United States Patent	12395340
Kind Code	B2
Date of Patent	August 19, 2025
Inventor(s)	Bruestle; Jeremy et al.

Zero knowledge prover

Abstract

Methods, systems, and apparatus, including computer programs encoded on a computer storage medium, for implementing a zero knowledge prover are disclosed. In one aspect, a method includes the actions of accessing an instruction set of a processor. The actions include generating a representation of a computing instruction using Boolean logic operations. The actions include assigning a polynomial constraint of a group of polynomial constraints to each Boolean logic operation. The actions include providing, to the processor, an executable program that includes various computing instructions and a request to execute the executable program. The actions include monitoring a value of a register of the processor. The actions include determining whether the value of the register complies with polynomial constraints of the group of polynomial constraints that correspond to instructions performed on the register. The actions include determining whether the execution of the executable program by the processor has been interfered with.

Inventors: **Bruestle; Jeremy (Seattle, WA), Retford; Brian (Seattle, WA), Laub; Frank (Seattle, WA)**

Applicant: **RISC Zero, Inc.** (Seattle, WA)

Family ID: **1000008762447**

Assignee: **RISC Zero, Inc. (Seattle, WA)**

Appl. No.: **18/104754**

Filed: **February 01, 2023**

Prior Publication Data

Document Identifier	Publication Date
US 20230267195 A1	Aug. 24, 2023

Related U.S. Application Data

Publication Classification

Int. Cl.: **G06F21/00** (20130101); **G06F11/362** (20250101); **G06F21/52** (20130101); **H04L9/32** (20060101)

U.S. Cl.:

CPC **H04L9/3218** (20130101); **G06F11/3636** (20130101); **G06F21/52** (20130101); G06F2221/033 (20130101)

Field of Classification Search

CPC: H04L (9/3218); H04L (2209/34); H04L (9/50); G06F (11/3636); G06F (21/52); G06F (2221/033); G06F (11/0763); G06F (21/57); G06F (21/64)

References Cited

U.S. PATENT DOCUMENTS

Patent No.	Issued Date	Patentee Name	U.S. Cl.	CPC
7036106	12/2005	Wang	703/15	G06F 30/30
7369636	12/2007	Goko	375/370	H04L 25/45
2005/0244000	12/2004	Coleman	380/44	H04L 9/0662
2021/0255861	12/2020	Ramesh	N/A	G06F 9/30036

Primary Examiner: Shehni; Ghazal B

Attorney, Agent or Firm: Goodwin Procter LLP

Background/Summary

CROSS-REFERENCE TO RELATED APPLICATION (1) This application claims the benefit of U.S. Application 63/311,809, filed Feb. 18, 2022, which is incorporated by reference.

BACKGROUND
(1) A zero knowledge proof (ZKP) may include a method by which one party (prover) can prove to another party (verifier) that a given assertion is true while the prover avoids conveying additional information apart from the asserted fact that the statement is indeed true. A feature of ZKPs is that it is trivial to prove that one possesses knowledge of certain information by simply revealing it; the challenge may be to prove such possession without revealing the information itself or any additional information. ZKP proofs may include probabilistic proofs rather than deterministic proofs. However, there are techniques that may lessen soundness error to negligibly small values.

Description

BRIEF DESCRIPTION OF THE DRAWINGS

(1) The detailed description is described with reference to the accompanying figures, in which the left-most digit(s) of a reference number identifies the figure in which the reference number first appears. The use of the same reference numbers in different figures indicates similar or identical items.

(2) FIG. **1** is an example processor model that can be used to implement a ZKP system.

(3) FIG. **2** illustrates example code for a program in a case where the processor includes an RISC-V.

(4) FIG. **3** illustrates an example execution of the code in FIG. **2**.

(5) FIG. **4** illustrates a possible specific set of constraints for a set of verification constraints.

(6) FIG. **5** illustrates an example of the permutation checking constraints.

(7) FIG. **6** illustrates an example pseudocode for the mixing function.

(8) FIG. **7** illustrates an example where code is transformed to operate on an array of actual/expected (A/E) values so that each update sets the latest value of NE values on the prior four values.

(9) FIG. **8** is an example architecture of a ZKP system that may facilitate the creation of ZKPs to prove the correct execution by a machine in a Von Neumann architecture or an existing instruction set architecture.

(10) FIG. **9** illustrates example components involved in a ZKP system.

(11) FIG. **10** illustrates an example sequence diagram performed during a ZKP.

(12) FIG. **11** is a flowchart of generating representations of example processor instructions using Boolean logic and using those representations in an example ZKP process.

(13) FIG. **12** is a flowchart of translating polynomial constraints into an arithmetic circuit over values of a finite field and using those constraints in an example ZKP process.

DETAILED DESCRIPTION

(14) A zero knowledge proof (ZKP) system that may facilitate a creation of ZKPs to prove a correct execution by a machine in a Von Neumann architecture or an existing Instruction Set Architecture (ISA) e.g., RISC-V is described herein. The created ZKPs may include a recursive feature to prove verification of other proofs. The ZKPs may also be post-quantum secure by not relying on cryptographic primitives. In one or more embodiments, the created ZKPs may include secure mathematical transformations that can be implemented by arithmetic circuits. The secure mathematical transformation may include a set of arithmetic constraints in a proper form, which can represent the execution behavior of the machine. The ZKP system may execute the arithmetic circuits. In addition to the secure mathematical transformations, the ZKP system may further implement cryptographic strength of a proof system and various primitives (e.g., hashing algorithms) when evaluating the ZKP system. In this regard, the ZKP system may be able to prove the correct execution of a virtual processor—enabling compilable programs to be run in a provable manner.

(15) In the Von Neumann architecture, a compiler may transform a recursive ZKP algorithm or algorithm representation into an arithmetic circuit that the ZKP system can execute. The compiler may include a computer program that translates computer code written in one programming language into another language e.g., target language to create an executable program. In at least one embodiment, the ZKP system may implement a RISC-V core instruction set along with a representation of RAM when executing the arithmetic circuit. The ZKP system may further implement a processor model for a proof of system requirements and memory verifications as further described in detail below.

(16) Zero Knowledge Proofs (ZKPs)

(17) Improvements in the ZKPs may include non-interactive versions, probabilistically-checkable proofs, zk-SNARKs, and zk-STARKS, which allow the proving of arbitrary statements in NP

languages.

(18) Recursive ZKP systems may provide additional scaling properties to ZKPs. Some ZKP systems may be succinct such that as the complexity of the statement to be proven grows, verification time goes up sub-linearly. For any succinct proof system, for a sufficiently large proof, the verification of the proof may fit within the proof itself. For earlier proof systems, this 'breakeven' point was impractically large.

(19) Using recursion, the ZKP system may be used to chain proofs, verifying the prior proof and adding additional statements. Interested parties may need only to verify the head of the proof chain to prove that the entire history is correct. This allows 0(n) proof time and 0(1) verification time of arbitrarily large histories, avoiding the need for all nodes to retain the full chain history.

(20) Recursion can also be used to 'roll up' a proof of n transactions into a single transaction in parallel via a tree construction, enabling horizontal scaling of transaction proofs. Each transaction may be proven once by a single party and recursively proven up a binary tree, eventually representing the entire tree with a single proof.

(21) The core performance metrics of recursive proof systems may include the overall speed of the proof system (measured in arithmetic gates proven per second or similar), the time to prove a recursive proof, and the speed of any VMs being employed. The cryptographic strength of the proof system and the various primitives it uses (e.g., hashing algorithms) may also factor in evaluating ZKP systems.

(22) Arithmetization

(23) To implement a ZKP to prove a complex predicate, the predicate may be first converted into a mathematical form that is compatible with the underlying proof system. In the case of zk-STARKs, this may include a system of polynomial constraints over elements of a finite field. By analogy with digital circuits, this set of constraints can also be called a 'circuit', but in an arithmetic circuit each 'wire' has a value from a finite field rather than a binary value, and the 'gates' are polynomial constraints. The values being proven correct may be called the 'witness'.

(24) In the specific arithmetization used in what follows (which is a variant of AIR) the witness may be made up of various 'registers', each of which has a specific value for each 'cycle', the value being an element of a finite field. Accordingly, the full state of the witness can be considered a 2-dimensional grid or matrix of finite field elements, where each row can be a register and each column a cycle. Notationally, $r(c)$ may be used to represent the access of register r at cycle c.

(25) In at least one embodiment, the cycles used in a witness may include elements of a subgroup of the multiplicative group of the finite field used for the circuit. This subgroup may include a generator, written as ω which when multiplied by a given cycle advances it one logical time step. So to refer to register r three cycles forward of c, one may write $r(\omega^{3c})$. Because the subgroup is finite, for some N, we have $\omega^N=1$, and thus the cycles are circular in nature (that is after N forward steps we arrive again at the same cycle).

(26) The constraints of a circuit may be made invariant over the cycle, true for all cycles in the subgroup. The constraints can be further represented as a polynomial over some 'taps', where each tap represents the state of a specific register at a specific relative cycle offset. For example, a constraint forcing a register r to be 0 or 1 for all cycles might be written as:

$r(c)(1-r(c))=0$

(27) A constraint requiring a register to increase by 1000 every cycle would be written as:

$r(\omega c)=r(c)+1000$

(28) Registers can be grouped into logical 'register groups'. The underlying zero knowledge proof system may commit to registers via encoding them in a Merkle tree. Registers in separate groups can be committed to in different Merkle trees. This may allow, for example, one register group to be public, known by both the prover and the verifier, and another register group to be private, known only to the prover. Additionally, this may allow the prover to commit to one set of registers before the selection of some verifier chosen random numbers, and others after the selection.

(29) Support for non-local permutation constraints based on the mechanisms used may also be presumed, and relies on the notion of register groups above. This may allow a verification that the values of a first set of registers are a permutation of the values of some second set of registers.

(30) Recursive zk-STARK Proof System

(31) Recursive proof systems may be built on zk-SNARK based methods. The verification circuit for zk-SNARKs can be made very small, making the breakeven point for recursion reachable; however, zk-SNARK based approaches may exhibit drawbacks—the provable circuit size is smaller and the performance per gate is lower. They are not post-quantum either, owing to their reliance on elliptic curve cryptography. zk-SNARK systems cannot use ordinary hash functions such as SHA-256 due to the size of the hash circuit, relying instead on a risky family of hashes developed explicitly for finite fields.

(32) Because zk-STARK circuits are log(N) and succinct they can be used in recursive proof systems; however, these have been largely considered impractical due to the very large size of the verification circuit. Due to efficiency improvements described herein, the system introduced here may be zk-STARK based.

(33) Encoding of User Predicate

(34) In at least one embodiment, the user may be able to represent the computations that the user desires to prove in some manner. The representation of the computations may then be transformed into an arithmetic 'circuit' which the proof system executes. The efficiency of this translation is useful for many scenarios, most notably recursion.

(35) Various systems have been created to enable programmers to write circuits using EDSLs or in bespoke languages. These systems can be impractical to utilize for many applications. For example, recent efforts include designing a custom VM or emulating the Ethereum VM. While VMs may be an improvement over directly writing circuits, they can limit the audience for and applications of these systems.

(36) In one or more embodiments, a ZKP procedure may describe a complete implementation of the RISC-V core instruction set along with a representation of RAM into an arithmetic circuit. The ZKP system can be used to prove the correct execution of this virtual processor—enabling almost any compilable program to be run in a provable manner. This method may allow proofs to be written in ordinary software and for the system behavior to be reasoned about by ordinary developers. For example, code may appear to be executing on a fully secure chip, similar to an enclave; however, the security need not come from specialized hardware with secret keys, but may come from consumer hardware and provably secure mathematical transformations—the only cryptographic assumption being the safety of the chosen hash (typically SHA-256).

(37) Hashing Circuits

(38) To reduce the cost for recursive proofs, a size of the verification circuit may be reduced or minimized. A computationally significant part of verifying zk-STARK proofs may consist of cryptographic hashing. Commonly used hashes such as SHA-256 may not perform well in arithmetic circuits. This means that most existing recursive proof systems may use less standard 'field friendly' hashes such as Rescue. This may have a side effect of slowing down the non-recursive use of hashing, and in addition, lowering practical security, as the new field-friendly hashes have had much less cryptanalysis. Through careful analysis and the utilization of improved arithmetization as described herein, a suitable SHA-256 circuit may be constructed. Similar carefully developed circuits may accelerate other core parts of the verification process, enabling the zkVM ZKP circuit to be proven recursively.

(39) Specification

(40) We now describe how to efficiently instantiate a von Neumann architecture within a zero knowledge proof system. As a concrete example, we consider building an emulation of a processor along with associated RAM. In this example, we consider the processor to be of the RISC-V ISA, in part due to the simplicity of implementation, although the methods used are general and could be

applied to another architecture. To achieve this, we may represent the execution behavior as a set of arithmetic constraints of the proper form. In at least one embodiment, we represent the operation of the processor and the memory over time as an execution trace. We may then constrain the execution trace via cycle invariant polynomial constraints between the current cycle and some prior cycles.

(41) Converting Combinational Digital Circuits

(42) In one or more examples, a digital combinational circuit can be represented as a constraint system consisting of a set of registers, one for each wire in the digital circuit, and a set of polynomial constraints whose degree is at most 2.

(43) First, a register which is a value in a finite field can be constrained to 0/1 values for all of c via the constraint:

$r(c)(1-r(c))=0$

(44) Additionally, we can implement the standard logic gates via degree <2 polynomials as follows: not(r).fwdarw.(1−r(c)) and(r, s).fwdarw.r(c)s(c) or(r, s).fwdarw.1 +r(c)+s(c)−r(c)s(c)

(45) We also describe how to implement encoding/decoding from a set of registers representing bits (often used in a combinational circuit) to/from a single field element, so long as the characteristic of the field is greater than 2.sup.n, where n is the number of bits so encoded.

(46) We define registers r.sub.i, for $0 \leq i < n$, and a register s which represents the encoded value. Furthermore, we can constrain the s to be the value encoded by the bit registers r via:

(47) $\forall_i r_i(c) * (1 - r_i(c)) = 0 \quad s(c) = \underset{i}{.\text{Math.}} 2^i r_i(c)$

(48) This equation represents n separate constraints, each of degree 2, and the second constraint is of degree 1, since the values 2.sup.i are constants.

(49) Representing Constraints

(50) When representing constraints, to simplify presentation, we may use two bits of 'syntactic sugar'.

(51) First, we may allow the notion of an alias, which allows some sub-expression to be reused by multiple constraints, or even just computed into a more meaningful name. We can represent aliases using a syntax of <alias>:=<expr>, for example isValid :=isValidA(c)+isValidB(c).

(52) Second, we may allow the notion of conditional constraints. We can allow constraints to exist within a control structure similar to an 'if' in a program language, where the condition is an expression. For example, the following:

(53) TABLE-US-00001   if (cond(c)) {   val1(c) = = val2(c);   } Can be reduced to: cond(c) * (val1(c) − val2(c));

Proof System Requirement

(54) In at least one example, we may presume the proofs described by the constraints herein are instantiated as a zk-STARK. We may have a minimum requirement on the field size: for example, it can be larger than the number of cycles for the process under simulation. Additionally, we may assume that the field is as large as the word size of the processor under simulation for simplicity, however, one can simply replace any data and or address registers with multiple registers encoding ranges of bits. The field may also be sufficiently large to guarantee security under the Plonky protocol, as well as the DEEP-ALI and FRI protocols. In many cases, this requirement can be reduced by making the register trace use a subfield of a larger field, and using the larger field only in security-critical points. In addition, we presume that the protocol supports accessing multiple prior cycles and that the proof system can allow multiple register groups, encoded in different Merkle trees.

(55) FIG. **1** is an example processor model **100** that can be used to implement the ZKP system as described herein. In at least one embodiment, the processor model **100** may be a black box that can be represented by a set of combinational circuits. Each execution of an instruction may be divided into three phases such as, without limitation, decoding, computing, and applying. The decoding

may include loading of the instruction from RAM and decoding the instruction. The computing may optionally perform a data load and perform instruction computation. The applying may include updating the register state and optionally performing a data store.

(56) In one or more embodiments, the decoding and computing phases may be logically divided into two combinational parts. The first part may include computing an address (if any) to read from, and the second part may include operating on the inputs from the prior phase as well as the value returned from memory.

(57) Referencing the processor model **100**, an initial reset cycle may be run to clear the state of processor registers and establish the initial PC. During a decode cycle, the RAM may be queried to get the next instruction, which is then decoded, and used to extract the set of registers used by the computation from the full set of registers from the prior cycle. This instruction data may be read by the compute cycle to initially get the memory address (if any) to read, the results of which can be used along with the instruction data to compute new values to place into registers or write to memory. Finally, in the apply cycle, the registers may be updated based on the computed results and the prior version of the registers (read via cycle—3), and optionally memory is written to.

(58) In at least one example, memory transactions may be presumed to be full word sized and aligned. For partial writes, for example, a byte sized write, the full word that a byte resides in may be read during the compute cycle, updated with the byte to write, and then the full word can be written back in the apply cycle. In the processor model 100, each box shown in the processor logic (the lower half of the diagram) may be purely combinatorial and can be converted to constraints in a straightforward way.

(59) Memory Verification

(60) A normal processor may be represented efficiently by providing a mechanism to emulate RAM. To do this in embodiments described herein, we verify that the set of all memory transactions during execution is valid. However, absent knowing the distance in cycles between a read and write a priori, the transactions cannot be validated via normal polynomial constraints. Instead, we represent all memory transactions twice during the execution trace, once ordered by the cycle on which they occur, and concurrent with the use of that read/write, and a second time ordered first by the address and secondarily by the original cycle. This second ordering of memory transactions allows them to be verified since all accesses which relate to each other are immediately next to each other in the execution trace. In combination with the verification that the two sets of memory transactions are a permutation of one another via the Plonk mechanism, we can verify the correctness of the memory subsystem.

(61) Register Groups

(62) In one or more examples, the program code may be placed in a separate register group, called the 'code group' for which the full trace is known to both the prover and verifier. The majority of the registers used for validation are in the 'data group'. To additionally validate the permutation for Plonk, and to perform this validation after the prover has committed to the two orders of memory transactions, we may also have a third register group, called the 'acc group' for accumulation, since its job is largely to accumulate the products of the two mixed permutations. These three groups are visible in the example execution trace in FIG. **3**.

(63) Initialization, Finalization, and Code Loading

(64) In addition to normal execution, the startup of the processor and loading of the program code may be represented in a similar way. This code trace includes the code to load (which is public), and may additionally contain control registers that provide setup and teardown support. For example, the code group may begin with an 'initialization' cycle, followed by some number of load cycles, each of which writes a single instruction or static data word to its appropriate location. A 'reset' cycle prepares for execution, followed by some number of decode/compute/apply cycles., and a 'finalize' cycle verifies that the program has terminated and validates the full memory trace completed correctly. Following this, the code group consists of all zeros. Since constraints are

dependent on some element of the code group, this means that the data group and acc group can have arbitrary values for this set of cycles, enabling zero knowledge in the proof system.

(65) FIG. **2** illustrates an example code **200** for a program in a case where the processor includes an RISC-V like ISA. The example code **200** may read a word from memory address 8 and write the word to memory address **24**. The code **200** may then terminate using the mnemonic "halt." Note that halt may not be considered a standard RISC-V mnemonic. Another method to represent terminations may be used so long as the instruction decoder recognizes the termination. In the illustrated example, we presume that the code is loaded at address 0. This means that the load is for instruction 13.

(66) FIG. **3** illustrates an example execution **300** of the code **200** in FIG. **2**. Note that during the generation of a trace, the same basic structure may be followed, as the values to be used in the check section of data can be determined until the program ends. In some implementations, the execution trace **300** may illustrate the main execution phase.

(67) Main Execution Phase

(68) The example execution **300** may include a code group, data group, and an acc group. In cycle 0, the code group has 'init' set for the initialization cycle. This cycle may write a zero to the highest memory address (in data/io), which can be used as a dummy address to read from when no other memory transaction is needed by the cycle in question.

(69) Cycles 1-3 may include a 'loading' of the program from the code group into memory transactions. When the 'load' bit is set in the code group, constraints that equate IdX and ioX and set write can be activated. In one example, the constraints are:

(70) TABLE-US-00002 if (load(c)) {   ioAddr(c) = = IdAddr(c);   ioData(c) = = IdData(c); ioWrite(c) = = 1; }

(71) Cycle 4 is a reset and may clear the registers (held in 'gen' during reset and apply cycles) and perform a dummy memory read. The next 12 cycles may represent operations of the processor and can follow the logic from FIG. **1**. The 3 cycle pattern can be encoded in the code group. The memory I/O for decode may include the next instruction to decode where reads are done in "comp" and writes are performed in "apply." In a case where there is no need for a memory transaction, a dummy read can be performed. Note that upon halt, the PC may no longer advance, and in general the code group can prepare as many 3 cycles instructions as may be needed to run the program, which is not determined in advance according to this example. In a case where the program halts early, the program may stay halted until final termination at fini However, if the program is not terminated by then, the constraints of fini may fail.

(72) Cycle 17 may include the fini cycle, which can verify the prior decode was a halt, and is important for memory verification as well.

(73) Finally, some cycles may be left unconstrained to allow the data and acc groups to be padded with random values, which support the zero knowledge properties of the underlying proof system.

(74) Memory Checking

(75) In one example, the memory transactions generated during execution (i.e., "ioX") may be sorted by address and the original cycle in which they occurred. The data may be placed into "ckX," which can additionally include the cycle (which was implied via the cycle register in the code group for the equivalent ioX transactions). To verify memory correctness, the "ckAddr" may move forward, which represents a new address in the correct order, or the "ckAddr" can remain fixed while "ckCycle" moves forward. To perform the verification, the difference may be computed and show that the difference is small via decomposing it into bits. Where the field used in the proof system includes a characteristic that is more than twice as large as the maximum allowed cycles, this condition may prevent an incorrect ordering. Additionally, if the transaction is a read, then the data value can be verified to match the prior cycle. This allows writes to modify the value but ensures all following reads until the next write see this same value. A possible specific set of constraints for such a set of verification constraints is shown in FIG. **4**.

(76) Permutation Checking

(77) After the memory checking, the I/O memory transactions may be confirmed to represent a valid CPU execution and that the check memory transactions represent valid RAM. In addition, these two transactions sets may be verified to be permutations of each other. In at least one example, and following the logic of Plonk protocol, the verifier may pick a random mixing value "mix" after receiving the data group commitment, or use a Fiat-Shamir heuristic mix, a function of the root hash of the data group commit. The elements of the "io" and "ck" transactions may be mixed (with the code cycle used as the cycle for io case) using "mix." The resulting mixed values should each be a permutation of the other. These can be accumulated in the acc group, with cycle 0 (init) ignoring prior cycles, and each further cycle multiplying the new mix by the prior accumulated value. In at least one example, in the "fini" cycle, the accumulated values may be verified to be matched and/or that the accumulated value is invertible, to verify that no 0s were present during accumulation which would invalidate the products. This may probabilistically prove that the two sets of memory transactions can be permutations of each other via the logic loading out in the Plonk paper. An example of the permutation checking constraints can be seen in FIG. **5**.

(78) Recursion

(79) In one or more examples, to make a recursive ZKP, the verification of the ZKP may be represented inside a proof efficiently. Because the mechanism described above can execute arbitrary Turing complete code, it may be straightforward to implement a verifier in the proof system. Thus, one issue remaining may be one of efficiency. For example, the performance-critical operation performed during proof-verification may include the evaluation of cryptographic hashes. By accelerating this operation, the time to prove a verification can be brought to a practical level even in zk-STARKs.

(80) SHA-256

(81) FIG. **6** illustrates an example pseudocode for the mixing function. In one example, a method to encode the computation of SHA-256 may facilitate a core compression that can be computed in 72 cycles, using, for example, less than 128 registers per cycle. In this example, we note a feature of the SHA-256 mixing function. Consider the pseudocode for the mixing function shown in FIG. **6**, where most of the state variables are simply being 'shifted' to other state variables, and only A and E are computed. In at least one example, the code can be transformed to operate instead on an array of A/E values (one per cycle) so that each update sets the latest value of A/E based on the prior 4 values, as shown in FIG. **7**. Additionally, we may move the computation of the message schedule into a loop and rearrange the setup and finalization.

(82) In at least one embodiment, the resulting code may be translated into 72 cycles, where each cycle represents one value of i, and for each cycle, a value of W, A, and E may be computed. Referencing the operations within the code, in a case where each of the three registers may be represented as 32 single-bit registers, then the operations can be easily implemented given a field larger than 2^{35}. For example, the rotates and shifts simply alias bits and so are effectively no-ops. The triple XOR, the choose operation, and the majority operations are all degree 3 polynomials. We can construct the 32-bit encoded values from the bitwise versions via a linear transform and perform the addition of 32-bit values in the field directly. Extra bits may be used to handle the carry/overflow from 32 to up to 35 bits, but this accounts for less than 3 bits for each addition, or 9 registers total (fewer if we allow a larger degree). Finally, constants such as the initial values for H and the values of K can be encoded in the code group so long as SHA operations may be performed with proper alignment.

(83) In one or more embodiments, a few additional control registers may be used for example to generate memory I/O for access to M or H. If only one memory access per cycle is allowed, reads and writes to H may can be handled specially. For example, regarding the initialization via constants, H is fixed and may be read from the code group. For initialization via continuations, H may be initialized by reading back a fixed distance. For writing of H at the end of execution, 4

additional cycles may be added, or alternately, the values to write may be computed via an oracle, written in cycles 64-67, and then verified by local fixed distance access.

(84) FIG. **8** is an exemplary embodiment of a ZKP system that may facilitate the creation of ZKPs to prove the correct execution by a machine in a Von Neumann architecture or an existing ISA e.g., RISC-V. The created ZKPs may include a recursive feature to prove verification of other proofs. The ZKPs may also be post-quantum secure by not relying on cryptographic primitives. In at least one embodiment, the created ZKPs may perform secure mathematical transformations of executable instructions into arithmetic circuits. The secure mathematical transformation may include a set of arithmetic constraints in a proper form, which represent the execution behavior of the machine. The ZKP system **800** may execute the arithmetic circuits. In addition to the secure mathematical transformations, the ZKP system **800** may further implement the cryptographic strength of a proof system and various primitives (e.g., hashing algorithms) when evaluating the ZKP system. In this regard, the ZKP system **800** may be used to prove the correct execution of a virtual processor — enabling compilable programs to be run in a provable manner.

(85) In the illustrated example, the ZKP system **800** may comprise one or more components of a server or that may operate in a server environment, for example a cloud infrastructure, and including without limitation a communication interface **810**, one or more processors **820**, and a memory **830**. The memory **830** may include a compiler **840**, an SHA **850**, a datastore **860** and arithmetic circuits **870**. The compiler **840** may convert ISA (e.g., RISC-V) instructions via a mathematical transformation module **842**, which includes executable instructions for converting the instructions into the arithmetic circuits **870** as described herein. The SHA **850** represents storage of one or more secure hash algorithms such as SHA-256 described herein. The datastore **860** may include information that can be used to support the operation of the ZKP system **800**. In one example, the datastore **860** may include, for example and without limitation, configurations of arithmetic circuits, historical configurations of arithmetic circuits, and corresponding historical mathematical transformations.

(86) Communication interface **810** may include hardware, software, or a combination of hardware and software to transmit and/or receive data. Communication interface **810** may include a transceiver that facilitates wired or wireless communications through a cellular network or a broadband network. For example, the communications can be achieved via one or more networks, such as, but are not limited to, one or more of WiMax, a Local Area Network (LAN), Wireless Local Area Network (WLAN), a Personal area network (PAN), a Campus area network (CAN), a Metropolitan area network (MAN), or any broadband network, and further enabled with technologies such as, by way of example, Global System for Mobile Communications (GSM), Personal Communications Service (PCS), Bluetooth, WiFi, Fixed Wireless Data, 2G, 5G (new radio), etc.

(87) Processor(s) 820 may be a central processing unit(s) (CPU), graphics processing unit(s) (GPU), both a CPU and GPU or any other sort of processing unit(s). Each of the processor(s) **820** may have arithmetic logic units (ALUs) that perform arithmetic and logical operations as well as one or more control units (CUs) that extract instructions and stored content from processor cache memory, and then execute these instructions by calling on the ALUs as necessary during program execution. The processor(s) **820** may also be responsible for executing computer applications stored in the memory, which can be associated with common types of volatile (RAM) and/or non-volatile (ROM) memory.

(88) In one or more examples, the processor(s) **820** may use the compiler **840** in the memory **830** to prove the correct execution of the virtual processor—enabling compilable programs to be run in a provable manner. In one example, such as in RISC-V architecture, the processor(s) **820** may depend on the compiler or programmer as the knowledge of the compiler may plays a role while converting CISC code to a RISC code and, thus, the quality of the generated code depends on the compiler.

(89) Memory **830** may be implemented using computer-readable media, such as computer-readable storage media. Computer-readable media includes, at least, two types of computer-readable media, namely computer-readable storage media and communications media. Computer-readable storage media includes, but is not limited to, Random Access Memory (RAM), Dynamic Random Access Memory (DRAM), Read-Only Memory (ROM), Electrically Erasable Programmable Read-Only Memory (EEPROM), flash memory or other memory technology, Compact Disc—Read-Only Memory (CD-ROM), digital versatile disks (DVD), high-definition multimedia/data storage disks, or other optical storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other non-transmission medium that can be used to store information for access by a computing device. As defined herein, computer-readable storage media do not consist of and are not formed exclusively by, modulated data signals, such as a carrier wave. In contrast, communication media may embody computer-readable instructions, data structures, program modules, or other data in a modulated data signal, such as a carrier wave, or other transmission mechanisms.

(90) The compiler **840**, mathematical transformation module **842**, arithmetic circuits **870**, and SHA **850** may include routines, program instructions, objects, and/or data structures that perform tasks or implement particular abstract data types. In at least one embodiment, the compiler **840**, via the mathematical transformation module **842** and SHA **850**, may generate the arithmetic circuits **870** and facilitate the proving of the verification of other proofs as described herein.

(91) In some instances, the ZKP system may be implemented in a virtual machine. This implementation may be known as a zero knowledge virtual machine (zkVM) may improve the security and trustworthiness of distributed applications. The zkVM may bridge the gap between ZKPs and various programming languages such as C++ and Rust. ZKP technology may enable the output of a program to carry proof of provenance and correct execution that can be cryptographically verified by a receiver without access to the input of the program. In other words, the output of the program can be checked for correctness by a party who cannot see the inputs. This verifiability allows applications to be decentralized that previously required a trusted third party, which may improve the resilience and economics of operating computing infrastructure.

(92) ZKP techniques such as zero knowledge scalable transparent arguments of knowledge (STARKs or zk-STARKs) and zero knowledge succinct non-interactive argument of knowledge (SNARKs or zk-SNARKs) have been able to show the potential of ZKP-based computations. However, these techniques have required the adoption of new programming languages with sparse tooling support. The ZKP techniques described in this application assist in removing those barriers by bring existing languages, tools, and skills to ZKP development. This application describes a high performance ZKP prover that is used to build a zkVM that implements a standard RISC-V instruction set. Emulation of RISC-V makes compatibility possible with existing mature languages and toolchains. In other words, this provides seamless integration between "host" application code written in a high level language running natively on a host processor (e.g., Rust on arm64 Mac) and "guest" code in the same language executing inside the zkVM (e.g., Rust on RISC-V, and specifically RV32IM).

(93) A zkVM is a virtual machine that runs trusted code and generates proof that authenticate the zkVM output. Some zkVM implementations may be based on the RISC-V architecture and may execute code and produce a computational receipt. FIG. **9** illustrates example components involved in a ZKP system **900**, which may be a zkVM.

(94) Guest source code **902** may be written for the zkVM and be complied into an executable and linkable format (ELF) binary **904**. The binary **904** may be executed by the prover, which may return a computational receipt **90** to the host program. Any party with a copy of the computational receipt **908** can verify the execution of the program and read its publicly shared outputs.

(95) Before being executed by the zkVM, guest source code **902** is converted into a RISC-V ELF binary **904**. A hash of the binary **904** is used to create the method identifier **910** that uniquely

identified the binary being executed. The method identifier **910** is added to the computational receipt **908**. The binary **904** may include code instructions to publicly commit a value to the journal **914**. The contents of the journal **914** may be read by any party of device with the receipt **908**. The journal **914** may include data that is for public output.

(96) After the binary **904** is executed, an execution trace **906** contains a complete record of the zkVM operation. The trace **906** may be inspected and the instructions in the binary **904** may be compared to the operations that were actually performed. A valid trace may indicate that the binary **904** was faithfully executed according to the rules of the RISC-V instruction set architecture.

(97) The trace **906**, the journal **914**, and the method identifier **910** may be used to generate a seal **912**, which may be a blob of cryptographic data that shows the receipt is valid. The seal **912** may have properties that reveal whether itself, the method identifier **910**, and/or the journal **914** have been altered. When the receipt **908** is verified, the seal **912** will be checked to confirm the validity of the receipt **908**.

(98) In some implementations, when a method executes inside the zkVM, the zkVM may produce the computational receipt **908** along with the output. The receipt **908** may serve as a cryptographic authentication that the given method was executed faithfully. In particular, the receipt **908** may include the method identifier **910** that may serve as an identifier for a particular computation method and a seal **912** that indicates that the associated execution trace **906** satisfies the rules of the RISC-V ISA.

(99) By linking the method identifier **910** to the asserted output of the computation, the computational receipt **908** offers a powerful model for trust in software. The option to check a computational receipt **908** opens the door to verifiable computing to bring trustable software into trustless environments.

(100) The execution trace **906** may be a complete record of computations when code **902** runs on a machine. The execution trace **906** may be a snapshot of the full state of the machine at each clock cycle of the computation.

(101) In some implementations, the execution trace **906** may be a rectangular array where each row shows the complete state of the machine at a given moment in time and each column shows a temporal record of some particular aspect of the computation at each clock cycle. For example, the aspect may be the value stored in a particular address in memory or a particular RISC-V register. A line by line analysis of the trace **906** allows for a computational audit with respect to the program instructions and the underlying computer architecture.

(102) FIG. **10** illustrates an example sequence diagram **1000** performed during a ZKP. The sequence diagram **1000** illustrates the data exchanged between the prover **1002** and the verifier **1004**. The prover **1002** is executing the code and attempting to prove to the verifier **1004** that the code has executed correctly without revealing execution data of the code. In some instances, this may entail the prover **1002** preventing from revealing the contents of memory during execution of the program to the verifier **1004**.

(103) The first phase **1006** of the sequence diagram **1000** may be committing the execution trace. The execution trace may include the contents of various registers of the system and/or contents of random access memory. The prover **1002** may run a computation in order to generate the execution trace. The execution trace is organized into columns, and the columns are categorized as control columns, data columns, and permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge (PLONK) columns. The control columns handle system initialization and shutdown, the initial program code to load into memory before execution, and other control signals that may not depend on the program execution.

(104) The data columns may contain the input and the computation data, both of which are private. The columns are committed in two orderings. A first order may be in order of program execution. A second order may be by register first and clock cycle second. The reordered columns allow for efficient validation of RISC-V memory operations. The PLONK columns are used to show the

validity that the reordered data columns are a valid permutation of the original data, according to the PLONK permutation argument. After computing the data columns and the PLONK columns, the prover **1002** adds some random noise to the end of those columns in order to ensure that the protocol is zero knowledge.

(105) The prover **1002** encodes the trace. To perform the encoding, the prover **1002** may convert each column into a polynomial using an inverse number theoretic transformation. These may be trace polynomials. The prover **1002** may evaluate the data polynomials and the control polynomials over an expanded domain and commit the evaluations into two separate Merkle trees (**1008**). Using these Merkle roots as an entropy-source, the system may, in some implementations, use Fiat-Shamir to choose PLONK mixing parameters, using a secure hash algorithm 2 cryptographic random number generator (SHA-2 CRNG) (**1010**).

(106) The prover **1002** may use the PLONK mixing parameters to generate the PLONK columns, interpolate them to form the PLONK polynomials, evaluate those polynomials over a larger domain, and commit those evaluations to a Merkle tree. The prover **1002** transmits the Merkle root of each tree to the verifier **1004** (**1012**). Using these three Merkle roots as an entropy source, the system may use Fiat-Shamir to choose a constraint mixing parameter, alpha, using SHA-2 CRNG.

(107) The second phase **1014** of the sequence diagram **1000** may be validating the execution trace. The prover **1002** may use the constraint missing parameter, the trace polynomials, and the rule checking polynomials to conduct one or more low degree validity polynomials. To do this, the prover **1002** may write k publicly known rule checking polynomials in terms of the private trace polynomials to generate k constraint polynomials. One of the aspects of these polynomials is that for each of the k rules and each input z that is associated with the trace, a given constraint polynomial will return zero if the trace effectively "passes the test."

(108) Using the constraint mixing parameter alpha, the prover **1002** combines the constraint polynomial into a single mixed constraint polynomial by computing the sum of the products of pairs of the constraint mixing parameter alpha and the constraint polynomial. In some implementations, if each constraint polynomial returns zero at some point z, then the mixed constraint polynomial will return zero at z.

(109) Using publicly known zeros polynomials, the prover **1002** computes the high degree validity polynomial which is the mixed constraint polynomial divided by the zeros polynomial. The zeros polynomial may be a divisor of any honest construction of the mixed constraint polynomial. In other words, an honest prover will construct the high degree validity polynomial to be a polynomial of lower degree than the mixed constraint polynomial. This may be referred to as the high degree validity polynomial being a high degree" relative to the trace polynomials.

(110) The prover **1002** splits the high degree validity polynomial into, in some instances, four low degree validity polynomials. The prover **1002** evaluates the low degree validity polynomials, encodes them in a Merkle tree, and sends the Merkle root to the verifier **1004** (**1018**). In some implementations, the system may use Fiat-Shamir to choose the domain extending for eliminating pretenders (DEEP) test point.

(111) The third phase **1020** of the sequence diagram **1000** may be performing the DEEP technique. The verifier **1004** may be attempting to check the asserted relation between the mixed constraint polynomial, the zeros polynomial, and the high degree validity polynomial. In other words, the verifier **1004** would like to confirm that the mixed constrain polynomial is the product of the zeros polynomial and the high degree validity polynomial.

(112) The prover **1002** transmits the evaluations of each of the low degree validity polynomials evaluated at z, which allows the verifier **1004** to compute the high degree validity polynomial at z. Computing the mixed constraint polynomial at z may be more involved. Because rule check can check relationships across multiple columns and multiple clock cycles, evaluating the mixed constraint polynomial at z may require evaluations of the trace polynomials at various factors of z. The various factors of z may be related to selected column and cycle. The prover **1002** transmits

these necessary evaluations of each mixed constraint polynomial to the verifier **1004** so the verifier **1004** may be able to evaluate the mixed constraint polynomial. These calculations may be referred to as "necessary evaluations" of a respective trace polynomial at the various factors of z may be related to selected column and cycle or the taps of the respective trace polynomial at z.

(113) With this information, the verifier **1004** may now be able to check that the that the mixed constrain polynomial is the product of the zeros polynomial and the high degree validity polynomial. In some implementations, these asserted evaluations may not have associated Merkle branches. The DEEP technique may offer an alternative to the Merkle proof.

(114) The prover **1002** may construct the DEEP polynomial using the taps of the respective trace polynomial at z. The DEEP polynomial may be defined as P′.sub.i(x)=(P.sub.i(x)-P.sub.i(x))/((x−x.sub.1) . . . (x−x.sub.n)) where P.sub.i(x) is the trace polynomial and .sup._P.sub.i(x) is the polynomial formed by interpolating the taps of the trace polynomial. The prover **1002** computes the DEEP polynomial for a given i, runs an inverse number theoretic transformation on the result and sends the coefficients of the DEEP polynomial to the verifier **1004**. Using this technique, the prover **1002** constructs and sends a DEEP polynomial for each of the DEEP polynomial for each/and each low degree validity polynomial.

(115) At this point, the claim of trace validity has been reduced to the claim that each of the DEEP polynomials is actually a low degree polynomial. To conclude the proof, the prover **1002** mixes the DEEP polynomials into the Fast Reed-Solomon Interactive Oracle Proofs of Proximity (Fast RS IOPP (FRI)) polynomial using a DEEP mixing parameter and using the FRI protocol to show that the FRI polynomial is a low degree polynomial.

(116) FIG. **11** is a flowchart of generating representations of example processor instructions using Boolean logic and using those representations in an example ZKP process **1100**. The process **1100** accesses an instruction set for a processor and generates a representation of each instruction using Boolean logic. The process **1100** generates a polynomial constraint for each Boolean logic operation. During the ZKP process, the process **1100** uses the constraints to determine whether the registers of a process complies with the constraints. The process **1100** will be described as being performed by the system **800** of FIG. **8** and will include references to components of FIG. **8**. In some implementations, the process **1100** may be performed by the processor executing the monitored executable program. In some implementations, the process **1100** may be performed by another processor monitoring the processor executing the monitored executable program.

(117) The system **800** accesses an instruction set of a processor (**1110**). In some implementations, the instruction set of the processor is associated with a von Neumann architecture. In this instance, the system **800** monitors the processor, which is included in a system with a von Neumann architecture. In some implementations, the instruction set of the processor is the RISC V instruction set architecture.

(118) For each instruction of the instruction set, the system **800** generates a representation of the instruction using Boolean logic operations (**1120**). In some implementations, the Boolean logic operations include a not operation, an and operation, and an or operation. The system **800** assigns a polynomial constraint of a group of polynomial constraints to each Boolean logic operation (**1130**). In some implementations, the group of polynomial constraints are encoded in one or more arithmetic circuits.

(119) The system **800** provides, for output to the processor, an executable program that includes various instructions of the instruction set and a request to execute the executable program (**1140**). The system **800** may determine whether execution of the program was interfered with during executing by the processor. Malware, viruses, and/or other malicious software may cause interference with the execution.

(120) During execution of the executable program, the system **800** monitors a value of a register of the processor (**1150**). In some implementations, the system **800** monitors the value of an additional register of the processor. In some implementations, the register and/or the additional register are

private. Outside parties and devices are unable to view the values in private registers. In some implementations, the register and/or the additional register are public. Outside parties and devices are able to view the values in public registers. In some implementations, the value of the register and/or the additional register is constrained to a finite field.

(121) In some implementations, the polynomial constraint assigned to the not operation is related to the register. In some implementations, the polynomial constraint assigned to the and operation is related to the register and the additional register. In some implementations, the polynomial constraint assigned to the or operation is related to the register and the additional register. The polynomial constraint is related to a register if the polynomial constraint references the value of the register. Some polynomial constraints may reference more than one register.

(122) The system **800** determines whether the value of the register complies with polynomial constraints of the group of polynomial constraints that correspond to instructions performed on the register (**1160**). In some implementations, the system **800** determines whether the value of the additional register complies with the polynomial constraints of the group of polynomial constraints that correspond to instructions performed on the additional register.

(123) Based on determining whether the value of the register complies with the polynomial constraints, the system **800** determines whether the execution of the executable program by the processor has been interfered with (**1170**). In some implementations, the system **800** determines that the value of the register complies with the polynomial constraint. In this case, the system **800** determines that the execution of the executable program by the processor has not been interfered with. In some implementations, the system **800** determines that the value of the register does not comply with the polynomial constraint. In this case, the system **800** determines that the execution of the executable program by the processor has been interfered with.

(124) FIG. **12** is a flowchart of translating polynomial constraints into an arithmetic circuit over values of a finite field and using those constraints in an example ZKP process **1200**. The process **1200** describes the translation of polynomials constraints to an arithmetic circuit over values of a finite field. The process **1200** stores the value in a set of registers of a processor. The process **1200** stores the progression of time of a register as a time series of cycles. The process **1200** will be described as being performed by the system **800** of FIG. **8** and will include references to components of FIG. **8**. In some implementations, the process **1200** may be performed by one or more processors that are executing the monitored software program. In some implementations, the process **1200** may be performed by another one or more processors that monitors the one or more processors executing the monitored software program.

(125) The process **1200** utilizes one or more processors that include a set of n registers and that are executing a software program. The process **1200** also utilizes a clock generator that generates a clock signal with clock cycles. The one or more processors may implement a change detector.

(126) The change detector generates, during each clock cycle, a hash of each register of the set of n registers, wherein each hash is consistent with one or more polynomial constraints (**1210**). In some implementations, a portion of the set of n registers are public and a remaining portion of the set of n registers are private. In some implementations, generating the hash of each register includes accessing a value of the register by calculating a cycle offset that is n cycles away from a current cycle by applying an offset n times. In some implementations, the change detector is configured to store the hashes of each of the clock cycles in a Merkle tree.

(127) The change detector stores a time series of the hashes indexed by the clock cycles (**1220**). In some implementations, the time series of the generated hashes are chained. The system **800** receives at least one of the polynomial constraints (**1230**). In some implementations, the at least one of the polynomial constraints is restricted over values of a finite field. In some implementations, the least one of the polynomial constraints is restricts values to a finite field. In some implementations, the size of the finite field corresponds to a size of a word of the one or more processors. In some implementations, each bit of the word of the one or more processors

corresponds to a register of an arithmetic circuit. In some implementations, the size of the finite field does not correspond to a size of a word of the one or more processors. In some implementations, the one or more processors are configured to utilize register swapping techniques. (128) The system **800** encodes the at least one of the polynomial constraints (**1240**). The system **800** samples at least one of the hashes (**1250**). The system **800** determines whether the sampled hash is consistent with the time series (**1260**). Based on determining whether the sampled hash is consistent with the time series, the system **800** determine whether the execution of the software program by the one or more processors has been interfered with (**1270**).

(129) In some implementations, the system **800** determines that the sampled hash is not consistent with the time series. In this case, the system **800** determines that the execution of the software program by the one or more processors has been interfered with. In some implementations, the system **800** determines that the sampled hash is consistent with the time series. In this case, the system **800** determines that the execution of the software program by the one or more processors has not been interfered with.

(130) In some implementations, the system **800** determines whether the execution of the software program by the one or more processors has been interfered with based on a zero knowledge proof using the remaining portion of the set of n registers that are private. In some implementations, the zero knowledge proof uses zk-STARKS. In some implementations, the **800** determines whether the execution of the software program by the one or more processors has been interfered with based on the Merkle tree.

(131) Although the subject matter has been described in language specific to structural features and/or methodological acts, it is to be understood that the subject matter defined in the appended claims is not necessarily limited to the specific features or acts described above. Rather, the specific features and acts described above are disclosed as example forms of implementing any claims based on this disclosure.

(132) Although a few implementations have been described in detail above, other modifications are possible. In addition, the logic flows depicted in the figures do not require the particular order shown, or sequential order, to achieve desirable results. In addition, other actions may be provided, or actions may be eliminated, from the described flows, and other components may be added to, or removed from, the described systems. Accordingly, other implementations are within the scope of the following claims.

Claims

1. A system comprising: one or more processors comprising a set of n registers and that are executing a software program; a clock generator that generates a clock signal with clock cycles; and memory including a plurality of computer-executable components that are executable by the one or more processors to perform acts, the plurality of computer-executable components comprising: a change detector that is configured to: generate, during each clock cycle, a hash of each register of the set of n registers, wherein each hash is consistent with one or more polynomial constraints; and store a time series of the hashes indexed by the clock cycles, wherein the acts comprise: receiving at least one of the polynomial constraints; encoding the at least one of the polynomial constraints; sampling at least one of the hashes; determining whether the sampled hash is consistent with the time series; and based on determining whether the sampled hash is consistent with the time series, determining whether the execution of the software program by the one or more processors has been interfered with.

2. The system of claim 1, wherein: determining whether the sampled hash is consistent with the time series comprises determining that the sampled hash is not consistent with the time series, and determining whether the execution of the software program by the one or more processors has been interfered with comprises determining that the execution of the software program by the one or

more processors has been interfered with based on determining that the sampled hash is not consistent with the time series.

3. The system of claim 1, wherein: determining whether the sampled hash is consistent with the time series comprises determining that the sampled hash is consistent with the time series, and determining whether the execution of the software program by the one or more processors has been interfered with comprises determining that the execution of the software program by the one or more processors has not been interfered with based on determining that the sampled hash is consistent with the time series.

4. The system of claim 1, wherein the time series of the generated hashes are chained.

5. The system of claim 1, wherein the at least one of the polynomial constraints is restricted over values of a finite field.

6. The system of claim 5, wherein: a size of the finite field corresponds to a size of a word of the one or more processors, and each bit of the word of the one or more processors corresponds to a register of an arithmetic circuit.

7. The system of claim 5, wherein: a size of the finite field does not correspond to a size of a word of the one or more processors, and the one or more processors are configured to utilize register swapping techniques.

8. The system of claim 1, wherein a portion of the set of n registers are public and a remaining portion of the set of n registers are private.

9. The system of claim 8, wherein determining whether the execution of the software program by the one or more processors has been interfered with is further based on a zero knowledge proof using the remaining portion of the set of n registers that are private.

10. The system of claim 9, wherein the zero knowledge proof uses zk-STARKS.

11. The system of claim 1, wherein generating the hash of each register comprises: accessing a value of the register by calculating a cycle offset that is n cycles away from a current cycle by applying an offset n times.

12. The system of claim 1, wherein: the change detector is configured to store the hashes of each of the clock cycles in a Merkle tree, and determining whether the execution of the software program by the one or more processors has been interfered with is further based on the Merkle tree.

13. A method, comprising: accessing, by a computing device, an instruction set of a processor; for each instruction of the instruction set, generating, by the computing device, a representation of the instruction using Boolean logic operations; assigning, by the computing device, a polynomial constraint of a group of polynomial constraints to each Boolean logic operation; providing, for output by the computing device and to the processor, an executable program that includes various instructions of the instruction set and a request to execute the executable program; during execution of the executable program, monitoring, by the computing device, a value of a register of the processor; determining, by the computing device, whether the value of the register complies with polynomial constraints of the group of polynomial constraints that correspond to instructions performed on the register; and based on determining whether the value of the register complies with the polynomial constraints, determining, by the computing device, whether the execution of the executable program by the processor has been interfered with, wherein the Boolean logic operations comprise a NOT operation, an AND operation, and an OR operation, and wherein: the polynomial constraint assigned to the NOT operation is related to the register, the polynomial constraint assigned to the AND operation is related to the register and an additional register, and the polynomial constraint assigned to the OR operation is related to the register and the additional register.

14. The method of claim 13, wherein the group of polynomial constraints are encoded in one or more arithmetic circuits.

15. The method of claim 13, wherein: determining whether the value of the register complies with the polynomial constraints comprises determining that the value of the register complies with the

polynomial constraint, and determining whether the execution of the executable program by the processor has been interfered with comprises determining that the execution of the executable program by the processor has not been interfered with based on determining that the value of the register complies with the polynomial constraints.

16. The method of claim 13, wherein: determining whether the value of the register complies with the polynomial constraints comprises determining that the value of the register does not comply with the polynomial constraint, and determining whether the execution of the executable program by the processor has been interfered with comprises determining that the execution of the executable program by the processor has been interfered with based on determining that the value of the register does not comply with the polynomial constraints.

17. The method of claim 13, wherein the instruction set of the processor is associated with a von Neumann architecture.

18. The method of claim 13, wherein the instruction set of the processor is RISC V.

19. The method of claim 13, comprising: during execution of the executable program, monitoring, by the computing device, a value of the additional register of the processor; and determining, by the computing device, whether the value of the register and the value of the additional register comply with other polynomial constraints of the group of polynomial constraints that correspond to instructions performed on the register and the additional register, wherein determining whether the execution of the executable program by the processor has been interfered with is further based on determining whether the value of the register and the value of the additional register comply with other polynomial constraints.

20. The method of claim 13, wherein the value of the register is constrained to a finite field.

21. A system comprising: one or more processors comprising a set of n registers; one or more monitoring processors that are configured to monitor the one or more processors; a clock generator that generates a clock signal with clock cycles; and memory including a plurality of first computer-executable components that are executable by the one or more monitoring processors to perform first acts comprising: accessing an instruction set of the one or more processors; for each instruction of the instruction set, generating a representation of the instruction using Boolean logic operations; assigning a polynomial constraint of a group of polynomial constraints to each Boolean logic operation; providing, to the one or more processors, an executable program that includes instructions of the instruction set and a request to execute the executable program; during execution of the executable program by the one or more processors: generating, during each clock cycle, a hash of each register of the set of n registers, wherein each hash is consistent with one or more polynomial constraints; and storing a time series of the hashes indexed by the clock cycles; sampling at least one of the hashes; determining whether the sampled hash is consistent with the time series; and based on determining whether the sampled hash is consistent with the time series, determining whether the execution of the executable program by the one or more processors has been interfered with.

22. The system of claim 21, wherein: determining whether the sampled hash is consistent with the time series comprises determining that the sampled hash is not consistent with the time series, and determining whether the execution of the software program by the one or more processors has been interfered with comprises determining whether the sampled hash is not consistent with the time series.

23. The system of claim 21, wherein: determining whether the sampled hash is consistent with the time series comprises determining that the sampled hash is consistent with the time series, and determining whether the execution of the software program by the one or more processors has been interfered with comprises determining that the execution of the software program by the one or more processors has not been interfered with based on determining that the sampled hash is consistent with the time series.