MinerRay: Semantics-Aware Analysis for Ever-Evolving Cryptojacking Detection

Alan Romano
alanroma@buffalo.edu
The State University of New York at Buffalo

Yunhui Zheng
zhengyu@us.ibm.com
IBM Thomas J. Watson Research Center

Weihang Wang
weihangw@buffalo.edu
The State University of New York at Buffalo

ABSTRACT

Recent advances in web technology have made in-browser cryptomining a viable funding model. However, these services have been abused to launch large-scale *cryptojacking* attacks to secretly mine cryptocurrency in browsers. To detect them, various signature-based or runtime feature-based methods have been proposed. However, they can be imprecise or easily circumvented. To this end, we propose MinerRay, a generic scheme to detect malicious in-browser cryptominers. Instead of leveraging unreliable external patterns, MinerRay infers the essence of cryptomining behaviors that differentiate mining from common browser activities in both WebAssembly and JavaScript contexts. Additionally, to detect stealthy mining activities without user consents, MinerRay checks if the miner can only be instantiated from user actions.

MinerRay was evaluated on over 1 million websites. It detected cryptominers on 901 websites, where 885 secretly start mining without user consent. Besides, we compared MinerRay with five state-of-the-art signature-based or behavior-based cryptominer detectors (MineSweeper, CMTracker, Outguard, No Coin, and minerBlock). We observed that emerging miners with new signatures or new services were detected by MinerRay but missed by others. The results show that our proposed technique is effective and robust in detecting evolving cryptominers, yielding more true positives, and fewer errors.

ACM Reference Format:
Alan Romano, Yunhui Zheng, and Weihang Wang. 2020. MinerRay: Semantics-Aware Analysis for Ever-Evolving Cryptojacking Detection. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416580

1 INTRODUCTION

Since Bitcoin [51] was introduced in 2009, over 2,300 cryptocurrencies have emerged [4]. In just a decade, cryptocurrencies have grown from a tiny niche to a huge industry with a $250 billion market capitalization [29]. Thanks to recent advances in web technologies, certain cryptocurrencies can be mined directly in the web browser. Since the first in-browser cryptomining service CoinHive was launched [3] September 2017, such services have become a possible revenue model for website owners. Simply by including a script, websites can use client browsers to make money [58].

However, this paradigm has been abused to mine cryptocurrencies without user consent. For example, just within three days after CoinHive was released, *The Pirate Bay*, a BitTorrent search engine, started to mine secretly, making it the first high-profile site to deploy CoinHive in scale. In the next few months, this malicious practice has become increasingly rampant. In Q4 2017, instances of cryptojacking have skyrocketed (increased by 8500%) [53], where cryptojacking malware that embedded CoinHive miners were found on over 30,000 websites [47]. Since then, in-browser cryptojacking has become one of the most prominent online threats [45, 63, 66]. In March 2019, CoinHive shutdown its site and servers in March 2019 [26]. However, this has not removed the threat of in-browser cryptojacking attacks [70].

Some miners use WebAssembly [71], a new type of language that can be run in browsers at near-native speed. For instance, Sumokoin [57] and Lethean [30] are built atop the CryptoNight hashing algorithm [62] or its variants like CryptoNight-Heavy [56]. Some are implemented in JavaScript. For example, CoinIMP [14] is leveraging asm.js [2] as an alternative to WebAssembly. Others use both JS and WebAssembly. For example, WebDollar [73], a newly invented web cryptocurrency, implements the Argon2 hash algorithm [23]. Moreover, mining services keep introducing new cryptominers and updating underlying hashing algorithms. For instance, CryptoLoot [27], a popular cryptomining service, will completely switch to mining uPlexa [68] from Monero [58].

Considerable research efforts have been made to develop browser cryptojacking defense techniques [33, 40, 42, 43, 49, 61]. In particular, CMTracker [40] identifies cryptominers by calculating the cumulative time of the websites spent on signature hash functions and profiling stack structures to look for repetitive behavioral patterns. MineSweeper [43] counts the number of bit operations to recognize cryptographic hash functions. Outguard [42] is another state-of-the-art detector that uses machine learning techniques to detect cryptominers. It instruments a web browser and collects features such as the number of web workers and the presence of WebAssembly to train an SVM classifier. Unfortunately, existing approaches fall short because they are either limited to specific hashing algorithms/implementations or assume particular runtime behaviors, which vary greatly among different cryptocurrencies. These limitations lead to imprecise results and difficulties in detecting new cryptocurrencies. In particular, we made the following observations on the ever-evolving landscape of in-browser mining:

- We argue that the majority of CPU-minable cryptocurrencies can be mined in browsers, providing the market value is attractive

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416580

and the mining mechanism is anonymous. Consequently, miners will exhibit variance that makes detection challenging.

- Detectors targeting particular hashing algorithms cannot detect miners using different algorithms. For example, MineSweeper focuses on the CryptoNight used by Monero [43]. It cannot detect uPlexa [67], which uses CryptoNight-UPX [69].
- Detectors based on signature functions or particular runtime behaviors are less effective in detecting cryptominers for new cryptocurrencies, as such patterns can be significantly different.
- Several techniques assume miners are always implemented in WebAssembly. They cannot detect JavaScript-based miners such as CoinIMP and WebDollar.

In this paper, we propose a robust cryptojacking detector, MinerRay, which is resilient to variants of hashing algorithms and implementations. Unlike existing strategies based on fragile patterns such as the number of particular operations, execution time, or WebAssembly features, our technique relies on *program semantics* that are *invariant across programming languages and implementations of cryptomining algorithms*. Moreover, MinerRay distinguishes malicious cryptojackers from benign ones, by inspecting interactions between users and cryptomining modules.

Contributions. This paper makes the following contributions:

- We propose an intermediate representation (IR) to abstract underlying semantics of miners written in WebAssembly and JavaScript, which supports cross-language analysis and is resilient to variants (e.g., different versions of the same hashing algorithm or binaries generated by different compilers) (Section 5.1.2). To the best of our knowledge, MinerRay is the first to perform cross-language analysis on WebAssembly and JavaScript for detecting cryptominers.
- We develop a light-weight static analysis that infers the critical steps of hashing and reasons about user consent (Section 5.3).
- We evaluate MinerRay on over 1 million websites. It identified miners on 901 websites connecting to 12 unique mining services, where 885 websites start mining without user consent.
- We perform an extensive comparison study with five state-of-the-art detectors, namely: MineSweeper, CM-Tracker, Outguard, No Coin, and minerBlock. MinerRay detected the most websites with the least errors (false positives and false negatives combined). Besides, miners with new signatures and new services were detected by MinerRay but missed by others.
- We make our implementation and results publicly available [15].

2 PRELIMINARY

2.1 In-Browser Cryptomining

In-browser cryptominers are getting more sophisticated. An in-browser cryptominer can create a distributed cryptomining pool with its processing power without client-side software installation [49]. Cybercriminals can stealthily deliver such code and mine cryptocurrency using the client-side resources without consent. Figure 1 shows a typical workflow of such a cryptominer:

❶ A user visits a web page with a link to a miner.
❷ The miner is delivered as JavaScript and creates a WebSocket to connect the miner and mining pool.

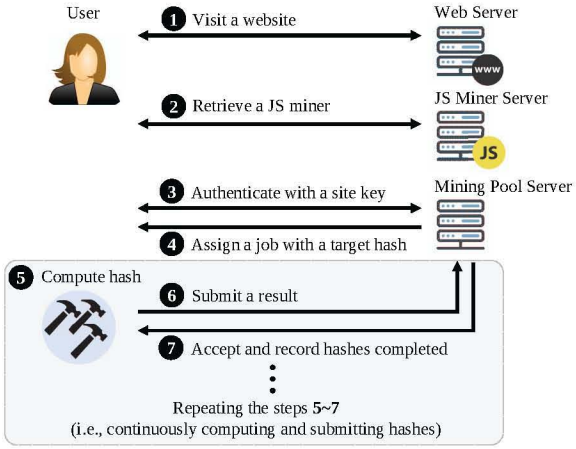


Figure 1: In-Browser Cryptomining.

❸ The miner is authenticated with a site key that ties computed hashes to the miner.
❹ Once the miner is authenticated, the mining pool server assigns a job to the miner with a target hash.
❺ Web workers are created to compute the hash in parallel.
❻ The satisfying hash is sent to the mining pool server.
❼ Once verified, the mining pool server accepts the result and assigns new tasks. The web workers repeat steps ❺ – ❼.

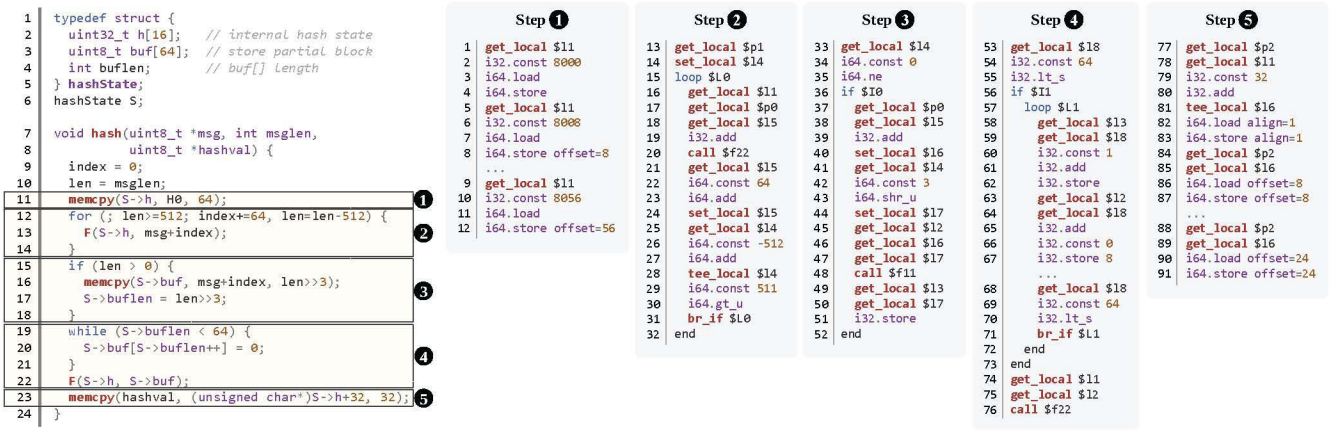
To be stealthy, miners are usually configured to use less than 100% of the CPU. If throttling is specified, the web worker thread is paused momentarily to restrict CPU usage. Otherwise, the worker thread keeps running. When the goal is met, miners are rewarded with cryptocurrencies for computing a certain number of hashes. In cryptojacking scenarios, the funds go to the attacker's wallets.

2.2 Hash Functions

The core of cryptomining is to compute hashes via hash functions. A hash function maps an input of arbitrary length to a fixed-length output. Such computations are usually done in an *iterative* manner. In particular, the input message M is partitioned into blocks of a specific size. Then, a compression function f is applied iteratively on each message block m_i to compute the hash $h_i = f(h_{i-1}, m_i)$ for $i = 1$ to n. Therefore, each iteration is fairly segregated due to the data dependencies from its previous iteration.

Figure 2 shows a typical hash function implemented in C and the corresponding WebAssembly. In Figure 2(a), the hash function consists of five steps: ❶ A 512-bit (i.e. 64-byte) hash state `H0` is initialized and copied to a buffer. ❷ The message is divided into 512-bit message blocks and processed by a compression function `F()`. ❸ The remaining part is copied to a temporary buffer `S→buf`. ❹ The last partial block is padded with 0s and processed by the compression function. ❺ The result is a 256-bit hash value.

Compared to common programs running in browsers, this iterative procedure is unique and a likely indicator of cryptomining. Based on this observation, MinerRay looks for the semantics corresponding to the critical steps above and identifies hash functions.



(a) C Implementation.

(b) WebAssembly Opcodes.

Figure 2: Hash Function (C implementation and WebAssembly Opcodes of the Function).

3 MOTIVATION

Background: Evolving Cryptominers. In-browser cryptominers are becoming more diverse and sophisticated. For example, CryptoNight, one of the most popular cryptomining algorithms, has more than 9 versions originated from 3 variants [12].

In addition, cryptominers are written in more diverse languages. WebAssembly is no longer the only choice for in-browser cryptominers. The recent trend of developing less computation extensive PoW (Proof-of-Work) algorithms [5] also opens up the opportunities for JavaScript (and asm.js) based cryptominers. In fact, there are already miners using both WebAssembly and JavaScript [73].

Besides, cryptominers are trying to be less noticeable by maintaining reasonable workloads. This is because, in part, earlier versions of cryptominers excessively consume resources, making users notice them and remove them quickly. Recent cryptominers do not aggressively take the computation resources over. To this end, the above trends lead to the three significant challenges in detecting cryptominers in the wild: (1) various cryptomining algorithms, (2) cryptominers written in diverse programming languages, and (3) cryptominers becoming less greedy in taking resources.

In this section, we use three emerging cryptocurrencies to explain the three challenges of detecting evolving cryptominers.

Variant Algorithm (uPlexa [68]). uPlexa is an untraceable digital currency atop CryptoNight-UPX algorithm. It has gained considerable popularity. CryptoLoot switched from Monero to uPlexa in October 2019. Given Monero and uPlexa are based on different hashing algorithms (CryptoNight and CryptoNight-UPX respectively), we conjectured detection tools focusing on CryptoNight will fail to detect uPlexa. To verify this speculation, we run MineSweeper on a Monero miner and a uPlexa miner found on websites.

MineSweeper [43] identifies CryptoNight-based miners by counting the number of bit operations and matching them against five hash functions used by the CryptoNight algorithm: BLAKE [18], AES [32], Groestl [36], Keccak [21], and Skein [35]. If more than three hash functions are observed, MineSweeper concludes a CryptoNight based cryptominer is detected. However, MineSweeper failed to detect the uPlexa miner, indicating that the number of bit operations is not a robust feature. Therefore, classification methods atop it can be circumvented or misclassify new algorithms. By contrast, MinerRay successfully detects the uPlexa miner as it focuses on the hash function semantics and is thus algorithm-agnostic.

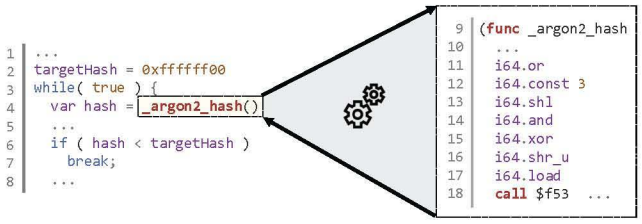
JavaScript-based Miners. There are JavaScript based cryptominers including CoinIMP [14] and JSECoin [65]. CoinIMP is written in JavaScript implementing lyra2-webchain egalitarian algorithm [11]. In particular, it leverages asm.js implementation as an alternative to WebAssembly, providing better user-experience when WebAssembly technology is not available at runtime. CoinIMP and JSECoin vary greatly in program languages, signature functions and runtime behaviors. We made the following observations:

- CMTracker cannot detect them because they do not contain the modeled signatures. Besides, their repetitive behaviors do not match the threshold patterns.
- We were not able to run Outguard on JSECoin miners. Outguard does not support Chrome 65+, while JSECoin requires the `Array.flat()` that is available after Chrome 69. Instead, we manually inspected the code and found the signatures modeled by Outguard are not present in JSECoin miners.
- MineSweeper does not support JS miners and misses them.

Because our analysis is based on language-independent intermediate representation (IR), MinerRay supports both languages and successfully detects such JS-based miners.

JS-WebAssembly Hybrid Miner (WebDollar [73]). As shown in Figure 3, WebDollar uses both JavaScript and WebAssembly. The Argon2 hash function is implemented in WebAssembly and its output is processed by JavaScript to iteratively computes the hash until the result contains the desired number of leading 0s.

MineSweeper, CMTracker, and Outguard cannot detect the WebDollar miner because they were not able to handle both JavaScript and WebAssembly at the same time. To detect such a hybrid miner, in addition to understanding both JavaScript and WebAssembly programs, the interoperation (as shown in Figure 3(c)) should be analyzed to connect the JavaScript and WebAssembly programs.



(a) JavaScript Code. (b) JS-WebAssembly Interoperation. (c) WebAssembly Opcodes.

Figure 3: WebDollar Miner.

Table 1: Comparison with State-of-the-art Detectors.

	MineSweeper	CMTracker	Outguard	MinerRay
Traditional	Yes	Yes	Yes	Yes
Variant Algo.	No[1]	No[1]	No[1]	Yes
JS-based/Hybrid	No[2]	Limited[3]	Limited[4]	Yes
Stealthy	No[5]	No[5]	No[5]	Yes

1: They focus on CryptoNight algorithms. New variants are often outside of their models. 2: Does not support JavaScript. 3: Insufficient models. 4: Does not support Chrome 65+, while some of those miners require Chrome 69 or newer. 5: Models focus on aggressive cryptominers.

To the best of our knowledge, MinerRay is the first technique that understands and connects logics in both languages to detect such a cryptominer. MinerRay successfully identifies the miner.

Summary. Table 1 summarizes the comparison results. We observe existing techniques do not perform very well to detect cryptominers using variant algorithms, multiple languages, and stealthy methods.

In particular, miners using variant algorithms (e.g., uPlexa and Proof-of-Work based miners) are usually missed. JavaScript and hybrid cryptominers (e.g., CryptoLoot, CoinIMP, JSECoin, and WebDollar) also impose challenges to the existing techniques. CMTracker and Outguard are partially affected by those cryptominers written in diverse programming languages because they focus on runtime behaviors. However, the way of obtaining the behaviors are partially dependent on the languages as they count known languagle dependent primitives (e.g., key operations and known computation functions). Finally, stealthy cryptominers are designed to circumvent detections by exposing different resource usage patterns. Because most existing techniques rely on external patterns that can be easily changed, they do not perform well. In contrast, MinerRay can detect these miners successfully because MinerRay focuses on program semantics of hashing.

4 THREAT MODEL

In-browser Cryptojacking. We assume crypto miners are delivered via web pages and steal client-side computing resources without consent. Any user visiting that page becomes a victim of in-browser cryptojacking attacks. Attackers can host such code on their websites or compromised websites. They can even deliver miners via malicious advertising or other third-party services. Cryptojacking outside of the web browsers is out of the scope.

Impact of User Consent. Existing cryptominer detectors often focus on identifying cryptomining activities, without considering users' consent. Note that cryptomining activity itself is not malicious, while it is malicious if the mining activity is done without users' awareness. MinerRay takes users' consent into account.

5 SYSTEM DESIGN

System Overview. Figure 4 shows the workflow of MinerRay, consisting of three major components: *Programming Language Lifting* (Section 5.1), *CFG Construction* (Section 5.2), and *Cryptojacking Detection* (Section 5.3).

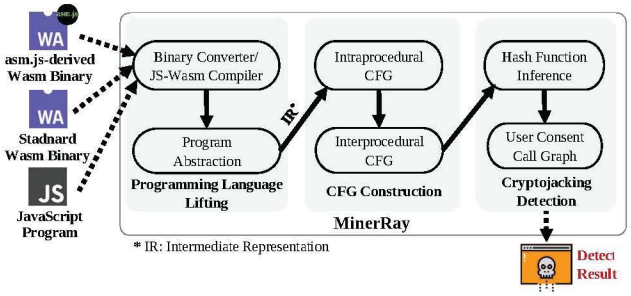


Figure 4: Overview of MinerRay.

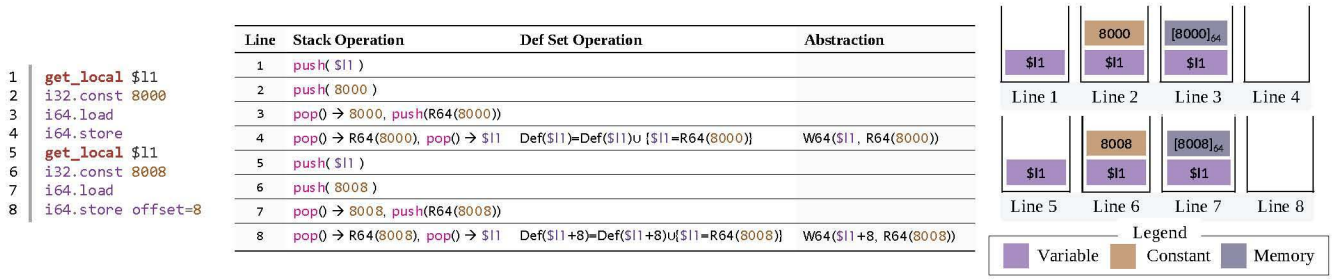
5.1 Programming Language Lifting

Cryptominers are usually implemented in multiple languages such as WebAssembly and JavaScript. Although the main module is usually written in one language, its cryptomining algorithm components can be written in both WebAssembly and JavaScript [73]. Moreover, there are non-standard WebAssembly binaries such as asm.js-derived modules that include customized opcodes. Such diversity imposes a significant challenge in analyzing cryptominers in the wild, as they can be implemented by a combination of the above diverse languages.

To effectively analyze cryptominers, we first convert both JavaScript and WebAssembly programs, including the non-standard asm.js-derived WebAssembly, to standard WebAssembly binaries (Section 5.1.1). Then, we lift the standard WebAssembly into a high-level intermediate representation (Section 5.1.2).

5.1.1 Converting Programs Written in Diverse Programming Languages into Standard WebAssembly. We convert a JavaScript program to WebAssembly in a way similar to AssemblyScript [13]. For the asm.js-derived modules, we convert non-standard asm.js opcodes to standard WebAssembly according to the equivalent instruction mapping shown in Table 2. The detailed mapping (i.e., the complete list of equivalent instructions) can be found on our project website [15].

5.1.2 Lifting Programs via Program Abstraction. We introduce a set of abstraction rules to translate WebAssembly opcodes to our IR, which is later used to capture high-level semantics correspond to the critical tasks in cryptominers. Table 3 shows a subset of the abstraction rules. Due to the space limit, we only list the key WebAssembly instructions: accessing local variables and selective unary/binary operations with operand(s) of type `i32` (32-bit integer). Operations on global variables (`get_global`, `set_global`)



```
1 get_local $l1
2 i32.const 8000
3 i64.load
4 i64.store
5 get_local $l1
6 i32.const 8008
7 i64.load
8 i64.store offset=8
```

Line	Stack Operation	Def Set Operation	Abstraction
1	push($l1)		
2	push(8000)		
3	pop() → 8000, push(R64(8000))		
4	pop() → R64(8000), pop() → $l1	Def($l1)=Def($l1)∪ {$l1=R64(8000)}	W64($l1, R64(8000))
5	push($l1)		
6	push(8008)		
7	pop() → 8008, push(R64(8008))		
8	pop() → R64(8008), pop() → $l1	Def($l1+8)=Def($l1+8)∪{$l1=R64(8008)}	W64($l1+8, R64(8008))

(a) WebAssembly Code. (b) Abstraction Rules Applied (c) Stack States

Figure 5: Running Example of Abstraction Rules.

Table 2: Asm.js Opcode Mapping Examples

Instruction Type	Count	Opcode Mapping	Mnemonic Mapping
Integer Division	4	0xd3 ↦ 0x6d	I32AsmjsDivS ↦ i32.div_s
Floating Point Math	11	0xa3 x y ↦ (0x44 x, 0x94) × y	F64Pow(x, y) ↦ (f64.const x, f64.mul) * y
Memory Load	7	0xd7 ↦ 0x2c	I32AsmjsLoadMem8S ↦ i32.load8_s
Memory Store	5	0xde ↦ 0x3a	I32AsmjsStoreMem8 ↦ i32.store8
Type Conversions	4	0xe3 ↦ 0xa8	I32AsmjsSConvertF32 ↦ i32.trunc_f32_s

and value types other than `i32` (`i64`, `f32`, and `f64`) are modeled similarly and thus omitted.

Abstracting Stack Operations. WebAssembly execution is based on a stack machine architecture, so, our abstraction models WebAssembly instructions based on the effects on the stack. It captures the data and control flows. In particular, for instructions that do not assign variables or contain control constructs, we only model their execution on a virtual stack.

"`i32.const` c" pushes an `i32` constant c onto the stack. "`i32.shl`" pops two values from the stack, performs a shift left operation and pushes the resulting `i32` value back to the stack. "`i32.load`" pops the top value on stack as an address, reads 4 bytes from that address and pushes the resulting `i32` value onto the stack. Note that the prefix `i32` indicates the size of the operand is a 32-bit integer. For a 64-bit integer, `i64` will be used as a prefix.

For assignments, we update the alive variable definitions and generate C-like abstractions. For example, "`set_local` v" pops a value e from the stack and assigns e to v. This instruction is abstracted to "$v = e$". "`tee_local`" is similar to "`set_local`", except that the top value on the stack is not popped. "`i32.store`" is modeled by "`W32(`e_2`,` e_1`)`", where the e_1 and e_2 are the values popped from the stack firstly and secondly.

Abstracting Structured Control Flow Constructs. WebAssembly supports control flow constructs such as `block`, `loop` and `if`. We model them as abstract control flow modules.

Conditionals and loops are modeled by `goto` and guarded `goto`: "`loop` g:" creates a label g and is abstracted as "g:". "`if` I" is abstracted to "`if` (e)", where the condition e is popped from the stack.

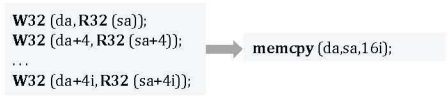
Table 3: Abstraction Rules.

Instruction	Stack Operation[1,2,3]	Def Set Computation	Abstraction
get_local v	push(v)		
set_local v	pop()→e	Def(v) = Def(v) ∪ {e^l}	v = e;
tee_local v	top()→e	Def(v) = Def(v) ∪ {e^l}	v = e;
i32.const c	push(c)		
i32.add	pop()→e_1, pop()→e_2, push(e_1 + e_2)		
i32.shl	pop()→e_1, pop()→e_2, push(e_2 ≪ e_1)		
i32.gt_s	pop()→e_1, pop()→e_2, push(e_2 > e_1)		
i32.load	pop()→e, push(R32(e))		
i32.store	pop()→e_1, pop()→e_2	Def(e_2) = Def(e_2) ∪ {addr(e_1)^l}	W32(e_2,e_1);
loop g:			g:
if I	pop()→e		if (e) {
br g			goto g;
br_if g	pop()→e		if (e) goto g;
return	if(retLen()==1) pop()→e		return e; or return
block B			B: {
end			}
call f	paraNum(f)→n, for(i = n; i > 0; i--) pop()→e_i		f(e_1,e_2,...,e_n);
call_indirect	pop()→e, paraNum(e)→n, for(i = n; i > 0; i--) pop()→e_i		f(e_1,e_2,...,e_n);

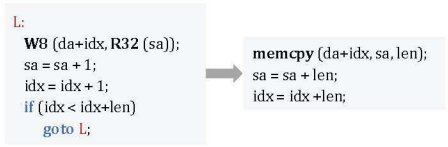
1. retLen() gives the number of return values (either 1 or 0) of current function.
2. paraNum(f) returns the number of return values of function f.
3. func(e) returns the function name by checking the function table with index e.

"`br` g" is a jump to label instruction. "`br_if` g" is a conditional jump to label g with the condition popping from the stack.

A return instruction either returns a value that is popped from the stack or returns a special value "`None`". Direct calls are parameterized with explicit function names whereas indirect calls are parameterized with an index to the function table. Function parameters are obtained from the stack. Note that each instruction is annotated with a label, denoting its line number in the program.



(a) Merge Sequential Writes.



(b) Merge Consecutive Writes in a Loop.

Figure 6: Abstraction Merging Rules.

Abstracting Memory Operations. We observe that hash functions extensively use *consecutive* memory copy and access operations. We further merge consecutive memory buffer writes based on the rules summarized in Figure 6. Sequential writes to consecutive memory buffers are simplified to a single `memset` or `memcpy` statement (Figure 6(a)), where parameters "*da*" and "16*i*" represent the destination address and the number of bytes to be written respectively. The semantics of writing to consecutive memory buffers can also be realized by using a loop to write each byte (Figure 6(b)).

Running Example. Figure 5 presents how the MinerRay lifts a WebAssembly program using the abstraction rules shown in Table 3. Figure 5(a) shows a few WebAssembly opcodes extracted from the motivation example in Figure 2 (precisely from Step ❶). Figure 5(b) presents which abstraction rule is applied while each line of the code in Figure 5(a) is interpreted, including the corresponding stack and Def set operations, as well as resulting abstractions (i.e., IR). Figure 5(c) shows a state of the stack during the interpretation of each line.

Initially, the virtual stack is empty (Line 1 in Figure 5(c)). At lines 1-2, the value of "$l1" and address 8000 are pushed onto the stack, according to the stack operations shown in Table 3. The second column of Figure 5(b) shows the concrete stack operations. Next, line 3 loads the content at 8000 and pushes the result R64(8000). At line 4, "`i64.store`" pops "$l1" and R64(8000) from the stack. It then performs a store operation using "$l1" as the address and R64(8000) as the value. We generate an abstraction (i.e., a statement) "W64($l1, R64(8000))" and update the definition set (def set) of "$l1".

Similarly, lines 5-8 are abstracted to "W64($l1 + 8, R64(8008))". Essentially, the semantics of each four-line block is copying the memory content starting at 8000 and 8008 to address "$l1" and "$l1 + 8". To facilitate analysis, we further merge consecutive memory copies so lines 1-8 are abstracted to "memcpy($l1, 8000, 16)", according to the abstraction merging rules shown in Figure 6.

5.2 CFG Construction

Given the lifted IR obtained in Section 5.1, we extend our analysis to understand the entire program. Specifically, we construct a control flow graph for each procedure and link them together to obtain an interprocedural control flow graph for the program.

Intraprocedural CFG. We first build an intraprocedural CFG for each function. Then, we identify loops and repetitive operations (i.e., manually unrolled loops) to further abstract them to higher level representations. For example, a "`for`" construct that stores the same constant value (e.g., 0) in consecutive memory will be replaced by a `memset` abstraction.

Interprocedural CFG (ICFG). We link intraprocedural CFGs together to build an interprocedural CFG (ICFG) and represent the entire program. There are two major challenges: (1) A recursive call in a program can introduce loops. When a recursive function call is found, we simply treat it as an iterative operation for further analyses. (2) An indirect function call may result in an imprecise interprocedural CFG. As targets of indirect calls are only known at runtime, MinerRay conservatively assumes that any functions that match with the callee's function type can be possible targets. This results in a larger ICFG but does not introduce false negatives.

5.3 Hash Function Detection

A hash function generally has these critical steps to (1) initialize hash state, (2) hash each message blocks, (3) store remaining data in a temporary buffer, (4) pad and hash the last partial block, and (5) generate the hash from the final state.

As suggested in [20, 31, 55], almost all existing cryptographic hashes can be described as functions based on a block cipher, where step 2, 3 and 4 are necessary in block cipher implementations. Therefore, the key idea of our approach is to see if a program exhibits semantics matching above steps.

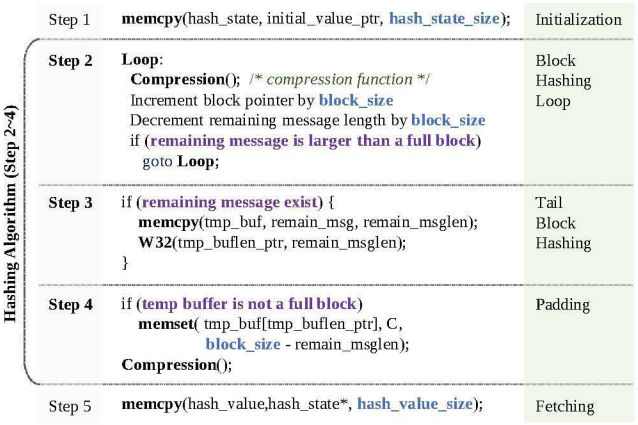


Figure 7: Hash Function Semantics.

5.3.1 Hash Function Semantics. We formulate the semantics of above critical steps as a template (Figure. 7). Steps 1 and 5 model the initialization and the result fetching. They are not strong signals due to the variances in hashing algorithms and excluded as patterns.

- Step-2 captures the iterative block hashing loop. In each iteration, the compression function is invoked on each block. The *block size*, the *message pointer*, the *original message length* and the *remaining message length* should be inferred.
- Step-3 processes the remaining message. If the remaining message exists, it is copied to a temporary buffer. The *temporary buffer* and *buffer length* should be identified.

- Step-4 represents the padding process for the remaining message. A `memset` or `memcpy` is used to fill out the buffer to the full block size, starting from the current temporary buffer length.

5.3.2 Matching Models to Programs. Algorithm 1 describes the algorithm to identify the critical steps: `D_HashEachBlock` recognizes message block hashing loops. `D_StorePartialBlock` checks if a partial block is copied to a temporary buffer. `D_PadLastBlock` detects if the last partial block is padded.

The detection algorithm takes the control flow graph (CFG) of the abstracted program and the common sizes of block ciphers (e.g., 256 bits) as input. The final output represents potential hash algorithms identified from the program.

Algorithm 1: Inferring Hash Functions

```
   Input: CFG(V, E), BSIZES[]={256,512,1024}
   Output: HashCandidates[], HashEachBlock[], StorePartialBlock[]
1  D_HashEachBlock(CFG(V,E),BSIZES):
2  for loop l ∈ V do
3      for size ∈ BSIZES do
4          cond_stmt ← conditional statement in l
5          de_stmt, in_stmt ← statement in l de/increased by size
6          if (de_stmt||in_stmt) && MoreFullBlocks(cond_stmt) then
7              block_size ← size
8              rem_msglen, block ← operand in de_stmt, in_stmt
9              msglen ← get last definition of rem_msglen
10             HashEachBlock = HashEachBlock ∪ {l, block_size, rem_msglen, block,
                 msglen}
11 D_StorePartialBlock(CFG(V,E), HashEachBlock):
12 for construct l ∈ HashEachBlock do
13     for conditional node m ∈ Successors(l) do
14         cond_stmt ← conditional statement in m
15         if (PartialBlockExists(cond_stmt) && memcpy exists) then
16             tmp_buf, rem_msg, rem_msglen ← memcpy args
17             tmp_buflenptr ← store rem_msglen
18             StorePartialBlock = StorePartialBlock ∪ {l, m, tmp_buf,
                 tmp_buflenptr}
19 D_PadLastBlock(CFG(V,E), StorePartialBlock):
20 for construct m ∈ StorePartialBlock do
21     for conditional node n ∈ Successors(m) do
22         cond_stmt ← conditional statement in n
23         if IsNotFullBlock(cond_stmt) && memcpy/set exists then
24             if PadConstantsToBuffer(args of memcpy/set) then
25                 l ← HashEachBlock construct of m
26                 HashCandidates = HashCandidates ∪ {l, m, n}
```

D_HashEachBlock iterates all loop nodes in the CFG to detect nodes that partition a message to fixed-size blocks. Specifically, the function checks the loop condition and in/decrement expressions (lines 4-10). If a variable is decremented by a particular block size, it is a candidate of the remaining message length (`rem_msglen`). Similarly, the current message pointer `block` is inferred if a variable is increased by the block size. It compares the lengths of the remaining message and the full block in a conditional statement to match against hashing iteration predicates.

D_StorePartialBlock checks the successors of the nodes in `HashEachBlock` and looks for predicates guarding the statements that copy the remaining message to a temporary buffer. At line 15, a helper function `PartialBlockExists` checks if a partial block exists. The condition should either compare the remaining message length (`rem_msglen`) with 0 or compare the current message pointer (`block`) with the entire message size (`msglen`). If the partial block exists and `memcpy` can be found within this node, the pointer to the temporary buffer (`tem_buf`) and its length (`tem_buflenptr`) can be inferred. Then, `StorePartialBlock` is updated with the ancestor loop node (belongs to `HashEachBlock`), the current predicate node and the inferred variables.

D_PadLastBlock checks successors of the obtained nodes and looks for predicates padding the last partial block. `IsNotFullBlock` compares the temporary buffer length with the block size and checks memory operations (`memcpy`/`memset`). Particularly, if the destination address starts from "`tmp_buf` + `tmp_buflenptr`" and the length equals to "`block_size` - `rem_msglen`", we consider the last partial block is padded.

Once the detection algorithm identifies the nodes representing above hashing steps, MinerRay further checks if the nodes are within a loop construct to decide the presence of a cryptominer.

5.4 User Consent Inference

We consider miners that inform users of crypto mining activity legitimate. If a detected miner does not seek consents from users, we say it is involved in a cryptojacking attack. To this end, we inspect a web page with miners and see if the user is informed or not. Specifically, we first look for simple strings like "mining", "CPU", etc. on the web page. To determine if the user permission is requested, we explore HTML user events and see if the miner instantiation can be triggered without user actions.

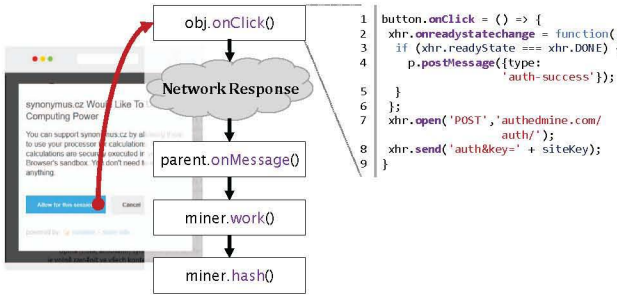


Figure 8: User Consent Call Graph on `synonymus.cz`

Figure 8 illustrates how `synonymus.cz` requests user consent using a pop-up message. The JavaScript snippet creates an `onClick` event handler for the "Allow for this session" button. When the button is clicked, the `onClick` handler will eventually triggers the miner instantiation, starts the WebWorker threads (`Miner.work()`), and starts mining by calling the `Miner.hash()`.

Without losing generality, we assume user consent is collected by clicking HTML elements. We focus on exploring invocations of the `onclick` events of HTML objects that may instantiate WebAssembly cryptominers. However, because the JavaScript snippets instantiating cryptominers are usually obfuscated (especially in malicious scenarios), static methods are unlikely to work.

Therefore, we develop a simple dynamic approach that explores the `onclick` events and checks if WebAssembly instantiation APIs such as `WebAssembly.instantiate` can be invoked. Although event explorations could be improved using the methods presented in [17, 52], we observed this simple approach is sufficient in practice, because, in part, most malicious websites instantiate WebAssembly miners without user interference.

6 EVALUATION

Implementation. We use WABT [16] to convert WebAssembly binary to its text format. We develop a parser in Node.js to construct control flow graphs. To infer user consent, we use *Esprima* [39] to instrument JavaScript and build the dynamic call graphs.

Dataset. MinerRay was evaluated on the top 1.2 million websites from the Alexa Top 10 Million Websites list. In total, we have crawled and investigated 1,246,074 websites. These websites were crawled starting from January 2019 to January 2020. For each website, we crawled the homepage and waited an additional 5 seconds after the page is fully loaded. We also visited all links on the homepage and crawled these sub-pages. We compiled the Chromium web browser with "`-dump-wasm-module`" flag to dump all WebAssembly binaries that it encounters.

6.1 Cryptominer Detection Results

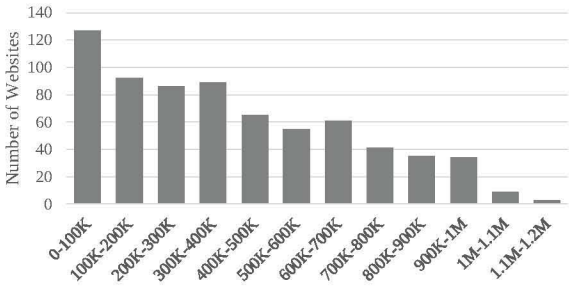


Figure 9: Miners by Alexa Rank.

Prevalence of Cryptominers. In total, MinerRay found cryptominers on 901 websites. Figure 9 shows the number of sites serving cryptominers. We observed the number of cryptominers decreases for lower ranked websites.

We investigate miner distributions on the landing pages or subpages. Among the 901 websites, we observed 560 miners on the landing pages and 341 on subpages. We manually check the purpose of the websites. Most of detected sites are torrent websites, movies and videos, etc., which is probably because users are likely to stay longer on these streaming websites for more profits.

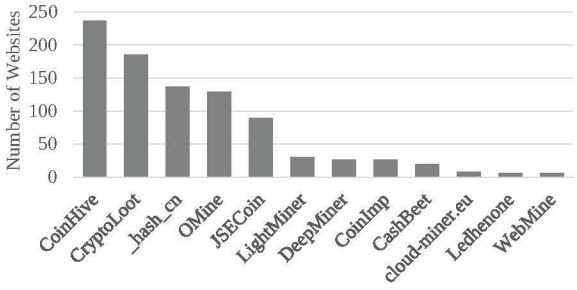


Figure 10: Miners by Cryptomining Services.

Cryptomining Services. We observed 12 unique mining services. Figure 10 shows the number of sites using each service. Note that the same site could have used multiple miners that connected to different mining services. As shown in the figure, CoinHive was the most popular mining service deployed on 237 websites, followed by CryptoLoot found on 186 websites.

6.2 Comparison Study

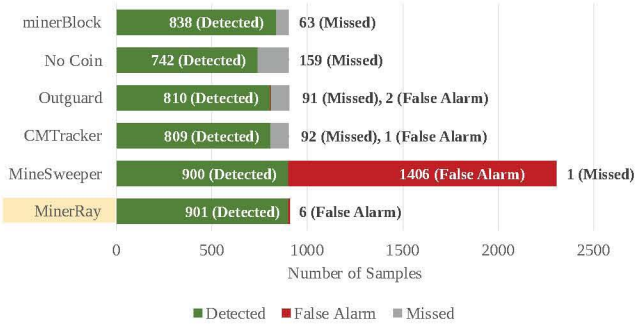


Figure 11: Comparison with State-of-the-Art Detectors.

We compare MinerRay with three state-of-the-art cryptomining detectors (MineSweeper [43], CMTracker [40] and Outguard [42]) and two signature-based browser extensions (No Coin [41] and minerBlock [28]).

Methodology. From our dataset, we scan 3,825 websites. Specifically, we run the three existing tools and MinerRay on the websites. To that end, we find that there are 2,306 websites that are detected by at least one detector. Among the 2,306 websites, we manually analyze each of them to verify whether the website conducts cryptomining or not.

Interpretation and Ground truth. As shown in Figure 11, there are three outcomes: "Detected", "False Alarm", and "Missed". "Detected" means that the websites detected by each tool are correctly identified as cryptominers. "False Alarm" represents the cases that are not cryptominers. We manually verified all false alarms and observed that they are typically from online games, cryptography programs, and compression utilities. "Missed" indicates the websites have cryptominers but are not detected. Note that we manually verified all 2,306 websites detected by at least one detector in our experiment, which forms the ground truth of our experiment.

Summary. As shown in Figure 11, MineSweeper and MinerRay detect the most samples correctly. Other techniques miss many websites containing malicious cryptominers (92, 91, 159, and 63 by CMTracker, Outguard, No Coin, and minerBlock, respectively). However, MineSweeper significantly suffers from false alarms. It detects 1,406 websites that turned out to be benign. The result shows that MinerRay outperforms existing techniques. It detects the most malicious websites, without causing a significant number of false alarms. Note that MinerRay raises 6 false alarms where it incorrectly identified non-hashing or benign WebAssembly code as miners, which are from a game and cryptographic library. Those cases follow similar patterns described in Section 5.3.1 (e.g., block-based data computations and copy operations). However, the data content and sources differ from real cryptominers. To handle the cases, we may need to implement data-flow analysis to understand more details regarding the data going through the computations. We leave this as our future work.

In the following paragraphs, we explain the detail of each detector and compare it with MinerRay.

First, MineSweeper detects CryptoNight-based miners by counting the number of bit operations to recognize cryptographic hash functions. The result shows that the number of bit operations is not a precise way to detect cryptominers, leading to a significant number of false alarms. Unlike MineSweeper, MinerRay leverages hashing function semantics that better capture the essence of the cryptomining behaviors.

Second, CMTracker detects cryptominers by calculating cumulative time spent on signature hash functions and profiling stack structures for repetitive behavioral patterns. If 10%+ of the execution time is spent on hash functions or a repeated call chain occupies 30%+ of the whole execution time, it is considered to be a miner. CMTracker produces one false alarm and misses 92 websites. The majority of these cases use hashing libraries built into the browser and implemented as native code. As a result, the profiling information collected by CMTracker may not contain expected signature hash functions or repetitive patterns. By contrast, MinerRay achieved better results because MinerRay focuses on hash function semantics and supports JS miner detection.

Third, Outguard uses machine learning techniques to detect cryptominers. It develops a set of features (such as the presences of WebAssembly and signature hash functions, and the numbers of web workers/WebSocket connections) from the dynamic trace collected by an instrumented web browser. Similar to all supervised learning techniques, the quality of the training data and the features considered are critical to its performance. Compared to MinerRay, Outguard misses 91 websites, which used the JSECoin service. The recent trend of diversified cryptomining algorithms and implementations would break Outguard's detection as the features (e.g., signature hash functions) are changed. As MinerRay does not rely on signature functions and other features (e.g., the use of WebAssembly), MinerRay detected these websites.

Fourth, No Coin and minerBlock are blacklist based techniques. No Coin uses a blacklist to block network requests matching the URL patterns on the list. minerBlock leverages both blacklist and script-scanning to look for potentially dangerous mining patterns, which makes it effective in detecting miners with code embedded into the JavaScript file. No Coin and minerBlock miss 159 and 63 websites, respectively. Websites that were missed by No Coin and minerBlock used several evasion techniques such as non-blacklisted URLs and function name minification to avoid function name matching. MinerRay was able to catch them as we focus on internal semantics instead of URLs and function names.

The results essentially show that cryptominers evolve fast, making the predefined features and blacklists outdated hence circumvented quickly.

6.3 User Consent Results

Out of 901 websites with cryptominers, we found only 16 websites informed users of the background cryptomining (Table 4). In particular, 13 (81%) started to mine automatically except for only 3 websites that ask for consent before starting the mining. Nearly half of them (7 out 16) do not offer a way to disable the miner. 15 do not allow users to limit CPU usage. 5 websites present unnoticeable text warnings at the bottom of the page. Additionally, the text messages generally describe that a cryptocurrency (e.g., XMR) will be mined, which is difficult for a non-technical user to understand.

Note that, if we consider those cryptominers that inform users as non-malicious, other cryptominer/cryptojacking detectors cause false alarms because of them. In particular, we find that the 16 websites MinerRay identified to inform users are all detected by existing detectors, leading to 16 additional false alarms for other detectors except for MinerRay.

Figure 12: Landing Page of *ikwilthepiratebay.org*

Example. The website *ikwilthepiratebay.org* (visited in January 2019 [6]), is a clone of The Pirate Bay and is one of the sixteen websites that informs users about cryptomining activities. As shown in Figure 12, it displays a text message stating that XMR will be mined. However, the text message is placed at the bottom of the page in a visibly-smaller text. The message uses the terms "XMR" and "mined", which do not clearly explain the process and should not be considered as a valid user consent request. Note that MinerRay can automatically identify the mining relevant UI components (e.g., messages), informing users explicitly (e.g., by increasing the message's font size).

6.4 Performance and Memory Overhead

Table 5 presents the average space and runtime overhead for scanning a sample program (e.g., JS and WebAssembly). The values are the average values observed during the evaluation. Specifically, the average file size (the second column) MinerRay processed during the evaluation is 447.39 KB. The third column shows the average memory overhead by MinerRay at runtime. It is 37.23 MB, which is fairly small and negligible in modern machines. The last two columns show the runtime overhead. To scan a program, MinerRay constructs both intra- and inter-procedure control flow graphs. The average time spent on the graph construction is 427.78 ms. Then, MinerRay takes 1,398.6 ms to scan the program to determine whether it is a malicious cryptominer or not. Overall, the average scanning time is less than 1.9 seconds (1,825 ms), which we believe reasonably fast, considering the average size of the input is 447.39 KB. We also observe that the performance and the file size have a linear relationship, indicating that MinerRay is scalable. Note that our prototype can be further optimized. We leave optimizing our implementation as our future work.

Table 4: 16 Websites That Inform User Cryptomining

Rank	Website	Purpose	Auto Opt-In[1]	Disable[2]	How to Disable	Throttle[3]	Format
12448	http://thepiratebay.se.net	Torrents	✓	×	–	×	Text at footer
21716	http://pirateiro.com	Torrents	✓	×	–	×	Text at top
54870	http://stevenuniver.se	TV Show	✓	✓	Button Click	×	Button at top
55980	http://ikwilthepiratebay.org	Torrents	✓	×	–	×	Text at footer
60744	http://filmclub.tv	TV & Movies	✓	×	–	×	Text at footer
80243	http://browsermine.com	Cryptomining	✓	✓	Slide to 0	✓	Slider at top
114592	http://pirateunblocker.com	Torrents	✓	×	–	×	Text at footer
120040	http://pirateunblocker.me	Torrents	✓	×	–	×	Text at footer
148450	http://synonymus.cz	Thesaurus	×	✓	Dismiss Popup	×	Popup
230354	http://dragonballzpolo.blogspot.com	TV Show	✓	✓	Dismiss Popup	×	Popup
330224	http://artistic-nude-images.com	Adult	×	✓	Dismiss Popup	×	Popup
361840	http://whatwouldyoudoif.se	Game	✓	×	–	×	Banner at footer
445252	http://fluxustv.blogspot.com	TV	×	✓	Dismiss Popup	×	Popup
645235	http://dariodomi.de	Imagemap	✓	✓	Link Click	×	Text at top
652324	http://justhodl.blogspot.com	Blog	✓	✓	Click "x" in Sidebar	×	Text in sidebar
2079986	http://pokemongoinformer.com	Blog	✓	✓	Dismiss Popup	×	Popup

1: ✓ means Auto Opt-In, × represents No Auto Opt-In. 2: ✓ means it can be disabled, × means no mechanism to disable.
3: ✓ indicates the cryptominer has a throttle mechanism, × means no throttle mechanism employed.

Table 5: Runtime and Space Overhead.

	File Size	Total Memory	Time (ms)	
	(KB)	Usage (MB)	Graph Const.	Scanning
Average	447.39 KB	37.23 MB	427.78 ms	1398.60 ms

7 DISCUSSION

Evaluation Data Collection Methodology. The web crawlers we built visited the homepage of each website and waited an additional 5 seconds after the page is fully loaded. In addition, we visited all available links on the homepage and crawled these sub-pages to increase the chance of observing cryptominers. It is possible that cryptominers are on particular pages that the web crawlers did not visit or that may be triggered only under certain user navigation patterns or idle time. Even though testing a website with comprehensive coverage is orthogonal to our work, MinerRay may find more cryptominers if being incorporated with more effective web crawling techniques.

Compression Functions. We do not consider compression function signatures yet. A possible improvement may be achieved by integrating AI techniques on signatures.

Obfuscation Techniques. Attackers can leverage existing obfuscation methods [1, 7, 19, 25, 37, 44, 50] to mask cryptominers to avoid detection. Source code level obfuscation techniques such as inserting opaque predicates [50], adding bogus control flow [7, 19], changing variable names [1, 7], and injecting dummy code [44] do not hinder MinerRay's detection because they do not change the underlying semantics. However, obfuscation techniques that encode or encrypt the entire program code [8–10] may break MinerRay and other existing static methods. We leave better handling of obfuscations to our future work.

Generality of Our Technique. As we discussed in Section 5.3, almost all existing cryptographic hash functions can be described as being based on a block cipher, where the semantics are necessary for block cipher implementations. Thus, our technique is general. However, to support a new language other than WebAssembly or JavaScript, the IR would need to be defined on how the steps within the technique are commonly implemented in the new programming language. Depending on the complexity of the language, this may be challenging to create. Additionally, since other languages support many external libraries providing hashing functionality, it may increase the difficulty in creating a mapping to the IR that is complete.

Complementary to Existing Approaches. Existing detectors rely on signatures or runtime features and do not consider program semantics. We believe considering semantics could be complementary to existing techniques. In addition, existing tools require manual efforts to complete the user consent analysis, while MinerRay can infer user consents by analyzing the program.

8 RELATED WORK

URL Blacklisting. URL blacklisting tools rely on predefined blacklist of known cryptomining servers [34, 45, 49, 54]. Eskandari et al. [34] detected over 30,000 cryptomining websites by using strings such as "coinhive.min.js" to query the Censys dataset. Papadopoulos et al. [54] conducted a study on 107,511 cryptomining websites, aiming to compare the profitability of web-based cryptomining and advertising. The cryptomining websites were collected based on the blacklists from No Coin. Since December 2017, URL blacklisting has been incorporated into the Opera web browser [46]. This approach is easy to circumvent through file and code obfuscation.

CPU Usage Profiling. CPU Usage analysis looks for abnormally high CPU usage [43, 49, 61, 64] because miners are likely to cause a spike in CPU usage. Although greedy miners can be found, it's easy to circumvent by throttling [34]. In addition, benign CPU-intensive tasks, such as video processing or gaming, could trigger false alarms. Our approach is not restricted by the CPU usage patterns.

Behavioral Analysis. Several approaches used behavioral analysis to detect cryptominers. Papadopoulos et al. [54] capture several

metrics when visiting a website, such as memory consumption, CPU thread usage, and system temperature, to identify cryptominers. CMTracker [40] captures data based on a hashing-function profiler and a stack-structure profiler to identify miners. Rüth et al. [60] augment No Coin with dynamic analysis afterwords to identify mining sites that get past the static filter. Bijmans et al. [22] also use a modified Chromium build to capture WebAssembly modules and looking at the WebSocket messages passed in a large-scale study to identify cryptominers hidden in websites.

Machine Learning Techniques. Other approaches used machine learning classification techniques to identify cryptominers. Musch et al. [48] create n-gram sequences from WebAssembly binaries to use as features for classification. Outguard [42] collects several features from an instrumented browser to classify sites. RAPID [59] captures resource usage and API event data through instrumented HTML APIs to create features for an SVM classifier.

Code Analysis. Code analysis inspects program behaviors to see if the patterns of particular instructions match known cryptominers [43, 49, 61, 72]. For example, SEISMIC [72] detects WebAssembly cryptominers by counting the number of executed arithmetic operations. MineSweeper [43] detects CryptoNight-based miners by counting the number of bit operations and comparing the signatures observed in CryptoNight. However, operation count is not a robust feature. As our comparison results shown in Section 6.2, MineSweeper falsely identified 1,327 benign websites as cryptojacking websites. By contrast, our approach focuses on the hash function semantics and is thus algorithm-agnostic.

Identifying Cryptographic Functions. Our work is also similar to program analysis techniques that identify cryptographic applications. Gröbert et al. [38] identify cryptographic primitives and keys in binary programs with dynamic binary program instrumentation. Aligot [24] detects obfuscated cryptographic primitives by comparing their input-output parameters. CryptoHunt [74] uses symbolic execution to identify cryptographic functions in obfuscated binary code. CloudRadar [75] collects features from hardware performance counters to identify the execution of cryptographic applications to detect side-channel attacks in cloud systems.

9 CONCLUSION

In this paper, we have presented MinerRay, an effective detection technique which automatically detects the probable existence of stealthy cryptominers on a website. Instead of focusing on particular URLs or signature functions, MinerRay identifies the presence of malicious cryptominers by inferring hash function semantics and reasoning about user consent of mining activity.

We have provided a systematic study of in-browser cryptominers on Alexa Top 1.2 Million websites. Our evaluation results show that MinerRay achieves high accuracy and is more effective than signature-based approaches in detecting stealthy cryptominers. In addition, we have studied the methods websites alerted users to mining activity and provided recommendations for better practices.

AVAILABILITY

The MinerRay source code and data can be found at https://miner-ray.github.io//.

REFERENCES

[1] 2012. JavaScript Minifier. https://javascript-minifier.com/.
[2] 2014. asm.js. https://asmjs.org.
[3] 2019. Coinhive: A Crypto Miner for your Website. https://coinhive.com.
[4] 2019. CoinMarketCap. https://coinmarketcap.com/.
[5] 2019. Ethereum Core Devs to Move Forward With ASIC-Resistant PoW Algorithm. https://cointelegraph.com/news/ethereum-core-devs-to-move-forward-with-asic-resistant-pow-algorithm.
[6] 2019. ikwilthepiratebay.org. https://web.archive.org/web/20190130093638/https://ikwilthepiratebay.org/
[7] 2019. JavaScript Obfuscator. https://obfuscator.io/.
[8] 2019. jjencode - Encode any JavaScript program using only symbols. http://utf-8.jp/public/jjencode.html.
[9] 2019. JS Obfuscator. https://www.cleancss.com/javascript-obfuscate/index.php.
[10] 2019. JSFuck - Write any JavaScript with 6 Characters: []()!+. http://www.jsfuck.com/.
[11] 2019. White Paper · webchain-network/wiki Wiki. https://github.com/webchain-network/wiki/wiki/White-Paper#the-protocol.
[12] 2020. Algorithms. https://xmrig.com/docs/algorithms.
[13] 2020. AssemblyScript/assemblyscript: Definitely not a TypeScript to WebAssembly compiler. https://github.com/AssemblyScript/assemblyscript.
[14] 2020. CoinIMP 0% fee JavaScript Mining, Browser Mining, Browser Miner. https://www.coinimp.com/.
[15] 2020. MinerRay Source Code | miner-ray.github.io. https://miner-ray.github.io/.
[16] 2020. The WebAssembly Binary Toolkit. https://github.com/WebAssembly/wabt.
[17] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. 2011. A Framework for Automated Testing of Javascript Web Applications. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. ACM, New York, NY, USA, 571–580. https://doi.org/10.1145/1985793.1985871
[18] Jean-Philippe Aumasson, Willi Meier, Raphael Phan, and Luca Henzen. 2014. *The Hash Function BLAKE*. https://doi.org/10.1007/978-3-662-44757-4
[19] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code Obfuscation Against Symbolic Execution Attacks. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications* (Los Angeles, California, USA) *(ACSAC '16)*. ACM, New York, NY, USA, 189–200. https://doi.org/10.1145/2991079.2991114
[20] Timo Bartkewitz. 2009. Building Hash Functions from Block Ciphers, Their Security and Implementation Properties.
[21] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. 2013. Keccak. In *Advances in Cryptology – EUROCRYPT 2013*, Thomas Johansson and Phong Q. Nguyen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 313–314.
[22] Hugo L.J. Bijmans, Tim M. Booij, and Christian Doerr. 2019. Inadvertently Making Cyber Criminals Rich: A Comprehensive Study of Cryptojacking Campaigns at Internet Scale. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1627–1644. https://www.usenix.org/conference/usenixsecurity19/presentation/bijmans
[23] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. 2017. Argon2: the memory-hard function for password hashing and other applications. https://www.cryptolux.org/images/0/0d/Argon2.pdf.
[24] Joan Calvet, José M. Fernandez, and Jean-Yves Marion. 2012. Aligot: Cryptographic Function Identification in Obfuscated Binary Programs. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (Raleigh, North Carolina, USA) *(CCS '12)*. ACM, New York, NY, USA, 169–182. https://doi.org/10.1145/2382196.2382217
[25] Haibo Chen, Liwei Yuan, Xi Wu, Binyu Zang, Bo Huang, and Pen-chung Yew. 2009. Control Flow Obfuscation with Information Flow Tracking. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, New York) *(MICRO 42)*. ACM, New York, NY, USA, 391–400. https://doi.org/10.1145/1669112.1669162
[26] Catalin Cimpanu. [n.d.]. Coinhive cryptojacking service to shut down in March 2019. https://www.zdnet.com/article/coinhive-cryptojacking-service-to-shut-down-in-march-2019/
[27] CryptoLoot. 2019. Update: uPlexa Re-Added, Monero to be Delisted. https://crypto-loot.org/news.
[28] CryptoMineDev. [n.d.]. minerBlock. https://github.com/xd4rker/MinerBlock.
[29] CryptoSlate. 2019. European Central Bank: Bitcoin isn't a threat, cryptocurrency not a new asset class. https://cryptoslate.com/european-central-bank-bitcoin-isnt-a-threat-cryptocurrency-not-a-new-asset-class/.
[30] The Lethean developers. 2019. Lethean Cryptocurrency. https://lethean.io/.
[31] Morris Dworkin. 2017. Block Cipher Techniques. https://csrc.nist.gov/projects/block-cipher-techniques
[32] Morris J. Dworkin, Elaine B. Barker, James R. Nechvatal, James Foti, Lawrence E. Bassham, E. Roback, and James F. Dray Jr. 2001. Announcing the ADVANCED ENCRYPTION STANDARD (AES).
[33] JEFF EDWARDS. 2018. How To Detect And Stop Cryptomining On Your Network.

https://blog.ipswitch.com/how-to-detect-and-stop-cryptojacking-on-your-network.
[34] Shayan Eskandari, Andreas Leoutsarakos, Troy Mursch, and Jeremy Clark. 2018. A first look at browser-based Cryptojacking. *CoRR* abs/1803.02887 (2018). arXiv:1803.02887 http://arxiv.org/abs/1803.02887
[35] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. 2009. The Skein Hask Function Family.
[36] Praveen Gauravaram, Lars Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren Thomsen. 2008. Grøstl - a SHA-3 candidate. (09 2008).
[37] Jun Ge, Soma Chaudhuri, and Akhilesh Tyagi. 2005. Control Flow Based Obfuscation. In *Proceedings of the 5th ACM Workshop on Digital Rights Management* (Alexandria, VA, USA) *(DRM '05)*. ACM, New York, NY, USA, 83–92. https://doi.org/10.1145/1102546.1102561
[38] Felix Gröbert, Carsten Willems, and Thorsten Holz. 2011. Automated Identification of Cryptographic Primitives in Binary Programs. In *Recent Advances in Intrusion Detection*, Robin Sommer, Davide Balzarotti, and Gregor Maier (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 41–60.
[39] Ariya Hidayat. [n.d.]. ECMAScript parsing infrastructure for multipurpose analysis. http://esprima.org/.
[40] Geng Hong, Zhemin Yang, Sen Yang, Lei Zhang, Yuhong Nan, Zhibo Zhang, Min Yang, Yuan Zhang, Zhiyun Qian, and Haixin Duan. 2018. 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*. ACM, 1701–1713.
[41] Keraf. [n.d.]. No Coin - Block miners on the web! https://github.com/keraf/NoCoin.
[42] Amin Kharraz, Zane Ma, Paul Murley, Charles Lever, Joshua Mason, Andrew Miller, Nikita Borisov, Manos Antonakakis, and Michael Bailey. 2019. Outguard: Detecting In-Browser Covert Cryptocurrency Mining in the Wild. In *The World Wide Web Conference* (San Francisco, CA, USA) *(WWW '19)*. ACM, New York, NY, USA, 840–852. https://doi.org/10.1145/3308558.3313665
[43] Radhesh Krishnan Konoth, Emanuele Vineti, Veelasha Moonsamy, Martina Lindorfer, Christopher Kruegel, Herbert Bos, and Giovanni Vigna. 2018. 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* (Toronto, Canada) *(CCS '18)*. ACM, New York, NY, USA, 1714–1730. https://doi.org/10.1145/3243734.3243858
[44] Aniket Kulkarni and Ravindra Metta. 2014. A Code Obfuscation Framework Using Code Clones. In *Proceedings of the 22Nd International Conference on Program Comprehension* (Hyderabad, India) *(ICPC 2014)*. ACM, New York, NY, USA, 295–299. https://doi.org/10.1145/2597008.2597807
[45] Malwarebytes. [n.d.]. Cryptojacking. https://www.malwarebytes.com/cryptojacking/report.
[46] Kornelia Mielczarczyk. 2017. Opera 50 Beta RC with Cryptocurrency Mining Protection. https://blogs.opera.com/desktop/2017/12/opera-50-beta-rc-cryptocurrency-mining-protection/.
[47] Troy Mursch. 2017. CRYPTOJACKING MALWARE COINHIVE FOUND ON 30,000+ WEBSITES. https://badpackets.net/cryptojacking-malware-coinhive-found-on-30000-websites/.
[48] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. [n.d.]. New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. ([n. d.]).
[49] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. 2018. Web-based Cryptojacking in the Wild. *CoRR* abs/1808.09474 (2018). arXiv:1808.09474 http://arxiv.org/abs/1808.09474
[50] Ginger Myles and Christian Collberg. 2006. Software Watermarking via Opaque Predicates: Implementation, Analysis, and Attacks. 6, 2 (April 2006), 155–171. https://doi.org/10.1007/s10660-006-6955-z
[51] Satoshi Nakamoto. 2009. Bitcoin: A Peer-to-Peer Electronic Cash System. *Cryptography Mailing list at https://metzdowd.com* (03 2009).
[52] Bao N. Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. 2014. GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering* 21, 1 (01 Mar 2014), 65–105. https://doi.org/10.1007/s10515-013-0128-9
[53] Patrick Nohe. 2018. Cryptojacking up 8500% in Q4 2017. https://www.thesslstore.com/blog/cryptojacking-8500-q4-2017-symantec/.
[54] Panagiotis Papadopoulos, Panagiotis Ilia, and Evangelos P Markatos. 2018. Truth in web mining: Measuring the profitability and cost of cryptominers as a web monetization model. *arXiv preprint arXiv:1806.01994* (2018).
[55] Anupam Pattanayak. 2012. Revisiting Dedicated and Block Cipher based Hash Functions. *IACR Cryptology ePrint Archive* 2012 (2012), 322.
[56] BitcoinWiki project. 2018. CryptoNight-Heavy. https://en.bitcoinwiki.org/wiki/CryptoNight-Heavy.
[57] Sumokoin Project. 2019. Sumokoin - Digital Cash For Highly-Confidential Transactions. https://www.sumokoin.org/.
[58] The Monero Project. 2019. Monero: the Secure, Private, Untraceable Cryptocurrency. https://getmonero.org.
[59] Juan D Parra Rodriguez and Joachim Posegga. 2018. RAPID: Resource and API-Based Detection Against In-Browser Miners. In *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 313–326.
[60] Jan Rüth, Torsten Zimmermann, Konrad Wolsing, and Oliver Hohlfeld. 2018. Digging into browser-based crypto mining. In *Proceedings of the Internet Measurement Conference 2018*. ACM, 70–76.
[61] Muhammad Saad, Aminollah Khormali, and Aziz Mohaisen. 2018. End-to-End Analysis of In-Browser Cryptojacking. *CoRR* abs/1809.02152 (2018). arXiv:1809.02152 http://arxiv.org/abs/1809.02152
[62] Seigen, Max Jameson, Tuomo Nieminen, Neocortex, and Antonio M. Juarez. 2013. CryptoNight Hash Function. (2013). https://cryptonote.org/cns/cns008.txt
[63] Symantec. 2018. 2018 Internet Security Threat Report. https://www.symantec.com/security-center/threat-report.
[64] Rashid Tahir, Muhammad Huzaifa, Anupam Das, Mohammad Ahmad, Carl Gunter, Fareed Zaffar, Matthew Caesar, and Nikita Borisov. 2017. Mining on someone else's dime: Mitigating covert mining operations in clouds and enterprises. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 287–310.
[65] The JSE Team. 2019. JSECoin - JavaScript Embedded Cryptocurrency. https://jsecoin.com/.
[66] Trend-Micro. 2018. 2018 Midyear Security Roundup: Unseen Threats, Imminent Losses. https://documents.trendmicro.com/assets/rpt/rpt-2018-Midyear-Security-Roundup-unseen-threats-imminent-losses.pdf.
[67] uPlexa. 2019. uPlexa. https://github.com/uPlexa/uplexa.
[68] uPlexa. 2019. uPlexa: Incentivizing the mass compute power of IoT devices to form a means of anonymous blockchain payments. https://uplexa.com/.
[69] uPlexa Team. 2018. uPlexa Whitepaper. https://uplexa.com/content/uPlexa-Whitepaper-EN.pdf.
[70] Said Varlioglu, Bilal Gonen, M. Ozer, and Mehmet Bastug. 2020. Is Cryptojacking Dead After Coinhive Shutdown? 385–389. https://doi.org/10.1109/ICICT50521.2020.00068
[71] W3C. [n.d.]. WebAssembly. https://webassembly.org/.
[72] Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W Hamlen, and Shuang Hao. 2018. SEISMIC: SEcure in-lined script monitors for interrupting cryptojacks. In *European Symposium on Research in Computer Security*. Springer, 122–142.
[73] WebDollar. 2019. WebDollar - Currency of the Internet. https://webdollar.io/.
[74] D. Xu, J. Ming, and D. Wu. 2017. Cryptographic Function Detection in Obfuscated Binaries via Bit-Precise Symbolic Loop Mapping. In *2017 IEEE Symposium on Security and Privacy (SP)*. 921–937. https://doi.org/10.1109/SP.2017.56
[75] Tianwei Zhang, Yinqian Zhang, and Ruby Lee. 2016. CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds, Vol. 9854. 118–140. https://doi.org/10.1007/978-3-319-45719-2_6