2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)

JITBULL: Securing JavaScript Runtime with a Go/No-Go policy for JIT Engine

Jean-Baptiste Decourcelle, Boris Teabe, Daniel Hagimont
IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3
Toulouse, France

***Abstract*—Nowadays, most services are delivered through the web and thus heavily rely on JavaScript (JS). To accommodate the need for more performance, JS runtimes integrated Just-In-Time (JIT) compilation engines, which compile frequently-called portions of code for faster execution. To produce efficient machine code, the JIT applies complex optimization passes on the code in question. However, inadequate modeling of the side effects of these optimizations can introduce vulnerabilities in certain optimization passes. Such vulnerabilities are regularly discovered, and often have a high impact. Once a vulnerability is identified, it is eventually patched, but not without involving several steps (development, testing, release, user consent), leaving the system vulnerable for a relatively long period: the *vulnerability window*.**

We propose JITBULL, a solution that secures the JIT engines of JS runtimes during the vulnerability window by leveraging a vulnerability's *demonstrator codes*. To that end, JITBULL extracts the effects of JIT compiler optimization passes on said vulnerability demonstrator codes. For every subsequent JITed code, JITBULL compares the effects of its optimization passes with those on the demonstrator codes. If similarities are detected, JITBULL assumes that the currently executing script may be malicious and disables the related optimization passes, or if that's not possible, the whole JIT engine.

We implemented JITBULL in Firefox's JS runtime (SpiderMonkey) and tested it against several known vulnerabilities with public demonstrator codes. Our results demonstrate that JITBULL consistently safeguards the JIT engine against exploitation by a variant of a known vulnerability. Moreover, we show that JITBULL exhibits a false positive rate of less than 5% on the JS Octane benchmark suite, while causing an acceptable overhead of less than 20%.

I. Introduction

Today, many complex services are provided through user-friendly web interfaces, which places a significant security responsibility on the web. Initially, all business processing was done on web servers using programming languages like PHP. In 1996, the JavaScript (JS) programming language was introduced [1], enabling rich client-side business processing. Since then, browsers have been equipped with a JS runtime [2] responsible for interpreting the JS script used to modify page content without requiring server interaction. Part of the workload is therefore relocated to the client machine. In order to be faster, JS runtimes have integrated a Just-In-Time (JIT) compiler [3]. The principle behind a JIT compiler is to compile frequently-called code portions into machine code on-the-fly to benefit from the faster execution of compiled code compared to interpreted code. Due to its performance impact [4], the JIT compiler is a crucial component of the JS runtime. However, despite all the efforts made by JS runtime developers to secure this component, it remains susceptible to vulnerabilities. Our work thus focuses on vulnerabilities in the JIT engine.

Before diving into JIT engine vulnerabilities, it's essential to understand how JIT engines work. They define a threshold of the number of times a function must be invoked before it is considered frequently-invoked. If a function is executed more times than as specified by the threshold, it will be compiled into machine code. The JIT engines of the three most widely used JS runtimes (V8, JavaScriptCore and SpiderMonkey) utilize an Intermediate Representation (IR) generated from scripts to perform successive optimizations on the code before producing machine code. Each optimization *pass* applies various modifications to the code, which can involve renumbering, reorganizing, modifying, moving, or deleting instructions.

By analyzing recent vulnerabilities over the 2015-2021 period targeting V8 and SpiderMonkey, we discovered that **most flaws resulted from the optimization passes performed by the JIT engine.** In fact, in some cases, optimization passes modify the original code without proper control over side effects, which opens up exploitable security vulnerabilities to compromise the JS runtime. For an example, let us look at the `CVE-2019-17026` vulnerability [5] affecting SpiderMonkey. For this vulnerability, the Global Value Numbering (GVN) optimization pass, which aims to eliminate redundant code, removes a *boundcheck* instruction, which checks if an index is within the bounds of an array, from the IR. Poor modeling of the instruction's side effects leads this pass to remove this instruction in conditions where it is essential. Without this check, an attacker is able to exploit JS arrays to obtain read and write primitives on memory areas. These primitives will allow redirecting the execution to previously injected malicious code. We also noted that the vast majority of these vulnerabilities targeting JIT have a very high CVSS (Common Vulnerability Scoring System) score[1] (greater than or equal to 8.8), which means they have a significant impact on JIT engines' security.

The typical approach to address a software vulnerability is to apply a security patch. In practice, patching is a long process beginning with the vulnerability's discovery, which is often done by a security specialist who submits it to the software authors for analysis, along with a *vulnerability demonstrator code* (VDC). The software authors then have a duty to notify users of the vulnerability's existence and to work on producing a

[1]CVSS is a metric for assessing the severity of vulnerabilities. The CVSS score ranges from 0 to 10, with 0 for the least critical vulnerability and 10 for the most critical.

2158-3927/24/$31.00 ©2024 IEEE
DOI 10.1109/DSN58291.2024.00028

patch. Producing a patch involves a development phase followed by careful testing to ensure that the patch does not impact the product's operation or cause a regression in its security, as it was the case with Log4j [6]. In other words, the cure should not be more harmful than the disease. When performed with all the recommended precautions, this stage can be time-consuming [7]. Next, the development team selects a release method for the patch. Typically, to prevent overwhelming users with updates, patches are grouped into batches. Finally, once the update is available, applying the patch may be delayed as user consent is often required before proceeding with the update. This period from vulnerability discovery to system update is known as the **vulnerability window** [7]; it corresponds to the period when the vulnerability is known, but the system is still vulnerable. This period of insecurity is a significant problem given that the majority of services and applications are now accessed through the web. As the browser serves as the primary point of entry for these services, it is therefore crucial to ensure a high level of security for its JS runtime, and especially JIT engine during a vulnerability window.

In this work, we introduce the JITBULL system, which protects the JS runtime from known attacks exploiting the JIT component until a security patch is applied. JITBULL secures the JS runtime during a vulnerability window using vulnerability demonstrator codes provided by the vulnerability's reporter. The intuition behind JITBULL is that if a vulnerability occurs due to incorrect optimization, as shown by a demonstrator code, then all other scripts that intend to exploit this vulnerability should demonstrate the same pattern of effect on the optimization process. Therefore, we can detect attempts to exploit bad optimizations from any script by comparing the impact of optimization passes between said script and a set of VDCs. In other words, the basic principle of JITBULL is to: firstly, extract the effects of the JIT compiler's optimization passes on the JITed codes from a set of VDCs; and secondly, compare them to those of running scripts. JITBULL's unique strength is its response when strong similarities are detected in an optimization pass: instead of completely disabling the JIT, JITBULL only disables a specific optimization pass on the JITed code, while still benefiting from other speedups granted by JIT.

Before presenting the workflow of JITBULL, let us formalize several core concepts. Denote Δ_i the modifications on a JITed code's intermediate representation (IR) made by optimization pass $i \in [1..n]$, n being the total number of passes. We call IR_{i-1} the IR prior to the optimizations of pass i and IR_i the IR code in the post-optimization, therefore loosely speaking $\Delta_i = IR_i - IR_{i-1}$. We consider the vector $(\Delta_1, \Delta_2, ...\Delta_n)$ of a JITed code as its *JIT DNA* (or simply *DNA*). The two steps of JITBULL are as follows:

1) For each vulnerability's demonstrator code, JITBULL executes it while extracting its *DNA vectors* for all optimization passes, i.e. $(\Delta_1^{'}, \Delta_2^{'}, ...\Delta_n^{'})$, and saves them in a database.
2) For every attempt to JIT-compile a code script, JITBULL extracts the *DNA vector* for all optimization passes from the executed code, $(\Delta_1, \Delta_2, ...\Delta_n)$, and compares it with all vectors from the demonstrator code, i.e. comparing the $(\Delta_1, \Delta_2, ...\Delta_n)$ with all $(\Delta_1^{'}, \Delta_2^{'}, ...\Delta_n^{'})$ vectors. If a strong similarity is found for certain passes, i.e. loosely speaking $\Delta_i \approx \Delta_i^{'}$, then JITBULL disables these passes to prevent them from being used as entry points for a vulnerability for this specific JITed code.

The strength of JITBULL lies in two factors. Firstly, it does not completely disable the JIT engine; through its analysis, JITBULL targets modifications made by each pass and determines which passes are dangerous. Furthermore, the pass is disabled just for specific risky JITed codes. This secures the JIT engine while keeping it functional until a patch is available and applied. Secondly, unlike other security solutions, JITBULL does not require any prior work on demonstrator codes to identify the parts that are responsible for the flaw. Nor is there any need to tell JITBULL which passes are potentially dangerous, as JITBULL finds them by comparing the outcomes of optimization passes.

JITBULL's architecture consists of two components: the Δ **extractor** and the Δ **comparator**. The role of the Δ **extractor** is to extract the *DNA vectors* of JITed codes. It is used preemptively to extract the *DNAs* of demonstrator codes mentioned in **step 1**, as well as each time a JS code is JITed as stated in **step 2**. The Δ **extractor** uses the IR to generate an instruction dependency graph and uses this graph to identify the modifications after each pass. The Δ **comparator** compares two DNA vectors Δ_i and $\Delta_i^{'}$, and determines whether they are similar to each other. Upon a confirmed similarity, the Δ **comparator** disables the pass i in the JIT engine.

We implemented JITBULL on SpiderMonkey [8], Mozilla Firefox's JS runtime, but the approach used is valid for all recent JS runtimes. We evaluated JITBULL on two criteria: security and performance. In terms of security, we found that once a demonstrator code is integrated into its database, JITBULL always manages to detect variants seeking to exploit the same flaw. In particular, using four vulnerabilities in SpiderMonkey, with four demonstrator code variants for each vulnerability, we achieved a 100% detection rate. This implies that JITBULL was able to identify the dangerous optimization passes and disable them on all exploit variants. We also evaluated JITBULL with the Octane JS benchmark [9], which contains no malicious code. Under this scenario, we found that the proportion of functions that JITBULL wrongly considered dangerous ranges between 0 to 5% for most scripts. Next, we evaluated the impact of JITBULL on JS performance. An evaluation using the Octane benchmark suite shows that JITBULL introduced an additional performance cost ranging from 1% to 20% on the benchmarks when four vulnerabilities are built into JITBULL's exploit database. However, it's worth noting that this performance impact is significantly less than what would result from a complete deactivation of the JIT engine. Additionally, this overhead is nil when there are no vulnerabilities in the database.

We provide the background the reader needs to understand our work in Section II. Subsequently, Section III presents an

analysis of vulnerabilities, including their origin, and impact and methodology, to explain the motivations behind our work. Section IV describes the workflow and main concepts of our solution JITBULL. Section V gives details of JITBULL's implementation. Then, in Section VI, we evaluate JITBULL using the scenarios described above. Section VII discusses the related works and compares them to our approach. Finally, Section VIII concludes our article.

II. Background

In this section, we introduce the prerequisites for understanding our contribution by presenting how a JS runtime and its JIT engine work.

To optimize the execution of JS code, runtimes today make use of several optimization layers, often integrated into the JIT compiler. The role of the JIT engine is to compile frequently invoked code portions in order to execute them more quickly. It is known that compiled code is faster than interpreted code. The idea behind the JIT compilation is that it is more advantageous to waste some time compiling a code and reuse this code several times than to rely solely on interpretation to execute a frequently called code. To pay back the compilation cost, the compiled code must be invoked several times. This is why JS runtimes define an invocation threshold beyond which the code is JITed. To make the machine code as efficient as possible, it is optimized through several optimization phases. In the rest of the section, we focus on Mozilla Firefox's JS runtime, *SpiderMonkey* and its JIT engine *IonMonkey* [10].

Figure 1 illustrates the principle of compiling a JS code using JIT. Within SpiderMonkey, JS code is first compiled into bytecode (step ❶ in Figure 1), which is initially interpreted. When a function is executed very frequently, it can either be compiled with Baseline JIT, which produces unoptimized binary code, or with IonMonkey JIT, which produces optimized code. Baseline JIT produces machine code without any optimization, and is called after 100 invocations of a function (step ❷). Ion JIT (IonMonkey), on the other hand, performs optimizations on the code and is called after 1500 or more executions of the function. In this situation, the bytecode is sent to IonMonkey to be transformed into a mid-level intermediate representation (MIR) by the compiler (step ❸). The MIR is a graph made up of blocks, each of which corresponds to a possible branch of the execution flow. A block consists of instructions in static single-assignment form (SSA), meaning that there are no variables. Each instruction returns an object with a type and value, and can take other instructions as operands, referencing them with their line number and opcode.

Over this intermediate representation, IonMonkey applies 32 optimization steps that we call passes (step ❹ in Figure 1). Each pass performs complex modifications designed to optimize the execution of the binary code that will be produced at the end of the process. For example, the GVN (global value numbering) pass is a classic method of eliminating redundant instructions and is responsible for the vulnerability we will discuss in section III. The LICM (loop-invariant code motion) pass searches for instructions that produce the same result in any iteration of a loop, and removes them to execute them only once beforehand. Some passes have a more basic role, such as eliminating empty blocks or renumbering instructions. IonMonkey then transforms the optimized MIR (MIR' on Figure 1) into a LIR (low-level intermediate representation) in step ❺. This representation is similar to the MIR, and also undergoes optimization passes (step ❻), but focuses on binary code generation and can be supplemented by platform-specific instructions. Finally, the optimized LIR code (LIR' on Figure 1) is transformed into binary code (step ❼). V8, the most widely used JS engine also uses the principle of optimizations that we observe in step ❹.

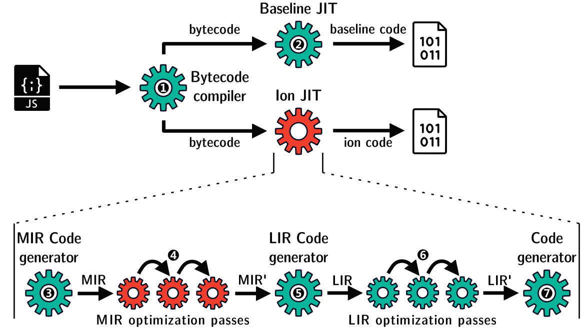

Fig. 1: SpiderMonkey JIT compilation steps.

III. Motivation

In this section, we analyze past vulnerabilities in JIT engines and discuss current solutions that act during a JIT runtime's vulnerability window.

A. Analysis of JIT Engine Vulnerabilities

We surveyed vulnerabilities that impact JIT engines, specifically V8's Turbofan, SpiderMonkey's IonMonkey, and Chakra's nameless JIT engine (as used in Microsoft Internet Explorer) during the period from 2015 to 2021. Table I displays the vulnerabilities that we identified.

TABLE I: List of vulnerabilities in the JIT engines of V8, SpiderMonkey, and Chakra runtimes. Vulnerabilities with a demonstrator code are bolded.

Target	Vulnerabilities
TurboFan	**CVE-2021-30632** , **CVE-2021-30551** CVE-2020-16009 , CVE-2020-6418 CVE-2019-2208 , **CVE-2018-17463** **CVE-2017-5121**
IonMonkey	CVE-2021-29982 , CVE-2020-26952 CVE-2020-15656 , **CVE-2019-17026** **CVE-2019-11707** , CVE-2019-9813 **CVE-2019-9810** , CVE-2019-9795 CVE-2019-9792 , **CVE-2019-9791** **CVE-2018-12387** , CVE-2017-5400 **CVE-2017-5375** , CVE-2015-4484 CVE-2015-0817
Chakra JIT	CVE-2021-34480 , **CVE-2020-1380**

Following is a description of our analysis procedure. We searched for all the vulnerabilities affecting these JIT engines

on the NIST National Vulnerability Database web site [11]. For each identified vulnerability, we thoroughly examined its description to understand its operation. Additionally, when available, we sought out demonstrator code that could exploit the vulnerability. In Table I, we listed all the vulnerabilities targeting these platforms, and highlighted in bold those for which there is one or more demonstrator codes available or a white paper explaining how they work. Our analysis has led us to draw two key conclusions:

1) Vulnerabilities affecting JIT engines have high CVSS scores, averaging 8.8 on a scale of 10. This demonstrates the dangerous nature of these vulnerabilities, which generally attack all three aspects of security: confidentiality, integrity, and availability.
2) Concerning the vulnerability exploitation mechanisms: as we mentioned in section II, current JIT engines operate in a very similar way, with a sequence of optimization passes over an intermediate code representation. In general, we observe that these vulnerabilities originate from optimization passes which, for the most part, remove portions of code for performance reasons, without appropriately taking into account their side effects.

JIT compilers typically optimize programs by deleting control instructions to speed up execution. However, as stated above, they often make logic errors like incorrect assumptions on optimized code, leading to flaws such as type confusion or incorrect bounds checking. To illustrate this, let us investigate the mechanisms behind a particular vulnerability in IonMonkey.

B. IonMonkey vulnerability: `CVE-2019-17026`

`CVE-2019-17026` stems from an error when manipulating JavaScript array objects. In JavaScript, an array is represented as an object with various properties, especially the array length. The length is used to limit accesses to the array's elements, and it can change during the array object's lifetime, either by increasing or decreasing. When the array size is decreased, SpiderMonkey reclaims memory areas that no longer belong to the array. During optimization passes applied by IonMonkey, more specifically during the Global Value Numbering (GVN) pass, the array bounds check is eliminated under certain conditions. However, a vulnerability is introduced by an incorrect check elimination resulting from an incorrect dependency analysis. The available proof-of-concept for `CVE-2019-17026` exploits this weakness by first allocating two adjacent arrays in memory, then reducing the size of the first array to force the elimination of the bounds check. This means accesses to the first array can overflow into the second array's memory region. As a result, using the first array, one can influence the properties of the second array, thus introducing a pair of *arbitrary read/write* primitives. The subsequent step involves utilizing JIT spraying [12] to inject executable malicious binary code into the JIT engine's memory. To elaborate, the read primitive can be used to obtain the shellcode's address, and then the write primitive changes the address of a function's JIT pointer to redirect the execution flow to this code. Note that the `CVE-2020-1380` [13] vulnerability in Chakra uses a similar approach on the *GlobOpt* optimization pass.

Furthermore, some of these vulnerabilities rely on the same flaw. For instance, `CVE-2019-9810` and `CVE-2019-17026` rely on the same system bug, despite their differences in the time of discovery. This implies that some attackers are, in fact, repeatedly using variant analysis methodologies to successfully identify similar bugs missed by the original reporters and fixers. Hence the need to build a system that protects the JIT engine once the vulnerability is known while waiting for an effective patch to be produced.

C. Handling Vulnerabilities in JS Runtimes

As with most other vulnerabilities, a vulnerability that targets JIT engines is remediated by applying a software patch. However, the time from patch production to its application, corresponding to a period during which the system remains vulnerable (referred to as the *vulnerability window*) can be very lengthy. Generally, it is challenging to determine the vulnerability window duration for JIT engine vulnerabilities because it is difficult to establish the discovery dates of vulnerabilities from the websites that reference them. But, through Mozilla's bug tracker[14], we were able to determine the date of reporting vulnerabilities in IonMonkey and the date when the security patches were available. In the list of vulnerabilities presented in Table I, we observed an average duration of **9 days** between the discovery of the vulnerability and the availability of the patch. This duration varies depending on the vulnerabilities. It is lengthy for some; for instance, `CVE-2019-11707` was reported on April 15th, 2019, and the patch became available on May 8th, 2019 (23 days). However, it is shorter for others; `CVE-2020-26952` was discovered on September 27th, 2020, and the patch was issued on October 2nd, 2020 (5 days). Depending on the difficulty of writing a functional patch with proper security and reliability guarantees, the production of patches can be short or long. More importantly, the patch should not cause additional problems when solving the original one, as observed with the initial fix of the Log4j vulnerability [6]. Furthermore, another factor contributing to delays in companies is the necessity to thoroughly test patches before deployment. This precaution is taken to mitigate the potentially significant costs associated with recovering from a flawed patch. Research indicates that a substantial majority of attacks exploit known vulnerabilities [15]. Therefore, we conclude that *it is necessary to define reliable methods for safeguarding the JS runtime during a vulnerability window*.

The straightforward approach to mitigate a JIT vulnerability during a vulnerability window is to disable the JIT engine completely. Many browsers already contain a setting to this effect, and it is specified as a way to reduce the JS runtime's attack surface [16], [17]. Unfortunately, this mitigation comes with two major limitations that curtail its use. Firstly, current browsers offer a global no-JIT option with per-site customization but do not provide a fine-grained mechanism to select which scripts to apply the mitigation to. Such a policy risks

being too wide (i.e. not covering malicious scripts injected into "trusted" sites) while being too narrow at the same time (i.e. disabling JIT on otherwise safe sites). Secondly, running without the JIT engine has an impact on the performance of executed scripts, by falling back to the interpreter for all code fragments. As we demonstrate later in our evaluation, disabling the JIT engine can double or even triple script execution times. In other words, a no-JIT solution would protect the JS runtime against JIT vulnerabilities at the cost of significant performance impacts. It is therefore important to provide a sustainable solution that can protect the JIT engine during a vulnerability window.

IV. JITBULL: Go/No-Go policy for JIT Engines

In this section, we present the architecture of our solution JITBULL. We start by introducing our threat model, followed by the definitions of JITBULL's core concepts, and finally a detailed overview of its architecture.

A. Threat Model

JIT compilers are complex software systems, like other categories of compilers. Combined with their tendency to process untrusted code from online sources, they are therefore prone to containing security-critical bugs. As mentioned by Bernhard et al. [18], JIT compiler bugs are logic bugs primarily caused by incorrect optimizations based on bad assumption about the optimized code, or in other words, poor modeling of the effects of code modifications carried out by optimizations.

JITBULL aims to protect a JS runtime during its vulnerability window, a period which extends from the discovery of the vulnerability to the application of a security patch. It specifically does so by addressing the vulnerabilities that arise in its JIT engine's optimization passes. In other words, we focus on vulnerabilities stemming from the JIT compiler. Note that JITBULL doesn't address attacks such as JIT spraying, code reuse, or heap spraying that affect the JS runtime; these JIT vulnerabilities are generally an attack's entry points rather than its root cause.

In our scenario, we assume that each discovered vulnerability comes with a demonstrator code. This code can be produced by the computer scientist who identifies the vulnerability or by the application maintainers. It is the cornerstone of JITBULL's operation; JITBULL only prevents a vulnerability if it has knowledge of its demonstrator code. However, note that VDCs do not need to originate from human experts; one way to use JITBULL is to feed the output of JIT fuzzers directly to its database. In this way, as soon as a crashing code example is detected, JITBULL will be able to automatically prevent similar exploit codes from running.

B. Concept Definitions and Formalization

Before explaining in more detail how JITBULL works, we need to define some notations that we will use throughout our contribution. We denote f to be a JITed function (or a JITed code) of a script being executed by the JS runtime. Let n be the number of optimization passes of a JIT engine (e.g. 32 passes in SpiderMonkey). We denote IR_i^f as the intermediate representation (IR) of the function f when the optimization pass i is applied, with $i \in [0..n]$. In other words, IR_0^f is the initial IR before the application of any optimization pass and IR_n^f the IR after the final optimization pass.

SpiderMonkey's intermediate representations are represented as an instruction graph, comprising an opcode and a list of operands. The operands are references to other instructions. Listing 1 shows an example of IR code, specifically targeting SpiderMonkey's MIR. We can see that an instruction is of the form: *num opcode operand1 operand2*, where each *operand* can be a literal value or a reference to the corresponding numbered instruction. In the given example, instruction number 8 has opcode *boundscheck* and takes as operands instruction number 2 having opcode *unbox* and instruction number 7 having opcode *initializedlength*.

Let us note as Δ_i^f the modifications made by pass i to IR_{i-1}^f to produce IR_i^f for the function f. Loosely speaking, $\Delta_i^f = IR_i^f - IR_{i-1}^f$. We thus define the *JIT DNA* of a JITed function f to be the vector $\Delta^f = (\Delta_1^f, \Delta_2^f, \Delta_3^f, ...\Delta_n^f)$.

Listing 1: An example of IR code, SpiderMonkey's MIR code

```
00 parameter THIS_SLOT
01 parameter 0
02 unbox parameter01 to Int32
03 constant object 7f532cf8e060
04 slots constant03
05 loadslot slots04 452
06 elements loadslot05
07 initializedlength elements06
08 boundscheck unbox02 initalizedlength07
```

C. JITBULL General Architecture and Overview

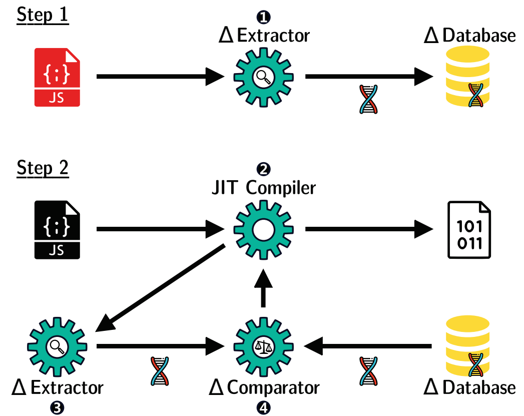

Fig. 2: JITBULL workflow.

To address the issues raised in the previous sections, we introduce JITBULL, a mechanism that is integrated into JIT engines to defend against attacks exploiting JIT vulnerabilities during the vulnerability window. As outlined in the introduction, JITBULL operates by comparing JITed functions from running scripts with JITed functions from vulnerability demonstrator codes. The two-step principle behind JITBULL is depicted in Figure 2.

– **Step 1.** As soon as a demonstrator code of a vulnerability v is available, JITBULL compiles this demonstrator code and extracts for each included JITed function f' its *DNA vector* $\Delta^{f'} = (\Delta_1^{f'}, \Delta_2^{f'}, ...\Delta_n^{f'})$. The resulting DNA vectors are stored in a database (step ❶ in Figure 2). When the security patch for v is applied, the associated DNA vectors can be removed from the DB.

– **Step 2.** When a JS script is executed, each time a function f is JITed (step ❷ in Figure 2), JITBULL extracts its DNA vector Δ^f (step ❸) and compares it with the vectors $\Delta^{f'}$ in the DB (step ❹). If an optimization pass's DNA vector element does not match any known elements of the same pass ($\Delta_i^f \not\approx \Delta_i^{f'}$) then that pass is considered not dangerous. Otherwise, if there exists $\Delta_i^f \approx \Delta_i^{f'}$, then the optimization pass i may be dangerous. Thus, JITBULL builds a list of optimization passes that could be dangerous for a given JS function. If all of the passes in said list can be disabled in the JIT compiler, then JITBULL instructs the JIT engine to compile the function without these optimization passes. Otherwise, if at least one optimization pass in the list cannot be disabled, JITBULL applies a conservative approach and disables the JIT compilation for that particular function. This approach allows for the fine-grained deactivation of specific optimization passes that might be the root cause of the flaw while keeping the JIT engine operational otherwise.

Our next topic is the management of demonstrator codes and their transformation into JITBULL database elements. In the current process of vulnerability reporting and patch production, when a security researcher discovers a potential vulnerability, they submit it to the software maintainer, often including a demonstrator code to show that their described vulnerability is repeatable. The software maintainer has the responsibility to notify users about the presence of the vulnerability before proceeding with the patch production. Two approaches are possible for adding a vulnerability to the database: either it is done by the user by submitting the demonstrator code to JITBULL, or it is done by the software maintainer through an update (the DNA vectors are already extracted and they are sent to the users). To give a demonstrator code to a user is like giving them a weapon that can be used against others. That's why in our model, we recommend that the extraction of *DNA vectors* is done by the maintainer and provided to users as an update for a new vulnerability. Once the patch is available and proposed to the user, the DNA vector associated with the vulnerability is removed from the DB by applying the patch. One should note that the DB might contain at any moment multiple vulnerabilities. Consequently, any JITed code will be checked against all of the VDCs' DNA vectors in the database.

As described previously, JITBULL verifies each optimization passes of a JIT engine (step ❶ in Figure 3). We divide JITBULL's architecture into two components, which we describe in the following sections: a Δ **extractor** (step ❷ in Figure 3), and a Δ **comparator** (step ❸ in Figure 3).

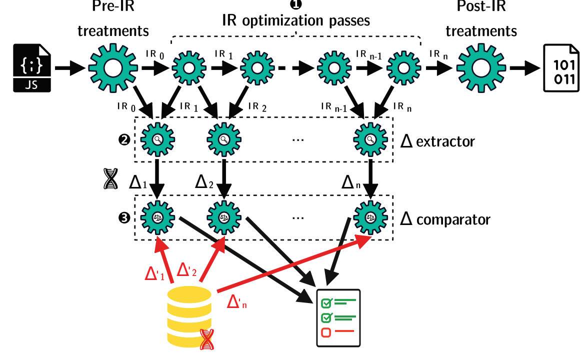

Fig. 3: JITBULL Δ extractor and Δ comparator in action.

D. Δ Extractor

The Δ extractor is used to extract a *JIT DNA vector* when compiling a JS function. It is used in the two steps mentioned in the previous section: in **step 1**, to generate the DNA vectors of the JITed functions of the demonstrator code, and in **step 2**, to generate the DNA vectors of the JITed functions of the executed code. Generating Δ_i^f for an arbitrary JITed code f for pass i involves several phases, which we describe below in Algorithm 1.

The first phase in generating Δ_i^f for pass i involves generating an instruction dependency graph from IR_{i-1} and IR_i that we call G_{i-1} and G_i respectively (function BUILDGRAPH and lines 23 and 24 of Algorithm 1). This transformation simply involves turning the JIT intermediate representation into a form suitable for our DNA vector comparisons. This form is a directed graph that represents all of the function's instruction dependencies starting from a set of root nodes. Any instruction B used as an operand of another instruction A of the intermediate representation is represented as a dependency of A in this form. To generate such a graph G from an intermediate representation IR, for each instruction V of IR having operands, if V does not belong to G then we add it as a root node (line 5). Then, for each operand V' of V, if it is a root node of G, we remove it (line 9), and we add it to G as a dependency of V anyway (line 11). This creates a graph where the root nodes are instructions that are not dependencies of any other instruction. Once the Δ extractor has generated the graphs G_{i-1} and G_i, we next traverse them to extract all possible paths between root nodes and leaves, resulting in a set of *instruction dependency chains* (function MAKECHAINS and lines 25 and 26). In short, for each node present in our dependency graph, we recursively extend the chain set using its dependencies (line 20) until we reach the chain's leaf node (line 18). We note these chains as C_i^j in our algorithm (line 29), with j varying according to the number of chains that can be extracted from the graphs.

The second phase of the Δ extractor involves comparing the equivalent instruction dependency chains C_{i-1}^j and C_i^j in the chain sets of pass i. We compute the *removed sub-chains* $\delta_i^- = \bigcup\{C_{i-1}^j - C_i^j\}$, which represents all sub-chains that

were removed by the pass (line 30); and the *added sub-chains* $\delta_i^+ = \bigcup\{C_i^j - C_{i-1}^j\}$, which represents the sub-chains that were added (line 31). For example, consider two chains of pass i: $C_{i-1}^j = A \rightarrow B \rightarrow C \rightarrow D$ and $C_i^j = B \rightarrow C \rightarrow E$. Then $\delta_i^- = \{A \rightarrow B, C \rightarrow D\}$ and $\delta_i^+ = \{C \rightarrow E\}$. Finally, Δ_i^f is simply the pair (δ_i^-, δ_i^+) (line 33).

Algorithm 1 Δ extractor with a function f for pass i.

Input: IR_{i-1}^f, IR_i^f
Output: Δ_i^f

```
 1: function BUILDGRAPH(IR)
 2:   G ← ∅
 3:   for all V ∈ IR having OPERANDS(V) ≠ ∅ do
 4:     if V ∉ G then
 5:       Add V to G as a root node: V ∈ ROOTS(G)
 6:     end if
 7:     for all V′ ∈ OPERANDS(V) do
 8:       if V′ ∈ ROOTS(G) then
 9:         Remove V′ from G
10:       end if
11:       Add V′ to G as a dependency of V
12:     end for
13:   end for
14:   return G
15: end function

16: function MAKECHAINS(V_1 → ··· → V_N)
17:   if DEPENDENCIES(V_N) = ∅ then
18:     return {V_1 → ··· → V_N}
19:   else
20:     return ⋃ MAKECHAINS(V_1 → ··· → V_N → W) |
      W ∈ DEPENDENCIES(V_N)
21:   end if
22: end function

23: G_{i−1} ← BUILDGRAPH(IR^f_{i−1})
24: G_i ← BUILDGRAPH(IR^f_i)
25: Chains_{i−1} ← ⋃ MAKECHAINS(R) |
    R ∈ ROOTS(G_{i−1})
26: Chains_i ← ⋃ MAKECHAINS(R) | R ∈ ROOTS(G_i)
27: δ_i^- ← ∅
28: δ_i^+ ← ∅
29: for all (C^j_{i−1}, C^j_i) ∈ Chains_{i−1} × Chains_i do
30:   δ_i^- ← δ_i^- ∪ {C^j_{i−1} − C^j_i}
31:   δ_i^+ ← δ_i^+ ∪ {C^j_i − C^j_{i−1}}
32: end for
33: Δ^f_i ← (δ_i^-, δ_i^+)
```

E. Δ Comparator

The Δ comparator is used to compare a DNA vector of a JITed function with those of exploits in the DB. We can observe in Figure 3, step ❸, that the Δ comparator uses the DNA vectors produced by the Δ extractor and those from the DB to compute similarities. The Δ comparator operates according to Algorithm 2. Let's consider f as a JITed function from an arbitrary script and f' a JITed function from a vulnerability demonstrator code. Therefore, the DNA vectors of f and f' are $\Delta^f = (\Delta_1^f, \Delta_2^f, ...\Delta_n^f)$ and $\Delta^{f'} = (\Delta_1^{f'}, \Delta_2^{f'}, ...\Delta_n^{f'})$ respectively. The goal of our Δ comparator is to check if $\Delta_i^f \approx \Delta_i^{f'}$.

For any JITed function f, the Δ comparator extracts from the JITBULL DB all vulnerability DNA vectors $\Delta^{f'}$ stored within and compares them to Δ^f. Recall that Δ_i^f is made up of the set of removed sub-chains δ_i^{f-} and the set of added sub-chains δ_i^{f+} for optimization pass i (lines 14 and 15 of Algorithm 2). Therefore, the Δ comparator needs to compare all the removed sub-chains δ_i^{f-} of f with those $\delta_i^{f'-}$ of f' for every demonstrator code in the database (function COMPARECHAINS and line 14). In order to exclude insignificant similarities between δ_i^{f-} and $\delta_i^{f'-}$, we consider that $\delta_i^{f-} \not\approx \delta_i^{f'-}$ if the number of sub-chains in common is below a predefined threshold Thr (line 7, first condition). Otherwise, we consider that $\delta_i^{f-} \approx \delta_i^{f'-}$ only if the actual number of sub-chains in common $EqChains$ is above the maximum possible number of sub-chains in common $MaxEqChains$ multiplied by a predefined ratio setting $Ratio$ (line 7, second condition). We chose a threshold of 3 and a ratio of 50% to optimize for a high detection rate, thanks to our low overhead in case of a false positive detection. $MaxEqChains$ is derived from the sub-chains sets compared (line 2). The same comparison operation is applied to the added sub-chains sets δ_i^{f+} and $\delta_i^{f'+}$ (function COMPARECHAINS and line 15). If either $\delta_i^{f-} \approx \delta_i^{f'-}$ or $\delta_i^{f+} \approx \delta_i^{f'+}$, then we consider that $\Delta_i^f \approx \Delta_i^{f'}$ and we add the pass i to the list of disabled passes $DisPass$ (line 17).

To summarize, the above algorithm defines, given a database of VDCs' DNA vectors and a JITed function f, a list of all the optimization passes that match the VDCs in its database. JITBULL first attempts to disable all passes if possible; if not, it completely disables the JIT engine, but only for the particular function f.

V. IMPLEMENTATION

We implemented JITBULL on the SpiderMonkey JS runtime of Firefox 65. However, JITBULL's principle can be applied to all JIT engines that implement optimization passes, meaning it is applicable to all modern browser JIT engines. In Section II, we provided an overview of the general functioning of SpiderMonkey. In this section, we will detail the implementation of JITBULL in SpiderMonkey. We directly integrated JITBULL into the IonMonkey code, specifically within the optimization passes for MIR code (step ❹ in Figure 1). Initially, we planned to include JITBULL as an extension to SpiderMonkey without the need for integration into the JIT engine. However, to minimize the performance overhead of JITBULL, we chose to directly modify the JIT engine. As a result, our implementation is in C++, the same language as SpiderMonkey.

JITBULL implementation consists of 6,000 lines of code (LOC), mainly located within the `OptimizeMIR` function, which is the function in which optimization passes are per-

Algorithm 2 Comparing a DNA vector Δ^f of a JITed function f with VDC DNA vectors $\Delta^{f'}$ in the JITBULL database DB.

Input: Δ^f, DB, $Ratio$, Thr
Output: $DisPass$

1: **function** COMPARECHAINS($\delta^f, \delta^{f'}, Ratio, Thr$)
2: $MaxEqChains = \text{MIN}(|\delta^f|, |\delta^{f'}|)$
3: $EqChains \leftarrow 0$
4: **for all** $C \in \delta^f \cap \delta^{f'}$ **do**
5: $EqChains \leftarrow EqChains + 1$
6: **end for**
7: **return** $EqChains \geq Thr$ **and** $EqChains \geq Ratio * MaxEqChains$
8: **end function**

9: $DisPass \leftarrow \emptyset$
10: **for all** $\Delta^{f'} \in DB$ **do**
11: **for all** $(\Delta_i^f, \Delta_i^{f'}) \in \Delta^f \times \Delta^{f'}$ **do**
12: $(\delta_i^{f-}, \delta_i^{f+}) \leftarrow \Delta_i^f$
13: $(\delta_i^{f'-}, \delta_i^{f'+}) \leftarrow \Delta_i^{f'}$
14: $\delta^-_{similar} \leftarrow$ COMPARECHAINS($\delta_i^{f-}, \delta_i^{f'-}, Ratio, Thr$)
15: $\delta^+_{similar} \leftarrow$ COMPARECHAINS($\delta_i^{f+}, \delta_i^{f'+}, Ratio, Thr$)
16: **if** $\delta^-_{similar}$ **or** $\delta^+_{similar}$ **then**
17: $DisPass \leftarrow DisPass \cup i$
18: **end if**
19: **end for**
20: **end for**

formed. This function returns either `SUCCESS` or `FAILURE`: if `OptimizeMIR` returns `SUCCESS` the compilation will go on, and the machine code generated will be used by the JS runtime, otherwise, if the return value equals `FAILURE`, the compilation is abandoned and the runtime continues with interpreted code. To implement the Δ extractor, we modified the code of the `OptimizeMIR` function to generate a Δ^f vector, i.e. a Δ_i^f for each optimization pass i, following the two-phase procedure outlined in section IV-D. At the end of `OptimizeMIR`, we added the Δ comparator code to compare the *DNA vector* of the JITed code with the *DNA vectors* from the DB. Note that JITBULL preloads the VDC DNA vector database into memory as soon as the JS runtime is loaded. If the DNA vector database is empty, JITBULL will not extract DNA vectors for running scripts. This allows us to have no overhead when there are no vulnerabilities in the DB (more details in Section VI).

The Δ comparator builds the list of dangerous optimization passes after DNA vector comparison, i.e. passes that should be disabled. Three scenarios are possible: (1) the list is empty, (2) all the optimization passes in the list can be disabled, and (3) there is at least one optimization pass in the list that is mandatory and cannot be disabled. SpiderMonkey provides means to disable optimization passes, but some passes cannot be disabled and are mandatory. To handle all these three scenarios, we introduce a boolean called *Recompile* that guides us in the decision-making process. The boolean is initialized to `False`. In scenario (1), if there are no dangerous passes, we return `SUCCESS` for `OptimizeMIR` and keep `Recompile` set to `False`. The next step is for the JIT engine to complete its activity and hand it over to the runtime, which will use the code generated. For scenario (2), where there are dangerous passes that can be fully deactivated, we return `FAILURE` with `OptimizeMIR` and set `Recompile` to `True`. We instrumented the JS runtime behavior to always retry a compilation when `Recompile` is `True`. Therefore, SpiderMonkey retries a compilation but this time with the dangerous passes disabled. In scenario (3), we return `FAILURE` from our function and let the JS runtime continue its execution as usual with no JITed code. As our evaluation in the next section will demonstrate, these modifications to the JIT engine do not generate a significant overhead on script performance and the observed performance with JITBULL is better than completely disabling the JIT engine.

We believe that JITBULL can be implemented in other JIT engines such as TurboFan, which also relies on complex optimization passes to generate machine code [19]. As we specified in Section III, TurboFan also faces bugs related to its optimizer. Therefore, it is possible and interesting to implement JITBULL in such a system by incorporating the computation of IR differences between optimization passes.

VI. Evaluation

In this section, we present our evaluation of JITBULL. We focus on two main aspects: security and performance. The security evaluation of JITBULL aims to verify if JITBULL ensures the security of the JIT engine against variants of a vulnerability for which a demonstrator code is present in its database. The questions we will answer in the security evaluation will be:

– Can JITBULL identify dangerous optimization passes corresponding to a demonstrator code?
– Can JITBULL disable these dangerous passes efficiently?
– Are the malicious codes harmless after JITBULL operations?

Regarding performance, the objective of our evaluation is to answer the following questions:

– What is the overhead of JITBULL on script performance?
– What is JITBULL's false positive rate? And how does it impact performance?
– How does the performance of JITBULL compare with completely disabling the JIT engine?

A. Experimental Setup

a) Hardware and software: We evaluated JITBULL on a computer with the characteristics described in Table II. The computer was fully dedicated to the JS runtime during our evaluation.

b) Benchmarks: Our evaluation relies on two micro-benchmarks and the Octane benchmark suite. We implemented a first micro-benchmark called *Microbench1* that performs an arithmetic operation on variables within a for loop; and a second benchmark that does the same but manipulates the size of an array, called *Microbench2*. Our evaluation is also

TABLE II: Hardware configuration.

Component	Characteristics
CPU	11th Gen Intel Core i7-11850H 2.50GHz
Memory	32 GB
Ethernet	USB Ethernet 1Gbit/s
Storage	Micron 2300 NVMe 1.0 TB
OS	Ubuntu 20.04.6 LTS

conducted with the Octane benchmark suite [9]. Octane is a benchmark that measures a JavaScript runtime's performance by running a suite of tests representative of certain use cases in JavaScript applications. We used these benchmarks to determine the impact of JITBULL on application performance and also to assess the error rate of JITBULL.

B. JITBULL security evaluation

We obtained vulnerability demonstrator codes for the evaluation of JITBULL. After research, we were able to obtain the demonstrator codes for the following four vulnerabilities in IonMonkey: `CVE-2019-9791` [20], `CVE-2019-9810` [21], `CVE-2019-11707` [22] and `CVE-2019-17026` [5]. To validate these demonstrator codes, we executed these exploits on the versions of SpiderMonkey affected by the vulnerability. Out of these 4 vulnerabilities, 2 lead to a crash (the first two in our list), and the last two result in the execution of a payload. Subsequently, we tested whether JITBULL can detect exploit variants from these demonstrator codes. To validate the proper functioning of our solution, for the first two vulnerabilities, we ensure that there is no crash of the JS engine, and for the other two, we check that the shellcode is not executed.

a) ***Variants of vulnerability*** **`CVE-2019-17026`***:* We could only find one vulnerability having multiple demonstrator code variants, meaning two implementations of the same vulnerability produced by different developers [23] and [24], and this is the vulnerability `CVE-2019-17026`. These codes exploit the same security flaw but are implemented by different developers. Our evaluation involves placing one of the implementations of CVE-2019-17026 in JITBULL's DB, then checking whether JITBULL can detect the other implementation as dangerous and disable the corresponding optimization passes. The execution of the variant was indeed perceived as dangerous by JITBULL. Furthermore, JITBULL managed to detect that the common optimization modification between the two implementations was the suppression of the `BoundCheck` in the GVN optimization phase. Thus, JITBULL disabled this optimization pass and therefore neutralized the vulnerability's variant.

b) ***Variant generation****:* Without having demonstrator code variants available for testing, we decided to generate variants to evaluate JITBULL's ability to detect exploit similarities. The variants were generated using 4 approaches:

- *Renaming script variables.* The first approach involves renaming the variables in our demonstrator codes. The goal here is to demonstrate that JITBULL is not tied to a syntactic analysis of the script. To rename the variables, we used Terser [25], a JavaScript code minifier. It reduces variable names, eliminates whitespace and comments, and discards unused code. We used it to rename the variables of the demonstrator codes and thus generated what we consider a variant.
- *Minifying code.* We also used Terser to generate new exploit variants by minifying demonstrator codes. In this case, the use of Terser involved code factorization and compression to make the variant as small as possible.
- *Mixing independent instructions and adding JITed functions.* Our third approach was manual, meaning that code modification was done by us. This approach involved manually reordering independent instructions in the demonstrator codes while keeping the exploit's functionality. The goal here is to validate that JITBULL can identify modifications made by optimizations even if the order of instructions is no longer the same. Additionally, we added functions in the codes that are JITed but do not participate in the vulnerability setup.
- *Adding sub-functions.* In this final approach, the idea is to split the JITed functions used to exploit the vulnerability into sub-functions. This operation increases the number of JITed functions since the JITed code is divided into multiple functions. The goal is to obfuscate the step of exploitation in the demonstrator code.

For each of the four aforementioned vulnerabilities, we generated variants using these four approaches and verified that the security vulnerability was still exploitable with the variants. Our experimental procedure is as follows: for each vulnerability, we only integrated its demonstrator code into the database, and we executed the four generated variants. The goal is to validate if the vulnerability was still exploitable under JITBULL. From these executions, we observed that none of these variants were able to exploit the vulnerability once the demonstrator code has been added to the database. Thus, for all the variants, JITBULL was always able to disable all the passes that could potentially be the cause of the vulnerability, and that JITBULL protected the system against all these vulnerabilities. For example, with the variants of vulnerability `CVE-2019-17026`, the GVN optimization pass was consistently detected as potentially dangerous for the variants, and completely disabled.

c) ***Evaluating JITBULL precision****:* The goal of our next evaluation is to determine the error rate of JITBULL, i.e., the number of times JITBULL considers a code malicious when it is not, given a set of demonstrator codes in its DB. For this purpose, we used the Octane benchmark suite [9] which consists of a series of test programs representing different use cases by JS applications. In this evaluation, Octane serves as a repertoire of safe applications. We tested two different scenarios in this evaluation: firstly, we included only one demonstrator code in the DB, that of the `CVE-2019-17026` vulnerability, and for the second, all demonstrator codes from the 4 vulnerabilities are in the DB. Secondly, we collected for each benchmark the number of JITed functions (called Nr_{JIT}), the number of JITed functions with one or more passes disabled (called Nr_{DisJIT}), and finally, the number of JITed function with the JIT engine disabled (called Nr_{NoJIT}).

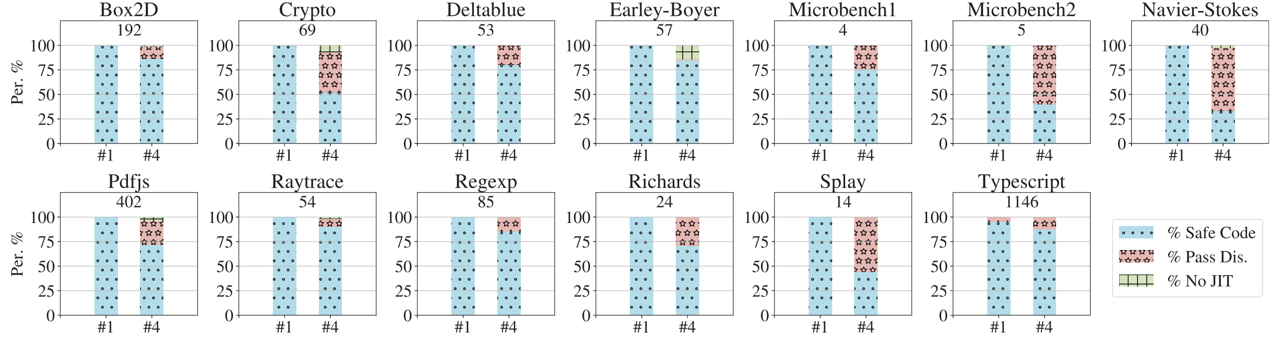

Fig. 4: False positive rates of JITBULL on a set of harmless benchmarks. "#1" and "#4" represent the number of VDCs installed into the JITBULL database (1 versus 4 VDCs respectively).

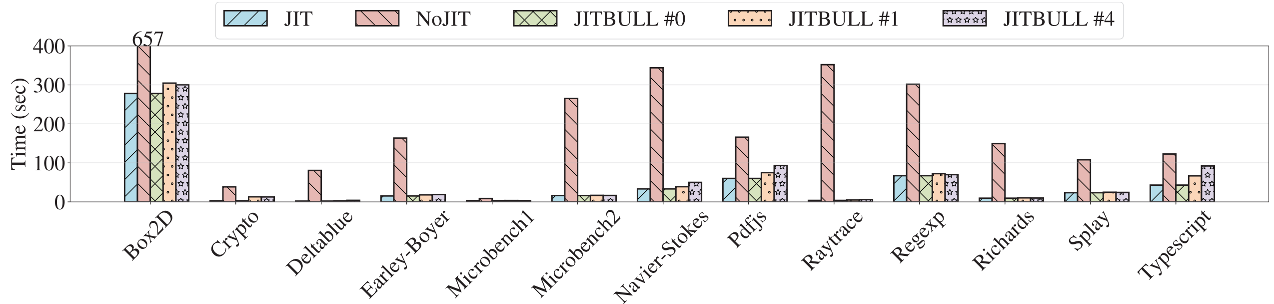

Fig. 5: Execution times for disabled JIT, normal JIT and JITBULL on a set of benchmarks.

Figure 4 shows the results obtained. The graph represents the proportion of functions considered safe (named $\%Safe\ Code$ on the figure), the proportion of functions with at least 1 pass deactivated ($\%Pass\ Dis.$), and finally, the proportion of code with the JIT engine disabled ($\%No\ JIT$). These proportions are calculated using the following formulas:

$$\%\ Pass\ Dis. = \frac{Nr_{DisJIT} \times 100}{Nr_{JIT}}$$

$$\%\ No\ JIT = \frac{Nr_{NoJIT} \times 100}{Nr_{JIT}}$$

The numbers on the bars in the figures represent the number of JITed functions when the script is executed without JITBULL. This provides an idea of how many times JITBULL needs to perform Δ extraction and comparison.

With a DB containing a single demonstrator code, the proportion of functions with optimization passes disabled is very low, varying from 0% to 5% (bar with the xlabel $\#1$). We observe practically no JITBULL match for all Octane benchmarks except TypeScript, where it shows a similarity with a vulnerability in the DB (`CVE-2019-17026`). Given that JITBULL does not perform any analysis of the execution context and focuses on analyzing changes to the IR, it is perfectly normal for similarities to exist between VDCs and innocent code. Nevertheless, JITBULL does not completely disable the JIT engine; it only disables the optimization passes, which leads to interesting performance results (more details in the next section). Moreover, the JIT engine never gets completely disabled with 1 vulnerability in the DB.

To recall, vulnerabilities are installed into the DB only during a vulnerability window, therefore the DB will likely contain most of the time 1 or 2 vulnerabilities. This result is satisfying as it corresponds to the most likely use case for JITBULL. With 4 vulnerabilities in the DB, the results are not the same. We can observe that the proportion of JITed functions with deactivated optimization passes varies, from 10% with some benchmarks (Box2D, Deltablue, Raytrace, TypeScript) to 65% with others (Splay, Navier-Stokes). Nevertheless, as we will see in the next section, despite this seemingly high false positive rate with a large number of vulnerabilities in the database, the impact on performance is quite reasonable.

C. Performance evaluation of JITBULL

In this section, we are interested in the overhead of JITBULL on script performance. As with the previous evaluations, we used the Octane benchmark and our two micro-benchmarks and measured the execution time in different scenarios: (1) No JITBULL (named JIT on the figure), (2) JIT engine is completely disabled ($NoJIT$), (3) JITBULL with no vulnerability in the DB ($\#0$), (4) a single vulnerability in the DB ($\#1$) and (5) 4 vulnerabilities in the DB ($\#4$).

Figure 5 presents the obtained results. The first observation in the case with 0 vulnerabilities (bars labelled $\#0$) is that JITBULL induces no overhead compared to the normal JIT engine. This is simply because JITBULL mechanics is used

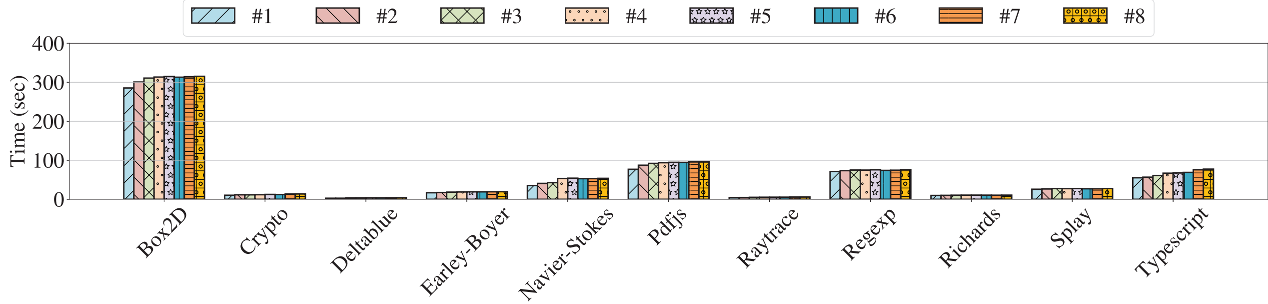

Fig. 6: Scalability evaluation of JITBULL, with various numbers of vulnerabilities in the DB, ranging from #1 to #8.

if there are vulnerabilities in the DB as we described in Section V. If there are no vulnerabilities, no processing is done by JITBULL, which explains the zero overhead. We can also observe that, overall, the degradation induced by JITBULL is between 1% and 20% with 1 to 4 vulnerabilities in the DB. We can note that with benchmarks like Box2D, Navier-Stokes, Pdfjs, and TypeScript, which involve a large number of JITed functions (see Figure 4), the overhead of JITBULL is most significant, reaching up to 20%. This is expected because the more JITed functions there are, the more work JITBULL has to do for DNA extraction and comparison. In contrast, with benchmarks having fewer JITed functions, the overhead is less significant, around 1%. Note that this overhead is much lower than what is observed when the JIT engine is disabled. Finally, with no JIT engine, the overhead ranges from 136% for Box2D to 3700% for DeltaBlue, which is very substantial. This demonstrates that the overhead induced by JITBULL is acceptable and far below the overhead of completely disabling the JIT engine. Moreover, it's important to note that JITBULL is designed to be used only during the vulnerability window. Therefore, the introduced overhead is temporary and will only be present until the patch is produced and applied.

D. JITBULL scalability evaluation

In this section, the goal is to evaluate the overhead of JITBULL with an increasing number of vulnerabilities in the DB. Before presenting the results, it is important to analyze the maximum number of vulnerabilities that could be present simultaneously in JITBULL. We focused on SpiderMonkey vulnerabilities of 2019 (see Table I) and studied the report date and patch availability date from the Mozilla Bug tracker. The idea is to analyze the overlapping period between vulnerabilities and therefore estimate the maximum number of vulnerabilities in JITBULL DB during that year. We found that at most 2 vulnerabilities throughout the year 2019 have overlapping time intervals, which are `CVE-2019-9813` and `CVE-2019-9810`. Therefore, during 2019, maximum of 2 vulnerabilities would have been in JITBULL DB. This allows us to conclude that in general, JITBULL will not have many vulnerabilities in its DB.

For the scalability analysis, we implemented 4 other vulnerabilities demonstrator codes based on the Mozilla Bug tracker descriptions. These vulnerabilities are `CVE-2019-9792`, `CVE-2019-9795`, `CVE-2019-9813` and `CVE-2020-26952`. We added them to JITBULL DB and reran the evaluation with the Octane benchmark suite. Fig 6 presents the execution times with various numbers of vulnerabilities in the database. We analyzed the degradation when having 1 vulnerability in the DB (noted #1) vs 8 vulnerabilities in the DB (noted #8). We observed that the maximum overhead is 22% with Typescript and the minimum is 5% with Splay, which we considered acceptable knowing JITBULL is a temporary solution used only until the patch is available and applied. Furthermore, we notice that for all the benchmarks, this overhead tends to stabilize beyond 4 vulnerabilities in the DB.

VII. Related work

In this section, we present an overview of the techniques employed to safeguard JIT engines. We will begin by introducing the various classes of attacks that target JIT engines and the responses proposed by existing research. Lastly, we will contextualize our work in relation to the current existing research works.

Over the years, JIT engines have turned to a central component of JS runtime, leading to the development of numerous approaches to bypass the security measures adopted by JITs [12]. Heap spray [26], [27] is a widely used attack approach against JIT. It involves injecting malicious code into the heap and redirecting the execution flow to it. Because the exact location of the injected code is not predictable, traditional heap-spraying attacks require the injection of a substantial amount of executable code to enhance the chances of success. Various defense mechanisms have been developed to counter heap-spraying attacks. The first approach involves the detection of shellcode by identifying common patterns associated with it [28]. The second defense approach involves analyzing the control flow structure of heap objects to recognize common structures employed in heap-spraying attacks. For instance, NOZZLE [26] disassembles potential x86 instructions within the object and constructs a control flow graph (CFG). Based on this CFG, NOZZLE computes the surface area of the entire heap and identify heap-spraying attack if most of the object redirects to a specific memory region. The third

mechanism and popular countermeasure for heap-spraying is W $\oplus$ X (Writable xor eXecutable), also known as DEP (Data Execution Protection) [29]. This mechanism involves forcing any memory region to be either executable or writable, but never both. Successfully writing malicious code to a writable region becomes pointless for an attacker since it becomes impossible to execute that code later, rendering heap-spray ineffective. In response, a second class of attacks emerged, known as code-reuse.

The general principle of any code reuse attack involves redirecting the logical program flow to instructions already present in memory and subsequently utilizing those instructions to introduce alternative program logic. By combining individual harmless pieces of existing code, the attacker manages to reconstruct malicious behavior. As an illustration, ROP (Return-Oriented Programming) [30] assembles code snippets situated at the end of functions by leveraging the return instruction and writing their addresses to the stack. An approach to mitigate this class of attack involves memory randomization, specifically using ASLR (Address Space Layout Randomization) [31]. The fundamental concept of ASLR involves a new stack memory allocator that introduces a random pad for stack objects. This results in the relocation of the start address of an executable between successive runs of the application. Consequently, an adversary must guess the location of the functions and instruction sequences needed for the successful deployment of her code reuse attack.

JIT-Spray [12], [32], [33] is a new class of attack that bypasses DEP and ASLR. It leverages JIT compilation of expressions with constant values from a high-level language into native code, allowing for the injection of malicious code bytes at runtime. This bypasses DEP because data is (indirectly) injected as code. Moreover, if the attacker successfully creates numerous regions of this code, the locations become predictable. Hence, by dispersing numerous code regions, they can anticipate the address of one region to circumvent ASLR. Ultimately, only control over the instruction pointer is needed to redirect the control flow to the injected code. Certain works propose countermeasures against JIT spraying. The aim of these studies is to prevent malicious code from being present in a program's variables. An example would be RIM [34], a technique that obfuscates arithmetic operations in the JIT-compiled code, thereby preventing attackers from reusing the native code to construct malicious code. However, all these vulnerabilities presented by the JIT engine are exploitable only if it is possible to obtain read and write primitives on the memory of the code generated by the JIT engine.

JIT compilers, like other compilers, are complex software systems. Therefore, it is natural that they may contain security-critical bugs. As mentioned in [18], these bugs often arise from subtle logic errors and miscalculations that result from optimization passes in JIT engines, and they persistently escape state-of-the-art testing methods. A popular approach to prevent these vulnerabilities remains fuzzing [18], [35], [36], [37]. The goal is to detect vulnerabilities/bugs in the software before the release. Unfortunately, despite all these efforts, vulnerabilities are still present in JIT engines. JIT compiler bugs are mainly caused by incorrect speculation or wrong optimization based on logic errors, and these bugs are challenging to detect with current fuzzers.

Several other research works have focused on securing the JS runtime. Notably, NOJITSU [38] proposes isolation of JIT engine components such as bytecode interpreters with controlled accesses to prevent the execution of unauthorized code. This is accomplished by utilizing hardware components like Intel's Memory-Protection Keys. While this solution effectively prevents unauthorized code execution, it does not address vulnerabilities that do not involve code execution as their primary purpose. For example, an attack aiming to crash the runtime due to JIT's poor optimization can succeed because the generated code is stored in executable memory. Furthermore, using such a solution requires hardware extension not available to all users. BrowserShield [39] addresses security during the vulnerability window. It involves rewriting page scripts to prevent malicious operations from being hidden. This rewriting relies on policies derived from known vulnerabilities. The limitations of this solution lie in the complexity of the policies, which requires expertise to implement, and imposes overhead on page execution due to the necessity of regenerating the entire page's script. On the other hand, JShield [40] addresses drive-by download attacks by generating a signature for vulnerabilities based on opcodes and comparing this signature with all downloaded JS scripts. The drawback of this approach again lies in the expertise required to generate the signature, which involves several complex steps.

JITBULL, the solution we propose, complements all the existing protection mechanisms for the JIT engine. Knowing that fuzzers are not capable of detecting all types of bugs, it is important to implement tools that will secure the JIT engine when a vulnerability is detected. Applying a patch in this case remains the only solution, but this solution does not guarantee the security of the JIT engine during the vulnerability window.

VIII. Conclusion

We presented JITBULL, a system that improves the security of web browsers by addressing JIT engines vulnerabilities in JS runtimes. We explained that JITBULL scope applies to known vulnerabilities for which a code demonstrator is available and that have not been patched yet. We showed that this could be achieved thanks to a two-steps strategy that relies on indirect comparison between code demonstrators and the running code. We discussed the generalization of our solution that can be applied to any recent JS runtime. We tested it on variants of demonstrator codes and showed that it effectively neutralized the exploitation of the vulnerabilities and noticed that the false positive rate was very much acceptable in a typical case. We also found that JITBULL's overhead was negligible compared to using the vanilla JIT engine. We emphasized that JITBULL is a tool that does not replace, and instead complements existing solutions to secure the JIT engine.

ACKNOWLEDGEMENTS

This work was supported by the French *Agence nationale de la recherche* under the projects ANR WalkIn (ANR-20-CE25-0005) and LabEx CIMI (11-LABX-0040). We would like to thank Tu Dinh Ngoc for his feedback on the paper and our shepherd Yu Liang and the anonymous reviewers for their helpful feedbacks.

REFERENCES

[1] A. Wirfs-Brock and B. Eich, "JavaScript: the first 20 years," *Proceedings of the ACM on Programming Languages*, vol. 4, no. HOPL, pp. 1–189, 2020.

[2] M. Heller, "What is Node.js? the JavaScript runtime explained," 2017.

[3] M. P. Plezbert and R. K. Cytron, "Does "just in time" = "better late than never"?" in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 120–131. [Online]. Available: https://doi.org/10.1145/263699.263713

[4] M. Selakovic and M. Pradel, "Performance issues and optimizations in JavaScript: an empirical study," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 61–72.

[5] "CVE-2019-17026." [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-17026

[6] R. Hiesgen, M. Nawrocki, T. C. Schmidt, and M. Wählisch, "The race to the vulnerable: Measuring the Log4j shell incident," vol. abs/2205.02544, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2205.02544

[7] T. D. Ngoc, B. Teabe, A. Tchana, G. Muller, and D. Hagimont, "Mitigating vulnerability windows with hypervisor transplant," in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 162–177. [Online]. Available: https://doi.org/10.1145/3447786.3456235

[8] R. Kellogg and E. A. Goldman, "Review of the spider monkeys," *Proceedings of the United States National Museum*, 1944.

[9] V8 project authors, "Octane: The JavaScript benchmark suite for the modern web." [Online]. Available: https://github.com/chromium/octane

[10] N. Pierron, "IonMonkey: Optimizing away," 2014. [Online]. Available: https://blog.mozilla.org/javascript/2014/07/15/ionmonkey-optimizing-away/

[11] "National Vulnerability Database." [Online]. Available: https://nvd.nist.gov/vuln

[12] R. Gawlik and T. Holz, "SoK: Make JIT-spray great again," in *WOOT @ USENIX Security Symposium*, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:52046503

[13] "CVE-2020-1380." [Online]. Available: https://www.trendmicro.com/en_us/research/20/h/cve-2020-1380-analysis-of-recently-fixed-ie-zero-day.html

[14] "Bugzilla." [Online]. Available: https://bugzilla.mozilla.org/home

[15] W. Arbaugh, W. Fithen, and J. McHugh, "Windows of vulnerability: a case study analysis," *Computer*, vol. 33, no. 12, pp. 52–59, 2000.

[16] "V8 security setting." [Online]. Available: https://support.google.com/chrome/a/answer/7679408#chromeBrsrG122

[17] "Browse more safely with Microsoft Edge." [Online]. Available: https://learn.microsoft.com/en-us/deployedge/microsoft-edge-security-browse-safer

[18] L. Bernhard, T. Scharnowski, M. Schloegel, T. Blazytko, and T. Holz, "JIT-Picking: Differential fuzzing of JavaScript engines," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 351–364. [Online]. Available: https://doi.org/10.1145/3548606.3560624

[19] A. Parravicini and R. Mueller, "The cost of speculation: Revisiting overheads in the V8 JavaScript engine," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*, 2021, pp. 13–23.

[20] "CVE-2019-9791." [Online]. Available: https://github.com/tunz/js-vuln-db/blob/master/spidermonkey/CVE-2019-9791.md

[21] "CVE-2019-9810." [Online]. Available: https://github.com/xuechiyaobai/CVE-2019-9810-PoC/

[22] "CVE-2019-11707." [Online]. Available: https://github.com/vigneshsrao/CVE-2019-11707/blob/master/exploit.js

[23] "CVE-2019-17026 implementation 1." [Online]. Available: https://github.com/lsw29475/CVE-2019-17026/tree/main

[24] "CVE-2019-17026 implementation 2." [Online]. Available: https://github.com/maxpl0it/CVE-2019-17026-Exploit/blob/master/calc.html

[25] Terser, "A JavaScript parser and mangler/compressor toolkit for ES6+." [Online]. Available: https://github.com/terser/terser

[26] P. Ratanaworabhan, V. B. Livshits, and B. G. Zorn, "NOZZLE: A defense against heap-spraying code injection attacks." in *USENIX security symposium*, 2009, pp. 169–186.

[27] Y. Ding, T. Wei, T. Wang, Z. Liang, and W. Zou, "Heap taichi: Exploiting memory allocation granularity in heap-spraying attacks," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 327–336. [Online]. Available: https://doi.org/10.1145/1920261.1920310

[28] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda, "Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 6th International Conference, DIMVA 2009, Como, Italy, July 9-10, 2009. Proceedings 6*. Springer, 2009, pp. 88–106.

[29] Microsoft Corporation, "Data execution prevention." [Online]. Available: https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc738483(v=ws.10)?redirectedfrom=MSDN

[30] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 30–40. [Online]. Available: https://doi.org/10.1145/1966913.1966919

[31] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133)*. IEEE, 1997, pp. 67–72.

[32] P. Chen, R. Wu, and B. Mao, "JITSafe: a framework against just-in-time spraying attacks," *IET Information Security*, vol. 7, no. 4, pp. 283–292, 2013.

[33] M. Athanasakis, E. Athanasopoulos, M. Polychronakis, G. Portokalidis, and S. Ioannidis, "The devil is in the constants: Bypassing defenses in browser JIT engines." in *NDSS*, 2015.

[34] R. Wu, P. Chen, B. Mao, and L. Xie, "RIM: A method to defend from JIT spraying attack," in *2012 Seventh International Conference on Availability, Reliability and Security*, 2012, pp. 143–148.

[35] H. Han, D. Oh, and S. K. Cha, "CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines." in *NDSS*, 2019.

[36] S. Lee, H. Han, S. K. Cha, and S. Son, "Montage: A neural network language model-guided JavaScript engine fuzzer," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2613–2630. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/lee-suyoung

[37] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing JavaScript engines with aspect-preserving mutation," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1629–1642.

[38] T. Park, K. Dhondt, D. Gens, Y. Na, S. Volckaert, and M. Franz, "NoJITsu: Locking down javascript engines." in *NDSS*, 2020.

[39] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, "Browsershield: Vulnerability-driven filtering of dynamic html," *ACM Trans. Web*, vol. 1, no. 3, p. 11–es, sep 2007. [Online]. Available: https://doi.org/10.1145/1281480.1281481

[40] Y. Cao, X. Pan, Y. Chen, and J. Zhuge, "Jshield: towards real-time and vulnerability-based detection of polluted drive-by download attacks," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 466–475. [Online]. Available: https://doi.org/10.1145/2664243.2664256