Understanding Wasm Malware Detection and Evasion

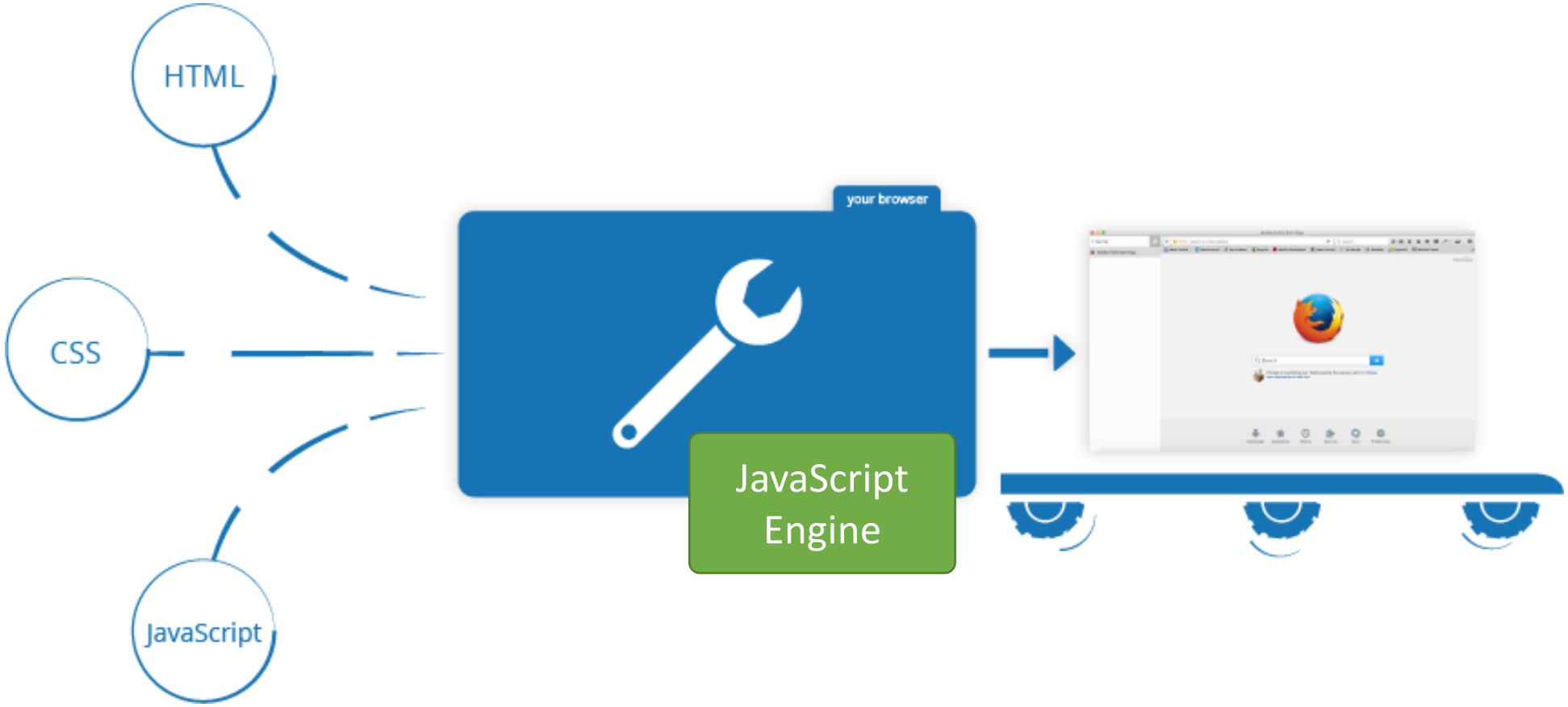
Hyoungshick Kim

Hyoungshick Kim (김형식)

Short Bio

- Professor, Department of Computer Science at Sungkyunkwan University
- Fellow at Sungkyunkwan University
- PC members at CCS, USENIX Security, ACSAC, ASIACSS, ESORICS, WWW, CHI, AAAI, WiSec
- Associate Editor in Chief, IEEE Transactions on Service Computings
- Court Expert Commissioner, Advisor for Samsung Electronics, LG Electronics, Dunamu
- Presented 31 papers at top-tier conferences (IEEE S&P, ACM CCS, USENIX Security, NDSS, etc.)
- Web page: https://seclab.skku.edu/people/hyoungshick-kim/

2013 ~ 현재	Sungkyunkwan University, Republic of Korea	Full Professor
2019 ~ 2020	CSIRO Data61, Australia	Distinguished Visiting Scientist
2012 ~ 2013	University of British Columbia, Canada	Postdoctoral Fellow
2008 ~ 2011	Computer Laboratory, University of Cambridge, UK	Ph.D Degree
2004 ~ 2008	Samsung Electronics	Senior Engineer



In every browser, code is interpreted and executed by a JavaScript engine.

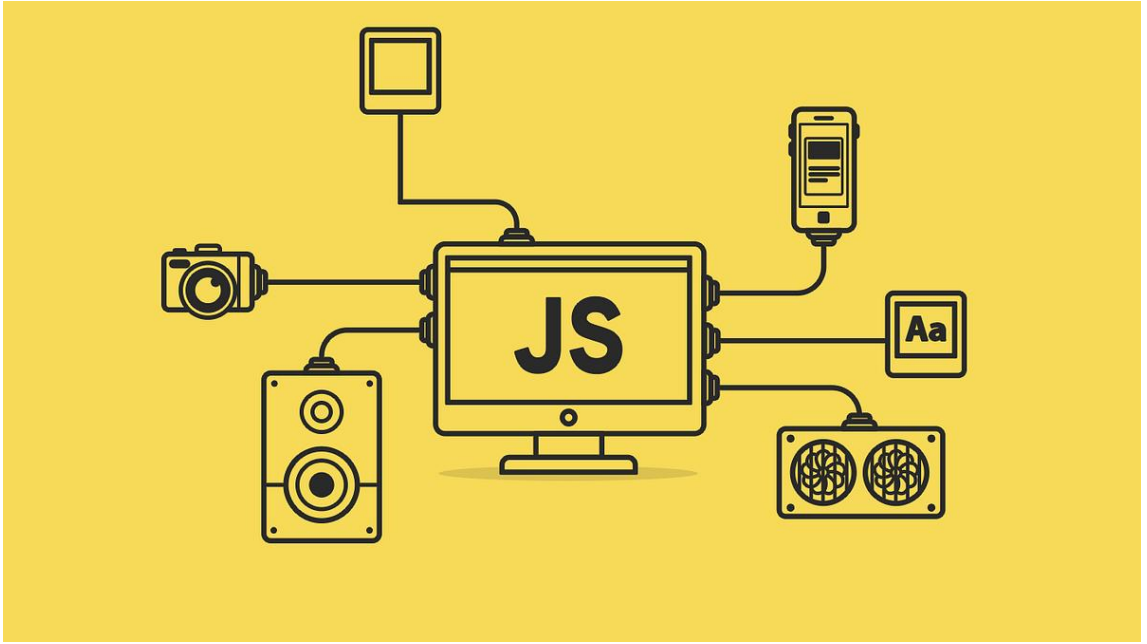
Web Script Language War

- Silverlight (Microsoft, 2007), ActionScript (Macromedia, 2006), or JavaScript (Sun, 1995) …

- Most were insecure
- Challenging to secure/control at runtime by browser
- Proprietary solutions - Never standardized

JavaScript

- Object oriented
- Interpreted
- Using for Web
- Dynamically typed language
- Low performance
  - ✓ It is hard to run applications (e.g., video games) requiring a high degree of performance



NaCl

- Google Native Client (2011)
- Designed to execute native code (x86/x64, ARM) in a sandbox
- Close to native performance
- 3D acceleration
- Debuggable using GDB-Remote
- Not a standard
- Google Chrome (Browser, OS) specific

구글 크롬, 게임플랫폼으로 진화한다

일반 | 입력 :2012/03/10 08:48 수정: 2012/03/10 09:05

임민철 기자 | 기자의 다른기사 보기

[이벤트] AI 인프라·자동화·플랫폼 전략을 한 번에! Red Hat Summit: Connect 2025 다시보기

구글은 크롬이 품은 네이티브클라이언트(NaCl) 기술을 미래 유망한 브라우저용 게임 엔진으로 제시했다. 회사가 이달초 미국 샌프란시스코 모스콘센터에서 열린 '게임개발자컨퍼런스(GDC)'를 통해 NaCl에 잠재된 게임 개발 플랫폼의 가능성을 펼쳐 눈길을 끈다.

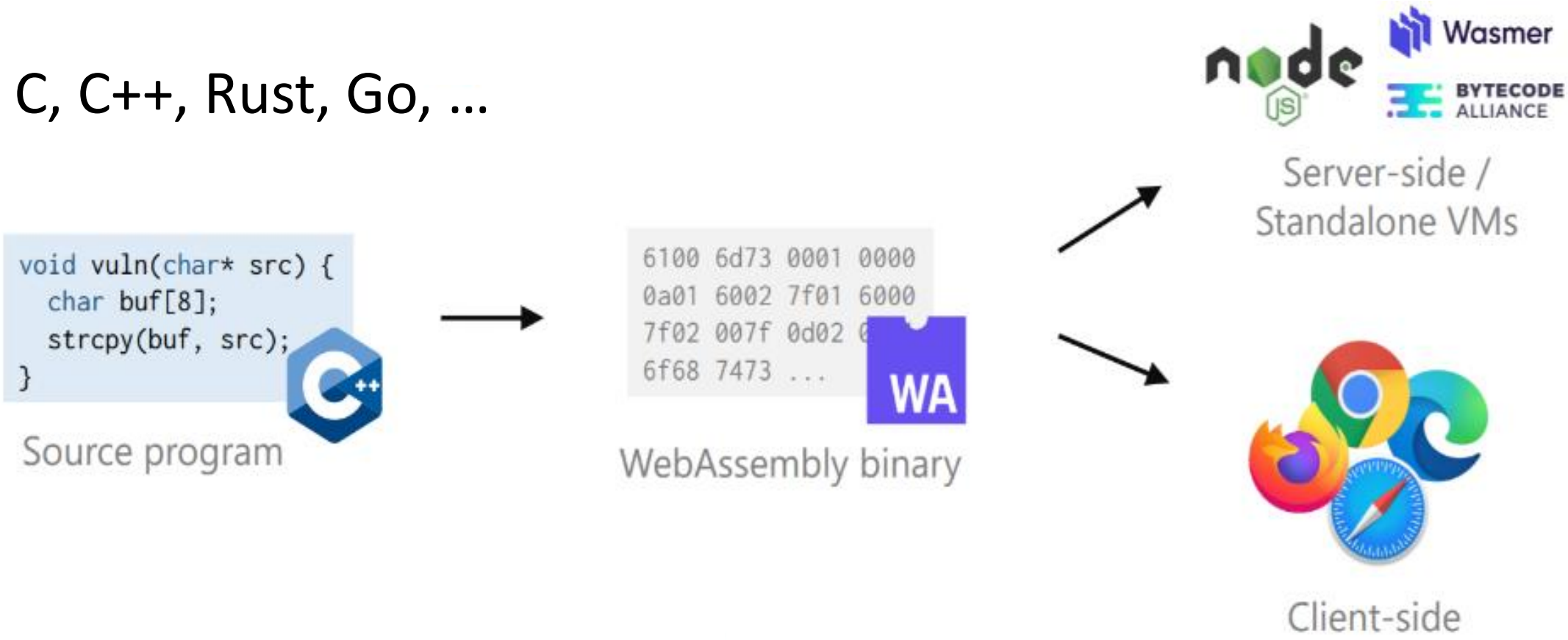
asm.js

- Proposed by Mozilla in 2013 as a **strict subset of JavaScript**
- Restricts JavaScript to a predictable, compiler-friendly form
- Uses explicit type coercions (e.g., int32/double) to reduce speculation and deoptimizations
- Represents memory as a linear buffer via TypedArrays for efficient loads/stores
- Avoids many dynamic JS features, enabling faster validation and more aggressive JIT/AOT-like optimization
- Can deliver up to ~2× speedups over conventional JavaScript

WebAssembly (Wasm)

- A new type of code that can be run in modern browsers
- The 4th language to run natively in browsers (following HTML, CSS, and JavaScript)
- Supported in Nods.js, standalone VMs
- Developed by a W3C working group in 2017
- Low-level binary format for a stack-based virtual machine
- The code size is significantly smaller than the JavaScript code
- The execution time of WASM binaries is just 20% slower than the execution of same native code (near-native performance)

C, C++, Rust, Go, …



Complied from C, C++, Rust, Go, …

Instruction Set

Control Flow instructions

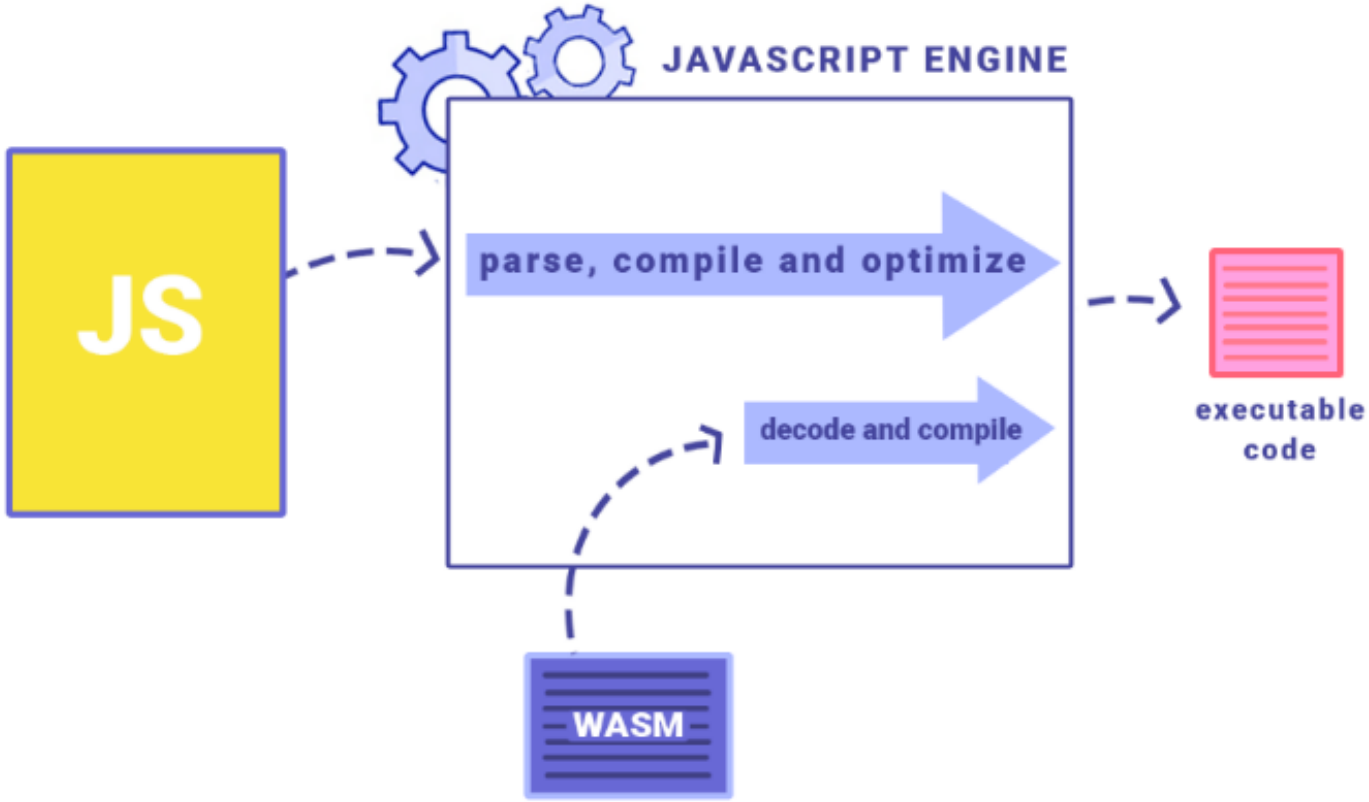
Mnemonic	Opcode	Description
unreachable	0x00	Trap execution
nop	0x01	NOP instruction
if <block> / else / end	0x04 / 0x05 / 0x06	Conditional branch
br / br_if / br_table	0x0c / 0x0d / 0x0e	BREAK from a block
call / call_indirect	0x10 / 0x11	Function call
return	0x0f	RETURN from function

Memory access instructions

Mnemonic	Opcode	Description
i32.load / i64.load	0x28 / 0x29	Load integer from memory
f32.load / f64.load	0x2a / 0x2b	Load float from memory
i32.store / i64.store	0x36 / 0x37	Store integer from memory
i32.store / i64.store	0x36 / 0x37	Store float from memory
current_memory	0x3f	Get the current memory size
grow_memory	0x40	Increase the memory size

Arithmetic and logic instructions

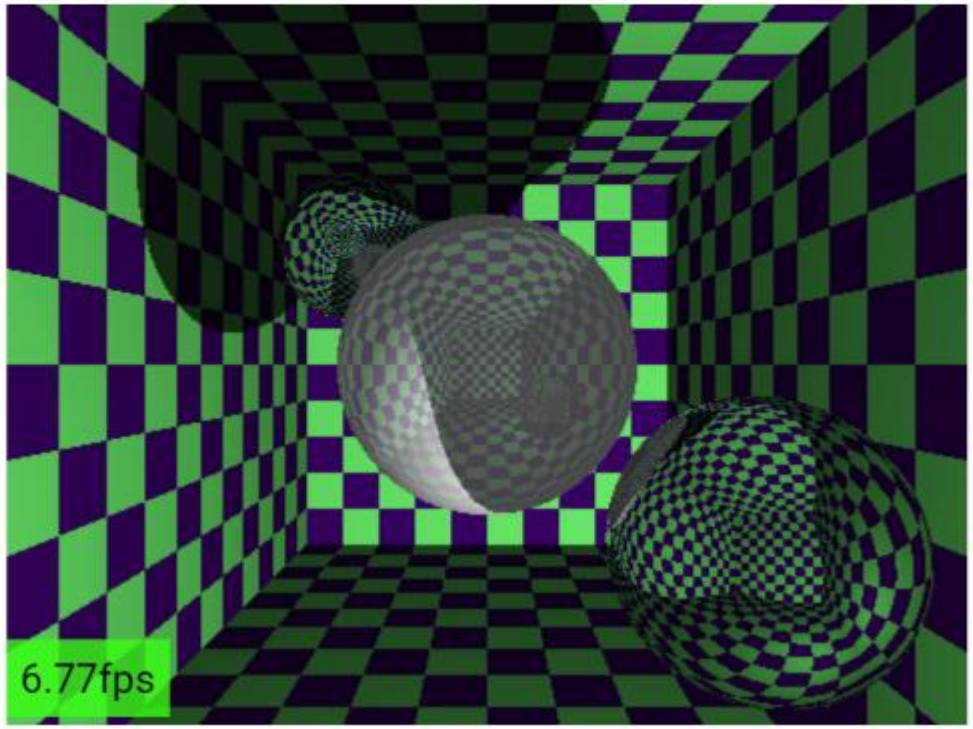
Mnemonic	Opcode	Description
i32.add / i32.sub / i32.mul	0x6a / 0x6b / 0x6c	Add / subtract / divide
i32.div_s / i32.div_u	0x6d / 0x6e	(Un)signed Division
i32.rem_s / i32.rem_u	0x6f / 0x70	(Un)signed Modulo
i32.and / i32.xor / i32.or / i32.shl	0x71 / 0x73 / 0x72 / 0x74	Logic operators



- We can load a WebAssembly module into a web application and call it from JavaScript
- It’s not a replacement for JavaScript, it works alongside JavaScript

Raytracing: WebAssembly vs JavaScript

Select renderer: WebAssembly | JavaScript



https://mtharrison.github.io/wasm-raytracer/


Google Earth

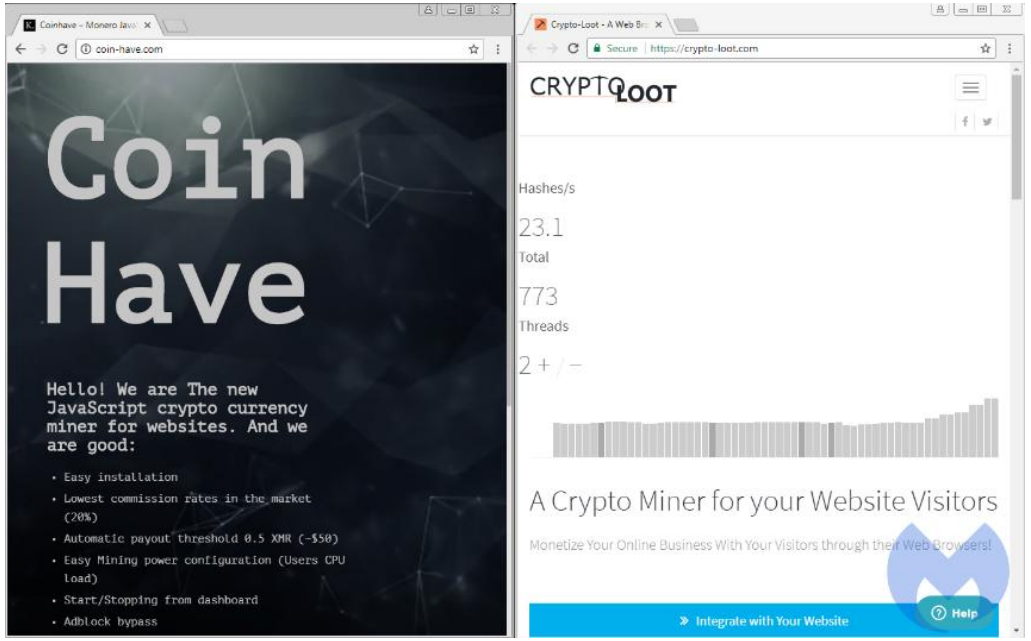
Cryptomining using WASM

- CryptoJacking
  - ✓ Unauthorized use of computing resources to mine cryptocurrencies
- CoinHive
  - ✓ Created in 2017
  - ✓ “Our miner uses WebAssembly and runs with about 65% of the performance of a native Miner.”
- Attackers just need to insert the following snippet of code on victim websites

```
<script src="https://coinhive.com/lib/coinhive.min.js"></script>
<script>
    var miner = new CoinHive.User('SITE_KEY', 'john-doe');
    miner.start();
</script>
```

Monero Cryptominer

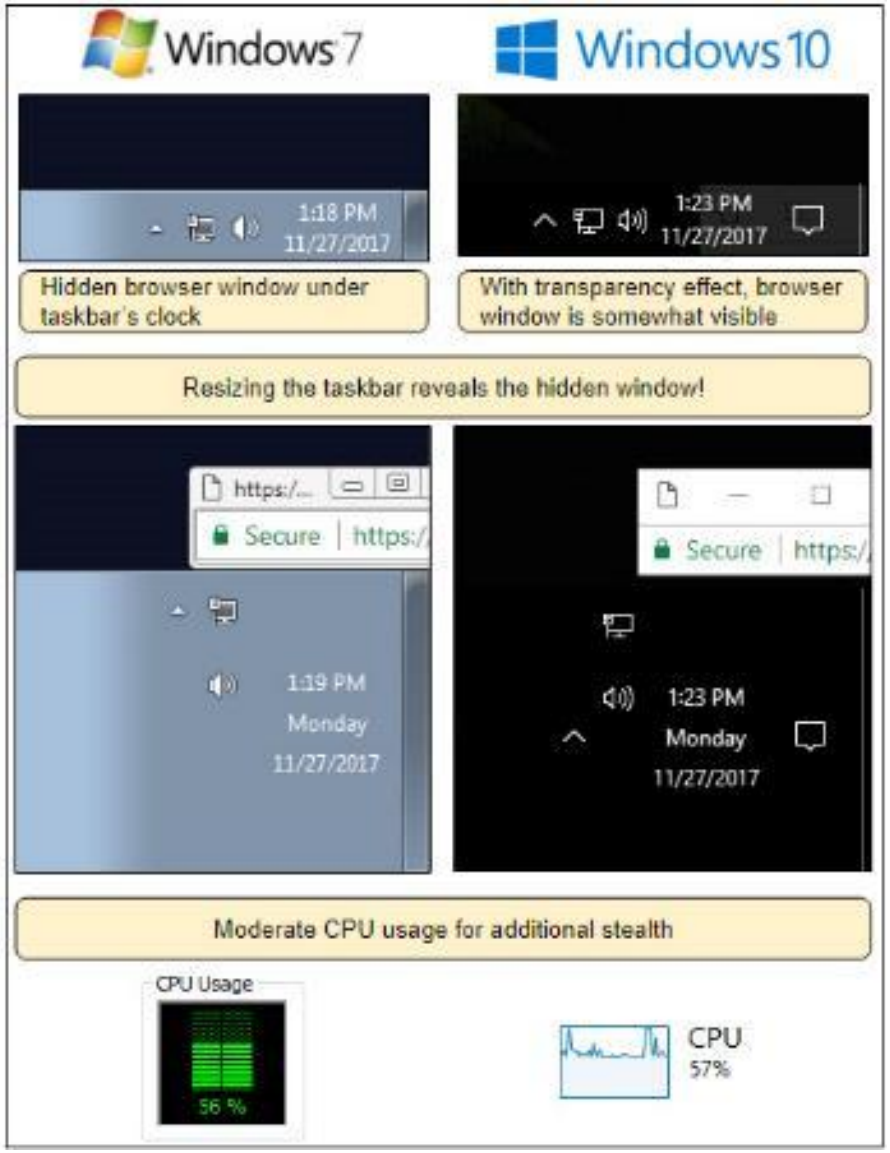
- Cryptonight PoW hash algorithm
- ASIC-resistant
- It uses computing resources to mine Monero Cryptocurrency



Stealthy in-browser cryptomining continues even after you close window

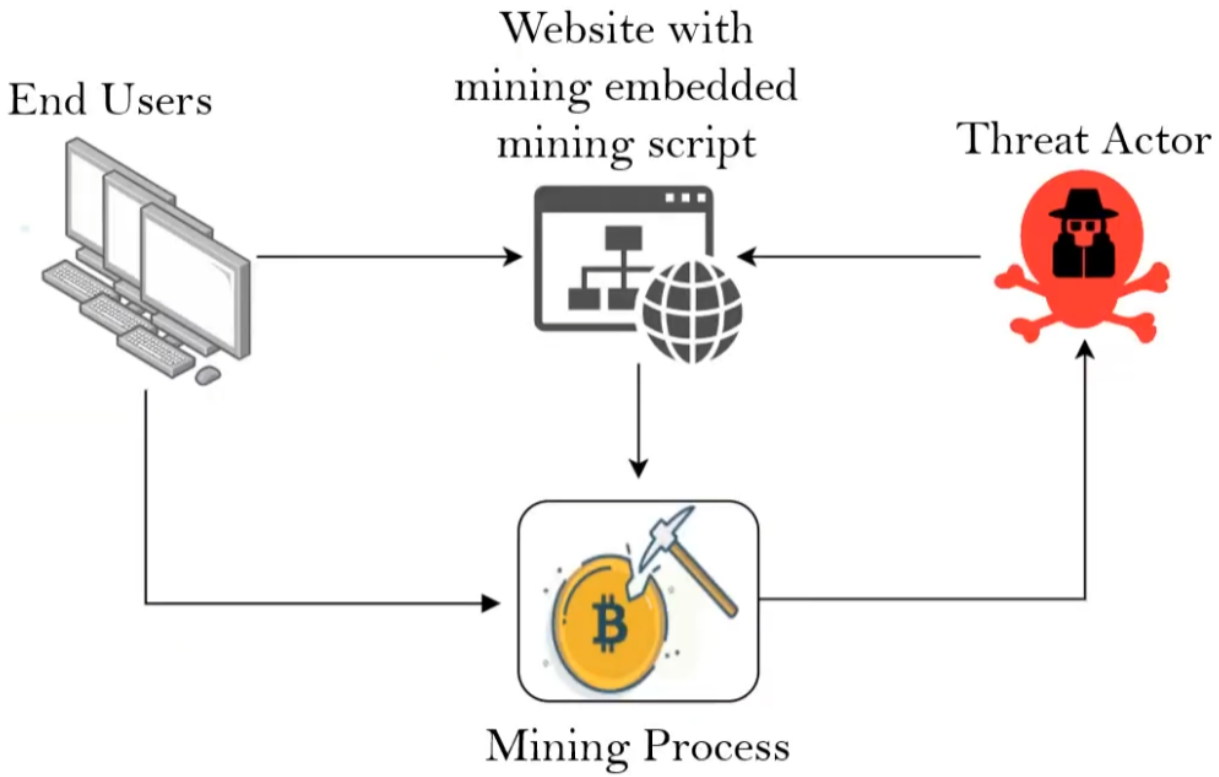
In-browser cryptocurrency mining is, in theory, a neat idea: make users' computers "mine" Monero for website owners so they don't have to bombard users with ads in order to earn money.

Unfortunately, in this far-from-ideal world of ours, mining scripts – first offered by Coinhive but soon after by other outfits – are mostly used by unscrupulous web admins and hackers silently compromising websites.



Cryptojacking Malware

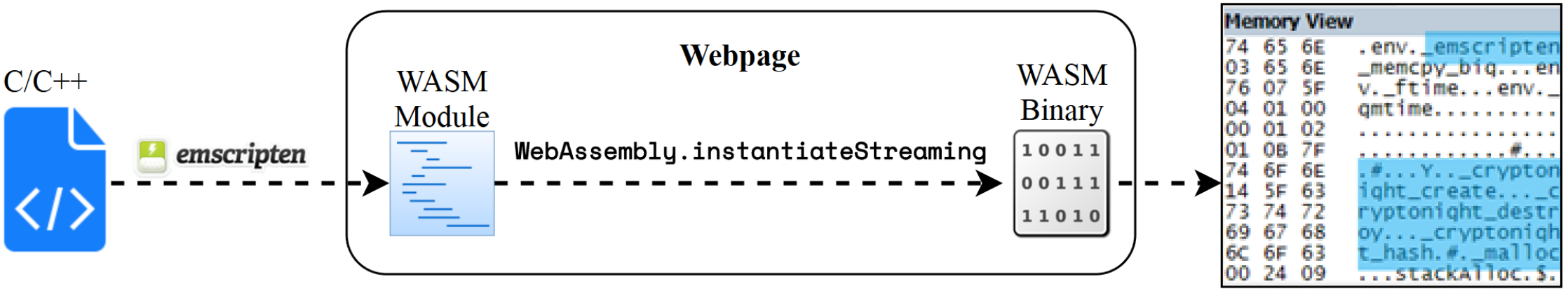
- Fileless malware
- Implemented in browsers
- Mine cryptocurrency without user's knowledge/consent



How can we detect cryptojacking malware?

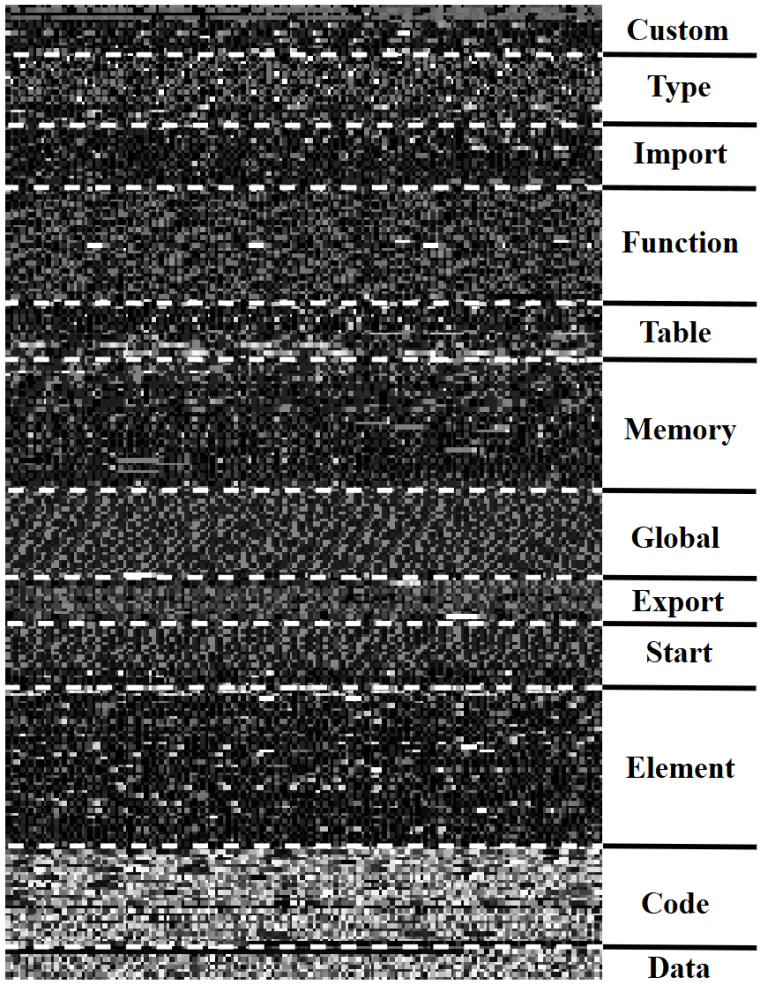
```
<script src="https://coinhive.com/lib/coinhive.min.js"></script>
<script>
    var miner = new CoinHive.User('SITE_KEY', 'john-doe');
    miner.start();
</script>
```

Why Wasm Malware?

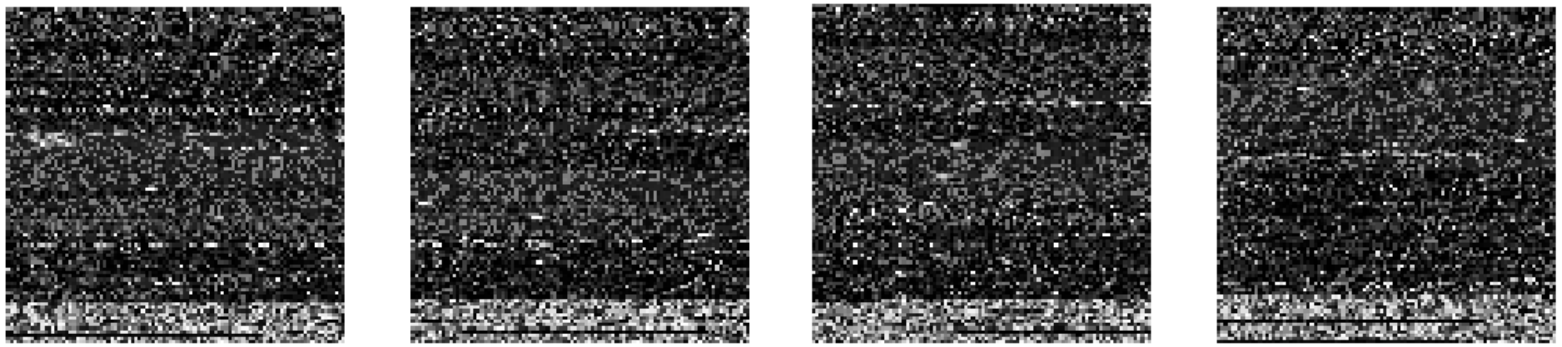


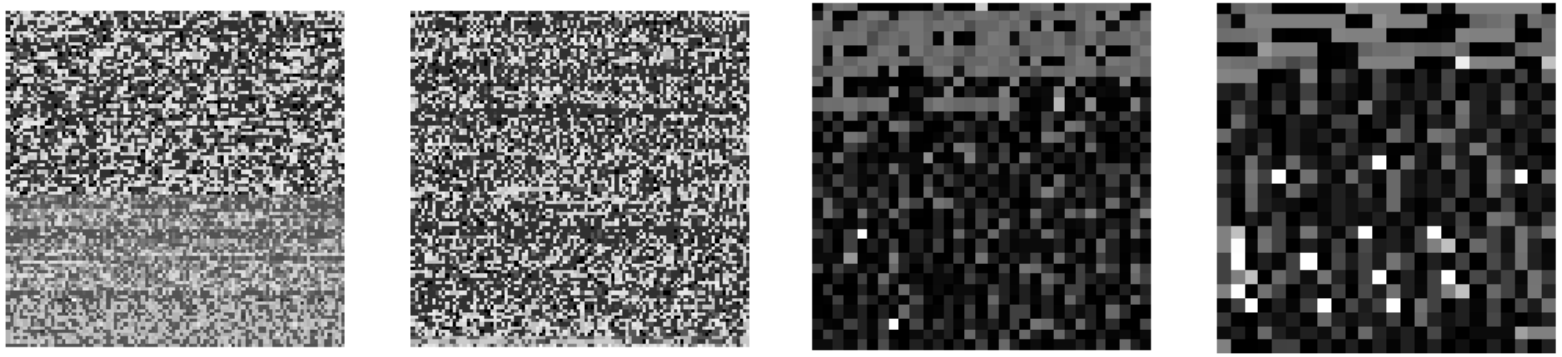
- Wasm allows code to be run on browers in near native speed
- Malware authors write code that performs mining functions in C/C++ and then compile it to Wasm using Emscripten
- Majority of cryptojacking malware is implemented in Wasm

Motivation & Research Goal



Crypojacker





Benign

Motivation & Research Goal

- Several detectors (Minos, MineSweeper, MinerRay, WasmGuard, etc.) exist.
  - ! Are they robust against (automatically generated) Wasm-specific binary perturbations?

Motivation & Research Goal

- Several detectors (Minos, MineSweeper, MinerRay, WasmGuard, etc.) exist.
  - ! Are they robust against (automatically generated) Wasm-specific binary perturbations?

- Real-world adversarial capabilities for malware diversification.

Motivation & Research Goal

- Several detectors (Minos, MineSweeper, MinerRay, WasmGuard, etc.) exist.
  - ! Are they robust against (automatically generated) Wasm-specific binary perturbations?

- Real-world adversarial capabilities for malware diversification.
  - ! Existing diversification methods' impact on real-world detectors remains unexplored.

1. C source-level or LLVM-level diversification only (*e.g.,* CROW, Tigress, emcc-obf)

2. Coarse-grained diversification methods (*e.g.,* wasm-mutate)

3. Not designed for targeting malware detectors (*e.g.,* WASMixer)

4. Limited diversification methods tested on selected detectors (*e.g.,* Madvex)

Motivation & Research Goal

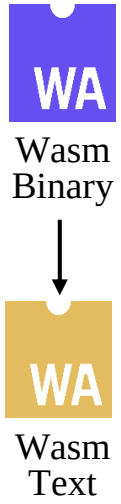
- Several detectors (Minos, MineSweeper, MinerRay, WasmGuard, etc.) exist.
  - ! Are they robust against (automatically generated) Wasm-specific binary perturbations?

- Real-world adversarial capabilities for malware diversification.
  - ! Existing diversification methods' impact on real-world detectors remains unexplored.

When Does Wasm Malware Detection Fail?

- Systematically evaluate **which types** and **magnitudes** of fine-grained, semantics-preserving binary-level perturbations cause Wasm malware detectors to fail.

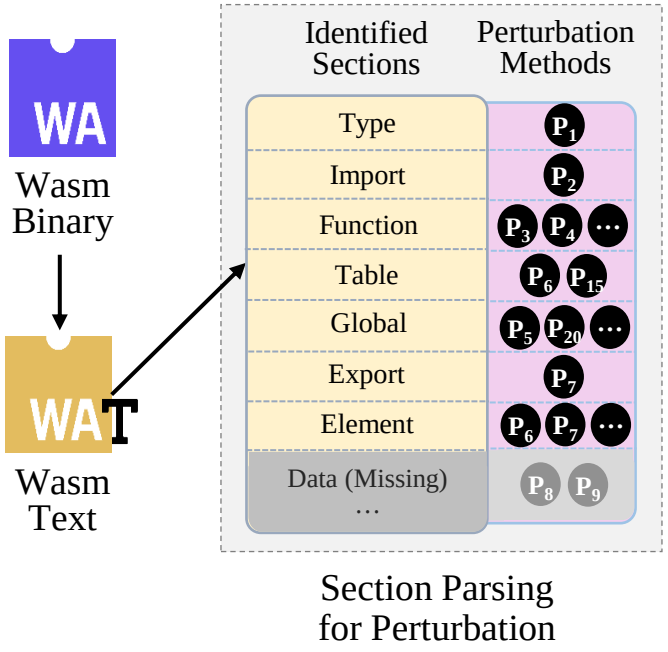
Our Framework: SWAMPED



1. **Parse** the Wasm binary into text format.

Disassemble the binary for structural and instruction-level visibility.

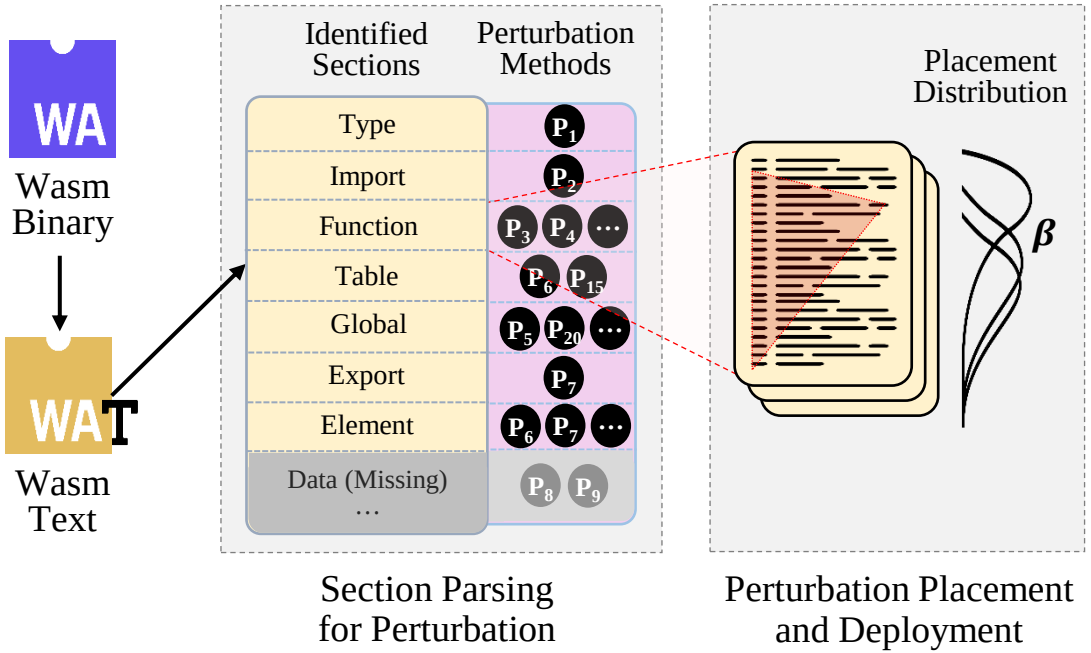
Our Framework: SWAMPED



2. **Identify** sections and **select** applicable perturbation methods.

Determine which perturbations can be applied based on the binary's composition (*e.g.,* skip data encryption if no data entries exist).

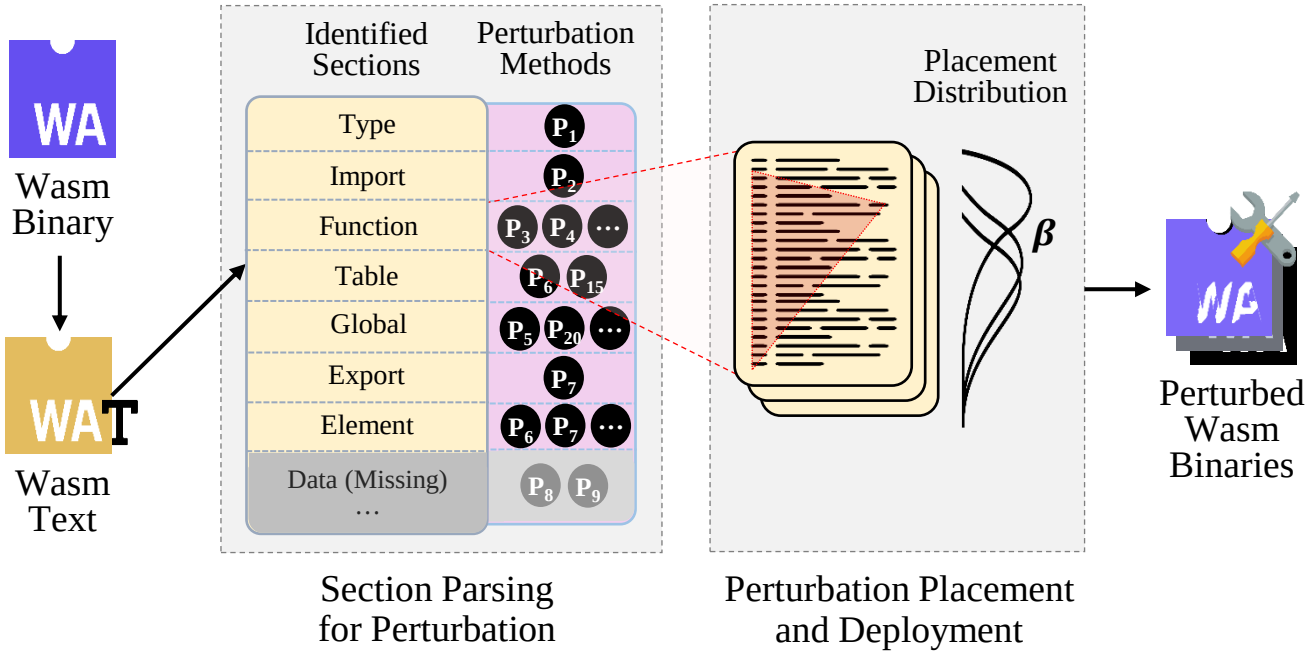
Our Framework: SWAMPED



3. **Distribute** perturbations using the **Beta-distribution model** to control where changes occur.

Adjust α and β parameters to bias the spatial distribution of perturbations within applicable regions.

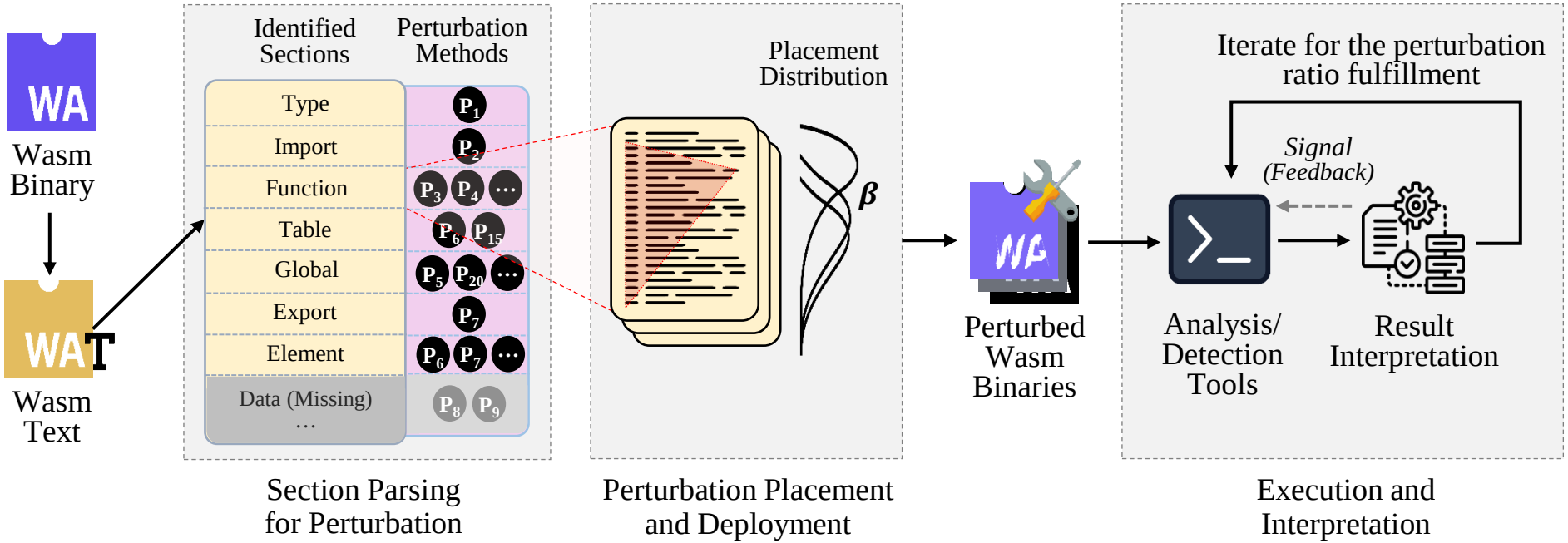
Our Framework: SWAMPED



4. **Generate** perturbed binaries by varying **perturbation ratios** to control how many changes are applied.

Tune the perturbation ratio (0–100%) to control density and quantify detector tolerance.

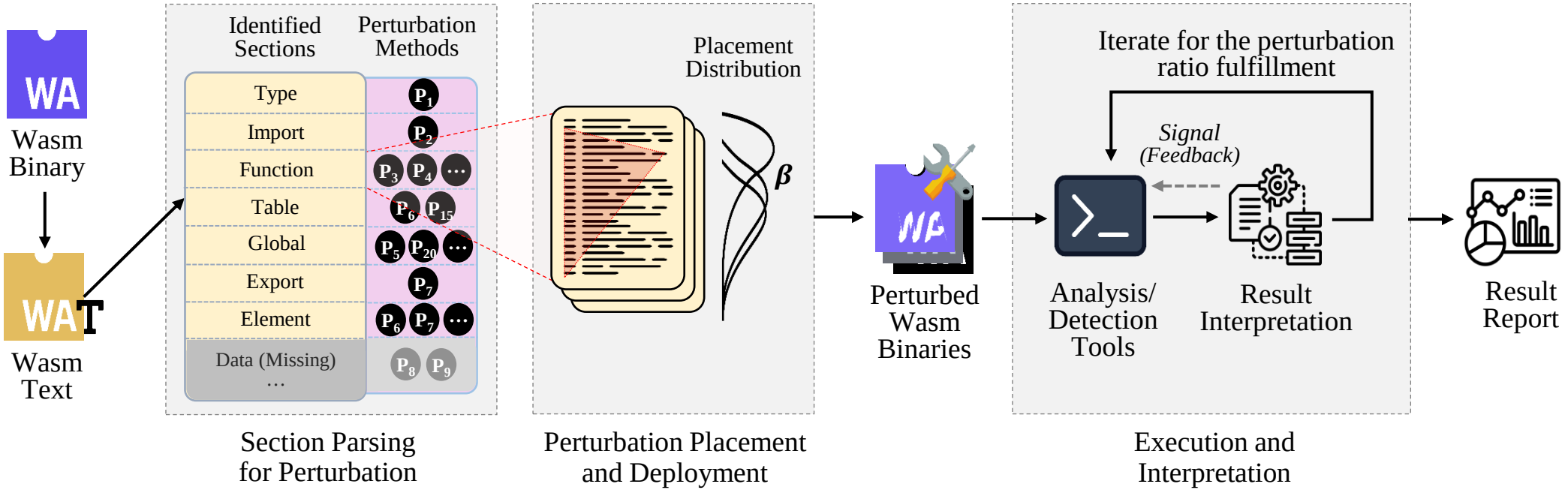
Our Framework: SWAMPED



5. **Evaluate** the perturbed binaries with detection tools and **normalize** the results into Detected, Suspected, and Benign.

Standardize heterogeneous outputs across detectors to consistently determine evasion outcomes.

Our Framework: SWAMPED

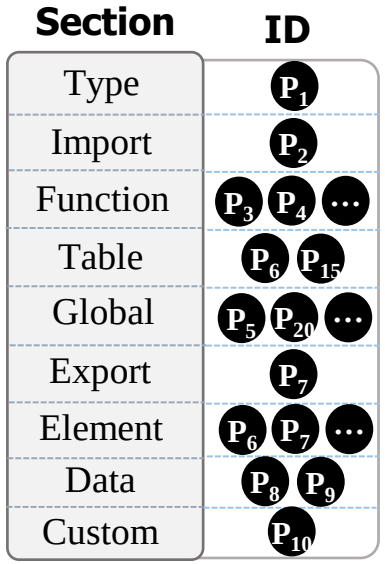


6. **Analyze** robustness degradation by checking binaries whose results shift from Detected/Suspected to Benign.

Measure how much and which perturbations erode detection confidence.

Semantics-Preserving Perturbation Methods

! Our perturbations comprehensively cover all major Wasm sections and instruction categories



Structural Perturbation

ID	Perturbation Method
P1	Function Signature Insertion
P2	Import Insertion
P3	Function Insertion
P4	Function Body Cloning
P5	Global Insertion
P6	Element Insertion
P7	Export Insertion
P8	Data Insertion
P9	Data Encryption
P10	Custom Section Insertion

Code Perturbation (Insertion)

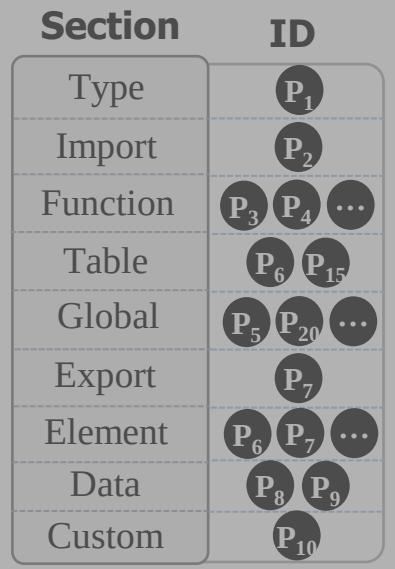
P11	NOP Insertion
P12	Stack OP Insertion
P13	Opaque Predicate Insertion
P14	Proxy Function Insertion

Code Perturbation (Transformation)

P15	Direct to Indirect Call Transformation
P16	Add/Sub Operation Transformation
P17	Shift Operation Transformation
P18	Eqz Operation Transformation
P19	Offset Expansion
P20	Transforming XOR/OR to MBA
P21	Constant Value Splitting
P22	Constant Value Transformation

Semantics-Preserving Perturbation Methods

! Our perturbations comprehensively cover all major Wasm sections and instruction categories



Structural Perturbation

ID	Perturbation Method
P1	Function Signature Insertion
P2	Import Insertion
P3	Function Insertion
P4	Function Body Cloning
P5	Global Insertion
P6	Element Insertion
P7	Export Insertion
P8	Data Insertion
P9	**Data Encryption**
P10	Custom Section Insertion

Code Perturbation (Insertion)

P11	NOP Insertion
P12	Stack OP Insertion
P13	Opaque Predicate Insertion
P14	Proxy Function Insertion

Code Perturbation (Transformation)

P15	Direct to Indirect Call Transformation

Before Data XOR Encryption:

```
(data $d0 (i32.const 1024) "Thisisdata...")
```

After Data XOR Encryption:

```
(start $decryptData)
(data $d0 (i32.const 1024)"+\02R\06\07fAm...")
```

Semantics-Preserving Perturbation Methods

! Our pertu
all major

Section

Type
Import
Function
Table
Global
Export
Element
Data
Custom

Before Stack OP Insertion:

```
i32.const 1
i32.and
i32.or
```

After Stack OP Insertion:

```
i32.const 1
i32.and
i64.const -7
i64.const -1
i64.add
drop
i32.or
```

P10	Custom Section Insertion

Code Perturbation (Insertion)

P11	NOP Insertion
P12	**Stack OP Insertion**
P13	Opaque Predicate Insertion
P14	Proxy Function Insertion

Code Perturbation (Transformation)

P15	Direct to Indirect Call Transformation
P16	Add/Sub Operation Transformation
P17	Shift Operation Transformation
P18	Eqz Operation Transformation
P19	Offset Expansion
P20	Transforming XOR/OR to MBA
P21	Constant Value Splitting
P22	Constant Value Transformation

Semantics-Preserving Perturbation Methods

Before Transforming XOR to MBA:

```
i32.add
i32.xor
local.get $l21
```

After Transforming XOR to MBA:

```
i32.add
global.set $global_1
global.set $global_2
global.get $global_1
global.get $global_2
i32.add
i32.const 2
...
i32.sub
local.get $l21
```

Code Perturbation (Insertion)

P11	NOP Insertion
P12	**Stack OP Insertion**
P13	Opaque Predicate Insertion
P14	Proxy Function Insertion

Code Perturbation (Transformation)

P15	Direct to Indirect Call Transformation
P16	Add/Sub Operation Transformation
P17	Shift Operation Transformation
P18	Eqz Operation Transformation
P19	Offset Expansion
P20	**Transforming XOR/OR to MBA**
P21	Constant Value Splitting
P22	Constant Value Transformation

Experimental Setup

- 74 Wasm malware samples (unique samples from all public sources)
- 48K perturbed variants
- 6 target detectors
- Evaluation metric: Evasion rate (%), Average Perturbation Ratio for successful evasions (%)

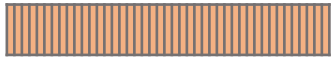
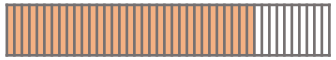
Detector	Base Technique	Target	Granularity
Minos	CNN-based image recognition	Cryptojacker	Program
MineSweeper	Profiling-based instruction frequency	Cryptojacker	Function
MinerRay	Semantic-aware IR & CFG analysis	Cryptojacker	Function
WasmGuard	Adversarially trained DNN	Malware	Program
MalConv*	Byte-pattern DNN	Malware	Program
AvastNet*	Hierarchical byte-feature DNN	Malware	Program

* We trained MalConv and AvastNet on the WasmGuard dataset

Key Findings (1/4)

Feature-based detectors: Over-fitted to targets

- Dynamic behavior or program structure-based detectors (*i.e.,* **MineSweeper**, **MinerRay**) fail when minor perturbations alter instruction frequencies or control/data-flow links.

Detector	Perturbation	Evasion rate	Perturbation ratio
MineSweeper	Shift Operation Transformation	 43/43	15.6%
MinerRay	Direct to Indirect Call Transformation	 33/43	45.2%

Key Findings (2/4)

💡 **Raw byte-based models:** Fragile to layout & distribution shifts

- **Minos**, **WasmGuard**, **MalConv** and **AvastNet** easily break when instruction or data distributions change.

Detector	Perturbation	Evasion rate	Perturbation ratio
AvastNet	Data Encryption	31/31	54.8%
Minos	Stack Operation Insertion (Numeric Arithmetic)	43/43	10.9%

Key Findings (3/4)

Adversarial training: More effective than other techniques

- **WasmGuard**'s hardens the detector on seen perturbations, yet **unseen perturbations** (e.g., XOR→MBA) still achieve full evasion.

Detector	Perturbation	Evasion rate	Perturbation ratio
WasmGuard	Global Insertion	0/31	-
	Transforming XOR to MBA	31/31	42.3%

Key Findings (4/4)

💡 **Practical implication:** Reported accuracy ≠ Robustness

- Even detectors scoring >99 % on clean samples collapse under semantics-preserving changes.

For every detector, at least one perturbation method effectively bypassed the majority of samples.

❗ Robustness and accuracy vary significantly across systems, depending on the perturbation type.

❗ We highlight the need for systematic evaluation and defense against perturbations in future designs.

Summary and Question

- 💡 Accuracy under the systematic adversarial perturbation is largely unknown.
- 💡 Essential to test Wasm analysis/detection techniques with **SWAMPED**

GitHub
SKKUSecLab/SWAMPED
Framework code,
perturbed samples,
and statistical results

Taeyoung Kim — Ph.D. Student, Security Lab
Sungkyunkwan University (SKKU), South Korea
Email: tykim0402@skku.edu

"When Does Wasm Malware Detection Fail? A Systematic Analysis of Their Robustness to Evasion," ASE 2025