mean. This goes beyond our goal with the UTF, but we are well aware of this unavoidable challenge, which touches upon the philosophical nature of rigor and truth; we refer the reader interested in philosophical arguments on this topic to [this beautiful Pollak paper](#). Our pragmatic view is that the intended meaning of the claim is the responsibility and liability of its owner and stays outside of the UTF. In practice, claims will be generated automatically by tools which are validated by intensive usage in various applications.

Application of UTF — Proof Chain

The applications of the UTF that I mentioned previously are natural, not surprising, once we assimilate how Proof of Proof works. Indeed, it will be pretty clear that we can have verifiable computing, as well as ZK variants for all programming languages for which we have formal semantics. It will also be pretty clear that we can generate, from formal security audits, ZK

correctness proof certificates. But we would like to propose an application that goes beyond the state-of-the-art in blockchain. Way beyond.

Let me first highlight some of the limitations of the current blockchains:

- *Duplication of computation.* Indeed, all the validator nodes re-execute the same code, the same smart contracts (programs). That's a waste!
- *Hardwired VMs.* Indeed, validator nodes come equipped with predefined virtual machine languages, like EVM, Cairo, Move VM, etc., for all programs. Why!?
- *Extrinsic correctness.* Indeed, the security or correctness or formal verification arguments, why a certain program on the blockchain is correct, are external activities, done off-chain via uncheckable documentation (PDF files). Why!?

The UTF will enable a new generation of blockchain, which we call the *Proof Chain*, that will allow arbitrary claims, like execution, or correctness, or literally any truth, to be made, checked and stored. We will be able to write smart contracts in any programming or specification language for which we have a formal semantics. We will be able to execute any code/transaction once and for all, locally, and then broadcast its proof of proof (ZK) certificate to the validators. Every single claim made this way will be backed by a mathematical proof which will be made succinct, as a cryptographic proof.

The Proof Chain will require *no virtual machines* to be run by validators. The VMs and all the program execution infrastructure will be off-chain, with the purpose to produce ZK certificates. The blockchain validators only check the ZK certificate and reach consensus, but will not re-execute the programs. VMs are complex and can have implementation bugs, especially when they incorporate ZK technology. Why take that risk!? Moreover, VMs complicate the blockchain network and infrastructure. Validator nodes with incorporated VMs need resources to re-execute all the transactions and require regular upgrades.

The mathematical proofs of the Proof Chain claims will be produced off-chain using a variety of available methods, ranging from tools like K, instrumented execution engines, theorem provers, proof assistants, or even manually or using advanced AI. We will not have to trust these methods, because they are ultimately just best effort searchers for mathematical proofs of claims. The mathematical proofs will be checked by trusted proof checkers, which will yield succinct cryptographic/ZK proofs for the Proof Chain.

If this future vibes with you, let us continue our journey by diving into specifics. The next few sections discuss the K framework, which was the inspiration for the UTF and the Proof Chain, as well as recent K-powered tools and envisioned products. If you are already familiar with K and its tools, or want to first see how the UTF puts together K, Matching Logic and ZK, you can skip to the [Matching Logic section](#). If you have a good feel for how the UTF works and are rather interested in seeing how it can be used to power the ultimate Layer Zero blockchain, fast forward to the [Proof Chain section](#).

What is K?

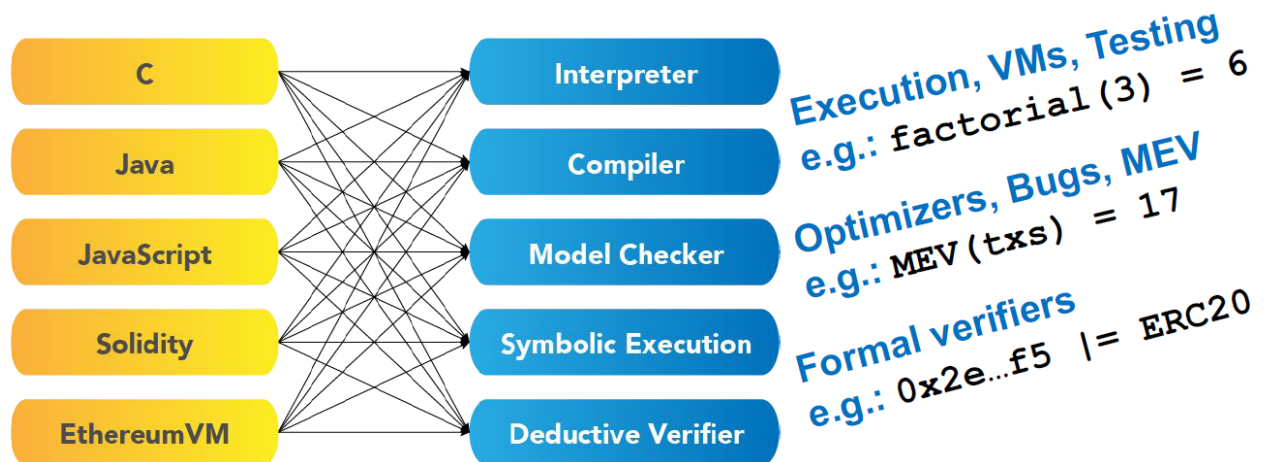
Let me set the scene first. We tell computers what to do using programming languages. There are many languages already and many of them are invented as we speak. Especially in new domains that propose novel means of computation, like the blockchain. Unless it is very new, each language usually comes with a suite of tools.

Interpreters and compilers are the most basic tools, because they allow us to execute programs, to implement virtual machines, and to test programs before deployment. For example, the `factorial(3)`, in Java, evaluates to 6. In applications where program correctness is paramount, like in mission or safety or security critical systems, conventional testing is not sufficient. We need deeper program analyses.

Broadly speaking, model checkers systematically analyze the space of behaviors of a program, up to specified boundaries or abstractions. They allow us to find corner cases, which can be bugs, or optimal solutions to specified constraints. For example, the maximum extracting value (MEV) of a set of transactions is some value M.

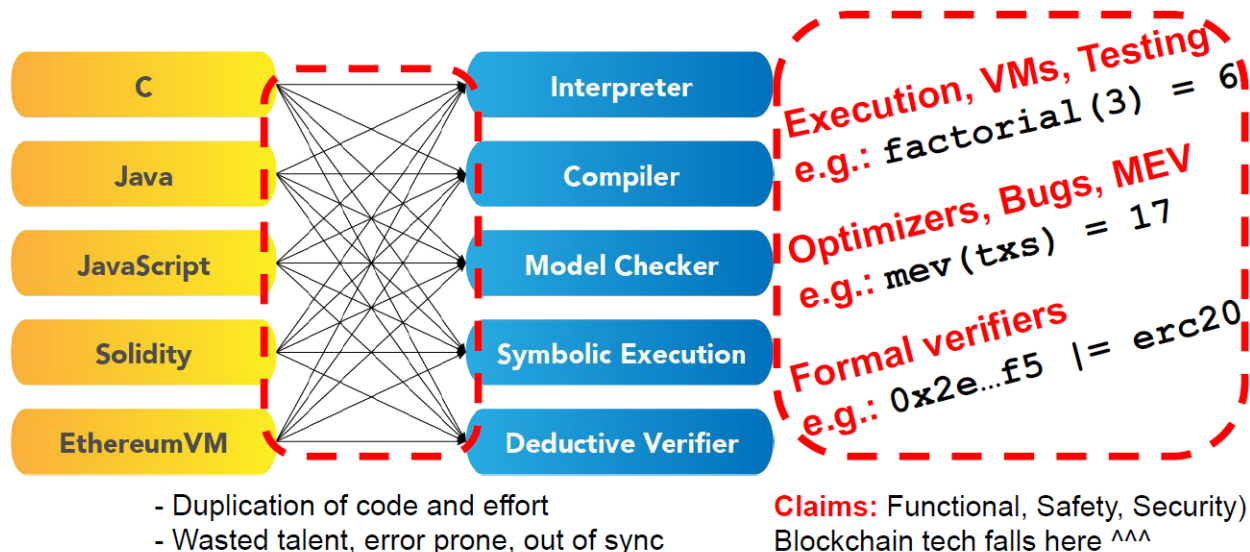
Formal verification tools give us the highest level of assurance, because they cover all the behaviors of a given program. You can think of them as exhaustive testing tools. To cover potentially very large or even infinite spaces of program behaviors or states, formal verifiers may make use of symbolic execution and logical deductive reasoning, to mathematically prove that programs satisfy their requirements. For example, to prove that the smart contract at some Ethereum address is a correct implementation of the ERC20 token specification.

Basically, each arrow in this picture



represents a certain tool, a whole system, for a certain language, which usually is complex, sometimes taking many years of work. For example, geth is an interpreter for EVM – the arrow from Ethereum VM to Interpreter. GCC is a compiler for C. Java Pathfinder is a model checker for Java. Certora is a deductive verifier for EVM. And so on and so forth. Some tools aim at implementing two or more of the above capabilities. In short, we have lots of languages and lots of tools for them, which are used in critical infrastructure, applications and products.

The current state of affairs comes with major pain points. Both for developers (left red surrounded area in the picture below) and for users (right red area). That is, for all of us:



For developers, duplication of code and work is always a pain. Indeed, different tools for the same language require a parser, or a control flow graph (CFG) extractor, and each of these implements their own, usually by copying and adapting an existing code base. Worse, the same conceptual tool for two different languages, for example model checkers for C and for Java, are perceived and implemented as two completely different tools, most likely by two different teams of talented engineers or researchers, although they implement the same well understood algorithm or technique, just adapted to a different language. This is a waste of time, talent and effort. And let's not forget that programming languages evolve as well, from version 1 to version 2 to version 17, right?; C11, C17 ... More often than not, these tools are versions behind.

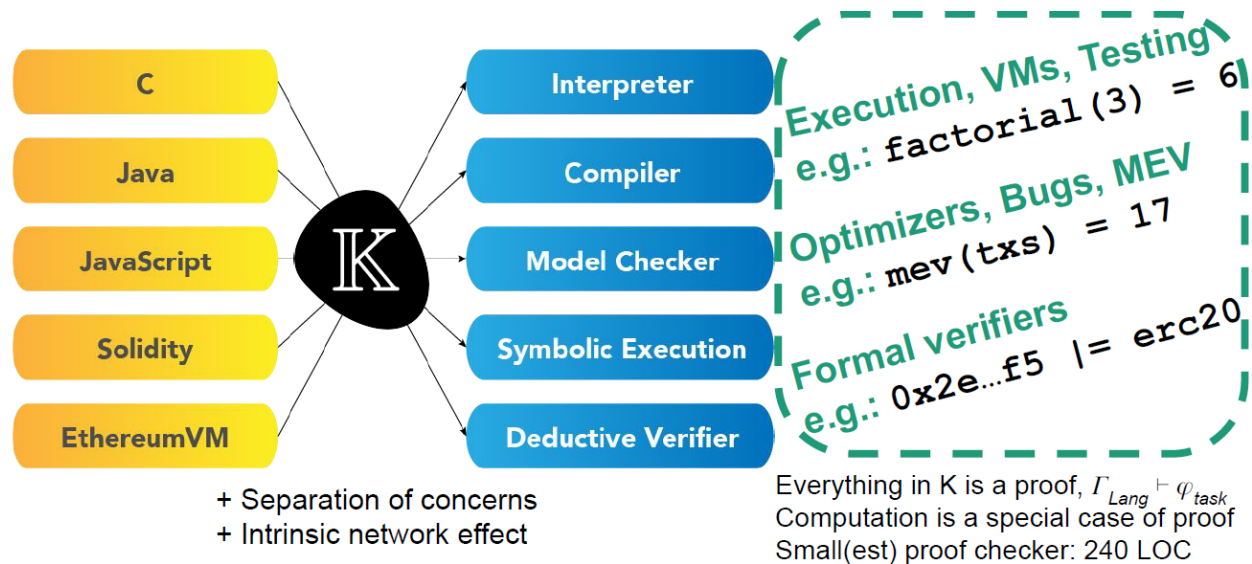
While developers' pain may seem serious enough, in fact it pales in comparison with its consequences for the rest of us, the users of these tools. For example, why should we trust that a program executed correctly; for example, that the factorial(3) correctly evaluated to 6, in some given implementation of Java, or C, or Solidity? After all, there are hundreds of thousands of lines of code implementing the compilers and interpreters of these languages. The situation is even worse for tools meant to ensure correctness, such as model checkers or deductive verifiers. Why should we trust a tool claiming that the contract at address .0x2e...f5 correctly implements the ERC20 token specification?

All the tools for all the languages, in the end, make claims that we are forced to accept. Claims of execution, claims of optimality, claims of correctness. We take risks on a daily basis by trusting all these claims. Risks that sometimes result in loss of money, or of expensive machinery, or even of human life ([Wormhole](#), [Ariane-V88](#), [Therac-25](#)).

Is there any alternative, though? Would it be possible to have a universal framework, where any claim made by any tool, or any programming language, can be independently checked with a unique, universal claim checker, ideally one which is so simple that it can be implemented in a matter of hours, if one does not want to trust existing implementations? Yes, it is. The

[K Framework](#). We proposed it 20 years ago and perfected it ever since. We are now on a mission to take it to the next level. To disrupt the state of the art and to change the way we think about Computing and Trust.

This is how K addresses the two pain points, of developers and of users:



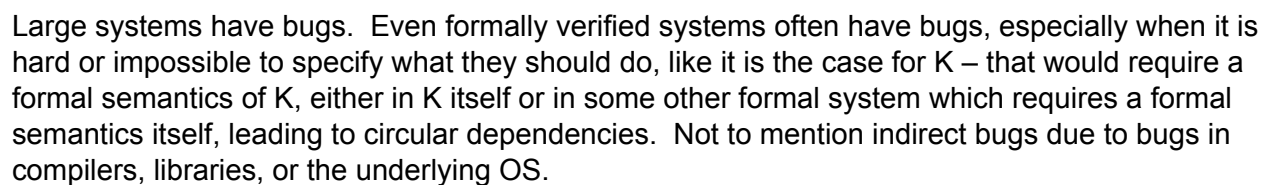
On the developers' side, K allows and provides a programming language for languages – a domain specific language to implement programming languages, or to define programming languages. We call these programming language definitions, the *formal semantics* of the respective languages. The various tools that the K framework provides, like interpreters, model checkers, symbolic execution engines, deductive verifiers, etc., are implemented in a generic, language agnostic way. In other words, you define a PL, plug it into the K framework, and then pick a tool, say symbolic execution, and you've got a symbolic execution engine for your PL.

This separation of concerns has major implications in terms of increasing network effect, because you implement a language once and for all, or define a language once and for all, and then you get all the tools for that language. You improve the language, say go from Java 1.4 to Java 5, and now you've got all the tools already upgraded automatically for your new version of the language. And also, similarly, you are very motivated now to fix bugs or add optimizations, improvements to the tools that are language- parametric. For example, if you make the symbolic execution engine faster in K, then you get a faster symbolic execution engine for all languages. But importantly, you don't have to maintain $L \times T$ different systems, where L is the number of languages and T is the number of tools. Instead, you only have to maintain the languages themselves plus the tools themselves: so $L + T$ instead of $L \times T$. This reduces the complexity enormously and makes tool development more convenient.

For users, on the other hand, everything that the K tools do when instantiated with a programming language, basically any claim that any K tool makes, is a theorem. A theorem which is valid in a mathematical theory, namely the formal semantics of the instance programming language. The K tools have the capability to search for and produce a mathematical proof for the claim. I want to re-emphasize that computation is a particular case of

We have a [small proof checker](#), of only 200 lines of code, which can check any of the proofs done by any of these K tools, or any other tool that can produce mathematical proofs. Although K was invented 20 years ago and the first ideas underlying it derived from work on similar topics done when I was a NASA research scientist (2000-2001), the proof checker was implemented, evaluated and published in its current minimal form only relatively recently, in [CAV'21](#) and [OOPSLA'23](#). The reason for this late development is that the underlying logical foundation of the proof checker, matching logic, which we proposed in 2008 during a sabbatical at Microsoft Research, required more than 10 years of work and it was finalized only in 2019, with the addition of support for the μ construct for least-fixed points ([LICS'19](#)). More about this shortly.

Considering how much it does, it is not surprising that the K framework is rather large and complex. It has more than 500,000 lines of code in four different languages. K has always been open source (Github, MIT license). It is likely the most complex formal methods system. To give you a glimpse at its complexity, the diagram below shows the architecture of its tools and their dependencies that we use in our internal development:



have bugs. Fortunately, this is no more than an inconvenience, one which cannot lead to invalidating any final claim made with any of its tools. Indeed, thanks to K's capability to generate proof objects, which can be independently (of K) checked with a small and trusted proof checker, we don't need to trust any of K's tools. This approach is called [Translation Validation](#) in the literature: do not validate the translator itself, but each translation instance done with the translator, separately.

Therefore, our philosophy is that you don't need to trust complexity, you simply check it.

To summarize the discussion so far, the K framework allows you to rigorously define a PL through its formal semantics, once and for all, and then “plug” it in the framework and “play” it, meaning that you can derive all the tools that you need for your PL from its formal semantics. These tools are capable of producing mathematical proofs of everything they do, and everything they do can be stated as a mathematical theorem. At least that was and continues to be the founding principle of K. In reality, the instrumentation of the tools to produce proof objects is a relatively new development, started two years ago. Not all the tools have been instrumented to work with all the languages yet. There's still engineering work needed, which is in progress. But we will get there. And when we get there, everything K does will result in a proof object that can be checked by a proof checker, which I'm going to talk about later.

Next I'm going to talk a bit about what's new in K, what happened in the last 18 months or so after our first raise. There are many new recent developments in K, which I cannot mention here. I'm only going to focus on a few of them.

New: K Summarizer

One of the most important recent K infrastructure developments is the K Summarizer. It basically generates a control flow graph (CFG) of all the behaviors of a given program. That's very similar to what a compiler does, except that it is completely driven by the PL semantics! And completely automatically! And correct by construction!

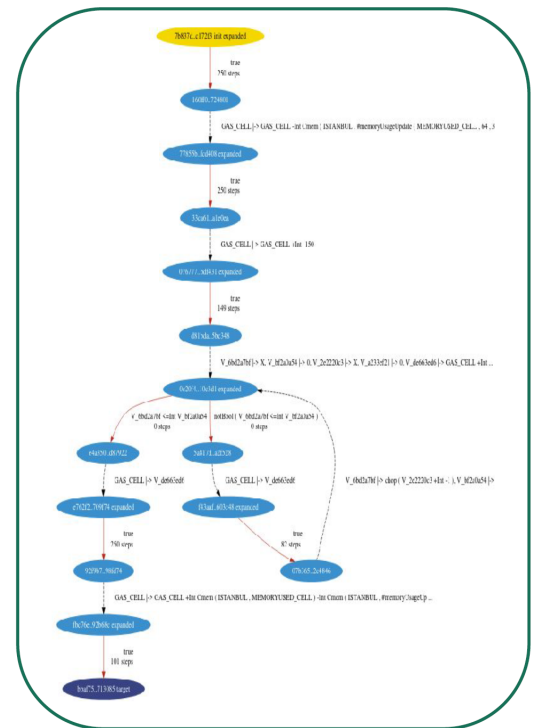
Let me illustrate the K summarizer with an example. Let's consider the [semantics of EVM](#) in K, KEVM (see also <https://jellopaper.org/>, a human readable EVM semantics generated automatically from the KEVM semantics), and the following code fragment in Solidity, namely a smart contract that calculates the sum of numbers from 1 to n.

```
contract SumContract {  
  
    function sum(int n) external pure returns (int s) {  
        s = 1;  
        int i = n;  
        while (i > 1) {  
            s += i;  
            i -= 1;  
        }  
    }  
}
```


This code is only for illustrative purposes, it is not meant to be a practical smart contract. The program takes an input n , then assigns the sum s , which is the return value of the contract as well, to 1 and then goes through a loop and adds all the numbers in some optimal way, in the sense that it only adds the numbers larger than or equal to 2, because it starts with s equals 1. Well, as a matter of fact, this program has a small, intended bug: it must be called with n larger than or equal to 1 in order to output the correct sum: if n equals 0, the program returns 1 instead of 0, which is incorrect. But for any n larger than or equal to 1, the function `sum` works correctly.

The K Summarizer takes a programming language semantics as input, here the EVM semantics, and a program in said language, here the `sum` program above, which we compiled first to get its EVM code. From these two inputs, the programming language semantics and the program, the K Summarizer generates a graph like this one to the right. This graph looks very much like the program, in the sense that there is some code that initializes things, the s equals 1 and i equals n before the loop, then the loop takes place, and then there is the exit from the program.

What's different here from what conventional compilers do, is that every edge in this control flow graph (CFG) is actually a theorem, a reachability claim saying that if you are in a state that matches the source pattern of the arrow then, if the conditions on the arrow hold, you reach a state that matches the target pattern of the arrow. Note that sometimes there are multiple arrows from a node, so there are conditions splitting the potential behaviors. In other words, this K tool summarizes all the behaviors of the program automatically, driven by the programming language semantics and the program only.



The K Summarizer is now at the core of several other K tools, like efficient interpreters, compilers, program verifiers, model checkers, and symbolic execution engines. Once you summarize the program, you can use the resulting summary in these tools instead of the program itself, to achieve the analyses more efficiently. Intuitively, the summarizer computes the accumulated semantics of all the instructions in each of the basic blocks of the program. That is, each basic block is semantically summarized into an arrow in the resulting CFG summary. You can think of these arrows like “big steps”, each comprising all the small steps of the block. Because it is correct by construction according to the language semantics, we can actually associate a mathematical proof to each arrow. This will allow us to generate proof objects, eventually, from everything that our tools do, which is necessary to implement our UTF.

Importantly, the CFG produced by the K Summarizer is a K definition itself, where each arrow in the CFG is a rewrite rule. A good way to think of it is as the formal semantics of the summarized program. Or as a domain-specific language (DSL), defined in K, which is very specific to the summarized program; so specific, that it only knows about that one program and some necessary fragments of it (its basic blocks). Another way to look at it, is as a way to optimally inline the PL semantics within the program itself, until there is nothing left from the PL

except what matters for the one program that is summarized. This is similar to what compilers do, but what is different here is that this process is entirely driven by formal semantics of languages and mathematical proofs.

Since the output of the K Summarizer is an ordinary K definition, we can use it as input to any of the tools offered by the K framework. In particular, as input to the LLVM backend of K, which is normally used to generate interpreters from language semantics. When we do that, we get an interpreter specialized for one program — this is, again, similar to what compilers do: for example, a C program is compiled to machine code, which is then interpreted by the processor. This effectively gives us an *EVM compiler*, automatically derived from the formal semantics of EVM, correct by construction!

The above raises the following question: how much faster are the EVM programs compiled using the K Summarizer? Our experiments so far show a performance boost close to 100x compared to interpreted EVM, which is not surprising: usual performance rates between compiled and interpreted code is between 10x (Java) and 1000x (LLVM). This means that we can use our K Summarizer to execute EVM smart contracts two orders of magnitude faster than Ethereum nodes. This further means that an MEV engine can spend 100x more time searching for optimal transaction orderings than its competitors. As explained in our [monitoring & recovery section](#), we plan to incorporate our EVM compiler obtained using the K Summarizer in our Ethereum Keeper products.

The K Summarizer was not available six months ago. In our view, it is a game changer. Not only in the overall design and implementation of the K framework, but also in the scientific community. Generating compilers correct by construction from language semantics is the Holy Grail of the programming language and semantics field. For the academically inclined reader, broadly speaking this work falls under the area called [partial evaluation](#), and more specifically under the subarea of “partial evaluation of interpreters”, aka [Futurama Projections](#). There, the idea is that we can regard an interpreter as a function taking a program as first input and an input to the program as second input; then through partial evaluation of that interpreter function in its first argument, the program, you get another function, which now behaves as the compiled program: it takes the input of the program as input and produces a result. However, little to no progress has been made in more than 50 years of research. Not until now. Not until the K Summarizer. Why? We believe that’s because the community looked at the problem from the functional programming angle instead of from the language semantics angle, that is, as specializing a *function* instead of a *semantics*. The semantics is more general, because the interpreter can be generated from it, as we do with K. Sometimes solving the more general problem is easier and gives more clarity.

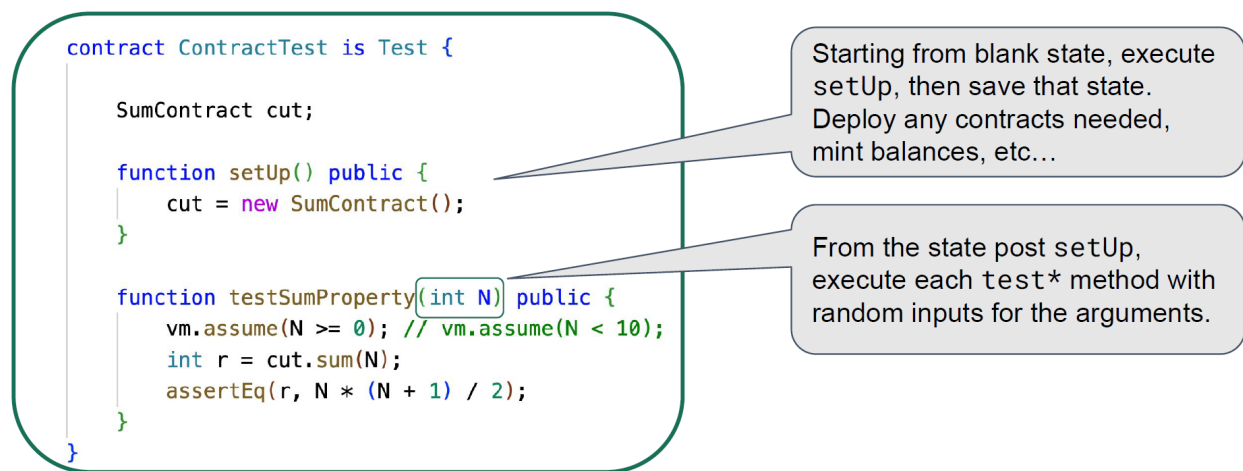
Just to be clear, I’m not saying that we have completely solved the semantics-based compilation problem, for all languages. But we have made significant progress, that it works for some languages, like EVM, and that we are not far from a general solution.

New: KEVM-Foundry

Another important new development is the integration of KEVM with Foundry, called KEVM-Foundry in this document (we are considering a dedicated, better name), essentially

giving Foundry access to the EVM semantics in K by means of cheat codes.

[Foundry](#) is an increasingly popular parametric property testing framework for Solidity. It is like [JUnit](#) (for Java), but for Solidity. Parametric property testing frameworks usually ask the user to provide some setup code that initializes the state, together with a set of (property) tests that exercise various program behaviors against that state. Let us illustrate how Foundry works using our example sum contract from the previous section:



The `setUp()` function here only creates an instance of the sum contract, the contract under test (`cut`). The names of the property tests are currently required to start with “test”. We have only one property here, that checks the closed form solution of the sum. Specifically, it takes a symbolic input `N`, then assumes `N` is larger than or equal to zero (intended mistake, for demonstration only — recall that `sum(n)` works correctly only when `n` is larger than or equal to 1), then makes a call with input `N` to the `sum` function under test, then it asserts that the function call returns the correct closed form solution.

Note how Foundry extended the language with new instructions, like `vm.assume(...)`. These so-called “cheat codes” are implemented as hooks into the VM and allow tools like those of Foundry to interact with the state of the VM, in particular to customize it or extend it according to the specific tool needs. The default Foundry tool, “forge”, fuzzies the property parameters in order to obtain a potentially very large set of concrete unit tests, in the hope that some of them may expose corner case bugs. Specifically, it generates lots of concrete random inputs for the property parameters that satisfy the assumptions, then executes the code with those concrete inputs; in particular, all the `assert` commands must hold in order for the test to pass. Each concrete test is executed in the same initial state generated by the `setup` function; that is, the states obtained after executing each test instance are irrelevant and thus discarded.

Foundry’s fuzzer is fast, but it only covers a finite input set. In our example, remember that our code was incorrect when the input `n` of `sum` equals 0. So you may expect Foundry to find a counter-example with this property test when `N` is 0. Unfortunately, the fuzzer cannot find this issue (as of March 2023), generating and trying concrete inputs `N`, different from 0, for as long as we let it run. The input space is infinite and it just happens that the fuzzer doesn’t guess the case when the input `N` equals 0; this may change in the future, with improved versions of the

fuzzer. Nevertheless, if we bound the input to a certain range, say if we also assume N smaller than 10, then Foundry will exhaustively explore the finite input space and it will find the issue.

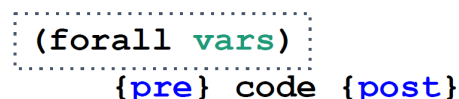
Foundry is a tool that Solidity developers use regularly these days. Many of our RV clients are already familiar with Foundry. If not, we teach them, because we believe Foundry is currently the best testing framework for Solidity code.

Our KEVM-Foundry tool incorporates KEVM semantics within Foundry. KEVM-Foundry takes the same tool input as Foundry, so the same property specifications and the same assumptions and the same assertions, but instead of fuzzy testing, it does symbolic execution. So instead of instantiating the property test parameters with random concrete values, it leaves them symbolic and executes the code using the KEVM semantics under the hood. This way, *the parametric tests become formal specifications* and the KEVM-Foundry tool formally verifies them. For the example above, the KEVM-Foundry tool reports an error, as expected, because the property does not hold for $N=0$. Currently, KEVM-Foundry requires manual intervention to guide the proofs, but the user is rewarded with an error found. We plan to translate these detected errors into concrete test inputs, and thus to have KEVM-Foundry generate test inputs, same like Foundry's forge command, but only inputs that break the property tests! When no such input can be generated anymore, to generate a proof object that attests that the property tests are correct, when regarded as specifications.

In other words, *KEVM-Foundry is a formal verification tool for EVM*. This is an elegant way to expose the power of the K framework to developers, without requiring them to learn K. Actually, the learning curve of KEVM-Foundry is effectively zero, once you already know Foundry. It is the same familiar user interface that Foundry offers and K is completely hidden under the hood.

I would like to make a theoretical argument here, namely that parametric property tests are quite expressive in terms of program specification and verification. Beginners to formal verification, or formal verification reductionists, may think or claim that formal verification is more powerful than parametric property testing, because with formal verification we can prove properties, while with parametric property testing we can only test properties. Well, it turns out that Hoare triples, which are the core of formal verification, actually can be faithfully expressed as parametric properties. I would like to explain this aspect well, because I think understanding this is important to understanding why the integration of K with tools like Foundry is so important.

In program verification, Hoare triples, or correctness triples, are of the form "{pre} code {post}" and these three entities – preconditions, code and postconditions – share some free variables. Since the free variables are important in our narrative here, we write Hoare triples as in the picture below, where “vars” are the free variables that occur in the precondition, the code, and the postcondition:



```
(forall vars)
{pre} code {post}
```

In our previous property of the sum of natural numbers, e.g., the *correct* Hoare triple should have the precondition $N > 0$ – remember for n equals 0 the code is incorrect –

```

forall N
{N > 0} r = sum(N) {r = N*(N+1)/2}

```

Notice that the same N is used in the precondition, in the code, and also in the postcondition. We say that N is a free variable, or a parameter, of the Hoare triple.

Hoare triples can be expressed as parametric property tests as follows, where the free variables in the Hoare triple become parameters in the corresponding property:

```

function testProperty(vars) {
    assume pre;
    code;
    assert post;
}

```

Therefore, we assume the precondition, execute the code, assert the post condition. In our sum of natural numbers example, our Hoare triple becomes the following Foundry parametric property test (written this time correctly, with assumption $N > 0$):

```

function testSumProperty(int N) public {
    vm.assume(N > 0);
    int r = cut.sum(N);
    assertEq(r, N * (N + 1) / 2);
}

```

Now KEVM-Foundry is able to formally verify this property, currently with a bit of manual help from the user (which we hope to soon eliminate, via automated invariant inference). Of course, Foundry's fuzzer runs for as long as we are willing to wait without finding any violations, because the property is valid, which is the same behavior shown by Foundry as with the previous property test, with $N \geq 0$, which was incorrect.

Therefore, parametric properties are expressive. They are theoretically equivalent to Hoare triples, so verifying parametric properties is equally powerful to conventional formal verification. Both approaches are language-specific, that is, unlike K, both Hoare logic and parametric properties are constructed for a specific language. In practice, however, language-specific formalisms are limited by design. For example, support for loop invariants, contract invariants, collection invariants (eg, the sum of all values in an ERC20 token map is the total supply), temporal logic constructs, etc., will have to be added to Foundry via cheat codes or other ways, if we want to naturally write certain properties that require them. This is unavoidable and, indeed, it has happened in the world of language-specific formal verification since the beginning.

An important advantage that a language framework like K (Coq, Lean, etc.) gives us is that it allows us to express complex properties that are harder or impossible to express using conventional Hoare logic or symbolic property testing. For example, we may want to require the precondition that the function `sum` can only be called when the stack size is smaller than 1000, and/or that it uses no more than 3 locations in the storage, or to also specify a closed form expression for its gas consumption. Fortunately, we have the full expressiveness of K to our disposal: at any given moment we can write the harder properties directly in K. The details of

how this is done is beyond our scope here, but the reader can contact us for details.

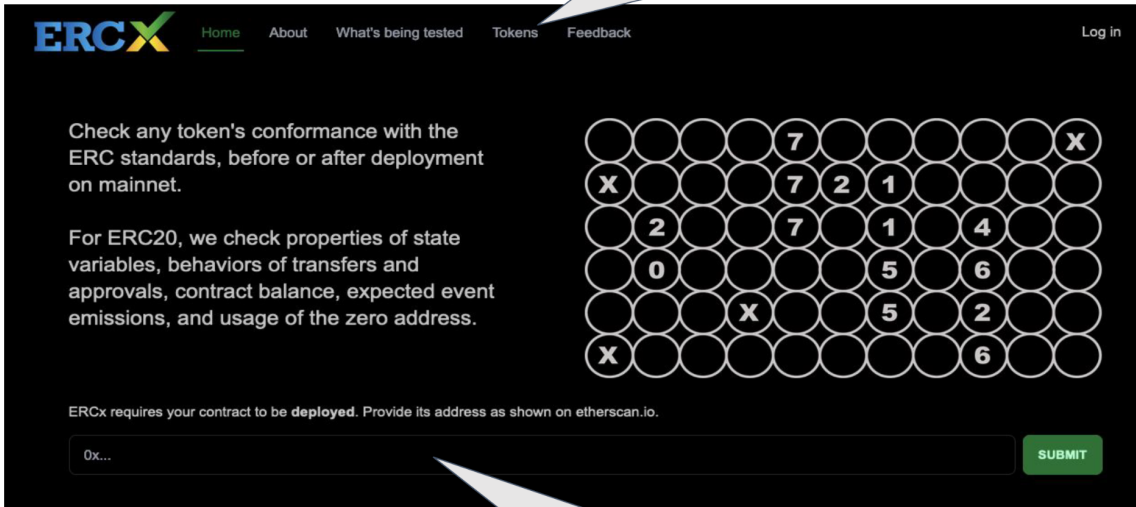
In conclusion, we found Foundry's parametric properties sufficiently expressive for our formal specification and verification needs, and KEVM-Foundry to be a practical and user-friendly frontend to the KEVM semantics, which hides the complexity, but also the power, of full K. We are in the process of developing similar property testing and formal verification tools for other languages, following the same pattern: define a new or use an existing K formal semantics of the target language, then build a new or adjust an existing parametric property testing framework to become a formal program verifier for the target PL based on semantics-driven symbolic execution. This will allow us to bring the success and practicality of Foundry and KEVM-Foundry to other blockchains. Current targeted efforts include Elrond and Algorand.

New: The ERCX Tool/Product

Another new development is the [ERCx](#) tool/product. This is an ERC compliance checker for token implementations, offered to the developers and users at large through an easy to use, automatic frontend web interface. All you have to do to use ERCx is to provide the Ethereum address of your token code:

ercx.runtimeverification.com

Deep dive investigation of ERC tokens deployed on mainnet.



Check any token's conformance with the ERC standards, before or after deployment on mainnet.

For ERC20, we check properties of state variables, behaviors of transfers and approvals, contract balance, expected event emissions, and usage of the zero address.

ERCx requires your contract to be **deployed**. Provide its address as shown on etherscan.io.

0x...

Submit your token, too. See how you stack up!

Completely Automatic!

ERCx is implemented as an instance of [KEVM-Foundry](#) with pre-defined sets of property tests, each corresponding to a specific ERC token or variant of it. Indeed, each ERC token standard, for example ERC20, is proposed by its authors to the community as an API specification, whose intended meaning is usually described informally, in words. We formalized the well-known but informal specification of ERC20 as a collection of parametric property tests, which as explained previously, in the presence of KEVM-Foundry, becomes a formal specification. We formalized not only the main ERC20 standard, but also its variants that are tacitly assumed "ERC20 tokens" by the community. For example, the ERC20 standard has no whitelists, no blacklists,

no transfer fees, etc., yet many variants have these. ERCx understands all these variants and tells you exactly in which category your token falls. We ran ERCx against all the ERC20 tokens on Ethereum (more than 200,000 of them) and it classified them in more than [20 categories](#). Specifications for other ERC token standards are currently being formalized by our team and will be available soon as part of ERCx.

Therefore, the idea underlying ERCx is that commonly used ERC token specifications, as well as variants of them, are already formalized and incorporated in the tool. That is, ERCx “knows” them. All the ERCx user has to do is to provide the Ethereum address of their token code. The tool will check the code against the hardwired specification and yield a report of compliance for that token. Go to <https://ercx.runtimeverification.com>, submit your code and see how you do. Note, importantly, that ERCx is completely automatic. As a user, you don't have to provide specifications, you don't have to provide invariants, you don't have to provide lemmas, you don't have to provide anything but your code. ERCx will check compliance automatically for you.

We intend to develop more tools of this kind on top of our language semantics. This is what the K framework has been designed for. We define the semantics of a PL once, and then all the tools that make use of that semantics are defined on top of it. The tools I showed so far were instantiating the K framework with the EVM semantics. But the K framework is generic. We can instantiate it with any programming language, even C.

New: RV-Match

[RV-Match](#) is one of the tools in our company where the K framework is instantiated with the C programming language. It is one of the oldest tools in RV. However, there are some new developments and improvements to RV-Match. Specifically, in the context of verifying the Firedancer Solana validator. This is a project in collaboration with Jump.

RV-Match, same like [KEVM-Foundry](#), builds upon K instantiated with a formal semantics of a PL, but hides all that complexity under the hood and minimizes the learning curve for users. Essentially, as a user, you run your C tests normally; however, instead of using GCC, or CLANG, or whatever compiler you use, you replace that compiler with the RV-Match corresponding tool, which is KCC. So instead of GCC you type KCC.

This is how it works, on a code snippet from Firedancer:

```
typedef unsigned short ushort;
static inline ushort fd_ushort_rotate_right(ushort x, int n) {
    n &= 15;
    return (ushort)((x >> n) | (x << (16 - n)));
}

int main(int argc, char **argv) {
    ushort i = fd_ushort_rotate_right(60164, 0);
    return 0;
}
```

firedancer snippet

If you compile it with GCC and run it, the code runs just fine. However, if you replace GCC with KCC, that is, if you compile with KCC, exactly the same parameters, the same command line, and then you run the resulting binary, you get an error:

Conventional compilers do not detect problem

```
$ gcc fd_ub_example.c -o gcc.out
$ ./gcc.out
$
$ kcc fd_ub_example.c -o kcc.out
$ ./kcc.out
fd_ub_example.c: In function `fd_ushort_rotate_right':
fd_ub_example.c:6:3: error: undefined behavior: result of signed left shift not representable in result type [-Wno-signed-left-shift-overflow]
   Refer to c18 §6.5.7/4 file:///src/.build/dist/lib/kcc/html/shifts.html
   called by fd_ub_example.c:10:14(main)
```

RV-Match's kcc tool precisely detects and reports error, and points to ISO C11 standard

The error above is an infamous “undefined behavior” and you get exact precise pointers to the C standard, explaining this kind of undefinedness. Basically, undefined behavior means that this program can behave completely differently on another platform. That's important to know in the case of blockchain validators, because you want the validators to run your validation code on different platforms, even on FPGAs, so it's very important to make sure that validators implemented in C are free of undefined behaviors.

We only discussed the KCC tool of RV-Match. Other tools, like a symbolic bounded model checker and a program deductive verifier for C, are also part of RV-Match, but they are still under development and harder to use, so we do not discuss them here. RV-Match is therefore a framework for C program analysis, an instance of the K framework with the formal semantics of C. It is the most comprehensive C formal semantics and, importantly, it is ISO compliant.

I certainly didn't do justice to all the new developments in the K framework in the last one year and a half. For example, I didn't say anything about the semantics of Tezos' Michelson, KMichelson, or Algorand's virtual machine semantics, KAVM, or the semantics of Cardano's Plutus, KPlutus, which basically all enabled tools like the ones we saw previously for all these blockchains. What I really wanted to illustrate with the previous few examples, is that, first, K matured enough to be used as the core infrastructure of very practical tools for the respective languages defined in K. And second, that K with all its complexity and power, can be hidden under the hood by such tools that have a user-friendly interface.

In other words, you have the full power of K at your service, but at the same time you don't need to access that complexity unless you really want to and you know what you are doing. And with this, we are ready to move to the next section, where we talk about ... why should you trust K?

Matching Logic — Foundation of K (and Coq, Lean, etc)

As explained in the [UTF – How?](#) section, simplistically speaking, UTF = K + ZK. That is, in order to provide claims with correctness certificates in the UTF, we first use the tools in the K framework to search for mathematical proofs of the claims and then compress those into succinct cryptographic certificates using a ZK circuit implementing a checker for the mathematical proofs. What makes the UTF possible in the first place is that everything K does for a particular PL instance is a proof of a theorem in the mathematical theory

corresponding to that PL formal semantics. We want to produce ZK certificates for these proofs. In order to do that, we need to first understand how we can generate such mathematical proof objects. In order to understand proof objects, we need to understand the logic underlying everything K does, which is Matching Logic.

[Matching Logic](#) is the foundation of K. And not only of K; it can also very well and it does serve as a foundation of languages like Coq, Lean, and other interactive theorem provers; more details are discussed shortly. It is the smallest logical foundation known for languages and formal verification, that has this expressiveness (monadic second-order logic). Its most general form, which includes support for least fixed points, was proposed in a [LICS'19 paper](#) and it is so small that you can write it on a napkin:

FOL Rules	(Propositional 1)	$\varphi \rightarrow (\psi \rightarrow \varphi)$
	(Propositional 2)	$(\varphi \rightarrow (\psi \rightarrow \theta)) \rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \theta))$
	(Propositional 3)	$((\varphi \rightarrow \perp) \rightarrow \perp) \rightarrow \varphi$
	(Modus Ponens)	$\frac{\varphi \quad \varphi \rightarrow \psi}{\psi}$
	(\exists -Quantifier)	$\varphi[y/x] \rightarrow \exists x. \varphi$
	(\exists -Generalization)	$\frac{\varphi \rightarrow \psi}{(\exists x. \varphi) \rightarrow \psi} \quad x \notin FV(\psi)$
Frame Rules	(Propagation $_{\perp}$)	$C[\perp] \rightarrow \perp$
	(Propagation $_{\vee}$)	$C[\varphi \vee \psi] \rightarrow C[\varphi] \vee C[\psi]$
	(Propagation $_{\exists}$)	$C[\exists x. \varphi] \rightarrow \exists x. C[\varphi] \text{ with } x \notin FV(C)$
	(Framing)	$\frac{\varphi \rightarrow \psi}{C[\varphi] \rightarrow C[\psi]}$
Fixpoint Rules	(Substitution)	$\frac{\varphi}{\varphi[\psi/X]}$
	(Prefixpoint)	$\varphi[(\mu X. \varphi)/X] \rightarrow \mu X. \varphi$
	(Knaster-Tarski)	$\frac{\varphi[\psi/X] \rightarrow \psi}{(\mu X. \varphi) \rightarrow \psi}$
Technical Rules	(Existence)	$\exists x. x$
	(Singleton)	$\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$

This is the entire logic. It has 7 syntactic constructs for building *formulae*, called *patterns* in Matching Logic, and 15 *proof rules*. We only showed the proof rules above (the syntactic constructs can be inferred from the syntax of patterns). These are mathematical, logical proof rules, not cryptographic proof rules. Mathematical proof rules, aka inference rules, allow you to derive a *conclusion* once you derive the rule *premises*. When a rule has premises, they are written above a horizontal line, while the conclusion is written underneath the line. See the *modus ponens* rule, for example: you can derive ψ once you derived φ and $\varphi \rightarrow \psi$. There are five rules with premises in Matching Logic. When an inference rule has no premises, it is called a *fact*, or an *axiom*. There are ten axioms in Matching Logic. By abuse of language, patterns that are part of the mathematical *theory* Γ defining the semantics of a target PL, are also called axioms; we may use *axiom of* Γ , or *axioms in* Γ , to avoid confusion with the general, theory independent Matching Logic axioms. For all practical purposes, the generic axioms of Matching Logic and the axioms in a theory Γ are used the same way in proofs. But it is common to keep

them separate and think of the former as part of the logic, while of the latter as part of the theory. The former can be used in all proofs, while the latter only in proofs of Γ theorems.

With these 15 Matching Logic proof rules you can derive, starting with axioms, theorems of interest. And such theorems are *correct by construction*, because you construct actual proofs for them, using the proof system. This very small proof system allows you to define any programming language as a theory Γ and any claim you make about that language, including that a certain program execution is correct wrt the language semantics itself, that a certain program is a correct implementation of a certain specification, that a certain “bad” state (e.g., division by zero) cannot be reached in a given program regardless of the input, etc., as a theorem φ . Why? Because we have a translation from K to Matching Logic. Let’s be more formal. The fact that φ is *provable* in Γ is written $\Gamma \vdash \varphi$, and a (Hilbert-style) proof, or a *proof object* for it, is a finite sequence $\varphi_1, \varphi_2, \dots, \varphi_n$, where φ_n is φ and each φ_i with $1 \leq i \leq n$ is either an axiom (of matching logic or in Γ) or derivable using one of the five matching logic proof rules with premisses, from previously derived patterns used as premisses; i.e., there is one proof rule instance whose conclusion is φ_i and whose premisses are among $\varphi_1, \varphi_2, \dots, \varphi_{i-1}$.

In fact, in spite of its simplicity, Matching Logic is very general and expressive. Everything that not only K does, but also what Coq, Lean, as well as other interactive theorem provers do, some based on complex type systems, all these can also be framed as provable Matching Logic theorems of the form $\Gamma \vdash \varphi$, where Γ is the theory and φ is the theorem that is proved. We have a [paper in ICFP’20](#) – the International Conference on Functional Programming – in which we show how the proof systems underlying these interactive theorem provers can be *shallowly embedded* in Matching Logic. That means that they are just notations, that they can be desugared mechanically into Matching Logic theories, and proofs. In other words, we can use not only K, but also Coq, Lean, and other interactive theorem provers, to generate proofs that then can be checked with the Matching Logic proof checker (discussed shortly).

However, we prefer K. First of all because we as a team understand its code base well: if we need anything, we know where to ask help. But also, in K, *computation is proof*. Directly. Indeed, when you execute a program using the K generated interpreter, say that you execute `factorial(3)` and get result 6 using the semantics of Java, an actual direct proof of that particular claim is being produced by that particular execution. In other words, the program execution yields a proof object $\varphi_1, \varphi_2, \dots, \varphi_n$, where we can think of each pattern in the proof sequence as an execution step of the program. This is in contrast to other systems, like Coq, where you do a proof and then there is a certain extraction mechanism, where you extract a program from a proof, but only if that is done in a certain fragment of Coq, which is constructive in such a way that you can extract actual programs from it. We don’t need this extraction mechanism in K; the execution of the program is a straightforward and rather boring (for humans) proof object.

Second, K is very fast: its interpreters automatically generated from formal language semantics, for example, compete at performance with existing interpreters specifically hand-crafted for those languages. K has been specialized for programming languages, while many of these other provers (Coq, Lean, and so on) are not specialized for programming languages; they are general purpose mathematical provers, with emphasis on human interactivity and friendliness, not on automation or performance.

Finally, we already have many programming languages formalized in K: C, Java, EVM, AVM,

WASM, and so on. It would be a huge effort to translate all those into other other formalisms or provers. Besides that, as I mentioned, we already have Coq and Lean backends to K. Actually even more generally, formalizations of Matching Logic in these. With these, we can take anything in Matching Logic and *deeply embed* it (i.e., encode it, in contrast to desugar it, which would be a shallow embedding) into Coq or Lean. Thus, we can use these interactive provers to derive proofs in Matching Logic. In fact, we actually use Coq significantly in our company. Probably not as much as K, but almost as much as K. We are fans of and believers in interactive theorem provers. The reason we use K, again, is that it is very intrinsically and directly connected to Matching Logic, and Matching Logic allows us to produce very low level proof objects that can be checked with the minimal proof checker that I will discuss shortly.

200 LOC Proof Checker for Matching Logic

To implement a proof checker for a logic, there are usually two levels. You first implement the proof checker in some programming language. Then you compile that programming language into something else, in order to execute it. For example, Coq's proof checker is implemented in several thousands lines of OCaml code, and then that OCaml code is compiled down to lower level architectures, like x86, going through LLVM and its optimized pipeline or through some other compiler infrastructures.

In our case, for Matching Logic, we chose [Metamath](#) as an implementation language for our proof checker. The reason we chose Metamath is mainly because it is a *very simple* language, yet *very low level and expressive* at the same time. There are more than 20 [Metamath verifier implementations](#) already: in C, Rust, Haskell, OCaml, and so on. Most of these Metamath implementations have only a few hundred lines of code; the largest, in C, has 2500 lines of C code. What we did was to [implement Matching Logic in Metamath](#), or to *define* Matching Logic in Metamath, as a Metamath theory, in 200 lines of Metamath code (199, in fact). It looks like in the left column in the picture below, where we can see some axioms (axiom-1, axiom-2) and the modus ponens rule:

<pre> 1 \$c \imp () #Pattern - \$. 2 3 \$v ph1 ph2 ph3 \$. 4 ph1-is-pattern \$f #Pattern ph1 \$. 5 ph2-is-pattern \$f #Pattern ph2 \$. 6 ph3-is-pattern \$f #Pattern ph3 \$. 7 imp-is-pattern 8 \$a #Pattern (\imp ph1 ph2) \$. 9 10 axiom-1 11 \$a - (\imp ph1 (\imp ph2 ph1)) \$. 12 13 axiom-2 14 \$a - (\imp (\imp ph1 (\imp ph2 ph3)) 15 (\imp (\imp ph1 ph2) 16 (\imp ph1 ph3))) \$. 17 18 \${ 19 rule-mp.0 \$e - (\imp ph1 ph2) \$. 20 rule-mp.1 \$e - ph1 \$. 21 rule-mp \$a - ph2 \$. 22 \$} </pre>	<pre> 23 imp-refl \$p - (\imp ph1 ph1) 24 \$= 25 ph1-is-pattern ph1-is-pattern 26 ph1-is-pattern imp-is-pattern 27 imp-is-pattern ph1-is-pattern 28 ph1-is-pattern imp-is-pattern 29 ph1-is-pattern ph1-is-pattern 30 ph1-is-pattern imp-is-pattern 31 ph1-is-pattern imp-is-pattern 32 imp-is-pattern ph1-is-pattern 33 ph1-is-pattern ph1-is-pattern 34 imp-is-pattern imp-is-pattern 35 ph1-is-pattern ph1-is-pattern 36 imp-is-pattern imp-is-pattern 37 ph1-is-pattern ph1-is-pattern 38 ph1-is-pattern imp-is-pattern 39 ph1-is-pattern axiom-2 40 ph1-is-pattern ph1-is-pattern 41 ph1-is-pattern imp-is-pattern 42 axiom-1 rule-mp ph1-is-pattern 43 ph1-is-pattern axiom-1 rule-mp 44 \$. </pre>
---	--

Matching logic syntax
and proof system

Claims with proofs
(machine checked)

Very mathematical, almost identical to our proof system. Therefore, the Matching Logic proof system can be rigorously implemented, or defined, in 200 lines of Metamath code.

We can also encode claims and proofs of these claims in Metamath, like we have in the right column in the picture above. Those proof objects are *very long* in practice. The proof above, for $\varphi \rightarrow \varphi$, is simple and unusually short. We can similarly prove any claim. Anything that K does can be translated into such a proof, possibly having millions of steps. Very long and likely very boring proofs. But very precise and very low level. Exactly as we want them to be!

Basically, any claim derived with K or any other formal systems or interactive theorem provers, in the end reduces to verifying a formal claim $\Gamma \vdash \varphi$ in Metamath: under given theory Γ , a given theorem φ is true because a proof object of it has been provided and has been checked.

The trust base consists of the 200 LOC definition of Matching Logic in Metamath, plus the few hundred LOC implementation of Metamath itself. Indeed, we have to trust both of these. Besides its soundness proved on paper (LICS'19), the (tiny) Matching Logic definition is validated also by testing it against a large variety of languages, programs and proofs in those languages. To minimize risk in the correctness of the Metamath implementation itself, which can become an important issue in security-critical applications like blockchains, proof validators can actually run all the Metamath implementations in parallel and only accept proofs that pass all of them. Note that the Metamath language is precise and unambiguous. That is, if two Metamath implementations disagree on an input, one of them has a bug that needs to be fixed.

The trust base of our Matching Logic approach is therefore orders of magnitude smaller than the trust base of other frameworks, like Coq or Lean. This is *not* an accident: Matching Logic is the result of more than 20 years of targeted work, designed from basic principles to be the smallest possible without making compromises wrt expressiveness. The price to pay for its succinctness and simplicity is that proofs are not human comprehensible. This was never a requirement, not even remotely. Matching Logic can truly be thought of as the “machine-code of proofs”. On the other hand, frameworks like Coq and Lean provide high-level, human-friendly abstractions and properties, such as builtin (dependent) types, proof tactics and strategies, program extraction from proofs, and many other goodies. The price to pay for these abstractions is that their underlying logical formalism and thus proof checkers are more complex.

ZK-ing the Matching Logic Proof Checker

The Matching Logic proof checker is a program $\mathbf{pc}(\Gamma, \varphi, \Pi)$ that takes three inputs:

- The theory Γ , which is the formal semantics of the programming or specification language in which the claim is stated.
- The theorem φ that is claimed to hold under Γ .
- The mathematical proof object Π for the claim, which is usually very large.

Note that the first two arguments form the actual claim $\Gamma \vdash \varphi$ that is being made, which is public. The third argument, Π , is expected to be automatically produced by tools and frameworks like K; in theory, they can also be produced by hand, or any other means.

Next we want to implement the Matching Logic proof checker as a ZK circuit, to produce

succinct cryptographic proofs. Unfortunately, none of the existing implementations of Metamath use programming languages with ZK proofs or cryptographic technology support. We have recently completed the first steps in this direction. What we've done, specifically, was to re-implemented Metamath in a version of Rust, more precisely in a fragment of Rust using some libraries that are supported by [RiscZero](#)'s compiler and their [zkVM](#) infrastructure. Executing the Matching Logic proof checker implemented in this new version of Metamath on the RiscZero infrastructure, we can now generate a succinct SNARK proof certificate that a mathematical proof object for the claim exists.

Formally, our SNARKed implementation of the Matching logic proof checker produces a cryptographic certificate π for the fact that there *exists* a Π such that $\text{pc}(\Gamma, \varphi, \Pi)$. Although there could be situations where one may want to keep Π private, e.g., for commercial reasons or because it may reveal information intended to stay private, currently our main reason to SNARK Π away is its size: Π is simply too large to be practically shipped as a correctness certificate for the claim that it proves.

This is still a work in progress, a collaboration with [Tim Carstens](#) and his colleagues at RiscZero, and also with my UIUC colleague [Andrew Miller](#) and his research group. Note that Andrew Miller is also a [cryptography advisor](#) in our company. The results so far are encouraging, although performance of SNARK proof generation is currently the main bottleneck; this is consistent with other ZK projects, which appear to all suffer from the same fundamental problem – ZK proof generation is slow. We are now in the process of experimenting with RiscZero zkVM's nascent support for recursive STARKs.

It should be clear that, unlike other ZK projects which are bound to a particular ZK infrastructure (blockchain, language and/or library), we do not depend on RiscZero's infrastructure. If RiscZero eventually gives us an efficient ZK proof generator, then we will happily take it. But we can similarly use other ZK infrastructures. In fact, we are currently in the process of implementing our Matching Logic proof checker also in Solidity, in order to compile it and check it with Polygon's and others' zkEVM, and in Cairo, in order to use StarkNet's infrastructure, as well as in (a Rust variant that compiles to) LLVM, in order to use NIL Foundation's zkLLVM. We hope to be soon able to report on these experiments. The beauty and strength of our approach is that if any of the aforementioned ZK languages deliver to their promise, namely that it will efficiently generate ZK proofs for its hardwired language, then we can immediately leverage that and provide the same efficient ZK infrastructure for *all* languages!

The unique feature of our UTF is the crystal clear separation between producing Matching Logic proofs (with frameworks like K, Coq, Lean, etc) and verifying those proofs with a ZK-based proof checker, which is same for the entire Matching Logic, that is, for all PLs and all specification languages. This way we can compress large mathematical proof objects into succinct, even fixed size ZK certificates. Unlike other approaches, like the ones mentioned above, we need no ZK compilers, no ZK virtual machines, no program-specific or even language-specific circuits. Thus no complex ZK artifacts that need to be trusted or formally verified. Of course, if we implement our approach in any of the ZK languages above, we will need to trust or formally verify our implementation as well as the infrastructure upon which it depends.

However, once we complete the experiments above, our plan is to implement a circuit directly for our Matching Logic proof checker. After all, Matching Logic's proof checker has only 200

lines of code. We don't need to go through another ZK infrastructure or language. We can manually craft a circuit, super optimized directly for Matching Logic. One circuit to rule them all, because combined with a language semantics, it will be capable of producing ZK certificates specific to that PL. Once we have an efficient ZK proof checker for Matching Logic, we can therefore have the same benefits as all these other existing ZK approaches for specific PLs.

By going to the meta-level and observing the program execution as a rigorous mathematical proof, we avoid the hard task of implementing a specific VM as a circuit. We keep the VM as is. Actually, we don't even need nor want a specific VM implementation, as that may come with bugs and inconsistencies. From the formal semantics of the VM, which is non-negotiable in any context where security is a must, we generate an executable model of the VM which allows the Matching Logic proof checker to observe from the meta-level its execution and check it and produce a ZK certificate of its existence. We re-emphasize that Matching Logic, as well as its ZK-based proof checker, are unique for all languages, all specifications, and all properties. So there is only one ZK circuit or cryptographic artifact.

The curious reader is encouraged to note that our approach generalizes not only all the language-specific ZK approaches discussed above, but also [Verifiable Computing](#) and [Proof-Carrying Code](#). Indeed, it offers those for any language with a formal semantics.

More Powerful than Language-Specific Solutions

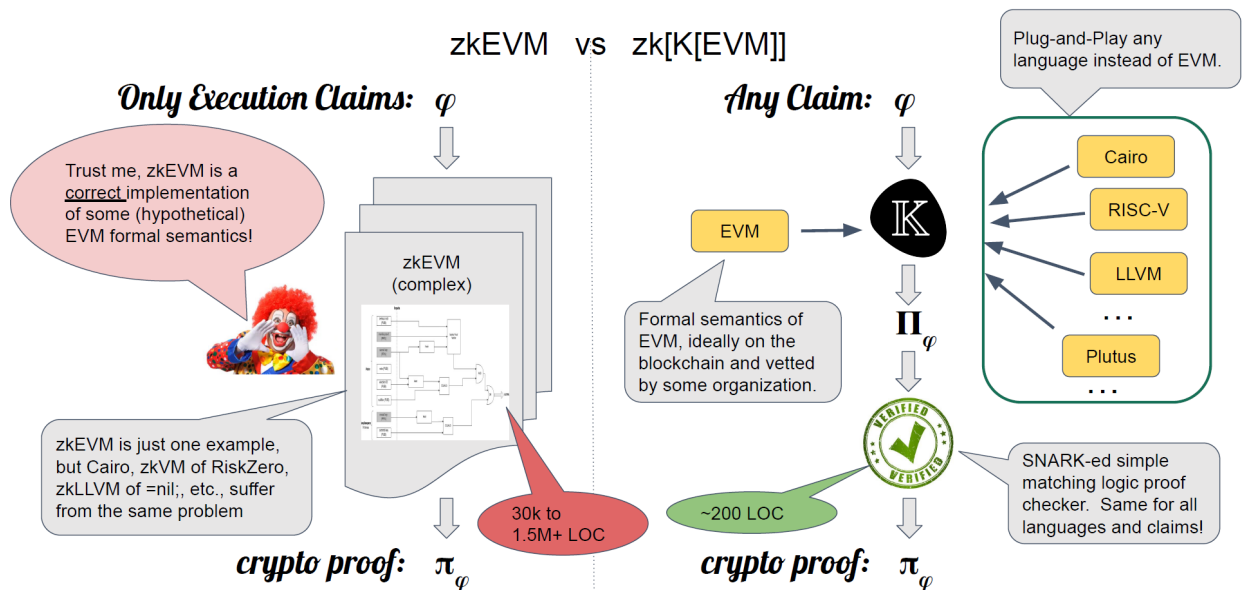
The UTF is more general, because it is universal, and yet it yields a smaller trust base and ultimately a smaller and simpler ZK circuit than language specific solutions.

We next only consider zkEVM, for the sake of concreteness. This is just one example. The same would happen with the Cairo PL (StarkWare), the zkVM of RiscZero, or the zkLLVM of the NIL Foundation. All of these would suffer from the same problem. The UTF is more powerful than all these, for the same arguments discussed below in the context of zkEVM. Before we dive into details, recall that the UTF has only one cryptographic artifact which needs to be trusted or formally verified, a ZK circuit implementing a 200 LOC proof checker, which works with all claims about all programs in all programming languages. Everything else being equal, this is already a big and unique advantage of our approach over the language-specific solutions.

I'd like to say it upfront that we are very impressed with what the Polygon team achieved overall and we are users of Polygon ourselves and grateful to their ecosystem! Our discussion here is purely factual and scientific. Our goal is not to criticize their methods or approach, but simply to explain our approach and convey to the reader our new technology, by comparison with an already established technology. If anything, we acknowledge zkEVM as the most well-established competitor. Besides, our idea, method and approach were not available when the Polygon team started implementing zkEVM. Even now, our approach still requires some more work and experiments before it becomes a viable replacement of the existing language-specific approaches.

The way zkEVM works, at a high level and only for our specific purpose here, is that you make an execution claim using EVM, φ let's say, and get it through the zkEVM circuit. The zkEVM will

produce a ZK proof of that execution, say π_φ . So at a very high level, you have an execution claim φ , you get it through the zkEVM which gives you a certificate π_φ that confirms execution is correct. This is a very strong correctness claim, in the sense that it requires a significant amount of *blind trust* from us, trust in informal arguments and unverified code, as discussed shortly. The first question is: correct with respect to ... what!? I will get back to that, but for now let's move on with this high-level intuition for zkEVM. See the left side of the picture below.



In contrast, the UTF, shown to the right in figure above, produces cryptographic certificates for the correctness of *any claim* given as input, not only for execution claims like zkEVM. A non-execution claim can be that a given EVM bytecode is a correct ERC20 implementation, for example. So the UTF approach is more powerful than zkEVM even if we just stop here. But there is more. As a reminder, the “UTF approach” here means that we plug the [EVM formal semantics](#) into the K framework, and then we use the generic ZK Matching Logic proof checker to certify mathematical proofs for EVM execution claims that are produced by the K framework. For that reason, we here refer to our generic approach instantiated with EVM as $ZK[K[EVM]]$: plug EVM into the language-parametric K framework, then plug the generated mathematical proofs into the framework-independent (i.e., not specific to K) ZK Matching Logic proof checker. The EVM semantics plugged into K yields an executable model of EVM, an interpreter basically, but one which can produce the mathematical proofs of execution.

Again, the UTF approach works for any claim, not only execution claims. But for now, let's just deliberately limit the UTF to input claims that zkEVM supports, namely just normal executions of EVM programs. So we have such a claim, φ , that some program execution produces a certain result, and $K[EVM]$ produces a mathematical proof, Π_φ , which is precise but likely very long. The large size of Π_φ is not a problem, though, because we check it with our ZK Matching Logic proof checker, get a succinct cryptographic certificate that a mathematical proof for the claim φ has been produced and has been checked, and then we can discard the long mathematical proof Π_φ . Note that the cryptographic certificate produced by our approach $ZK[K[EVM]]$ may be different from the one produced by zkEVM, because different ZK methods may produce different certificates, both succinct and both confirming the same truth, that φ is correct.

Hence, both zkEVM and ZK[K[EVM]] get the same overall result on execution claims. But the question is how they do it. zkEVM implements a very complex circuit. With the help of several zkEVM experts and friends from the EVM community, we tried to measure how large the trust code base for the zkEVM circuit is. It turns out that it is anywhere between 30,000 LOC and more than 1.5 million LOC, depending on how you count. Would the generator of the circuit also be part of the trusted code base? Would the implementations of the two additional programming languages that the zkEVM team invented in order to specify the circuit also count as part of the trusted code base? None of these were formally verified, not to mention that it is not even clear what formal verification means in this context, so they just need to be trusted. But even in the most optimistic possible counting scenario, where all languages and compilers involved are assumed correct, zkEVM still has more than 30,000 LOC that implements a rather complex VM as a circuit. Why should we trust all this code? Folklore and experience in software engineering tells us that good quality code still has a bug in every 1000 LOC.

The big claim the Polygon team makes is that their zkEVM is a correct implementation of EVM as a ZK circuit. That is, that 1.5+ million LOC implement a hypothetical language specification. Indeed, there is no EVM rigorous specification that they claim their circuit implements. They're just saying, in words, that they try to adhere as much as possible to the [Yellow Paper](#). The Yellow Paper is already obsolete, that's broadly accepted. The only complete and religiously maintained formal specification of EVM that we are aware of is the [KEVM semantics](#), which they do not mention anywhere in their documentation. It is therefore safe to assume that there is no reference formal specification of EVM that zkEVM implemented. Moreover, even informally, it is very likely that they implemented an older version of EVM, because in the meantime, EVM itself suffered a few upgrades. That is, there is likely a gap between the EVM version implemented by zkEVM and the actual EVM on Ethereum they claim compliance with. Unless zkEVM is generated directly from an EVM specification, like our ZK[K[EVM]] does, there is a good chance that there is a semantic gap between EVM and zkEVM.

My point is that zkEVM has a complex code base, manually crafted and specific to one VM language. *One language!* A complex, adhoc circuit claimed to implement something which is not rigorously specified. Suppose that the Polygon team decides to do things right and increase confidence in their zkEVM by formally verifying their zkEVM circuit. How is that even possible? Against what EVM specification?

Contrast that to our UTF approach, ZK[K[EVM]], where we just plug and play the EVM formal semantics into the framework and obtain the same results as zkEVM, but in a correct-by-construction manner. In our [Proof Chain proposal](#), formal semantics of languages will be stored on the blockchain, vetted/signed by appropriate entities that are in charge of the language. For example, the EVM formal semantics should be stored at some address on the blockchain, vetted by, probably, the Ethereum Foundation (EF). When this happens, the trust base of ZK[K[EVM]] reduces to only the circuit implementing the 200 LOC proof checker! Indeed, notice that *we don't have to trust K*; the role of K is only to search for and produce mathematical proof objects. The proof objects are checked with the ZK Matching Logic proof checker, which takes as input the exact same EVM formal semantics that K used to produce the proof objects, vetted by the EF.

There is, therefore, a sharp contrast between our generic approach and the VM specific

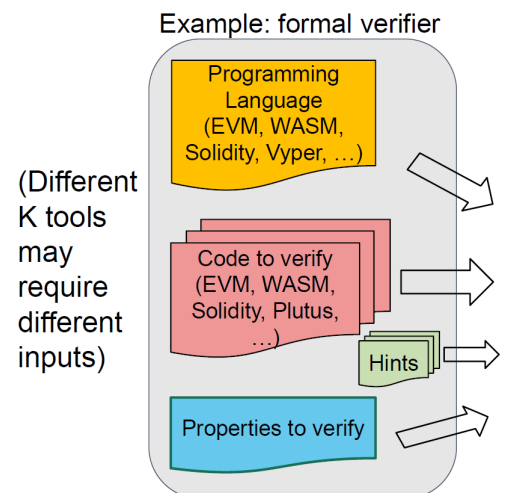
approaches. Recall that the discussion above used zkEVM / EVM as an example, but that we can instantiate our framework with any other language instead of EVM. Indeed, neither K nor the Matching Logic proof checker know anything about EVM, or about C, or Java, or Cairo, or RISC-V, or LLVM, or Plutus, or about any particular language. These are just formal semantics in K, or mathematical theories in Matching Logic, ideally available in some public and trusted places, like on blockchains, vetted by their respective committees. Moreover, there is only one ZK artifact, implementing the 200 LOC Matching Logic proof checker, to be used with *all languages*. When you plug a given formal language semantics into our framework, you can then produce cryptographic proofs for any claims about that language, in particular for executions.

As I mentioned earlier, I believe that blockchains of the future will not even have VMs in their validator nodes. VMs are complicated and error prone. Further, to implement a ZK circuit for an entire VM is a very complex task which, as I showed, is unnecessary. I firmly believe that in the future, we will have formal semantics of languages that we want to use to describe protocols, to describe smart contracts, transactions, and those are stored on the blockchain, vetted by their communities, and then you can generate from them directly an execution engine on your local machine and run your transactions locally, produce the mathematical proofs and then generate a ZK certificate of existence of that mathematical proof. You present that to the other nodes in the blockchain and with a lightweight consensus protocol that executes only one program, the ZK Matching Logic proof checker, the entire network synchronizes. This way you can write smart contracts in any programming language for which we have a formal semantics. It is that powerful, yet that simple and correct-by-construction.

Next I'd like to talk about two imminent recurrent revenue products that we are going to launch very soon, in 2023. One of them will produce revenue pre-deployment of client smart contracts, and the other one will produce revenue post-deployment. If you are more interested in how the UTF can be used to give birth to the Proof Chain, then [skip there](#).

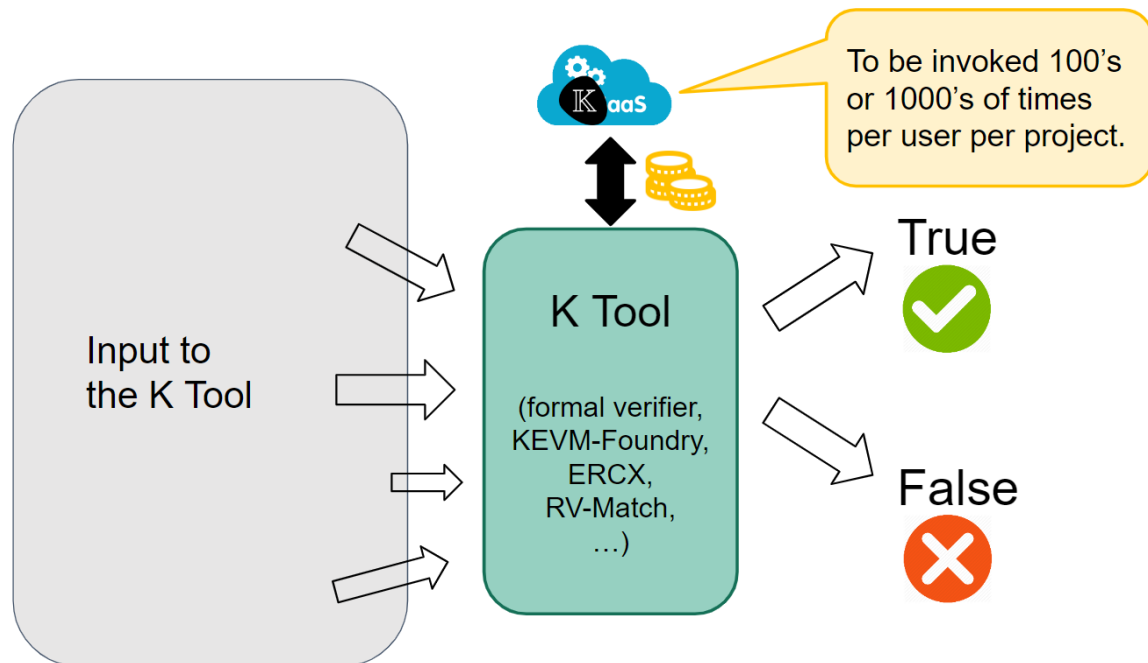
2023 Recurrent-Revenue Pre-Deployment Product KaaS — K as a Service

The pre-deployment product that we will launch in 2023 is the "K (prover) as a service", or KaaS, which works as follows. Take any of the tools that the K framework provides. For simplicity, think of a program verifier. But it can also be an execution engine / interpreter, or a model checker, or a symbolic execution engine. Usually, a K tool takes inputs. The PL semantics is almost always an input, but there could be other inputs as well, such as hints, lemmas, etc. The figure to the right, for example, shows the inputs that the K formal program verifier currently takes. The arrows indicate that the inputs are passed to the K tool. Based on the inputs, the K tool will do its job and it will tell us its output, for example, True or False.



During the last few months, we refactored the K framework and its tools to identify and isolate those components which do the actual mathematical proofs, the actual hard work. Then we

implemented an API that encapsulates those components in charge of proving and exposes them programmatically. In other words, we can now interact with the K prover using an explicit API. For internal use, we configured a SaaS-style server, which we call KaaS, that runs on AWS (we are experimenting with other cloud services, too). The various K tools now call into KaaS whenever they need to do or generate any mathematical proofs. The following picture illustrates how KaaS is used by the K tools:



Separating the K prover code base and offering it through an API via KaaS has the technical benefit that it can be now advanced independently of the other K tools. Indeed, we have already added several optimizations and parallelization to make it fast, and more optimizations and improvements are added daily.

Our plan is to monetize our K proving technology, by means of KaaS subscriptions. The idea here is to offer lots of tools, like the K program verifier which works with any programming language, like KEVM-Foundry and like ERCx for Solidity and EVM, as well as similar tools that will work with other languages and blockchains, like Rust and WASM on Polkadot, MultiversX, Solana and others, like KAVM and KTeal on Algorand, like KMichelson on Tezos, like Plutus on Cardano, etc., but also C or Java in safety- or mission-critical systems (aviation, medical, automotive, embedded). We have almost completed the K formal semantics of many of these languages, as paid engagements with various clients (detailed in our [RV Blog](#)), and we are currently working on developing formal verification, symbolic model checking, and testing tools for these respective ecosystems. These tools will be offered to the community, together with documentation and education material, including regular workshops.

We ultimately want a large number of smart contract developers as well as developers of safety- and mission-critical software to use our K-powered tools on a daily basis. We remind the reader that the K framework code base has always been [open source on Github](#), under the permissive MIT license. In fact, many of our clients explicitly said that they chose RV over other formal

verification auditors for this exact reason, that our technology is open source. We have no plans to change that, K will continue to be open source and available to everybody everywhere. However, users of the K-powered tools who are interested in special features and functionalities, good user experience, in particular in performance and in our undivided attention, will have the option to buy a subscription to KaaS. Subscribers will take advantage of maximum parallelism, optimal proof generation and caching with KaaS in the cloud, which can result in waiting seconds for a proof to be done in the cloud instead of minutes or hours with the free version of the prover on a local machine.

Subscription-based models for software analysis products are common practice. That's how many tools in the embedded systems and mission-critical space are delivered, for languages like C and Java, and also how formal verification companies like Certora offer their technology to paying Ethereum smart contract developers. Therefore, developers will find no surprise in our offering, except, perhaps, that we will also offer a free version of our prover. However, we have several unique advantages over our competitors, which we believe will position us favorably long-term:

1. K's semantics-based approach to formal verification results in many, possibly thousands, *small proof obligations* for a given property to be proved, instead of one large proof obligation like in traditional formal verification. Indeed, traditional Hoare-style formal verification approaches, including Certora's, generate large SMT (Z3, CVC5, etc.) formulae that incorporate not only the assertions to be proved, but also a significant portion of the programming language semantics. That is, the program semantics is encoded as part of the formula, which is then sent to the SMT solver. More often than not, the solvers crash or run out of time or memory due to the inherent complexity of solving such complex constraints. In contrast, our K-powered tools handle all the programming language semantics themselves, in a correct-by-construction manner executing the programs symbolically step-by-step driven by the semantics, same like debuggers do, and only generate SMT proof obligations when a domain property that requires actual domain reasoning is needed and cannot be discharged by the K tool itself. Basically only to check the side conditions of the semantic rules, mostly. This way, the SMT queries we make with our K-based approach are more tractable, usually being discharged instantaneously. More importantly, our approach is almost embarrassingly parallelizable, in the sense that the various SMT queries can be solved in parallel. This is a perfect scenario for our KaaS server solution.
2. K's semantics-driven approach makes it possible to *generate proof objects* for the properties that are proved, whose trust base is only the formal language semantics. I re-emphasize that this is only possible because our proofs are driven by the language semantics itself and various general-purpose, language-agnostic automated procedures implemented by our K tools. Tools based on traditional formal verification approaches are language-specific, that is, the language semantics is hardwired into the tool, and thus the tool itself becomes the trust base. One cannot thus separately audit and validate the language semantics itself, because that does not exist as a separate artifact. The translation itself to an SMT formula needs to be trusted in those approaches. It is not easy, in fact in our experience it is close to impossible, to disentangle the language semantics from the actual property being proved from SMT formulae and their solutions using Hoare-logic, traditional verification approaches. Not to mention that the Hoare

logic itself, which is the mathematical foundation underlying the tool, needs to be proved sound, which is well-known to be a hard task in itself even for relatively small languages. For that reason, the soundness of the underlying Hoare logic is usually skipped in practice. I am not aware of any such task for a real-world language, like C or Java, to have been ever completed, in spite of attempts. This is a serious impediment when trustlessness and correctness are paramount: without a soundness proof for each Hoare logic proof rule one cannot generate proof objects as correctness certificates!

3. K's semantics-driven approach allows us to generate proofs not only of verified properties, like in formal verification, but also of program executions, like in verifiable computing. It is this crucial capability of our approach, that *computations are proofs*, which enables the UTF and, eventually, BoT. With traditional formal verification approaches, even if all the technical problems like those mentioned above are resolved (i.e., dealing with large and complex SMT proof obligations, and generating proof objects as correctness certificates), one could only handle formal verification claims, but not program execution claims.
4. The above will directly translate into multiple revenue streams for KaaS and our company. Indeed, KaaS will be used not only by developers of smart contracts, but potentially by *any blockchain user*. Developers need proofs in order to formally verify or validate their code, and to also convince their users that their code is high quality. But blockchain users will need to generate proofs with every transaction they perform on the blockchain, proofs which will be checked by validators before they validate the transaction. Considering that developers are only a fraction of a percent of the total number of blockchain users, and that the same user can make many transactions per day, the possibilities for KaaS to produce revenue take us to another dimension compared to our competition.
5. Finally, thanks to K's language-parametric nature, KaaS works with all programming languages and all blockchains! That is, there is no specific support added to KaaS for Solidity, or EVM, or Rust, or WASM, or C or Java. These language semantics will be stored in some public repository or on the blockchain, and KaaS will simply read the language semantics from there. The more languages, the more developers, the more users, the more KaaS usage. All these without us having to touch KaaS. We will help the community develop high-quality semantics of popular languages and tools for them that will make use of KaaS. KaaS has the potential to become the *universal proof generator, to work with all languages, all tools, and all blockchains based on ZK technology*.

2023 Recurrent-Revenue Post-Deployment Product Invariant Monitoring & Recovery

The second product line we have slated for 2023 is blockchain runtime monitoring. These smart contract specification and invariant monitoring and recovery products leverage the specifications and invariants statically proven during the audit to offer value to our clients (and revenue to our company) *after* the target smart contracts are deployed. They can be used to monitor and inform clients about the health of their deployed contracts, and can even automatically launch

corrective action when invariants are broken.

These products are perfectly aligned with the mission of Runtime Verification. The primary outcome of a formal verification audit is the identification of the specifications and invariants that must hold true when the smart contract is deployed and operational. In other words, since auditors create the exact formal specifications and invariants, they have the *best* understanding of what needs to be monitored at runtime.

We have a rich history of developing runtime monitoring techniques, systems, and tools, and even [coined the term “runtime verification”](#) in 2001. In conjunction with my NASA colleagues, we envisioned a technology that takes the same input as formal verification to perform light-weight runtime monitoring and recovery after deployment. Some of our early runtime monitoring and recovery systems developed at NASA and UIUC can be found on the [RV Wikipedia](#) page. We also developed some monitoring systems at RV Inc., before we moved into the world of blockchain: [RV-Monitor](#) for embedded systems and [RV-ECU](#), to monitor vehicle ECUs in the automotive domain, to name a few.

The key learning we took away from developing runtime monitoring and recovery products over the past 20+ years is that *the hardest part, by far, was to come up with the right properties to monitor*. Typically it requires working closely with clients, domain experts, safety engineers, and standards committees who often do not have a background in formal methods and are more concerned with politics than system safety and security. All that aside, once these properties are available, the rest was relatively easy. Simply instrument the system to correctly observe the relevant events, automatically generate monitors from the safety specifications, and steer the system through recovery actions when violations happen. We cannot emphasize enough how fortunate we are to *already have the specifications and invariants* that must be monitored, as well as the adequate recovery actions! All we have to do now is leverage our expertise to generate the correct monitors from these specifications, observe the activity of the monitors, and trigger transactions if necessary.

Over the last few months we have been experimenting with runtime monitoring of smart contracts for our clients. Through this work we identified two broad categories used to classify our monitoring and recovery products:

Without Recovery. In this category the monitor observes the behavior and activity of the specified accounts and notifies account stakeholders off-chain. These monitors are harmless to the blockchain as they trigger no transactions. The notifications produced by these monitors can trigger corrective actions by the stakeholders based on recommendations identified during the audit process. RV has been working with clients to monitor invariants identified during their smart contracts audits, as well as any additional properties of interest to the customer. These monitors inform stakeholders of the health status of their protocol, particularly in cases where specifications and invariants were violated.

With Recovery. The second category of monitoring services involves triggering recovery transactions. Monitors that fall into this category include protocols in which correctness is conditioned by assumptions that must hold at runtime. For example, in a lending/borrowing protocol like [Aave](#), the *primary* property proven is that all loans *must* be overcollateralized *under certain assumptions*. In these cases, these assumptions must be monitored *and* recovered as

quickly as possible to prevent damaging results. In the Aave example, the only way to protect protocol health is to liquidate any undercollateralized loans. If these undercollateralized positions remain un-liquidated, the invariant is broken.

Runtime Verification is developing monitoring solutions to address three primary use cases:

1. Smart contract owners who completed their audit and are interested in analyzing the health of their smart contract post-deployment. In particular, they want to know if the security audit claim assumptions are indeed maintained. If these assumptions are violated, they may take actions to maintain their smart contract runtime health, or have third parties (e.g. Keepers) do it on their behalf.
2. Keepers interested in maintaining smart contract health for smart contract owners, other stakeholders, or simply because they want to make a profit. Keepers subscribe to our monitoring services to help them maintain protocol health.
3. RV as a Keeper to protect the protocols of the customers we audit. We are currently performing Keeper Services on three different blockchains (Ethereum, Algorand, and MultiversX, formerly Elrond) , for protocols that we audited and/or understand well.

Through these projects we have learned several important lessons:

First, efficient instrumentation and observation of transactions is critical for successful monitoring. We experimented with existing blockchain indexing services, like those offered by The Graph, and to our disappointment they were too slow for our needs. By the time our monitors were triggered the invariants were already violated or the liquidation transaction had already been made by another keeper. We ended up using combinations of such services and our own instrumented validator or observer nodes, reducing the monitor latency from seconds to milliseconds.

Second, MEV plays an important role that must be taken seriously. Some recovery transactions which were good in principle were MEV-ed by other players. We are currently investigating how to make optimal use of our K-powered technology, which should give us at least two advantages over our competitors:

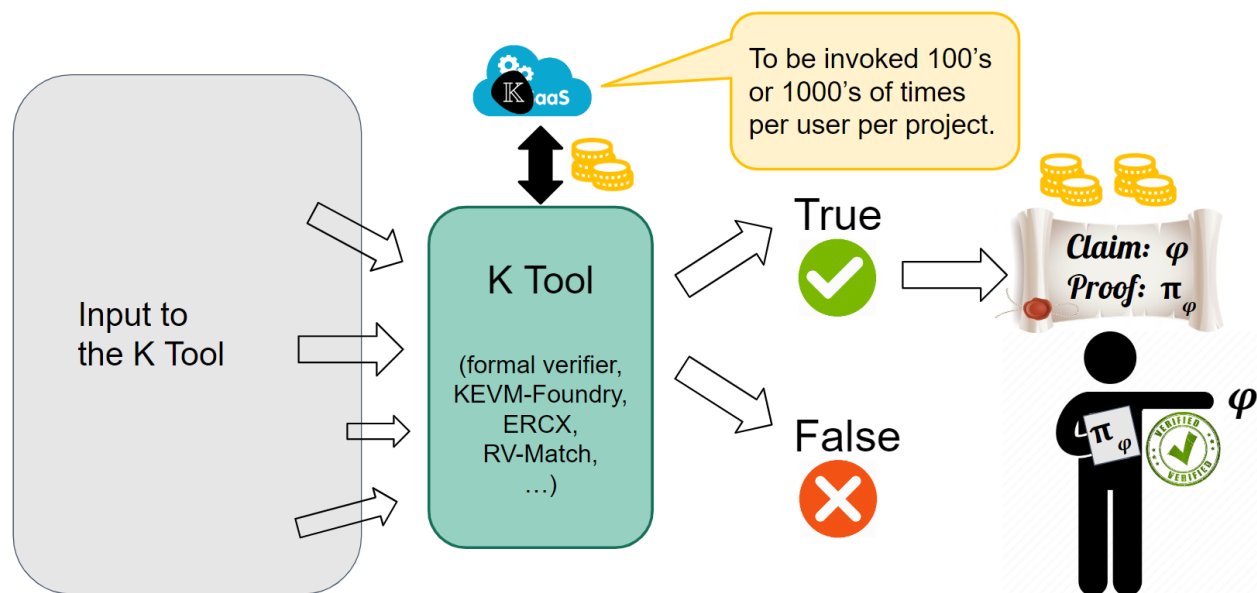
1. K has a built-in search capability, which finds optimal sequences of transactions to maximize a certain function on the program configuration; it was for this reason that [K was used](#) as a modeling language in projects that lead to the introduction of the MEV concept itself.
2. [KEVM](#) is a faithful semantics of EVM, which allows us to measure and model the exact execution of transactions using the aforementioned search capability. Moreover, our [K Summarizer](#), described in a previous section, yields a semantics-based compiler for EVM which, in our experiments, results in almost 10x faster EVM program execution than with existing interpreters. Combined with the search capability, this will give us 100x more time to search for optimal grouping of transactions than the MEV competition.

The two products that I mentioned, K as a service (KaaS) and runtime monitoring and recovery,

leverage both our reputation as formal verification centric security auditors, and our open-source K-powered technology. These cornerstones of RV represent more than 20 years of cutting edge experience that goes beyond the state-of-the-art in programming languages and formal methods. These products will augment the revenue from our security auditing services, tools, and other products we provide for our clients. As these products grow we will redirect our security auditing services to focus on essential, strategic engagements. Furthermore, this product vision excludes the impact of ZK proof certificates, which will be an entirely new revenue stream leveraging the unique capabilities of K, on top of everything we do already.

2024-2025: Commercializing Proof (of Proof) Certificates

Here we discuss our short term strategy to commercialize our capability to generate zero-knowledge (ZK) proof certificates. In 2023 we plan to mature our prototypes and to do more experiments and comparisons with existing ZK language implementations (like zkEVM, Cairo, zkLLVM, etc.). In 2024 we plan to incorporate the ZK capability into KaaS and into our formal verification security audits, and also start a few partnerships. By 2025, we plan to launch our Layer 0, the Proof Chain.



First, recall how KaaS works (left and middle of picture above). The various K tools in the K toolkit, which work with various programming languages and various blockchain ecosystems, uniformly interact with KaaS, using a generic API, whenever efficient proof generation or checking capabilities are needed. KaaS incorporates optimized algorithms, specialized hardware, and state-of-the-art automated reasoning to prove claims as efficiently as possible for the clients who are willing to pay a subscription fee in exchange for the best performance and user experience that K and RV can offer. Tools usually produce an output, which for the sake of simplicity we assume is binary: **False**, which usually means the tool was not able to find the answer to the claim; or **True**, meaning that the tool has found a positive answer to the claim.

However, that answer, **True**, is based on the *belief* that the K prover and KaaS did their job correctly. But K has 500k lines of code, in several programming languages, and mathematical

proofs can be and usually are quite complex. Then why should you trust K!? Or why should you trust what RV or I or anybody else claims? The point and beauty of proof objects is that you don't have to trust anything and anybody who cannot ultimately produce a proof for their claim. That is, there is nothing and nobody staying between a claim and its validity, except for a mathematical proof which can be checked by anybody who has it. This is the ultimate correctness argument, developed by humans over thousands of years, which no one can temper with if implemented correctly. And our job here at RV is to do precisely that!

As mentioned throughout this article, the main problem with mathematically rigorous proofs is their size. They can be and usually are huge. However, this issue is elegantly addressed by our ZK Matching Logic proof checker, which simply checks the long proof objects with a small proof checker program that produces corresponding ZK proof certificates, as shown to the right of the picture above. The overall effect is that the huge proof objects are now *massively compressed into succinct ZK proof certificates*, ideally fixed size, which can be checked by any third party instantaneously.

There could be many different ways and architectures to generate ZK proof certificates and deliver them to users. Since KaaS is already specialized and optimized to produce mathematical proofs, and to do that effectively it will make use of high-performance computing/hardware and pipelining, we believe that KaaS is uniquely positioned to also produce ZK proof certificates. We re-emphasize that although KaaS is a centralized service by design, meant to bring RV revenue over the next several years, its role is exclusively to provide its subscribers with proof certificates as efficiently as possible. KaaS will not pose any risk to the correctness of the overall framework. It will not be part of the trust base, by design. In other words, there is nothing that KaaS, or anything or anybody can do to “trick the system” into making an incorrect claim appear as true.

We truly hope that KaaS is only the first product of its kind, and that other companies will be inspired to develop similar products. Indeed, recall that Matching Logic is more general and powerful than K. It can similarly serve as a minimal foundation for other proof frameworks, like Coq, Lean, etc. And in theory, any program verifier and any interpreter can be instrumented to produce rigorous mathematical proof objects. K was designed in this spirit from first principles starting 20+ years ago, and thus has the competitive advantage of being a first mover, but there is nothing specific to K in our overall idea. On the other hand, for all the reasons already discussed, mainly for its minimality and expressiveness, we are firmly convinced that matching logic is perfectly suited for proof objects and the ZK proof checker. Because of that, we will design KaaS in a way that allows us to separate the ZK Matching Logic proof checker from the K prover itself, in order to offer it as a separate service if/when needed.

We next discuss our commercialization plan for ZK proof certificates.

First, notice that KaaS will already give us a stream of revenue from subscriptions even without the ZK proof capability, because it will be invoked by all tools built on top of K, for all blockchains. Subscriptions from developers, e.g. using tools like [KEVM-Foundry](#) to check their smart contracts; subscriptions from users and investors, e.g. those using tools like [ERCx](#) to check the tokens they want to acquire or bridge; and subscriptions from other security auditors who will use K-powered tools in their audits.

Second, notice that KaaS' capability to generate ZK proof certificates will open to our company, at a minimum, all the revenue streams available to other businesses producing ZK certificates. Take zkEVM as an example. We will be able to produce ZK proof certificates for any EMV smart contract execution, same as zkEVM. Moreover, as explained in a [previous section](#), in fact our approach conveys higher confidence in the correctness of the produced ZK certificates, because of a crystal clear separation between formal language semantics, proof objects, and a unique, language agnostic ZK circuit to check them. Recall that the latter implements a proof checker which has only 200 LOC. Besides its correctness, our approach also has the benefit of working with all languages and all VMs, and thus with all blockchains. In layman terms, it offers the benefits of zkEVM, Cairo, zkLLVM, zkVM, etc., all together using the same infrastructure and the same ZK circuit, by just plugging and playing the respective language or VM.

We believe that the generality and strength of our approach will position RV as a major player in the ZK arena over the next few years. Shorter term, over the next one to two years, however, we plan to achieve the following concrete objectives:

1. Offer our formal verification artifacts as ZK proof certificates, in addition to PDF documents and/or github readme files. This is a low hanging fruit, because we already do the formal verification proofs using the K prover, and we have already [implemented](#) and [published](#) a proof object generator for the K prover. Several of our clients have asked for such a capability. This is not only the right thing to do, but having undeniable evidence of smart contract correctness on the blockchain can open a new range of applications. Consider, for example, a bridge meant to allow transfers of arbitrary ERC20 tokens. Since some ERC20 tokens are scams, like rugpulls, making sure that a token correctly implements the ERC20 specification becomes critical for such an application. This can be checked with tools like our [ERCx](#) tool/product, but wouldn't it be nice for such proof evidence to be stored on-chain, trustlessly, by that token's creators or stakeholders or anybody else, once and for all, so that such tokens can be added to the bridge automatically, by anybody? We plan to work closely on this topic with our clients who hired us to audit related smart contracts, such as [Blockswap](#) (regarding their [Gateway](#)) and [Jump Crypto](#) (regarding [Wormhole](#)).
2. As mentioned, we can do with our general approach everything that other entities (companies, foundations, etc.) producing ZK certificates can do, and as explained, we even have several advantages. However, we believe that partnering with such entities and helping them push their agenda faster and better is more beneficial to RV. At least in the short term. For our long term vision, see the [Proof Chain section](#) below. There are three categories of entities that we identified as suitable for potential partnerships:
 - a. Existing Ethereum Layer 2 ZK Rollups, like zkEVM (different variants), Cairo, zkSync, etc. For these, we would provide alternative implementations for their ZK languages or virtual machines, all using a unique verifier of ZK (matching logic) proof certificates, that we would provide on Ethereum, implemented as a smart contract. The Layer 2 / ZK Rollup off-chain infrastructure would use a K semantics of their language (EVM or Cairo) piped with our unique ZK Matching Logic proof checker.
 - b. New ZK-based blockchains or solutions, like RiscZero, zkLLVM, Urbit, etc., which

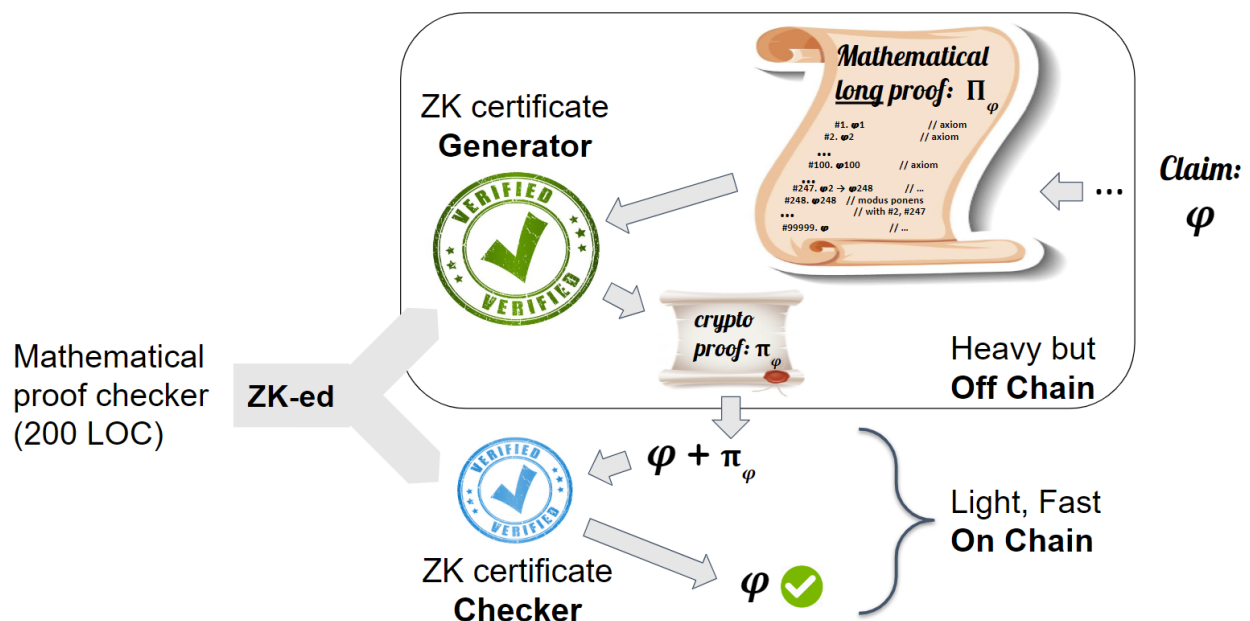
chose to not present themselves as Ethereum ZK Rollups (although there is no limitation in their design preventing them from doing it). Our proposed approach would be essentially the same for these as for the Layer 2 ZK Rollups above, except, of course, the verifier for our ZK matching logic proof checker would need to be customized for each.

- c. Existing Layer 1 blockchains, which do not have ZK rollups yet but which may want to. Here the immediate candidates are our close collaborators, namely: Cardano, Algorand, MultiversX (former Elrond). But we plan to contact more, perhaps we will even have an open call to gauge their interest. From a technological perspective, the only difference for us is the implementation of blockchain-specific verifiers for our ZK certificates.

Our goal is to form at least one partnership in 2023, in which our ZK approach will become their main ZK solution afterwards. In 2024, we aim at least at another one, but in a different category among the three categories above. And in 2025 our goal is to add at least one new partnership in each of the categories above, for a total of at least five partnerships in total in 2025 onwards.

Proof Chain — The Ultimate Layer Zero

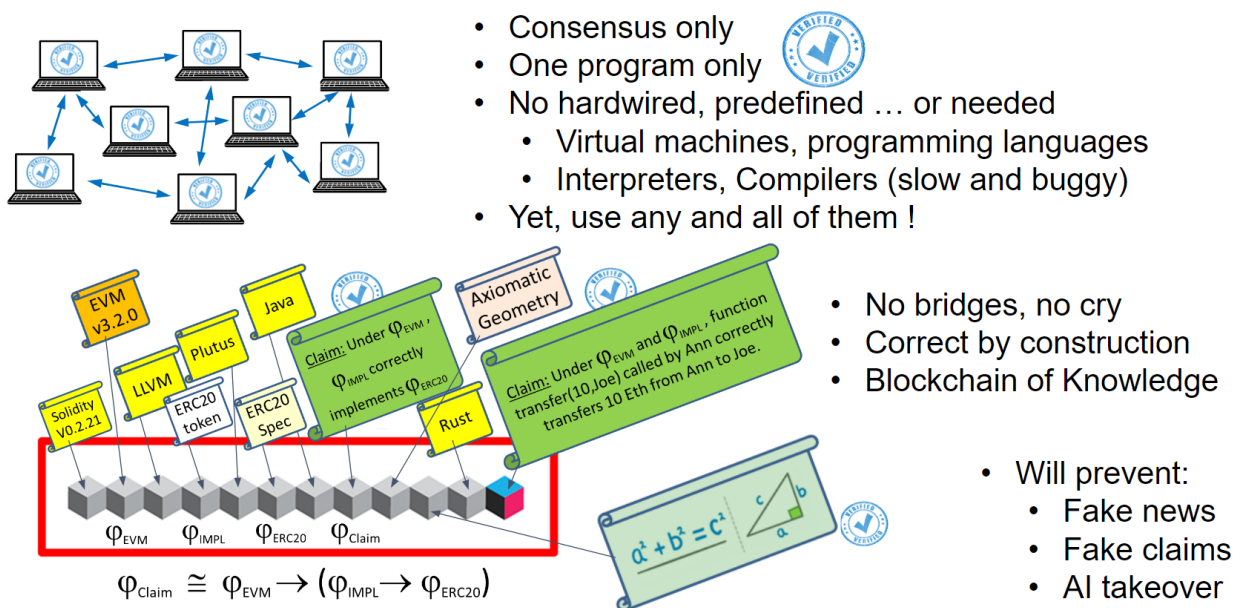
We gave a high-level introduction to our ultimate Layer Zero proposal, the Proof Chain, in a [previous section](#). Here we give more details and highlight the architecture and usage of the Proof Chain. Before discussing the benefits of the Proof Chain, it is important to understand what goes on-chain and what goes off-chain. Recall that at the core of the Proof Chain we have the 200 LOC Matching Logic proof checker, which will be implemented as a ZK circuit. Specifically, this results into two different ZK components, a common split in ZK projects where one component is off-chain and the other on-chain, as shown below:



The first component, the *ZK Certificate Generator*, is the one which takes the long mathematical proof object Π_φ of the claim φ as input, and produces the (succinct) ZK certificate π_φ as output. The ZK certificate generator component will be part of the off-chain stack and will likely be executed on fast machines in clouds and warehouses and will be offered as part of larger services by various for-profit companies, whose main role is to search for proofs of claims. Searching for proofs is hard and open ended in terms of techniques used, which is good. Good because it will stimulate competition, creativity, and new technologies to be used for a noble purpose (eg AI/GPT, automated reasoning, specialized hardware, etc).

The second component is the *ZK Certificate Checker*. Its role is to check the ZK certificates π_φ and thus confirm the validity of φ . The ZK Certificate Checker is fast and it will be incorporated in the Proof Chain's validators. In fact, validators execute only this program, the ZK Certificate Checker, as part of the overall consensus. They will run no virtual machines and know nothing about any specific programming languages. These will be part of the claims they check and will likely be pulled from their Proof Chain accounts, for checking (not execution) purposes.

The picture below illustrates how the Proof Chain operates and shows its benefits:



I next elaborate on some of the advantages of the Proof Chain compared with the state-of-the-art, and in the process of doing so I give more details on how it works:

- **Simplicity.** Proof Chain will essentially consist of only a lightweight consensus layer and only one program to be executed by validators, the ZK Certificate Checker. In particular, there will be no hardwired or predetermined virtual machine (VM) or programming language (PL). The practical importance of this degree of simplicity cannot be overestimated, because VMs and PLs usually evolve at a fast pace and thus would require the validators, i.e., the blockchain, to be regularly upgraded. No VM or PL stood the test of time unchanged. For example, EVM was so far upgraded at a rate of [2-4](#)

[times per year](#). Even PLs or VMs created and developed by top language/compiler experts, like Java and LLVM, suffered many upgrades: Java released [20 official versions](#) since its proposal in 1995 and LLVM released [16 major versions](#) since its birth in 2003 in our CS department at UIUC (by [Prof. Vikram Adve's group](#)); if these languages powered blockchains, then those blockchains would have been upgraded at least 20 and, respectively, 16 times. This is not only inconvenient and risky, but can also raise concerns with regulators, who might assume an entity in charge.

Once Proof Chain is launched, there will be no need to upgrade it! The rationale for our claim is that while PLs and VMs need to upgrade in order to keep up with the evolution of machines, programming abstractions, and education of new generations, mathematical logic does not change. So it is a better choice for blockchains. This will also simplify the interaction with regulators: there will be no entity in charge of or responsible for upgrading it. Proof Chain will be as simple as Bitcoin, but with all the additional features and benefits. Likely a commodity.

- *Generality.* In spite of having no predetermined VMs or PLs, Proof Chain will allow programs (smart contracts) to be written in virtually all PLs and to be executed either directly, by interpreting the PL, or via compilation to any VM if one chooses so. This will be possible by simply adding the desired PL or VM on the chain, as normal data in a normal account. Programs in said PL or VM are data as well, in other accounts. Any execution or correctness claims of programs in such PLs become normal claims, which will come with proofs that will be checked by the validators. Blockchains as we know them today, or subnets or sidechains, become clusters of nodes in Proof Chain, all related through a PL or a VM, or a topic of interest, or geopolitical regulations, etc. Proof Chain's mission is to provide the general infrastructure for such clusters to form.
- *Correctness / Security.* All claims on Proof Chain are checked by validators, using their lightweight builtin ZK certificate checkers, and thus all claims are provably correct. Correctness means that claims are mathematical theorems, rigorously derived from their corresponding mathematical theories. For example, the claim that an ERC20 token was correctly transferred from Ann to Joe is such a theorem, derived from the mathematical theory corresponding to the EVM semantics and the ERC20 token implementation. Or the claim that the ERC20 token implementation in EVM is correct is a theorem derived from the theory corresponding to the EVM semantics and the ERC20 specification. Proof Chain provides the infrastructure to know that claims made by any third party, trusted or not, are provably correct. The "Garbage in, Garbage out" phenomenon cannot be avoided, of course, no matter how rigorous proofs are. For example, if EVM has inconsistent semantics, or if ERC20 specification is flawed, then the proved claims are also inconsistent/flawed. See Separation of Concerns below, too.
- *Bridge Security.* This is a consequence of the above, but we list it separately due to the big concerns around bridges. On Proof Chain, no bridges as we know them will be needed in order to transfer assets from one cluster to another. Every transfer will be on the chain, with normal transactions, secure as anything else. For example, to transfer 10 ETH from an account in the "Ethereum cluster" to an account in the "Cardano cluster", one would need to sign two transactions, one burning or freezing the 10 ETH in the Ethereum cluster and another minting or unfreezing the 10 ETH on the Cardano cluster.

The underlying smart contracts on the two clusters still need to be trusted, but Proof Chain will allow stakeholders to formally verify them and post their correctness claim on the chain as well. Otherwise, users may choose to use another bridge Ethereum <> Cardano, which was formally verified.

- *Efficiency.* All the heavy lifting, that is, both searching for mathematical proofs and generating the ZK certificates for their correctness, takes place off-chain. Only checking the ZK certificates happens on-chain, which is fast and cheap. That means that once the transactions are fully proved off-chain, using any mechanisms there ranging from instrumenting interpreters to AI to search for mathematical proofs, they can be efficiently deployed on the Proof Chain. Both the off-chain and the on-chain components can take advantage of parallelism and transaction independence. The off-chain components will dictate the latency, while the on-chain component will dictate the throughput.

When it comes to applications of ZK in practice, the elephant in the room almost always is whether it will be fast enough. This is a place where we will put a lot of our effort in the near future. But that future is optimistic, for three reasons. First, there are many recent advances on the ZK front, and implementations, showing that ZK can be quite feasible. In the worst case scenario for us, we will simply implement our matching logic proof checker on top of the best of the existing ZK languages and we will get similar results to theirs; indeed, mathematical proofs are linear in size with the executions they are generated from, so there will be no complexity explosion due to generating proofs instead of executions. Second, we will take the solution above generated using the best off-the-shelf ZK compiler and dissect it and reconstruct it into a custom ZK circuit specifically crafted for Matching Logic proofs; after all, we need only one circuit, for one program, so we don't need to pay the price of generality that comes with ZK compilers. Third, although producing mathematical proofs is an inherently sequential task, which in our approach requires no ZK support, checking such proofs is an embarrassingly parallel process: the entire proof is correct iff each of its steps is correct; there is no need to wait for proving the premisses of a proof rule application, they can all be checked in parallel, yielding a perfect instance of the map-reduce paradigm.

The advantages mentioned above, namely simplicity, generality, correctness, (bridge) security, and efficiency, are undoubtedly critical for the success and mass adoption of the blockchain technology. Below I would like to mention two other advantages of the Proof Chain when compared to existing blockchains, of a more economical nature:

- *Separation of Concerns.* One of the most important principles in engineering. When separating a system into two or more well-defined components, each of the components can be improved independently, even by different teams, without having to understand or depend on unnecessary parts of the system. The Proof Chain massively separates concerns, on different levels and dimensions.

First, the Proof Chain separates the PLs, VMs and other specifications from the blockchain itself. Indeed, PLs, VMs, etc., become data (their formal semantics) stored in accounts on the blockchain. Data that is vetted (signed) by corresponding entities, such as the formal semantics of the EVM vetted by the Ethereum Foundation, or that of C vetted by the C Standards committee, or that of the ERC20 standard vetted by its

authors or some other entity or community.

Second, the Proof Chain separates the PLs and VMs from their own implementations! In particular, compilers can still be used, but are not really needed anymore, because one can execute the program (off-chain) directly with the formal semantics of the language itself, without translation to a lower-level (VM) language; in fact, this can even yield a faster overall user experience, because there are fewer execution steps that need to be proved. Also, implementations of interpreters or compilers or VMs can forgo the expensive formal verification process, which is unavoidable when these are hardwired in a blockchain that claims correctness. Indeed, these implementations will be executed off-chain and produce a proof for each execution; if that particular proof is correct, which is also verified off-chain, then it is ready to go to Proof Chain. In other words, the implementations can be buggy and still produce correct instance executions in all practical cases. We encourage the interested reader to check our [ASPLOS'21](#) paper, where we show how this principle, called [Translation Validation](#), applies to real-world compilers (from LLVM to x86).

Third, it separates responsibilities and, ultimately, blame when things go wrong. For example, if a PL or a VM has an inconsistent formal semantics, then those who vetted that language should take the blame. That may sound unfair for them, but it is better than the current state-of-the-art, where we have 20 different implementations of EVM following an informal specification that nobody is responsible for. In fact, the separation of the language semantics from its implementations will improve both: implementers will yell at language designers when semantics is unclear, so they will improve it; and once a semantics is clear, implementers can go for aggressive optimizations without worrying about other implementers following them as well to get the same results so they will not be slashed. As far as the proof checks, execution is correct. Period. Optimizations mean smaller proofs, so faster service, so happier clients for those doing them.

- *Foster Innovation.* We believe that the separation of concerns that Proof Chain offers will stimulate and inspire participants at all levels, even more than the combined blockchains today. First, and the most basic, Proof Chain can be regarded as a store of value, similar to Bitcoin. Like Bitcoin, the basic capability of Proof Chain will be to store and transfer its native token, plus to achieve consensus. Proof Chain's consensus is on claims, using one stable, simple and fast program (the ZK certificate checker). Nobody will be in charge of Proof Chain, and it should never need to be upgraded.

Second, starting a "new" blockchain, or sidechain or subnet will be achieved with one transaction, essentially adding its "theory" (the semantics of their PL, VM, DSL, specification language, rules, etc.) to Proof Chain. Dependence on particular languages or VMs will gradually disappear. Businesses can invent their own DSL targeted to their application domain, then have their clients use it right away, as if they had their own blockchain built on top of their DSL. Languages will evolve more systematically and less painfully for developers, from one formal specification to another, everything transparently on the Proof Chain.

Third, and perhaps most importantly, off-chain businesses focused on searching for proofs will flourish. Indeed, anybody and anything can produce a proof to a claim,

because the problem is well-defined and completely transparent. RV uses the K framework for that, but other companies may use different formal methods approaches, some/many/most even more automated than K. RV will have a competitive advantage initially, because K was designed in the same spirit of separation of concerns as the Proof Chain, and can already produce matching logic proof objects for certain claims across different PLs and VMs. But we believe that in 2-3 years we will see many other similar service providers, specialized on various types of claims: execution, formal verification, (un)fake news, NFTs, etc.

Fourth, and closest to my heart, Proof Chain will encourage good practices, like formal semantics, verification, formal reasoning, and mathematics education. For example, on Proof Chain, a formally verified program is not only more trustworthy, but it can also be ... faster! Consider, for example, the sum-to-n program [discussed previously](#). When we formally verify it, we come up with its loop invariant. The loop invariant can be now used to short-circuit the loop when we generate the mathematical proof. That is, instead of going through the loop 100 times and thus generating a long proof, we go through the loop only once, extract it as a lemma, then prove the main result " $\text{sum}(100) = 5050$ " using the lemma. In other words, with intelligent proof engineering, we can extract a proof of this program's execution which is much smaller than the naive proof obtained by executing the program blindly. Developers of smart contracts will now be incentivised to formally verify their code not only because that is the right thing to do, but also because their users will see faster end-to-end execution and pay less gas.

Last but not least, Proof Chain will lead to applications and innovations beyond the reach of current blockchains. For example, various mathematical theories can be formalized for educational purposes, such as Natural Numbers and Mathematical Induction, Euclidean Geometry, Algebra, Trigonometry, Group Theory, Category Theory, etc., the same way PLs or VMs are formalized. Indeed, most of these can be axiomatized using first- or second-order logic and least-fixed points, all of which subsumed by Matching Logic. From Proof Chain's perspective, there is no difference between such mathematical theories and the formal semantics of EVM, Rust, or Java. Then classic results in these domains, such as Pitagora's theorem, become claims on the chain, same as any program execution, or any transaction, or any formally verified token. I see a future where much of our mathematical knowledge will be stored on the Proof Chain. It is enough to formalize one or two of them, to serve as an example, and then enthusiasts will define the rest, because this is not only intellectually interesting, but it has an immense educational impact, teaching students how to do proofs rigorously. Importantly, this body of knowledge will be immediately available to everybody and every application, to be used to construct other claims and results. At a minimum, elementary logic on top of the elementary cryptographic infrastructure offered by the Proof Chain can be used to filter out deepfakes and fake news: the theory forming the assumptions (author of the claim, other facts it builds upon, etc.) needs to be explicitly vetted by some entity, and then the reasoning steps leading to the claim must be correct. It will put an end to journalist's favorite claim "not A implies not B" concluded from "A implies B".

Why Now and Why Runtime Verification?

UTF is all about mathematics and logic and proofs, which have been around since forever. A natural question then is why hasn't UTF been done before, even before Ethereum or even Bitcoin? Why *now*? And why us, why RV?

First, we believe that the idea underlying the UTF, namely the separation between uniformly producing mathematical proofs as justifications for claims followed by their checking using a universal ZK proof checker to produce succinct ZK certificates, is not obvious. Even if one knew all the mathematics and had all the required technology available and all of it working well. Moreover, even if the underlying idea may look somewhat natural in hindsight, to conceive it and to find the energy and motivation to work out the details, one needs a foundational understanding of computation, languages, formal executable semantics, formal verification, logic, and cryptography, paired with a very strong belief that blockchain technology is the future.

As lifelong academics specialized in the above mentioned fields at a [top research university \(UIUC\)](#), with extensive corporate experience in mission and safety critical systems even before the blockchain era and then as top-tier security auditors of blockchain systems with focus on language-independent formal semantics and verification, we believe that we were in a unique position to put such an idea and vision together. All the scientific fields and beliefs mentioned above were equally and critically important, in our view. For example, if we had a strong and rigid opinion that formal verification is exclusively about proving symbolic properties of programs, while program execution is “something else” and “not interesting”, as the traditional formal verification community thinks, then we would have missed the chance to capture verifiable computing and ZK language variants as special instances of UTF.

Second and more importantly, what made the UTF possible now and not before is the convergence of three completely different and major scientific innovations, none of each available a few years ago but all together truly available only now, starting with 2023:

- *Scalable Zero Knowledge*. While the concept of zero knowledge proofs was proposed more than 30 years ago, [in 1985](#), it was only recently that the ZK technology has been significantly advanced, to a point where it has become usable at scale. This happened thanks to a concerted effort by researchers, practitioners and investors, all motivated to a large extent by ZK's unmatched potential and benefits not only in enhancing privacy and security, but also in significantly increasing the performance and scalability of blockchains. We refer the interested reader to [Ethereum's ZK page](#) for history and learning material. Finally, it was only very recently, in 2022 and 2023, that ZK variants of languages, like zkEVM, Cairo, zkVM, zkLLVM, etc., have demonstrated the feasibility of the ZK technology to general-purpose computation.
- *Matching Logic*. Although the first paper mentioning the name “matching logic” was published [in 2009](#), it was only 10 years later, in [LICS'19](#), when we finally figured out the full strength and succinctness of the logic. The missing part that added its ultimate expressiveness was the inclusion of set variables and the μ construct for least fixed points. With these, matching logic has taken a central place in the realm of logics: any

other logic of interest, from first-order to higher-order, from untyped to dependently typed, can be captured as a *matching logic theory*; specifically, as a finite set of symbols and axioms, plus a shallow embedding allowing us to desugar all the target logic's constructs as matching logic notations and to prove all its deduction rules as matching logic theorems. Our [ICFP'20 paper](#), eg, shows how to do it for dependent type systems.

The importance of such a logic cannot be overstated: it can serve as a uniform and unique proof foundation for any other formal logic, and thus formal system, e.g., theorem provers like Coq and Lean, language frameworks like K and thus interpreters and compilers, or any program verifiers or symbolic engines or model checkers. The first Matching Logic implementation was done in 2021, but the final [200 LOC](#) implementation was only finalized in 2023. Its minimality and generality make Matching Logic ideal for UTF. As explained, Matching Logic was not an accident, but the result of a quest that took several decades. The curious reader is referred to our [papers on matching logic](#), where we explain in detail why formalisms like Hoare Logic or Structural Operational Semantics are not suitable (these are “design patterns”, to be adapted to each language, while Matching Logic is a logic, where each language becomes a theory), why first-order logic is too weak, why higher-order logic and dependent types are too complex/overkill, why separation logic is both too specific (to heaps) and a design pattern (like Hoare Logic), etc. The reader interested in what makes a logic more expressive than another via theories and notations, without adding a deep embedding layer, is referred to our [2002 paper](#) on institution morphisms and co-morphisms.

- *Proof Objects*. In theory, any formal system backed by a logical formalism can and should produce proof objects for every claim it proves. In practice, very few such systems are capable of doing it. That's because it is an unbelievably hard engineering challenge, which challenged generations of top-notch formal verification engineers. Unfortunately, not even the most prominent SMT solvers, like Z3 or CVC5, can produce such proof objects yet; there are several attempts to do it, but no complete and satisfactory solution. Proof assistant like Coq and Lean can produce proof objects for theorems they prove, but those require proof checkers of thousands of lines of code in complex languages that require compilation to be trusted, like OCaml, and to our knowledge do not generate proof objects for program execution, a critical feature for UTF (needed to capture verifiable computing and ZK language variants). To our knowledge, the first work that demonstrated the “folklore” capability to reduce program executions to mathematical proofs in a practical setting was our [CAV'21 paper](#), where we showed the first method to produce rigorous, machine checkable proof objects for program executions, based on Metamath. That work was extended in our very recent [OOPSLA'23 paper](#) to work with symbolic executions and thus arbitrary formal verification proofs based on formal semantics. Consequently, there was no satisfactory solution to proof objects as needed for UTF until 2023.

Therefore, it is the timely convergence of advances in zero knowledge technology, matching logic, and proof objects engineering that makes the UTF and the Proof Chain possible only now, in 2023. We were the creators and developers of two of these three general purpose scientific advancements, as well as the creators and developers of the [K Framework](#), which will play an instrumental role in further advancing and commercializing these initiatives. Although we cannot claim major discoveries in the realm of ZK, we were the first who proposed the idea of a

proof checker for math theorems as a ZK circuit, back in 2020. Indeed, see our [2020 K Approach and Vision](#) presentation (slide 99), where we envisioned a blockchain similar in spirit to the Proof Chain, but restricted to K. Also, through our cryptography adviser [Prof. Andrew Miller](#) and his PhD student [Bolton Bailey](#), whose research I follow closely as a member in his thesis committee, we have access to the newest and most advanced ZK research. For example, they connected us and initiated our collaboration with RiscZero, because Bolton works as an intern there on a Metamath implementation on their zkVM, jointly with Tim Carstens and others at RiscZero. We are also carrying out discussions with the teams behind all the other major ZK languages (zkEVM, zkLLVM, Cairo), to see which one is the most suitable for a first implementation of our ZK proof checker for Matching Logic. As mentioned, our long term solution will be to craft our own ZK circuit.

Operational Approach and Hiring: Three Connected Pillars

I founded RV Inc more than 12 years ago, in January 2010. During all this time, we developed our own approach and methodology for how to operate, how to do things.

The three main activities in our company are Research, Services, and Products. They are tightly interconnected and rooted in the K framework, as depicted below:



RV was initially formed to bring research done in my lab at UIUC to the real world. Now, RV's research competes with any top research lab in the world, both in academia and in industry, with research papers published in top conferences and journals, with large open source tools and products, like the K framework. Our [RV research](#) team looks into the future, as it was probably clear in this article. We try to reach far into the future, while making sure that our research produces results that can be used in our products, eventually. There is a lot of research that we need to do which cannot be used right away, but we try to minimize that. We try to focus on those research challenges that can be used or have immediate applications to our products.

How do we pick our research challenges? Actually, we don't. They pick themselves. Our Services, for example our security audits, play a key role here. When we offer services to our clients, we understand their needs and the needs of the tools that we need to offer them as products, eventually. This way the right research problems to work on take shape. Once addressed, they then get incorporated into our Products.

In general, when we hire, we look for people with strong background in formal methods, formal verification, functional programming, and/or compilers. We know from experience, both in academia and industry, that these are some of the hardest topics to master in computer science. Those who become strong in these areas tend to learn more quickly than others related topics, if interested and motivated. We were never and will never be in a desperate rush to hire people by any means. Instead, we carefully headhunted our talent and made sure they were very strong in the areas above, at the same time fitting the culture of the existing team (being friendly, helpful, and obsessed with solving hard problems). We found that such people grasped the needed blockchain and web3 technology relatively easily.

We have not attempted to explicitly hire blockchain or Web3 experts so far. Instead, we created such experts simply by exposing our engineers with their strong background to blockchain and web3 challenges, such as complex security audits. But, it takes some non-negligible time and patience. Usually, the new hires shadow experienced auditors initially. This way they not only learn the domain, but, importantly, they also face the limitations of the tools that we use internally, or tools that we want to productize eventually. After a few audits, they'll have a pretty good idea what they would like to add to the basic infrastructure. Then they switch to developing infrastructure, or products, and while doing so discover ... research challenges. Then they communicate them to the [RV research team](#). Sometimes they want to work on those research problems as well. In general, our folks have the freedom to move across different teams and areas we work on.

Our agile approach to work and team formation took us more than a decade to grow. And it works really well for our company, for the kind of challenges that we face. I'm emphasizing this because sometimes we talk to investors or other entrepreneurs who think that we should minimize Services and Research, that we should hide or even eliminate them, and only focus on Product. We understand the need for recurrent revenue products, we want those as well. However, we firmly believe that the best way to come up with the best products in this domain is actually ... to do very solid security audits and learn, from those, what the community needs in terms of products/tools.

More often than not, what is needed for smooth services is not possible without addressing some fundamental research problems. A metaphor that I like to bring here is searching for the cure for cancer: you cannot hope to find the cure for cancer if you only focus on the product; you need to do a lot of research and experiments and patient engagements in the process; it is easy to package and sell once it works, but the hard problem is to find the right product. In our case, the right products cannot be properly identified without expert led services and a lot of research.

This approach served us well so far. We were *cash positive from day one*. Very few companies, like ours, can claim that. I am very proud of our approach and of our team.

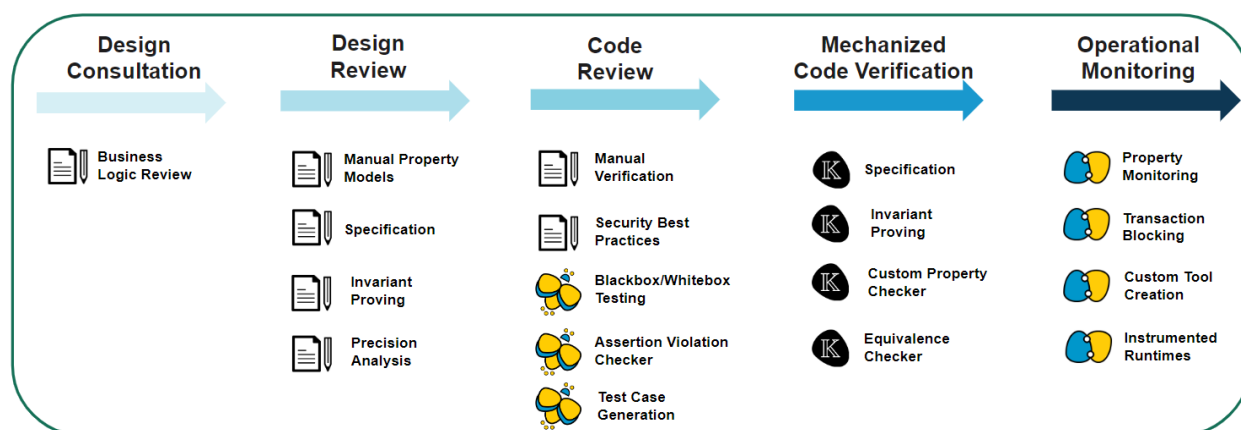
Engagement Methodology, Continuous Auditing, Product Usage

Before being a professor at UIUC, I was a researcher at NASA. In 2000-2001. One thing I quickly learned at NASA was that there was so much emphasis on the process and on how to split the resources of a project so that you increase confidence and assurance in that project.

What was unexpected, almost shocking to an immature fresh-out-of-school me back then, was that only one-fifth, 20%, of the budget of a project was allocated to code development! The rest of 80% was all allocated to verification and validation (V&V). They even had separate teams for these tasks. Once I wrote some code, so I was in the 20% budget slice for a while. Then the code moved into the V&V phase. It was passed to different teams, with different clearance levels. I was not even allowed to see my own code, once they took it over. A very rigorous process. But keep this in mind for now: at NASA, 20% development and 80% V&V.

In sharp contrast and to our disappointment, many blockchain project teams who contact us to audit their code appear to have a different mindset. They think that almost all of their project's resources should go into writing the actual code. Once written, launch it asap. Sometimes they don't even write enough tests, sometimes they even present us with no tests at all. Yet, they expect from us a "formal verification security audit" to be done quickly (a few days) and cheaply (less than \$10k) right before their launch, as a "stamp of approval" from a reputable auditor. It's simply ridiculous. Much education is clearly needed in the blockchain space, in order for devs to truly understand the need for strong methodologies and the proper use of security auditors. As a side note, we hope that our Proof Chain will wipe out the superficial players, as they will never be able to produce proof certificates of correctness this way, and users will incrementally learn that unproven protocols cannot be trusted.

Some of our clients, however, understand the need for correctness. Especially after they have been our clients, or of other reputable security auditors, previously. They start working with us early in the process, sometimes before they even write the first lines of code. Thanks to such mature clients and their high-value projects, we eventually developed what we call *continuous auditing*: we start working with the client from the very beginning and help them with everything until the end of the project, including with designing and writing the code itself; we do not write the code per se, but we "look over their shoulder" to help them write their code in a way that makes it easier to audit later on. Our ideal engagement methodology is as shown below:



We start with a design consultation, to understand the protocol business logic. We read their documentation, written usually in English on a couple of pages, and we look over their code, if any; we can work even without any code, as the point at this level is not to understand the code or find bugs in the code, but only to understand what the protocol does. Then we attempt to formalize it, either using K or on paper, using mathematics. Our more sophisticated clients are

genuinely interested in this step, as they really want to see how their business logic translates into a rigorous specification. Some confess that they had no idea that this was even possible without the code. Thus they are pleasantly surprised when they see that we can in fact identify invariants at that abstract stage and that we can even prove some useful properties.

The next stage of the engagement is to review the actual code. Our clients write their code, we don't write code for them. We may help with integration with tools like Foundry, though. If the code is not already written, we like to participate in the brainstorming and design process as if we were part of their team, to help them write the code in the best possible way ... for security audits later on. While doing that, we encourage them in the strongest possible way to use tools like Foundry and to write as many tests as they can bare, to even try to have full coverage of their smart contract code. Once we get to that stage, we all have a good understanding of the code and how it relates to the business logic, but the code is still not yet formally verified.

Now we go to the next step, mechanical code verification, where we use formal analysis and verification tools, like KEVM-Foundry; or even K itself, directly, when the properties cannot be easily formalized as Foundry parametric property tests. During this stage, we can and usually do write more properties, and use all our tools in our arsenal of tools in the K framework, to maximize our confidence in the correctness of that code. Note that only the most naive of the developers, or their media teams, claim that “their code is correct because it is formally verified”; everybody else knows that the correctness of the code is only as good as the specifications are, and that even if the specifications were perfect, errors would still surface if assumptions about the environment were violated (e.g., what if the EVM implementation, or the consensus protocol of the blockchain, is buggy?). Finally, the code gets fully verified and we write an audit report, usually as a PDF file. We hope to change that in the future, to [generate proof certificates](#).

Post-audit, the client launches the protocol and at that point, normally, the job of a security auditor ends. But with the [monitoring & recovery product](#), now we engage with our clients also post-deployment of their contracts. We can monitor, for example, the invariants that we verified statically, pre-deployment. You may wonder why monitor them if we verified them: if we verify them, then they are correct. Yes, they are correct under certain assumptions, but those may be violated during operation. Sometimes we monitor the assumptions directly, other times we monitor the actual invariant. Monitoring the invariant often implies the best recovery when violated. If the invariant is broken, for example, if a loan is under-collateralized, then typically we know what to do at that moment, how to fix the problem. Liquidate, in that case.

We developed our client engagement methodology, continuous auditing, and product usage described above over a period of many years of security audits and tool/product development. It may look standard in hindsight, but to my knowledge, it is more thorough and systematic than what we used at NASA. The constraints are also different, as all code is usually open source and there are no clearance levels in blockchain. However, blockchain is more amenable to hacker attacks than spacecraft. Needless to say that we prefer clients who understand the need for our entire stack of services and products. When we have a choice between a client who wants a quick “stamp of approval” audit, just before launch, versus a client who starts working with us several months in advance of launching and understands the need for the entire process, we obviously prefer the second client. As we get more revenue from products like KaaS and monitoring, we will be more selective with our clients, basically only picking those who go through the entire continuous auditing process described above. We will encourage the

others to use our tools and products themselves and to publish the proofs generated by these on the blockchain themselves, once the Proof Chain is available.

K Advantage: Multi-Chain, Multi-Language Audits & Tools/Products

The fact that the K framework is language parametric, or language agnostic, gave us a competitive advantage in the blockchain space. This capability of K to instantiate its generic tools with any programming language, which was there before blockchain was even a word, allowed us to rapidly bring the benefits of formal methods to multiple blockchains, and to do audits across the following blockchains and their languages:



Languages to the right are those we used in blockchain audits so far. We formalized the semantics to all of them, both the classic ones like C and the new ones lacking formal semantics. Being able to do multi-language and multi-blockchain audits should not be taken for granted! We are probably the only security company that can handle so many different blockchains and languages, thanks to our core technology, the K framework. I'm very proud of what we've achieved so far and the fact that we can truly be multi-chain and multi-language. The reader interested in specifics is encouraged to consult our audits and client engagements reachable from our [Blog](#) and from [Github](#).

Conclusion

The Universal Truth Framework (UTF) will give us the infrastructure that will allow us to know, with maximum certainty, whether claims are true. Claims can be anything provable, including everything computable, basically everything that can be derived from a set of assumptions, or axioms, using well-defined rules: code execution in any programming language or VM, formal verification or correctness claims, mathematical results in any mathematical theory, medical / aviation / automotive procedures, results produced or searched by complex AI / machines / robots, and so on. The UTF will allow and enforce any such claims producers, humans or machines, to also produce succinct proofs for them, which attest to their claims' truthfulness. The truth is the truth, it cannot be changed, and the UTF now gives us a way to certify it.

The claims proved using the UTF need to be stored sometimes, as evidence for certain actions or as basic blocks to build other claims. There is no better store for claims than a blockchain. Indeed, any of the existing blockchains with support for smart contracts can be used as Layer 1 for a ZK rollup implementing the UTF. That will provide the Layer 1 blockchain with a Layer 2 that has the usual benefits of ZK rollups (scalability, lower fees) plus the additional benefits that the UTF brings: it will support all programming languages for smart contracts, it will support arbitrary claims to be made and proved, including correctness of contracts wrt specifications, and so on. However, there is massive fragmentation in the blockchain world today, each blockchain promoting its own programming language or virtual machine, with regular upgrades and thus the inherent risk that some day things will go wrong. Moreover, blockchain interoperability is essential, but in our view it is done wrong. Can we do better?

The UTF gives us the opportunity to start fresh and do things right. To implement the ultimate Layer 0, which we call the Proof Chain. The Proof Chain will be at least as fundamental as Bitcoin and Ethereum. It will be as simple, stable and secure as Bitcoin because, like Bitcoin, the Proof Chain has only one job: to achieve consensus. It will be as versatile as Ethereum, because, like Ethereum, the Proof Chain will allow arbitrary programs to be executed. The Proof Chain completely separates computation from consensus. Computation is done off-chain, using arbitrary languages, virtual machines, powerful tools, and fast hardware. Computation, like any other proved claim, produces a ZK certificate off-chain. The Proof Chain's validators check the ZK certificate and then admit the claim. This separation of concerns, computation vs consensus, will allow users to write contracts in any programming languages, to port existing smart contracts from other blockchains, and to interoperate like never before. For example, sending a token from an EVM contract to a WASM contract is a normal transaction. No bridges needed anymore. Innovation and creativity will proliferate, because there will be no language barriers. So will education, because mathematical proofs will also go on the Proof Chain. There is no upper limit on how transformative the Proof Chain will be.

I would like to conclude by saying that the bold vision underlying the UTF and the Proof Chain would have not been possible without the K Framework. They share the same philosophy, that languages should not be hardwired in tools, and that tools should generate proofs. The main part that was missing was the ZK dimension, which now allows us to massively compress proof objects into succinct certificates. I wrote the first version of K in 2003, as a Perl translator to [Maude](#). Since then, the amazing K team has contributed more than 500k LOC in several languages, has made 6 major releases and more than 1000 minor ones, and has improved performance orders of magnitude. These days, it takes special skill and effort to manually implement an interpreter better than the one that K generates automatically. For example, KEVM outperforms most of the EVM interpreters. It took 20 years of hard work to get here. The current LLVM backend of K represents the 5th major re-implementation of K's concrete execution engine and incorporates the state-of-the-art in term rewriting and functional programming compilation. Yes, things are slow initially when you go for the ultimate, most general solution. But in my experience, it does pay off to do things right from the beginning. Performance can always be improved, and it has been. Some of K's tools still require performance improvements, but others are blazingly fast.