# MINERLAMP: Behavior Modeling for Cryptojacking Detection

Zixu Wang[1,2,3], Dongxu Peng[1,2,3], Weiyuan Tong[5], Zheng Wang[6], and Zhanyong Tang[1,2,3,4]
[1]School of Computer Science, Northwest University, Xi'an, Shaanxi, 710127, China
[2]Xi'an Key Laboratory of Advanced Computing and Software Security
[3]Shaanxi Key Laboratory of Passive Internet of Things and Neural Computing
[4]Shaanxi International Joint Research Centre for the Battery-Free Internet of Things
[5]Xiangtan University, Xiangtan, Hunan, China
[6]University of Leeds, Woodhouse Lane, Leeds, West Yorkshire LS2 9JT, United Kingdom

**In-browser cryptojacking attacks, where a website injects malicious code to secretly use a user's computing resources for cryptocurrency mining during web browsing, are a significant concern for many users. Unfortunately, existing approaches for in-browser cryptojacking detection are inadequate as more sophisticated attacking techniques have emerged. This paper presents MINERLAMP, an improved approach for detecting in-browser cryptojacking activities. MINERLAMP models dynamic web browsing events as a multilateral relation graph. It then uses a graph neural network to reason about the relational graph and infer the likelihood of in-browser cryptojacking activities. Our extensive evaluation of real-life website datasets shows that MINERLAMP outperforms eleven state-of-the-art in-browser cryptojacking detection methods by providing a higher detection accuracy with a lower false-positive rate. We showcase that the overhead of MINERLAMP is small, enabling it to successfully detect in-browser cryptojacking activities in less than a minute on a low-end laptop by today's standards.**

*Index Terms*—**In-browser cryptojacking, Evasion techniques, Graph neural networks, Behavior modeling, Multi-relational graph**

## I. INTRODUCTION

Cryptocurrencies such as Bitcoin and Ethereum have attracted significant attention in recent years due to their financial implications. Cryptocurrency transactions and rewards are typically verified and generated by solving proof-of-work (PoW) challenges [1], which require substantial processing power and energy; this process is known as coin mining. This renders cryptocurrency mining an extremely resource-intensive and costly task. Cybercriminals reduce the cost of mining operations by illicitly exploiting the computing and energy resources of other users. This attack, known as cryptojacking, is typically carried out by gaining unauthorized access to end-user systems and covertly performing computational tasks. One example of such a malicious technique is CryptoNight [2].

With the development of WebAssembly (Wasm) [3] and other web technologies such as JavaScript, web browsers have emerged as primary targets for cryptojacking attacks. In in-browser cryptojacking, a malicious script is downloaded when a user visits a compromised or malicious website and is subsequently executed in the background to perform cryptocurrency mining without the user's consent or knowledge [4]. Although many browser-based mining services have become obsolete since the shutdown of Coinhive in March 2019, recent studies indicate that browser mining remains prevalent [5], [6], [7]. This resurgence does not constitute a mere continuation of previously observed threats but instead signifies a qualitative escalation in both technical sophistication and operational complexity. In July 2025, the security firm c/side reported the discovery of a large-scale cryptojacking campaign that compromised more than 3,500 websites globally[8] [9]. The malicious scripts exhibited advanced evasion and persistence mechanisms: they were embedded within heavily obfuscated JavaScript code, performed runtime assessment of device computational resources to adaptively regulate mining intensity, instantiated multiple parallel Web Workers to offload mining tasks without noticeably impairing the main browser thread, and funneled all communication through WebSocket channels to render the associated network traffic indistinguishable from ordinary browser activity. Therefore, browser-based cryptojacking remains a significant concern for users, as it can occur when they access compromised or malicious websites [10], [11], connect to public Wi-Fi networks [12], or interact with online advertising networks [13].

To mitigate cryptojacking, researchers have proposed a variety of effective detection methods. Common approaches to detecting cryptojacking include maintaining blocklists [14], [15], [16], [17] and analyzing CPU usage patterns [18], [19]. In addition, researchers have proposed detection methods based on code feature matching techniques. These approaches can be broadly categorized into static and dynamic methods. Static methods typically utilize code-level features for matching, such as specific JavaScript (JS) function signatures [20], [21] and the distribution of WebAssembly instructions [22], [23], [24], [25], [26], [27], [28]. Dynamic methods are primarily based on the runtime characteristics of web applications, including hardware performance metrics [29], [30], [31], [32], [33], [34], [35], network traffic patterns [36], [37], [38], [39], [40], [41], [42], [43], opcode sequences [44], [45], and trace events [46], [47]. The primary differences between static and dynamic methods lie in their data sources, feature representations, and classification algorithms.

Although many published studies report accuracies exceeding 95%, these methods remain prone to false positives, particularly by misclassifying benign websites as malicious [48], [23]. Static detection methods tend to extract more robust

Manuscript received August 31, 2026; revised November 4, 2026. Corresponding author: Zhanyong Tang (email: zytang@nwu.edu.cn).

features [24], [28] and identify browser-based mining behavior through rule generation. However, many commonly used operational features may exhibit characteristics similar to malicious mining behavior, such as specific keywords [14] and WebAssembly features [28]. A typical example is that some mining websites have ceased operation [21]. If the detection model relies solely on access patterns for decision-making, it may also generate false positives. Dynamic detection methods are currently more widely adopted; however, their results remain unreliable. This is primarily because the data contain noise introduced by external programs or applications [23], including CPU utilization, memory usage, and network-related features [49], [34].

This paper presents MINERLAMP, an improved approach for detecting in-browser cryptojacking during website visits at runtime. The core of our approach is a graph neural network (GNN) that identifies potentially abnormal cryptojacking behaviors based on runtime events collected from modern web browsers. Our GNN takes as input a multi-relational graph constructed from dynamic traces and associated event attributes. It then predicts the likelihood that a website is engaged in cryptojacking activities. Our GNN-based Graph Isomorphism Network (GIN) [50] is specifically designed to model the complex relationships captured from web browser events. Compared to previous approaches, nodes in the graph more accurately represent event properties associated with mining, while edges capture the dynamic behaviors of in-browser cryptojacking, such as the occurrence of events and function calls. The distinguishing design choice is the event-graph representation: function nodes retain behavioral attributes, while edges encode execution order and associations with resource, communication, and scheduling events. GIN aggregates information over this constructed topology to classify website traces. This design allows us to evaluate whether event connectivity provides discriminative information beyond node-level behavioral attributes.

We have implemented a functional prototype of MINERLAMP[1] . We evaluate MINERLAMP by applying it to both benign and malicious web samples. We compare MINERLAMP with eleven representative state-of-the-art approaches [16], [14], [20], [25], [28], [46], [23], [34], [27], [29], [51]. Experimental results demonstrate that MINERLAMP can identify cryptojacking behavior in samples within one minute while maintaining a low false positive rate and achieving high detection accuracy.

In summary, we make the following contributions:

- We systematically analyze real-world mining services and distill a common seven-step workflow of in-browser cryptojacking. We observe that this behavioral workflow, rather than isolated features, constitutes the intrinsic and evasion-resistant core of mining activities.
- We propose an event-graph representation that combines function-execution order with associations to resource requests, WebSocket creation, and scheduling events. The representation supports joint analysis of event connectivity and node-level behavioral attributes for in-browser cryptojacking detection.
- We build a functional prototype and release a labeled dataset of 21,957 websites. Across extensive evaluations against eleven state-of-the-art methods, our approach detects cryptojacking within one minute, achieves higher accuracy, and reduces false positives by at least 39.69%. Ablation studies show that jointly modeling node-level features and edge structures is crucial for minimizing false alarms.

[1]Code and data are publicly available at https://github.com/DeShawn-Phang/MinerLamp

## II. BACKGROUND AND MOTIVATION

Cryptocurrency is a form of digital asset that relies on cryptographic methods and blockchain technology to provide security and protect privacy. Blockchain, the core technology behind cryptocurrencies, functions as a decentralized ledger that logs all digital transactions and is maintained by miners. To add new blocks to the blockchain, miners must solve a Proof-of-Work (PoW) puzzle, which requires repeatedly executing hash-based computations, a process known as cryptomining. The miner who first solves the PoW puzzle is granted a fixed amount of cryptocurrency as a reward. In order to earn more cryptocurrency, miners compete by using increasingly sophisticated hardware, such as application-specific integrated circuits (ASICs). While ASIC-based mining greatly boosts the network's overall computational power, the difficulty of the PoW puzzle also increases since it is automatically adjusted according to the total computing capacity of the network. Consequently, miners who cannot bear the substantial purchase and maintenance costs of such mining equipment are eventually pushed out of the market because their operations are no longer profitable.

### A. In-Browser Cryptomining

To address the issue that ASIC mining undermines the decentralized nature of cryptocurrencies, an ASIC-resistant mining algorithm, CryptoNight, has been designed and adopted by several alternative cryptocurrencies, such as Monero [52] and Sumokoin [53]. This algorithm is memory-hard, enabling CPUs to fully utilize their cache hierarchies for efficient mining, while other hardware platforms cannot achieve similar efficiency. With the increasing market value of these alternative cryptocurrencies, CPU-based mining has become profitable again.

In addition, mining pools provide greater opportunities for miners with limited computational resources to earn rewards. A mining pool aggregates multiple miners and distributes the overall mining task into smaller sub-tasks, thereby increasing the likelihood of successfully adding new blocks and sharing the corresponding rewards. Mining pools typically employ the Stratum protocol for authentication and WebSocket for communication. Miners are required to register accounts with the mining pool to receive rewards proportional to their contributions.

The development of modern web technologies has significantly improved code execution performance in web browsers. For example, WebAssembly is a binary instruction format that can be executed in web browsers at near-native speed. Due to

the ease of compiling C/C++ code into Wasm, it was rapidly adopted in mining programs [25], [54]. Subsequently, with the rising value of alternative cryptocurrencies, in-browser cryptomining became increasingly prevalent. Many services provided convenient APIs for embedding mining scripts into websites, such as CoinHive [55].

However, with the growth of the in-browser cryptomining market, some cybercriminals have begun to exploit public mining services to launch attacks known as cryptojacking. Such malicious code is easy to deploy and has been observed across a wide range of platforms [56], [57]. It exploits the computing resources of unsuspecting users to perform mining operations, thereby generating illicit profits for attackers [58]. Such attacks can target not only web browsers but also mobile devices [59] and network routers [60].

To evade standard detection methods, existing mining applications widely employ various evasion techniques to enhance their stealthiness. For example, CryptoLoot [61] employs code obfuscation to conceal critical code segments, while CoinIMP [62] utilizes CPU throttling to limit resource consumption.

### B. Evasion Techniques

In-browser cryptojacking differs from traditional forms of malware. It does not directly cause system failure or disruption; instead, it covertly consumes computational resources and electrical energy. Therefore, it is more difficult to detect. In addition, mining scripts often employ various evasion techniques to remain undetected.

#### 1) Domain Fluxing

Attackers may adopt domain fluxing to frequently change the domains associated with malicious websites. A common approach involves the use of a domain generation algorithm (DGA), which produces a set of pseudo-random domains, from which attackers periodically select targets to launch cryptojacking activities and evade blacklist-based detection mechanisms.

#### 2) Code Obfuscation

Code obfuscation is employed to conceal critical information and prevent reverse engineering. Common code obfuscation techniques include global variable and function name obfuscation, control flow flattening, virtualization, numeric expression transformation, literal encoding, and dead code injection. String obfuscation can modify variable and function names in the code, rendering detection methods based on string matching ineffective. For example, attackers may encode strings using ASCII or Unicode representations, or encrypt the entire text using cryptographic algorithms, thereby preventing detection tools from directly recovering the original variable or function names. Control flow obfuscation can alter the structural organization of a program's control flow. For example, without modifying the program's functionality, an `if-else` statement can be transformed into a `switch-case` construct, thereby altering the static control flow characteristics of the code and reducing the effectiveness of corresponding detection methods.

#### 3) CPU Throttling

CPU throttling is a commonly adopted technique by attackers to conceal mining scripts. In this way, mining websites can bypass monitoring systems that rely on high CPU utilization as a detection indicator. In addition, legitimate mining website operators also adopt CPU throttling as a default configuration to provide a better user experience. Tekiner et al. [48] identified 6,269 mining scripts, among which 1,384 samples set the throttling level to 90%. This finding indicates that CPU throttling is a prevalent practice in mining scripts. Cryptojacking that employs CPU throttling is more likely to persist in visitors' browsers for extended periods, thereby maximizing illicit gains.

### C. Motivation

We find that existing non-behavior-based methods exhibit significant limitations and can be easily bypassed by various evasion techniques. For example, blacklist-based approaches can be circumvented through domain fluxing, JavaScript function signature-based methods can be easily obfuscated, and approaches based on hardware performance statistics can be manipulated through CPU throttling. Moreover, attackers can combine multiple evasion techniques simultaneously to evade most detection methods. Behavior-based detection methods are generally more reliable, as they rely on the internal execution logic of cryptojacking rather than external features such as domains or function names.

Existing dynamic detectors represent runtime behavior using different abstractions. Behavior-feature-based methods summarize selected runtime observations into feature vectors [46], while temporal methods model their evolution over time [47]. Although these representations capture useful behavioral information, they do not explicitly represent the event-level associations among heterogeneous runtime activities considered in this work. In contrast, relational behavior modeling represents runtime events as nodes and encodes selected associations among them as graph edges. Thus, the key distinction lies in the input representation: MINERLAMP combines node-level behavioral attributes with event connectivity, capturing how computation, resource access, communication, and scheduling events are organized within an execution trace.

Motivated by this representational difference, we propose MINERLAMP to model cryptojacking behavior through both runtime attributes and event associations. This representation captures not only what runtime activities occur, but also how they are organized within an execution trace.

## III. METHOD ASSUMPTIONS AND SCOPE

### A. Analysis Scope and Objectives

This work addresses the problem of detecting in-browser cryptojacking through the lens of runtime software behavior analysis. The objective of our system, MINERLAMP, is to determine whether the execution trace of a web page contains behavioral patterns consistent with unauthorized cryptocurrency mining. We aim to achieve this by modeling runtime events as structured graphs and applying graph neural networks to learn discriminative behavioral signatures. Our focus is on detection

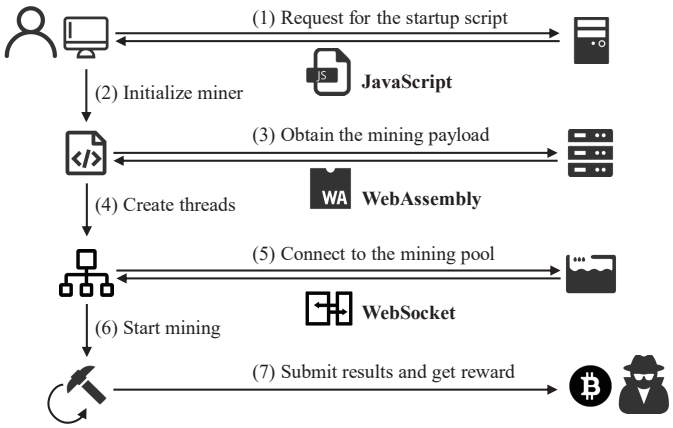



Fig. 1: Overview of the typical workflow of cryptojacking. We can catch the intrinsic relationship between different behaviors in the workflow and specific tasks for each behavior.

accuracy and reducing false positives, as misclassifying benign sites undermines user trust and practical utility. MINERLAMP is designed as an analyzer that reports suspicious behavior; it does not perform active blocking or mitigation.

### *B. Assumptions on the Analysis Environment*

Our methodology operates under the following assumptions regarding the execution context and data collection process:

- **Trace availability:** We assume access to a fine-grained, temporally ordered trace of browser execution events, including function calls, resource requests, thread operations, and WebSocket communications. These traces are collected through browser instrumentation (e.g., via the DevTools Protocol) in a controlled environment.
- **Observation window:** The analysis is conducted on a fixed-duration trace, typically covering the first 45 seconds of page load and execution. This duration is empirically sufficient for mining scripts to initialize and manifest their core behavior [46]. We do not assume continuous, indefinite monitoring.
- **Behavioral manifestation:** We assume that active cryptojacking, to be economically viable, must exhibit a set of observable interactions within the analysis window, including payload fetching, parallel computation, and external coordination. Scripts that remain entirely dormant or delay execution beyond the observation window are outside the scope of this analysis.

### *C. Characterization of the Target Behavior: A Workflow Model*

To derive the workflow model, we analyzed seven mining implementations, including CoinIMP, CryptoWebMiner, MoneroMiner, WebMinePool, WebMine, CryptoLoot, and Bmst. These implementations differ in their script implementations and runtime configurations, providing diverse realizations of in-browser mining behavior. We examined their runtime traces and compared the execution sequences and interactions involving script loading, resource requests, worker threads, task scheduling, and WebSocket communication. We then abstracted implementation-specific events into common behavioral stages according to their functional roles in the mining process. Despite implementation differences, these stages capture the common functional roles of active in-browser mining and serve as a generalized behavioral abstraction rather than a fixed execution sequence.

Figure 1 shows the resulting seven-stage workflow of typical in-browser cryptojacking. The workflow begins with the browser loading the host page and its startup script, which initializes the miner object with critical parameters including the beneficiary address and pool endpoint. The miner subsequently fetches the core mining payload, typically a WebAssembly module or heavily obfuscated JavaScript, from a remote server. To maximize throughput, the script spawns multiple worker threads for parallel computation while establishing a persistent WebSocket connection with the mining pool for task coordination. Each worker thread then enters a continuous loop of hash-like operations, with the execution rate regulated by timers to manage CPU consumption. Finally, the results are periodically collected from workers and submitted back to the pool through the established channel. This entire sequence, from resource acquisition to parallel execution and coordinated submission, forms a coherent behavioral chain rarely observed in benign web applications, rendering it a robust and detectable signature.

The key insight for our methodology is that it is not the presence of any individual step, but rather their *specific configuration and interconnection* that characterize malicious behavior. For instance, a benign video streaming site may also use WebSockets and worker threads, but its graph lacks the particular edge patterns connecting these events to a Wasm payload fetch and a sustained, timer-driven computational loop. This relational signature is the primary target of our graph-based learning approach.

### *D. Limitations and Boundary Conditions*

Our methodology has inherent limitations that define its scope of applicability:

- **Delayed execution:** Attacks that postpone all core mining activities beyond the 45-second observation window may evade detection. This represents a fundamental trade-off between detection latency and coverage.
- **Event observability:** The method relies on events exposed through the browser's instrumentation interface. Mining logic implemented using non-instrumented APIs or exploiting browser vulnerabilities beyond the scope of event tracing would not be detectable.
- **Generalization to novel workflows:** While our approach is designed to learn relational invariants, genuinely novel attack patterns that differ substantially from the training distribution may not be recognized. We address this limitation through empirical evaluation of generalization (Section V-F).

By explicitly stating these assumptions and boundaries, we provide a clear context for interpreting our empirical results

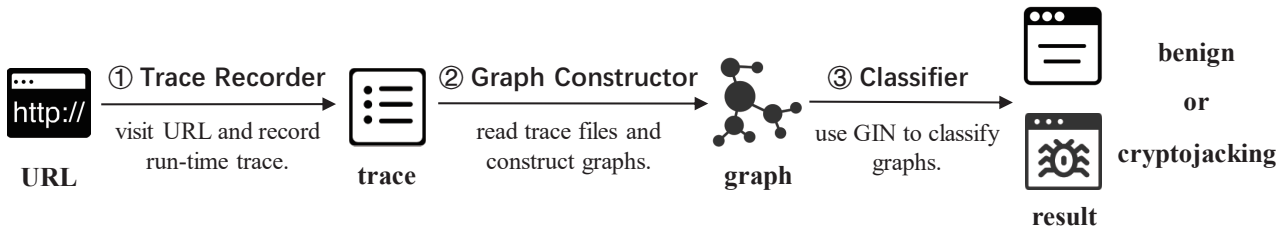



Fig. 2: Overview of MINERLAMP.

and for understanding the intended deployment scenarios of the proposed methodology.

## IV. OUR APPROACH

Inspired by the above analysis, the complex relationships among these mining behaviors can be effectively modeled using graph structures. Consequently, we propose a GNN-based approach for detecting in-browser cryptojacking. This approach aims to construct a multi-relational graph based on website runtime events and subsequently perform classification using a graph neural network. By capturing the behavioral patterns of in-browser mining, the proposed method can accurately identify cryptojacking websites while maintaining a low false-positive rate.

### A. Overview

As illustrated in Figure 2, the MINERLAMP framework comprises three main components. The first component is the trace recorder, which is responsible for visiting user-specified URLs and capturing the runtime execution traces of websites. The second component is the graph constructor, which processes the recorded traces, extracts relevant information, represents trace events as nodes, models relationships between events as edges, and constructs corresponding graphs. The third component is the classifier, which employs a GIN to classify the constructed graphs and identify cryptojacking websites. The remainder of this section details the implementation of each component and the model training process.

### B. Trace Recorder

To enable automated data collection, we require a component capable of receiving URL inputs and recording the corresponding execution traces as output. Specifically, we deployed the trace recorder in a Docker-based environment. This design ensures that system resources remain controllable by constraining the CPU utilization of the Docker container, thereby preventing abuse by malicious websites. The trace recorder first receives a URL as input. It then utilizes the advanced APIs provided by the Puppeteer [63] library to control Chromium via the DevTools Protocol and access the target website in headless mode.

After receiving a response, the trace recorder records the first 45 seconds of page loading and stores the runtime traces generated by the website. As shown by Kharraz et al. [46], 45 seconds is sufficient for a cryptojacking website to initiate its mining script, establish communication with the mining pool, and execute mining tasks. We use the default category parameters for trace recording, as they provide sufficient information for our analysis.

### C. Graph Constructor

Transforming the collected traces into a corresponding multi-relational graph is a critical step. Using a graph construction module, we accurately capture the semantics of the trace data. As shown in Figure 3a, we use a simplified trace as an illustrative example, retaining only events relevant to cryptojacking. Next, we elaborate on this trace and describe in detail the processes of node selection, edge definition, and graph construction. The details of the graph construction are provided in Algorithm 1.

#### 1) Nodes Selection

For graph classification tasks, node selection is a critical step. If events in the trace are not appropriately filtered, the resulting graph may become excessively large and noisy, and redundant or irrelevant information may negatively impact classification performance. Therefore, we conduct a detailed analysis of trace files generated by various mining services to extract semantically meaningful events and map them to corresponding node types. As shown in Table I, we summarize the types of nodes (and edges) involved in the constructed graph. The following principles guide the node selection process.

- Select events labeled "FunctionCall" to construct ❶`FunctionCall` nodes. A script comprises a sequence of function calls, and their execution forms the overall program workflow. A `FunctionCall` node is identified by its thread identifier ("tid") and function descriptor extracted from the trace arguments ("args"). Thus, calls to the same function in different threads are represented as different nodes. Repeated executions of the same function within the same thread are aggregated into a single node, with their execution frequency and cumulative CPU usage time recorded as node attributes.
- Select events labeled "ResourceSendRequest" to construct ❷`Resource` nodes. Based on the "URL" attribute, these nodes are further categorized into `Resource-wasm`, `Resource-blob`, and `Resource-js` nodes. These nodes capture the types of resources requested by the website and can help identify the use of specific resources (e.g., Wasm) as mining payloads.
- Select events labeled "WebSocketCreate" to construct ❸`WebSocket` nodes. Since most mining applications use




(a) The trace of the sample (only the necessary events related to cryptojacking are retained)

(b) The multilateral relationship graph of the sample

Fig. 3: A sample of graph construction. We give an example of a typical mining script graph construction. The eight nodes shown in different colors in the graph belong to four categories, and the four different line segments represent other edges.

**Algorithm 1:** Graph construction

**Input:** *trace*
**Output:** *graph*

*events* = read(*trace*);
*graph* = {"nodes":{}, "edges":{}};
*nodes* = []; /* Nodes selection */
**for** *event in events* **do**
  **if** *event["name"] == "FunctionCall"* **then**
    *nodes*.append(get_function_node(*event*));
  **end**
  **if** *event["name"] == "ResourceSendRequest"* **then**
    *nodes*.append(get_resource_node(*event*));
  **end**
  **if** *event["name"] == "WebSocketCreate"* **then**
    *nodes*.append(get_websocket_node(*event*));
  **end**
  **if** *event["name"] == "RunTask"* **then**
    *nodes*.append(get_task_node(*event*));
  **end**
**end**
*graph*["nodes"] = node_to_vector(*nodes*);
*edges* = [[],[],[],[]]; /* Edges generation */
**for** *event in events* **do**
  **if** *event["name"] == "thread_name"* **then**
    identify_tids(*event*);
  **end**
**end**
*edges*[0] = generate_main-thread_edges(*nodes*);
*edges*[1] = generate_sub-thread_edges(*nodes*);
*edges*[2] = generate_related-action_edges(*nodes*);
*edges*[3] = generate_related-task_edges(*nodes*);
*graph*["edges"] = *edges*;

TABLE I: Different types of nodes and edges in building a GIN network

| Node Type | Node Name | Edge Type |
|---|---|---|
| ❶ Function call nodes | ***FunctionCall*** | ① Main-thread edge |
| ❷Resource nodes | ***Resource-wasm*** ***Resource-blob*** ***Resource-js*** | ② Sub-thread edge |
| ❸WebSocket nodes | ***WebSocket*** | ③ Related-action edge |
| ❹Task nodes | ***Task-message2worker*** ***Task-message2global*** ***Task-timer*** | ④ Related-task edge |

WebSocket to communicate with mining pools, these nodes are informative for analyzing mining pool connections.

- Select events labeled "RunTask" to construct ❹`Task` nodes. Based on the "src_func" attribute, these nodes are further divided into three types: `Task-message2-worker`, `Task-message2global`, and `Task-timer`. These types represent message passing from the main thread to worker threads, message passing from worker threads to the main thread, and timer-based execution, which may correspond to task assignment, result submission, and execution frequency control in mining operations.
- Apply one-hot encoding to represent node types. For `FunctionCall` nodes, we incorporate two additional features: function execution frequency and cumulative CPU usage time. These features enable the identification of functions with high invocation frequency and sustained CPU consumption, which are indicative of mining behavior.

*2) Edge Rules*

A complete graph is formed by establishing edges between nodes. This process integrates isolated event features through well-defined relationships, enabling the graph to capture complete behavioral semantics. As summarized in Table I, we define four types of edges to model thread execution flows and event associations. The detail of edge construction is described in Algorithm 2. Specifically, after node extraction, the algorithm first identifies thread identifiers from events labeled "thread_name" and then generates different types of edges according to the corresponding construction rules.

- All `FunctionCall` nodes on the main thread are connected using ①`Main-thread` edges. This process corresponds to identifying the main thread through the "tid" associated with "CrRenderMain" and then invoking `generate_main-thread_edges(`*nodes*`)`. Based on this identifier, all `FunctionCall` nodes on the main thread are linked in chronological order. Repeated function calls form self-loops naturally in the graph. When consecutive `FunctionCall` events are merged into the same node during node construction, the corresponding temporal execution relationship is represented as a self-loop edge. Such self-loops preserve repeated execution patterns, which are important characteristics of mining behaviors, while avoiding unnecessary expansion of the graph size.
- All `FunctionCall` nodes on worker threads are connected using ②`Sub-thread` edges. The "tid" associated

with "DedicatedWorker thread" identifies worker threads. Based on this identifier, `FunctionCall` nodes on worker threads are connected sequentially in temporal order. The first `FunctionCall` node on a worker thread is linked to the temporally nearest `Function-Call` node on the main thread.

- `FunctionCall` nodes are connected with semantically related `Resource` and `WebSocket` nodes using ③`Related-action` edges. Due to the presence of anonymous functions, name-based matching may be unreliable. Therefore, we adopt a temporal proximity matching strategy to connect nodes based on their temporal closeness.
- `FunctionCall` nodes are connected with associated `Task` nodes using ④`Related-task` edges. We similarly employ temporal proximity matching to associate nodes. Since a function may participate in multiple tasks, redundant task associations are removed to reduce noise.

Overall, Algorithm 1 first extracts nodes from trace events, transforms them into feature vectors, and then generates four categories of edges to form the final multi-relational graph. Through this design, the graph preserves both the execution order of functions across threads and the semantic associations between computation, resource access, communication, and task scheduling, which are crucial for characterizing cryptojacking behavior.

The four relations define different connectivity rules during graph construction, thereby shaping the topology of the event graph. GIN does not explicitly distinguish edge types during aggregation, but operates on the resulting topology, where part of the relational semantics is implicitly reflected by the connected node types.

*3) Example*

Taking the above trace as an example, we construct a corresponding multi-relational graph, as illustrated in Figure 3b. Eight different colors are used to represent node types, while edges are distinguished by four different line styles. We designate `Node0` as the initial `FunctionCall` node. *Node1* and *Node2* represent JavaScript request operations, corresponding to Step 1 in the cryptojacking workflow. *Node3* represents the miner initialization function, corresponding to Step 2 in the workflow. *Node4*, *Node5*, and *Node6* represent payload downloading operations, corresponding to Step 3 in the workflow. A thread branch emerges after *Node7*, corresponding to Step 4. *Node7* and *Node8* represent a WebSocket connection, corresponding to Step 5. *Node9*, *Node10*, and *Node11* represent periodic function call behavior, corresponding to Step 6. *Node12*, *Node13*, and *Node14* represent periodic result submission behavior, corresponding to Step 7.

So far, we have modeled the seven-step cryptojacking workflow as a graph representation. In the subsequent classification task, our goal is to identify subgraphs that resemble the pattern shown in Figure 3b, which indicate the presence of suspicious cryptojacking behavior during website execution.

**Algorithm 2:** Edge construction

```
def generate_main-thread_edges(nodes):
    preNode = get_node("CrRenderMain");
    mainThreadPid = preNode.get_pid();
    for node in nodes do
        if node.get_Type() == "FunctionCall" then
            if node.get_pid() == mainThreadPid then
                Mainthread-connect(preNode, node);
                preNode = node;
            end
        end
    end
def generate_sub-thread_edges(nodes):
    subThreadPid = allEnentPid("DedicatedWorker");
    for node in nodes do
        if node.get_Type() == "FunctionCall" then
            if node.get_pid() in subThreadPid then
                preNode = getPreMainNode(node);
                Subthread-connect(preNode, node);
            end
        end
    end
def generate_related-action_edges(nodes):
    for node in nodes do
        if node.get_Type() == "Resource" or node.get_Type() ==
          "WebSocket" then
            pid = node.get_pid();
            preNode = getNearestFCallNode(node, pid);
            Relatedaction-connect(preNode, node);
        end
    end
def generate_related-task_edges(nodes):
    for node in nodes do
        if node.get_Type() == "Task" then
            pid = node.get_pid();
            preNode = getNearestFCallNode(node, pid);
            Relatedtask-connect(preNode, node);
        end
    end
```

### D. Classifier

With the constructed multi-relational graph structure, dependencies among website trace events can be effectively learned using graph-based machine learning algorithms. Prior work has applied various graph-based learning algorithms to similar classification tasks, including DeepWalk [64], graphlet-based methods [65], and GNNs [66]. Among these approaches, graph neural networks are particularly suitable for our task, as they can capture complex relationships among functions and other events, as well as nonlinear interactions arising from multithreaded execution. In this work, we employ GNNs to identify characteristic subgraphs corresponding to cryptojacking behaviors within the multi-relational graph. To the best of our knowledge, prior work has not leveraged graph neural networks for cryptojacking detection.

Numerous advanced graph neural network architectures exist, including Gated Graph Neural Networks (GGNN) [67], Graph Attention Networks (GAT) [68], Graph Convolutional Networks (GCN) [69], and GIN [50]. During model training, we employ the widely used Adaptive Moment Estimation (Adam) optimizer with a learning rate of 0.001. Adam dynamically adjusts parameter update steps by computing adaptive learning rates for individual parameters, thereby accommodating varying update dynamics across parameters. This adaptiveness is particularly beneficial for handling sparse gradient scenarios commonly encountered in graph neural networks. We evaluate these four GNN architectures on the multi-relational graph classification task using the training set,

as shown in Figure 4. Since our graph construction effectively captures critical information in malicious samples, mining-related features become readily learnable at the graph level. All four methods achieve strong performance on the training set, with accuracy exceeding 99%. We adopt GIN as our classifier due to its superior interpretability. Furthermore, GIN has been shown to possess representational power comparable to the Weisfeiler–Lehman graph isomorphism test [70].

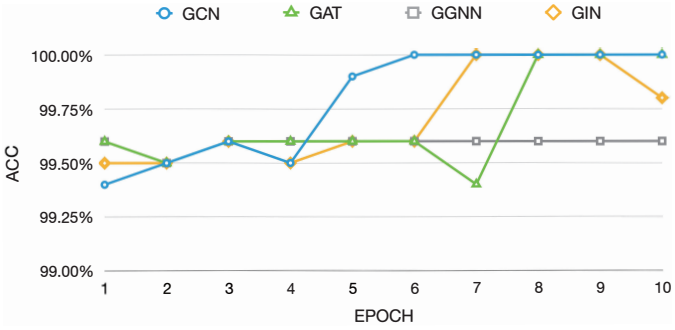



Fig. 4: The comparison of different GNNs

GIN updates node representations as shown in

$$h_v^{(k)} = MLP^{(k)}((1+\epsilon^{(k)}) \cdot h_v^{(k-1)} + \sum_{u \in N(v)} h_u^{(k-1)}) \quad (1)$$

where $h_v^{(k)}$ denotes the representation of node $v$ at the $k$-th iteration, and $N(v)$ denotes the set of neighbors of node $v$ across all four edge categories. $MLP$ denotes a multi-layer perceptron that approximates complex function compositions, and $\epsilon$ is either a learnable parameter or a fixed scalar. After multiple iterations, each node aggregates information from its neighborhood to learn a distinctive representation.

GIN concatenates graph-level representations across all iterations, as shown in

$$h_G = CONCA(ROUT(\{h_v^{(k)} | v \in G\}) | k = 0, 1, ..., K) \quad (2)$$

To obtain a graph-level representation, $ROUT$ aggregates node features at each layer, and $CONCA$ concatenates these features across layers to capture global graph information.

In this work, GIN is responsible for capturing graph structures indicative of cryptojacking in the multi-relational graph and generating an embedding for the entire graph. If the resulting embedding score exceeds 0.5, the graph is classified as containing cryptojacking-related structures, indicating the presence of malicious mining behavior on the website.

### *E. Model Training*

It is well known that neural networks typically require a large number of training samples. However, collecting a large number of cryptojacking samples in real-world settings is challenging. On the one hand, the proportion of cryptojacking websites is relatively low, making it difficult to obtain sufficient representative positive samples; on the other hand, websites are continuously evolving. Therefore, even if previous studies have compiled blacklists, their effectiveness cannot be fully guaranteed without further manual verification.

**Algorithm 3:** Data collection

**Input:** *url*
**Output:** *trace*

```
miners = ["coinIMP", "cryptowebminer", "monerominer",
  "webminepool", "webmine", "cryptoloot", "bmst"]; /* Scripts
  for mining to be selected */
minerScript = random_select(miners);          /* Randomly
  select a mining script */
threads = random_int(1, 4);                   /* Thread count */
throttle = random_float(0, 0.9);              /* CPU usage */
browser = puppeteer.launch();                 /* Creating a browser
  with the puppeteer */
page = browser.newPage();
page.tracing.start(trace);                    /* Start record tracing */
page.goto(url);
page.addScriptTag(minerScript, threads, throttle);
  /* Injecting mining scripts */
page.waitFor(45000);
page.tracing.stop();                          /* Records completed */
browser.close();
```

To address this issue, we adopt the approach proposed by Petrov et al. [47]. By collecting mining scripts and injecting them into the browser, we can simulate a large number of cryptojacking websites, as described in Algorithm 3. Specifically, we collected seven distinct mining scripts (denoted as *miners* in Algorithm 3); some of these scripts allow manual configuration of parameters such as CPU throttling and the number of threads. We randomly assign these optional parameters to increase sample diversity. The number of threads is set to $threads \in \{1, 2, 3, 4\}$, and the throttling parameter is set to $throttle \in \{0.1, 0.2, \ldots, 1.0\}$. Among the seven mining scripts, **`webmine`** and **`bmst`** do not support parameter configuration; **`coinIMP`**, **`monerominer`**, **`webminepool`**, and **`cryptoloot`** support both throttling and thread configuration, each generating up to 40 variants; **`cryptowebminer`** supports only throttling configuration, yielding 10 variants.

We randomly selected 10,000 websites from the top 20,000 entries of the Alexa Top-1M dataset as our URL list and visited each URL twice. During the first visit, we collected execution traces of the websites and treated these traces as benign samples. During the second visit, mining scripts were injected into the browser to generate additional malicious training samples containing cryptojacking-related behaviors. This procedure enables efficient augmentation of the training dataset while preserving the original characteristics of the visited websites.

Script injection may introduce script-specific artifacts that simplify classification. To mitigate this risk, we use seven mining implementations and vary runtime parameters, including the number of worker threads and CPU throttling levels, to increase behavioral diversity. Moreover, MINERLAMP uses runtime behavioral attributes and event associations rather than static script signatures, reducing its direct dependence on specific script contents.

The generated samples are used exclusively for model training. We split the generated training dataset into training and validation sets with a ratio of 8:2 and train the model for 50 epochs. The evaluation datasets consist of real-world websites collected from existing sources, including CoinBlockerLists and Tekiner et al., rather than injected samples. The training

and evaluation datasets are disjoint, with no overlap.

## V. Evaluation

We conduct an extensive evaluation of our graph-based behavior analysis methodology, instantiated in MinerLamp, to address the following key research questions (RQs):

- **RQ1 (Performance):** How does MinerLamp compare to state-of-the-art methods in detecting in-browser cryptojacking, particularly in terms of accuracy and robustness to false positives?
- **RQ2 (Efficiency):** What is the runtime overhead of MinerLamp, and how does it balance detection performance with practical efficiency?
- **RQ3 (Methodological Analysis):** How do different design choices in our graph construction (nodes, edges, features) impact the detection effectiveness?
- **RQ4 (Robustness):** How resilient is MinerLamp against common evasion techniques such as code obfuscation and delayed execution?

The evaluation is designed to validate not only the effectiveness of our specific system but also the merits of the underlying graph-based behavior modeling paradigm.

### A. Evaluation Method

#### 1) Environment Setup

Our experiments are conducted on a 13-inch MacBook Pro (2015 model). The system runs macOS 10.15.7, with Python 3.7 and Docker 2.5 installed.

#### 2) Test Dataset

As shown in Table II, we categorize all websites into three classes: i) *Running*: Websites that automatically execute mining scripts without user consent are considered cryptojacking websites. ii) *Benign*: Websites that do not contain any mining scripts. iii) *Not-Running*: Websites that contain mining scripts that are inactive due to specific reasons, such as the associated mining service being discontinued. These websites are ambiguous in nature and are therefore excluded from the test set in comparative experiments. These cases are further discussed in Section VI-A.

TABLE II: Test Dataset Breakdown

| Source | Dataset | Running | Benign | Not-Running |
|---|---|---|---|---|
| Dataset 1 | CoinBlockerLists [71] | 8 | 1181 | 121 |
| Dataset 2 | Tekiner et al. [48] | 245 | 3328 | 1175 |
| Dataset 3 | Alexa-Top-1M [72] | 5 | 15894 | 0 |

Our dataset consists of three subsets. The evaluation datasets consist of naturally occurring cryptojacking websites collected from existing sources rather than injected samples.

- **Dataset 1** is derived from the blacklist provided by CoinBlockerLists [71]. After manual inspection, we identified eight cryptojacking websites and 1,181 benign websites.
- **Dataset 2** is sourced from the study by Tekiner et al. [48], where keyword-based searching is used to collect suspicious URLs from PublicWWW [73]. After manual inspection, we identified 245 cryptojacking websites and 3,328 benign websites.
- **Dataset 3** is derived from the Alexa Top 1M list [72]. We collected 15,899 samples from the 30,000–50,000 ranking range and identified five cryptojacking websites and 15,894 benign websites.

In summary, the dataset contains 21,957 samples, including 258 cryptojacking websites, 20,403 benign websites, and 1,296 websites containing inactive mining scripts.

#### 3) Evaluation Metrics

We consider the following "*higher-is-better*" evaluation metrics for detecting mining samples:

**Accuracy** is defined as the ratio of correctly predicted samples to the total number of test samples.

**Precision** is defined as the ratio of correctly predicted positive samples to the total number of samples predicted as positive. This metric answers the question: "*Of all detected mining samples, how many are correct?*" High precision indicates a low *false-positive* rate, meaning that MinerLamp rarely misclassifies benign samples as mining.

**Recall** is defined as the ratio of correctly predicted positive samples to the total number of actual positive samples. This metric answers the question: "*Of all mining samples, how many are detected by* MinerLamp*?*" High recall suggests a low *false-negative* rate, indicating that MinerLamp can identify most mining samples that might otherwise be missed.

The **F1-score** is the harmonic mean of Precision and Recall, calculated as $2 \times \frac{P \times R}{P+R}$. A high F1-score indicates that MinerLamp achieves a good balance between detecting mining samples and minimizing misclassification of benign samples.

### B. Competitive Approaches

In terms of performance and overhead, we compare our method (MinerLamp) with existing approaches, including NoCoin [16], MinerBlock [14], CMTracker [20], MinerSweeper [25], MinerRay [28], Outguard [46], Capjack [29], CJ-Sniffer [51], Coinspy [34], MINOS [23], and Seismic [27], as shown in Table III.

TABLE III: Description of competitive methods

| Name | Description |
|---|---|
| NoCoin | monitor network traffic and requests |
| MinerBlock | monitor network requests and scripts |
| CJ-Sniffer | analyze mining behavior through network traffic |
| CMTracker | analyze hash functions and stack |
| MinerSweeper | match the primitives of Wasm instructions |
| MinerRay | analyze the semantics of code |
| MINOS | analyze the binary image of Wasm |
| Seismic | analyze the semantics of Wasm |
| Outguard | analyze mining behavior through trace |
| Capjack | analyze processor, memory, network and disk |
| Coinspy | analyze CPU, memory and network |

**NoCoin** and **MinerBlock** are browser extensions that detect cryptojacking using blacklist-based approaches, matching URL requests against known malicious domains and blocking suspicious connections.

**CJ-Sniffer** employs a two-sample Kolmogorov–Smirnov test [74] to detect cryptojacking by comparing packet inter-arrival time distributions with known malicious traffic patterns.

**CMTracker** employs behavior-based analysis, utilizing both hash-based and stack-based analyzers to detect cryptojacking based on cumulative hash execution time and repeated call chain patterns, respectively.

**MinerSweeper** focuses on Wasm instruction analysis, identifying cryptojacking by matching functions to cryptographic primitives using statistical heuristics. **MinerRay** enables cross-language detection by converting code into standardized Wasm representations, abstracting semantics into intermediate representations, and analyzing control flow graphs for hash operations within loops. **MINOS** leverages deep learning by transforming Wasm binaries into grayscale images and applying convolutional neural networks (CNNs) for classification. **SEISMIC** performs semantic matching by modifying Wasm binaries and extracting instruction execution frequencies, which are then classified using a Support Vector Machine (SVM).

**OutGuard** extracts runtime features related to mining behavior and classifies them using an SVM. **CapJack** utilizes system-level features such as processor, memory, network, and disk usage, feeding them into Capsule Networks (CapsNet) for detection. **CoinSpy** integrates browser behavior analysis by collecting JS execution stacks, network traffic, and heap memory usage, and employs deep CNN models for classification.

For the machine learning and deep learning methods mentioned above, we retrain their models using our training data combined with the datasets provided by the original authors. For model optimization, we employ the Neural Network Intelligence (NNI) framework [75] for hyperparameter tuning.

### C. RQ1: Detection Performance Comparison

As shown in Figure 5, our method achieves the best overall performance among all compared methods, reaching 99.91% accuracy, 92.81% precision, 100% recall, and an F1-score of 96.27% in this experimental setting. Compared with existing approaches, MINERLAMP demonstrates a more balanced detection capability in our evaluation setting, achieving 100% recall while substantially reducing false positives. Unlike approaches that rely on vulnerable static features (e.g., raw Wasm binaries), which can be easily evaded through techniques such as control flow flattening [76], our method does not depend on such representations. Instead, our method leverages dynamic runtime features and explicitly models their relationships, enabling more robust and accurate classification.

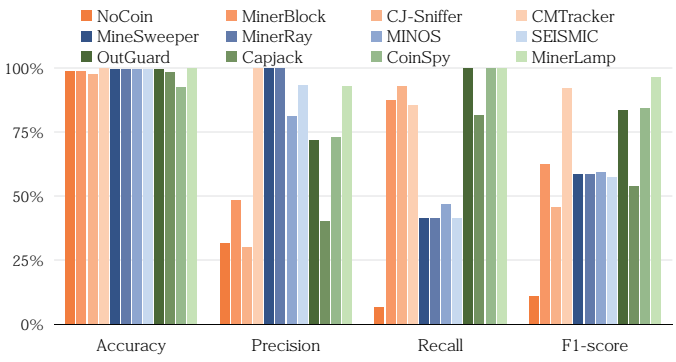



Fig. 5: The evaluation result of different detection methods

NoCoin and MinerBlock, two blacklist-based methods, exhibit limited performance on this dataset. They achieve lower precision than most competing methods and exhibit particularly poor recall. In particular, NoCoin achieves a recall of only 6.61%, indicating that many cryptojacking websites can evade blacklist-based methods due to their infrequent updates.

CJ-Sniffer relies on network traffic analysis for cryptojacking detection. This method achieves a high recall of 92.64%, but its precision is relatively low at 30.06%. The high false positive rate arises because video streaming and live broadcast websites often exhibit traffic patterns similar to cryptojacking behavior.

MinerSweeper and MinerRay focus on mining algorithm detection, resulting in high precision. However, they struggle when mining websites adopt new algorithms. For example, MinerRay fails to detect websites deploying CoinIMP in this experiment. This is because CoinIMP changes its mining target from Monero to MintMe Coin, leading to differences in mining algorithms and Wasm structures.

Both MINOS and SEISMIC are designed for Wasm-based cryptojacking detection. MINOS focuses on static Wasm features, whereas SEISMIC leverages dynamic instruction distributions. Experimental results indicate that SEISMIC achieves higher precision than MINOS. However, instrumentation overhead required for extracting runtime instruction features may disrupt normal website functionality, leading to reduced recall.

CMTracker, OutGuard, and CoinSpy outperform most baseline methods, except for MINERLAMP. CMTracker relies on a hash-based analyzer and specific function signatures, achieving 100% precision due to conservative threshold settings. However, it struggles to detect obfuscated mining scripts, resulting in a recall of only 85.21%. OutGuard achieves perfect recall by leveraging dynamic website features. CoinSpy produces 11 false negatives, primarily due to the absence of observable mining activity within the first 10 seconds of website loading. However, these features lack sufficient discriminative power, as similar patterns may also appear in real-time video or online gaming websites, resulting in a high false positive rate. A similar issue is observed in CapJack, which, despite high accuracy and recall, suffers from low precision due to false positives. CoinSpy further enhances robustness by incorporating Fourier transform-based frequency-domain features, achieving a 1.1% improvement in precision compared to OutGuard.

Figure 6 presents the ROC curves for different cryptojacking detection methods evaluated in this study. The ROC curve plots the true positive rate (TPR) against the false positive rate (FPR) across varying classification thresholds, where a positive sample corresponds to a cryptojacking website. Lowering the classification threshold increases the likelihood of labeling more samples as cryptojacking, thereby increasing both true positives and false positives.

The area under the ROC curve (AUC) for MINERLAMP reaches 0.99, indicating near-perfect separability between benign and mining samples in the evaluation dataset. This result demonstrates that MINERLAMP achieves exceptionally high detection capability while introducing only a negligible number of classification errors. Compared with existing approaches, the proposed method provides a more effective balance between detection accuracy and false positive control,

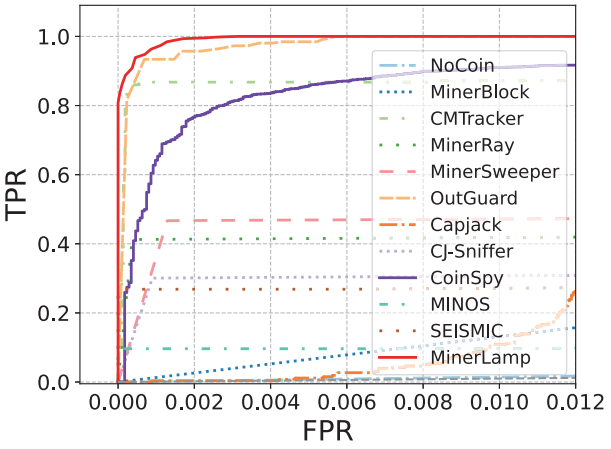



Fig. 6: ROC curves of different detection methods.

further confirming the effectiveness of leveraging dynamic runtime features and their interrelationships for cryptojacking detection.

### D. RQ2: Efficiency and Overhead

We also compare the actual execution time of each method, as shown in Table IV. The total execution time consists of preparation time and inference time. For blacklist-based methods, the preparation time primarily corresponds to website response latency, averaging approximately 5.5 seconds. For Wasm-based methods, the average preparation time is 18.8 seconds, mainly consisting of website response time and Wasm binary download latency. For trace-based methods, the average preparation time is 50.5 seconds, primarily due to website response time and the duration required to collect runtime execution traces.

TABLE IV: Overhead of different methods

| Method | Preparation (s) | Test (s) | Overall Time (s) |
|---|---|---|---|
| NoCoin | 5.5 | 0.2 | 5.7 |
| MinerBlock | 5.5 | 6.8 | 12.3 |
| CJ-Sniffer | 50.5 | 0.1 | 50.6 |
| CMTracker | 50.5 | 1582.8 | 1627.8 |
| MineSweeper | 18.8 | 163.6 | 182.4 |
| MinerRay | 18.8 | 167.1 | 185.9 |
| MINOS | 19.3 | 0.5 | 19.8 |
| Seismic | 520.2 | 2.1 | 522.3 |
| OutGuard | 50.5 | 5.2 | 55.7 |
| Capjack | 16.7 | 0.8 | 17.5 |
| Coinspy | 17.5 | 4.5 | 22 |
| MINERLAMP | 50.5 | 8.2 | 58.7 |

NoCoin and MinerBlock achieve the highest efficiency among all evaluated methods. Their efficiency stems from the simplicity of blacklist-based matching, which enables fast detection. However, these methods suffer from low precision and recall, often requiring extensive manual verification. Therefore, they are unsuitable for security-critical environments requiring high detection accuracy. In addition, the size of the blacklist significantly affects detection latency. Expanding the blacklist to improve detection coverage inevitably degrades efficiency, making such an approach impractical in practice.

CapJack, CoinSpy, and MINOS demonstrate relatively strong efficiency compared to other methods. Their efficiency improvements stem from the minimal requirement for collecting system-level features within the first 10 seconds of website interaction before being input into the detection model. For MINOS, only the time required for Wasm response and download needs to be considered. Despite their efficiency advantages, these methods exhibit lower accuracy and recall compared to MINERLAMP.

MinerSweeper, MinerRay, and Seismic suffer from significant computational overhead. MinerSweeper requires analyzing and counting all Wasm instructions; for example, when analyzing a web-based game (Tank), the analysis time exceeds one hour. MinerRay's average execution time exceeds three minutes, and when JavaScript is involved, additional time is required to convert it into standardized Wasm representations. The primary bottleneck of the Seismic method lies in its feature collection phase, requiring approximately 500 seconds to generate each profile instance.

CMTracker exhibits the longest testing time among all evaluated methods. This is primarily because it requires substantial time to read trace files and analyze complete stack relationships, resulting in increased computational overhead. On average, CMTracker requires approximately 27 minutes to process a single sample. For malicious samples, this delay implies that the user's device may remain compromised for nearly half an hour before detection results become available; moreover, the method may also generate false positives. The average running time of OutGuard is 55.2 seconds, of which 89.6% is attributed to data collection. During sample processing, OutGuard primarily counts the occurrence of target features or determines their presence. For classification, it relies on a trained SVM model, resulting in high computational efficiency.

The time overhead of CJ-Sniffer is comparable to that of OutGuard, with data collection accounting for 99.8% of the total runtime. During the detection phase, CJ-Sniffer evaluates the collected data using a two-sample Kolmogorov–Smirnov test, comparing the inter-arrival intervals of network packets with those characteristic of mining traffic.

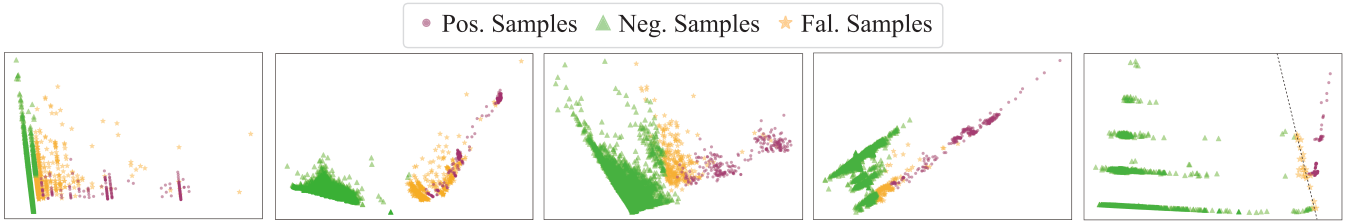
The efficiency of our method is comparable to that of OutGuard. The average preparation and testing time of MINERLAMP is less than one minute. Due to additional data analysis and processing during graph construction, the total runtime of MINERLAMP is approximately three seconds longer than that of OutGuard. However, OutGuard is more prone to false positives, and handling such cases typically incurs additional manual verification costs. Compared with the time required for manual verification, the additional three-second overhead is negligible. Therefore, our method is more practical in security-critical environments.

### E. RQ3: Methodological Analysis - Selection of Different Edges and Nodes

This section investigates why graph neural networks are more effective for detecting malicious mining scripts. As shown in Table V, we conduct a series of experiments based

TABLE V: We choose different nodes, edges, and the corresponding node features to embed in the graph network and then conduct the experiments. Experiments 1, 2, 3, and 5 compare different nodes and edges, and experiments 4 and 5 compare whether they have node characteristics or not.

| Experiment | Function call nodes | Resource nodes | WebSocket nodes | Task nodes | Main-thread edge | Sub-thread edge | Related-action edge | Related-task edge | Node Characteristics |
|---|---|---|---|---|---|---|---|---|---|
| Experiment 1 | • | | | | • | | | | • |
| Experiment 2 | • | • | | | • | • | | | • |
| Experiment 3 | • | • | • | | • | • | • | | • |
| Experiment 4 | • | • | • | • | • | • | • | • | |
| Experiment 5 | • | • | • | • | • | • | • | • | • |




(a) Experiment 1 (b) Experiment 2 (c) Experiment 3 (d) Experiment 4 (e) Experiment 5

Fig. 7: Program embedding space is given by different experiments. Experiment 5 is more effective in mapping code samples into a space where a more distinctive boundary can be drawn to separate mining scripts and benign code samples. Compared with experiments 1–4, experiment 5 contains only a small number of false positive samples.

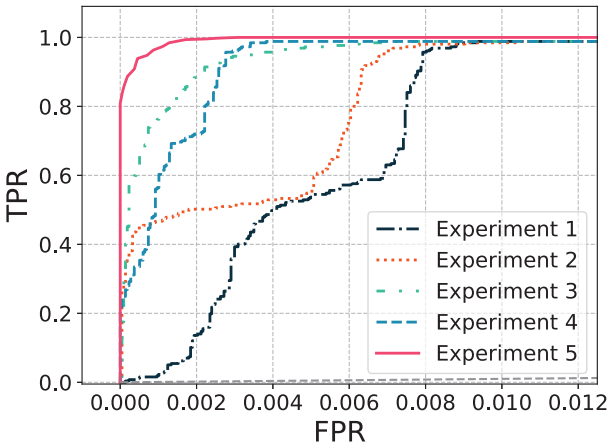



Fig. 8: ROC curves of five experiments.

on different combinations of node and edge configurations. Among the 21,957 samples, 258 correspond to actively running mining scripts. The experimental results indicate that the selection of nodes, edges, and critical features plays an important role in reducing false positive rates. In particular, Experiment 5 achieves the lowest false positive count, with only 20 false positives, whereas Experiment 1 produces 823 false positives.

The results further demonstrate that incorporating edge features can significantly reduce false positives. Specifically, the number of false positives decreases from 297 in Experiment 3 to 175 in Experiment 4. However, without incorporating node-level characteristics, Experiment 4 still introduces a small number of false positives.

Compared with Experiment 4, Experiment 5 incorporates node-level characteristics, including cumulative CPU usage time and execution frequency. Malicious mining samples typically involve the repeated execution of computationally intensive functions, leading to high CPU utilization and frequent function invocations. In contrast, the false positive samples observed in Experiment 4 generally exhibit lower CPU utilization and less intensive execution behavior. By modeling these node-level characteristics, Experiment 5 further reduces the number of false positives to only 20.

TABLE VI: Obfuscating sample counts and the relative distribution of programs in the dataset.

| Website Category | JS | Wasm | # Sample | Portion of Dataset |
|---|---|---|---|---|
| Games | 5 | 8 | 13 | 42.0% |
| Libraries | 0 | 1 | 1 | 3.2% |
| Applications | 5 | 3 | 8 | 25.8% |
| Cryptojackers | 5 | 4 | 9 | 29.0% |
| Total | 15 | 16 | 31 | 100% |

To facilitate visualization, we apply principal component analysis (PCA) [77], a widely used dimensionality reduction technique, to project the high-dimensional embedding space into two dimensions. Compared with the other methods, Experiment 5 (Figure 7e) maps the test samples into a feature space in which a clearer binary decision boundary separates mining and benign samples. Although a small number of false positives remain, Experiment 5 still exhibits a more compact and discriminative embedding distribution than the other experimental configurations. These results demonstrate that Experiment 5 is more effective in capturing essential program structures for cryptojacking detection.

Figure 8 presents the ROC curves for the five experimental configurations and different detection methods evaluated in this study. The ROC curve plots the true positive rate (TPR) against the false positive rate (FPR) under different node and edge configurations, where a positive sample corresponds to mining-related behavior.

### F. RQ4: Robustness Analysis

#### 1) Robustness Against Code-Level Obfuscation

To evaluate the robustness of MINERLAMP against obfuscation, we adopt *Tigress* [78] as the obfuscation tool for WebAssembly, following the methodology of Bhansali et al. [76]. To further enhance obfuscation, we additionally apply the *javascript-obfuscator* [79] tool to mining scripts.

We follow the dataset used in prior work [76] and augment it with additional mining website samples collected from publicly available sources. The dataset composition is summarized in Table VI.

The C source code of each web application in the dataset was individually processed using *Tigress* for obfuscation. The process is as follows: each C source file is input into the obfuscator, where a selected obfuscation technique is applied. The applied technique transforms the C code (as summarized in Table VII), producing a new obfuscated source file. Both the original and obfuscated source files are then compiled into Wasm modules using *Emscripten*, a widely used Wasm compiler. We then re-collect the website features and use MINERLAMP to perform detection. For the JavaScript source code, we apply three preset obfuscation configurations provided by the *javascript-obfuscator* tool and re-evaluate the obfuscated samples. This process is repeated for each program in the dataset.

Table VII shows that MINERLAMP successfully detects obfuscated cryptojacking samples across eight obfuscation techniques, while failures occur in two JavaScript obfuscation cases involving Crypto-Loot samples. Out of 205 obfuscated samples, MINERLAMP correctly identifies 203 samples. Further analysis reveals that the two failed samples do not exhibit actual mining behavior after obfuscation, as the obfuscated code fails to load the embedded Wasm miner. These results demonstrate the robustness of MINERLAMP against code obfuscation.

#### 2) Effect of Time Capture Window Size

In this section, we investigate the effect of deliberately delaying the initiation of mining scripts on the MINERLAMP platform. We conduct delay-based experiments on mining scripts corresponding to the 10 patterns detailed in Table VIII. Due to the discontinuation of mining services such as Bmst, CoinHive, WebMr, and JSECoin, we replace them with alternative mining scripts provided by CryptoWebMiner (specifically Haven-Protocol (XHV), Bytecoin, Turtlecoin, and Sumokoin). The start time of each mining script is incrementally delayed in 5-second intervals within the first 45 seconds of data collection. MINERLAMP is then used to re-evaluate these 90 samples and estimate the probability of evasion for each sample (i.e., the probability of the model predicting the sample as negative).

The experimental results are presented in Figure 9. The results indicate that MINERLAMP maintains a low evasion probability when the data collection window includes at least 10 seconds of mining-related behavior. Our dataset includes 258 mining sites. More than 95% of samples initiate mining activity within the first 10 seconds, while the remaining 5% do so within 15 seconds. Therefore, longer delays are uncommon in real-world mining scenarios.

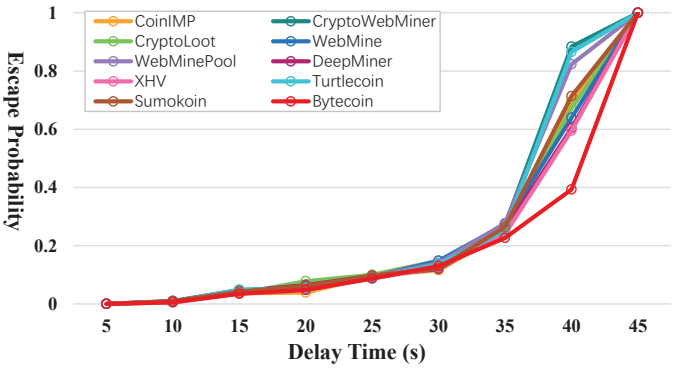



Fig. 9: Impact of delaying mining script startup on MINERLAMP predicted probabilities

Certain types of websites, such as those offering illegal streaming or adult content, may potentially interfere with the detection capabilities of MINERLAMP. This is primarily due to prolonged user engagement on such sites, which provides opportunities to delay the execution of mining scripts. However, this limitation can be mitigated by integrating MINERLAMP with browser plugins. Such plugins can monitor browser CPU usage and trigger MINERLAMP when significant anomalies are detected, thereby mitigating the impact of adversaries deliberately delaying mining scripts.

### G. Case Studies

In this section, we present two case studies illustrating how MINERLAMP improves detection accuracy and reduces the false positive rate in real-world environments.

***Case A: Detection accuracy analysis.*** MINERLAMP can identify previously unseen mining scripts based on the runtime behavior of websites. For example, our approach identified a previously unknown mining script on a news website in Mexico, ranked 35,202 in the Alexa Top 1M list. As shown in Figure 10a, the multi-relational graph constructed by MINERLAMP reveals that the website spawns a total of 16 threads for mining. These high-overhead mining threads impose a significant computational burden on the system. This behavior is characteristic of malicious mining and cannot be reliably detected using blacklist-based methods alone. In addition, due to the dependencies and sequential relationships among behaviors, methods that rely solely on detecting isolated behavioral patterns—such as MineSweeper and MinerRay—may fail to identify such malicious mining activity.

***Case B: False positive analysis.*** False positives represent one of the most significant challenges in real-world detection scenarios, as they require substantial manual verification effort. Our method effectively reduces false positives. As shown in Figure 10b, it illustrates the multi-relational graph of Twitch (Twitch.tv), a popular live-streaming platform that receives substantial daily traffic. We observe that this website may cause other detection methods (e.g., OutGuard) to generate false positives, while our method correctly classifies it as benign. This is because the website exhibits several features

TABLE VII: Results of MINERLAMP cryptojacking detector (✓: MINERLAMP detected sample as malware, ✗: MINERLAMP detected sample as benign), ✪: no code obfuscation and no predicted results. red: false positive, false negative and no results; green: true positive and true negative results, Unobf.: Unobfuscated, HOLP: High obfuscation Low performance, MOOP: Medium obfuscation Optimal performance, LOHP: Low obfuscation High performance)

| Website Category | name | JS or Wasm | Unobf. | Obfuscating C code | | | | | | | Obfuscating JS code | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Anti Alias Analysis | Anti Taint Analysis | Function Copy | Encode Literals | Flatten | Opaque Predicates | Virtualize | HOLP | MOOP | LOHP |
| Game | 2048 | Wasm | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| | Breakout | Wasm | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| | Circle Collide | Wasm | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| | Dave | Wasm | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| | Interplanetary | Wasm | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| | Asteroids | Wasm | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| | Snake | Wasm | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| | Tetris | Wasm | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| | 2048 | JS | ✗ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✗ | ✗ | ✗ |
| | Tetris | JS | ✗ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✗ | ✗ | ✗ |
| | Breakout | JS | ✗ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✗ | ✗ | ✗ |
| | Fruit Ninja | JS | ✗ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✗ | ✗ | ✗ |
| | Astray | JS | ✗ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✗ | ✗ | ✗ |
| Libraries | OpenGL | Wasm | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| Applications | BF Interpreter | Wasm | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| | Conway Life | Wasm | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| | Markdown Parser | Wasm | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| | BF Interpreter | JS | ✗ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✗ | ✗ | ✗ |
| | Conway Life | JS | ✗ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✗ | ✗ | ✗ |
| | Markdown Parser | JS | ✗ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✗ | ✗ | ✗ |
| | Calculator | JS | ✗ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✗ | ✗ | ✗ |
| | CyberChef | JS | ✗ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✗ | ✗ | ✗ |
| Cryptojackers | Garliccoin | Wasm | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | xmr-wasm | Wasm | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | deepMiner | Wasm | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | webminerpool | Wasm | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Crypto-Webminer | JS | ✓ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✓ | ✓ | ✓ |
| | WebMinePool | JS | ✓ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✓ | ✓ | ✓ |
| | Crypto-Loot | JS | ✓ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✗ | ✗ | ✓ |
| | Monero Webminer | JS | ✓ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✓ | ✓ | ✓ |
| | BrowserMine | JS | ✓ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✪ | ✓ | ✓ | ✓ |

TABLE VIII: Pattern list

| Miner | Patterns |
|---|---|
| Bmst | "*://bmst.pw/*", "*://*.bmst.pw/*" |
| CoinIMP | "*://*.hostingcloud.racing/*", "*://*.hostcontent.live/*" |
| CryptoWebMiner | "*://trustiseverything.de/*", "*://*.crypto-webminer.com/*", "*://*/*?perfekt=wss://*", "*://donttbeevils.de:*/*" |
| CryptoLoot | "*://statdynamic.com/*", "*://*/lib/crypta.js*", "*://*.reauthenticator.com/*" |
| WebMine | "*://webmine.cz/*", "*://*.webmine.pro/*" |
| WebMinePool | "*://webminepool.com/*", "*://webmp.gq:*/*" |
| DeepMiner | "*://*/deepMiner.*", "*://*deepminer/proxy*" |
| WebMr | "*://*/*webmr.js*", "*://*/*webxmr.js*" |
| CoinHive | "*://*coinhive.min.js", "*://coinhive.com/lib*", "*://*coinhive.com/proxy*" |
| JSECoin | "*://*.jsecoin.com/*" |

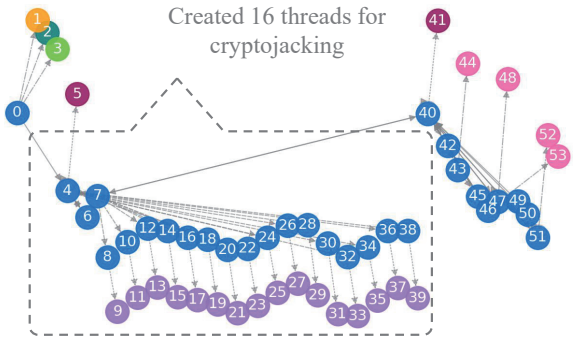
that resemble in-browser mining behavior, such as establishing WebSocket connections, retrieving Wasm files, and using timers. However, a deeper analysis reveals that its behavior is fundamentally different from that of mining websites. For example, although both types of websites establish WebSocket connections, their purposes differ significantly. Twitch uses WebSocket connections to retrieve Blob objects for video streaming, whereas mining websites use them to obtain mining payloads and communicate with mining pools.

As described in Section IV-C, in addition to modeling feature relationships, our method also analyzes function execution frequency and cumulative CPU usage time to determine whether functions repeatedly executed in worker threads correspond to high-overhead mining operations. In this case, the website is correctly classified as benign because it does not exhibit suspicious mining behaviors or high-overhead mining functions. In contrast, OutGuard relies on isolated dynamic and static features and fails to capture relationships among them, leading to false positives. This example illustrates why MINERLAMP is more effective at reducing false positives than OutGuard. False alerts on high-traffic websites are particularly undesirable, as they negatively impact user experience. Our method mitigates such issues and improves overall usability.

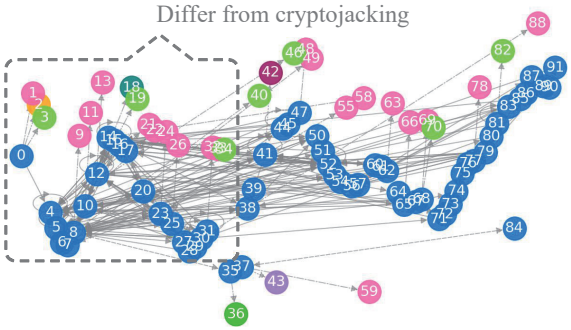
## VI. DISCUSSION

### A. Not-Running Scripts

We identified mining scripts via JS request and WebSocket patterns (Table VIII) and classified them as running or not running using keyword matching. Figure 11 shows that not-running scripts significantly outnumber running ones. CoinHive contributes the most not-running samples, while CoinImp has the most running ones. Scripts may be not run because the mining service is unavailable, the execution environment is unsupported, or user consent is required. MINERLAMP excludes not-running scripts since they exhibit no mining behavior. Our goal is to detect mining websites that perform




(a) The multilateral relationship graph of a news website




(b) The multilateral relationship graph of twitch.tv

Fig. 10: Two cases that may lead to false negatives or false positives. Figure (a) generates many threads with typical characteristics of the dynamic behavior of Cryptojacking; the left half of Figure (b) explains an entirely different feature of Cryptojacking.

unauthorized mining, regardless of whether known mining scripts are used.

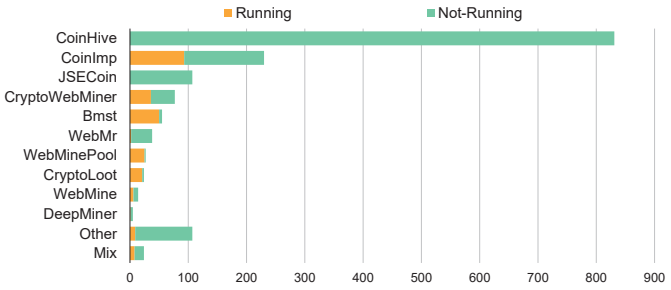



Fig. 11: The statistics of mining scripts. For some reason, many mining sites are not running, and we estimated the unrun scripts in these datasets.

### B. Threats to Validity

*Internal Validity:* Our synthetic data generation, while necessary and common practice in security research, may not capture all nuances of real-world adversarial evolution. However, our evaluation on entirely separate real-world datasets (Table II) mitigates this concern.

*External Validity:* The evaluation is based on current cryptojacking techniques. While our method is designed to be robust to behavioral invariants, fundamentally new attack workflows may require model retraining with new examples.

*Construct Validity:* Our graph construction uses heuristics (e.g., temporal proximity for linking events). More sophisticated program analysis could create more precise graphs, potentially improving performance further. However, our results show that even simple heuristics yield excellent performance.

## VII. RELATED WORK

To date, numerous methods have been proposed for cryptojacking detection. One line of research identifies cryptojacking based on hardware performance metrics. Elevated CPU utilization caused by a website often indicates potential cryptojacking activity. Accordingly, many detection methods monitor CPU behavior to identify malicious mining activity [48], [80]. In addition to CPU utilization, other hardware performance metrics can also be leveraged. For example, Ning et al. [29] use processor time, memory read/write speed, network packet transmission rates, and disk I/O as features to train a capsule network for website classification. Tahir et al. [30] utilize Hardware Performance Counters (HPCs) to profile browser behavior and employ a Random Forest classifier for detection. However, malicious mining applications may employ throttling to reduce resource consumption, thereby weakening hardware-based features and leading to increased false negatives. Moreover, such data are susceptible to noise from other applications, which may result in false positives in multi-tasking environments.

Another line of work detects cryptojacking by analyzing network traffic [51], [29], [34]. Since cryptocurrency mining generates substantial network traffic, many detection techniques rely on analyzing network traffic patterns to identify anomalies, such as irregular data transfer behavior, communication with known mining pools, or algorithm-specific traffic characteristics. For example, Feng et al. [51] compare probability distributions of benign and mining-related traffic to detect potential mining activities. However, relying solely on network traffic is not always reliable. Changes in encryption mechanisms may require additional decryption efforts or retraining of detection models.

Signature-based approaches often rely on function names or statistical code features to identify malicious behavior. Such methods have also been applied to cryptojacking detection. For instance, Saad et al. [21] leverage reconstructed control flow graphs (CFGs) to extract features such as cyclomatic complexity and lines of code, and apply fuzzy C-means clustering to identify suspicious websites. However, such methods are vulnerable to code obfuscation, which can alter both syntactic and statistical properties of the code.

Given that most mining applications now rely on WebAssembly for implementing core workloads, many studies focus on Wasm-based detection. For example, Bian et al. [24] use specific instruction patterns as fingerprints of mining behavior. Naseem et al. [23] transform Wasm binaries into grayscale images and apply convolutional neural networks (CNNs) for classification. However, these methods are susceptible to obfuscation and can be bypassed by alternative implementations such as asm.js. Furthermore, variations in mining algorithms (e.g., CryptoNight variants) may exhibit different instruction distributions, reducing detection effectiveness.

TABLE IX: Comparison of runtime representations in closely related dynamic detectors

| Aspect | OutGuard | CoinPolice | MINERLAMP |
|---|---|---|---|
| Input representation | Behavioral feature vector | Function-execution time series | Runtime event graph |
| Behavioral information | Selected runtime activities | Temporal execution patterns | Event attributes and connectivity |
| Event associations | Not explicitly retained | Temporal patterns in execution sequences | Execution order and inferred event associations |
| Classifier | SVM | Recurrent neural network | GIN |

Behavior-based detection methods aim to capture runtime characteristics of mining activities. For example, Petrov et al. [47] construct time-series models to capture CPU throttling behavior, assuming that only mining applications utilize throttling. Within this line of research, MINERLAMP explores a graph-based representation of runtime behavior, investigating whether representing selected associations among heterogeneous runtime events as graph connectivity can provide useful evidence for cryptojacking detection.

To clarify this representational distinction, Table IX compares MINERLAMP with two closely related dynamic detectors, OutGuard [46] and CoinPolice [47], in terms of their input representations, retained behavioral information, and event associations. OutGuard summarizes runtime behavior using selected features, while CoinPolice models temporal execution patterns. In contrast, MINERLAMP combines node-level behavioral attributes with selected event associations represented as graph connectivity, preserving how computation, resource access, communication, and scheduling activities are organized within an execution trace.

## VIII. CONCLUSION

This paper introduces a graph-based methodology for modeling runtime software behavior through Multilateral Relationship Graphs and Graph Neural Networks, instantiated in MINERLAMP for in-browser cryptojacking detection. MINERLAMP transforms execution traces into structured graphs that capture temporal and logical dependencies among events, enabling a Graph Isomorphism Network to learn discriminative behavioral patterns. Evaluated on 21,957 real-world website traces, MINERLAMP achieves 92.81% precision and 100% recall, outperforming eleven state-of-the-art methods while reducing false positives by 39.69%. Ablation studies confirm that both graph structure and relational edges are essential to this performance. The system resists code obfuscation, completes analysis in under one minute, and extends to system-level malware detection with promising results. This work introduces a generalizable paradigm for software behavior analysis, advances GNN-based runtime understanding, and releases a public dataset. By reframing security detection as behavior modeling, it provides a new technique for analyzing complex, dynamic software systems.

## REFERENCES


[1] E. Wustrow and B. VanderSloot, "DDoSCoin: Cryptocurrency with a malicious Proof-of-Work," in *10th USENIX Workshop on Offensive Technologies (WOOT 16)*. Austin, TX: USENIX Association, Aug. 2016. [Online]. Available: https://www.usenix.org/conference/woot16/workshop-program/presentation/wustrow I

[2] "Cryptonote v 2.0," https://bytecoin.org/old/whitepaper.pdf, 2013. I

[3] "Webassembly," https://webassembly.org/, 2022. I

[4] D. Tanana, "Performance analysis of common browser extensions for cryptojacking detection," in *2025 XIX International Symposium on Problems of Redundancy in Information and Control Systems (Redundancy)*. IEEE, 2025, pp. 1–4. I

[5] E. Tekiner, A. Acar, A. S. Uluagac, E. Kirda, and A. A. Selcuk, "In-browser cryptomining for good: An untold story," in *2021 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*. IEEE, 2021, pp. 20–29. I

[6] S. Varlioglu, B. Gonen, M. Ozer, and M. Bastug, "Is cryptojacking dead after coinhive shutdown?" in *2020 3rd International Conference on Information and Computer Technologies (ICICT)*. IEEE, 2020, pp. 385–389. I

[7] Z. Zhang, G. Hong, X. Li, Z. Fu, J. Zhang, M. Liu, C. Wang, J. Chen, B. Liu, H. Duan *et al.*, "Under the dark: A systematical study of stealthy mining pools (ab) use in the wild," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 326–340. I

[8] (2025, 7) Hackers infect over 3,500 websites with monero cryptojacker. Accessed: 2026-06-29. [Online]. Available: https://forklog.com/en/hackers-infect-over-3500-websites-with-monero-cryptojacker/ I

[9] (2025, 7) Security report: At least 3500 websites have been maliciously injected with hidden monero mining scripts. Accessed: 2026-06-29. [Online]. Available: https://www-tc.aicoin.com/en/news-flash/2484659 I

[10] "Uk and usa government websites hacked to mine monero," https://tinyurl.com/y9cm2gtp, 2020. I

[11] "Government websites hit by cryptocurrency mining malware." https://tinyurl.com/y9vt24xc, 2020. I

[12] "Catalincimpanu. cryptojackers found on starbucks wifi network,github,piratestreamingsites." https://www.bleepingcomputer.com/news/security/cryptojackers-found-on-starbucks-wif-network-github-pirate-streaming-sites/., 2017. I

[13] "Chaoying liu and joseph c.chen. google's doubleclick abused to deliver miners," https://blog.trendmicro.com/trendlabs-security-intelligence/malvertising-campaign-abuses-googles-doubleclick-to-deliver-cryptocurrency-miners/, 2018. I

[14] "Minerblock," https://github.com/xd4rker/MinerBlock, 2021. I, V-B

[15] "Dr. mine," https://github.com/1lastBr3ath/drmine, 2019. I

[16] "Nocoin," https://github.com/keraf/NoCoin, 2021. I, V-B

[17] M. A. Razali and S. Mohd Shariff, "Cmblock: In-browser detection and prevention cryptojacking tool using blacklist and behavior-based detection method," in *International Visual Informatics Conference*. Springer, 2019, pp. 404–414. I

[18] A. D. Yulianto, P. Sukarno, A. A. Warrdana, and M. Al Makky, "Mitigation of cryptojacking attacks using taint analysis," in *2019 4th International Conference on Information Technology, Information Systems and Electrical Engineering (ICITISEE)*. IEEE, 2019, pp. 234–238. I

[19] M. Musch, C. Wressnegger, M. Johns, and K. Rieck, "Thieves in the browser: Web-based cryptojacking in the wild," in *Proceedings of the 14th International Conference on Availability, Reliability and Security*, 2019, pp. 1–10. I

[20] G. Hong, Z. Yang, S. Yang, L. Zhang, Y. Nan, Z. Zhang, M. Yang, Y. Zhang, Z. Qian, and H. Duan, "How you get shot in the back: A systematical study about cryptojacking in the real world," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1701–1713. I, V-B

[21] M. Saad, A. Khormali, and A. Mohaisen, "End-to-end analysis of in-browser cryptojacking," *arXiv preprint arXiv:1809.02152*, 2018. I, VII

[22] T. Sermchaiwong and J. Shen, "Proof-of-theft: Dynamic graph-based fingerprinting of in-browser cryptomining," in *40th European Conference on Object-Oriented Programming (ECOOP 2026)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2026, pp. 25–1. I

[23] A. Ar, F. Naseem, L. Babun, E. Tekiner, and S. Uluagac, "Minos: A lightweight real-time cryptojacking detection system," in *The Network and Distributed System Security Symposium (NDSS) 2021*, 2021. I, V-B, VII

[24] W. Bian, W. Meng, and M. Zhang, "Minethrottle: Defending against wasm in-browser cryptojacking," in *Proceedings of The Web Conference 2020*, 2020, pp. 3112–3118. I, VII

[25] 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 *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1714–1730. I, II-A, V-B

[26] J. Rüth, T. Zimmermann, K. Wolsing, and O. Hohlfeld, "Digging into browser-based crypto mining," in *Proceedings of the Internet Measurement Conference 2018*, 2018, pp. 70–76. I

[27] W. Wang, B. Ferrell, X. Xu, K. W. Hamlen, and S. Hao, "Seismic: Secure in-lined script monitors for interrupting cryptojacks," in *European Symposium on Research in Computer Security*. Springer, 2018, pp. 122–142. I, V-B

[28] A. Romano, Y. Zheng, and W. Wang, "Minerray: semantics-aware analysis for ever-evolving cryptojacking detection," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 1129–1140. I, V-B

[29] R. Ning, C. Wang, C. Xin, J. Li, L. Zhu, and H. Wu, "Capjack: Capture in-browser crypto-jacking by deep capsule network through behavioral analysis," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 1873–1881. I, V-B, VII

[30] R. Tahir, S. Durrani, F. Ahmed, H. Saeed, F. Zaffar, and S. Ilyas, "The browsers strike back: Countering cryptojacking and parasitic miners on the web," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 703–711. I, VII

[31] A. Hussain, J. Flores, and P. Papadimitratos, "Deep learning-based cryptojacking detection in cloud containerized environments," in *2025 IEEE Middle East Conference on Communications and Networking (MECOM)*. IEEE, 2025, pp. 1–6. I

[32] A. Gangwal, S. G. Piazzetta, G. Lain, and M. Conti, "Detecting covert cryptomining using hpc," in *International Conference on Cryptology and Network Security*. Springer, 2020, pp. 344–364. I

[33] G. Mani, V. Pasumarti, B. Bhargava, F. T. Vora, J. MacDonald, J. King, and J. Kobes, "Decrypto pro: Deep learning based cryptomining malware detection using performance counters," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2020, pp. 109–118. I

[34] C. Kelton, A. Balasubramanian, R. Raghavendra, and M. Srivatsa, "Browser-based deep behavioral detection of web cryptomining with coinspy," in *Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb) 2020*, 2020, pp. 1–12. I, V-B, VII

[35] D. Tanana, "Behavior-based detection of cryptojacking malware," in *2020 Ural Symposium on Biomedical Engineering, Radioelectronics and Information Technology (USBEREIT)*. IEEE, 2020, pp. 0543–0545. I

[36] E. N. Mutombo, M. W. Nkongolo, and M. Tokmak, "Cryptojacking detection using model-agnostic explainability," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 41, no. 1, pp. 394–408, 2026. I

[37] E. Tekiner, A. Acar, and A. S. Uluagac, "A lightweight iot cryptojacking detection mechanism in heterogeneous smart home networks," in *The Network and Distributed System Security Symposium (NDSS) 2022*, 2022. I

[38] J. Rauchberger, S. Schrittwieser, T. Dam, R. Luh, D. Buhov, G. Pötzelsberger, and H. Kim, "The other side of the coin: A framework for detecting and analyzing web-based cryptocurrency mining campaigns," in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, 2018, pp. 1–10. I

[39] M. Ishii, S. Shibuya, and K. Tanaka, "Effective analysis of encrypted traffic for mining detection." in *ICISSP (1)*, 2026, pp. 61–72. I

[40] A. Ahmad, W. Shafiuddin, M. N. Kama, and M. M. Saudi, "A new cryptojacking malware classifier model based on dendritic cell algorithm," in *Proceedings of the 3rd International Conference on Vision, Image and Signal Processing*, 2019, pp. 1–5. I

[41] H. N. C. Neto, M. A. Lopez, N. C. Fernandes, and D. M. Mattos, "Minecap: super incremental learning for detecting and blocking cryptocurrency mining on software-defined networking," *Annals of Telecommunications*, vol. 75, no. 3, pp. 121–131, 2020. I

[42] J. Z. i Muñoz, J. Suárez-Varela, and P. Barlet-Ros, "Detecting cryptocurrency miners with netflow/ipfix network measurements," in *2019 IEEE International Symposium on Measurements & Networking (M&N)*. IEEE, 2019, pp. 1–6. I

[43] T. O. Adigun, A. Oshinubi, and I. Eweoya, "Cryptojacking detection using ml techniques." *IUP Journal of Computer Sciences*, vol. 19, no. 1, 2025. I

[44] H. Darabian, S. Homayounoot, A. Dehghantanha, S. Hashemi, H. Karimipour, R. M. Parizi, and K.-K. R. Choo, "Detecting cryptomining malware: a deep learning approach for static and dynamic analysis," *Journal of Grid Computing*, vol. 18, no. 2, pp. 293–303, 2020. I

[45] N. Lachtar, A. A. Elkhail, A. Bacha, and H. Malik, "A cross-stack approach towards defending against cryptojacking," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 126–129, 2020. I

[46] 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 *The World Wide Web Conference*, 2019, pp. 840–852. I, II-C, III-B, IV-B, V-B, VII

[47] I. Petrov, L. Invernizzi, and E. Bursztein, "Coinpolice: Detecting hidden cryptojacking attacks with neural networks," *arXiv preprint arXiv:2006.10861*, 2020. I, II-C, 12, VII

[48] E. Tekiner, A. Acar, A. S. Uluagac, E. Kirda, and A. A. Selcuk, "Sok: Cryptojacking malware," *arXiv preprint arXiv:2103.03851*, 2021. I, II-B3, II, V-A2, VII

[49] J. D. P. Rodriguez and J. Posegga, "Rapid: Resource and api-based detection against in-browser miners," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 313–326. I

[50] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018. I, IV-D

[51] Y. Feng, J. Li, and D. Sisodia, "Cj-sniffer: Measurement and content-agnostic detection of cryptojacking traffic," in *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, 2022, pp. 482–494. I, V-B, VII

[52] "Monero (xmr): Private digital currency," https://www.getmonero.org/, 2022. II-A

[53] "Sumokoin: Privacy without compromise," https://sumokoin.org/, 2022. II-A

[54] M. Musch, C. Wressnegger, M. Johns, and K. Rieck, "New kid on the web: A study on the prevalence of webassembly in the wild," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*. Springer, 2019, pp. 23–42. II-A

[55] "Coinhive snapshots that taken form webarchive," https://web.archive.org/web/20181101000000*/coinhive.com, 2022. II-A

[56] R. Holz, D. Perino, M. Varvello, J. Amann, A. Continella, N. Evans, I. Leontiadis, C. Natoli, and Q. Scheitle, "A retrospective analysis of user exposure to (illicit) cryptocurrency mining on the web," *arXiv preprint arXiv:2004.13239*, 2020. II-A

[57] P. H. Meland, B. H. Johansen, and G. Sindre, "An experimental analysis of cryptojacking attacks," in *Nordic Conference on Secure IT Systems*. Springer, 2019, pp. 155–170. II-A

[58] P. Papadopoulos, P. Ilia, and E. Markatos, "Truth in web mining: Measuring the profitability and the imposed overheads of cryptojacking," in *International Conference on Information Security*. Springer, 2019, pp. 277–296. II-A

[59] S. Dashevskyi, Y. Zhauniarovich, O. Gadyatskaya, A. Pilgun, and H. Ouhssain, "Dissecting android cryptocurrency miners," in *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, 2020, pp. 191–202. II-A

[60] H. L. Bijmans, T. M. Booij, and C. Doerr, "Just the tip of the iceberg: Internet-scale exploitation of routers for cryptojacking," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 449–464. II-A

[61] "Cryptoloot - earn more from your traffic," https://crypto-loot.org/, 2022. II-A

[62] "Free javascript mining - browser mining use our javascript web miner and earn money with your page traffic!" https://www.coinimp.com/, 2022. II-A

[63] "Puppeteer: Headless chrome node api," https://developers.google.com/web/tools/puppeteer, 2022. IV-B

[64] C. Tu, W. Zhang, Z. Liu, M. Sun *et al.*, "Max-margin deepwalk: Discriminative learning of network representation." in *IJCAI*, vol. 2016, 2016, pp. 3889–3895. IV-D

[65] T. Gao, W. Peng, D. Sisodia, T. K. Saha, F. Li, and M. Al Hasan, "Android malware detection via graphlet sampling," *IEEE Transactions on Mobile Computing*, vol. 18, no. 12, pp. 2754–2767, 2018. IV-D

[66] J. Yan, G. Yan, and D. Jin, "Classifying malware represented as control flow graphs using deep graph convolutional neural network," in *2019 49th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 2019, pp. 52–63. IV-D

[67] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," *arXiv preprint arXiv:1511.05493*, 2015. IV-D
[68] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017. IV-D
[69] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. v. d. Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *European semantic web conference*. Springer, 2018, pp. 593–607. IV-D
[70] A. Leman and B. Weisfeiler, "A reduction of a graph to a canonical form and an algebra arising during this reduction," *Nauchno-Technicheskaya Informatsiya*, vol. 2, no. 9, pp. 12–16, 1968. IV-D
[71] "Coinblockerlists," https://gitlab.com/ZeroDot1/CoinBlockerLists, 2021. II, V-A2
[72] "Alexa-top-1m," http://s3.amazonaws.com/alexa-static/top-1m.csv.zip, 2022. II, V-A2
[73] "Publicwww: Source code search engine," https://publicwww.com/, 2022. V-A2
[74] F. J. Massey Jr, "The kolmogorov-smirnov test for goodness of fit," *Journal of the American statistical Association*, vol. 46, no. 253, pp. 68–78, 1951. V-B
[75] Microsoft, "Neural Network Intelligence," 1 2021. [Online]. Available: https://github.com/microsoft/nni V-B
[76] S. Bhansali, A. Aris, A. Acar, H. Oz, and A. S. Uluagac, "A first look at code obfuscation for webassembly," in *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2022, pp. 140–145. V-C, V-F1, V-F1
[77] A. Tharwat, "Principal component analysis-a tutorial," *Inderscience enterprises*, 2009. V-E
[78] "The tigress obfuscator," https://tigress.wtf, 2023. V-F1
[79] "javascript-obfuscator," https://github.com/javascript-obfuscator/javascript-obfuscator, 2023. V-F1
[80] K. Jayasinghe and G. Poravi, "A survey of attack instances of cryptojacking targeting cloud infrastructure," in *Proceedings of the 2020 2nd Asia pacific information technology conference*, 2020, pp. 100–107. VII