TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly

William Fu, Raymond Lin, Daniel Inge
Harvard University
{wfu, rlin, dinge}@college.harvard.edu

***Abstract*—WebAssembly (wasm) has recently emerged as a promisingly portable, size-efficient, fast, and safe binary format for the web. As WebAssembly can interact freely with JavaScript libraries, this gives rise to a potential for undesirable behavior to occur. It is therefore important to be able to detect when this might happen. A way to do this is through taint tracking, where we follow the flow of information by applying taint labels to data. In this paper, we describe TaintAssembly, a taint tracking engine for interpreted WebAssembly, that we have created by modifying the V8 JavaScript engine. We implement basic taint tracking functionality, taint in linear memory, and a probabilistic variant of taint. We then benchmark our TaintAssembly engine by incorporating it into a Chromium build and running it on custom test scripts and various real world WebAssembly applications. We find that our modifications to the V8 engine do not incur significant overhead with respect to vanilla V8's interpreted WebAssembly, making TaintAssembly suitable for development and debugging.**

***Keywords*—WebAssembly, Taint Tracking, V8, Chromium**

I. Introduction

There has been a trend toward more demanding applications on the web. This necessitates a delivery format that is portable across platforms, compact in size, fast to execute, and comparatively safe to use. JavaScript is too slow for most complex applications, and it is inconvenient as a compilation target. Various solutions have been proposed for this problem, including `asm.js`, which has gained reasonable popularity as of late [1]. The newest, cross-browser solution is WebAssembly (wasm) [2]. Although it is executed in a partially restricted environment, wasm can interact freely with JavaScript libraries and functions [3], providing a vector for wasm to interact with other areas of the users' computer in unpredictable and potentially undesirable ways.

With this in mind, it would be nice for developers to be able to detect possible vulnerabilities and unexpected behavior in their wasm programs. In particular, it would be useful to track data flows between wasm programs and JavaScript, as that is the primary way in which wasm is exposed. This motivates dynamic taint tracking, in which we mark and follow the flow of information through a system.

As far as we know, no tools currently exist for performing such analysis on wasm. Given wasm's current early stage of development, it would be helpful for such a utility to be developed before wasm becomes ubiquitous. We therefore propose TaintAssembly, a modified version of the V8 JavaScript engine that implements dynamic taint tracking for interpreted WebAssembly.

We begin with describing relevant background and previous research on WebAssembly and taint tracking in Section 2. We follow this with a description of the TaintAssembly engine that we implemented in Section 3. Section 4 describes various testing and benchmarking that we performed on our engine, Section 5 described some limitations to our approach, Section 6 denotes some difficulties faced, and Section 7 contains some concluding remarks.

II. Background and Related Work

A. WebAssembly (wasm)

WebAssembly is a new portable, size- and load-time efficient format, suitable for compilation to the web. WebAssembly is not intended to be a standalone programming language, but rather is primarily designed to act as a compile target for C/C++ code (or JavaScript [4]). The wasm specifications [3] lay out the primitives and structures that must be available, and all of the major browsers have implemented some way of executing wasm code. WebAssembly has a "linear memory," which is analogous to the heap in C/C++. WebAssembly does not enforce memory safety within its memory. However, this memory is kept isolated from everything else, including the code space, execution space, and the executing engine's code and data. WebAssembly also has a standard interpreter, used for testing production code and prototyping new code [2].

Chromium's V8 JavaScript engine is able to execute wasm, and will be the primary engine considered in this paper. V8 uses two different methods to process wasm bytecode.

1) *Interpreted Method:* This is primarily used for debugging and testing, and executes the wasm bytecode line by line at runtime [2].
2) *Compiled Method:* This compiles a wasm file to native bytecode before execution, mitigating the warm-up time that slows down JavaScript [2]. This means that at execution time, the wasm code runs extremely quickly. This is the default way in which most real-world wasm is handled.

Although V8 compiles most WebAssembly applications to native bytecode, implementing taint tracking for native bytecode is difficult as it would require targeting a user's native architecture (`mips`, `arm`, `x86`, etc.). As this varies from user to user, this would greatly increase the complexity of taint tracking. We foresee our tool as something used primarily by researchers and developers, and not in a production environment, so we will focus solely on interpreted wasm.

B. Taint Tracking

Taint tracking is a method for following specific data throughout the execution of a program. Each item that you want to track, be it a primitive or object, is assigned an additional variable that will store its taint value. Each bit of the taint value corresponds to a different source of data, such as network packets, user input, or other information that might be considered important. When these data sources are used in calculations, or interact with each other, a taint tracking engine updates the taint values accordingly, allowing a developer to observe how information has interacted.

Taint tracking has been implemented previously for a number of different goals and applications. It can be used to detect vulnerabilities in binaries and assist in the analysis of malware and network protocols, as well as actively guard against attacks in runtime [5]. It has been implemented on a variety of systems, ranging from TaintDroid [6], which modified the Dalvik interpreter to perform taint tracking for Java bytecode on Android, to static and dynamic taint analysis implemented for JavaScript bytecode in Safari's Webkit Javascript engine [7].

Taint tracking has also been implemented at a lower level. Panorama [8] implements taint tracking at the instruction level for a whole system with fine granularity using QEMU. They use shadow memory, to store taints for every byte of memory, all registers and the network buffer. They also implement conditional taint propagation for certain functions, notably those taking keyboard input.

A lot of work has been done to speed up taint tracking for both debugging and production use. Some approaches to taint tracking use heuristics to determine the granularity of dynamic taint tracking. For example, the CloudTaint system [9] uses certain triggers from data sources to decide whether or not to activate taint tracking at the instruction level. This prevents the CloudTaint system from constantly using up processing power needed for other VMs in the group.

Unfortunately, static taint analysis suffers somewhat from a lack of precision while dynamic taint tracking has a high performance overhead [10]. Therefore, most recent work into taint tracking has taken a combined approach.

One such ensemble approach is to pre-process taint tracking for specific functions, establishing function summaries. Zhu et. al. [11] use the pre-computed function summaries along with semantic analysis [12] [13] to speed up taint tracking by an order of magnitude. Through static analysis, some functions can be determined to not propagate taints from certain inputs to outputs. Instead of running through the entire function with full taint propagation, only a patch function is needed to propagate taint from the inputs to the outputs, eliminating most of the overhead and context switching associated with propagating taint. While most of the previous work has dealt with compiled binaries, the idea of using function summaries to speed up dynamic taint propagation is still applicable to wasm binaries.

Another approach that has worked well is not optimization through static analysis of binaries or code, but rather a dynamic optimization of taint propagation itself. Bruening et al. [14] constructed a dynamic optimization system that optimizes linear sequences of code with changing levels of details to describe instructions. Although Bruening's work did not directly involve taint propagation, similar ideas can be applied to dynamic taint tracking as well. Saxena et al. [15] demonstrated a method of storing a metadata stack and implemented metadata caching in registers, which allowed for quick retrieval of small metadata such as taint during runtime.

In our TaintAssembly engine, our basic taint tracking is modeled off of TaintDroid [6], while our taint tracking on the wasm linear memory draws from the idea of shadow memory presented in Panorama [8]. The other work presents interesting ideas for optimizing and improving dynamic taint tracking, but are significantly trickier to implement in the WebAssembly interpreter, so for the sake of time and simplicity we do not implement these in our TaintAssembly engine.

III. Implementation

We implemented taint tracking for WebAssembly by modifying the V8 JavaScript Engine, used in Google Chrome, Chromium, and Node.js [16]. A GitHub repository containing our modifications, as well as some testing scripts, can be found at https://github.com/wfus/WebAssembly-Taint.

A. Basic Taint Tracking

Our first modification was implementing basic taint tracking. For interpreted WebAssembly, V8 wraps the WebAssembly standard values `i32`, `i64`, `f32`, `f64` using a `WasmValue` class. Therefore, for caching purposes, we introduced taint labels as part of each `WasmValue` object, similar to TaintDroid's modification to Dalvik [6]. Our structure is exhibited in Figure 1. Note that while we allow the user to set the size of `taint_t`, our default is 4 bytes, which is what we have exhibited in the diagram, and what we will assume for the rest of this paper.

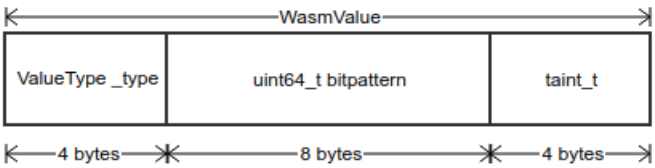

Fig. 1. The structure of a `WasmValue` with taint appended.

In order to initialize the taints, we implemented signature overloading, so a user simply needs to add extra parameters describing taint values to their function calls. For example, if we have a sample JavaScript function invoking a wasm function

```
myfunction = exports._wasm_function
```

with three integer parameters A, B, C, we could have function calls

```
myfunction(50, 100, 200);
myfunction(50,100, 200, 0x000000f0);
myfunction(1, 2, 3, 0x1, 0x2, 0x4, 0x8);
```

where in the first line, all parameters have no taint, in the second the first parameter has taint `0x000000f0`, and in

Wasm Opcode	Operation	Taint Propagation	Description
0x45 - 0x4f	i32.binop v_1 v_2 (comparison)	$T(R) = 0$	Comparison ops do not propagate taint.
0x50 - 0x5a	i64.binop v_1 v_2 (comparison)	$T(R) = 0$	Comparison ops do not propagate taint.
0x5b - 0x60	f32.binop v_1 v_2 (comparison)	$T(R) = 0$	Comparison ops do not propagate taint.
0x61 - 0x66	f64.binop v_1 v_2 (comparison)	$T(R) = 0$	Comparison ops do not propagate taint.
0x67 - 0x69	i32.unop v_1	$T(R) = T(v_1)$	Unops should receive taint of the parameter.
0x70 - 0x78	i32.binop v_1 v_2 (non-comparison)	$T(R) = T(v_1) \vee T(v_2)$	Non-comparison binops should receive taint of both parameters.
0x79 - 0x7b	i64.unop v_1	$T(R) = T(v_1)$	Unops should receive taint of the parameter.
0x7c - 0x8a	i64.binop v_1 v_2 (non-comparison)	$T(R) = T(v_1) \vee T(v_2)$	Non-comparison binops should receive taint of both parameters.
0x8b - 0x91	f32.unop v_1	$T(R) = T(v_1)$	Unops should receive taint of the parameter.
0x92 - 0x98	f32.binop v_1 v_2 (non-comparison)	$T(R) = T(v_1) \vee T(v_2)$	Non-comparison binops should receive taint of both parameters.
0x99 - 0x9f	f64.unop v_1	$T(R) = T(v_1)$	Unops should receive taint of the parameter.
0xa0 - 0xa6	f64.binop v_1 v_2 (non-comparison)	$T(R) = T(v_1) \vee T(v_2)$	Non-comparison binops should receive taint of both parameters.

TABLE I. TAINT PROPAGATION LOGIC

the third the parameters have taint `0x1`, `0x2`, and `0x4`, respectively. Note that in the third example, the `0x8` is thrown away since there are more taint labels provided than there are parameters.

The reason for overloading function signatures as a method of inputting taint is to allow for compatibility between TaintAssembly and the clean default V8 engine. Clean implementations of V8 will ignore overloaded arguments, without error. Therefore, users of TaintAssembly are able to modify any wasm functions in their code to inject taint, while still being able to run the original, unchanged files in clean V8.

Finally, we made modifications to the wasm interpreter to implement the taint semantics similar to the ones described in other taint tracking engines. We changed the way that the interpreter handles binary and unary operations to include taint propagation. Specifically, as exhibited in Table 1 (where R denotes the result of the operation), for all non-comparison operators, we want to generate the taint of the output using the taints of the inputs. This means that for non-comparison binary operations, the taint of the output is the bitwise OR of the taints of the inputs. For unary operations, the taint of the output is simply the taint of the input. With comparison operators, we do not propagate taint.

B. Tainting Linear Memory

WebAssembly utilizes linear memory, analogous to the heap in C, with the operations `T.load` with `T` being one of the four primitives. In V8, both the wasm interpreter and compiler need to be able to write to and read from arbitrary memory addresses in linear memory, which makes it difficult to store the taint alongside data. For example, we have here a simple C program could be converted into the corresponding `.wasm` binary using the WebAssembly Explorer [17]:

```
void rudewrite(int* r, int N) {
    for (int i = 0; i < N; i++) {
        memcpy(r, i, sizeof(int));
        r++;
    }
}
```

```
(i32.load align=1
    (i32.add
        (get_local $2)
        (i32.const 8)))
```

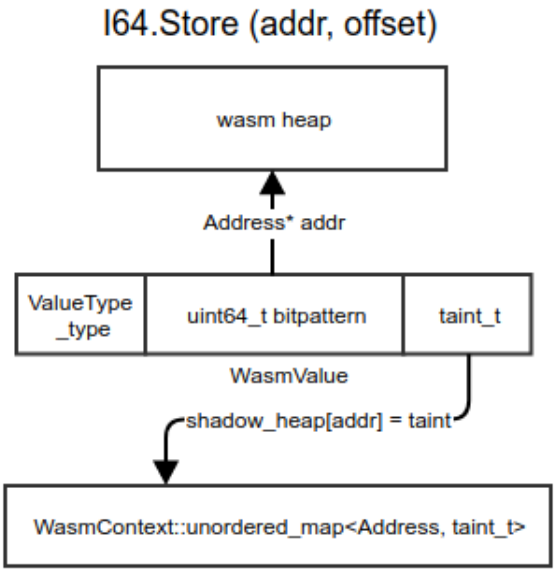

Fig. 2. Tracking taint inside the wasm linear memory (heap). Note the unique WasmContext

This computes a memory address by adding a parameter and a constant value, and loads a value located in linear memory at that address. Since it offsets by `sizeof(int)`, it would be difficult to place taints alongside data without substantial revision to the original code. Using a function to map memory locations with spacing left for taint would also be difficult, because our `taint_t` type defaults to 4 bytes while single byte values are common in C.

Therefore, similarly to Panorama's shadow heap [8], we implemented a mapping from memory addresses to taint values. Instead of using page tables, we implemented it using an `std::unordered_map`. Whenever a memory value is read in through `T.load`, the address is checked to see if it has taint before wrapping it inside a `WasmValue`. Likewise, when a `T.store` command is executed, the WasmValue's taint is unwrapped and the address and taint placed as key and value in our map. Whenever a wasm context destroys itself, it clears the map. This design is illustrated in Figure 2.

We originally meant to modify V8's memory allocation for linear memory be allocating a section of memory opposite of linear memory for storing taints. We decided to use a C++ `std::unordered_map`, however, because V8

uses a generic Handler for managing memory allocations and offsets. Given the time constraints, we were unable to ensure that our tainted linear memory would grow with the regular linear memory without trampling over other V8 internal memory structures. Therefore, our current implementation uses a `unordered_map` kept for each `WasmContext`.

C. Probabilistic Taint Tracking

Finally, we included an option for probabilistic taint, where taint propagates only some of the time. The propagation semantics are as follows. Let us have values v_1, v_2 with respective taints t_1, t_2 and probabilities of propagation p_1, p_2 and operation op. Let r be the result of operation op on those values. Then, r will have taint $t_1 \vee t_2$ with probability $p_1 p_2$, t_1 with probability $p_1(1-p_2)$, t_2 with probability $(1-p_1)p_2$, and 0 with probability $(1-p_1)(1-p_2)$. In other words, the taint from value v_i is propagated with probability p_i. Finally, the propagation probability associated with this r will be $\max(p_1, p_2)$.

In terms of representing the taint value and probability of propagation, we preserve the same taint label structure as from basic taint tracking. The difference is that we reserve some number of higher order bits of the taint label for representing the propagation probability, as in Figure 3.

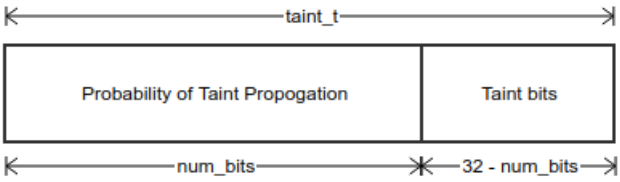

Fig. 3. The structure of a `taint_t` when probablistic taint propagation is being used.

The way this encoding works is that if there are n bits reserved for the probability and the value stored there is m, the propagation probability being represented is $p = \frac{m}{2^n - 1}$.

The motivation for probabilistic taint propagation is primarily that it allows us to set some sort of "lifetime" for taint. As a taint gets passed through the data, the propagation probability makes it possible for the taint to disappear over time. Specifically, the smaller the propagation probability, the shorter a taint's lifetime is. This might desirable if you have some value with unimportant taint, and you only wish for it to only taint values that it acts upon most directly.

D. Additional Features

In addition to basic taint tracking, we also implemented logging. We provide two levels of detail to our logging. In one, we only log when a tainted value is returned from WebAssembly to JavaScript, while in the other we log all operations and function calls.

We also wrote functionality that would terminate execution of a program if a tainted value is returned from WebAssembly to JavaScript. Specifically, we provided an option for terminating the program on the appearance of return values with specific taint flags set, where the relevant flags would be user-input. For instance, a user would input `0x15` if they would like we could termination whenever an output with taint `0x1`, `0x4`, or `0x10` is returned.

The primary motivation for this feature is giving the developer a nice and easy way of seeing when tainted values are getting inadvertently passed to JavaScript libraries without having to go through a taint log. In addition, although our taint tracking engine is primarily meant for developer and debugging use, this feature could hypothetically allow a user to prevent their own information from being leaked out by running our taint tracking engine.

IV. Performance Evaluation

In this section, we describe various tests and benchmarks that we used to evaluate our TaintAssembly engine. We performed tests involving a variety of web applications employing WebAssembly to assess the runtime overhead resulting from our modifications, as well as an examination of a specific source of overhead arising from our method for introducing taint. Finally, we do some analysis of our probabilistic taint feature, looking at how propagation probability relates to taint lifetime.

A. Program Runtime

The basic taint tracking features of our TaintAssembly engine were tested on some online applications that use WebAssembly. Since TaintAssembly is essentially a modified version of V8's `chromium/3270` branch, all testing was performed on a Chromium build (version `64.0.3270.2`) on Ubuntu 17.1 with the custom V8 engine. As a comparison, we used a clean build of Chromium version `64.0.3270.2` to benchmark against.

In our analysis, we noted down the runtime of the following

1) Compile wasm on clean V8 engine
2) Interpreted wasm on clean V8 engine
3) Interpreted wasm on TaintAssembly engine (no taint passed in)
4) Interpreted wasm on TaintAssembly engine (taint passed in)

The specific applications that we used for benchmarking are described in Table 2.

The definitions that we used for runtime in each of the applications are as follows. For the factorial, we defined runtime to be the amount of time that it took for the wasm factorial function to compute a particular factorial. For the skeletal simulation, we let the runtime be the amount of time that it took to perform the calculation for a step of the animation. For the video editor, we chose to analyze one of its filters, the invert filter. We set runtime here to be the time it took to call the wasm function performing the calculations for the transformation on each pixel in the frame. For Funky Karts, we used the in-game clock to help us measure runtime. In particular, we timed the amount of real time that corresponded with three seconds in game time.

The results from our tests are presented in Tables 3 and 4. In Table 3, we present the runtimes relative to the clean V8 engine with compiled wasm, and in Table 4, we present them relative to the clean V8 engine with interpreted wasm.

Program	Description
Factorial [19]	A basic page that advertises the new Rust-To-Wasm (TM) compiler! As everyone knows, if something already exists, let's rewrite it in rust! In any case, this is just a simple web page that uses wasm to calculate the factorial function.
Skeletal Simulation [20]	A WebGL simulation of spooky dancing humanoid figures that compares rendering with WebAssembly and Javascript.
Video Editor [21]	Basic video editor demo that takes in webcam input. Applies zany filters to the video and compares the time it takes for WebAssembly and JavaScript to perform the same filters.
Funky Karts [22]	A WebGL game that utilizes WebAssembly. It's a fun side-scrolling racing game featuring a badger driving through treacherous terrain on in a wild race to save his missing friends. Uses cartoon type graphics that does not seem resource intensive.

TABLE II. TESTING PROGRAMS

Program	Clean (compiled)	Clean (interpreted)	TaintAssembly (untainted)	TaintAssembly (tainted)
Factorial	1.000	2.381	2.400	3.562
Skeletal Simulation	1.000	230.644	247.088	240.960
Video Editor	1.000	200.342	208.261	216.702
Funky Karts	1.000	9.695	10.835	10.839

TABLE III. BENCHMARKING RESULTS (RELATIVE TO CLEAN (COMPILED))

Program	Clean (interpreted)	TaintAssembly (untainted)	TaintAssembly (tainted)
Factorial	1.000	1.008	1.496
Skeletal Simulation	1.000	1.071	1.045
Video Editor	1.000	1.040	1.082
Funky Karts	1.000	1.118	1.118

TABLE IV. BENCHMARKING RESULTS (RELATIVE TO CLEAN (INTERPRETED))

We notice that the factorial and Funky Karts programs have reasonable overhead relative to compiled wasm on a clean V8 engine, while the overheads for the Skeletal Simulation and Video Editor are substantial. However, from looking at the runtimes relative to interpreted wasm on a clean V8, the overheads for TaintAssembly are all fairly reasonable. We observe that much of the apparent slowness of TaintAssembly versus compiled WebAssembly arises from simply using interpreted WebAssembly, and that our taint tracking modifications do not add large overhead (generally only around $5 - 12\%$) to the engine. Therefore, it is reasonable for someone to use our TaintAssembly engine instead of a vanilla wasm interpreter when developing.

B. Taint Insertion Overhead

One of the primary sources of overhead is the introduction of taint through parameter overloading, so we would like to see exactly how much slowdown occurs. To do this, we wrote simple C scripts that did not do any operations and simply took in parameters. We then converted these to wasm with the WebAssembly Explorer [17]. Then, we timed executions of functions calls to the resultant wasm binary on both a clean, unmodified V8 engine and our TaintAssembly engine. More specifically, the functions that we tested each took in 100 arguments. We had one function taking in all `INT32`, one taking in all `FLOAT32`, and one taking in all `FLOAT64`. Note that we do not have any results for `INT64` since 64 bit integer types are not currently supported in JavaScript [2]. The results are in Figure 4.

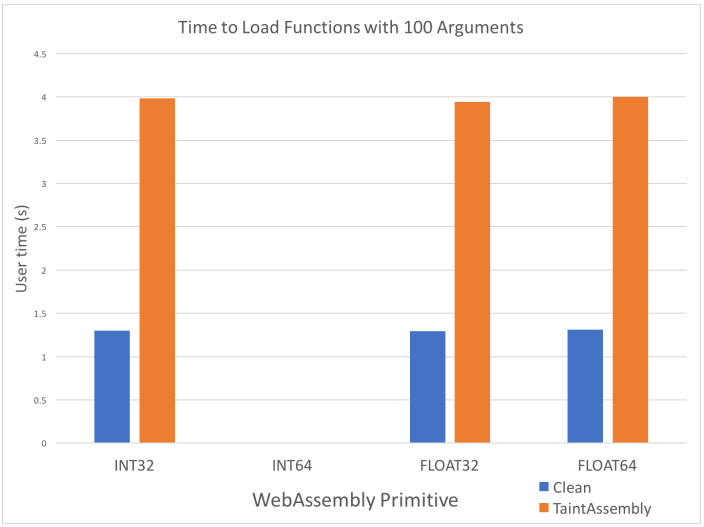

Fig. 4. Overhead of passing taint parameters to functions with 100 arguments.

In an extreme case with 100 parameters, we note that our taint tracking engine is only around 3 times slower than the clean engine. Although there is overhead from injecting taint into our function, the overhead is small enough for development and debugging purposes. Most of the additional time comes from using the function signature to pass in the taint; it allows for minimal modifications to the source code

when passing in taint, with the cost of some small overhead.

C. Probabilistic Taint Benchmarks

We analyze our probabilistic taint feature. As a test of this propagation, we pass tainted data through a simple hash function from [23], which takes a `uint32_t` and returns a `uint32_t`. The hash function does a total of 2700 binary operations. We implemented the hash function in C and converted to wasm with the WebAssembly Explorer [17]. Using a resolution of 8 bits for the probability and running each possible probability k for 100000 iterations, we can see the approximate lifetime of our taint given a specific probability. To quantify this lifetime for a given propagation, we use the probability that applying the hash function to tainted values results in a tainted output. The resultant plot is exhibited in Figure 5.

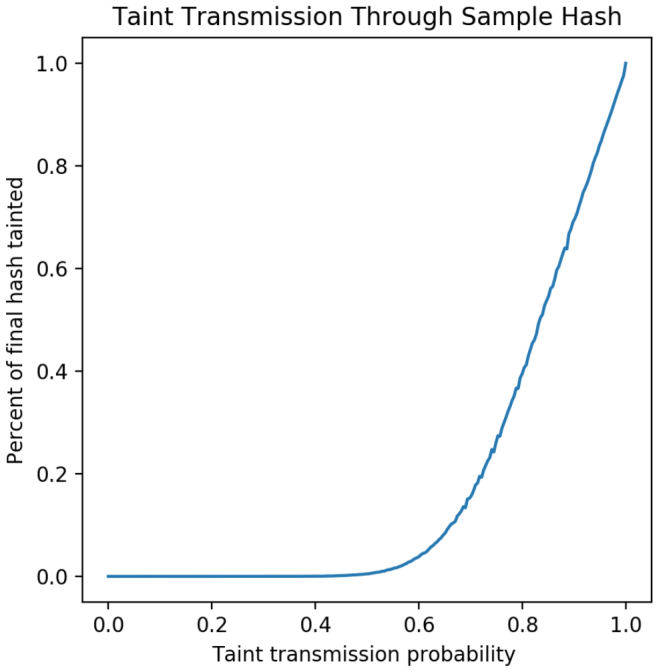

Fig. 5. Percent of return values that retain taint after passing through a simple hash function

In general, the exact shape of this plot is determined by the exact hash function and the number of passes used. In this particular case, it appears to approximate a softplus function. We see that there is a quick drop-off in taint lifetime if the propagation probability goes below 0.6, and that we have around a 50% chance of seeing a tainted result if the propagation probability is around 0.8.

V. Limitations

A. Approach Limitations

Our taint propagation semantics explicitly do not assign taint to comparison operators, as the simplest forms of such taint transmission readily result in taint explosion. However, this has the side effect of not propagating indirect taint. As in Panorama [8], we could have handled this by only tainting the program counter to propagate indirect taint in specific functions, but we did not have a chance to implement such a solution due to time constraints.

B. Taint Source Limitations

In our implementation, our option for probabilistic taint takes up some of the most significant bits of our taint, and requires random number generation. This has some potential drawbacks:

1) Probability resolution is limited by the number of bits
2) Some taint resolution is sacrificed for taint lifetime information
3) A `rand()`[1] call and `min` calculation are required per-op

Therefore, for some computationally heavy programs with lots of taint sources and binary operations, it may not be feasible to run TaintAssembly with probabilistic taint.

C. WebAssembly Linear Memory

Unfortunately, it was difficult to implement efficient taint tracking for WebAssembly's linear memory. Since memory locations can be manipulated directly, for complete taint tracking we need to create another data structure to taint values in linear memory [8]. The current implementation causes a slight runtime overhead for C/C++ programs that frequently read from and write to the heap and has a large memory overhead.

VI. Challenges

Much of our time was spent looking at the V8 source code. Since the code for the runtime was complex and linked with most of the other source files, the builds took a long time to compile. Furthermore, we had to trace through the source code manually and test with the debug shell for most of our initial testing. Tracing the program calls of the default wasm interpreter was also difficult, because V8's JIT caused backtraces to fail.

We originally attempted to follow V8's master branch, but due to active development in `master` we had to rewrite our code to target a stable branch. We ultimately ended up choosing the V8 branch `chromium/3270` and Chromium version `64.0.3270.2` [24].

VII. Conclusion

In this paper, we have implemented dynamic taint tracking for interpreted WebAssembly in the V8 JavaScript Engine. We modified the V8 wasm interpreter, allowing it to perform basic taint tracking with function parameters and local variables, taint tracking in linear memory, and probabilistic taint tracking. In addition, we tested our modified engine on some web applications using wasm and our own custom scripts. Compared to the default interpreter, our TaintAssembly engine has a reasonable overhead of around $5 - 12\%$ for all of the web applications we tested. Therefore, we have exhibited a simple WebAssembly taint tracking engine that is suitable for development and debugging.

[1] V8's default settings are not optimal as they require the user to seed their own entropy source.

ACKNOWLEDGMENTS

The authors would like to thank James Mickens for his guidance and his excellent lectures on systems security. The authors would also like to thank the anonymous members on the Google V8-users and V8-devs mailing list for standing in for V8's documentation, for no benefit of their own. Without their help, we would never have been able to navigate through V8's internal code structure without significant difficulty.

REFERENCES

[1] asm.js. https://asmjs.org.

[2] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the Web Up to Speed with WebAssembly. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.

[3] Web Assembly. https://www.webassembly.org.

[4] M. Reiser and L. Blser. Accelerate JavaScript applications by cross-compiling to WebAssembly. *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, 2017.

[5] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (But Might Have Been Afraid to Ask). *IEEE Symposium on Security and Privacy*, 2010.

[6] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[7] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information Flow Control in WebKit's JavaScript Bytecode. *3rd International Conference on Principles of Security and Trust*, 2014.

[8] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. *Proceedings of the 14th ACM conference on Computer and Communications Security*, 2007.

[9] J. Yuan, W. Qiang, H. Jin, and D. Zou. CloudTaint: An Elastic Taint Tracking Framework for Malware Detection in the Cloud. *The Journal of Supercomputing*, 2014.

[10] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. *Proceedings of the 8th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2012.

[11] D. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. TaintEraser: Protecting Sensitive Data Leaks Using Application-Level Taint Tracking. *ACM SIGOPS Operating Systems Review*, 2011.

[12] A. Sabelfeld and A. Myers. Language-based Information-flow Security. *IEEE Journal on Selected Areas in Communications*, 2003.

[13] J. Ming, D. Wu, J. Wang, G. Xiao, P. Liu. StraightTaint: Decoupled Offline Symbolic Taint Analysis. *Proceedings of the 31st IEEE/ACM International Conference on automated software engineering*, 2016.

[14] D. Bruening, T. Garnett and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," International Symposium on Code Generation and Optimization, 2003. CGO 2003., 2003, pp. 265-275.

[15] P. Saxena, R. Sekar, and V. Puranik. Efficient Fine-grained Binary Instrumentation with Applications to Taint-tracking. *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2008.

[16] V8. https://chromium.googlesource.com/v8/v8.git.

[17] The WebAssembly Explorer. https://mbebenita.github.io/WasmExplorer.

[18] Emscripten. https://kripken.github.io/emscripten-site/.

[19] Factorial in WebAssembly. https://www.hellorust.com/emscripten/wasm-fact/.

[20] Animation Demo. https://aws-website-webassemblyskeletalanimation-ffaza.s3-website-us-east-1.amazonaws.com/.

[21] WebAssembly Video Editor. https://d2jta7o2zej4pf.cloudfront.net.

[22] Funky Karts. https://www.funkykarts.rocks/demo.html.

[23] R. Jenkins. Hash Functions. *Dr. Dobb's Journal*, 1997.

[24] Chromium. https://chromium.googlesource.com/chromium/chromium.git.