Democratizing the Cryptocurrency Ecosystem by Just-In-Time Transformation of Mining Programs

Wei Liu[1], Zhenhua Li[1]✉, Feng Qian[2], Feiyu Jin[1], Hao Lin[1]
Yannan Zheng[3], Bo Xiao[3], Xiaokang Qin[3], Tianyin Xu[4]
[1]Tsinghua University [2]University of Southern California [3]Ant Group [4]UIUC

***Abstract*—Democracy is crucial to a cryptocurrency ecosystem, as the diversity of miners (farms, personal computers, web clients, or even cloud functions) underlays the credibility of the cryptocurrency. Among miners, web clients used to be the vast majority, e.g., 50M+ as of March 2018. As time went on, however, cryptomining was gradually monopolized by mining farms with dedicated hardware (e.g., ASICs), and web clients scaled down to ∼0.1M. To suppress mining farms, certain cryptocurrencies (like Monero) adopted new mining algorithms such as RandomX whose execution relies on general-purpose hardware architectures. Unfortunately, this further impairs web-based cryptomining as web clients cannot provide the desired architecture support to these algorithms. This paper explores how to revive software democracy of efficient web-based cryptomining, using a novel program transformation technique termed Vectra. Vectra employs just-in-time (JIT) transformations of mining programs for web architectures; it effectively identifies and merges isomorphic instructions upon execution. Vectra ensures correct transformations based on symbolic constraints of the instructions. Real-world deployments show that Vectra reduces WASM instructions by about 7× and achieves a 3×–16× speedup for web cryptomining in diverse execution environments like PCs, mobile phones, and serverless platforms, which translates to a high (69%–274%) return-on-investment (ROI) for common users.**

***Index Terms*—cryptocurrency ecosystem, web cryptomining, serverless computing, just-in-time program transformation**

I. Introduction

Cryptomining (or mining for short) is an essential pillar of cryptocurrency systems like Bitcoin [1], Ethereum [2], and Monero [3]. Miners solve computationally difficult mathematical puzzles for new transactions that need to be added to the blockchain; the first miner to solve the puzzle is paid in cryptocurrency. To maintain the cryptocurrency's credibility, cryptomining was designed to be decentralized and support a variety of miners [4], [5] including mining farms, personal computer (PC) applications, web clients, or even serverless cloud functions. In principle, no individual entity can dominate computation power and monopolize the market, i.e., the cryptocurrency ecosystem should be *democratic*.

However, the democracy of the cryptocurrency ecosystem has been recently threatened by the increasing cost-efficiency of dedicated mining hardware (e.g., ASICs and FPGAs), which facilitates centralized mining farms' monopolizing the mining of major cryptocurrencies like Bitcoin and Monero. This renders common users' cryptomining less profitable or unprofitable, because the vast majority of mining rewards are taken by mining farms while the remainder (gained by common users) cannot even cover their electricity and network traffic bills. In particular, web clients, which used to take the majority of miners (e.g., 50M+ as of March 2018) for their pervasive accessibility on various devices and OS independence [6], [7], had dramatically scaled down to <0.1M as of May 2022. In the meantime, PC applications scaled down to ∼0.8M.

To break the monopoly of mining farms, in recent years mainstream cryptocurrencies like Monero, ArQmA [8], and Scala [9] enforced new mining algorithms like RandomX [10], RandomARQ [11], and Panthera [12]. These algorithms rely on the architectural features of general-purpose hardware, such as dynamic instruction execution [13] and vectorization [14] in the instruction set architectures (ISAs) of CPUs, effectively reducing the advantage of dedicated mining hardware that only supports fixed instruction patterns [15], [16]. The mining efficiency of PC applications thus surpasses that of mining farms by two orders of magnitude (>140×) [10].

Unfortunately, the adoption of new mining algorithms impairs web cryptomining, because web is designed to hide underlying hardware details to achieve platform independence. To validate a transaction, web clients have to emulate architectural features of general-purpose hardware. This severely slows down web cryptomining (∼25× slower than a PC application) and makes it unprofitable. Notably, Coinhive (the biggest web cryptomining provider by then) asserted web mining to be "*not economically viable anymore*" and shut down its service [17]. Hence, the democracy has not been fully recovered as web cryptomining has a much larger legitimate[1] user base (i.e., 50M+); also, it is an important monetization channel for many websites to maintain their existence [22], [23].

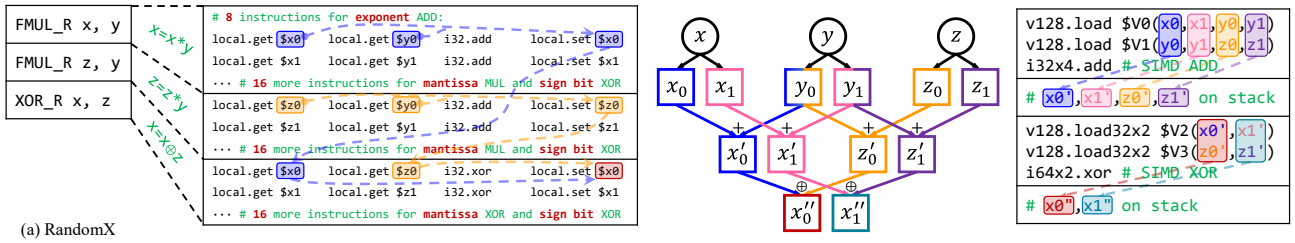
Key insight. We aim to revive the democracy of cryptocurrency ecosystems by making web cryptomining efficient (and thus profitable). Through our analysis of mainstream mining algorithms, we find that the bottleneck of web cryptomining comes from the inefficient execution of mining instructions (in particular, vector instructions that dominate the mining process) in WebAssembly (WASM)[2]. WASM uses a *stack-based* virtual machine (VM) in a push-pop manner for portability, which differs greatly from the *register-based* ISAs of CPUs.

We take the three vector instructions dynamically generated by RandomX (the most popular algorithm to resist mining

[1]The legitimate user base of web cryptomining is far more than the illegal one that conducts web cryptojacking [18], [19] and illicit transactions [20], [21]. In fact, when legal users are prevented by new mining algorithms, web cryptomining is being dominated by illegal usages.

[2]We focus on WASM for its performance advantage over JavaScript [24].

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00040



(a) RandomX instructions (b) 72 element-wise WASM instructions that emulate floating-point arithmetics (c) Element propagation graph (d) Resulting WASM SIMD instructions

Fig. 1: Vectra accelerates web cryptomining by analyzing and utilizing the element propagation paths for vector instructions.

farms) in Figure 1a as an example. Each instruction operates on two floating points. For PC applications, these instructions can be efficiently translated to single-cycle SIMD (Single Instruction Multiple Data) instructions [25] of CPUs. However in Figure 1b, web clients need to translate each vector instruction to 24 (SISD-style) WASM element instructions. This is because WASM cannot represent most (∼80%) vector-related data types in CPU ISAs (e.g., directed-rounded floating points [26] and 128-bit integers [27]) that need special registers for state maintenance, thus having to emulate their arithmetics. Also, due to the mutable jump targets of conditional instructions widely used in the new mining algorithms that make the control flow quite complicated [28], these element instructions are hard to be re-vectorized by traditional code optimization techniques like SLP vectorization [29], [30].

Our approach. We propose a program transformation technique termed Vectra to achieve efficient web cryptomining by accelerating the execution of mining instructions. The key idea is to merge instructions of dynamically generated mining programs based on the isomorphism of data propagation across vector elements in a just-in-time manner.

Specifically, Vectra introduces an abstraction termed *Element Propagation Graph* (EPG) to track how the value of each element is propagated along instructions. Figure 1c shows an example of EPG of instructions in Figure 1b. Each node represents a vector element; each edge represents a data propagation path through an arithmetic operation. For example, element x''_0 is generated from two propagation paths, i.e., the two graph components in blue and orange. Similarly, there are two other propagation paths (pink and purple) for element x''_1. Vectra constructs EPGs *in-situ*, because the mining programs are generated dynamically based on the mining algorithm.

The four propagation paths (blue, pink, orange, and purple) in Figure 1c are isomorphic—they have the same data type, operator sequence, and topological structure. Such isomorphism is not a coincidence—the majority (85%) of element propagation paths are isomorphic within a block of vector instructions (cf. §III-A). The high density of isomorphism offers an opportunity to optimize the execution of related instructions. Instead of following the default operation sequence of WASM in Figure 1b, Vectra rearranges the elements of isomorphic propagation paths together on top of the WASM processing stack. Thus, multiple instructions corresponding to adjacent elements can be transformed into a WASM SIMD instruction. In Figure 1d, 24 element instructions from Figure 1b are merged into 6 vector instructions. Note that such an instruction reduction (4× in the simple example) will be further improved (6×–12×) when we have more (typically 10K+) instructions with a higher degree of isomorphism.

We further optimize Vectra by automatically finding optimal isomorphic components across the EPG to minimize the number of resulting WASM instructions. Conventional approaches based on backtracking [31] bear a high time complexity of $O(n!)$, where n is the number of graph nodes. To address this, we leverage the locality of data dependencies among vectors together with a branch-and-bound strategy [32] to divide the graph into subgraphs. By constraining the search within subgraphs, we reduce time complexity to $O(n^2)$.

Since the transformation can rearrange the elements in a vector, we must check whether the vector is modified ("dirty") before accessing it for correct referencing at runtime. However, this needs to examine all the previously executed instructions and introduces great (>120%) time overhead. Instead, before executing the mining program for a new transaction, we capture every vector's *symbolic constraints* [33], [34] that indicate the conditions in which the vector will be manipulated. Based on this foresight information, we only need to check if the conditions of dirty vectors are satisfied at runtime, thereby cutting the time overhead to merely 4%.

We implement the above approach into a practical system. Our real-world deployments with extensive workloads show that Vectra can reduce emulated WASM instructions by 7× on average, which is 3×–16× faster at mining Monero under diverse browser execution environments: PCs, mobile phones, and serverless platforms (involving distributed metalog synchronization for stateful functions). Compared to existing methods that suffer from a negative (-65% to -24%) ROI [35], Vectra brings a promising ROI of 69%–274% to users.

Summary. This paper makes the following contributions.

- We conduct an in-depth study on the democracy of the cryptocurrency ecosystem, and show how program analysis can benefit web cryptomining with new mining algorithms.
- We reveal the root cause of inefficient web cryptomining—the gap between the ISAs of WASM (a high-level, portable language for web) and the native CPU.

TABLE I: Devices used for our web-based mining tests. PC denotes Personal Computer and MP denotes Mobile Phone.

Device	CPU	RAM	OS
PC-1	Intel i9-10980HK (2.90 GHz)	64 GB	Windows 11
PC-2	Intel i7-10700F (2.90 GHz)	64 GB	Windows 11
PC-3	Intel E5-2420 (1.90 GHz)	64 GB	Ubuntu 22.04
PC-4	Apple M2 Pro (3.49 GHz)	16 GB	MacOS 13.4
MP-1	Snapdragon 888 (2.84 GHz)	12 GB	Android 13
MP-2	Kirin 990 (2.86 GHz)	6 GB	Android 10

- We develop Vectra, a just-in-time program transformation technique to close the gap and enable profitable web cryptomining. It speeds up web cryptomining by 3×–16× and provides an ROI of 69%–274% to users.
- All the code and data involved in this work are released at https://WebCryptomining.github.io.

II. Understanding Web Cryptomining

To combat mining farms that have gradually monopolized cryptomining with dedicated hardware such as ASICs and FPGAs, vulnerable cryptocurrencies (e.g., Monero [3], ArQmA [8], and Scala [9]) have recently enforced the new mining algorithms such as RandomX [10] and Panthera [12]. These new algorithms are anticipated to greatly reduce the efficiency of dedicated hardware (thus suppressing mining farms) through *randomized instruction execution* on top of the common hardware architectures. Among them, RandomX is the most widely adopted one, while the other algorithms are mostly its variants [11]. In the following, we illustrate relevant concepts based on RandomX, but note that all of them can be generalized to other mining algorithms.

With new mining algorithms, a subscribing miner gathers the information of new blocks with a nonce [36]. It then transforms these data to a *random* seed and uses it to generate special instructions in the RandomX instruction set. These dynamic instructions utilize the architectural features of the instruction set architectures (ISAs) of common CPUs, including registers and instructions for vectors. Hence, they can be efficiently translated to the machine code of CPUs [10]. In contrast, they are unfriendly to dedicated hardware, which is designed for static instruction patterns and is costly to be reconfigured. Thus, mining farms can be effectively suppressed.

Unfortunately, these new algorithms further hinder web-based mining due to the architectural differences between the web and general-purpose CPUs, while users constantly expect to mine cryptocurrencies over the web given its convenience, compatibility, and being widely supported and effectively powered by the emerging serverless platforms [37], [38], as well as decentralized monetization of web contents [22], [39].

Mining performance over web clients. To understand how the efficiency of web cryptomining is degraded, we compare the performance of CPU-based mining and web-based mining with the new algorithms. We connect three mining algorithms (i.e., RandomX, RandomARQ, and Panthera) corresponding to three major cryptocurrencies (i.e., Monero, ArQmA, and Scala) to the mining pool [40] to perform real-world mining tasks with devices listed in Table I.

For CPU-based mining, we directly mine on four PCs and two mobile phones as listed in Table I. For mobile devices, we only run Panthera because the other two mining algorithms currently do not natively support mining on mobile phones [12]. To run the three algorithms over the web, we compile their open-source releases into WASM binaries using Emscripten toolchains [41]. We incorporate the WASM binaries into a web page, and load it for each algorithm with Chrome 118.0 on each device in an automated manner [42], [43]. We continuously perform mining tasks for a month and record the number of calculated hash values per second (i.e., hash rate). Finally, we run a total of 257,290 mining tasks.

Figure 2 shows the performance of the Panthera algorithm on different platforms. It exhibits superior performance in all native environments, with an average hash rate of 326 H/s on PCs and 113 H/s on mobile phones. In the web environment, however, its hash rate drops significantly by >10× (note that y-axis is using a log scale). The hash rates on PCs are all less than 20 H/s, while the hash rates on mobile phones are all less than 10 H/s. Such low mining efficiency directly makes web cryptomining unprofitable—web clients can hardly become the first to generate valid hash for a new transaction. Similar situations happen to Monero and ArQmA.

Register-based ISA vs. stack-based ISA. We dissect the execution process of the mining algorithms to understand how the architectural differences affect the mining efficiency. We find that the instructions involved in the execution of the RandomX program can be mainly divided into *vector instructions*, *scalar instructions*, and *control instructions*. The execution time of vector instructions over the web is significantly longer than that of the other two types of instructions. As shown in Figure 3a, for regular CPU platforms, the execution time of vector instructions is close to that of scalar instructions, each accounting for ∼40%. However, in the WASM environment (Figure 3b), the time usage of vector instructions accounts for over 85%. This implies that vector instructions are particularly slowed down, leading to the inferior web mining performance.

Delving deeper, we find that the inefficiency of vector operations in mining programs is caused by the high instruction translation cost across register-based and stack-based ISAs. Algorithms like RandomX employ register-based ISAs [44], where vector operations are directly mapped to SIMD (Single Instruction Multiple Data) instructions on general-purpose architectures (e.g., SSE/AVX for x86 CPUs and Neon for ARM CPUs [45]) as they are all register-based. However, WASM is featured by a stack-based virtual machine (VM) for better portability across diverse underlying platforms [24].

Figure 4 shows the difference in integer addition over the two architectures. For the register-based architecture (i.e., the RandomX ISA in Figure 4a), we first move the two operand integers to two source registers (i.e., `r0` and `r2`). The `IADD_RS` instruction is then executed to add the integers from

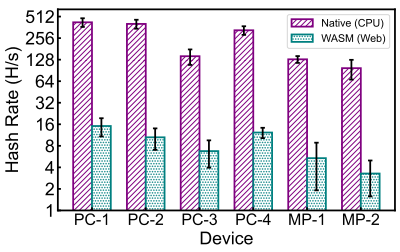
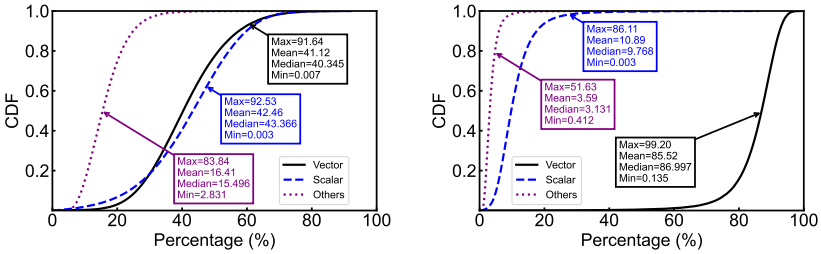


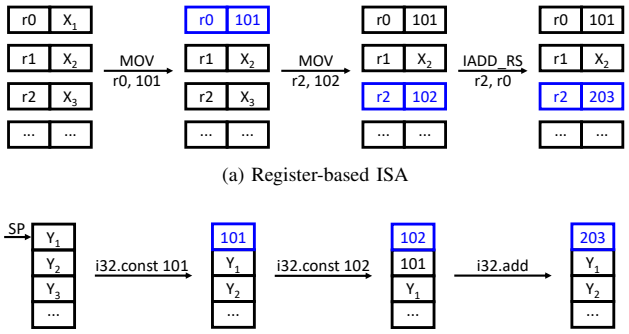
Fig. 2: Hash rates of Panthera on PC apps and web clients.



(a) Regular CPU Platforms

(b) WASM

Fig. 3: Percentages of the execution time for different types of instructions.



(a) Register-based ISA

(b) Stack-based ISA

Fig. 4: Example of adding two constant integers ($101+102=203$) in register-based and stack-based ISAs.

the two sources. The result is stored in the destination register (i.e., `r2`) of the `IADD_RS` instruction. For the stack-based architecture (i.e., the WASM VM in Figure 4b), all operations are accomplished by manipulating the data on top of the stack. We need to first push the two operands onto the top of the stack as referenced by the stack pointer (SP) in Figure 4b. When adding the two operands, the WASM VM pops the top two data elements and adds them up, and the result is pushed back.

Although the stack-based WASM ISA provides excellent portability, it cannot represent many vector data types when translating mining instructions. This is because the stack-based WASM VM has no persistent registers for state maintenance [46]. For example, the RandomX ISA supports directed rounding of floating points, which requires the underlying platform to have special registers to record towards which direction (e.g., up, down, to nearest, and to zero) a floating point should be rounded. Today's CPUs all provide registers (e.g., the MXCSR register of the x86 CPU) to maintain such a state, but the WASM VM conceals these interfaces for portability[3]. Worse still, they cannot be addressed by type casting, which introduces precision loss and incorrect results.

Consequently, most (>80%) vector instructions are executed by *element-wise* arithmetic emulation in WASM. For floating-point vectors, the WASM VM emulates the arithmetic by processing their mantissas, exponents, and sign bits as individual elements. For 128-bit integers, each element is represented as two 64-bit integers and its arithmetic is emulated by processing the high-64 and low-64 bits separately. This substantially increases the number of WASM instructions (by 10–35×), rendering web cryptomining inefficient and unprofitable.

[3]Specifically, the `FADD_R` instruction can be directly mapped to `ADDPD` or `FADD` SIMD instructions in x86 or ARM ISAs respectively [26], but cannot be directly mapped to WASM SIMD instructions.

III. Design

We present Vectra for efficient web cryptomining through just-in-time program transformation, so as to revive the democracy of the cryptocurrency ecosystem.

A. Design Overview

As we demonstrate in §II, the inefficiency of web cryptomining derives from the lack of data-level parallelism during the execution of vector instructions of dynamic mining programs in WASM. As shown in Figure 1b, for vector data types that are not natively supported by WASM, their corresponding instructions have to be emulated with element-wise operations, which greatly increases the number of translated WASM instructions. Nevertheless, we find that these element-wise operations can still have a high degree of data-level parallelism if we transform and streamline a block of instructions by investigating their *vector element propagation* processes.

For example, each floating-point multiplication instruction is emulated by mantissa multiplication, exponent addition, and sign bit XOR. This results in considerable repeated WASM instruction patterns across different floating point instructions. Further, even for instructions that are not originally in an SIMD form (i.e., scalar instructions), we can still seek out isomorphism in their (element) propagation processes to achieve speedups. Thus, isomorphism among instructions is not a coincidence but an intrinsic feature, accounting for 85% of instructions within a code block according to our measurements. We have also examined RandomX's variants including Panthera, RandomARQ, and RandomXEQ, finding that all of them have the same level of isomorphism. In fact, exploiting the SIMD features of CPU is a common practice in cryptomining, so high degrees of isomorphism can also be observed in other algorithms.

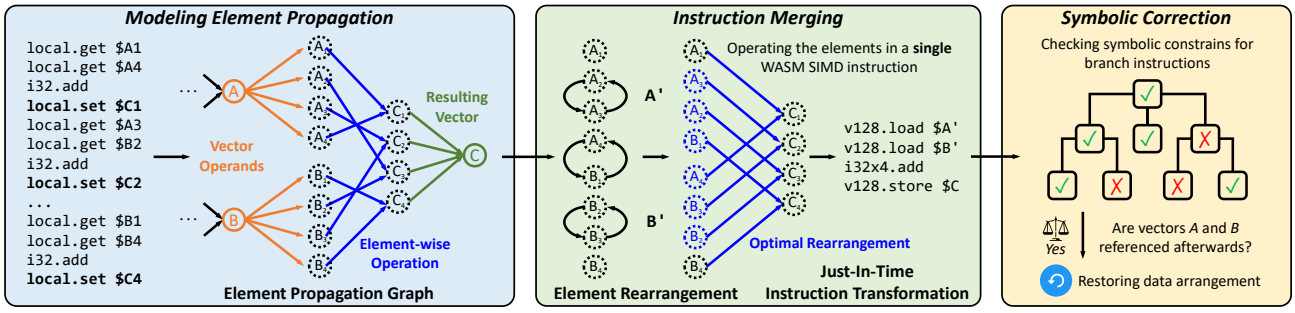


Fig. 5: Architectural overview of Vectra.

Vectra transforms a dynamic mining program in-situ by searching and merging isomorphic propagation paths into a single SIMD instruction in WASM to significantly accelerate web-based mining. Figure 5 visualizes our approach in Vectra, which takes a sequence of instructions of dynamically generated mining program as the input, and performs the following steps to realize the approach:

- *Modeling Element Propagation* (§III-B). For a new sequence of instructions with respect to the mining program of a new transaction, we first model its element propagation process as an *element propagation graph* (EPG) in a just-in-time manner, so as to describe how the value of a certain element is spread across the element-related instructions. Given that the WASM ISA does not natively support many types of vector operations (cf. §II) on general-purpose hardware architectures, we decompose the input instructions into fine-grained operations over basic data types in WASM during the construction of EPGs.
- *Efficient Program Transformation* (§III-C). After the just-in-time EPG construction, we search for isomorphic propagation paths in the graph and rearrange the data elements of isomorphic components onto the WASM processing stack for SIMD executions. Unfortunately, directly searching all graph nodes bears a high time complexity of $O(n!)$, where n is the number of graph nodes. To address this, our key observation is that the input instructions have a strong locality regarding data dependencies, which confines the isomorphic data propagation in neighbor instructions. We cluster the graph nodes based on their *locality density*, and only search for isomorphic components in the same clustered subgraph. This reduces the complexity to $O(n^2)$.
- *Runtime Symbolic Correction* (§III-D). During the execution of mining algorithms, conditional branch instructions can lead to incorrect results as they interrupt the data propagation among vectorized instructions. Intuitively examining all the previously executed instructions to ensure correctness is not efficient. Instead, we introduce *symbolic constraints* to the EPG. They capture the conditions in which elements will propagate across a branch instruction. With the symbolic constraints, we can ensure correct branching in a speculative and efficient manner at runtime.

B. Modeling Element Propagation

We take RandomX-series mining algorithms (which are used by major cryptocurrencies and have derived many variants) as an example to illustrate the element propagation modeling. It should be noted that such modeling is generalizable to other algorithms (cf. §IV). The execution of RandomX involves a huge number of vector operations in different stages. It first generates instructions in its own ISA format based on transaction information, and then translates them to the instructions on specific platforms (i.e., the general-purpose CPUs). The operations of RandomX instructions can be directly mapped to either scalar or SIMD instructions on regular CPU platforms, since they are all register-based. However, when they are executed in WASM, we need to translate them into WASM stack operations that have no special registers for state maintenance.

Decomposing element propagation. We introduce a new abstraction called Element Propagation Graph (EPG) to track the propagation of vector elements during the WASM execution of mining programs (which is mostly in SISD forms). EPGs facilitate the search for isomorphic propagation paths and vectorize them into WASM SIMD instructions. To validate a new transaction, Vectra iterates through the RandomX instructions (the dynamic program) for graph construction.

To achieve this, Vectra first decomposes each RandomX instruction into basic operations in the WASM ISA. The WASM VM only supports the round-to-nearest ties-to-even mode for floating point rounding [47], while the RandomX ISA (and CPU platforms) additionally supports round-to-negative, round-to-positive, and round-to-zero modes, which are controlled by the `fprc` register [10]. For the other three modes not supported by WASM, we translate their corresponding instructions by simulating floating point calculation. We use the 64-bit integer type in WASM to store the mantissas, and use the 32-bit integer type to store the exponents and sign bits. We follow the IEEE 754 standard [48] to manipulate the three integers. We also decompose the calculation on 128-bit integers in RandomX into the high-64 bits and low-64 bits operations in WASM. The other scalar instructions do not require decomposition since they have no operations that are not supported by the WASM ISA.

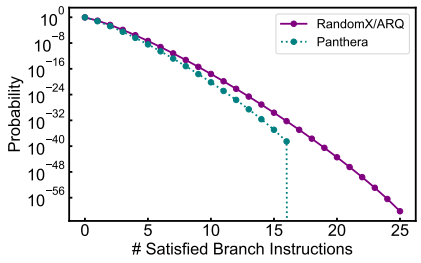


Fig. 6: Probability of satisfied branch instructions.

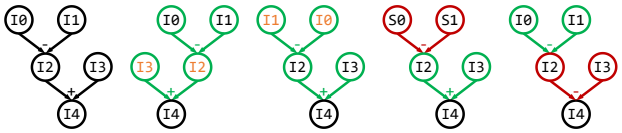


Fig. 7: Examples of graph isomorphism, where "I" represents the 64-bit integer type and "S" represents the 32-bit integer type. Graph components a, b, and c are all isomorphic, while d and e are partially isomorphic with a, b, and c.

Constructing EPGs. We next transform a dynamic mining program into an EPG. An EPG is a directed acyclic graph where each node represents a vector element together with the arithmetic operation that produces it, and each edge represents the propagation of data from one element to another element accompanied by the operation.

For each RandomX instruction, Vectra constructs corresponding graph nodes and edges based on the instruction decomposition. As exemplified in Figure 1a, suppose we have three RandomX instructions. We first decompose them into the form that is compatible with WASM (Figure 1b). Figure 1c shows the resulting EPG for the exponents of floating points in the example of Figure 1b. We also model element propagation of scalar instructions into EPGs, as a single scalar instruction can still be vectorized with other adjacent instructions. In addition, for immediate values in an instruction, we transform them into a new graph node without the preceding operator.

The construction of the EPG is only a static analysis over the dynamic program, and we cannot determine which address a branch instruction actually jumps to if the jump condition is satisfied. Thus, during the EPG construction, Vectra assumes that branch instructions are all not satisfied and the program counter does not jump but simply increases by one. This is reasonable as the probability of a single RandomX branch instruction's condition to be satisfied is only 0.39% according to RandomX specifications [10]. As shown in Figure 6, the probability of no satisfied branch instruction in a dynamic mining program is 90.7% for RandomX and RandomARQ, and 93.9% for Panthera. Also, the probability of one satisfied branch instruction in a dynamic program is merely 8.9% for RandomX and RandomARQ, and 5.9% for Panthera. For the probability of more than two branch instructions satisfied in a dynamic program, it becomes negligible (<0.01%). Of course, such an assumption will lead to incorrect results when the branch condition is actually met during the program execution. We will show our approach based on the concept of symbolic constraints to addressing this issue in §III-D

C. Efficient Program Transformation

Having obtained the EPG for a dynamic program of a new transaction, Vectra searches for isomorphic propagation paths in the graph to rearrange vectorizable data elements and merge them into WASM SIMD instructions. Vectra determines two element propagation paths are isomorphic if their corresponding graph components are non-overlapped with the same data type, operator sequence, and topological structure.

As exemplified in Figure 7, graph components a, b, and c are all isomorphic. For propagation b, the operands $I2$ and $I3$ are swapped compared with a. However, we still regard a and b as isomorphic as their topological structures are the same (i.e., two data elements first propagate to a new element over the "$-$" operation, and this new element then propagates with another element over the "$+$" operation). Besides, although the operands $I0$ and $I1$ in propagation c are swapped compared with a, we still regard a and c as isomorphic since their topological structures are the same. In practice, we can load counterpart elements from these isomorphic components with a proper order to achieve SIMD execution, e.g., we can load $I0$ of a and $I1$ of c into one vector, and $I1$ of a and $I0$ of c into another vector for a vectorized "$-$" operation.

For components d and e, they are not isomorphic with a, b, or c. The "$-$" operator in d operates on two 32-bit integers, which differ from the 64-bit integers in a, b, and c. Such a difference makes the "$-$" operation unable to be vectorized with SIMD instructions in WASM, as all SIMD instructions only support data elements with the same data type. For the element propagation in e, its operator sequence (i.e., the two "$-$" operators) is different from that in a, b, and c, which cannot be vectorized in WASM, either. Thus, propagation e is also not isomorphic with the other three components. It is worth noting that d and e are *partially* isomorphic with a, b, and c, as they have common sub-components (i.e., nodes in green). Besides, we determine memory operations are isomorphic only when they have consecutive memory accesses based on the WASM ISA features [49].

Searching for isomorphic components. To pinpoint all isomorphic components in an EPG, a direct method is to compare each pair of graph nodes with the same operator and the same data type, and then backtrack their parent nodes. Unfortunately, this method enumerates all combinations, incurring a time complexity of $O(n!)$ where n is the number of graph nodes. To address this, Vectra adopts a branch-and-bound strategy when searching for isomorphic components, which only considers the nodes that are close to each other. This is based on our key observation that a dynamic mining program mainly involves the following two types of isomorphism that present a strong locality in data dependencies:

- **Intra-instruction isomorphism.** After the decomposition of vector instructions, the calculation on each data element of the same data type is emulated separately, but their applied WASM operation sequences are all the same since the emulation on the same data type is the same.
- **Inter-instruction isomorphism.** Two instructions that are close to each other have a higher probability to have isomorphic data dependencies for vectorizations, since they have simpler data dependencies than those of the instructions that are far from each other.

As a result, in order to measure the locality of data dependencies for graph nodes, we define a *locality density* metric between two graph nodes x and y as

$$L(x,y) = L_{intra}(x,y) \cdot L_{inter}(x,y), \tag{1}$$

where $L_{intra}(x,y) = |D(x,r) - D(y,r)| + 1$, and $L_{inter}(x,y) = I(x,y)+1$. Specifically, $D(x,r)$ and $D(y,r)$ are the depth of x and y to their common ancestor r in the graph. If they have no common ancestor r, we set $D(x,r) = 0$ and $D(y,r) = 0$. During the emulation of a vector instruction in WASM, each data element forms its own data dependency in different steps of the emulation, and thus the graph nodes from different data elements at the same emulation step (i.e., at the same depth to the common ancestor r) have a higher probability to be vectorized. $I(x,y)$ is the number of intermediate instructions between the instructions that produce the graph nodes x and y. This corresponds to the inter-instruction isomorphism where two graph nodes close to each other with a smaller $I(x,y)$ have a higher probability to be vectorized.

The locality metric in Equation 1 represents the probability of two graph nodes to be vectorizable. A smaller L indicates that two nodes are more likely to have isomorphic precedent element propagation. With this metric, Vectra clusters graph nodes into different subgraphs with the DBSCAN algorithm [50] to bound the searching space, by only searching isomorphic propagation within a cluster. Specifically, Vectra first compares each leaf node in the cluster. If they have the same operator on the same data type, Vectra backtracks to their parent nodes respectively. Once they have no parent nodes or their parent nodes are different in either the operator or the data type, Vectra stops the searching process and records the current isomorphic propagation.

Element merging and instruction emission. After pinpointing all isomorphic element propagation paths based on the clustered graph nodes, Vectra rearranges the data elements for each isomorphic graph component onto the WASM processing stack to enable their SIMD execution. Vectra emits WASM instructions based on the topological order of graph nodes (i.e., the instructions that have no precedent data dependencies). We map the basic arithmetic operation (e.g., $+$, $-$, and so forth) of graph edges to WASM instructions like `i64.add` and `i64.sub`. For vectorizable isomorphic graph components, Vectra loads them together with their counterpart elements onto the WASM processing stack using WASM SIMD load instructions like `v128.load`. It then applies the SIMD arithmetic operations on them such as `i64x2.add` and `i64x2.sub`. When all graph nodes are processed based on the topological order, we finally get the WASM instructions that are largely vectorized for validating the transaction block.

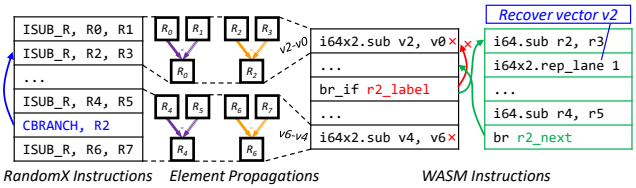


Fig. 8: Incorrect executions of WASM vector instructions caused by branches. The green box is the correct execution.

Complexity analysis. Graph isomorphism calculation and instruction emission are all performed ahead of the execution of a dynamic program. It is important to keep them with a low time consumption to reduce the startup delay for WASM translation during the continuous mining process. Since the construction of the element propagation graph only goes through all instructions once, it incurs a time complexity of $O(k)$, where k is the number of instructions in the original program. During the construction, we can record the depth of each node to each of its parent nodes. The calculation of graph isomorphism based on the locality metric in Equation 1 therefore only incurs a time complexity of $O(n^2)$, where n is the number of graph nodes. Besides, the topological instruction emission process incurs a time complexity of $O(n)$. Thus, the time complexity of the whole transformation process is $O(k + n^2 + n)$. In practice, the number of generated graph nodes for a RandomX instruction is no more than a constant value c. The overall time complexity is then under $O(k + c^2 \cdot k^2 + ck) = O(k^2)$. We show in §V that such a complexity is satisfactory for web cryptomining.

D. Runtime Symbolic Correction

When handling conditional branch instructions, the aforementioned EPG can induce incorrect results. This is because Vectra assumes that the conditions are always not satisfied, so as to reduce the complexity during the graph construction. However, the actual execution may perform the jump operation when the condition is met. As exemplified in Figure 8, if we consider that the `CBRANCH` instruction is not satisfied (which holds true in most cases as we mentioned in §III-B), the first two `ISUB_R` instructions can be vectorized into a single WASM SIMD instruction (i.e., `i64x2.sub` on vectors `v0` and `v2`). However, when the `CBRANCH` instruction is satisfied in between, the translated WASM instructions become incorrect. In the example of Figure 8, we cannot simply jump to the first `i64x2.sub` instruction since in this case the data elements in `v0` and `v2` are not consistent with those in the original mining program. In fact, the branch instructions will jump to the last instruction that modifies the target register as detailed in RandomX specifications [28], i.e., `R2` in Figure 8. Thus, the correct execution should jump to a scalar instruction

that performs subtractions on R2 and R3 (i.e., i64.sub in the example correct executions).

Meanwhile, when the propagation of data elements is across a branch instruction in the original mining program (i.e., the two ISUB_R instructions right before and after the CBRANCH instruction in Figure 8), we cannot directly execute the vectorized instructions of such interrupted propagation when the branch condition is satisfied. This is because in the example, only the instructions before the satisfied branch instruction are executed, while the instructions after the branch are not. Instead, we should execute the original instructions before the branch instruction, and then perform actual jump operations in WASM (i.e., the i64.sub and br instructions at the end of the green box in Figure 8).

Symbolic constraints. To address the issue, a direct solution is to record the dynamic program execution at runtime, which incurs a high overhead. We thus introduce symbolic constraints, which capture (and simplify) the conditions where a branch instruction is satisfied, so that we can ensure correct branching in a speculative manner. Concretely, after generating the dynamic program, Vectra calculates the range of a tested register value that will cause the program counter to jump to a specific instruction. Such a range (i.e., the symbolic constraint) limits the register values of different execution paths.

Specifically, according to the RandomX specifications [10], a branch condition is satisfied and the program counter jumps when the following equation derived from the corresponding branch instruction is equal to zero

$$(R_x + IMM \& (\sim (1 \ll (S-1)))) \,\&\, (\texttt{0xFF} \ll S), \quad (2)$$

where R_x is the destination register accumulated with the immediate value IMM, and S is the shift value. They are all encoded in a branch instruction of the RandomX-series algorithms. Based on Equation 2, we can derive the concise symbolic constraints on the value of the register Rx that satisfies the branch condition. For example, if $S = 2$ and $IMM = 3$, the symbolic constraint of the value in register Rx that satisfies the branch condition is $(Rx + 1)\,\&\texttt{0x3FC} = 0$. At the offline phase, we calculate all the symbolic constraints for the RandomX branch instructions.

With these constraints, Vectra builds a jump table, where each table entry records the target instruction address and the recovery instructions (as exemplified in the green box of Figure 8). At runtime, only before executing the vectorized instruction that is across a branch instruction, Vectra tests whether the relevant symbolic constraints are satisfied based on the (emulated) register value. If so, Vectra switches to the corresponding scalar instructions in the jump table instead of continuing the SIMD execution to recover correctness. We will show in §V-B that compared with the real-time monitoring approach, we speed up the branch recovery process by 4.5×.

IV. Implementation

We implement Vectra on top of RandomX with 800 lines of JavaScript code and 5K lines of C/C++ code. We modify the byte code machine module in the RandomX reference implementation [10] to translate its instructions into WASM instructions. To compile the resulting translations into WASM byte code, we use Emscripten emcc v3.1 [41] as the backend compiler. We disable the RANDOMX_FLAG_FULL_MEM flag in RandomX to limit the memory usage to a low level (<512 MB), so that Vectra can work well on low-end devices. Also, for mobile devices, we dynamically adjust the computation intensity (e.g., the number of active threads) based on the device's in-situ overhead to avoid potential side effects on user experience. Besides, we fine-tune the neighborhood parameter and the minimum number of points for the EPG node clustering process through an offline test over their parameter grids, seeking for a balance between the number of generated SIMD instructions (speedup) and the time cost of program transformation (overhead). Once Vectra receives a mining job from the mining pool over the Stratum protocol [51], it initializes the memory in WASM for the translations and executions of the RandomX instructions.

Deployment strategies. Vectra can be directly integrated into the official websites of mainstream cryptocurrencies. Web users can register an account on the official website and start mining without any further actions. Third-party websites can also provide mining services in a similar way, or even serverless mining microservices by means of stateless/stateful cloud functions. As Vectra runs as long as a web page is opened, users are able to easily mine on either their PCs or mobile phones. Moreover, web users can simply install a browser plugin of Vectra without any configurations on the host OS. They only need to login to their cryptocurrency accounts and then the plugin can automatically start mining in background when they are surfing on the web. In this way of allowing users to mine on their devices with minimal efforts, Vectra can significantly enhance the democracy of the cryptocurrency ecosystem. We also explore the deployment of Vectra on the emerging serverless platforms (detailed in https://WebCryptomining.github.io), which involves a distributed metalog synchronization mechanism for stateful functions to achieve strong consistency and high reliability.

Generalizability. Currently, at least 26 cryptocurrencies such as Monero and Zephyr have enforced RandomX-series algorithms [52], indicating that RandomX is becoming the de-facto standard to combat mining farms. Thus, for a long time in the future, Vectra can be directly applied to relevant variants. Nevertheless, the design principle of Vectra is not specific to RandomX as its general idea is to leverage isomorphic patterns during program transformation across different ISAs. As a result, for cryptocurrencies that need to enforce other algorithms in the far future, Vectra can still be adapted with affordable efforts. We only need to determine the mapping between algorithm-specific operations and WASM instructions, which is a one-shot effort. In essence, Vectra can be reused by different mining algorithms for efficient instruction transformations to adapt to the web environments.

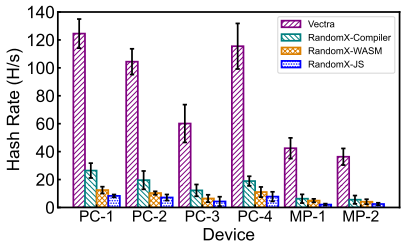


Fig. 9: Comparing the hash rates of Vectra with the others.

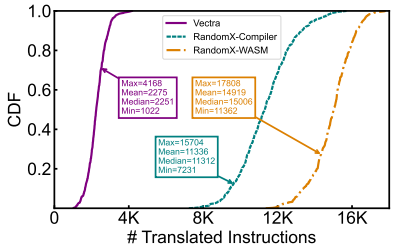


Fig. 10: Comparing the number of translated instructions.

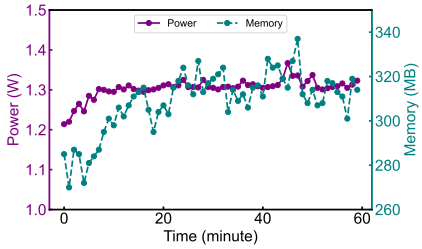


Fig. 11: Resource consumptions of Vectra on mobile phones.

V. Evaluation

A. Experiment Setup

We compare the performance of Vectra with the compiler-optimized RandomX over traditional vectorization schemes like SLP (denoted as RandomX-Compiler) [29], [30], as well as the default RandomX implementations on WASM and JavaScript (denoted as RandomX-WASM and RandomX-JS). The basic settings of our testbed are similar to those introduced in §II. We connect the three implementations to the MoneroOcean [53] mining pool for running the same set of real-world mining tasks. We use both PCs and mobile phones as listed in Table I for the evaluation, which cover the mainstream CPU architectures (i.e., x86 and ARM CPUs from four vendors including Intel, Apple, Qualcomm, and HiSilicon). Note that we can mine cryptocurrencies on mobile phones even if the cryptocurrency does not natively support it, as mobile devices can visit mining pages through browsers.

B. Effectiveness on Cryptomining

Overall efficiency. Figure 9 shows the hash rate of the compared approaches. RandomX-WASM bears a quite low efficiency, which has only a hash rate of 12.4 H/s even on the high-end PC-1 (with 8 cores at 2.90 GHz and a maximum turbo frequency of 5.30 GHz [54]). RandomX-JS is even worse, with a hash rate of only 8.3 H/s on PC-1. For mobile phones, the hash rates of the default RandomX-WASM and RandomX-JS are all below 5 H/s. Such performance is similar to the results for the Panthera algorithm of Scala (cf. §II).

Although RandomX-Compiler can improve the hash rate to some extent, its speed is still very low and not practical for web cryptomining. For example, on PC-1, RandomX-Compiler achieves a hash rate of 26.42 H/s, which is only 2× faster than RandomX-WASM. In some cases, we also observe that RandomX-Compiler is even slower than RandomX-WASM. This is easy to understand because the complex conditional jumps of the RandomX instruction set make RandomX-Compiler unable to optimize the mining program proactively.

In contrast, Vectra significantly improves the hash rate. As shown, Vectra achieves a hash rate of 124.53 H/s, 104.42 H/s, and 60.12 H/s on x86-based PC-1, PC-2, and PC-3 respectively, which are 10× faster than that of RandomX-WASM. For ARM-based PC-4, Vectra achieves a hash rate of 115.50 H/s, which is also at least 10× faster. For mobile devices, Vectra achieves a hash rate of 42.42 H/s and 36.32 H/s for MP-1 and MP-2, respectively, which are 8× faster. Such a high efficiency makes web cryptomining practical and profitable in the real world (cf. §V-D).

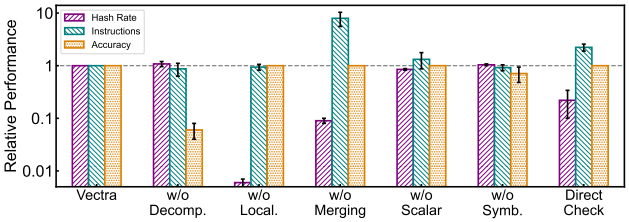


Fig. 12: Performance breakdown of Vectra.

Instruction reduction. To understand Vectra's improvements in web mining performance, we analyze the resulting WASM instruction patterns. Figure 10 shows the number of translated instructions of WASM-based approaches. RandomX-WASM translates an average transaction into nearly 15K instructions (i.e., 58.5 WASM instructions for an average RandomX instruction), while RandomX-Compiler translates an average transaction into 11K instructions (i.e., 44.2 WASM instructions for an average RandomX instruction). In contrast, Vectra significantly reduces the number of translated instructions by merging isomorphic element propagation paths to transform a mining program in-situ, which only translates an average transaction into 2.3K WASM instructions (i.e., 8.9 WASM instructions for an average RandomX instruction). Compared to RandomX-WASM, Vectra reduces the number of translated instructions by ∼7×. The actual performance improvements (∼10× for PCs and ∼8× for mobile phones) of Vectra are slightly higher than the reduced number of translated instructions (∼7×). This is because after merging the instructions, the number of time-consuming memory operations is further reduced as many elements are operated on together in a single vectorized memory operation.

C. Performance Breakdown

We perform an ablation study to demonstrate the effectiveness of the components and design choices in Vectra, by disabling or replacing each component separately in program transformation. When instruction decomposition is removed,

we directly translate RandomX instructions into approximate WASM instructions. For example, we simply use 64-bit integers to achieve 64-bit integer multiplication (which is the most suited type supported by WASM but could still result in overflow). Similarly, we directly use the native 64-bit floating-point numbers of WASM for floating-point operations, without considering the precision problem introduced by directed rounding. When locality-aware isomorphism searching is disabled, we search for isomorphic propagation paths over the whole EPG. When instruction merging is disabled, we do not merge the discovered isomorphic element propagation paths. We also disable the rearrangement for scalar elements, which accounts for a small portion of computation (cf. §II). Besides, when runtime symbolic correction is disabled, we do not add correction byte code and do not correct the data arrangements at runtime. Further, we replace the runtime symbolic correction with a direct monitoring mechanism that examines all the previously executed instructions for comparison.

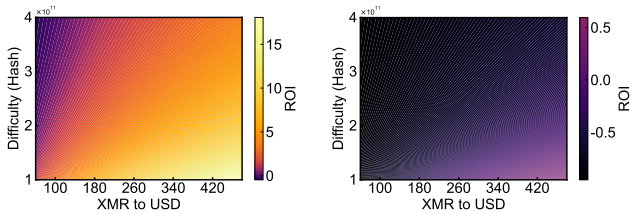
Effectiveness of program transformation. As shown in Figure 12, when we disable any of the components related to program transformation including instruction decomposition, locality-aware isomorphism searching, as well as instruction merging, the performance of Vectra on PC-1 degrades significantly. Specifically, when instruction decomposition is disabled, the accuracy of Vectra drops to 6%, which makes web cryptomining impractical. This is because the approximate translation of RandomX instructions into WASM introduces precision loss, particularly for floating-point operations. When we disable locality-aware isomorphism searching, the accuracy does not drop but the hash rate drops significantly by >99%. This is because searching for isomorphic element propagation paths over the entire EPG bears prohibitively high overhead.

Note that we use the locality metric (and the corresponding branch-and-bound search strategy) to reasonably balance between speedup and complexity, and thus can indeed miss some opportunities for cross-cluster isomorphism and performance gain. For small-scale EPGs that can be exhaustively examined in acceptable time, the loss lies between 5%–12%.

In addition, disabling element migration and instruction merging leads to a 91% (i.e., ∼11×) drop in the hash rate. Such a performance lines up with the performance of RandomX-WASM on PCs, which is ∼10× slower.

Optimizations on scalars. The ablation study also demonstrates the effectiveness on rearranging scalar elements together with vector elements. When we disable the rearrangement for scalar elements, the performance of Vectra drops by 15%, and the number of resulting WASM instructions increases by 32%. This indicates that except for vector operations that dominate the computation in web mining, merging similar scalar propagation also has the potential to improve efficiency of web mining, despite the less significant improvements compared to merging vector elements.

Benefits of symbolic correction. As shown in Figure 12, when we disable symbolic corrections, the hash rate improves slightly by 4%. However, the accuracy drops by 29%. These incorrect results arise when the branch conditions are satisfied and the data need rearrangements. Compared with the minor degradation in hash rate, the symbolic correction achieves accurate mining results, which is crucial to obtain profits in the real world. We also compare the symbolic correction with a direct monitoring mechanism that examines all instructions. Such a direct checking results in a 78% hash rate drop and a 123% increase in the number of translated instructions, which is 4.5× less efficient than Vectra.



(a) Vectra (b) RandomX-WASM

Fig. 13: ROI with Vectra and RandomX-WASM.

Traffic overhead. Vectra incurs additional traffic on users' devices, which mainly comes from the compiled WASM binaries for transforming RandomX instructions. Nevertheless, the incurred traffic is negligible compared to the overall traffic of a web page which is usually a couple of MBs. According to our measurements, the total additional traffic of Vectra is only 34 KB after the `gzip` compression on average.

Performance on mobile devices. We evaluate Vectra on mobile devices that have significant resource constraints. As shown in Figure 11, at the beginning the power consumption of Vectra slightly increases when the device temperature goes up. Then, it remains stable at around 1.3 W, while the memory consumption remains stable at 310 MB, which will not affect the usage of mobile devices. Although RandomX-WASM has very similar resource consumption, its hash rate is much lower.

D. Economic Analysis

We model the economic profits of users who use Vectra to conduct web cryptomining. The cost of deploying Vectra and conducting web cryptomining comes from the required electricity and network traffic. Thus, suppose that the electricity price of the user is E per kilowatt-hour (kWh) and the Internet service price is N per month. Then, the cost of web cryptomining for a month is

$$Cost = Pt/E + N, \tag{3}$$

where P is the power of the CPU used by the user, and t is the total seconds in a month (i.e., 2.592×10^6 seconds). The revenue of web cryptomining comes from mining rewards. It can be represented as

$$Revenue = RHt/D, \tag{4}$$

where H is the total hash rate of Vectra over the web, D is the difficulty of mining the cryptocurrency, and R is the exchange

rate of USD against Monero. Thus, the return-on-investment (ROI) of web cryptomining for a month is

$$ROI = \frac{Revenue - Cost}{Cost} = R \times \frac{H/D}{P/E + N/t} - 1. \quad (5)$$

The electricity price in the USA is 0.16 USD/kWh on average, and the price of Internet service is 2.75 USD/GB [55], [56]. The average CPU power can be regarded as 30 W for mobile phones and PCs [57]. We can then calculate the ROI of web cryptomining based on the average hash rate. Figure 13a shows the ROI with Vectra regarding mining difficulties and exchange rates. A user can easily obtain a positive (and satisfying) ROI. For example, the ROI on Aug. 1st, 2024 is 120%, when the difficulty of Monero is 281.73 GH [58] and the exchange rate is 156.1324 USD/XMR [59]. However in Figure 13b, users using RandomX-WASM can hardly achieve a satisfying ROI and in most cases their ROI is negative.

It should be noted that the real-world ROI might be affected by more factors. Browser idle time and intermittent mining behaviors can both reduce the hash rate and ROI. Quantitatively, if the idle rates are 10% and 20%, the ROI would drop to 98% and 76% respectively, which however are still positive. On the other side, most countries (e.g., USA, UK, and China) enforce time-of-use pricing, where the electricity price is much lower (or even free) during non-peak hours. Also, browser caching can help save the network traffic for cryptomining. These factors improve the ROI in practice.

VI. Related Work

Cryptomining and its algorithms. Over the past decade, there has been a surge of various mining algorithms [60]–[63] for different cryptocurrencies. For example, Bitcoin [1] uses the SHA-256 function, while Monero used to adopt the CryptoNight algorithm [64]. They can be efficiently executed on dedicated hardware like ASICs and FPGAs used by large mining farms, and thus pose threats to the democracy of the cryptocurrency ecosystem. In contrast, the recently proposed RandomX algorithm [10] is designed to resist the dedicated hardware, and has been widely adopted by cryptocurrencies like Monero. In this work, we propose a new approach for efficient and profitable web cryptomining based on the RandomX framework, which exploits the intrinsic isomorphism among element propagations of vectors in RandomX-series algorithms to accelerate web-based mining.

Cryptojacking and its detection. As an important security topic, there have been a number of methods and tools [18], [19], [65] to detect and defend against cryptojacking (which hijacks user devices for illegal cryptomining and monetization). For example, MineSweeper [65] identifies web cryptojacking attacks based on the intrinsic properties of the mining algorithm such as specific cryptographic operations and memory access patterns. MineThrottle [18] is another tool that detects web cryptojacking by instrumenting WASM code on the fly and throttles drive-by mining behaviors based on a user-configurable policy. In essence, democratizing legitimate web mining can open doors for potential abuses of the capability of Vectra in cryptojacking. However, we argue that the defense should focus on addressing malicious websites, instead of taking away democracy itself.

Instruction translation and translation. Instruction translation is a common technique used in system-level emulation [66]–[69]. Prior studies have proposed various techniques to achieve efficient instruction translation between different ISAs. QEMU [70] is a fast and portable dynamic binary translation system featured by block translation and caching. In recent years, learning-based approaches are used to generate the principled rules for system-level dynamic binary translation [71]. During code translation and compilation, a number of optimization techniques can be used, such as outer-loop vectorization and register reorganization [72]–[74]. Differently, Vectra leverages the isomorphism among element propagation paths of vectors in the mining algorithms to accelerate the web mining process. Such isomorphism is rarely used in prior work to optimize the translated binaries.

VII. Conclusion

As a keystone of the cryptocurrency ecosystem, democracy has been severely threatened by the monopolization of mining farms in recent years. Existing countermeasures by enforcing new mining algorithms can effectively address this, but on the other hand have rendered web cryptomining unprofitable and constantly fading. In this paper, we propose the methodology of *just-in-time transformation* of mining programs and implement Vectra to make web cryptomining efficient, in the hope of reviving the democracy. This is achieved by discovering and exploiting the pervasive isomorphic propagation paths of vector elements in the mining programs. Our evaluation with real-world mining tasks and devices validates the effectiveness of Vectra. In a broader sense, our study provides a new perspective on the translations of vector instructions across different ISAs, which can be applied to building high-performance system-level emulators and binary translators.

Acknowledgment

We thank the anonymous ASE reviewers for their valuable comments and suggestions. Also, we thank Xinlei Yang and Jianwei Zheng for the insightful discussions in the early stage of this study. This work is supported in part by the National Key R&D Program of China under grant 2022YFB4500703 and the National Natural Science Foundation of China under grants 62332012 and 62472245.

References

[1] R. Böhme, N. Christin, B. Edelman, and T. Moore, "Bitcoin: Economics, Technology, and Governance," *Journal of Economic Perspectives*, vol. 29, no. 2, pp. 213–238, 2015.

[2] R. McLaughlin, C. Kruegel, and G. Vigna, "A Large Scale Study of the Ethereum Arbitrage Ecosystem," in *Proc. of USENIX Security*, 2023, pp. 3295–3312.

[3] D. A. Wijaya, J. K. Liu, R. Steinfeld, D. Liu, and J. Yu, "On The Unforkability of Monero," in *Proc. of ACM Asia CCS*, 2019, pp. 621–632.

[4] K. Yan, J. Zhang, X. Liu, W. Diao, and S. Guo, "Bad Apples: Understanding the Centralized Security Risks in Decentralized Ecosystems," in *Proc. of ACM WWW*, 2023, pp. 2274–2283.

[5] X. T. Lee, A. Khan, S. Sen Gupta, Y. H. Ong, and X. Liu, "Measurements, Analyses, and Insights on the Entire Ethereum Blockchain Network," in *Proc. of ACM WWW*, 2020, pp. 155–166.

[6] "Who and What Is Coinhive? – Krebs on Security," https://krebsonsecurity.com/2018/03/who-and-what-is-coinhive/, 2018.

[7] "Crypto Mining Service Coinhive to Call it Quits – Krebs on Security," https://krebsonsecurity.com/2019/02/crytpo-mining-service-coinhive-to-call-it-quits/, 2019.

[8] "Introduction to Oxen," https://docs.oxen.io/oxen-docs/, 2023.

[9] "Scala - Mobile, Distributed and Anonymous," https://scalaproject.io/, 2024.

[10] tevador, "RandomX," https://github.com/tevador/RandomX, 2024.

[11] "Arqma/RandomARQ: Proof of Work Algorithm Based on Random Code Execution," https://github.com/arqma/RandomARQ, 2023.

[12] "Scala-Network/Panthera: Proof of Work Algorithm Based on Random Code Execution," https://github.com/scala-network/Panthera, 2023.

[13] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks," in *Proc. of ACM CCS*, 2003, pp. 281–289.

[14] Y. Chen, C. Mendis, M. Carbin, and S. Amarasinghe, "VeGen: A Vectorizer Generator for SIMD and Beyond," in *Proc. of ACM ASPLOS*, 2021, pp. 902–914.

[15] K. Vipin and S. A. Fahmy, "FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications," *ACM Computing Surveys*, vol. 51, no. 4, pp. 1–39, 2018.

[16] J. Nurmi, *Processor Design: System-on-Chip Computing for ASICs and FPGAs*. Springer Science & Business Media, 2007.

[17] "Blog – Coinhive – Monero Mining Club," https://web.archive.org/web/20190405011157/https://coinhive.com/blog#:~:text=to%20be%20completely%20honest,%20it%20isn%E2%80%99t%20economically%20viable%20anymore, 2019.

[18] W. Bian, W. Meng, and M. Zhang, "MineThrottle: Defending against WASM In-Browser Cryptojacking," in *Proc. of ACM WWW*, 2020, pp. 3112–3118.

[19] A. Kharraz, Z. Ma, P. Murley, C. Lever, J. Mason, A. Miller, N. Borisov, M. Antonakakis, and M. Bailey, "Outguard: Detecting in-Browser Covert Cryptocurrency Mining in the Wild," in *Proc. of ACM WWW*, 2019, pp. 840–852.

[20] F. Victor and A. M. Weintraud, "Detecting and Quantifying Wash Trading on Decentralized Cryptocurrency Exchanges," in *Proc. of ACM WWW*, 2021, pp. 23–32.

[21] D. Lin, J. Wu, Y. Yu, Q. Fu, Z. Zheng, and C. Yang, "DenseFlow: Spotting Cryptocurrency Money Laundering in Ethereum Transaction Graphs," in *Proc. of ACM WWW*, 2024, pp. 4429–4438.

[22] A. Robertson, "Salon Asks Ad-Blocking Users to Opt into Cryptocurrency Mining Instead," https://www.theverge.com/2018/2/13/17008158/salon-suppress-ads-cryptocurrency-mining-coinhive-monero-beta-testing, 2018.

[23] "The UNICEF CryptoFund | UNICEF Office of Innovation," https://www.unicef.org/innovation/stories/unicef-cryptofund, 2020.

[24] B. L. Titzer, "A Fast In-Place Interpreter for Webassembly," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA2, pp. 646–672, 2022.

[25] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey, "Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1313–1324, 2008.

[26] J. M. Muller, N. Brisebarre, F. De Dinechin, C. P. Jeannerod, V. Lefevre, G. Melquiond, N. Revol, D. Stehlé, S. Torres *et al.*, *Handbook of Floating-Point Arithmetic*. Springer, 2018.

[27] J. Van Hoey, *Beginning X64 Assembly Programming*. Springer, 2019.

[28] "CBRANCH Instruction in RandomX," https://github.com/tevador/RandomX/blob/master/doc/specs.md, 2024.

[29] Y. Chen, C. Mendis, and S. Amarasinghe, "All You Need Is Superword-Level Parallelism: Systematic Control-Flow Vectorization with SLP," in *Proc. of ACM PLDI*, 2022, pp. 301–315.

[30] D. Nuzman, I. Rosen, and A. Zaks, "Auto-Vectorization of Interleaved Data for SIMD," in *Proc. of ACM PLDI*, 2006, pp. 132–143.

[31] P. Civicioglu, "Backtracking Search Optimization Algorithm for Numerical Optimization Problems," *Applied Mathematics and Computation*, vol. 219, no. 15, pp. 8121–8144, 2013.

[32] E. L. Lawler and D. E. Wood, "Branch-and-Bound Methods: A Survey," *Operations Research*, vol. 14, no. 4, pp. 699–719, 1966.

[33] T. Wang, T. Wei, Z. Lin, and W. Zou, "IntScope: Automatically Detecting Integer Overflow Vulnerability in x86 Binary Using Symbolic Execution," in *Proc. of NDSS*, 2009.

[34] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A Survey of Symbolic Execution Techniques," *ACM Computing Surveys*, vol. 51, no. 3, pp. 1–39, 2018.

[35] G. T. Friedlob and F. J. Plewa Jr, *Understanding Return on Investment*. John Wiley & Sons, 1996.

[36] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder, *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, 2016.

[37] W. Liu, K. Qian, Z. Li, F. Qian, T. Xu, Y. Liu, Y. Guan, S. Zhu, H. Xu, L. Xi, C. Qin, and E. Zhai, "Mitigating Scalability Walls of RDMA-based Container Networks," in *Proc. of USENIX NSDI*, 2025.

[38] W. Liu, K. Qian, Z. Li, T. Xu, Y. Liu, W. Wang, Y. Zhang, J. Li, S. Zhu, X. Li, H. Xu, F. Feng, and E. Zhai, "SkeletonHunter: Diagnosing and Localizing Network Failures in Containerized Large Model Training," in *Proc. of ACM SIGCOMM*, 2025, pp. 527–540.

[39] R. Pass and A. Shelat, "Micropayments for Decentralized Currencies," in *Proc. of ACM CCS*, 2015, pp. 207–218.

[40] L. Luu, Y. Velner, J. Teutsch, and P. Saxena, "SmartPool: Practical Decentralized Pooled Mining," in *Proc. of USENIX Security*, 2017, pp. 1409–1426.

[41] "Emscripten Tutorial — Emscripten 3.1.51-Git (dev) Documentation," https://emscripten.org/docs/getting_started/Tutorial.html, 2023.

[42] W. Liu, X. Yang, H. Lin, Z. Li, and F. Qian, "Fusing Speed Index during Web Page Loading," in *Proc. of ACM SIGMETRICS*, 2022, pp. 1–23.

[43] W. Liu, X. Yang, Z. Li, and F. Qian, "SkipStreaming: Pinpointing User-Perceived Redundancy in Correlated Web Video Streaming Through the Lens of Scenes," in *Proc. of ACM MM*, 2023, pp. 3944–3953.

[44] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg, "Virtual Machine Showdown: Stack Versus Registers," *ACM Transactions on Architecture and Code Optimization*, vol. 4, no. 4, 2008.

[45] R. Zheng and S. Pai, "Efficient Execution of Graph Algorithms on CPU with SIMD Extensions," in *Proc. of IEEE/ACM CGO*, 2021, pp. 262–276.

[46] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and JF. Bastien, "Bringing the Web up to Speed with WebAssembly," in *Proc. of ACM PLDI*, 2017.

[47] A. Rossberg, "WebAssembly Specification," *WebAssembly Community Group*, vol. 2, 2021.

[48] W. Kahan, "IEEE Standard 754 for Binary Floating-Point Arithmetic," *Lecture Notes on the Status of IEEE*, vol. 754, no. 94720-1776, p. 11, 1996.

[49] "WebAssembly Opcode Table," https://pengowray.github.io/wasm-ops/, 2024.

[50] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, "DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN," *ACM Transactions on Database Systems*, vol. 42, no. 3, 2017.

[51] M. Dotan, Y.-A. Pignolet, S. Schmid, S. Tochner, and A. Zohar, "Survey on Blockchain Networking: Context, State-of-the-Art, Challenges," *ACM Computing Surveys*, vol. 54, no. 5, 2021.

[52] "Monero (XMR) RandomX | Mining Pools," https://miningpoolstats.stream/monero, 2025.

[53] "Monero (XMR) Mining Pool - MoneroOcean," https://moneroocean.stream/, 2024.

[54] S. Kondguli and M. Huang, "A Case for a More Effective, Power-Efficient Turbo Boosting," *ACM Transactions on Architecture and Code Optimization*, vol. 15, no. 1, 2018.

[55] "Average Price: Electricity per Kilowatt-Hour in U.S. City Average," https://fred.stlouisfed.org/series/APU000072610, 2024.

[56] "Cellular Data Average Price in US 2018-2023," https://www.statista.com/statistics/994913/average-cellular-data-price-per-gigabyte-in-the-us/, 2024.

[57] A. Carroll and G. Heiser, "An Analysis of Power Consumption in a Smartphone," in *Proc. of USENIX ATC*, 2010.

[58] "Monero Difficulty Chart - XMR Difficulty," https://www.coinwarz.com/mining/monero/difficulty-chart, 2024.

[59] "Monero (XMR) Price, Real-time Quote & News," https://www.google.com/finance/quote/XMR-USDT, 2024.

[60] A. Romano, Y. Zheng, and W. Wang, "Minerray: Semantics-Aware Analysis for Ever-Evolving Cryptojacking Detection," in *Proc. of IEEE/ACM ASE*, 2020, pp. 1129–1140.

[61] W. Wang, "Empowering Web Applications with WebAssembly: Are We There Yet?" in *Proc. of IEEE/ACM ASE*, 2021, pp. 1301–1305.

[62] Z. Chen, Y. Liu, S. M. Beillahi, Y. Li, and F. Long, "Demystifying Invariant Effectiveness for Securing Smart Contracts," in *Proc. of ACM FSE/ESEC*, vol. 1, 2024, pp. 1772–1795.
[63] P. Garamvölgyi, Y. Liu, D. Zhou, F. Long, and M. Wu, "Utilizing Parallelism in Smart Contracts on Decentralized Blockchains by Taming Application-Inherent Conflicts," in *Proc. of ACM/IEEE ICSE*, 2022, pp. 2315–2326.
[64] Z. Feng and Q. Luo, "Evaluating Memory-Hard Proof-of-Work Algorithms on Three Processors," *Proceedings of the VLDB Endowment*, vol. 13, no. 6, pp. 898–911, 2020.
[65] R. K. Konoth, E. Vineti, V. Moonsamy, M. Lindorfer, C. Kruegel, H. Bos, and G. Vigna, "MineSweeper: An In-depth Look into Drive-by Cryptocurrency Mining and Its Defense," in *Proc. of ACM CCS*, 2018, pp. 1714–1730.
[66] P. Kedia and S. Bansal, "Fast Dynamic Binary Translation for the Kernel," in *Proc. of ACM SOSP*, 2013, pp. 101–115.
[67] A. D'Antras, C. Gorgovan, J. Garside, and M. Luján, "Low Overhead Dynamic Binary Translation on Arm," in *Proc. of ACM PLDI*, 2017, pp. 333–346.
[68] B. Hawkins, B. Demsky, D. Bruening, and Q. Zhao, "Optimizing Binary Translation of Dynamically Generated Code," in *Proc. of IEEE/ACM CGO*, 2015, pp. 68–78.
[69] D. Y. Hong, C. C. Hsu, P. C. Yew, J. J. Wu, W. C. Hsu, P. Liu, C. M. Wang, and Y. C. Chung, "HQEMU: A Multi-Threaded and Retargetable Dynamic Binary Translator on Multicores," in *Proc. of IEEE/ACM CGO*, 2012, pp. 104–113.
[70] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *Proc. of USENIX ATC*, 2005.
[71] J. Jiang, C. Liang, R. Dong, Z. Yang, Z. Zhou, W. Wang, P.-C. Yew, and W. Zhang, "A System-Level Dynamic Binary Translator Using Automatically-Learned Translation Rules," in *Proc. of IEEE/ACM CGO*, 2024, pp. 423–434.
[72] D. Nuzman and A. Zaks, "Outer-Loop Vectorization: Revisited for Short SIMD Architectures," in *Proc. of ACM PACT*, 2008, pp. 2–11.
[73] H. Zhou and J. Xue, "Exploiting Mixed SIMD Parallelism by Reducing Data Reorganization Overhead," in *Proc. of ACM CGO*, 2016, pp. 59–69.
[74] Y. P. Liu, D. Y. Hong, J. J. Wu, S. Y. Fu, and W. C. Hsu, "Exploiting Asymmetric SIMD Register Configurations in ARM-to-x86 Dynamic Binary Translation," in *Proc. of IEEE PACT*, 2017, pp. 343–355.