Version 1.01

The Core of CVTS

Augusto Teixeira Diego Nehab

Abstract

CVTSis a layer-2 platform for the development and deployment of scalable decentralized applications. CVTSDApps are composed of both blockchain and off-chain components. Off-chain components run inside *CVTSNodes* that represent the interests of each DApp user. CVTSNodes provide DApp developers with reproducible *CVTSMachines*, where large scale verifiable computations can be run. These verifiable computations are easily integrated into smart contracts by powerful primitives that provide strong conflict-resolution guarantees. More precisely, any dispute arising over the result of computations run inside CVTSMachines can be fairly adjudicated at negligible cost on the blockchain. CVTSNodes also allow DApp developers to run native code. Native computations can leverage the node's full processing power, including any available GPUs. Whether performed natively by the node or inside CVTS Machines, off-chain components run under a complete Linux operating system that provides the full ecosystem required by complex computations. CVTSenables DApp developers to use all the programming languages, tools, libraries, software, and services they are already familiar with. By moving most of the complex logic of their DApps to portable off-chain components, developers are freed from the limitations and idiosyncrasies imposed by blockchains. In this way, CVTSempowers developers to select the best run-time environment in which to host each part of their DApps.

1 Introduction

Public blockchains are mechanisms through which networks can maintain decentralized consensus over a shared state. Typically, this state holds, among other data, a payment system. The stake held by participants in the resulting economy works as their incentive for making the state widely available to others and for rejecting invalid transactions. In this virtuous cycle, the payment system is built on top of the decentralized consensus, which only functions due to incentives created by the payment system itself. Both the payment system and the consensus can then be used for other purposes.

As new applications for blockchain technology are envisioned, the demands on the underlying infrastructure are constantly increasing. At the moment, the two major obstacles to widespread adoption of blockchain technology are its poor scalability and lack of a solid development environment. The main contribution of CVTSto the blockchain ecosystem is overcoming both these issues.

Scalability Currently deployed consensus mechanisms are based on full redundancy [Nakamoto 2009; Wood 2018]. They require every transaction to be stored permanently and to be validated by every participant. This inefficiency is the key limiting factor to the growth of the transaction rate, the amount of data involved, and the intensity of computations within transactions. High transaction costs and increased latency have become a barrier to many innovative applications that would otherwise benefit from the flexibility that smart contracts bring to the blockchain.

Attempts to improve blockchain scalability can be divided into *layer 1* and *layer 2* solutions. Layer 1 scalability solutions change the underlying blockchain infrastructure itself. Examples include the optimization of block sizes, sharding, and Delegated Proof of Stake (DPoS). Because they operate at the infrastructure level, these solutions are burdened by the requirement of preserving global consensus. Certain aspects of the state, such as the payment system, are of critical importance to all parties and therefore require global consensus. Otherwise, for most interactions mediated by the blockchain, it is perfectly safe to limit access and verification responsibility to the few parties that can potentially be affected. The blockchain can then be used to provide finality and to guarantee local consensus in the rare occasions where a dispute arises between these parties. In other words, global consensus is a precious resource that should be used with parsimony. In recognition of this fact, layer-2 scalability solutions such as plasma, side chains, TrueBit, or state channels move as much data and computation as possible off-chain. Layer-1 and 2 scalability solutions are discussed at some depth in section 2.

Computation environment Every computation that can influence a transaction, whether performed on-chain or off-chain, must be reproducible by all parties playing a validating role. Reproducible computational models must be self-contained and deterministic. In other words, the complete state for the computation and the entire sequence of modifications to this state must be fully specified and agreed upon. Sadly, real computing architectures were not designed with these constraints in mind, and therefore are not reproducible. Blockchain platforms solve this problem by employing custom virtual machines (VMs) when processing smart contracts. These VMs are reproducible, but also domain specific. On the one hand, they offer native support for features useful to smart contracts (e.g., accounting, rollback, associative memory, authentication, cryptography etc). On the other hand, they lack valuable features found in general-purpose architectures (e.g., floating-point arithmetic, virtual memory, interrupts etc).

The revolution in software capability the world experienced over the last few decades can be attributed to two key factors. The first is an exponential increase in the speed at which modern hardware platforms can process vast amounts of data. The second, and equally important, is the ever-increasing expressive power of software development environments. Indeed, general-purpose computations do not happen in isolation. Rather, they are assembled from inter-dependent building-blocks created by a worldwide collaboration of software developers. These components and services rely on standard-library facilities hosted by an underlying operating system (memory management, process management, file systems, networking, etc). It is the operating system that ties everything together. Such facilities are not available from the free-standing programming languages and compilers that typical blockchains offer to smart contract developers.

Reproducibility and scalability concerns have made on-chain computation environments very restrictive. To boost productivity and widen the scope of blockchain development, we need a reproducible computation model that supports modern operating systems.

CVTS

This paper describes CVTS, a layer-2 platform for the development and deployment of scalable decentralized applications. CVTS DApps are *hybrid*, i.e., they include both blockchain and off-chain components.

The off-chain component runs in a network of *CVTSNodes* (section 6), each representing the interests of a DApp user. The off-chain component is further divided into two modalities. *Native* computations run directly in the host hardware. Although native computations have access to the node's full processing power (including GPUs), the computations are not reproducible, at least not a pri-

ori. *Reproducible* computations run instead inside *CVTSMachines* that are controlled by the CVTSNode. These are general, fully self-contained Linux systems, that run on a deterministic RISC-V architecture (section 3). Nodes interact with CVTSMachines by means of a well-defined *host interface* (section 4).

Within the blockchain, a CVTSDApp can specify reproducible off-chain computations to be performed over large amounts of off-chain data (section 5). CVTSNodes can automatically follow these specifications to perform the computations off-chain. DApp developers can instruct the nodes to submit results or verify and dispute results submitted by others. From the blockchain's perspective, undisputed computations take negligible resources. Even in case of disputes, the settlement cost is only the logarithm of the storage and time required during the computation. Off-chain, CVTSNodes never experience more than twice the space and time required by the computation. In this way, CVTSvirtually eliminates the gap in storage and computation power between smart contracts and traditional computer programs.

Moving computations off-chain brings several advantages beyond scalability. CVTSMachines enable DApp developers to use all the programming languages, tools, libraries, software, and services they are already familiar with. Moreover, the way in which computations are formulated is agnostic to the underlying blockchain. By isolating all the complex smart-contract logic into reproducible off-chain computations, developers can make their DApps more portable across different blockchains.

The focus of this document is the core of CVTS. It includes the full specification for the CVTSMachine, the host interface for controlling it, the blockchain interface for specifying complex off-chain computations, and the CVTSNode interface for performing and verifying these computations. Higher-level tools, interfaces, and a variety of use cases built on top of this core functionality will be described in a future document [Teixeira and Nehab 2019a]. Detailed documentation on all interfaces, as well as the development environment for CVTSNodes and CVTSMachines will be available from the CVTSSDK [Teixeira and Nehab 2019b].

2 Related work

The work most closely related to CVTSis TrueBit [Teutsch and Reitwießner 2017]. The connection between CVTSand Truebit comes from the fact that both technologies move intensive computations off-chain and then, within the blockchain, use a verification game [Feige and Kilian 1997] to efficiently settle disputes regarding the results of these computations. Despite this similarity, many other design decisions set these two technologies apart.

TrueBit is based on WebAssembly [2018], a VM ISA designed by a W3C Community Group to support efficient web applications.[1] In contrast, CVTSis based on RISC-V [Waterman and Asanovic´ 2017a,b], an open ISA designed in UC Berkley for implementation by native hardware. The WebAssembly and RISC-V ISAs are of similar complexity. The key difference is their position in relation to applications and the operating system. WebAssembly was designed to sit *between* applications and the underlying operating system. RISC-V is instead meant to sit *under* the operating system and the applications it supports. TrueBit's choice is consistent with a focus on extending the computational power of smart contracts, which tend to operate under severely restricted environments. Real-world applications, however, cannot exist in isolation. They depend on rich, complex run-time environments that are invariably built on top of a modern operating system. To give developers of decentralized applications access to the tools, libraries, services, and software they are already familiar with, CVTSchose to support Linux. A realistic ISA, such as RISC-V, is much better suited for this purpose.

One of the differences of greatest consequence is in how CVTS aligns the interest in off-chain computations with the responsibility for their execution. In TrueBit, there is no such alignment. A smart contract posts the computation to a pool of untrusted parties and waits for one of them to perform it off-chain and post the result back. In this sense, TrueBit can be seen as a means for increasing the computational power of individual smart contracts. Cheating is prevented with a complex incentive layer that rewards pool members for successively disputing incorrect results. To keep the members engaged, computations with incorrect results must be artificially injected by the incentive layer. This inefficiency is a fundamental part of the design of TrueBit. Conversely, CVTScan be seen as a way for off-chain computations to be endorsed by a smart contract. All parties that could be affected by this endorsement are responsible for performing the computation off-chain and, if needed, starting a dispute. Although the ensuing verification can be outsourced to a dispute resolution market (see section 6.2), there is no built-in inefficiency and no need for an incentive layer.

The large storage requirements of real-world computations pose a significant challenge that TrueBit does not address. Explicit representations of code and data do not fit within the blockchain. Instead, a CVTSMachine, together with its code and data, are represented on-chain by a hash of its state. This arrangement allows for complex transactions built from several rounds of off-chain computations to be fully specified. The states themselves are only ever known explicitly off-chain, by interested parties. Some applications can face data availability issues to which CVTSoffers a range of original solutions [Teixeira and Nehab 2019a].

Finally, CVTSis committed to making off-chain computations portable across different blockchain platforms.

2.1 Other related technologies

New blockchain technologies emerge at such a high rate that any attempt at a comprehensive survey is doomed to become obsolete before it is even published. Nevertheless, some general trends merit discussion. Specific examples cited in the discussion should be seen as representatives of entire categories, rather than exhaustive lists.

Layer-1 scalability solutions The *scalability trilemma* put forth by Buterin [2018a] poses that no simple workaround can improve transaction throughput without reducing the decentralization or security of a blockchain. On the one hand, requiring higher throughput from nodes raises their operating cost, which centralizes power in the hands of the few players that can afford them. On the other, fragmenting consensus into independent chains makes each of them more vulnerable to 51% attacks.

The two leading proposals for breaking the trillema are refinements of these two ideas. *Delegated Proof of Stake* (e.g., Larimer [2017]) requires a small number of powerful nodes to validate every single transaction. However, these nodes are democratically elected by all parties that hold a stake in the blockchain. The alternative proposal is to split the state and history into multiple *shards* that are verified independently [Buterin 2018b]. These shards are then connected to a main chain in a way that reduces the odds of successful attacks.

Both ideas are expected to significantly improve the throughput of transactions processed by the blockchain. They must nevertheless achieve universal consensus among nodes on each transaction. This imposes a super-linear global cost as computation sizes grow.

[1] It was originally based on the LLVM back-end for an obscure Myricom NIC embedded processor design used internally at Google.

CVTS, on the other hand, is designed to achieve local (instead of universal) consensus over its computations. More precisely, only affected parties are required to perform the computation, while any possible dispute among them can be resolved in logarithmic time. This design allows for intensive computations to be performed off-chain only by the impacted participants, who still enjoy strong conflict resolution guarantees.

In this way, DPoS and sharding are orthogonal to CVTS. CVTS benefits from the faster and cheaper transactions provided by the underlying infrastructure. Conversely, the blockchain benefits from CVTS's ability to specify and adjudicate large-scale realistic computations.

Layer-2 scalability solutions Various solutions have been proposed on the second layer to increase blockchain scalability in terms of transaction throughput, such as Plasma or State Channels. These developments have particularities of their own, but are generally designed to register large amounts of transactions off-chain, which are only committed on-chain in order to reach finality or in the case of a dispute. A common requirement of these solutions is that the blockchain should be able to resolve any dispute that may arise (after a Plasma exit, or when a channel is closed). These exit mechanisms impose strong limitation on the maximum transaction size that either Plasma or State Channels can handle. So for example, if two parties disagree on an off-chain transaction that requires a large computation to be completed, they would be unable to settle who is correct on the main chain.

CVTScan greatly improve these technologies, as it allows both a Plasma chain or a State Channel to specify full CVTScomputations within its transactions. And in case a dispute lifts the computation to the main chain, the settlement can still be efficiently and safely resolved.

Reputation solutions Several projects intend to bring large scale and flexible computations to the blockchain by relying on a system of redundancy, reputation, and verification (e.g., SONM [2018], Golem [2016], and iExec [2017]). Although all these systems have different designs, the main idea is that computations are sent to a pool of off-chain providers, who submit their results independently. If someone challenges the presented results, a randomly assigned verifier decides on the correct outcome.

Although these systems have a built-in reputation mechanism to discourage misbehavior, it is not yet clear whether they are resistant to collusions or bribing in case the outcome of a computation can have financial consequences. CVTSgives instead mathematical guarantees on its dispute resolutions.

Trusted execution environment Several projects are integrating *enclaves* (e.g., Intel's SGX, ARM's TrustZone, or AMD's SEV) with the blockchain [TEEX 2018; Song et al. 2018; Enigma 2018; Cheng et al. 2018]. In a nutshell, enclaves are hardware-supported features that allow user-level code to create an execution environment whose privacy and integrity is protected even from processes running at higher privilege levels.

Computations running inside enclaves are not, a priori, reproducible. Using remote attestation in lieu of verification is equivalent to trusting the hardware manufacturer. Whether this level of centralization is justifiable or not depends on the application at hand, the manufacturer's competence, and on the potential for conflicts of interest. Even setting these issues aside, privacy is not always guaranteed, as recent attacks show [Weichbrodt et al. 2016; Brasser et al. 2017; Moghimi et al. 2017; Lee et al. 2017].

Enclaves may yet play an important role in the future of blockchain technology. However, their threat model and security guarantees are very different from CVTS's. Future applications may wish to combine both technologies by running part of their native computations inside hardware enclaves within CVTSNodes, or by running an entire CVTSNode inside a hardware enclave.

Zero knowledge proofs Another approach to enable private and scalable computations on the blockchain is to use general purpose zero-knowledge proof systems such as zk-SNARKS [Blum et al. 1988; Bitansky et al. 2012] or zk-STARKS [Ben-Sasson et al. 2018]. These systems achieve verifications in a fast, non-interactive and private manner. However, the computations that can be specified within these models are very restrictive as they are derived from arithmetic circuits without control flow. Therefore, they are not Turing complete and cannot run arbitrary code.

In off-chain environments, this technology is reaching maturity through projects such as Pinocchio [Parno et al. 2013] and libsnark [Ben-Sasson et al. 2013]. Meanwhile, zero-knowledge proofs have first appeared in blockchain technology with privacy coins, where shielded addresses protect the identities of parties involved in a transaction [Miers et al. 2013; van Saberhangen 2013]. In the Metropolis (Byzantium) fork, Ethereum introduced zk-SNARKS primitives into its virtual machine and some libraries have already started to be developed [Schaeffer 2018]. In this context, CVTS can accelerate this development by bringing more mature off-chain implementations of zk-SNARKS, such as Pinocchio or libsnark, directly into the blockchain.

Compilers and Virtual Machines Besides the scalability solutions mentioned above, several projects try to provide different languages for developers to write smart contracts in, such as Vyper, LLL, Bamboo, etc. Although these projects provide a welcome set of languages and tools for the development of smart contracts, they do not fully alleviate the most fundamental restrictions that currently hamper blockchain development. The main reason being that all these languages run on a free standing environment and therefore they do not offer the advantages that come with CVTS's underlying operating system, as described in section 3 below.

3 CVTSMachine specification

The CVTSMachine is a self-contained and deterministic computational model that can host modern operating systems. Real-world computations happen inside operating systems for good reasons. Developers are trained to use toolchains that operate at the highest possible abstraction level for any given job. These toolchains isolate them from operating system details and ease the porting to new architectures would then require the porting of a toolchain and operating system. Instead, CVTSMachines are based on a proven architecture for which a standard toolchain and operating system are already available.

On the other hand, off-chain computations performed by CVTS Machines must be verifiable by a blockchain. The blockchain must therefore host a reference implementation of the entire architecture. If it is ever to be trusted, this implementation must be easily auditable. To that end, both the architecture and the implementation must be open and relatively simple. Together, these requirements point to RISC-V. The RISC-V ISA is based on a minimal 32-bit integer instruction set to which several extensions can be

added [Waterman and Asanović 2017a]. Orthogonally, operand and address-space widths can be extended to 64-bits (or even 128-bits). Additionally, the standard defines a privileged architecture [Waterman and Asanović 2017b] with features commonly used by modern operating systems, such as multiple privilege levels, paged-based

Table 1: *Instruction counts by extension. Entries in the form $x + y$ refer to 32- and 64-bit variants of the same facility.*

Integer	Mul/Div	Atomics	Privileged	Total
47+12	8+5	11+11	5	71+28=99

Table 2: *The processor state. Memory-mapped to the lowest 512 bytes in physical memory for external read-only access.*

Offset	State	Offset	State
0x000	x0	0x160	misa
0x008	x1	0x168	mie
...	...	0x170	mip
0x0f8	x31	0x178	medeleg
0x100	pc	0x180	mideleg
0x108	mvendorid	0x188	mcounteren
0x110	marchid	0x190	stvec
0x118	mimplid	0x198	sscratch
0x120	mcycle	0x1a0	sepc
0x128	minstret	0x1a8	scause
0x130	mstatus	0x1b0	stval
0x138	mtvec	0x1b8	satp
0x140	mscratch	0x1c0	scounteren
0x148	mepc	0x1c8	ilrsc†
0x150	mcause	0x1d0	iflags†
0x158	mtval		

†CVTS-specific state.

virtual-memory, timers, interrupts, exceptions and traps, etc. Implementations are free to select the combination of extensions that better suit their needs.

RISC-V was born of research in academia at UC Berkeley. It is now maintained by its own independent foundation. Larger corporations, including Google, Samsung, and Tesla, have recently joined forces with the project [Tilley 2018]. The platform is supported by a vibrant community of developers. Their efforts have produced an extensive software infrastructure, most notably ports of the Linux operating system and the GNU toolchain [RISC-V 2018d]. It is important to keep in mind that RISC-V is not a toy architecture. It is suitable for direct native hardware implementation, which is indeed currently commercialized by SiFive Inc. This means that, in the future, CVTS will not be limited to emulation or binary translation off-chain.

The CVTSMachine can be separated into a *processor* and a *board*. The processor performs the computations, executing the traditional fetch-execute loop while maintaining a variety of registers. The board defines the surrounding environment with an assortment of memories (ROM, RAM, flash) and devices. To make verification possible, CVTSMachines map their entire state to physical memory in a well-defined way. This includes the internal states of the processor, the board, and of all attached devices. Fortunately, this modification does not limit the operating system or the applications it hosts in any significant way.

3.1 The processor

Following RISC-V terminology, CVTSMachines implement the RV64IMASU ISA. The letters after RV specify the extension set. This selection corresponds to a 64-bit machine, Integer arithmetic with Multiplication and division, Atomic operations, as well as the optional Supervisor and User privilege levels. In addition, CVTS Machines support the Sv48 mode of address translation and memory protection. Sv48 provides a 48-bit protected virtual address space, divided into 4KiB pages, organized by a four-level page table. This set of features creates a balanced compromise between the simplicity demanded by a blockchain implementation and the flexibility expected from off-chain computations.

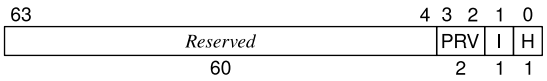

Figure 1: *The* `iflags` *register gives the current privilege level, and specifies whether the machine is temporarily idle waiting for interrupts, or has been permanently halted.*

There are a total of 99 instructions, out of which 28 simply narrow or widen, respectively, their 64-bit or 32-bit counterparts. Table 1 breaks down the instruction count for each extension. This being a RISC ISA, most instructions are very simple and can be simulated in a few lines of high-level code.[2] In fact, the only complex operation is the virtual-to-physical address translation. Instruction decoding is particularly simple due to the reduced number of formats (only 4, all taking 32-bits).

The entire processor state fits within 512 bytes, which are divided into 64 registers, each one holding 64-bits. Most of these registers are defined by the RISC-V ISA, and consist of 32 general-purpose integer registers and 26 control and status registers. The remaining are CVTS-specific. The processor makes its entire state available, externally and read-only, by mapping individual registers to the lowest 512 bytes in physical memory. The adjacent 1.5KiB are reserved for future use. The entire mapping is given in table 2.

The registers whose names start with `i` are CVTS-specific, and have the following semantics. The layout for register `iflags` can be seen in figure 1. PRV gives the current privilege level, I is set to 1 when the processor is idle (i.e., waiting for interrupts), and H is set to 1 to signal the processor has been permanently halted. Register `ilrsc` holds the reservation address for the LR/SC atomic memory operations.

Default initialization fills the state with the following values:

- PRV in `iflags` is set to 3 (for the Machine privilege level);
- `misa` is set to RV64IMASU;
- SXL and UXL in `mstatus` are set to 2 (for 64-bits);
- `pc` starts at `0x1000` (pointing to ROM);
- `marchid` is set to CVTS ␣ in ASCII.

`mvendorid` is used to test for matching on-chain and off-chain implementations. `mimplid` is incremented with each update to a matching pair. The remaining default state is set to zero.

3.2 The board

The interaction between board and processor happens through devices mapped to the processor's physical address space. Table 3 shows this mapping. There are 64KiB of ROM starting at address `0x1000`, where execution starts. The central role of this ROM is holding the *devicetree* [DTSpec 2017] describing the system hardware. In addition, a bootstrap program at ROM-base sets register `x10` to 0 (the value of `mhartid`), `x11` to point to the devicetree, and then jumps to RAM-base at `0x80000000`. This is where the entry point of the boot image is expected to reside. Finally, a number of additional physical memory ranges can be set aside for flash-memory devices. These will typically be preloaded with file-system images.

[2]The x86 ISA defines at least 2000 (potentially complex) instructions.

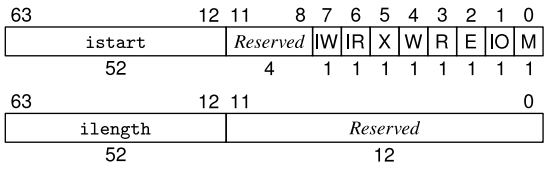

Figure 2: *Physical Memory Attributes. The* `istart` *and* `ilength` *of each range are aligned to a 4KiB boundary. The 12 LSBs of each 64-bit word give attributes for the range.*

Two non-memory devices are mapped to the address space. The Core Local Interruptor (or CLINT) controls the timer interrupt. The active addresses are `0x0200bff8` and `0x02004000`, respectively mapped to registers `mtime` and `mtimecmp`. The CLINT issues a hardware interrupt whenever `mtime` equals `mtimecmp`. To ensure reproducibility, the processor's clock and the timer are locked by a constant frequency divisor of 100. In other words, `mtime` is incremented once for every 100 increments of `mcycle`. The Host-Target Interface (HTIF) mediates communication with the external world. Its active addresses are `0x40000000` (`tohost`) and `0x40000008` (`fromhost`). It halts the machine when `tohost` is written to with bits 63–48 set to 0 and bit 0 set to 1. (Bits 47–1 can be set to an arbitrary exit code.) It also works as a rudimentary communications port during interactive sections.

The physical memory mapping is described by Physical Memory Attribute records (PMAs). Each PMA consists of 2 64-bit words. The first word gives the start of a range and the second word its length. Since the ranges must be aligned to 4KiB page boundaries, the lowest 12-bits of each word are available for attributes. Figure 2 shows the meaning of each attribute field. The M, IO, and E bits are mutually exclusive, and respectively mark the range as memory, I/O mapped, or excluded. Bits R, W, and X grant read, write, and execute permissions, respectively. Finally, the IR and IW bits mark the range as idempotent for reads and writes, respectively.

The board supports a total of 32 PMAs, and makes them available, read-only, starting at offset 2KiB in physical memory. Another 2KiB are reserved for future use. PMA 0 describes RAM, and PMAs 16–23 describe flash devices 0–7. These PMAs are user-configurable during initialization and read-only thereafter. (The RAM `istart` field is hard-coded to `0x80000000`.) Together, these records bound the maximum amount of storage accessible during computations.

3.3 The state transition function

A computation is a sequence of machine states $s_0, s_1, \ldots, s_h$, governed by a *transition function* step such that

$$s_{i+1} = \mathsf{step}(s_i). \tag{1}$$

Here, s_0 is the initial state and s_h is a halting state. The previous sections described the state space and the transition function for CVTSMachines in great detail.

Recall the state consists of the value of each word in the entire 64-bit address space of the CVTSMachine. In practice, it takes vastly fewer than 2^{64} bytes to represent a state. Only regions described in table 3 must be defined explicitly. All remaining values are implicitly filled with zeros.

The RISC-V ISA manuals [Waterman and Asanović 2017a,b] specify the state transitions corresponding to the execution of each instruction. This means that states are well defined between executed instructions. Since all instructions can be implemented in $O(1)$ time, CVTS*defines* each state transition to take exactly 1 cycle. The index of a given state in the sequence can be read from the corresponding value of `mcycle`. (Note that, since the machine can be occasionally idle, `minstret` does not track `mcycle`.)

Table 3: *Physical memory layout for a CVTSMachine.*

Physical address	Mapping
`0x00000000`–`0x000003ff`	Processor shadow
`0x00000800`–`0x00000Bff`	Board shadow
`0x00001000`–`0x00010fff`	ROM (Bootstrap & Devicetree)
`0x02000000`–`0x020bffff`	Core Local Interruptor
`0x40000000`–`0x40007fff`	Host-Target Interface
`0x80000000`–∗	RAM
∗–∗	Flash 0 (Disk 0)
...	...
∗–∗	Flash 7 (Disk 7)

The only salient CVTS-specific modification pertains to the halting of the machine. When field H in `iflags` is set to 1, no further state transitions are allowed. The condition is set explicitly when HTIF is instructed to halt the machine.

3.4 The Linux port

Setting up a Linux system from scratch involves a variety of steps. Unlike stand-alone systems, embedded systems are not usually self-hosting. Instead, components are built in a separate *host* system, on which a cross-compiling toolchain for the *target* architecture has been installed. The key components are the GNU Compiler Collection and the GNU C Library. This infrastructure can be found in the RISC-V GNU toolchain repository [RISC-V 2018a]. Building this infrastructure is the first step.

The toolchain can then be used to cross-compile the Linux kernel. Kernel sources can be found in the RISC-V Linux repository [RISC-V 2018b]. The kernel runs in supervisor mode, on top of a Supervisor Binary Interface (SBI) provided by a machine-mode shim: the Berkeley Boot Loader (BBL). BBL can be found in the RISC-V Proxy Kernel repository [RISC-V 2018e]. The BBL is linked against the Linux kernel and this resulting *boot image* is preloaded into RAM. The SBI provides a simple interface through which the kernel interacts with CLINT and HTIF. Besides implementing the SBI, the BBL also installs a trap that catches invalid instruction exceptions. This mechanism can be used to emulate floating-point instructions (See section 4.3). After installing the trap, BBL switches to supervisor mode and cedes control to the kernel entry point.

The final step is the creation of a root file-system. This process starts with a skeleton directory in the host system containing a few subdirectories (`sbin`, `lib`, `var`, etc) and text files (`sbin/init`, `etc/fstab`, `etc/passwd` etc). Tiny versions of many common UNIX utilities (`ls`, `cd`, `rm`, etc) can be combined into a single binary [Vlasenko 2018]. Target executables often depend on shared libraries provided by the toolchain (`lib/libm.so`, `lib/ld.so`, and `lib/libc.so`). Naturally, these libraries must be copied to the root file-system. Once the root directory is ready, it is copied into an actual file-system image (e.g., using `gene2fs`).

These steps can be automated. CVTS's SDK makes a preconfigured host environment available to developers in the convenient form of a Docker container. Complex Linux systems can be built with the help of Sifive's fork of Buildroot [Petazzoni 2018], or RISC-V's port of the Yocto project [RISC-V 2018c]. The environment in the container enables developers to customize the boot image and the root file-system according to the needs of their applications. Thousands of packages are available for installation.

```
memory@80000000 {
  device_type = "memory";
  reg = <0x0 0x80000000 0x0 0x8000000>;
};

flash@8000000000 {
  #address-cells = <0x2>;
  #size-cells = <0x2>;
  compatible = "mtd-ram";
  bank-width = <0x4>;
  reg = <0x80 0x0 0x0 0x4000000>;
  fs0@0 {
    label = "root";
    reg = <0x0 0x0 0x0 0x4000000>;
  };
};

chosen {
  bootargs = "root=/dev/mtdblock0 rw";
};
```

Figure 3: *Partial devicetree for a simple setup with 128MiB of RAM and a 64MiB flash device to be mounted as the root file-system.*

The root file-system image is installed as a flash device. Additional flash devices can be used to store the inputs to the computation, or to receive its outputs. The devicetree in ROM is used to inform Linux of the location of each flash device, the amount of RAM, and any kernel parameters. Figure 3 shows the relevant devicetree snippet. The first section specifies 128MiB of RAM starting at the 2GiB boundary. The middle section adds a 64MiB flash device, starting at the 512GiB boundary. The `mtd-ram` driver exposes the device as `/dev/mtdblock0` under Linux's virtual file-system. The last section, giving the kernel parameters `bootargs`, specifies the device to be mounted as root.

After completing its own initialization, the kernel eventually cedes control to `/sbin/init`. In CVTSDApps, this is typically a shell script that invokes the appropriate sequence of commands for performing the desired computation. The kernel passes to `/sbin/init` as command-line parameters all arguments after the separator `␣-␣` in `bootargs`. These can be used to define additional parameters for the computation to be performed. Upon completion, `/sbin/init` uses HTIF to halt the machine with an optional exit code. This can be used as part of the computation output. Arbitrarily complex inputs, parameters, and outputs can be passed as flash devices.

4 CVTSMachines off-chain

Off-chain implementations of CVTSMachines serve two purposes. Their main role is the execution of the computation itself. The secondary role is supporting the settlement of disputes over the results of computations. To provide these services, off-chain implementations of CVTSMachines must expose a programmable interface.

4.1 The scripting interface

The instantiation of a machine can only happen after the initial values for its entire physical address space have been defined. The physical memory layout is parameterized by the total amount of RAM, and by the starts and lengths of all flash devices. PMAs are automatically initialized from these parameters. The initial contents of RAM (e.g., with the boot image) and of the flash devices (e.g., the root file system), are given by `backing` files. The backing files for flash devices can be `shared`, in which case modifications to physical memory are saved to the file mapped to the corresponding memory location. Values that are not explicitly defined are default-initialized. In particular, the devicetree and the bootstrap in ROM can be either

```
m = machine{
  ram {
    ilength = 0x8000000,
    backing = "boot-image.bin"
  },

  rom = {
    bootargs = "root=/dev/mtdblock0 rw",
  }

  flash0 = {
    label = "root",
    istart = 0x8000000000,
    ilength = 0x4000000,
    backing = "root-file-system.bin",
    shared = true
  }
}
```

Figure 4: *The initialization of the off-chain machine automatically generates the devicetree of figure 3.*

filled automatically or loaded from a backing file. Initialization returns a *machine* handle that can be manipulated thereafter.

```
machine = machine{
  processor = processor,
  rom = rom,
  ram = ram,
  flash0 = drive,
  ...
  flash7 = drive,
  clint = clint,
  htif = htif,
}

processor ::= {
  x0 = word,
  x1 = word,
  ...
  pc = word
} | {
  backing = path
}

rom ::= {
  bootargs = string
} | {
  backing = path
}

ram ::= {
  ilength = word,
  backing = path
}

drive ::= {
  istart = word,
  ilength = word,
  backing = path,
  shared = bool,
  label = string
}

clint ::= {
  mtime = word,
  mtimecmp = word
} | {
  backing = path
}

htif ::= {
  fromhost = word,
  tohost = word
} | {
  backing = path
}
```

Figure 4 shows a sample `machine` call for the devicetree in figure 3.

The machine `run`s until `mcycle` exceeds a `limit` value. The function then returns `false` if the machine is halted, or `true` otherwise.

```
bool = machine:run{limit = word}
```

The machine can create a `snapshot` of its current state. This is a lightweight operation. It causes subsequent calls to `run` to modify the state under a copy-on-write policy. At a future time, a `rollback` operation can restore the snapshot state. Restoring the state lifts the copy-on-write policy. Consecutive calls to `snapshot` cause the previously snapshot state to be committed to memory. Calls to `rollback` without a previously snapshot state have no effect:

```
machine:snapshot()
machine:rollback()
```

The machine provides read-only access to its memory contents. To that end, invoking `word(address)` returns the value of a given 64-bit (aligned) word in the address space:

```
word = machine:word{address = word}
```

To `backup` the contents of the memory range associated to any component of the machine state, simply choose a file to store it:

```
bool = machine:backup{
  processor = path,
  rom = path,
  ram = path,
  flash0 = path,
  ...
  flash7 = path,
  clint = path,
  htif = path,
}
```

The machine exposes its entire state as a Merkle tree [Merkle 1979]. (For more on Merkle trees, see section 5.) It returns a proof that a target node belongs to the Merkle tree, given its `address` and `depth`:

```
proof = machine:prove{
  address = word,
  depth = word
}

proof ::= {
  address = word,
  depth = word,
  root = hash,
  siblings = {hash_1, ..., hash_depth},
  target = hash
}
```

The entries used for the initialization of individual devices can also be queried from their base `address`. The hash of the node at the `address` and `depth` is returned along with the entry. If the base address does not correspond to any device, or if the range length implied by the depth causes it to overlap with more than one device, an error is reported:

```
slice = machine:slice{
  address = word,
  depth = word
}

slice ::= {
  hash = hash,
  type = string,
  device = device
}
```

```
device ::=
  processor |
  rom       |
  ram       |
  drive     |
  clint     |
  htif
```

The `step` function advances the machine 1 cycle, logging every single access to the state along the way. All accesses are 64-bit aligned. Each log entry specifies the `operation` (`read` or `write`), the `address`, and the `word` read or written. In addition, each entry includes the proof produced by a corresponding call to `proof` for the address prior to the access.

```
log = machine:step()

log ::= {access_1, access_2, ..., access_k}

access ::= {
  operation = read | write,
  read = word,
  written = word,
  proof = proof
}
```

The importance of functions `prove`, `slice`, and `step`, and the convenience of this interface will become clear in section 5.

4.2 Reference implementation

CVTS's reference off-chain implementation is based on software emulation. The emulator is written in C/C++ with POSIX dependencies restricted to the terminal, process, and memory-mapping facilities. It is distributed as a library and scriptable in the Lua programming language according to the interface described above.

`Backing` files for RAM and file-system images take advantage of the host's support for sparse files. Only non-zero blocks take disk space. This enables the entire state of the machine to be specified in a convenient and compact form. The `snapshot` and `rollback` mechanism, as well as the `shared` attribute for `backing` files, are built on top of the host's support for virtual memory, using child processes and memory-mapped files with copy-on-write semantics. This makes them at the same time very simple to implement and very efficient.

The functionality for Merkle tree inspection requires additional support. For storage efficiency, the Merkle tree is maintained in its PATRICIA form [Morrison 1968]. For time efficiency, the tree is updated only when needed, in a lazy fashion. Each PMA range has a bitmap of dirty pages associated to it. Pages of physical memory are marked dirty in the TLB whenever they are written to. When the TLB entry for a dirty page is evicted, the corresponding bit is saved to the bitmap. When the `proof` function is called, it first updates the Merkle tree. The update proceeds bottom up, by visiting only the nodes that subtend the dirty pages. After the update, all bitmaps and TLB entries are marked clean.

For simplicity, the emulator follows a tight loop decoding and executing each instruction in turn. Other RISC-V emulators are based on the same approach [Waterman and Lee 2011; Bellard 2017]. In the future, the emulator will avoid repeated decoding of hot execution traces [Tröger et al. 2011]. It is possible to translate these traces to the host instruction set for even better performance [Bellard 2018]. However, the additional gains must be weighted against the reduced portability and significantly increased complexity.

4.3 Floating-point support

Floating-point operations are prevalent off-chain (except, perhaps, in the context of embedded devices). Therefore, programmers expect them to be available. If CVTShopes to bring a sense of normalcy to blockchain development, it must support floating-point operations. The difficulty is guaranteeing reproducibility.

Different floating-point implementations can disagree subtly when ostensibly performing the same operation on the same operands. Some of these differences arise from laxities in the IEEE 754-1985 standard. Although many of these have been tightened in the 2008 revision, several details remain unspecified. These include, but are not limited to, underflows, the sign of zero, operations involving infinity or NaNs, and the quantum for the rounding of certain recommended operations (e.g., $\sin$, $\log$ etc). Moreover, hardware implementations often take performance shortcuts that violate the standards they claim to adhere to. This leads to inconsistencies even across successive generations of the same architecture.

These issues argue strongly against adding floating-point support to any verifiable computation model. Accordingly, most blockchains wisely omit them entirely [Nakamoto 2009; Wood 2018; NEO's VM 2017; Cardano's VM 2017]. The only way to guarantee consensus is to emulate floating-point operations with a consistent software layer based on integer operations. Unlike floating-point operations, these are portable across different architectures.

In the RISC-V ISA, floating-point support is defined by extensions F and D (respectively for single- and double-precision). Together, these extensions augment the ISA with 32 floating-point registers, 1 control-status register, 30 new instructions, and 1 new instruction format. The specification adheres strictly to the IEEE 754-2008 standard [IEEE 2008]. Furthermore, it is limited to the required arithmetic operations that are fully specified.

There are many options for adding floating-point support to CVTS. In the first two approaches, the ISA does not include the F or D extensions. Therefore, they require no changes to the blockchain and off-chain implementations.

Emulation by RISC-V code: traps When a RISC-V machine does not support floating-point instructions, it raises a machine-level illegal-instruction exception whenever one is found. The idea is to use an exception handler installed by BBL to emulate the corresponding floating-point instruction using RISC-V integer instructions. This process is transparent to the supervisor and user levels, which work as if the ISA supported floating-point operations natively.

Emulation by RISC-V code: compiler Alternatively, the compiler can be instructed to target a RISC-V ISA that does not include floating-point operations. It substitutes them with calls to emulation routines it provides itself. The resulting binaries do not include floating-point instructions at all. This method is more efficient than using traps because it avoids the exception and decoding overhead.

The next two approaches add the F or D extensions to the ISA. Naturally, this means the off-chain machine must implement all RISC-V floating-point instructions. Furthermore, the blockchain verifier must include a perfectly matching implementation.

Emulation by native integers In this approach, floating-point instructions are still emulated off-chain. However, the emulation now uses native integer instructions, rather than RISC-V integer instructions. There are several high-quality open-source soft-float implementations from which to choose [Bellard 2016; Houser 2017]. This makes the off-chain machine even faster.

Native off-chain floating-point The next step in performance comes from using native floating-point instructions off-chain. Obtaining reproducible results from two distinct fully conforming IEEE 754-2008 implementations requires the cooperation between language standards, compilers, and, most regrettably, users. The prospects are improved in CVTS's context, since the reference implementation controls all these components and can fully specify any omissions in the standard. These corner cases include, but are not limited to, underflows, the sign of zero, and operations involving infinity or NaNs. Even with the added overhead, this approach should be the fastest one off-chain.

At present, the emulator makes the first two options available, that is, floating-point is emulated by RISC-V integer instructions. Support for the next two alternatives is planned for the future, when demand for faster floating-point grows.

5 CVTSMachines in the blockchain

Recall that CVTSis a platform for the development of decentralized applications. CVTSDApps enable parties that do not trust each other to enter into a binding contract in the blockchain that depends on the results of off-chain computations. It is convenient to use the characters *Alice* and *Bob* to represent these parties. Note that Alice and Bob are *roles*, not people. They may even represent competing collective interests. In fact, both roles will be played automatically by CVTSNodes that defend the interests of whomever controls the off-chain computer where the node runs. CVTSDApps are therefore a collaboration between a set of smart contracts running in the blockchain, and the off-chain software running on Alice's and Bob's nodes. As a general rule, the same DApp developer is responsible for the smart contracts and the Dapp specific off-chain software. The role of DApp developer will be played by *Charlie*. Alice and Bob trust Charlie, otherwise they would not engage with his DApp. Charlie, however, trusts neither Alice nor Bob. Naturally, Alice and Bob do not trust each other either.

CVTS's role is to support Charlie's work. To that end, CVTSoffers a variety of *primitives* that Charlie uses to mediate the potentially adversarial interactions between Alice and Bob. Some primitives require no interaction, and can be evaluated autonomously in the blockchain from their inputs. The interesting primitives, however, are those that, though completely defined by their inputs, can only be evaluated off-chain. By construction, when using a CVTSDApp, Alice and Bob always agree on the inputs to such primitives. Without loss of generality, Bob evaluates the primitive off-chain and submits the result. Alice is then given the chance to accept or reject Bob's result. Undisputed results can be used by Charlie's DApp for the purpose of his choice. In case of rejection, CVTSengages with Alice and Bob in a dispute resolution protocol that arbitrates in favor of the party with just cause. This adjudication always completes within a few interactions and at a negligible computational cost to the blockchain. CVTSautomates most of this process in a way that is extremely convenient to Charlie.

The most important of these primitives is the CVTSMachine. Smart contracts cannot afford to store the states for a CVTSMachine within the blockchain, let alone perform the implied computations. After all, the costs in terms of processing power and storage capacity would both be prohibitive. To solve these problems, CVTSuses cryptographic hashes to concisely represent machine states in the blockchain. From the blockchain's perspective, a computation is simply a pair of hashes corresponding to the initial and final states of the machine. The contents of the memory subtended by such hashes are known only off-chain. CVTSdefines a variety of additional primitives that allow smart contracts to conveniently manipulate the contents of the states corresponding to these hashes.

5.1 Machine state representation by hashes

Merkle trees [Merkle 1979] are binary trees where each node contains a hash. In CVTS, Merkle trees are based on the keccak hash function [Dworkin 2015].[3] Let s be a CVTSMachine state, giving the entire contents of its 64-bit address space. The Merkle tree m for s, or, equivalently, its root node, is

$$m = \mathsf{merkle}(s). \tag{2}$$

The tree is built up from its leaves in the following way. First, the state is partitioned into 2^{61} 64-bit words. The tree leaves contain the hashes for these words. Since there is no chance for ambiguity, we can simplify notation by identifying each node in the tree with the associated hash. Then, internal nodes v in the tree are built from their two children u_1 and u_2 by the relation

$$v = \mathsf{keccak}(u_1, u_2). \tag{3}$$

Here, keccak computes the hash of the concatenation of two input hashes. The procedure builds a tree of depth 61. The set of nodes at depth d partition the state into 2^d ranges with 2^{64-d} bytes each. Each node can therefore be identified by its depth and the starting address a for its range, which is aligned to a 2^{64-d} boundary.

Figure 5 shows a sample machine state s. It has the special property that devices have been aligned to nodes in m. For example, node v_4 subtends everything in the machine apart from the flash devices. It covers the shadows of the processor and the board, the ROM, the CLINT and HTIF devices, and the RAM. Nodes v_0–v_3 and v_5 each cover an independent flash device.

[3] This eases integration with the Ethereum blockchain. Like Ethereum, CVTSassumes there is no practical way to engineer collisions for the keccak hash function.

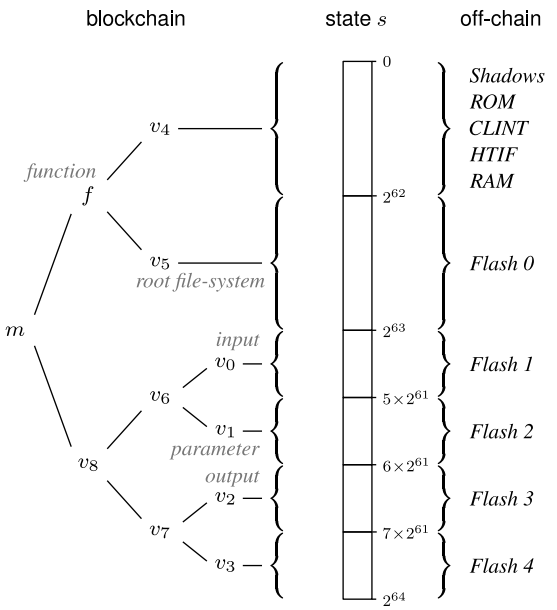

Figure 5: *The way in which the blockchain and off-chain representations for a machine state relate to each other.*

Think of s as the initial state for some machine. In this case, the ROM will contain the devicetree describing the hardware, and RAM will be preloaded with the Linux kernel. The particular choice of commands listed in `/sbin/init` in the root file-system (flash 0) decides what the machine does. It can, for example, perform an arbitrary computation over an *input* file-system (flash 1), and store results in an *output* file-system (flash 3). The computation can even be informed by the contents of an additional independent *parameter* file-system (flash 2).

Now assume s halts in state s'. From the blockchain's perspective, running m until it halts can be seen as the evaluation of an arbitrary function $v_2' = f(v_0, v_1)$. Here, v_2' is the node in $m' = \mathsf{merkle}(s')$ that corresponds to v_2 in m. Consider a library of hashes f, each corresponding to a different useful function. For example, one such function could decrypt the input file-system v_0 into the output file-system v_2', taking a key from the parameter file-system v_1. To specify this computation in terms of CVTSMachines with a single hash m, a smart contract must build m from its components f, v_0, and v_1. Once it receives m', it must be able to settle disputes over whether m indeed halts as m'. Finally, it must be able to verify that if v_1' is the node in m' that corresponds to v_1 in m.
The following property of Merkle trees is the foundation for all these operations: Given a node v, its depth d in tree m, and the starting address a for the associated memory range, it is possible to verify that v is indeed part of m. To see this, consider the path from v to m

$$w_d = v,\ w_{d-1},\ \ldots,\ w_1,\ w_0 = m. \tag{4}$$

Then, given the siblings u_i for every node w_i in the path:

$$w_{i-1} = \begin{cases} \mathsf{keccak}(u_i, w_i), & \text{if } a \wedge 2^{64-i}, \\ \mathsf{keccak}(w_i, u_i), & \text{if } \neg(a \wedge 2^{64-i}). \end{cases} \tag{5}$$

For this reason, the sequence

$$\mathsf{siblings}(m, a, d) = (u_1, u_2, \ldots, u_d) \tag{6}$$

serves as *proof* for the claim that v is at address a and depth d in m. To verify this, simply compute p_0 from (5) and compare with m.

Moreover, given a valid proof for v in m, the same procedure can be used to attest that a root hash m' results from replacing v in m with any given node v'. These verifications are very efficient, each requiring only d applications of the keccak hash.

We are now ready to define the first two CVTSprimitives

$$v = \mathsf{slice}(m, a, d) \quad \text{and} \quad m' = \mathsf{splice}(m, a, d, v'). \tag{7}$$

In the absence of disputes, slice returns v and splice returns the result m' of replacing v in m with v'. To successfully defend these results in disputes, one can simply present $\mathsf{siblings}(m, a, d)$ and v to the blockchain.

For convenience, CVTSdefines two additional primitives

$$w = \mathsf{read}(m, a) \ \Leftrightarrow\ \mathsf{keccak}(w) = \mathsf{slice}(m, a, 61), \quad \text{and} \tag{8}$$

$$m' = \mathsf{write}(m, a, w') = \mathsf{splice}\big(m, a, 61, \mathsf{keccak}(w')\big) \tag{9}$$

for directly manipulating words, rather than hashes. Disputes can be resolved once the blockchain receives $\mathsf{siblings}(m, a, 61)$ and w.

5.2 The verification game

The *verification game* [Feige and Kilian 1997] is a protocol that allows an arbiter with limited computational resources to referee a game between two computationally unlimited players. Its use in conjunction with Merkle trees was introduced by Canetti et al. [2011], and the application to blockchains first appeared in TrueBit [Teutsch and Reitwießner 2017]. In this scenario, the blockchain is the "referee", and the "game" is between a "player" Bob that defends a result for an off-chain computation, and a "player" Alice that disputes it.

Let s_0 be the initial state for the computation, and s_n its final state. Recall that $s_{i+1} = \mathsf{step}(s_i)$ and $m_i = \mathsf{merkle}(s_i)$. The verification game is the dispute resolution mechanism for CVTS's primitive

$$m_n = \mathsf{compute}(m_0, n) = \mathsf{merkle}\big(\mathsf{step}^{(n)}(s_0)\big). \tag{10}$$

It is divided into two stages. The first stage finds a single step of computation on which Alice and Bob disagree. The final stage effectively computes the step that follows. If it matches the state proposed by Bob, he wins the dispute. Otherwise, Alice wins.

The disagreement step The interval $[i, j]$, for $i < j$, is said to be a *disagreement interval* if the following two conditions are met:

1. Bob has sent to the blockchain hashes m_i and m_j, claiming they correspond to $\mathsf{merkle}(s_i)$ and $\mathsf{merkle}(s_j)$;
2. Alice has manifested to the blockchain that she agrees with m_i but disagrees with m_j.

A *disagreement step* is a disagreement interval where $j - i = 1$.

When Alice disputes that $m_n = \mathsf{compute}(m_0, n)$, the range $[1, n]$ becomes the initial disagreement interval. A *partition contract* can find the disagreement step with an interactive binary search, starting from $[1, n]$. At iteration ℓ, the contract starts with a disagreement interval $[i_\ell, j_\ell]$. It requests from Bob the hash $m_{k_\ell} = \mathsf{merkle}(s_{k_\ell})$, where k_ℓ is the middle point between i_ℓ and j_ℓ. Knowing Bob's m_{k_ℓ}, Alice then chooses between $[i_{\ell+1}, j_{\ell+1}] = [i_\ell, k_\ell]$ or $[k_\ell + 1, j_\ell]$ as the next disagreement interval. This continues until Alice selects an interval with length one.

The procedure finishes after $O(\log n)$ interactions between Alice, Bob, and the partition contract.[4] Any party that fails to react within a pre-determined deadline looses the verification game by timeout. At iteration ℓ, Alice and Bob must independently obtain the state s_{k_ℓ} for the machine. If the machine is always started from scratch, the

[4] This can be reduced by changing to an n-ary search.

total incurred off-chain computation is $O(n \log n)$. However, by preserving a snapshot of s_{i_ℓ}, they can bring the cost down to $O(n)$.

Settling the dispute At this point, the blockchain has found the disagreement step $[i, i+1]$, where $i \in \{0, \ldots, n-1\}$. It knows both m_i and m_{i+1} according to Bob. Alice agrees with m_i, but disputes m_{i+1}. To decide the party with just cause, the blockchain must effectively compute $m'_{i+1} = \mathsf{merkle}\big(\mathsf{step}(s_i)\big)$ and compare it with Bob's m_{i+1}. However, the blockchain does not have unrestricted access to s_i. All it has is the root hash $m_i = \mathsf{merkle}(s_i)$.

Alice is expected to post her off-chain state access log for $\mathsf{step}(s_i)$ to a *memory manager contract*. Off-chain, these accesses progressively modify the state $s_i = (s_i)_0$ into $(s_i)_1$, $(s_i)_2, \ldots$ until after the kth and last access it becomes $(s_i)_k = s_{i+1}$. All machine steps take $O(1)$ time to simulate and number k of accesses is always small. Entry j in the log, for $j \in \{1, \ldots, k\}$, contains

1. An operation o_j for the access (read or write);
2. An address a_j for the access;
3. The word r_j at a_j in $(s_i)_{j-1}$;
3′. In case of writes, the word w_j at a_j in $(s_i)_j$;
4. The $\mathsf{siblings}\big((m_i)_{j-1}, a_j, 61\big)$.

The blockchain implementation for the step function is hosted by an *emulator contract*. Alice's off-chain implementation must match this blockchain implementation down to the order in which the state accesses are logged. As a benefit of this restriction, the blockchain implementation can read and write to the state as if its whole contents were available. During its execution, the reference step function issues a sequence of state accesses to the memory manager. As long as the accesses match Alice's log, everything works transparently.

Formally, as the reference step function performs k' accesses, the memory manager progressively updates $m_i = (m'_i)_0$ to $(m'_i)_1$, $(m'_i)_2$, …, until it reaches $(m'_i)_{k'}$. Access j, for $j \in \{1, \ldots, k'\}$, contains the following information

1. An operation o'_j for the access (read or write);
2. An address a'_j for the access;
3. In case of writes, the word w'_j to be written.

As each access is processed, the memory manager:

- Checks that $j \le k$, $o'_j = o_j$, and $a'_j = a_j$;
- Checks that $r_j = \mathsf{read}\big((m'_i)_{j-1}, a_j\big)$ with the siblings.

Then, for a read access, the memory manager:

- Sets $(m'_i)_j = (m_i)_{j-1}$;
- Returns r_j to the emulator contract.

For writes, the memory manager:

- Checks that $w'_j = w_j$;
- Sets $(m'_i)_j = \mathsf{write}\big((m'_i)_{j-1}, a_j, w_j\big)$ with the siblings.

At any point, if a check fails, Alice loses the dispute. If, however, $k = k'$ and $m_{i+1} \neq (m'_i)_{k'}$, Alice wins the dispute.

5.3 CVTSMachines as one of many primitives

CVTSprimitives are made available to Charlie through a functional programming interface. The goal is isolating the primitives from the idiosyncrasies of specific smart contract programming languages and blockchains. The syntax itself is not important. What matters is the semantics associated to each primitive.

Charlie can use this interface to build expression DAGs that represent complex composite computations. The computations can involve several machines that exchange data with each other. An empty DAG is first initialized with a call to:

```
dag = dag()
```

Each DAG vertex corresponds to a primitive. A primitive's inputs come from the outputs of its children vertices. The primitive's output can in turn serve as input to one or more primitives.

Primitives are divided into two categories. *Disputable* primitives behave as *future* values *promised* to Alice by Bob. Their outputs are set by Bob, and must be accepted or disputed by Alice. Disputable primitives can only appear as internal vertices in the expression DAG. They are the read, write, slice, splice, step, and compute primitives described in sections 5.1 and 5.2.

Constant primitives have their outputs set by Charlie. They are are implicitly accepted by both Alice and Bob throughout their interactions with Charlie's DApp. These primitives potentially represent large chunks of data and the availability of their contents has to be guaranteed by Charlie, possibly with help from our data availability primitives. Naturally, word, hash, string, and id *literals* are constants. CVTS's *blob*, *resource*, and *machine* primitives, described below, are also constant. Only constant primitives can appear as DAG leaves.

CVTSsupported constant and future types are:

```
constant ::= word | hash | string | id
disputable ::= word-future | hash-future
```

Constant primitives accept only constants as input. In contrast, disputable primitives accept both constants and disputables:

```
word-type ::= word | word-future
hash-type ::= hash | hash-future
```

Constant primitives *Blob* primitives represent arbitrary binary data stored in the blockchain. The provided `hash` gives the output. It must match the root hash for a Merkle tree built from the `data`, padded with zeros to $2^{64-\texttt{depth}}$ bytes:

```
hash = dag:blob{
  hash = hash,
  depth = word,
  data = string
}
```

Resource primitives describe files stored off-chain. Files are assumed to be available from the given `uri`. The `download-size` can be used to bound, a priori, the total data transfer requirements. Only the first `range-length` bytes of the file are considered. This bounds the memory required to map it into the machine state. The provided `hash` output must match the root hash for a Merkle tree built from the first `range-length` bytes of the corresponding file, padded with zeros or truncated to $2^{64-\texttt{depth}}$ bytes:

```
hash = dag:resource{
  hash = hash,
  depth = word,
  range-length = word,
  download-size = word,
  uri = string
}
```

Machine primitives are used to describe CVTSMachine states. The specification follows the scripting interface described in section 4.1. The only the difference is that `backing` files are given as paths. Instead, they are resource constants. The provided `hash` output must correspond to the root Merkle tree hash for the corresponding CVTSMachine state:

```
hash = dag:machine{
  hash = hash,
  processor = processor,
  rom = rom,
  ram = ram,
  flash0 = drive,
  ...
  flash7 = drive,
  clint = clint,
  htif = htif
}
```

Disputable primitives The *compute* primitive executes a CVTS Machine. The `initial-state` gives the value of m_0 and `steps` the value of n, so that the future value is compute(m_0, n). `Steps` therefore bounds, a priori, the amount of computation required:

```
hash-future = dag:compute{
  initial-state = hash-type,
  steps = word-type
}
```

The Merkle tree manipulation primitives *slice*, *splice*, *read*, and *write* are available as:

```
hash-future = dag:slice{
  root = hash-type,
  address = word-type,
  depth = word-type
}

hash-future = dag:splice{
  root = hash-type,
  address = word-type,
  depth = word-type,
  target = hash-type
}

word-future = dag:read{
  root = hash-type,
  address = word-type
}

hash-future = dag:write{
  root = hash-type,
  address = word-type,
  word = word-type
}
```

CVTSalso provides a variety of *simple* primitives that increase the expressive power of expressions. Disputes over such primitives can be settled within the blockchain directly from their inputs. Several binary operations on words have the signature:

```
word-future = dag:bin-op(word-type, word-type)
```

and mirror the RISC-V ISA.They can be divided into arithmetic:

```
add, sub, mul, mulh, mulhu, mulhsu,
div, divu, rem, remu, sll, srl, sra;
```

bitwise:

```
or, and, xor;
```

and comparisons:

```
eq, ne, lt, ltu, ge, geu.
```

Signed integers are represented by two's complement. Boolean values are returned as words where `1` means *true* and `0` means *false*. Conversely, when a Boolean value is expected by a conditional, `0` is considered false and any other value is considered true.

Conditionals are available as ternary `if` primitives whose output is set to `if-true` if `condition` is true, and `if-false` otherwise:

```
word-future = dag:if{
  condition = word-type,
  if-true = word-type,
  if-false = word-type
}
```

The `hash` primitive builds a 256-bit hash by the concatenation of its 64-bit component words:

```
hash-future = dag:word4(word-type, word-type,
    word-type, word-type)
```

To help with Merkle tree construction, hashes can be built from words and from the concatenation of two hashes:

```
hash-future = dag:keccak(word-type)
hash-future = dag:keccak(hash-type, hash-type)
```

For completeness, hashes can also be tested for equality and used as input for a conditional:

```
word-future = dag:eq(hash-type, hash-type)
word-future = dag:neq(hash-type, hash-type)

hash-future = dag:if{
  condition = word-type,
  if-true = hash-type,
  if-false = hash-type
}
```

DAG and vertex interfaces Charlie must define the identity of the players for the roles of Alice and Bob. Recall that Bob *proposes* a result and Alice can *object* or accept it:

```
dag:proposing-role{id = id, stake = word}
dag:objecting-role{id = id, stake = word}
```

The `stake` argument gives the price for buying Alice's or Bob's position. It measures what is at stake for each of them depending on the results of the computation. Charlie sets this value to facilitate Alice and Bob to delegating their roles (section 6.2).

Primitive creation functions return:

```
vertex ::= constant | disputable
```

A DAG may contain multiple disconnected sub-DAGs. To specify or retrieve the root vertex for the DAG, Charlie invokes:

```
dag:root(vertex)
```

This value can be later obtained by anyone that calls:

```
vertex = dag:root()
```

The primitive for any vertex can be queried:

```
primitive = vertex:primitive()

primitive ::= word | hash | string |
  blob | resource | machine |
  read | write | slice | splice |
  add | sub | ... | geu | if |
  word4 | keccak | keccak-hh | eq-hh | if-hh
```

Likewise, the children can be obtained by name or index:

```
vertex = vertex:child-by-index(word)
vertex = vertex:child-by-name(string)
```

Together, the `primitive` and `child` functions enable the entire sub-DAG reachable from a vertex to be traversed.

The DAGs and vertices can change state during their lifetimes:

```
dag-state ::= undefined | proposed | accepted |
  objected | sustained | overruled

vertex-state ::= undefined | proposed | accepted
```

Constant vertices are always in the `accepted` state. At construction, the DAG and all its disputable vertices are in the `undefined` state. These states can be obtained from the DAG and vertex objects:

```
dag-state = dag:state()
vertex-state = vertex:state()
```

Proposing the value of any vertex changes its state to `proposed`:

```
vertex:propose-hash(hash)
vertex:propose-word(word)
```

To start a proposal, Bob first invokes:

```
dag:start-proposal()
```

Then, he proposes the output value for the root vertex. Finally, he completes the proposal with a call to:

```
dag:finish-proposal()
```

This changes the DAG to the `proposed` state.

The value proposed for any vertex can be checked with a call to:

```
word = vertex:proposed-word()
hash = vertex:proposed-hash()
```

To accept the proposed root for the DAG, Alice calls:

```
dag:accept()
```

This changes the DAG state to `accepted`.

To object to the root value for the DAG, Alice invokes:

```
dag:start-objection()
```

and then proposes a new, *distinct* value for the root vertex. There are now two cases to consider. If all immediate children of the root vertex are in the accepted state, the dispute can be settled by the root primitive resolution protocol. To that end, Alice simply calls:

```
dag:finish-objection()
```

This changes the DAG to the `objected` state while the protocol is completed. If Alice succeeds, the DAG is changed to the `sustained` state. Otherwise, it is changed to the `overruled` state.

If, however, there are any proposed or undefined children, she must first propose values for *all* vertices in the sub-DAG reachable from the root. Only then can she call:

```
dag:finish-objection()
```

This changes the DAG to the `objected` state. Bob must now find a vertex, accessible from the root, for which he accepts all inputs but objects to the output. To specify this vertex, Bob gives its path from the root. He also must specify a *distinct* value for its output:

```
dag:defend-word(path, word)
dag:defend-hash(path, hash)

path ::= {vertex_1, vertex_2, ..., vertex_k}
```

If the vertex is undefined, Alice's dispute is declared ill-formed and Bob wins immediately. If the path is invalid, Bob's defense is declared ill-formed and Alice wins immediately. Otherwise, the primitive dispute resolution protocol is engaged.

With this setup, arbitrarily complex expressions behave just like disputable CVTSprimitives: they have agreed-upon inputs and come equipped with a dispute resolution procedure for their output.

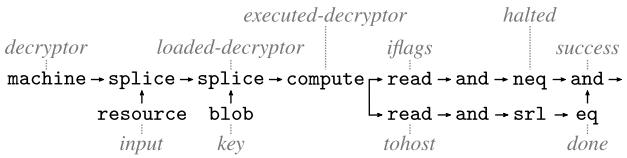

Figure 6: *Expression DAG corresponding to the decryptor example. Constant vertices corresponding to literal values have been omitted for brevity. Variable names are shown in gray.*

An example The following example illustrates the power of the expression DAG interface:

```
d = dag()

decryptor = library{
  hash = decryptor-machine-hash,
  dag = d
}

input = d:resource{
  hash = input-hash,
  depth = decryptor-flash-1-depth,
  uri = http://example.com/charlie/input.drive
}

key = d:blob{
  hash = password-hash,
  depth = decryptor-flash-2-depth,
  data = password
}

loaded-decryptor = d:splice{
  root = d:splice{
    root = decryptor,
    address = decryptor-flash-1-addr,
    target = input
  },
  address = decryptor-flash-2-addr
  target = key
}

executed-decryptor = d:compute{
  initial-state = loaded-decryptor,
  steps = 10*2^32
}

iflags = d:read{
  root = executed-decryptor,
  address = machine-processor-iflags-addr,
}

tohost = d:read{
  root = executed-decryptor,
  address = machine-htif-tohost-addr
}

halted = d:neq(d:and(iflags, 1), 0)
done = d:eq(d:srl(d:and(tohost, d:srl(-1, 16)), 1), 0)
success = d:and(halted, done)

d:root(success)
```

In the example, Charlie defines an expression DAG with root *success*. The implied computation starts with a *decryptor* machine, available from an off-line library (not shown). Charlie can install his own library of pre-defined machines in the CVTSNode, or use a library of machine specifications created and stored remotely by another developer. Either way, the hash constant ensures machines that have been tampered with can be easily identified. Next, *input* and *key* flash devices are spliced into the decryptor machine. This *loaded-decryptor* is run for at most `2*2^32` steps, and results in an *executed-decryptor*. The corresponding state is then probed. Inspection of the processor's *iflags* register tells if the machine is *halted*. The

value stored in the payload of HTIF's *tohost* register tells if the machine halted after it was *done* decrypting. Figure 6 shows the corresponding DAG, with literal values (i.e., strings, words, and hashes) omitted.

Charlie oversees Alice's and Bob's interaction with the DAG and its vertices. After all, Charlie is responsible for the blockchain DApp component where these objects live. Charlie's software acting on Alice or Bob's behalf can issue events that trigger reactions from the CVTSNodes representing Alice and Bob. These reactions are also under Charlie's control, since he is in charge of the off-chain DApp components installed in their nodes. He must, as usual, protect his DApp against attacks by rogue users. CVTSsimplifies part of this process by encapsulating access control in all DAG operations.

6 The CVTSNode

The *CVTSNode* is the software and hardware infrastructure that hosts the off-chain components of CVTSDApps. Each user that wishes to interact with a CVTSDApp must have a CVTSNode at his disposal. CVTSNodes will initially be made available as Docker containers to be run on a computer under the user's responsibility. Future plans include their distribution as a multi-platform library that developers can link to an executable for users to install as a self-contained DApp.

DApps can include native off-chain components that access the full storage and computational power of the hardware where the node is installed. In addition, all nodes contain a reference implementation of the CVTSMachine that DApps can control to perform verifiable computations. Finally, nodes contain the infrastructure necessary for the interaction of off-chain and blockchain DApp components.[5]

At the core level, the CVTSplatform gives DApp developers the freedom to combine native code, reproducible CVTSMachines, and the blockchain's API in any way they see fit. Given the fast pace in which novel applications for blockchain technology appear, this seems to be the only way to avoid restraining developers' creativity. Nevertheless, we can foresee a variety of common tasks, challenges, and patterns that are likely to arise when using the CVTSplatform. With time, the CVTSplatform will encapsulate these into a set of higher-level interfaces built on top of the core [Teixeira and Nehab 2019a]. The core includes only facilities for the automated execution and verification of CVTSMachines.

6.1 Off-chain expression DAGs

Off-chain DApp components need to interact with existing blockchain DAGs. After all, the computations implied by such DAGs must be performed off-chain within the CVTSNode. These interactions are mediated by off-chain representations for DAGs, which can be automatically built from a blockchain instance with a call to:

```
off-dag = off-dag(dag)
```

Off-chain DAGs provide DApps with a high-level interface that completely encapsulates typical use cases.

Bounds for the data transfer, memory, and computation requirements for the main sub-DAG can be obtained from:

```
bounds = off-dag:bounds()

bounds ::= {
  data = word,
  memory = word,
  compute = word
}
```

[5] In the case of Ethereum, a light client.

To download the data for all constant nodes in the main sub-DAG and verify the computed hashes match the declared hashes:

```
download-status = off-dag:download{timeout = word}

download-status ::= accepted | failed | timedout
```

Once download is complete, all vertices in the main sub-DAG can be evaluated with a single call to:

```
off-dag:evaluate()
```

The `upload` method submits results back to the blockchain. It also isolates DApp developers from the details of any potential dispute:

```
off-dag:upload{fee = word}
```

The `fee` argument is used for role delegation (section 6.2).

Let v be the root for the main sub-DAG. The `upload` method checks whether the role being played is proposing or objecting. The proposing role only acts if v is undefined in the blockchain. In that case, it proposes the value it obtained off-chain for v. The objecting role only acts if a value for v has been proposed to the blockchain. If the value matches what it obtained off-chain, it accepts the value. This is how the overwhelming majority of interactions will play out. If, however, the proposed value for v in the blockchain does not match the value computed off-chain, the objecting role starts a dispute. The ensuing interactions between the blockchain and the CVTSNodes of both proposing and objecting roles are automatically handled by CVTSthe platform.

With the exception of the compute primitive, disputes can be settled right away. Compute primitives require multiple interactions with the blockchain. If a party cannot guarantee the responsiveness of his CVTSNode throughout a dispute, he could lose by default judgement. To minimize this risk, CVTSoffers another convenience to DApp developers: the dispute delegation market.

6.2 Dispute delegation market

A principal party can delegate potential disputes to a proxy by setting the `fee` argument of the `upload` method to a non-zero value. This causes CVTSto advertise the disputed DAG in the *dispute delegation market*. The advertising principal is notified as soon as a proxy purchases a dispute. Proxy candidates own CVTSNodes on which *CVTS's dispute proxy DApp* has been enabled. Users of the proxy DApp are in the business of collecting fees for defending the interests of principal parties that are unwilling to conduct their own disputes.

The proxy DApp download an advertised DAG, computes its value off-chain, and checks if it matches the value proposed to the blockchain by the role for sale. If so, it can *purchase* the role for the `stake` specified in the DAG. This amount will be returned if and only if the proxy wins the ensuing dispute. In that case, the proxy is also rewarded the `fee`. If the proxy fails in the dispute, the `stake` is instead sent to the principal party.

Guarantees CVTSguarantees that an honest party can always win any dispute in which it is involved. This is a strong guarantee, but it is also the *only* guarantee. In the absence of disputes, the value for the DAG is *defined* to be whatever the interested parties proposed and accepted. It is therefore perfectly reasonable to act on the accepted value, whether or not it is indeed the true result of the computation defined by the DAG. Any unsatisfied party has the responsibility of contesting the value.

Disputed values, however, may or may not be useful. The situation is truly *exceptional*: At least one of the parties is being dishonest, perhaps even *both* are. The proper way forward must be decided

by the DApp developer. One potentially useful bit of information is whether the objection was overruled or sustained. Note that even this bit is only true when at least one of the parties is honest.

Whenever a principal party wishes to delegate a dispute to a proxy, it has no way to guarantee its interests will be defended in good faith. The only solution is to align the interests of the principal and proxy. This is why proxy roles must be purchased by the principal's stake in the dispute. If the proxy is honest and wins the dispute on the principal's behalf, the only cost to the principal is the fee. He may receive further benefits from the contract where the dispute originated. If the proxy loses the dispute, whether on purpose or by negligence, the principal keeps the stake. His interest in the original contract becomes moot.

Naturally, no proxy will ever buy disputes from principals playing dishonest roles. They are, as they should, on their own defending their interests. Even honest principals may fail to sell disputes if the fee is too low or the stakes too high. To guarantee service availability to honest principals, CVTSwill maintain a number of nodes with the dispute proxy DApp installed. These nodes will be configured to purchase, up to a maximum stake, advertised disputes that are profitable but have not found a proxy within a preset deadline.

7 Future work

The focus of this document on the core functionality, and on the interfaces DApps use to directly specify, control, and verify off-chain computations. The CVTSplatform will offer several additional components built over the core, or extending its reach. These will be described in more detail in future publications [Teixeira and Nehab 2019a,b].

Data availability CVTSremedies the severe storage limitations of the blockchain by keeping on-chain only Merkel tree hashes of off-chain data. As mentioned in section 5.2, CVTSassumes that all parties involved in a verification role have access to these data. In certain applications, this is difficult to guarantee. In particular, the risk for *data withholding attacks*, where one of the parties submits a hash to the blockchain while refusing to make this data available to others, must be mitigated.

The problem of data availability is a major concern in the design of blockchain consensus algorithms [Buterin 2012]. However, the issue becomes much simpler in the context of local consensus. Teixeira and Nehab [2019a] provide several design patterns for dealing with data availability during verification. *Data channels*, *device encryption*, and the *data ledger* ensure availability in all scenarios likely to be encountered by CVTSDApps.

Usability One of the key barriers to the wide adoption of blockchain technology is the inconvenience experienced by DApp users. Although the literature on usability of centralized applications still applies to decentralized ones, blockchain idiosyncrasies have not yet been fully addressed from the perspective of user experience. Teixeira and Nehab [2019a] describe several design patterns for the development of simple and intuitive DApps.

As an example, CVTSwill offer an automatic infrastructure for trading tokens. This will free the users from concerns over the different tokens used inside each DApp. A system for outsourcing deferred actions will also be provided. This will enable users to turn their machines off even when engaged in a protocol that requires interacting with the blockchain within strict deadlines. In this situation, a proxy party will act on the users' behalf in exchange for a fee. (Much like the dispute delegation market described in section 6.2.) The use of cryptographic time-locks [Rivest et al. 1996] will also accommodate situations in which the user must reveal a secret in the future that should not be immediately passed to the proxy party. Other usability constructs will be described to facilitate file transfers and reduce gas costs. Together, these facilities will bring the user experience of CVTSDApps closer to that of current centralized solutions.

The CVTSSDK A variety of higher-level APIs that encapsulate typical uses for the core will be available with the release of the CVTSSDK. These will include the usability and data availability solutions described above, as well as the containers for the CVTS Node and for the development of CVTSMachines. In time, the APIs available within the SDK will greatly reduce the size and complexity of DApps blockchain components. In turn, this will significantly increase the portability of DApps to multiple blockchains. The CVTSSDK will be distributed in open source and extensively documented [Teixeira and Nehab 2019b].

Extensions to the CVTSMachine CVTSMachines can be extended with two exciting new devices. The *dehashing device* gives applications the power to traverse hash pointer data structures. Programs running inside a CVTSMachine can use the dehashing device to read the contents of a block given only its hash. Although this operation is impossible in general, it becomes possible when the universe of allowed blocks is known by all parties in advance. The most direct application is to blockchains themselves. When a CVTSMachine is running, the dehashing device queries a hash table, preloaded in the host, for the block that matches the hash. If a dispute arises, any party can propose the block as proof it matches the required hash. In this way, the dehashing device enables *blockchain introspection*. Parties can enter into contracts that depend on the entire state of the blockchain where the contracts are themselves defined. This has a variety of valuable applications, notably in futures markets.

Another planned device is the *timely data port*. The port enables reproducible communication between CVTSMachines by tying the data packets entering or leaving the machine to the value of `mcycle` at the event. DApps can schedule packet delivery to happen at a given future `mcycle`. CVTSMachines can also be rolled back to the `mcycle` for delivery. The timely data port breaks new ground in the progress towards the Web 3.0. It will enable DApps that involve the direct collaboration between multiple CVTSMachines.

Crowd disputes It is possible to envision applications that involve many independent participants, each with some stake in the results of an off-chain computation. In such cases, it is vital to prevent a coordinated crowd of dishonest participants from using sequential disputes over an honest result as a denial-of-service attack on the contract. We have developed a variant of the verification game that enables any honest participant to defend his result against an entire crowd at negligible cost. When demand becomes apparent, the CVTSplatform will be extended to support this variant.

8 Conclusions

This paper laid the foundations on which the CVTSplatform stands. CVTS's mission is to help DApp developers build ever more compelling products to their clients. As any paradigm shift, the blockchain brings both opportunity for real innovation and the risk of "wheel reinvention". In a direct application of the *principle of least astonishment*, CVTS's core enables developers to leverage preexisting knowledge and tools to boost their productivity. The remaining components of the CVTSplatform, described in a future document [Teixeira and Nehab 2019a], will help developers unleash their creativity when taking advantage of the blockchain's unique potentials.

References

BELLARD, F. 2016. Softfp library. Webpage. https://bellard.org/softfp/.

BELLARD, F. 2017. Riscvemu. Source code. https://bellard.org/riscvemu/.

BELLARD, F. 2018. A generic and open source machine emulator and virtualizer. Webpage. https://www.qemu.org.

BEN-SASSON, E., BENTOV, I., HORESH, Y., and RIABZEV, M. 2018. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptology ePrint Archive*, 2018:46.

BEN-SASSON, E., CHIESA, A., TROMER, E., and VIRZA, M. 2013. Succinct non-interactive zero knowledge for a von neumann architecture. Cryptology ePrint Archive, Report 2013/879. https://eprint.iacr.org/2013/879.

BITANSKY, N., CANETTI, R., CHIESA, A., and TROMER, E. 2012. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, New York, NY, USA. ACM, 326–349. ISBN 978-1-4503-1115-1. http://doi.acm.org/10.1145/2090236.2090263.

BLUM, M., FELDMAN, P., and MICALI, S. 1988. Non-interactive zero-knowledge and its applications. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, New York, NY, USA. ACM, 103–112. ISBN 0-89791-264-0. http://doi.acm.org/10.1145/62212.62222.

BRASSER, F., MÜLLER, U., DMITRIENKO, A., KOSTIAINEN, K., CAPKUN, S., and SADEGHI, A.-R. 2017. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies*.

BUTERIN, V. 2012. A note on data availability and erasure coding. https://github.com/ethereum/research/wiki/A-note-on-data-availability-and-erasure-coding.

BUTERIN, V. 2018. On sharding blockchains. Wiki. https://github.com/ethereum/wiki/wiki/Sharding-FAQs.

BUTERIN, V. 2018. Sharding. GitHub repository. https://github.com/ethereum/sharding.git.

CANETTI, R., RIVA, B., and ROTHBLUM, G. N. 2011. Practical delegation of computation using multiple servers. In *Proceedings of the ACM Conference on Computer and Communications Security*, 445–454.

CARDANO'S VM. 2017. IELE OpCodes. Source code. https://github.com/runtimeverification/iele-semantics/blob/master/iele.md.

CHENG, R., ZHANG, F., KOS, J., HE, W., HYNES, N., JOHNSON, N. M., JUELS, A., MILLER, A., and SONG, D. X. 2018. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contract execution. *CoRR*, abs/1804.05141.

DTSPEC. 2017. *Devicetree specification*. Power.org, Freescale Semiconductors, IBM, Linaro, and ARM. http://devicetree.org.

DWORKIN, M. J. 2015. SHA-3 standard: Permutation-based hash and extendable-output functions. Technical report.

ENIGMA, T. 2018. Enigma documentation. https://enigma.co/protocol/.

FEIGE, U. and KILIAN, J. 1997. Making games short. In *Proceedings of STOC*, 506–516.

GOLEM, T. 2016. The golem project. https://golem.network/crowdfunding/Golemwhitepaper.pdf.

HOUSER, J. 2017. Berkeley softfloat. Webpage. http://www.jhauser.us/arithmetic/SoftFloat.html. Release 3d.

IEEE, C. S. 2008. Standard for floating-point arithmetic, IEEE Std 754-2008.

IEXEC, T. 2017. Blockchain-based decentralized cloud computing. https://iex.ec/whitepaper/iExec-WPv3.0-English.pdf.

LARIMER, D. 2017. Dpos consensus algorithm - the missing white paper. https://steemit.com/dpos/@dantheman/dpos-consensus-algorithm-this-missing-white-paper.

LEE, S., SHIH, M.-W., GERA, P., KIM, T., KIM, H., and PEINADO, M. 2017. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. In *26th USENIX Security Symposium, USENIX Security*, 16–18.

MERKLE, R. C. 1979. *Secrecy, Authentication, and Public Key Systems.* PhD thesis, Stanford University.

MIERS, I., GARMAN, C., GREEN, M., and RUBIN, A. D. 2013. Zerocoin: Anonymous distributed e-cash from bitcoin. http://zerocoin.org/media/pdf/ZerocoinOakland.pdf.

MOGHIMI, A., IRAZOQUI, G., and EISENBARTH, T. 2017. CacheZoom: How SGX amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*, 69–90.

MORRISON, D. R. 1968. Patricia—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4): 514–534.

NAKAMOTO, S. 2009. Bitcoin: A peer-to-peer electronic cash system. Whitepaper. http://bitcoin.org/bitcoin.pdf.

NEO'S VM. 2017. OpCodes. Source code. https://github.com/neo-project/neo-vm/blob/master/src/neo-vm/OpCode.cs.

PARNO, B., HOWELL, J., GENTRY, C., and RAYKOVA, M. 2013. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, 238–252.

PETAZZONI, T. 2018. Buildroot. Website. https://buildroot.org.

RISC-V. 2018. GNU toolchain. GitHub repository. https://github.com/riscv/riscv-gnu-toolchain.

RISC-V. 2018. Linux. GitHub repository. https://github.com/riscv/riscv-linux.

RISC-V. 2018. Poky: port of the Yocto project. GitHub repository. https://github.com/riscv/riscv-poky.

RISC-V. 2018. Project home. GitHub repository. https://github.com/riscv.

RISC-V. 2018. Proxy Kernel. GitHub repository. https://github.com/riscv/riscv-pk.

RIVEST, R. L., SHAMIR, A., and WAGNER, D. A. 1996. Time-lock puzzles and timed-release crypto.

SCHAEFFER, T. 2018. Zokrates. https://github.com/JacobEberhardt/ZoKrates.

SONG, C., WU, S., LIU, S., FANG, R., and LI, Q.-L. 2018. Distributed cloud computing in trusted hardware. https://ankr.network/.

SONM, T. 2018. Sonm documentation. https://docs.sonm.com/.

TEEX, T. 2018. TEEX—TEE-enabled eXecution platform for public blockchain. https://teex.io.

TEIXEIRA, A. and NEHAB, D. 2019. CVTSdesign patterns.

Whitepaper. *To appear*.

TEIXEIRA, A. and NEHAB, D. 2019. The CVTSSDK. *To appear*.

TEUTSCH, J. and REITWIESSNER, C. 2017. A scalable verification solution for blockchains. Whitepaper. `https://people.cs.uchicago.edu/~teutsch/papers/truebit.pdf`.

TILLEY, A. 2018. Google, Tesla get behind challenge to Arm chip design. *The Information*. `https://www.theinformation.com/articles/google-tesla-get-behind-challenge-to-arm-chip-design`.

TRÖGER, J., MIHOČKA, D., and KEPPEL, D. 2011. Fast microcode interpretation with transactional commit/abort. In *4th Workshop on Architectural and Microarchitectural Support for Binary Translation*, San Jose, CA. `http://www.emulators.com/docs/amas-bt2011.pdf`.

VAN SABERHANGEN, N. 2013. Cryptonote v 2.0.

VLASENKO, D. 2018. Busybox. Website. `https://busybox.net`.

WATERMAN, A. and ASANOVIĆ, K. 2017. *The RISC-V Instruction Set Manual*, volume I: User-Level ISA. RISC-V Foundation. Version 2.2.

WATERMAN, A. and ASANOVIĆ, K. 2017. *The RISC-V Instruction Set Manual*, volume II: Privileged Architecture. RISC-V Foundation. Version 1.10.

WATERMAN, A. and LEE, Y. 2011. Spike, a RISC-V ISA simulator. GitHub repository. `https://github.com/riscv/riscv-isa-sim`.

WEBASSEMBLY, C. G. 2018. WebAssembly specification, release 1.0. `https://webassembly.github.io/spec/core/index.html`.

WEICHBRODT, N., KURMUS, A., PIETZUCH, P., and KAPITZA, R. 2016. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In *European Symposium on Research in Computer Security*, 440–457.

WOOD, G. 2018. Ethereum: A secure decentralised generalised transaction ledger. Yellowpaper. `https://ethereum.github.io/yellowpaper/paper.pdf`. Byzantium version e94ebda – 2018-06-05.