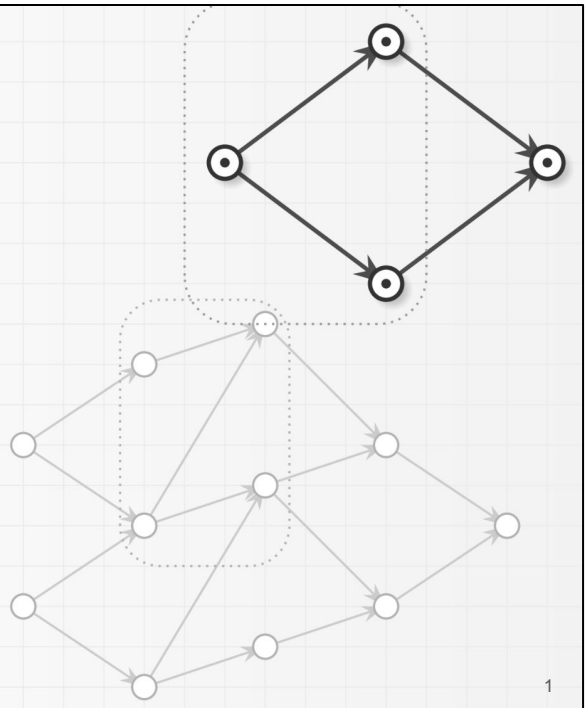
Proof-of-Theft: Dynamic Graph-Based Fingerprinting of In-Browser Cryptomining

Tanapoom Sermchaiwong (HKUST)
Jiasi Shen (HKUST)



ECOOP 2026

Cryptomining

- Cryptocurrencies rely on decentralized consensus
  - Proof-of-work (our focus), proof-of-stakes, etc.
- Proof-of-work: validating blocks by computing hashes with a certain property
  - Requires lots of repeated hashing
  - Computationally expensive
  - Participants (miners) are rewarded for successful validation (to incentivise validation)
  - “Cryptomining” can generate profit

Our problem starts with cryptocurrencies, which are peer to peer digital exchange systems without a central issuing authority, functioning as currencies,
and has become widespread in recent years.

Essentially, cryptocurrencies work because they rely on decentralized consensus to ensure the validity of transactions.

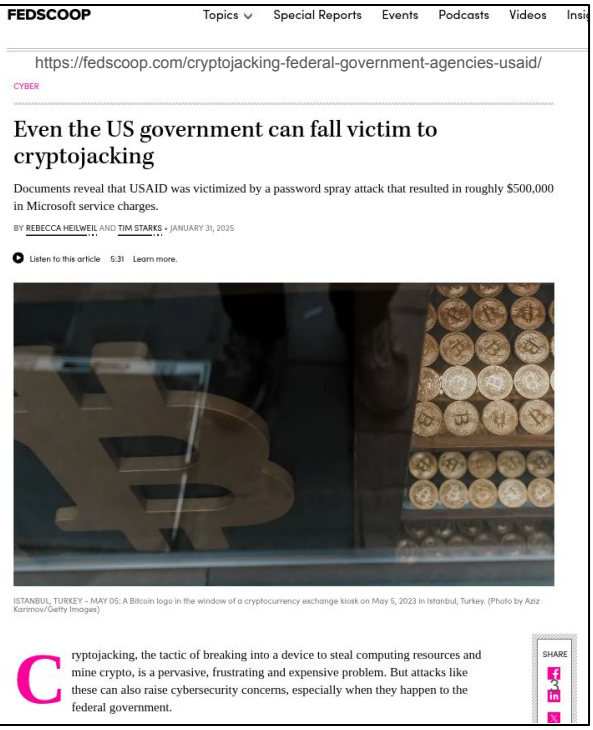
We focus on the most ubiquitous consensus mechanism, proof-of-work, in which validators
are required to provide proof that they possess a certain amount of computing resources before validating
new blocks.

Validation in proof of work often involves computing a hash with a difficult property on a block of transactions.
For example, a hash larger than a specific value.

This is computationally expensive, and miners are compensated with a reward for successful validation.
This process is called “Cryptomining” and can be profitable in many cases.

Cryptojacking: unauthorized mining

- Stealing computing resources to mine proof-of-work cryptocurrency
- Affects everything from large data centers to small IoT devices, causing financial and energy losses



FEDSCOOP Topics Special Reports Events Podcasts Videos Insi

https://fedscoop.com/cryptojacking-federal-government-agencies-usaid/

CYBER

Even the US government can fall victim to cryptojacking

Documents reveal that USAID was victimized by a password spray attack that resulted in roughly $500,000 in Microsoft service charges.

BY REBECCA HEILWEIL AND TIM STARKS - JANUARY 31, 2025

Listen to this article 5:31 Learn more.

ISTANBUL, TURKEY - MAY 05: A Bitcoin logo in the window of a cryptocurrency exchange kiosk on May 5, 2023 in Istanbul, Turkey. (Photo by Aziz Karimov/Getty Images)

Cryptojacking, the tactic of breaking into a device to steal computing resources and mine crypto, is a pervasive, frustrating and expensive problem. But attacks like these can also raise cybersecurity concerns, especially when they happen to the federal government.

SHARE 3

Cryptojacking is the unauthorized mining on stolen computing resources, meaning, using other people's computing power and taking the validation reward for oneself.

It affects everything from large data centers to small IoT devices and have caused major financial and energy losses even to large entities such as government agencies.

Our Focus: Detecting In-browser Cryptojacking

Cryptojacking delivered through websites

- “Fileless”
- Enabled by WebAssembly and WebWorkers (near native execution speed)
- Result: Attackers can deliver highly efficient mining scripts invisibly when a user visits a compromised or malicious webpage

Our focus is on detecting in-browser cryptojacking, which is when a user’s browser unknowingly runs a cryptominer delivered through a website.

This kind of malware is “fileless” as the payload is delivered transparently through the browser

It is enabled by WebAssembly and WebWorkers to execute at near native speed. And the result is attackers can deliver highly efficient mining scripts invisibly to a user visiting a malicious or compromised webpage

Key Challenge

- **Current detection methods are not robust**
  - **Network-based**: Susceptible to proxies, protocol deviations, new mining pools.
  - **Resource-based**: Sensitive to noise, throttling CPU usage
  - **Code Obfuscation** breaks semantic counters and ML binary analysis.
    - Easy to apply with tools such as Tigress, llvm-obf, etc.
    - Data encryption, control flow obfuscation, instruction substitution

The key challenge is that current detection methods can easily be evaded.

Network-based detection methods are susceptible to proxies or deviations from standard pool protocol.

Resource monitors are sensitive to noise, concurrently running processes, and throttling.

And lastly, semantic heuristics and static binary features have been shown to be easily broken by automatic code obfuscation,

which is particularly appealing because prior studies have shown that there is a low diversity of in-browser cryptominers in the wild originating from a small number of services (because cryptomining is only feasible at a large enough scale)

We focus on addressing this issue of code obfuscation resistant detection based on semantic features.

Key Insight

Mining algorithms perform **repeated hashing**.

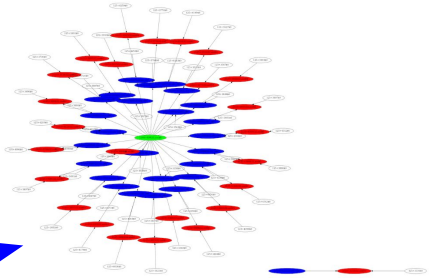
- **Many hashes computed** until a hash with certain property is found
  - (i.e. hash larger than some value)
  - This is true for all **PoW** cryptomining
- Each iteration, only the nonce field in the data block changes
- All hash are discarded except for one

The hashing function is called and discarded many times to find an acceptable hash

```
void cryptonight_hash(uint8_t* input, int len, uint8_t* hash) {
  // Initialize State
  ...
  for (int i = 0; i < 524288; i++) {
    // pseudo-random memory address
    ...
    // AES and mix into memory
    ...
    // pseudo-random memory read
    ...
    // multiply, shift, and mix
    ...
  }
  // Finalization
  ...
}
```

Memory-Hard Loop

- Executes 500,000 times per hash according to the design of *cryptonight*.
- Heavily uses 'and', 'xor', and 'shr'.
- Generates the massive, repetitive DFGs



(a) CryptoNight

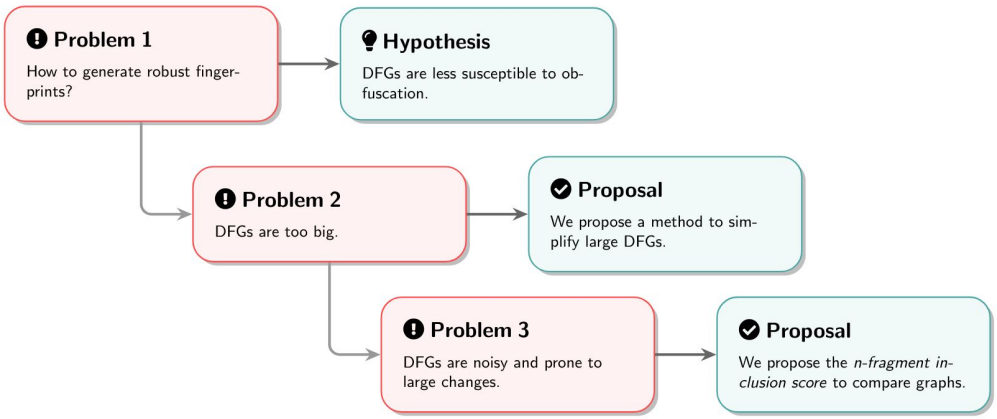
A key insight to detecting cryptominer is that mining algorithms perform repeated hashing.

Many hashes are computed until a hash with a certain property is found, for example, a hash larger than some value, and this is true of all proof of work cryptominers.

At each iteration, only the nonce field in the input data changes, and all hashes except for the last satisfactory one is discarded.

This repeated hashing manifests as massively repeated substructures as shown on the right image.
This data-flow graph is extracted from a popular mining algorithm called cryptonight and traces the data-flow into “and”, “xor”, and “shr” instructions.

The Proof-of-Theft (PoT) Framework



So, here are the problems and solutions that we propose.

The first problem is “how do we generate robust fingerprints for cryptominers”.

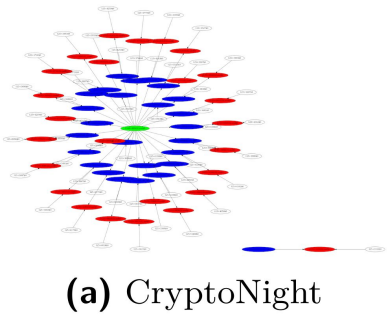
Since repetitive computation is an intrinsic property of proof-of-work schemes, and instruction-level data-flow graphs provide a structured and comprehensive view of computation that is difficult to manipulate. **We believe that DFGs are a good candidate**
for fingerprinting cryptominers.

Then, our **second problem** is that instruction level DFGs are massive due to the low abstraction level. So, we propose a method to simplify large DFGs

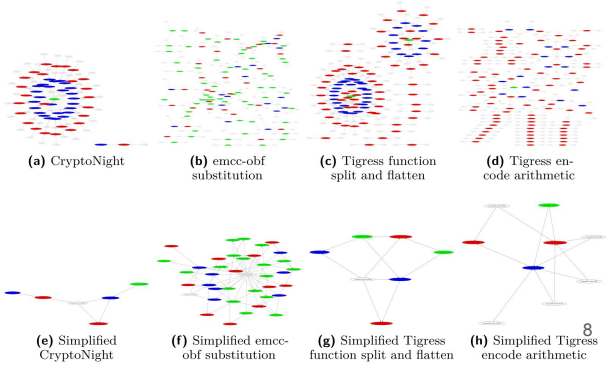
Finally, the **third problem** is that DFGs are noisy and prone to large changes, especially under the influence of obfuscation.
To solve this, we define the n-fragment inclusion score to compare graphs in a robust manner.

Analysis of Data-flow Graphs

- A directed acyclic graph representing the flow of data between executions of instructions. Variables are dynamically single-assigned.
- To reduce analysis overhead, focus on “and”, “xor”, “shr”.
  - Previous studies show these are highly characteristic of cryptographic hashing



(a) CryptoNight

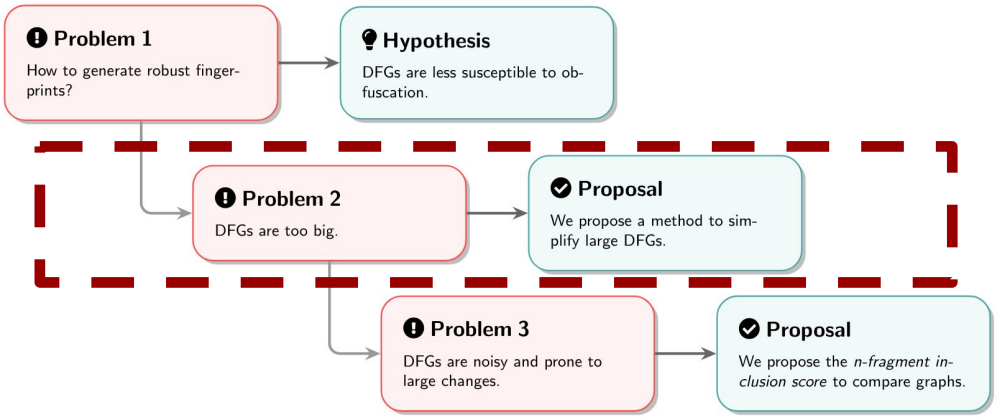


(a) CryptoNight **(b)** emcc-obf substitution **(c)** Tigress function split and flatten **(d)** Tigress encode arithmetic

(e) Simplified CryptoNight **(f)** Simplified emcc-obf substitution **(g)** Simplified Tigress function split and flatten **(h)** Simplified Tigress encode arithmetic

The data-flow graphs we use is a directed acyclic graph with no multiedges representing the flow of data between executions of instructions. And variables are dynamically single-assigned.

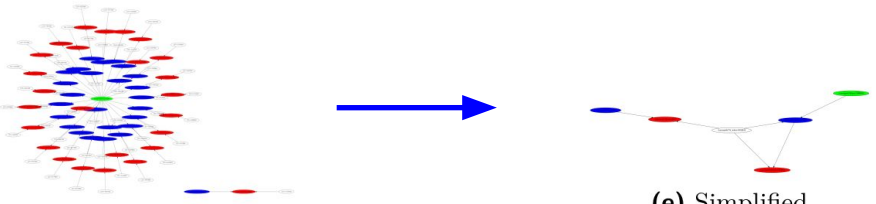
The left image is a very short snippet of the CryptoNight mining algorithm. On the right we show a few CryptoNight graphs and obfuscated versions, with the corresponding simplified versions at the bottom, which we will explain how to get later.



Next, we focus on solving the second problem, simplifying DFGs.

Part 1: Fingerprinting with Graph Simplification

- DFGs of dynamic single assigned variables contain regular substructures corresponding to redundant computations (loop iterations of the mining algorithm)
- Compress the graph by combining these structures
  - Preserves local semantics
  - Captures the core structure of the mining algorithm
  - Compressed graphs are our “fingerprints”



(a) CryptoNight

(e) Simplified CryptoNight

First we introduce a method to generate compact fingerprints from large DFGs. **DFGs of dynamic single assigned variables contain regular substructures corresponding to redundant computations (loop iterations of the mining algorithm)**

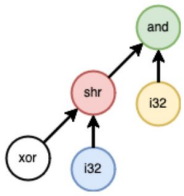
This is shown as branches in the cryptonight graph in the left image.

So intuitively, we can compress the graph by combining these repeated substructures until we get the representation on the right, and this still preserves local semantics and captures the core structure of the mining algorithm.

Defining Repeated Substructures

▶ **Definition 1** (Rooted Subgraph). *Let $G = (V, E)$ be a directed acyclic graph. A subgraph $S \subseteq G$ is a **rooted subgraph** if there exists $v \in V(S)$ such that every vertex in S is reachable from v. This v is unique when G is acyclic and is called the **root** of the subgraph. S is **maximal** if it is the largest subgraph with root v.*

▶ **Definition 2** (Depth). *The **depth** of a vertex $v \in V(G)$ is the number of edges on the longest path in G that ends in v. The depth of a rooted subgraph in G is the depth of the root vertex of the subgraph in G.*



11

To more precisely define the repeated substructures, we introduce the following concepts:

Rooted Subgraphs are basically subgraphs where all vertices are reachable from a
single "root" vertex, and maximal just means that it is the largest rooted subgraph for a particular vertex.

Next, the depth is an external property of a rooted subgraph H in the super graph G. It is essentially
the location of the rooted subgraph H in G, defined by the length of the longest path in G
ending in the root.

Taking this figure as an example, the blue i32 vertex induces a maximal rooted subgraph containing the blue, red, and green vertices.
And the depth of this rooted subgraph is zero.

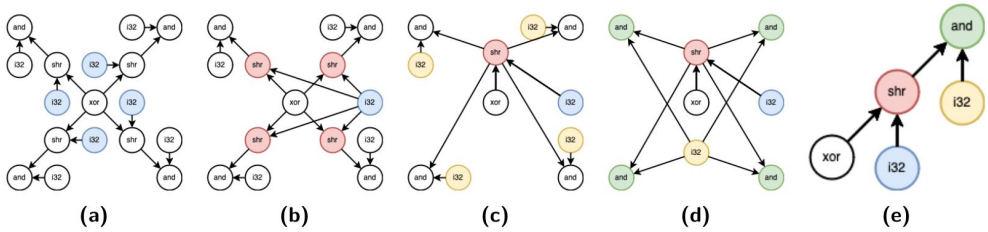
Intuitively, the depth of a rooted subgraph contains information about the
location and sequence of the corresponding instruction in the program that should be preserved in the simplified graph.

So the main idea here is that isomorphic maximal rooted subgraphs located at the same depth represent

the exact same computation happening in different iterations of the mining algorithm that we can merge.

Merging

Iteratively merge all distinct roots of maximal isomorphic rooted subgraphs of the same depth until we reach a fixed point.



This problem is GI-hard

We do this merging by iteratively merging all distinct roots of isomorphic maximal rooted
subgraphs of the same depth until we reach a fixed point. Note that merging just the root is equivalent to merging the whole subgraph because we do it until reaching a fixed point.

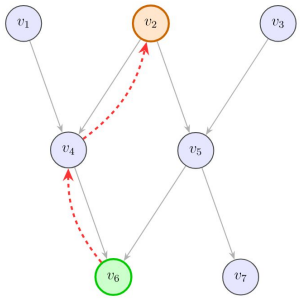
For example, in figure A, we have roots of isomorphic maximal rooted subgraphs
highlighted. They become merged in B, and we continue the process until we reach a fixed point in E, giving a simplified fingerprint.

But, we can show that this problem is GI-Hard, meaning that we probably can't solve it efficiently.

Approximation with Backward Random Walks

Intuition - Walking backward randomly on the graph starting from a random node, the probability of visiting a specific node is intrinsically tied to the shape, size, and structure of its descendant subgraph.

1. Pick a random vertex
2. Walk randomly opposite of edge
3. Stop when reaching a vertex with no incoming edge



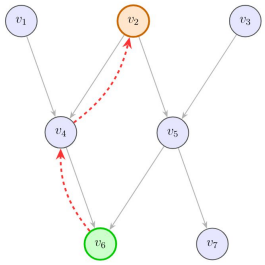
So instead, we use an approximation to identify isomorphic maximal rooted subgraphs to merge using backward random walks.

First, to perform a backward random walk, we pick a random vertex, and randomly choose a backward
edge to traverse at each step, stopping when we reach a vertex with no incoming edge

The intuition is that walking backward randomly on the graph starting from a random node,
the probability of visiting a specific node is intrinsically tied to the shape, size, and structure of its descendant subgraph.
So we can approximately identify similar maximal rooted subgraphs by the visit probability of their roots.

Mathematical Motivation

$$P(v) = \frac{1}{|V(G)|} + \sum_{v_c \in C(v)} \frac{1}{|I(v_c)|} P(v_c)$$



15

To put it more precisely, if we look at the probability that a vertex v is visited in a random backward walk,
we can show that it is equal to the probability that it is chosen as the first vertex in a backward walk (corresponding to the first term of this equation),
plus the probability that it visits a child of v first then proceeds backwards to v (corresponding to the summation term).

This essentially means **that two isomorphic maximal rooted subgraphs with isomorphic vertices containing the same number of incoming edges in G must have the same root vertex visit probability in a random backward walk.**

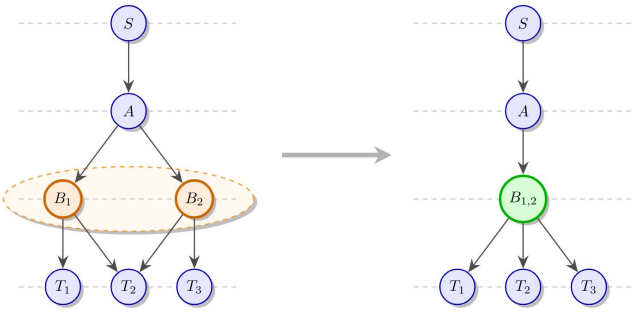
We describe this more precisely in our paper, but this essentially means that a lot of the structural information in the subgraphs is embedded in
the roots in a backward random walk. While **it is possible that distinct non-isomorphic maximal
rooted subgraphs may coincide with the same root vertex probability**,

or that isomorphic subgraphs may result in different root probabilities if some of the vertices have different
numbers of incoming edges, we expect this to happen rarely because this type of graphs
is very regular and don't contain diverse substructures.

So, we can approximately identify isomorphic maximal rooted subgraphs by

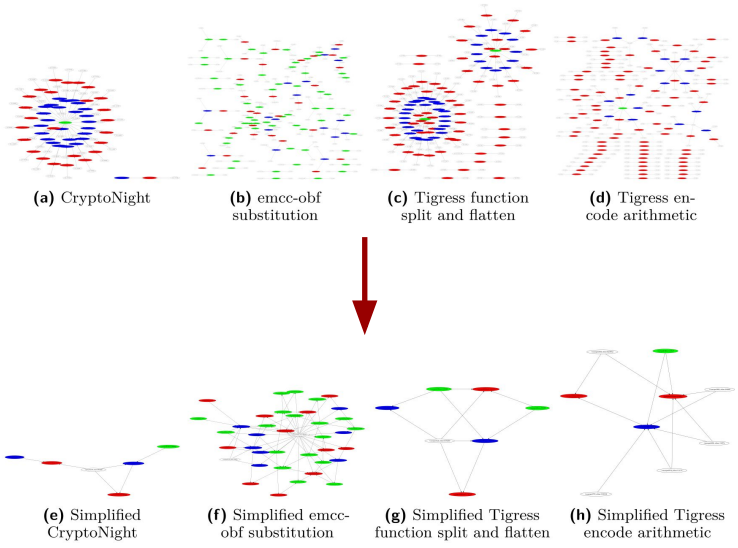
the random backward walk probabilities of their roots.

Approximate Simplification Algorithm



Therefore, our simplification algorithm is to merge all vertices with similar backward random walk visit probabilities at each depth in the graph.

Example Graphs



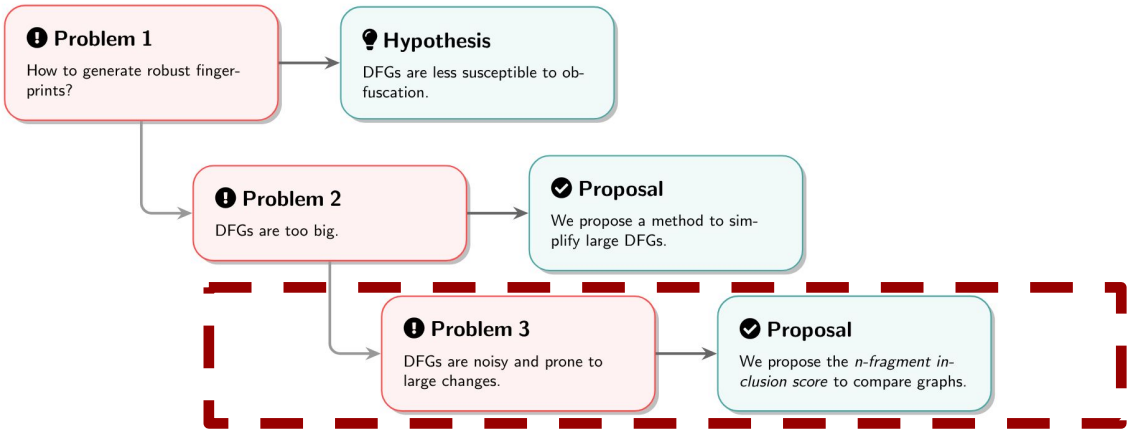
(a) CryptoNight **(b)** emcc-obf substitution **(c)** Tigress function split and flatten **(d)** Tigress encode arithmetic

(e) Simplified CryptoNight **(f)** Simplified emcc-obf substitution **(g)** Simplified Tigress function split and flatten **(h)** Simplified Tigress encode arithmetic

Here are some examples.

These 8 graphs are some obfuscated variants of cryptonight graphs.
The top row contains the original and a few obfuscations.
The bottom row has the corresponding simplified version.

And you can find more examples in our paper.



Next, we focus on solving the third problem, comparing noisy graphs.

n-fragment inclusion score (n-FIS)

An **asymmetric** measure

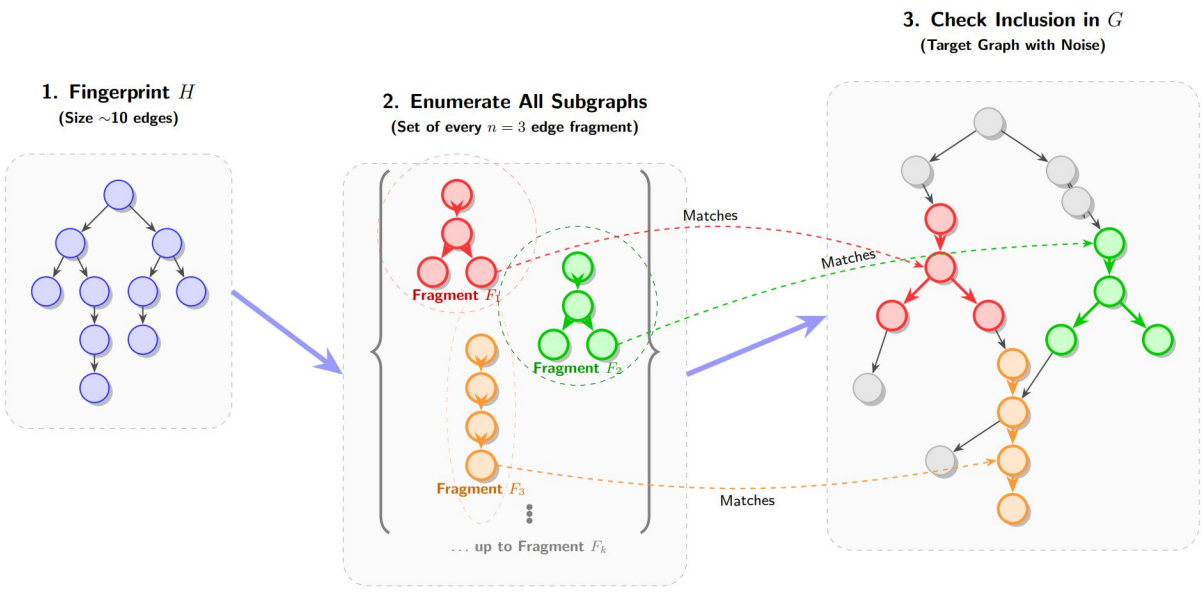
Concept: Look for **small, localized** fragments of the malicious graph.

▶ **Definition 10** (n-fragment inclusion score)**.** *The* ***n-fragment inclusion score*** *of a graph H in G, denoted n-$FIS_G(H)$, is the probability that an arbitrary connected subgraph $H' \subseteq H$ containing exactly n edges is a subgraph of G.*

18

To solve our problem, we define the n-fragment inclusion score as an asymmetric measure characterizing the degree to which a graph is included in another.
The main concept is to look for small, localized fragments of the malicious graph inside of our target sample.

Formally, the n-fragment inclusion score of a graph H in G is the probability that an arbitrary connected subgraph H' of H containing exactly n edges is a subgraph of G.
Basically, we are computing to what degree G contain small fragments of H.



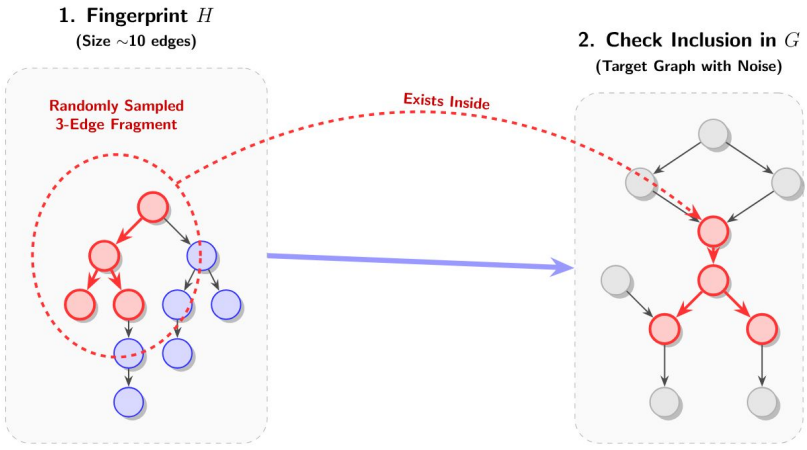
To visualize this measure, take the graph H on the left.
We take all fragments of size 3 edges from H. Here we illustrate only 3 of them.
Then, we check whether these fragments are inside of G somewhere.

And we do this for all fragments of size 3 edges in this case when n is 3.

This measure that we defined **satisfies our requirement for detecting subgraph similarity in that**

1. It identifies small local bevarior
2. It is asymmetric in the sense that it measures how much a graph is included in another
3. It tolerates noise that's a common side effect of obfuscation

A Simple and Performant Approximation



To simplify our implementation, avoid enumerating larger graphs, and tolerate inaccuracy in the subgraph matching tool we rely on, we use a simple and performant approximation.

Essentially we randomly sample the fragments instead of enumerating them, and the score is the percentage of the sampled subgraphs which are found in G by the subgraph matching tool.

Some Implementation Details

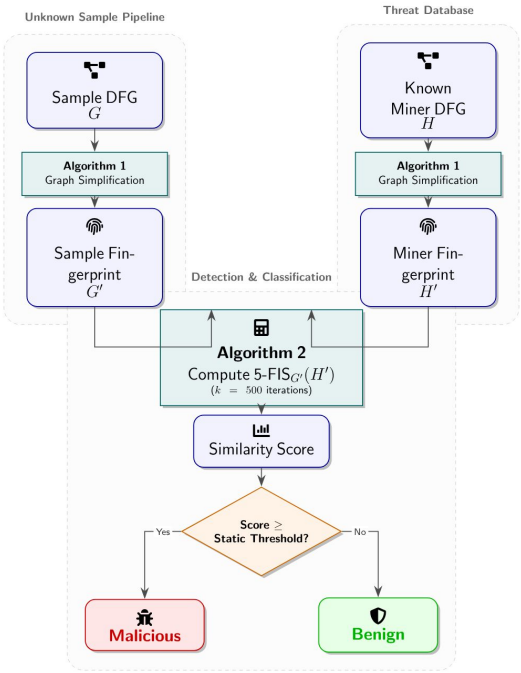
- **Wasabi Framework** for dynamic analysis
  - We use it to trace the execution and record data-flow
  - We implemented a shadow stack to trace the Wasm stack machine
- Collect the data-flow through the browser
  - Process the data-flow into graph format
- We only instrument "**and**", "**xor**", "**shr**", but can extend to more
- We perform subgraph matching using the ArcMatch tool

Next, these are some WebAssembly implementation details.

First, we use the Wasabi framework for dynamic analysis to trace and record data-flow. We implement a shadow stack to trace the wasm stack machine. **We then collect the data-flow graph through the browser's debug console and process them into a graph format.**

We choose to only instrument and, xor, and shift right instructions to reduce overhead because prior works have shown these to be highly characteristic of cryptomining, but we can easily extend the process to more instructions to support other platforms. **And finally, we perform the subgraph matching using the arcmatch tool.**

The PoT Framework Overview



The overall process is shown on this diagram.

Given a Sample DFG and a known miner DFG, we simplify both into fingerprints.
Then, we compare the fingerprints using Algorithm 2 to check the degree to which the fingerprint of H appears in G.

If the similarity score exceeds a threshold, we classify it as malicious.
The empirical threshold we found in our experiments is 0.65.
And we repeat this process for all malicious fingerprints H in the database.

Experimental Evaluation

- **Miners**: 6 open-source C/C++ Wasm miners
- **Obfuscated miners**: 39 obfuscated miner samples
  - Generated by applying Tigress, emcc-obf, WASMixer to miners
  - Arithmetic encoding, instruction substitution, function splitting, control flow insertion/flattening, memory encryption
- **Benign Apps**: 29 real-world Wasm web apps
  - (games, math simulators, video decoders, graphics software, etc.)
  - Derived from the Wasm-R3 benchmark by Baek et al.

Name	URL	Currency	Algorithm
btc	https://github.com/kinshukdua/cryptominer	Bitcoin	sha2
eth	https://github.com/Rachel-Hu/wasm-miner	Ethereum	ethash
zny	https://github.com/ohac/cpuminer	Bitzeny	yescrypt
cn	https://github.com/andrehrferreira/cryptonight-hash	any CryptoNight coins	CryptoNight v1
wmp	https://github.com/notgiven688/webminerpool	any CryptoNight coins	CryptoNight v4
xmr	https://github.com/jtgrassie/xmr-wasm	Monero	CryptoNight v1

Next, we experimentally evaluate our detection method.

We collected several working and open source C and C++ miners from GitHub.

Specifically, we require functional boilerplate code to execute the WebAssembly, preventing us from using binary samples from many previous works.
Additionally, we require C and C++ source code to apply the obfuscators.

We generate 39 obfuscated miner samples by applying 3 obfuscation tools to our miners.

And lastly, we source 29 real-world benign applications also used by prior works.

Baseline Detectors

- MINOS: CNN-based classification of Wasm binaries
- Minesweeper: Instruction count heuristics
- WASim: A collection of ML detectors on Wasm binary metadata
  - Random Forest, SVM, Naive Bayes Classifier, Neural Network

We tested our work against 3 baseline detectors. (You can find more details about how they work in our paper)

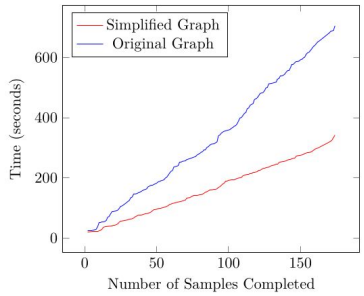
MINOS is a state-of-the-art convolutional neural network detectors which classify image representation of wasm binaries.

Minesweeper detects cryptominer binaries using a set of hand-crafted instruction count heuristics.

WASim uses machine learning models trained on features of the binary files attributes and metadata.

RQ1: The Effectiveness of Graph Simplification

- Compression stats: average 97.3% reduction in vertices and 96.5% in edges.
- Reduced analysis time by half.



The first question we answer in the experiment is how effective is the graph simplification.

First, among all the graphs in the dataset, there was an average of 97.3% reduction in vertices and 96.5% reduction in edges.
We also cut down the total analysis time by over half, as shown on the graph.

RQ2: The Effectiveness of PoT

- PoT:
  - 0/29 False Positives
  - 3/39 Missed Obfuscated Miner Samples
- MINOS (the best performing baseline):
  - 5/29 False Positives
  - 14/39 Missed Obfuscated Miner Samples

Method	Accuracy	Sensitivity	Specificity	Precision	f_1-score	Time (s)
PoT	95.6%	92.3%	100.0%	100.0%	96.0%	11.9
PoT (no simplify)	89.7%	87.2%	93.1%	94.4%	90.7%	19.8
MINOS	74.3%	68.9%	82.8%	86.1%	76.5%	4.78
Minesweeper	59.5%	95.6%	3.4%	60.6%	74.1%	2.01
WASim nn	45.9%	13.3%	96.6%	85.7%	23.1%	1.24
WASim rf	55.4%	33.3%	89.7%	83.3%	47.6%	0.110
WASim svm	39.2%	0.0%	100.0%	N/A	0.0%	0.095
WASim nb	50.0%	57.8%	37.9%	59.1%	58.4%	0.099

Our second question is, how well does PoT detect obfuscated cryptominers?

The result is that we greatly outperforms baselines in accuracy and f1 score, achieving a detection accuracy of over 95% on our benchmark.

Notably, PoT did not pick up any false positives from the benign samples, and only a few obfuscations had non-negligible effect on detection.

We also see that PoT works better with simplification, showing that **the simplification actually amplifies the differences between** cryptominers and benign applications.

Conclusion

- **Summary**: Data-Flow Graphs provide an intrinsic view of algorithmic behavior that obfuscators cannot easily hide
- **Contribution 1**: A scalable algorithm to simplify massive DFGs while preserving local substructures
- **Contribution 2**: The n-fragment inclusion score to compare noisy and fragmented graphs
- **Result**: PoT effectively detects obfuscated Wasm cryptominers

In summary, data-flow graphs provide an intrinsic view of mathematical computation that obfuscators cannot easily hide.
We created a framework for detecting in-browser cryptominers using DFGs.

To this end, we presented two main contributions

1. A scalable algorithm to simplify massive DFGs while preserving local substructures
2. And the n-fragment inclusion score to compare noisy and fragmented graphs

Our results show that PoT effectively detects obfuscated webassembly cryptominers and differentiates them from benign webassembly application.

That concludes my presentation. Thank you very much for listening, and I will be happy to take questions from the audience.