SEISMIC: SEcure In-lined Script Monitors for Interrupting Cryptojacks

Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu,
Kevin W. Hamlen, and Shuang Hao

The University of Texas at Dallas
{wenhao.wang,benjamin.ferrell,xiaoyang.xu,hamlen,shao}@utdallas.edu

Abstract. A method of detecting and interrupting unauthorized, browser-based cryptomining is proposed, based on semantic signature-matching. The approach addresses a new wave of cryptojacking attacks, including XSS-assisted, web gadget-exploiting counterfeit mining. Evaluation shows that the approach is more robust than current static code analysis defenses, which are susceptible to code obfuscation attacks. An implementation based on in-lined reference monitoring offers a browser-agnostic deployment strategy that is applicable to average end-user systems without specialized hardware or operating systems.

Keywords: web security, WebAssembly, cryptomining, intrusion detection, in-lined reference monitors

1 Introduction

Cryptojacking—the unauthorized use of victim computing resources to mine and exfiltrate cryptocurrencies—has recently emerged as one of the fastest growing new web cybersecurity threats. Network-based cryptojacking attacks increased 600% in 2017, with manufacturing and financial services as the top two targeted industries, according to IBM X-Force [31]. Adguard reported a 31% surge in cryptojacking attacks in November 2017 alone [32]. The Smominru botnet is estimated to be earning its owners about $8,500 each week via unauthorized Monero[1] mining, or an estimated $2.8–3.6 million total as of January 2018 [19].

The relatively recent escalation of cryptojacking threats can be traced to several converging trends, including the emergence of new mining-facilitating technologies that make cryptojacking easier to realize, next-generation cryptocurrencies that are easier to mine and offer greater anonymity to criminals, and the rising value of cryptocurrencies [23]. Among the chiefs of these new technologies is WebAssembly (Wasm),[2] a new bytecode language for web browsers that affords faster and more efficient computation than previous web scripting languages, such as JavaScript (JS). By implementing cryptomining algorithms in Wasm, legitimate miners can make more efficient use of client computing resources to generate greater revenue, and attackers can covertly establish illicit

[1] https://cointelegraph.com/news/monero
[2] http://webassembly.org

mining operations on browsers around the world with only average hardware and computing resources, thereby achieving the mass deployment scales needed to make cryptojacking profitable. For this reason, a majority of in-browser coin miners currently use Wasm [35].

Unfortunately, this availability of transparent cryptomining deployment models is blurring distinctions between legitimate, legal cryptomining and illegitimate, illegal cryptojacking. For example, in 2015, New Jersey settled a lengthy lawsuit against cryptomining company Tidbit, in which they alleged that Tidbit's browser-based Bitcoin mining software (which was marketed to websites as a revenue-generation alternative to ads) constituted "access to computers ... without the computer owners' knowledge or consent" [36]. The definition and mechanism of such consent has therefore become a central issue in protecting users against cryptojacking attacks. For example, numerous top-visited web sites, including Showtime [28], YouTube [11], and The Pirate Bay [17], have come under fire within the past year for alleged cryptojacking attacks against their visitors. In each case, cryptocurrency-generation activities deemed consensual by site owners were not deemed consensual by users.

In order to provide end-users an enhanced capability to detect and consent to (or opt-out of) browser-based cryptomining activities, this paper investigates the feasibility of *semantic signature-matching* for robustly detecting the execution of browser-based cryptomining scripts implemented in Wasm. We find that top Wasm cryptominers exhibit recognizable computation signatures that differ substantially from other Wasm scripts, such as games. To leverage this distinction for consent purposes, we propose and implement SEcure In-lined Script Monitors for Interrupting Cryptojacks (SEISMIC). SEISMIC automatically modifies incoming Wasm binary programs so that they self-profile as they execute, detecting the echos of cryptomining activity. When cryptomining is detected, the instrumented script warns the user and prompts her to explicitly opt-out or opt-in. Opting out halts the script, whereas opting in continues the script without further profiling (allowing it to execute henceforth at full speed).

This semantic signature-matching approach is argued to be more robust than syntactic signature-matchers, such as n-gram detectors, which merely inspect untrusted scripts syntactically in an effort to identify those that might cryptomine when executed. Semantic approaches ignore program syntax in favor of monitoring program behavior, thereby evading many code obfuscation attacks that defeat static binary program analyses.

Instrumenting untrusted web scripts at the Wasm level also has the advantage of offering a browser-agnostic solution that generalizes across different Wasm virtual machine implementations. SEISMIC can therefore potentially be deployed as an in-browser plug-in, a proxy service, or a firewall-level script rewriter. Additional experiments on CPU-level instruction traces show that semantic signature-matching can also be effective for detection of non-Wasm cryptomining implementations, but only if suitable low-level instruction tracing facilities become more widely available on commercial processors.

To summarize, this paper makes the following contributions:

- We conduct an empirical analysis of the ecosystem of in-browser cryptocurrency mining and identify key security-relevant components, including Wasm.
- We introduce a new proof-of-concept attack that can hijack mining scripts and abuse client computing resources to gain cryptocurrency illicitly.
- We develop a novel Wasm in-line script monitoring system, SEISMIC, which instruments Wasm binaries with mining sensors. SEISMIC allows users to monitor and consent to cryptomining activities with acceptable overhead.
- We apply SEISMIC on five real-world mining Wasm scripts (four families) and seven non-mining scripts. Our results show that mining and non-mining computations exhibit significantly different behavioral patterns. We also develop a classification approach and achieve $\geq$ 98% accuracy to detect cryptomining activities.

The remainder of the paper is structured as follows. Sections 2 and 3 begin with an overview of technologies of rising importance in web cryptomining, and a survey of the cryptomining ecosystem, respectively. Section 4 presents a new cryptojacking attack that demonstrates how adversaries can bypass current security protections in this ecosystem to abuse end-user computing resources and illicitly mine cryptocurrencies. Section 5 introduces our defense strategy based on semantic signature-detection and in-lined reference monitoring, and Section 6 evaluates its effectiveness. Finally, Section 7 summarizes related work and Section 8 concludes.

2 Background

2.1 Monero

Monero (XMR) is a privacy-focused cryptocurrency launched in April 2014. The confidentiality and untraceability of its transactions make Monero particularly popular on darknet markets. Monero's mining process is egalitarian, affording both benign webmasters and malicious hackers new funding avenues.

The core of Monero involves the CryptoNight proof-of-work hash algorithm based on the CryptoNote protocol [48]. CryptoNight makes mining equally efficient on CPU and GPU, and restricts mining on ASIC. This property makes Monero mining particularly feasible on browsers. A majority of current browser-based cryptocurrency miners target CryptoNight, and miner web script development has become an emerging business model. Page publishers embed these miners into their content as an alternative or supplement to ad revenue.

2.2 WebAssembly

Wasm [14] is a new bytecode scripting language that is now supported by all major browsers [7]. It runs in a sandbox after bytecode verification, where it aims to execute nearly as fast as native machine code.

Wasm complements and runs alongside JS. JS loads Wasm scripts, whereupon the two languages share memory and call each other's functions. Wasm is typically compiled from high-level languages (e.g., C, C++, or Rust). The most popular

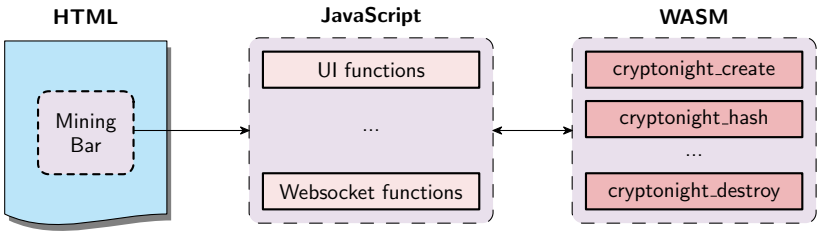

Fig. 1: Browser-based mining workflow

toolchain is Emscripten,[3] which compiles C/C++ to a combination of Wasm, JS glue code, and HTML. The JS glue code loads and runs the Wasm module.

Browsers can achieve near-native speeds for Wasm because it is designed to facilitate fast fetching, decoding, JIT-compilation, and optimization of Wasm bytecode instructions relative to JS. Wasm does not require reoptimization or garbage collection. These performance advantages make Wasm attractive for computationally intensive tasks, leading most browser-based cryptocurrency miners to use Wasm.

3 Ecosystem of Browser-based Cryptocurrency Mining

Although cryptomining is technically possible on nearly any browser with scripting support, efficient and profitable mining with today's browsers requires large-scale deployment across many CPUs. Webmasters offering services that attract sufficient numbers of visitors are therefore beginning to adopt cryptomining as an alternative or supplement to online ads as a source of revenue. This has spawned a secondary business model of cryptomining web software development, which markets mining implementations and services to webmasters.

Thus, although mining occurs on visitors' browsers, miner developers and page publishers play driving roles in the business model. As more miner developers release mining libraries and more page publishers adopt them, a browser-based cryptocurrency mining ecosystem forms. To better understand the ecosystem, we here illustrate technical details of browser-based mining.

Page publishers first register accounts with miner developers. Registration grants the publisher an asymmetric key pair. Publishers then download miner code from the miner developer and customize it to fit their published pages, including adding their public keys. The miner developer uses the public key to attribute mining contributions and deliver payouts to page publishers.

Figure 1 illustrates the resulting workflow. After publishers embed the customized miner into their pages, it is served to client visitors and executes in their browsers. The HTML file first loads into the client browser, causing the mining bar to trigger supporting JS modules, which share functionalities with Wasm modules. The Wasm code conducts computationally intensive tasks (e.g., `cryptonight_hash`), whereas UI and I/O interactions (e.g., Websocket communications) are implemented in JS. The code framework is typically created and maintained by miner developers.

[3] `http://kripken.github.io/emscripten-site`

Table 1: Security-related features of popular miners

	Wasm	Domain Whitelisting	Opt-In	CPU Throttle
Adless	✓	✗	✗	✓
Coinhive	✓	✗	✓	✓
CoinImp	✓	✗	✗	✓
Crypto-Loot	✓	✗	✗	✓
JSECoin	✓	✓	✗	✓
WebMinePool	✓	✗	✗	✓

Table 1 summarizes security-related features of top web miner products:

- *Wasm:* Most miners use Wasm for performance. For example, Coinhive mines Monera via Wasm, and has about 65% of the speed of a native miner.[4]
- *Domain Whitelisting:* To help deter malicious mining, some miner developers offer domain name whitelisting to webmasters. If miner developers receive mining contributions from unlisted domains, they can withhold payouts.
- *Opt-In Tokens:* To support ad blockers and antivirus vendors, some miner products generate opt-in tokens for browsers. Mining can only start after an explicit opt-in from the browser user. The opt-in token is only valid for the current browser session and domain.
- *CPU Throttling:* Using all the client's computing power tends to draw complaints from visitors. Miner developers therefore advise page publishers to use only a fraction of each visitor's available computing power for mining. Webmasters can configure this fraction.

4 Counterfeit Mining Attacks

To underscore the dangers posed by many browser-based mining architectures, and to motivate our defense, we next demonstrate how the ecosystem described in §3 can be compromised through *counterfeit mining*—a new cryptojacking attack wherein third-party adversaries hijack mining scripts to work on their behalf rather than for page publishers or page recipients.

Our threat model for this attack assumes that miner developers, page publishers, and page recipients are all non-malicious and comply with all rules of the cryptomining ecosystem in §3, and that mining scripts can have an unlimited variety of syntactic implementations. Specifically, we assume that miner developers and webmasters agree on a fair payout rate, publishers notify visitors that pages contain miners, and mining only proceeds with visitor consent. Despite this compliance, we demonstrate that malicious third-parties can compromise the ecosystem by abusing the miner software, insecure web page elements, and client computing resources to mine coins for themselves illegitimately.

To understand the attack procedure, we first illustrate how publishers embed miners into their web pages. Listing 1.1 shows the HTML code publishers must

[4] `https://coinhive.com`

```
1 <script src="https://authedmine.com/lib/simple-ui.min.js" async></script>
2 <div class="coinhive-miner"
3     style="width:256px;height:310px"
4     data_key="YOUR_SITE_KEY">
5     <em>Loading...</em>
6 </div>
```

Listing 1.1: Embedded miner HTML code

```
1 var elements = document.querySelectorAll('.coinhive-miner');
2 for (var i = 0; i < elements.length; i++) {
3     new Miner(elements[i])
4 }
```

Listing 1.2: JavaScript gadget

typically add. Line 1 imports the JS library maintained by miner developer. Line 3 specifies the dimensions of the miner rendered on the page. Line 4 identifies the publisher to the miner developer. To receive revenue, publishers must register accounts with miner developers, whereupon each publisher receives a unique data key. This allows miner developers to dispatch payroll to the correct publishers.

Our attack is predicated on two main observations about modern web pages: First, cross-site scripting (XSS) vulnerabilities are widely recognized as a significant and pervasive problem across a large percentage of all web sites [52,13]. Thus, we realistically assume that some mining pages contain XSS vulnerabilities. Second, although some XSS mitigations can block injection of executable scripts, they are frequently unsuccessful at preventing all injections of non-scripts (e.g., pure HTML). Our attack therefore performs purely HTML XSS injection to hijack miners via web gadgets [24]—a relatively new technique whereby existing, non-injected script code is misused to implement web attacks.

Examining the JS library called in line 1 reveals several potentially abusable gadgets, including the one shown in Listing 1.2. This code fragment selects all `div` elements of class `.coinhive-miner` on the page, and renders a miner within each. Unfortunately, line 1 is exploitable because it cannot distinguish publisher-provided `div` elements from maliciously injected ones. This allows an adversary to maliciously inject a `div` element of that class but with a different data key, causing the recipient to mine coins for the attacker instead of the publisher. We emphasize that in this attack, the exploited gadget is within the miner software, not within the publisher's page. Therefore *all web pages that load the miner* are potentially vulnerable, creating a relatively broad surface for criminals to attack.

To verify our counterfeit miner attack, we deploy two proof-of-concept attacks. Since the attacks begin with XSS exploits, we give two demonstrations: one using a reflected XSS vulnerability and one with a stored XSS vulnerability. The reflected XSS attack crafts a URL link containing the injected HTML code, where the injected code is a `div` element similar to Listing 1.1. After enticing visitors to click the URL link (e.g., via phishing), the visitor's browser loads and executes

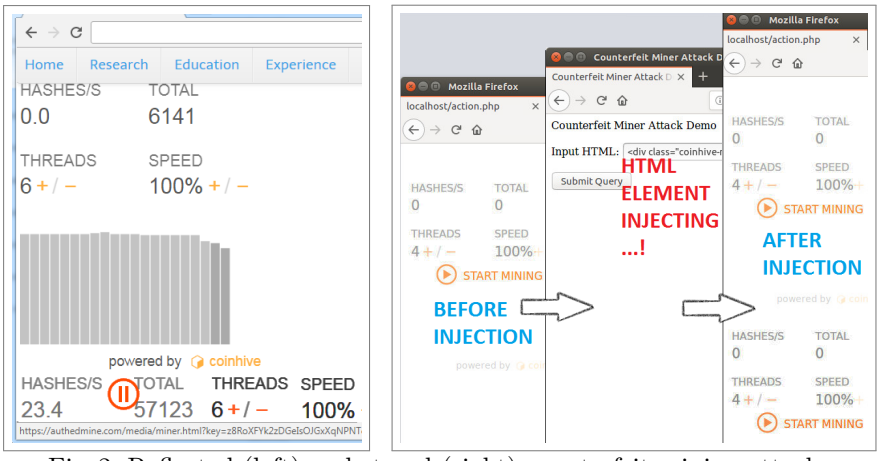

Fig. 2: Reflected (left) and stored (right) counterfeit mining attacks

the counterfeit miner. The left of Figure 2 shows a snapshot of the infected page, in which the counterfeit miner is visible at the bottom.

The stored XSS attack involves a page that reads its content from a database, to which visitors can add insufficiently sanitized HTML elements. In this scenario, injecting the malicious miner HTML code into the database causes the counterfeit miner to permanently inhabit the victim page. The right of Figure 2 illustrates the attack procedure. The three screenshots show sequential phases of the attack.

Counterfeit mining attacks illustrate some of the complexities of the cryptomining consent problem. In this case, asking users to consent to mining in general on affected web pages does not distinguish between the multiple miners on the compromised pages, some of which are working for the page publisher and others for a malicious adversary. The next section therefore proposes an automated, client-side consent mechanism based on in-lined reference monitoring that is per-script and is page- and miner-agnostic. This allows users to detect and potentially block cryptomining activities of individual scripts on a page, rather than merely the page as a whole.

5 Detection

In light of the dangers posed by counterfeit and other cryptomining attacks, this section proposes a robust defense strategy that empowers page recipients with a more powerful detection and consent mechanism. Since cryptojacking attacks ultimately target client computing resources, we adopt a strictly client-side defense architecture; supplementary publisher- and miner developer-side mitigations are outside our scope.

Section 5.1 begins with a survey of current static approaches and their limitations. Section 5.2 then proposes a more dynamic strategy that employs semantic signature detection, and presents experimental evidence of its potential effectiveness. Finally, Section 5.3 presents technical details of our defense implementation.

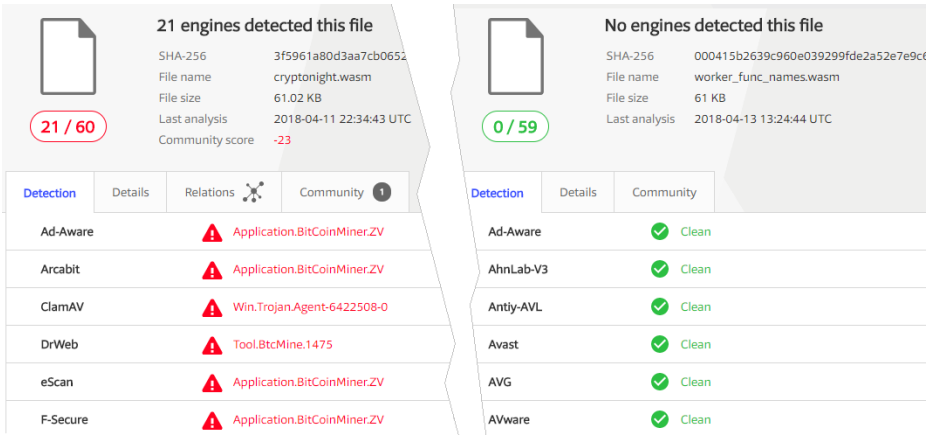

Fig. 3: Antivirus detection of CryptoNight before and after function renaming

5.1 Current Methods

Antivirus engines detect browser mining primarily via script file signature databases. The most popular Wasm implementation of the CryptoNight hashing algorithm [48] is flagged by at least 21 engines. A few of these (e.g., McAfee) go a step further and detect cryptomining implementations based on function names or other recognized keywords and code file structures.

Unfortunately, these static approaches are easily defeated by code obfuscations. For example, merely changing the function names in the CryptoNight Wasm binary bypasses all antivirus engines used on VirusTotal. Figure 3 shows detection results for the original vs. obfuscated CryptoNight binary.

Web browsers also have some detection mechanisms in the form of plugins or extensions, but these have similar limitations. The No Coin [21] Chrome extension enforces a URL blacklist, which prevents miners from contacting their proxies. However, criminals can bypass this by setting up new proxies not on the blacklist. MinerBlock[5] statically inspects scripts for code features indicative of mining. For instance, it detects CoinHive miners by searching for functions named `isRunning` and `stop`, and variables named `_siteKey`, `_newSiteKey`, and `_address`. These static analyses are likewise defeated by simple code obfuscations.

5.2 Semantic Signature-matching

A common limitation of the aforementioned detection approaches is their reliance on syntactic features (*viz.*, file bytes and URL names) that are easily obfuscated by attackers. We therefore focus on detection via semantic code features that are less easy to obfuscate because they are fundamental to the miner's computational purpose. Our proposed solution monitors Wasm scripts as they execute to derive a statistical model of known mining and non-mining behavior. Profiling reveals a distribution of Wasm instructions executed, which we use at runtime to distinguish mining from non-mining activity.

[5] https://github.com/xd4rker/MinerBlock

	i32.add	i32.and	i32.shl	i32.shr_u	i32.xor
A-Star	86.78	4.71	5.52	0.44	2.54
Asteroids	89.67	4.33	5.10	0.44	0.42
Basic4GL	75.78	8.43	13.75	1.78	0.27
Bullet(1000)	84.42	3.55	11.30	0.20	0.51
CoinHive	19.90	17.90	22.60	17.00	22.60
CoinHive_v0	20.20	17.50	22.70	17.00	22.70
CreaturePack	54.70	0.52	44.27	0.21	0.40
FunkyKarts	77.89	8.68	12.28	0.44	0.71
HushMiner	62.53	6.45	17.87	6.23	6.93
NFWebMiner	28.00	15.80	20.40	15.30	20.40
Tanks	61.90	12.29	22.27	2.02	1.51
YAZECMiner	57.99	4.37	30.75	3.26	3.63

Table 2: Execution trace average profiles

Using Intel Processor Tracing (PT), we first generated native code instruction counts for Wasm web apps. We recorded native instruction counts for 1-second computation slices on Firefox, for web apps drawn from: 500 pages randomly selected from Alexa top 50K, 500 video pages from YouTube, 100 Wasm embedded game or graphic pages, and 102 browser mining pages. Detailed results are presented in Appendix A. The traces reveal that cryptomining Wasm scripts rely much more upon packed arithmetic instructions from the MMX, SSE, and SSE2 instruction sets of CISC processors than do other Wasm scripts, like games.

Although PT is useful for identifying semantic features of possible interest, it is not a good basis for implementing detection on average client browsers since PT facilities are not yet widely available on average consumer hardware and OSes. We therefore manually identified the top five Wasm bytecode instructions that JIT-compile to the packed arithmetic native code instructions identified by the PT experiments. These five instructions are the column labels of Table 2.

We next profiled these top-five Wasm instructions at the Wasm bytecode level by instrumenting Wasm binary scripts with self-profiling code. We profiled four mining apps plus one variant, and seven non-mining apps. The non-mining apps are mostly games (which is the other most popular use of Wasm), and the rest are graphical benchmarks. For each app, we executed and interacted with them for approximately 500 real-time seconds to create each profile instance. For each app with configurable parameters, we varied them over their entire range of values to cover all possible cases.

Figure 4 displays the resulting distributions. There is a clear and distinct stratification for the two CoinHive variants and NFWebMiner, which are based on CryptoNight. YAZEC (Yet Another ZEC) Miner uses a different algorithm, and therefore exhibits slightly different but still distinctive profile. Table 2 displays an average across the 100 distributions for all of the profiled applications.

5.3 SEISMIC In-lined Reference Monitoring

Our profiling experiments indicate that Wasm cryptomining can potentially be detected by semantic signature-matching of Wasm bytecode instruction counts. To implement such a detection mechanism that is deployable on end-user browsers, our solution adopts an *in-lined reference monitor* (IRM) [39,8] approach. IRMs

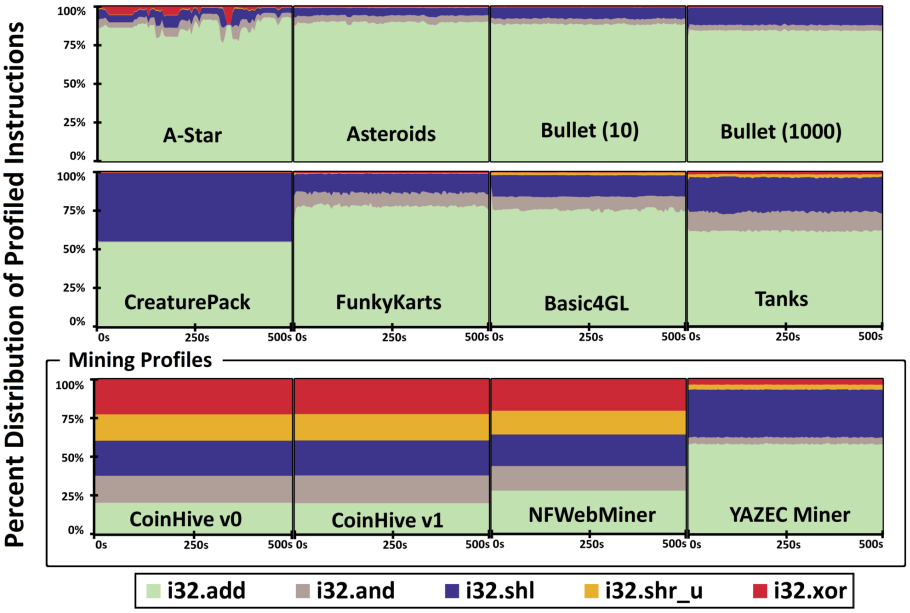

Fig. 4: Semantic profiles for mining vs. non-mining Wasm apps

automatically instrument untrusted programs (e.g., web scripts) with guard code that monitors security-relevant program operations. The code transformation yields a new program that self-enforces a desired policy, yet preserves policy-compliant behaviors of the original code. In browsing contexts, IRM formalisms have been leveraged to secure other scripting languages, such as JS and Flash (cf., [37]), but not yet Wasm. In this scenario, our goal is to design and implement an IRM system that automatically transforms incoming Wasm binaries to dynamically compute their own semantic features and match them to a given semantic signature.

Wasm scripts are expressed in binary or human-readable textual form. Each can be translated to the other using the Wasm Binary Toolkit (WABT). Typically scripts are distributed in binary form for size purposes, but either form is accepted by Wasm VMs. The programs are composed of *sections*, which are each lists of section-specific content. Our automated transformation modifies the following three Wasm section types:

- Functions: a list of all functions and their code bodies
- Globals: a list of variables visible to all functions sharing a thread
- Exports: a list of functions callable from JS

Figure 5 shows a high-level view of our Wasm instrumentation workflow. We here explain a workflow for a single Wasm binary file, but our procedure generalizes to pages with multiple binaries. As a running example, Listing 1.3 contains a small C++ function that computes the sum of the squares of its two inputs. Compiling it yields the Wasm bytecode in Listing 1.4.

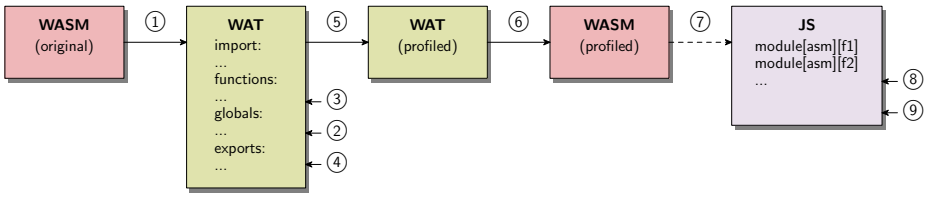

Fig. 5: SEISMIC transformation of Wasm binaries

```
1 int pythag(int a, int b) { return a * a + b * b; }
```

Listing 1.3: C++ source code for compilation to Wasm

```
1 (module (table 0 anyfunc) (memory $0 1)
2  (export "memory" (memory $0))
3  (export "pythag" (func $pythag))
4  (func $pythag (; 0 ;) (param $0 i32) (param $1 i32) (result i32)
5   (i32.add (i32.mul (get_local $1) (get_local $1))
6            (i32.mul (get_local $0) (get_local $0)))))
```

Listing 1.4: Original Wasm compiled from C++

Our prototype implementation of SEISMIC first parses the untrusted binary to a simplified abstract syntax tree (AST) similar to the one in Listing 1.4 using `wasm2wat` from WABT with the `–fold-exprs` flag (①). It next injects a fresh global variable of type i64 (64-bit integer) into the globals section for each Wasm instruction opcode to be profiled (②). The JS-Wasm interface currently does not support the transfer of 64-bit integers, so to allow JS code to read these counters, 32-bit accessor functions `getInstLo` and `getInstHi` are added (③). An additional `reset` function that resets all the profile counters to zero is also added, to allow the security monitor to separately profile different time slices of execution. All three functions are added to the binary's exports (④).

The transformation algorithm next scans the bodies of all Wasm functions in the script and in-lines counter-increment instructions immediately after each instruction to be profiled (⑤). Our prototype currently takes the brute-force approach of in-lining the counter-increment guard code for each profiled instruction, but optimizations that improve efficiency by speculatively increasing counters by large quantities in anticipation of an uninterruptable series of signature-relevant operations are obviously possible.

The modified Wasm text file is now ready to be translated to binary form, which we perform by passing it to `wat2wasm` from WABT (⑥). At this point, we redirect the JS code that loads the Wasm binary to load the new one (⑦). This can be done either by simply using the same name as the old file (i.e., overwriting it) or by modifying the load path for the Wasm file in JS to point to the new one.

Listing 1.5 shows the results of this process when profiling Wasm instructions `i32.add` and `i32.mul`. Lines 4–6 export the IRM helper functions defined in lines 15–17. Lines 18 and 19 define global counter variables to profile `i32.add` and

```
1  (module (table 0 anyfunc) (memory $0 1)
2   (export "memory" (memory $0))
3   (export "pythag" (func $pythag))
4   (export "__getAddsLo" (func $__getAddsLo))
5   ...
6   (export "__reset" (func $__reset))

8   (func $pythag (; 0 ;) (param $0 i32) (param $1 i32) (result i32)
9    (i32.add (set_global 0 (i64.add (get_global 0) (i64.const 1)))
10    (i32.mul (set_global 1 (i64.add (get_global 1) (i64.const 1)))
11     (get_local $1) (get_local $1))
12    (i32.mul (set_global 1 (i64.add (get_global 1) (i64.const 1)))
13     (get_local $0) (get_local $0))))

15  (func $__getAddsLo (; 1 ;) (result i32) (return (i32.wrap/i64 (get_global 0))))
16  ...
17  (func $__reset (; 5 ;) (set_global 0 (i64.const 0)) (set_global 1 (i64.const 0)))
18  (global (;0;) (mut i64) (i64.const 0))
19  (global (;1;) (mut i64) (i64.const 0)))
```

Listing 1.5: Instrumented Wasm

`i32.mul` instructions, respectively. The two `i32.mul` instructions are instrumented on lines 10 and 12, and the single `i32.add` instruction is instrumented on line 9.

SEISMIC's instrumentation procedure anticipates an attack model in which script authors and their scripts might be completely malicious, and adversaries might know all details of SEISMIC's implementation. For example, adversaries might craft Wasm binaries that anticipate the instrumentation procedure and attempt to defeat it. We therefore designed our instrumentation in accordance with secure IRM design principles established in the literature [39,15,29]. In particular, the Wasm bytecode language does not include unrestricted computed jump instructions, allowing our transformation to implement uncircumventable basic blocks that pair profiling code with the instructions they profile. Moreover, Wasm is type-safe [14], affording the implementation of incorruptible state variables that track the profiling information. Type-safety ensures that malicious Wasm authors cannot use pointer arithmetic or untyped references to corrupt the IRM's profiling variables (cf., [41,40]). These language properties are the basis for justifying other Wasm security features, such as control-flow integrity [50].

To start the enforcement, Listing 1.6 instantiates a JS timer that first executes at page-load and checks whether Wasm code has been loaded and compiled (⑧). If so, all Wasm instruction counters are queried, reset, and logged to the console. The timer profiles another slice of computation time every 5000 milliseconds. This affords detection of scripts that mine periodically but not continuously.

6 Evaluation

To evaluate our approach, we instrumented and profiled the web apps listed in Table 2. The majority of Wasm code we profiled was identifiable as having been

```
function wasmProfiler() {
 if (Module["asm"] != null && typeof _reset === "function") {
  console.log(_getAddsHi() * 2^32 + _getAddsLo() + " adds");
  console.log(_getMulsHi() * 2^32 + _getMulsLo() + " multiplies");
  _reset();
} else { console.log("Wasm not loaded yet"); }
 setTimeout(wasmProfiler, 5000);
}
wasmProfiler();
...
Module["asm"] = asm;
var _getAddsLo = Module["_getAddsLo"] = function() {
  return Module["asm"]["_getAddsLo"].apply(null, arguments) };
...
```

Listing 1.6: SEISMIC JavaScript code

Table 3: Mining Overhead

	Vanilla	**Profiled**
CoinHive v1	36 hash/s	18 hash/s
CoinHive v0	40 hash/s	19 hash/s
NFWebMiner	38 hash/s	16 hash/s
HushMiner	1.6 sol/s	0.8 sol/s
YAZECMiner	1.8 sol/s	0.9 sol/s

compiled with Emscripten, an LLVM-based JS compiler that yields a JS-Wasm pair of files for inclusion on web pages. The JS file contains an aliased list of exported functions, where we insert our new entries for the counters (⑨). The remaining Wasm programs we profiled have a similar structure to the output of Emscripten, so they can be modified in a similar manner.

We profiled every instruction used in the CoinHive worker Wasm, which is a variant of the CryptoNight hashing algorithm, and determined the top five bytecode instructions used: `i32.add`, `i32.and`, `i32.shl`, `i32.shr_u`, and `i32.xor`. Normalized counts of how many times these instructions execute constitute feature vectors for our approach.

Runtime Overhead. Table 3 reports runtime overheads for instrumented binaries. The data was obtained by running each miner in original and instrumented form over 100 trials, and averaging the results. CoinHive and NFWebMiner were set to execute with 4 threads and their units are in hashes per second. HushMiner and Yet Another ZEC Miner are single-threaded and display units in solutions per second. In general, the miners we tested incurred a runtime overhead of roughly 100%. We deem this acceptable because once mining is explicitly allowed by the user, execution can switch back to the faster original code.

Non-mining code overhead must be calculated in a different way, since most are interactive and non-terminating (e.g., games). We therefore measured overhead for these programs by monitoring their frames-per-second. In all cases they

Table 4: SVM stratified 10-fold cross validation

Miner	Fold	Precision	Recall	F_1	Fold	Precision	Recall	F_1
N	1	1.00	0.99	0.99	2	1.00	1.00	1.00
Y		0.96	1.00	0.98		1.00	1.00	1.00
N	3	1.00	1.00	1.00	4	1.00	1.00	1.00
Y		1.00	1.00	1.00		1.00	1.00	1.00
N	5	1.00	1.00	1.00	6	1.00	0.99	0.99
Y		1.00	1.00	1.00		0.96	1.00	0.98
N	7	1.00	1.00	1.00	8	1.00	1.00	1.00
Y		1.00	1.00	1.00		1.00	1.00	1.00
N	9	1.00	1.00	1.00	10	1.00	1.00	1.00
Y		1.00	1.00	1.00		1.00	1.00	1.00

remained at a constant 60 frames-per-second once all assets had loaded. Overall, no behavioral differences in instrumented scripts were observable during the experiments (except when mining scripts were interrupted to obtain user consent). This is expected since guard code in-lined by SEISMIC is implemented to be transparent to the rest of the script's computation.

Robustness. Our approach conceptualizes mining detection as a binary classification problem, where mining and non-mining are the two classes. Features are normalized vectors of the counts of the top five used Wasm instructions. For model selection, we choose Support Vector Machine (SVM) with linear kernel function. We set penalty parameter C to 10, since it is an unbalanced problem (there are far fewer mining instances than non-mining instances). To evaluate this approach, we use stratified 10-fold cross validation on 1900 instances, which consist of 500 miners and 1400 non-miners.

The results shown in Table 4 are promising. All mining activities are identified correctly, and the overall accuracy (F_1 score) is 98% or above in all cases. SEISMIC monitoring exhibits negligible false positive rate due to our strict threhhold for detection. Visitors can also manually exclude non-mining pages if our system exhibits a false positive, though the cross-validation results indicate such misclassifications are rare.

7 Related Work

Browser-based cryptocurrency mining is an emerging business model. A recent study has provided preliminary analysis of in-browser mining of cryptocurrencies [9], while we strive to inspect its security issues in depth. In particular, our work is the first to investigate the specific security ramifications of using Wasm for cryptomining.

7.1 Cryptocurrencies

Researchers have conducted a variety of systematic analyses of cryptocurrencies and discussed open research challenges [4]. A comprehensive study of Bitcoin

mining malware has shown that botnets generate additional revenue through mining [18]. MineGuard [47] utilizes hardware performance counters to generate signatures of cryptocurrency mining, which are then used to detect mining activities. Other research has focused on the payment part of cryptocurrencies. For example, EZC [1] was proposed to hide the transaction amounts and address balances. Double-spending attacks threaten fast payments in Bitcoin [20]. Bitcoin timestamp reliability has been improved to counter various attacks [46]. Through analysis of Bitcoin transactions of CryptoLocker, prior studies revealed the financial infrastructure of ransomware [27] and reported its economic impact [6]. In contrast, in-browser cryptomining, such as Monero, is less studied in the scholarly literature. In this work, we conducted the first analysis to study Wasm-based cryptomining, and developed new approaches to detect mining activities.

7.2 Cross-Site Scripting

Our counterfeit mining attack (§4) leverages cross-site scripting (XSS). The attacks and defenses of XSS have been an ongoing cat-and-mouse game for years. One straightforward defense is to validate and sanitize input on the server side, but this places a heavy burden on web developers for code correctness. XSS-GUARD [3] utilizes taint-tracking technology to centralize validation and sanitization on the server-side. Blueprint [30], Noncespaces [12], DSI [34], and CSP [42] adopt the notion of client-side HTML security policies [51] to defend XSS. Large-scale studies have also been undertaken to examine the prevalence of DOM-based XSS vulnerabilities [25] and the security history of the Web's client side [44], concluding that client-side XSS stagnates at a high level. To remedy the shortcomings of string-based comparison methods, taint-aware XSS filtering has been proposed to thwart DOM-based XSS [45]. DOMPurify [16] is an open-source library designed to sanitize HTML strings and document objects from DOM-based XSS attacks. Recently, attacks leveraging script gadgets have been discovered that circumvent all currently existing XSS mitigations [24]. We showed that in-browser crypomining is susceptible to such gadget-powered XSS attacks to hijack Wasm mining scripts.

Although our SEISMIC defense detects and warns users about cryptomining activities introduced through XSS, XSS can still potentially confuse users into responding inappropriately to the warnings. For example, attackers can potentially leverage XSS to obfuscate the provenance of cryptomining scripts, causing users to misattribute them to legitimate page publishers. This longstanding attribution problem is a continuing subject of ongoing study (cf., [38]).

7.3 Related Web Script Defenses

A cluster of research on defense mechanisms is also related to our work. Oblivi-Ad [2] is an online behavioral advertising system that aims to protect visitors' privacy. MadTracer [26] leverages decision tree models to detect malicious web advertisements. JStill [54] compares the information from both static analysis and runtime inspection to detect and prevent obfuscated malicious JS code. Analysis of access control mechanisms in the browser has observed that although CSP

is a clean solution in terms of access control, XS-search attacks can use timing side-channels to exfiltrate data from even prestigious services, such as Gmail and Bing [10]. Blacklist services provided by browsers to thwart malicious URLs have been shown to be similarly limited [49]. BridgeScope [55] was proposed to precisely and scalably find JS bridge vulnerabilities. Commix [43] automates the detection and exploitation of command injection vulnerabilities in web applications. Our system is orthogonal to these prior defense mechanisms, in that it profiles Wasm execution and helps users detect unauthorized in-browser mining of cryptocurrencies.

7.4 Semantic Malware Detection and Obfuscation

Our semantic signature-matching approach to cryptomining detection is motivated by the widespread belief that it is more difficult for adversaries to obfuscate semantic features than syntactic ones (cf., [5,22]). Prior work has demonstrated that semantic features can nevertheless be obfuscated with sufficient effort, at the cost of reduced performance (e.g., [33,53]). While such semantic obfuscations could potentially evade our SEISMIC monitors, we conjecture that the performance penalty of doing so could make obfuscated cryptojacking significantly less profitable for attackers. Future work should investigate this conjecture once semantically obfuscated cryptojacking attacks appear and can be studied.

8 Conclusion

SEISMIC offers a semantic-based cryptojacking detection mechanism for Wasm scripts that is more robust than current static detection defenses employed by antivirus products and browser plugins. By automatically instrumenting untrusted Wasm binaries in-flight with self-profiling code, SEISMIC-modified scripts dynamically detect mining computations and offer users explicit opportunities to consent. Page-publishers can respond to lack of consent through a JS interface, affording them opportunities to introduce ads or withdraw page content from unconsenting users. Experimental evaluation indicates that self-profiling overhead is unobservable for non-mining scripts, such as games (and is eliminated for miners once consent is granted). Robustness evaluation via cross-validation shows that the approach is highly accurate, exhibiting very few misclassifications.

Acknowledgments

This research was supported in part by NSF award #1513704, ONR award N00014-17-1-2995, AFOSR award FA9550-14-1-0173, and an NSF I/UCRC award from Lockheed-Martin.

References

1. E. Androulaki, G. Karame, and S. Capkun. Hiding transaction amounts and balances in Bitcoin. In *Proceedings of the 7th ACM International Conference on Trust and Trustworthy Computing (TRUST)*, pages 161–178, 2014.

2. M. Backes, A. Kate, and M. Maffei. ObliviAd: Provably secure and practical online behavioral advertising. In *Proceedings of the 33th IEEE Symposium on Security and Privacy (S&P)*, pages 257–271, 2012.
3. P. Bisht and V. Venkatakrishnan. XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 23–43, 2008.
4. J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten. SoK: Research perspectives and challenges for Bitcoin and cryptocurrencies. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, pages 104–121, 2015.
5. M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proceedings of the 26th IEEE Symposium on Security & Privacy (S&P)*, pages 32–46, 2005.
6. M. Conti, A. Gangwal, and S. Ruj. On the economic significance of ransomware campaigns: A Bitcoin transactions perspective. arXiv:1804.01341, 2018.
7. J. DeMocker. WebAssembly support now shipping in all major browsers. *Mozilla Blog*, November 2017.
8. Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop (NSPW)*, pages 87–95, 1999.
9. S. Eskandari, A. Leoutsarakos, T. Mursch, and J. Clark. A first look at browser-based cryptojacking. In *Proceedings of the 2nd IEEE Security & Privacy on the Blockchain Workshop (IEEE S&B)*, 2018.
10. N. Gelernter and A. Herzberg. Cross-site search attacks. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, pages 1394–1405, 2015.
11. D. Goodin. Now even YouTube serves ads with CPU-draining cryptocurrency miners. *Ars Technica*, January 2018.
12. M. V. Gundy and H. Chen. Noncespaces: Using randomization to defeat cross-site scripting attacks. *Computers & Security*, 31(4):612–628, 2012.
13. S. Gupta and B. Gupta. Cross-site scripting (XSS) attacks and defense mechanisms: Classification and state-of-the-art. *International Journal of System Assurance Engineering Management*, 8(1):512–530, 2017.
14. A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the Web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 185–200, 2017.
15. K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions On Programming Languages And Systems (TOPLAS)*, 28(1):175–205, 2006.
16. M. Heiderich and C. Späth. DOMPurify: Client-side protection against XSS and markup injection. In *Proceedings of the 22nd European Symposium on Research in Computer Security (ESORICS)*, pages 116–134, 2017.
17. J. Hruska. Browser-based mining malware found on Pirate Bay, other sites. *ExtremeTech*, September 2017.
18. D. Y. Huang, H. Dharmdasani, S. Meiklejohn, V. Dave, C. Grier, D. Mccoy, S. Savage, N. Weaver, A. C. Snoeren, and K. Levchenko. Botcoin: Monetizing stolen cycles. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS)*, 2014.

19. Kafeine. Smominru Monero mining botnet making millions for operators. *ProofPoint Threat Insight*, January 2018.
20. G. Karame, E. Androulaki, and S. Capkun. Double-spending fast payments in Bitcoin. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pages 906–917, 2012.
21. R. Keramidas. Stop coin mining in the browser with No Coin. `https://ker.af/stop-coin-mining-in-the-browser-with-no-coin`, September 2017.
22. J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting malicious code by model checking. In *Proceedings of the 2nd International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 174–187, 2005.
23. H. Lau. Browser-based cryptocurrency mining makes unexpected return from the dead. *Sympantec Threat Intelligence*, December 2017.
24. S. Lekies, K. Kotowicz, S. Groß, E. V. Nava, and M. Johns. Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, pages 1709–1723, 2017.
25. S. Lekies, B. Stock, and M. Johns. 25 million flows later: Large-scale detection of DOM-based XSS. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, pages 1193–1204, 2013.
26. Z. Li, K. Zhang, Y. Xie, F. Yu, and X. Wang. Knowing your enemy: Understanding and detecting malicious web advertising. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pages 906–917, 2012.
27. K. Liao, Z. Zhao, A. Doupé, and G.-J. Ahn. Behind closed doors: Measurement and analysis of CryptoLocker ransoms in Bitcoin. In *Proceedings of the 11th APWG Symposium on Electronic Crime Research (eCrime)*, pages 1–13, 2016.
28. S. Liao. Showtime websites secretly mined user CPU for cryptocurrency. *The Verge*, September 2017.
29. J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security (TISSEC)*, 12(3), 2009.
30. M. T. Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P)*, pages 331–346, 2009.
31. D. McMillen. Network attacks containing cryptocurrency CPU mining tools grow sixfold. *IBM X-Force SecurityIntelligence*, September 2017.
32. A. Meshkov. Cryptojacking surges in popularity growing by 31% over the past month. *AdGuard Research*, November 2017.
33. A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, pages 421–430, 2007.
34. Y. Nadji, P. Saxena, , and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS)*, 2014.
35. R. Neumann and A. Toro. In-browser mining: Coinhive and WebAssembly. *Forcepoint Security Labs*, April 2018. `https://blogs.forcepoint.com/security-labs/browser-mining-coinhive-and-webassembly`.
36. OAG, New Jersey. New Jersey Division of Consumer Affairs obtains settlement with developer of Bitcoin-mining software found to have accessed New Jersey computers without users' knowledge or consent. Office of the Attorney General, Department of Law & Public Safety, State of New Jersey, May 2015.

37. P. H. Phung, M. Monshizadeh, M. Sridhar, K. W. Hamlen, and V. Venkatakrishnan. Between worlds: Securing mixed JavaScript/ActionScript multi-party web content. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 12(4):443–457, 2015.
38. N. C. Rowe. The attribution of cyber warfare. In J. A. Green, editor, *Cyber Warfare: A multidisciplinary Analysis*, Routledge Studies in Conflict, Security and Technology. Routledge, 2015.
39. F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and Systems Security (TISSEC)*, 3(1):30–50, 2000.
40. M. Sridhar and K. W. Hamlen. ActionScript in-lined reference monitoring in Prolog. In *Proceedings of the 12th International Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 149–151, 2010.
41. M. Sridhar and K. W. Hamlen. Model-checking in-lined reference monitors. In *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 312–327, 2010.
42. S. Stamm, B. Sterne, and G. Markham. Reining in the Web with content security policy. In *Proceedings of the 19th International Conference on World Wide Web (WWW)*, pages 921–930, 2010.
43. A. Stasinopoulos, C. Ntantogian, and C. Xenakis. Commix: Automating evaluation and exploitation of command injection vulnerabilities in web applications. *International Journal of Information Security*, pages 1–24, 2018.
44. B. Stock, M. Johns, M. Steffens, and M. Backes. How the Web tangled itself: Uncovering the history of client-side Web (in)security. In *Proceedings of the 26th USENIX Security Symposium*, pages 971–987, 2017.
45. B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns. Precise client-side protection against DOM-based cross-site scripting. In *Proceedings of the 23rd USENIX Security Symposium*, pages 655–670, 2014.
46. P. Szalachowski. Towards more reliable Bitcoin timestamps. arXiv:1803.09028, 2018.
47. R. Tahir, M. Huzaifa, A. Das, M. Ahmad, C. Gunter, F. Zaffar, M. Caesar, and N. Borisov. Mining on someone else's dime: Mitigating covert mining operations in clouds and enterprises. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, pages 287–310, 2017.
48. N. van Saberhagen. CryptoNote v 2.0. Technical report, CryptoNote Technology, October 2013.
49. N. Virvilis, A. Mylonas, N. Tsalis, and D. Gritzalis. Security busters: Web browser security vs. suspicious sites. *Computers & Security*, 52:90–105, 2015.
50. WebAssembly Community Group. Security. `http://webassembly.org/docs/security`, 2018.
51. J. Weinberger, A. Barth, and D. Song. Towards client-side HTML security policies. In *Proceedings of the 6th USENIX Conference on Hot Topics in Security (HotSec)*, pages 8–8, 2011.
52. WhiteHat Security. Application security statistics report, vol. 12, 2017.
53. Z. Wu, S. Gianvecchio, M. Xie, and H. Wang. Mimimorphism: A new approach to binary code obfuscation. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pages 536–546, 2010.
54. W. Xu, F. Zhang, and S. Zhu. JStill: Mostly static detection of obfuscated malicious JavaScript code. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 117–128, 2013.

55. G. Yang, A. Mendoza, J. Zhang, and G. Gu. Precisely and scalably vetting JavaScript bridge in Android hybrid apps. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, pages 143–166, 2017.

A Processor Tracing Experiments

Table 5 itemizes the 30 most significant opcodes that discriminate between mining and non-mining web contents. Our feature selection process treats all distinct Intel native opcodes as features and their number of occurrences as feature values. We employ forests of trees to evaluate the importance of features. The ranking is based on the feature importance as determined by SVM (see §6).

Table 5: Top 30 native opcode features that distinguish mining from non-mining

Rank	Opcode	Rank	Opcode
1	SUB	16	LOCK
2	CMOVS	17	CMOVB
3	UNPCKHPS†	18	SETBE
4	DIVSD‡	19	SETNZ
5	SETB	20	ROL
6	MOVQ⋆	21	MUL
7	MAXPS†	22	SETNLE
8	CMOVNLE	23	CVTTSD2SI‡
9	COMVLE	24	MOVMSKPS†
10	PSUBUSW⋆	25	CMOVZ
11	CMOVNL	26	TEST
12	UNPCKLPS†	27	CMOVNZ
13	ROUNDSD†	28	ROUNDSS†
14	CMPPS†	29	STMXCSR†
15	MOVLHPS†	30	CMOVNB

⋆ MMX instruction † SSE instruction ‡ SSE2 instruction