YASA 工具原理详解

丁云山
山东大学

March 8, 2026

YASA 工具原理概述

Node.js 是一个以 v8 引擎为核心的集成（网络接口 + 文件系统接口 +Buffer 缓存区 +v8 引擎），功能很强大，可以直译 JavaScript 代码为机器码，然后运行。集成的 libuv 这个 C++ 库可以调用 Windows 的 API 。只能翻译描述代码结构的 JS 数据对象！

YASA 本质上是一堆.js 文件，扫描漏洞时，在 node.js 上运行，在不运行 Java 程序的前提下，通过阅读源码，构建出一张巨大的“程序逻辑地图”。

那它是怎么能够“看懂”除了 JS，其他的编程语言的呢?

我们把 YASA 的扫描过程分成几个阶段来研究:

第一阶段：UAST 转换（翻译阶段）

1.1 UAST 标准与结构

在 YASA 的体系里，UAST 是一个标准（比如规定所有的函数调用都必须叫 CallExpr）。同时他是数据结构，当 YASA 的 JS 引擎运行时，它在内存里维护着一棵巨大的树。这棵树的每一个节点都是按照 UAST 标准定义的 JS 对象。

1.2 动作与目的

- **动作**：调用 uast4py 或 uast4go 等二进制文件。（为适配运行环境，可下载 linux 版本）
- **目的**：把不同语言的代码（Java、Python、Go）统统“粉碎”，只保留了代码的逻辑骨架。

1.3 Extractor（提取器）工作原理

举例子：在 Java 里，写一个函数叫 `public void main()`；在 Python 里，叫 `def main():`。extractor 的工作是把这两者都变成一个标准化的 JSON 对象（它是一串字符串（或者在 Protobuf 下是二进制字节流），不能被执行，只能被读取）。

UAST 示例

```
{
”type”: ”FunctionDeclaration”,
”name”: ”main”,
”language”: ”Java/Python”
}
```

Extractor 内部集成了每种语言最权威的解析库，（比如 uast4go 内部用了 Go 官方的 go/parser）。它读入源码文件，先把 if、while、var 这些单词识别出来。然后生成这一门语言特有的 AST（抽象语法树）。然后它会进行一套复杂的“语义映射”。如果它看到 Python 的 `def foo():`，它会想：“这对应 UAST 里的 FunctionGroup 节点”。如果它看到 Java 的 `public void foo()`，它会想：“这也对应 UAST 里的 FunctionGroup 节点”。

Extractor 会计算变量的定义在哪里、被谁引用了。它需要告诉后面的 JS 引擎，某个变量是在哪一行定义的，它的作用域。最后，Extractor 把这些归一化的积木（其实就是很多 JSON 对象，纯文本，在追求性能时通常不会在磁盘生成永久性的.json 文件，但实际上也不尽然）按照 Protobuf（一种极快的数据传输协议）压缩打包，直接传输给 node.js 进程，这里其实利用了 Linux 的 Stdout（标准输出）重定向。

当运行 `./yasa-engine-linux-x64` 时，操作系统会在 RAM（内存条）里划出一块专属空间给这个进程。Protobuf 二进制流，被 Node.js 接收后，会立刻被“解压”并转换成一个个 JavaScript Object（JS 对象），以 uast 数据结构（树状）储存在该区域内存中。这样，yasa 就能“看懂”了！

By the way. 其实这里的逻辑极其复杂。源代码中的一个语句在最终的 uast 结构中会被翻译成很多互相嵌套的 JS 对象，也就是说，不同节点之间的关系是很复杂的……

第二阶段：准备阶段

2.1 概览

JS 引擎启动，把 Mapping 表（映射表）加载进内存，准备好那些模拟 Java 功能的 JS 脚本。内容比较复杂，而且对理解扫描进程的帮助并不是很大，不展开说了。V8 引擎（Node.js 的核心）会对这些 JS 文件进行词法分析和字节码编译。在预编译结束时，JS 脚本已经变成了内存里的 LdaNamedProperty、Star、CallRuntime 等低级指令，即字节码。实际上，它们是 V8 的 Ignition 解释器指令，完成了后文中提到的语义模拟的工作。

然后我们去看 yasa 如何在内存中加载 mapping 表，并探讨一些相关知识。

2.2 内存重构：Mapping 与规则块的物理合流

此片段涉及的原理和下文有紧密联系，看不懂也请坚持看下去，看到下文关于 rule.json 的片段时就明白了。

在 YASA 进入扫描状态前，内存中会发生一次关键的“合流”。引擎并不是分别去查询 `mapping.json` 和 `rules.json`，而是将二者物理合并为一个活跃的哈希表槽位。

当执行 `-checkerIds` 唤醒特定的规则“块”后，引擎在堆内存中开辟了基础的哈希槽位数组。此时，这些槽位仅包含来自 `rules.json` 的静态描述（如 Sink 类型或 Source 类型或可能存在的 Sanitizer 类型）。随后，`mapping.json` 开始发挥其“注入”作用：

1. **字节流加载与解析**：V8 引擎调用 `read` 系统调用读取 `mapping.json`，通过 `JSON.parse` 将其转化为临时的路径映射对象。
2. **语义脚本预编译**：针对每一个映射项（如 `requests.post`），V8 立即读取对应的 `.js` 模拟脚本，并将其预编译为低级的字节码（Bytecode）指令流（包含 `LdaNamedProperty`、`Star` 等指令）。
3. **逻辑指针注入（Pointer Injection）**：这是最核心的物理动作。引擎在已有的规则哈希表中寻址到对应 `fsig` 的槽位，将其中的 `behavior_ptr`（逻辑指针）从 `null` 强行修改为刚才编译好的字节码起始物理地址（如 `0x7FFF1234`）。

当扫描开始，引擎爬取到 `requests.post` 节点时，物理链路上只发生一次原子化的撞表操作：

- **原子寻址**：引擎对 `fsig` 进行哈希运算，直接定位到内存中唯一的槽位（Slot）。
- **双向解耦执行**：
  - **语义支线**：寄存器读取该槽位的 `behavior_ptr`。如果地址非空，指令流瞬间跳转至字节码区域，执行函数的语义模拟（如：将参数污染状态传递给返回值）。
  - **策略支线**：引擎同步读取同一槽位下的 `rule_ptr_list`。如果该节点被定义为 `Sink`，则立即触发污点校验。

这种设计规避了在扫描过程中频繁进行文件 IO 或字符串匹配的开销，使得“语义模拟”在物理层面上达到了接近原生代码的执行效率。

第三阶段：节点分析

有四类节点在经过 yasa 引擎时，需要经过 Hash 运算。其他类型节点，如控制流节点、基础表达式节点等，并不改变“毒素”性质（但是可以传播毒素，这并不是一个概念。不需要进行 hash 运算撞表模拟运行的根本原因是：不需要撞表的节点，其污点传递逻辑是普适的、原子化的。C++ 引擎通过硬编码的位运算，就已经足够支撑起这些基础节点的“毒素”搬运工作了。）。这四类节点计算内存地址所用的哈希函数是一样的，但是在各自地址中，它们所存的内容是有差别的。有趣的是，在地址中，尽管节点类型不同，他们对应的内存位置是相邻的。yasa 引擎是通过 `node.children` 指针，进行深度优先遍历。

A	B	C	D	E	F	G
节点类别 (UAST Type)	输入 HASH 的物理原料 (Key)	撞击的内存区域 (Base Address)	取回的指针数据 (Value/Pointer)			
1. Call (函数调用)	node.fullName (字符串指针指向的文本，如 java.io.File.delete)	Builtin 映射区 (由 mapping.json 转化而来)	JS 脚本实体的内存地址 (指向 FileDelete.js 的 execute 函数)			
2. Create (对象创建)	node.className (类的全限定名，如 java.net.Socket)	构造器监控区 (由 rules.json 或 builtins 转化)	初始化逻辑指针 (告诉引擎如何追踪这个新诞生的对象实体)			
3. Access (属性访问)	node.receiver.type + node.fieldName (对象类型 + 属性名)	属性安全策略区 (通常用于敏感字段监控)	访问权限/偏移量指针 (判定该属性是否携带"污点"标签)			
4. Assign (赋值操作)	node.left.type 或 node.left.name (左值容器的特征)	污点传播规则区 (Taint Propagation Table)	传播函数指针 (决定右边的"毒素"是否要传染给左边的变量)			

Figure 1: UAST 中的四类节点

3.1 常见的节点处理情况

A. 函数执行指针 (Execution Pointers)

- 对应节点：Call (函数调用)。
- 指向哪里：指向 V8 引擎中的 JS 函数对象实体（具体来说是 `execute` 函数的入口）。
- 审计作用：语义补全。比如遇到 AES.encrypt(data, key)。引擎原本不知道这会把数据变乱。
- 指针动作：跳到字节码入口，模拟运行后告诉引擎："这个操作会加密数据，虽然它是 Source 出来的，但经过这里后，'污点'标签可以去掉了"。

B. 状态初始化指针 (Initialization Pointers)

- 对应节点：`Create` (对象创建)。
- 指向哪里：指向类描述符（`Class Descriptor`）或特定的初始化策略对象。
- 审计作用：追踪对象家谱。
- 指针动作：比如 `new File(userInput)`。该逻辑会强制把这个 `File` 对象的 "内部属性" 与 `userInput` 绑定。引擎会遇到各种对该对象的操作：`file.getName()`、`file.getPath()`、`file.exists()`。这些方法本身可能并不直接读取用户输入，但因为它们的调用主体（`Receiver`）是那个 `File` 对象，引擎会顺着指针回溯到它的初始化状态（根）。引擎发现 `File` 的根是有毒的，于是判定 `file.getPath()` 的返回值也必须染红。在 rules.json 加载出的 hash 表中，它们通常也使用 fsig 进行撞表。

C. 策略判定指针 (Policy/Strategy Pointers)

- 对应节点：Access (属性访问)。
- 指向哪里：指向内存中的规则判定机（Rule Evaluator）。
- 指针动作：比如：`session.user=request.getParameter("id")`, 指针指向 C++ 引擎中 Assign 表中的一个 entry，它定义了："如果右边是 Tainted，左边也必须染红"，控制流与污点传播判定。在绝大多数情况下，在 rules.json 加载出的 hash 表中，它们是依靠 path 来匹配的。如果成功匹配，才执行上述染色操作。

D. 内置传播逻辑 (Built-in Propagation Logic)

- **对应节点**：`Assign` (赋值)、`BinaryExpression` (二元运算) 等。
- **物理特征**：**无需撞表**。此类节点属于引擎内核的“硬编码定律”。
- **内核动作**：
  - **赋值**：当遇到 `=` 节点时，引擎直接触发内核中的 `handle_assignment` 分支。若右值（`RValue`）已在之前的撞表中被标记为 `Tainted`，内核将自动把左值（`LValue`）在内存变量表中的状态位置为 1。
  - **运算**：在 `"hello " + tainted_user` 场景下，内核逻辑确保污染位沿字符串拼接链条物理传递。

写到这里，我必须指出，在 yasa 引擎实际运行中，撞表概率是极其之大的。这是第一处撞表，是为了补充代码逻辑。原因是：yasa 引擎是基于 node.js 运行的，它通过调用 C++ 来阅读 uast 数据结构，但是 C++ 引擎只能分析他看的到源码的代码。比如遇到 os.system() 时，源码在编译好的字节码里或者在 DLL（Dynamic Link Library，动态链接库，指的是那些已经编译成机器码、不再提供原始代码的外部函数库）里，C++ 引擎看不到内部结构。其次，作为硬编码，C++ 在静态分析中给一个不存在的源码对象建模极其复杂，对于函数会返回复杂的动态对象的情况，如果要它编码所有可能性，引擎会变得臃肿且极难维护。

现在我们可以给 Builtin 一个清晰的定义了：它就是针对 call 类型节点和少数 Create 类型节点以及极少数 access assign 类型节点，存在于哈希表中、指向特定 JS 脚本解析出的字节码的内存指针。涵盖了所有“语义不透明”的节点。

第四阶段：正式扫描与“染色”原理

4.1 配置文件结构与“块”激活逻辑

以 `rule_config_python.json` 为例，其物理结构并非简单的键值对，而是一个由多个独立“规则块”构成的数组。这种设计是 YASA 引擎实现“按需加载”与“物理隔离”的核心。

规则配置块的物理结构

```
[
{ // <--- 块 A: 专注于基础输入流 (例如基础 Source)
"checkerIds": ["taint_flow_python_input"],
"sources": { ... },
"sinks": [ ... ]
},
{ // <--- 块 B: 专注于反射等高级语义 (例如特定类模拟)
"checkerIds": ["java_reflection_ai"],
"sources": { ... },
"sinks": [ ... ]
}
]
```

这种“数组嵌套块”的结构并非多此一举。在实际执行指令时，我们需要手动指明 `-checkerIds`，其本质物理动作是 ** “从规则池中唤醒特定的激活块” **。

当执行命令 `-checkerIds java_reflection_ai` 时，引擎内部会发生以下连锁物理动作：

1. **全量读取 (Memory Loading)**：引擎首先将整个 `rules.json` 读入 RAM。此时，它在内存中表现为一个由多个 `Object` 构成的数组。
2. **物理筛选 (Linear Traversal)**：引擎启动线性遍历，对每一个“块”对象进行特征匹配。
   - **判定块 A**：扫描其 `checkerIds` 列表，发现不包含目标 ID。动作：物理丢弃，该块内的规则不占用后续扫描资源。
   - **判定块 B**：匹配成功！动作：激活。
3. **注入扫描器 (Hash Injection)**：只有被激活的那个块（块 B）内部定义的 `sources`、`sinks` 及其关联的方法签名（如 `Method.invoke`），才会被物理注入到引擎运行时的活跃哈希寻址表中。

这种“块”状结构确保了同一个配置文件可以集成多套审计逻辑。同一个块内可以定义多个 `checkerId`，指令中命中任意一个，即可物理激活整个块的物理功能，从而在保持规则灵活性的同时，极大提升了哈希碰撞的效率。

4.2 `fsig` 与 `path` 双重匹配机制

在 YASA 引擎的运行流程中，源码经过“翻译”、“传输”与“解压”后，在堆内存中表现为互相嵌套且关系复杂的 JS 对象节点。识别这些节点是否为“污染源”，本质上是依靠 `fsig` 与 `path` 两条物理索引链路与内存中的规则库进行“撞表”。

A. `fsig` (Function Signature) 匹配 `fsig` 是连接静态 `json` 规则与动态 `uast` 扫描引擎的唯一物理索引。必须强调的是，`fsig` 是函数调用类型节点和对象创建类型节点所特有的。

- **物理合成**：引擎在 UAST 树上爬行时，每遇到一个函数调用节点，都会通过回溯作用域与符号解析，动态合成一个格式为 `[命名空间].[类名].[方法名]` 的标准 `fsig`。
- **内存映射 (Key-Value) 构造**：在 `Node.js` 进程的堆内存中，激活的规则块会形成一个以 `fsig` 为键的哈希表。
  - **索引键 (Key)**：根据语言特性不同，格式可能为 `calleeType + "#" + fsig` 或纯 `fsig` 字符串。
  - **值对象 (Value) 的双指针结构**：每个槽位实质上维护了两个核心指针：
    1. `behavior_ptr` **(逻辑指针)**：指向内存中的 `ModelEntry`。它关联了预编译阶段生成的 JS 字节码函数地址。若该值为 `null`，则表示该函数只需 C++ 标准逻辑处理。事实上，在函数调用节点中，这种情况并不少见，譬如内置语义节点和纯规则判定节点。这时，C++ 引擎会去检查规则指针。
    2. `rule_ptr_list` **(规则指针)**：指向一个列表，包含该 `fsig` 在 `rules.json` 中定义的静态规则描述符（如 `SourceDescriptor` 或 `SinkDescriptor`）(其实就是对应块的 sources 实体和 sinks 实体)。

B. 从寻址到动作触发 当引擎在 UAST 树上爬到一个 `CallExpression` 节点并算出 `fsig` 后，会立即执行哈希运算进行第二次撞表：

1. **哈希寻址**：例如 `os.read` 被转换为哈希值 `0x5A2B`。引擎拿着这个地址去查内存里的 `Global Index Table`。
2. **行为判定 (Hit Logic)**：
   - **命中字节码索引**：若该地址指向了 `ModelEntry`，引擎会暂停当前的通用遍历，触发环境切入，跳转执行对应的 V8 字节码逻辑（模拟函数内部行为）。
   - **命中规则索引**：若该地址指向了 `SourceDescriptor`，引擎会读取其中的静态配置（如染色位置 `values` 或作用域 `scopeFile`），触发静态染色。
   - **未命中 (Miss)**：若双重索引均未命中，污点传播链条将在此节点发生物理断裂。

C. path (Object Path) 匹配 针对非函数调用的 `MemberExpression`（如直接读取 `request.args`）。引擎递归拼接路径字符串并计算哈希值。若在 `map_taint_sources`（rules.json 加载出的一张子表）中撞表成功，该路径即被直接判定为“天然污染源”，无需进入复杂的字节码模拟即可完成快速染色。

4.3 语义模拟与物理染色逻辑

当引擎命中具有语义模拟需求的节点（如 `os.read`）时，将触发从 C++ 主引擎到 V8 运行时的控制权转移：

- **环境切入 (Context Switching)**：引擎通过调用 V8 虚拟机接口暂停当前的 UAST 遍历逻辑。指令寄存器指向预编译的字节码入口，将当前节点的上下文（Context）注入脚本环境。
- **物理染色 (Taint Injection)**：字节码模拟函数执行的副作用（Side Effects）。它通过操作内存堆栈，将该节点的返回值（`ReturnValue`）或关联实例的 `state.attribute` 标志位强制修改为预设的污染掩码（即“涂红”）。
- **控制权回归**：脚本执行完毕后，V8 虚拟机返回执行结果，主引擎接管控制权，携带已染色的节点继续向下游 UAST 分支爬行。

4.4 漏洞爆破点 (Sink) 的判定机制

在 YASA 的设计中，绝大多数 `Sink` 规则集中于 `FuncCallTaintSink` 类型。这反映了 **95%** 以上的安全威胁（如 SQL 注入、SSRF）物理终结于函数调用节点。

启动阶段，配置文件中的 `sinks` 定义被解析为 `SinkDescriptor` 对象并常驻堆内存。这些对象的物理指针存储在全局索引表（Global Index Table）对应 `fsig` 的 `rule_ptr_list` 槽位中。当引擎命中 `CallExpression` 且检索到 `SinkDescriptor` 指针时，执行以下操作：

1. **参数寻址 (Addressing)**：引擎依据描述符中的 `args` 索引（如 `["0", "1"]`），物理定位至 UAST 节点的参数内存对象。
2. **位运算判定 (Bitwise Check)**：引擎提取参数对象的 64 位属性标记位，与 `Sink` 要求的毒性掩码执行物理位运算：

$$(Parameter.attribute \ \& \ Sink.required\_attribute) \neq 0 \tag{1}$$

   物理意义：若运算结果非零，证明该参数携带的特定毒性（如 SQL 注入位）与该 `Sink` 恐惧的漏洞类型物理匹配。

3. **净化校验 (Sanitization Check)**：在最终告警前，引擎会核验该参数的传播路径是否经过 `sanitizerIds` 关联的过滤函数。若路径中未探测到有效的净化动作，则物理触发告警生成逻辑。

第五阶段：总结与归纳

5.1 污点注入的三个物理入口

在 YASA 引擎的物理扫描链路中，污染源（Sources）的识别不分“形态”，只分“特征”。引擎通过统一的索引逻辑，在三个关键位置完成染色的“第一针”：

1. **函数调用返回值 (ReturnValue)**：针对 `CallExpression` 节点。当函数执行完毕（若存在 `behavior_ptr` 则在字节码模拟后），引擎根据规则指令将返回结果在内存中“涂红”。

2. **声明与参数入口 (Entry Point)**：针对 `FunctionDeclaration` 的 `Parameter` 列表。当引擎扫描到接口函数（如 Web Controller）时，会根据规则直接对入口参数积木进行染色。

3. **属性访问路径 (Property Path)**：针对 `MemberExpression`。例如访问 `request.args` 时，引擎不依赖函数调用，仅凭拼接后的物理路径匹配 `path_map` 即可完成染色。

5.2 64 位标志位 (Attribute/Taint Mask) 体系

为了实现“无差别的数据流传递”，YASA 为 UAST 中的每一个 JS 对象实体（就是每个节点）分配了一个 64 位的物理标志位（Attribute）。这一设计是引擎实现跨语言、跨框架分析的底层基石：

- **变量/表达式节点**：存储当前符号（如 `x`）或运算结果（如 `a + b`）的污染状态。
- **常量节点 (Literals)**：存储硬编码字符串本身的属性。例如 `"select * from"` 可能被标记为具有“SQL 语义”的特殊常量。
- **无差别传递**：内核逻辑通过对这些 64 位标志位进行位运算（如 `OR` 操作）来实现污点的自动扩散。

5.3 Sources 的物理分类与三表寻址

在内存层面，`Global Index Table` 根据 `sources` 的作用域拆分为三个独立的子表，以实现 $O(1)$ 级别的极速检索：

子表名称	对应规则类型	物理动作
`map_taint_sources`	`TaintSource`	变量/路径匹配成功即刻染色
`map_return_sources`	`ReturnValueTaintSource`	函数执行结束后对其返回值染色
`map_arg_sources`	`ArgTaintSource`	函数调用瞬间对传入的特定参数染色

底层特征观察：值得注意的是，这三类污点源并非只服务于函数。

- 对于 `TaintSource`，主要包含成员访问节点（`MemberExpression`）和函数声明节点（`FunctionDeclaration`），它们依靠 `path` 撞表。
- 在内存槽位中，只有“函数调用”类和存在模拟脚本的 Create 类节点的 `behavior_ptr` 可能指向 V8 字节码入口；而对于非调用的 `TaintSource` 节点，其逻辑槽位几乎均为 `nullptr`。这意味着此类节点的染色是“静态且即时”的，不触发虚拟机环境切入。

写到这里，yasa 引擎的基本工作流程和原理好像阐释的差不多了，其中还有诸多细节，因本人水平有限，无法做到深入分析……以上只是大致逻辑，在实现新 checker 和二次开发时，还需参考具体文件结构及内容。对上述内容，若能纠正潜在的不当之处，不胜感激。