The Taint Rabbit: Optimizing Generic Taint Analysis with Dynamic Fast Path Generation

John Galea
Department of Computer Science
University of Oxford
john.galea@cs.ox.ac.uk

Daniel Kroening
Department of Computer Science
University of Oxford
kroening@cs.ox.ac.uk

ABSTRACT

Generic taint analysis is a pivotal technique in software security but suffers from staggeringly high overhead. In this paper, we explore the hypothesis whether just-in-time (JIT) generation of fast paths for tracking taint can enhance the performance. To this end, we present the *Taint Rabbit*, which supports highly customizable user-defined taint policies and combines a JIT with fast context switching. Our experimental results suggest that this combination outperforms notable existing implementations of generic taint analysis and bridges the performance gap to specialized trackers. For instance, Dytan incurs an average overhead of 237x, while the Taint Rabbit achieves 1.7x on the same set of benchmarks. This compares favorably to the 1.5x overhead delivered by the bitwise, non-generic, taint engine LibDFT.

ACM Reference Format:
John Galea and Daniel Kroening. 2020. The Taint Rabbit: Optimizing Generic Taint Analysis with Dynamic Fast Path Generation. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS '20), October 5–9, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3320269.3384764

1 INTRODUCTION

Dynamic taint analysis [44] is an enabling technique in software security for tracking information flows. Typical applications include malware analysis [2, 31, 54], vulnerability discovery [8, 11, 40] and runtime attack detection [26, 37, 50]. The key feature is the tracking of memory locations and CPU registers that store "interesting" or "suspicious" data. Data of this kind is called *tainted*. Taint is checked at particular points during program execution to determine whether or not certain runtime properties hold, e.g., to check whether the instruction pointer could be controlled by an attacker [37], or to identify which parts of the user input influence path conditions to optimize fuzzing [11, 40].

Most taint analyzers, e.g., LibDFT [28], implement single, byte-sized tags and often just track whether data is tainted or not. This setup supports efficient propagation of taint (using a *bitwise or*) and efficient querying of a location's taint status. However, many interesting applications that build upon taint analysis require richer propagation logic and more complex taint labels. For instance, VUzzer [40] propagates sets containing offsets of input bytes, and Undangle [8] tracks heap pointers, storing taint information in a composite data structure. To support these use cases, previous work proposed to extend single-tags and deliver what is called *generic taint analysis*. Generic taint engines track richer labels (say via a 32-bit pointer) and support user-defined taint propagation policies. The first notable generic taint engine is Dytan [14].

The key problem is that the versatility of Dytan comes at a price: the authors of Dytan report a staggering runtime overhead of ~30x on `gzip`, which has led to the perception that generic taint analysis is, in essence, impractical. We challenge this perception, and explore the hypothesis whether or not generic taint analysis can be delivered with a runtime overhead that is practical enough for security applications.

We argue that a combination of two optimizations is able to deliver taint tracking that is both versatile and sufficiently fast for various software analyses. We present an implementation of our ideas in a tool called the *Taint Rabbit*. The Taint Rabbit achieves an overhead that is significantly lower than that of the generic taint analyzer Dytan. On the CPU-bounded benchmarks that Dytan manages to run, we observe an overhead of 237x compared to native execution. By contrast, the Taint Rabbit incurs only 1.7x. This is close to what can be expected: LibDFT, the leading bitwise taint engine, achieves an overhead of 1.5x on the same benchmarks. Therefore, our approach reduces the conflict between performance and versatility significantly.

The Taint Rabbit is Generic. Our taint propagation is not specific to a fixed taint policy. We map a 32-bit word (or pointer) to every tainted byte, enabling the storage of a reference to a custom taint label data structure. The Taint Rabbit propagates the pointers efficiently, and supports custom handlers, provided by the user, to update the taint labels according to a desired taint policy. Section 5 details our algorithms for generic taint analysis.

The Taint Rabbit is Optimized. The key idea behind the Taint Rabbit's high performance is to optimize taint analysis for dynamic binary instrumentation (DBI) [5]. This approach is standard in leading bitwise taint analyzers, such as LibDFT [28], but has not yet been thoroughly investigated for generic taint analysis, which is much harder to optimize. In particular, it is not possible to build instrumented instruction handlers for taint propagation using simple *bitwise or* operations; the taint propagation has to be optimized for a given, custom taint policy. We investigate two techniques to speed up generic taint analysis. First, we reduce analysis overhead just-in-time by dynamically generating fast paths according to *in* and *out* taint states of basic blocks. Second, our generic analysis avoids

ASIA CCS '20, October 5–9, 2020, Taipei, Taiwan

ACM ISBN 978-1-4503-6750-9/20/06…$15.00
https://doi.org/10.1145/3320269.3384764

expensive context-switching by limiting function calls. Section 6 details these optimizations.

The Taint Rabbit's generic capabilities are assessed using three security applications, which employ different taint policies. The applications have been proposed previously but have not yet leveraged our efficient taint engine. Specifically, we evaluate an exploit detector [37], a Use-After-Free debugger [8] and a fuzzer [40].

To measure performance, we use SPEC CPU 2017 [7], command-line utilities, PHP and Apache as benchmarks. As baselines for comparison, we conduct the same experiments on a wide range of alternative trackers, including LibDFT [28], DataTracker [48], DataTracker-EWAH [40], Traintgrind [29], BAP-PinTraces [6], Triton [41], Dr. Memory [4], DECAF [24] and Dytan [14]. Results shows that the Taint Rabbit is the fastest generic taint tracker among those evaluated.

In summary, we make the following contributions:

(1) **Optimized and generic taint analysis.** While optimized taint analyses have been proposed, none support extensible propagation logic, and thus lack versatility. Meanwhile, existing generic taint engines incur prohibitively high overheads. Our contribution bridges this gap via dynamic fast path generation and instrumentation that avoids calls.
(2) **The Taint Rabbit.** We introduce a framework for building security applications based on dynamic taint analysis. The Taint Rabbit and the tools built upon it are all available at https://github.com/Dynamic-Rabbits/Dynamic-Rabbits.
(3) **An extensive evaluation.** The Taint Rabbit is evaluated on several relevant benchmarks, including SPEC CPU 2017, and is compared with *nine* other taint-based systems. Three security applications are also assessed to demonstrate the versatility of the Taint Rabbit.

2 OVERVIEW

Figure 1 illustrates the high-level design of our approach. Every new basic block observed during runtime is passed to the Taint Rabbit for instrumentation by the DBI platform. The Taint Rabbit weaves efficient instruction handlers, responsible for propagating taint generically, into the application's code. Instruction handlers are implemented in assembly to limit context-switching done by transparent function calls.

The Taint Rabbit employs a JIT approach to adaptively enhance the performance of generic taint analysis. It generates copies of original basic blocks but leaves them uninstrumented to establish fast paths. In particular, an uninstrumented basic block is executed when all of its input and output registers/memory are not tainted. Otherwise, the slow path is taken, implementing full-blown taint analysis. The basic variant of this scheme has been proposed previously and implemented in Lift [39], but the Taint Rabbit can do more: it also **dynamically** generates fast paths. If the Taint Rabbit encounters a set of *in* and *out* taint states that is frequently executed at runtime, the basic block is duplicated again and instrumented specifically to handle the particular case. Irrelevant instructions that do not deal with taint for the given case are safely elided from instrumentation. Therefore, fully-instrumented code is executed less often than in conventional approaches as, due to additional fast paths, control is not always blindly directed to it upon tainted

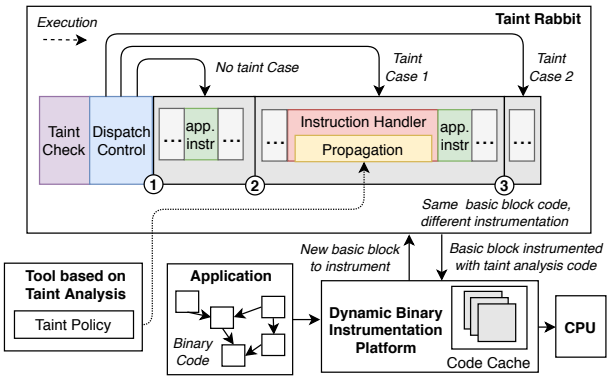


Figure 1: High-level design of the Taint Rabbit

data. Our technique is based on the hypothesis that basic blocks are usually executed with the same taint states. Therefore, the cost of generating fast paths pays off.

Our approach allows the user to focus on defining the desired taint propagation policy, while the Taint Rabbit facilitates fast and generic taint analysis. Inspired by previous work [10, 53], the user provides code describing how labels are merged and derived, without delving into intricacies regarding the internals of the engine.

3 APPLICATIONS OF TAINT ANALYSIS

There are numerous use-cases for taint analysis. We focus on three applications and emphasize that their taint policies and propagation logic differ.

Control-Flow Hijacking. Previous work [37] has shown that taint analysis can detect control-flow hijacking attacks. Since the analysis only has to taint check control data, *bitwise or* operations suffice for propagating taint flags.

UAF Detection. Use-after-free (UAF) bugs are exploitable [45]. Undangle [8] debugs such vulnerabilities by tracking heap pointers via taint analysis. Undangle monitors allocations and deallocations and sets pointer statuses stored in taint labels to *LIVE* and *DANGLING*, respectively. Taint is propagated when pointers are copied either directly or arithmetically, and a location is untainted if it is no longer a pointer. For instance, the subtraction of two pointers yields a taint-free distance even though both the sources are tainted.

A *bitwise or* operation is not suitable for pointer tracking; a location may be associated with one of three states, namely *UNTAINTED*, *LIVE*, and *DANGLING*, and their merging cannot be appropriately done with the operation. Moreover, apart from pointer statuses, Undangle's labels also contain debugging data, e.g., PCs of pointer creations, and thus are of composite type. Instead of using a *bitwise or*, Algorithm 3 (in the appendix) gives an implementation for propagating such labels via conditional statements.

Fuzzing. VUzzer [40] uses taint analysis to discover interesting input bytes to mutate. The label is a bit set, where each bit corresponds to a byte of the input file. Since registers or memory may be influenced by multiple bytes, propagation performs a union operation. A *bitwise or* is sufficient if bit sets fit within the operand size; however, this is unlikely due to large input files. VUzzer therefore uses a bit array, implying that union operations require branching.

While bitwise tainting is appropriate for some applications, others require richer capabilities. Yet, many taint engines are tuned solely for the former. Our approach is more versatile and suitable for all use cases.

4 TAINT ANALYSIS VIA DBI

Similar to previous research [12, 14, 28, 39], we focus on an online analysis that is implemented using dynamic binary instrumentation (DBI) [5]. In DBI, basic blocks of the application under analysis are instrumented and stored in a code cache at runtime. The inserted code needs to be transparent so that it does not affect the execution of the application. To simplify tool development, DBI frameworks [36], such as Pin [33] and DynamoRIO [5], allow the insertion of transparent calls, known as *clean calls* [20], which invoke a given function at runtime. Essentially, these functions implement the taint analysis. However, before the call, a context switch is performed, which creates a dedicated stack and comprehensively spills/restores CPU registers [49]. Since taint analysis requires instrumenting many instructions to track data movements, these context switches incur high overheads of at least ~15x[1].

Consequently, DBI frameworks attempt to avoid clean calls and automatically inline analysis code with the application's instructions. Ideally, the context switches only spill/restore live registers used by the routines and therefore are cheaper than full clean calls. Figure 7 (in the appendix) shows that this optimization reduces the overhead to ~3.3x. Routines are inlined by DBI frameworks only if they are *simple*, i.e., they are small, avoid control-flow and perform no function calls themselves [20, 38].

LibDFT exploits the inline optimization. Listing 1 in the appendix shows one of its taint propagation routines. Essentially, propagation is done by bitwise tainting, which avoids long complicated code with conditional branches. Notably, the use of bit flags as taint labels, coupled with bitwise operations for propagation, yields simple routines, thus activating the inline optimization.

However, the propagation supported by LibDFT is limited. Previous work [48] has extended LibDFT to track input file offsets. The work increased LibDFT's versatility, resulting in a new taint engine called DataTracker. With some modifications, DataTracker is used by VUzzer. However, the changes made in DataTracker break the original inline optimization. Listing 2 (in the appendix) gives the instruction handler that corresponds to the one in listing 1. Because of the function calls and the branching in the instruction handler, Pin fails to inline and the performance drops. We ran DataTracker and confirmed the failure to inline by inspecting the logs produced by Pin. Our results on `bzip2` also show that LibDFT is faster than DataTracker: LibDFT has an overhead of 2.6x, while DataTracker incurs 36x over native execution.

Although existing optimizations for propagating taint are effective, many are dependent on specific policies and taint label structures. LibDFT's inlining approach is mainly suitable for bitwise tainting. We believe that optimizations not tied to particular policies are desirable as they are more useful to the community who use taint analysis for a broad range of applications.

[1]To quantify this overhead, we ran the DynamoRIO tool `inscount` that uses clean calls to count the number of instructions executed by an application. We see a slowdown of ~15x on SPEC CPU 2017 (Figure 7 in the appendix). https://github.com/DynamoRIO/dynamorio/blob/master/api/samples/inscount.cpp

5 THE TAINT RABBIT

Generic taint analysis enables user-defined taint policies. The support of custom merging of labels during propagation is fundamental to avoid changing the internals of the taint engine for a particular application. We now describe the Taint Rabbit's high-level algorithms. Our optimizations are then detailed in the next section.

Binary Analysis. We scope our analysis to x86 binaries. The code that performs propagation consider the semantics of the instructions. This avoids tainting output locations unecessarily, e.g., tainting stack pointers.

Generic Label Structure. The unit of meta-data that the Taint Rabbit reasons over is a 32-bit word. The word may itself store tags or act as a pointer to any taint label structure[2]. A `NULL` value represents "no taint".

Byte Granularity. Meta-data is mapped to every byte in memory and registers; e.g, a `mov eax, ebx` propagates four labels, one for each byte in `ebx`. Labels are stored in shadow memory [55].

Generic Taint Propagation. As shown in Figure 2, taint labels are propagated via user-defined code called *taint primitives*. A taint primitive is a building block to taint propagation, and is responsible for deriving a taint label from a set of source labels. During propagation, the Taint Rabbit fetches the labels of source operands, applies appropriate primitives with respect to the semantics of x86 instructions, and assigns resulting labels to destination operands.

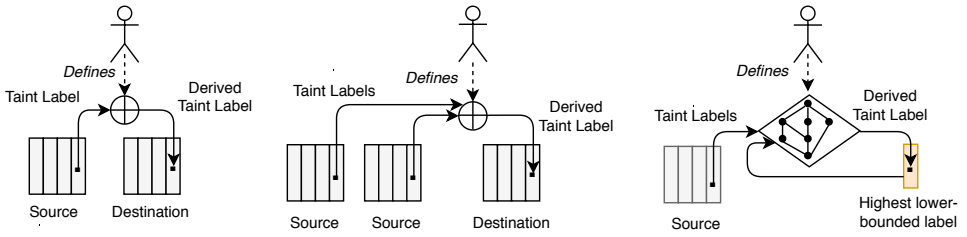
Three user-defined taint primitives are currently required for our supported instructions, and are informally defined as (1) $src \rightarrow dst$, (2) $src, src \rightarrow dst$, and (3) $src, src \rightarrow_M meet$. The first two primitives produce a label to associate with a destination byte from one and two sources respectively. For example, algorithm 3 (in the appendix) is a $src, src \rightarrow dst$ primitive. Meanwhile, inspired by previous work [10], the third primitive obtains the highest lower-bounded label in a lattice.

We employ such a taint-primitive-based approach to facilitate user implementation of taint policies through separation of concerns, and to achieve flexible design in relation to the internals of the Taint Rabbit. With reasonable imprecision, we found that the three taint primitives are sufficient for our instruction handlers to track taint effectively, even for complex instructions such as `punpckldq` and `pmaddwd`.

The $src, src \rightarrow_M meet$ and $src, src \rightarrow dst$ primitives have different purposes despite both having a tuple of labels as their domain. The former primitive merges labels all associated with a source operand, while the latter, which may not be absorptive, derives a label to associate with a destination byte. Essentially, the *meet* primitive provides a method for the Taint Rabbit to obtain a single label that overall describes the taint of a multi-byte-sized operand. Regarding pointer tracking, given two labels that store a *LIVE* and *DANGLING* status respectively, their *meet* is a *DANGLING* status, while the derived label is *UNTAINTED*.

Under and Over Approximation. Instruction handlers propagate taint both under approximately and over approximately. The choice between the two approaches depends on the instruction. Under-approximate propagation occurs if each destination byte of an instruction is affected by a single source byte. Many transfer

[2]In contrast to the Taint Rabbit, Dytan [14] exposes a bit vector as its label structure instead of a generic pointer.



(a) Primitive $src \rightarrow dst$ **(b) Primitive** $src, src \rightarrow dst$ **(c) Primitive** $src, src \rightarrow_M meet$

Figure 2: User-defined primitives used by the Taint Rabbit to spread taint

Algorithm 1: Under-Approx. Tainting via $src, src \rightarrow dst$

```
Data: ID dst, ID src1, ID src2, Integer opnd_size
1  for i ← 0 to opnd_size - 1 do
2      src_label1 ← lookup_label(src1 + i);
3      src_label2 ← lookup_label(src2 + i);
4      dst_label ← src_src_dst_primitive(src_label1, src_label2);
5      set_label(dst + i, dst_label);
6  end
```

Algorithm 2: Over-Approx. Tainting via $src, src \rightarrow dst$

```
Data: ID dst, ID src1, ID src2, Integer opnd_size
1  meet_label1 ← UNTAINT;
2  meet_label2 ← UNTAINT;
3  for i ← 0 to opnd_size - 1 do
4      label ← lookup_label(src1 + i);
5      meet_label1 ← meet_primitive(meet_label1, label)
6      label ← lookup_label(src2 + i);
7      meet_label2 ← meet_primitive(meet_label2, label)
8  end
9  for i ← 0 to opnd_size - 1 do
10     dst_label ← src_src_dst_primitive(meet_label1,
         meet_label2);
11     set_label(dst + i, dst_label);
12 end
```

instructions (e.g., `mov`) fall under this class. Meanwhile, instructions that have destination bytes influenced by multiple source bytes, such as arithmetic instructions (e.g. `add`), are handled over-approximately. In relation to explicit data-flow, we are not aware of other works that take such a hybrid approach when implementing instruction handlers; LibDFT and Triton perform under-approximate and over-approximate analysis respectively.

Algorithm 1 gives an instruction handler for propagating taint from two sources to a destination. While handlers for other instructions differ, the algorithm exemplifies the crux of under-approximate generic taint analysis. The handler takes the size of the operands and the location IDs corresponding to the starting bytes of the destination and the two sources as inputs. We associate numerical IDs to all bytes in registers, and use addresses to refer to memory. The handler fetches the taint labels associated with two source bytes (lines 2–3), and applies the $src, src \rightarrow dst$ primitive to obtain the resulting label *dst_label* (line 4). At line 5, *dst_label* is mapped to the destination byte.

At each iteration, a label is propagated to the i^{th} byte of the destination by solely considering the labels mapped to the i^{th} bytes of sources. The algorithm is therefore under approximate; sources may influence more than just the i^{th} destination byte, e.g., due to arithmetic. In such cases, the Taint Rabbit performs over approximate generic taint propagation, which is described in Algorithm 2. Under this setting, all bytes of the destination operand are tainted based on *meet* labels. Each source operand is considered individually, as the *meet* primitive is applied to all of their bytes (lines 3–8). The first loop results in obtaining the *meet* labels, namely $meet\_label_1$ and $meet\_label_2$. The next loop iterates over each byte of the destination operand, where at line 10, the $src, src \rightarrow dst$ primitive is applied to the *meet* labels. The derived label is then mapped to the corresponding destination byte at line 11.

Algorithms 4 and 5, which show the usage of the $src \rightarrow dst$ primitive, are in the appendix. They are similar to Algorithms 1 and 2, but only accept one source operand. We implement them to propagate taint for instructions such as `mov`, `inc`, and `bswap`.

6 OPTIMIZED DESIGN

The previous section describes the Taint Rabbit's high-level algorithms for generic taint analysis. We now focus on the Taint Rabbit's design optimized for DBI.

6.1 Challenges

We address the following non-trivial challenges:

High Tracking Rate. Dynamic taint analysis incurs overhead due to the high execution rate of instruction handlers. On a test run, we measured that at least 73% (over 8 billion) of the instructions executed by `bzip2` conventionally require instrumentation. This challenge is addressed in Section 6.2.

Expensive Context Switching. Unlike bitwise tainting, generic taint propagation is more complex, e.g., due to control flow. This leads to expensive context-switching incurred by clean calls. We address this challenge in Section 6.3.

6.2 Dynamic Fast Path Generation

The Taint Rabbit generates fast paths to reduce the execution of instruction handlers. We now detail the actual process of the Taint Rabbit. A *code example* is shown in Figure 10 in the appendix.

Truncation. When a new basic block is provided by the DBI platform, the Taint Rabbit starts off by finding any memory addresses that cannot be determined at the start of the basic block due to non-static dependencies. Such addresses are problematic as their taint statuses cannot be checked prior to fast path entry at runtime. The issue is mitigated by truncating basic blocks at points where memory dereferences are calculated based on register values that are inconsistent with their starting values[3]. The truncated code is no longer considered at this point, but is treated as a new separate basic block that undergoes its own analysis. The inputs and outputs of the basic block are then retrieved and stored in a set by simply inspecting the remaining, non-truncated, instructions.

Code Duplication. Next, the basic block is copied to produce multiple adjacent instances of it. A global map $\mathcal{M}$ associates a basic block ID with meta-data specifying the different cases of instrumentation; ergo, the number of cases determines the total number of basic block instances. By default, this meta-data is initialized with two defined cases where all or none of the basic block's inputs and outputs are tainted. An entry label is inserted prior to each instance, and direct jumps are inserted at the ends to span over the code of other instances and exit. To uphold the one-exit-point property of basic blocks, control-flow *app* instructions are not duplicated and left at the end.

Taint Checks and Control Dispatch. The Taint Rabbit proceeds by inserting initial code to determine the *in* and *out* runtime taint states of a basic block at point of entry. The result is encoded as a mask where each bit indicates whether or not an input/output is tainted. Compare and branch code sequences check the encoded mask with the masks of the defined cases (retrieved via $\mathcal{M}$) and direct control to appropriate basic block instances. The fall-through implies that a new case is encountered and poses as an opportunity for fast path generation. An unhandled case defaults to the execution of the fully-instrumented basic block instance.

Placed in the common path, taint checking is performance critical. The dispatcher *must* direct control fast. Determining the taint status of an input/output by inspecting all of its pointer-sized tags one-by-one is costly due to many comparison instructions and cache pollution. The Taint Rabbit alleviates this issue by quickly checking registers via an over-approximate scheme where a taint status bit is tracked for each register. Apart from conducting generic taint analysis, our instrumented paths also maintain these bit statuses. Therefore, a lot of the dispatcher's checking process is shifted down to less-critical paths, away for the uninstrumented fast path. To a certain extent, the idea of using over-approximate tags is similar to [42], but the Taint Rabbit cleverly uses the `pext` instruction [25] to aid quick construction of the mask. Although checks are overly imprecise due to higher granularity (e.g., partial registers may be incorrectly deemed as tainted), propagation is still done by our precise instruction handlers. Moreover, with regards to taint checks of memory, the Taint Rabbit leverages SIMD instructions, e.g., `ptest`, to efficiently test multiple labels simultaneously.

[3]Our actual implementation is more advanced with constant propagation. For example, instead of promptly truncating upon `push` and `pop` instructions, the Taint Rabbit patches operands with offsets resulting from stack pointer decrements and increments.

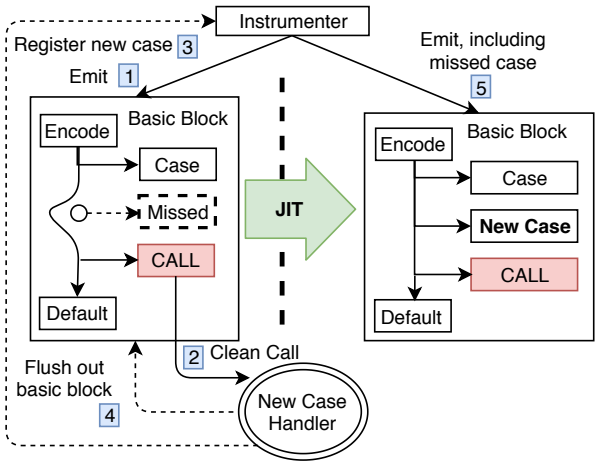


Figure 3: Dynamic Fast Path Generation

Data-flow Analysis and Instrumentation. The basic blocks are then instrumented with taint propagation code. The paths for the two default cases are established by having one basic block fully instrumented and another without any instrumentation at all. To handle other cases determined at runtime, forward data-flow analysis is performed on the basic block to identify which instructions deal with tainted operands. Such instructions propagate taint and are therefore instrumented, while others are elided. Naturally, the initial in-set for data-flow analysis includes the *in* and *out* taint states of the particular case.

Inlining instrumentation code which is based on user-defined taint primitives may result in large code fragments that stresses the instruction cache and the encoding to the DBI cache. This issue is exacerbated by the instrumentation of duplicated basic blocks. As a mitigation, the Taint Rabbit outlines instruction handlers to shared code caches at the user's discretion. Note, outlining does not use clean calls but trampolines.

Fast Path Generation. Figure 3 describes the process of dynamic fast path generation. A clean call is performed (infrequently) when no fast path exists for a new set of *in* and *out* taint states. The mask of the unhandled case is retrieved and registered by updating $\mathcal{M}$. The existing code fragment is then flushed out from the DBI cache, and instrumentation is re-triggered; **now** with the inclusion of the missed path. The Taint Rabbit also has a stopping mechanism that prevents basic blocks from attempting generation if intended fast paths do not actually elide any instructions. Finally, rather than immediately triggering dynamic fast path generation, we employ conventional JIT heuristics [43], based on execution count to reduce latency induced by flushing.

6.3 Efficient Instruction Handlers

Building a generic taint engine with a high-level programming language renders instruction handlers, responsible for propagating taint, too complex to be automatically inlined by a DBI tool. Therefore, instruction handlers are built using hand-crafted x86 assembly

code. Although previous work [4, 12] take a similar approach, their instruction handlers are coded for bitwise tainting, using simple *or* operations, rather than for generic taint analysis. Many instructions are supported including those related to SIMD. We designed the code to follow known practices for optimization to best of our abilities [25]. Iteration and branching are reduced with instruction handlers, amounting to over 970 in count, being specific not only to different opcodes but also operand sizes and types. The loops in Algorithms 2 and 3 are unrolled.

Instruction handlers do not use a stack but rely on thread-local storage and registers for memory. Although over approximation is slower than under-approximation, owing to the *meet* primitives, it is only performed when analyzing certain instructions as described in Section 5. Taint primitives are provided memory operands referring to source taint labels and two general purpose (GP) scratch registers for their implementation. More spillage is performed if the primitive requires more registers.

6.4 Other Optimizations

We also adopt previously proposed optimizations [4, 28, 39]. First, live register analysis is done to only spill/restore register values that are relied upon by subsequent application instructions. Second, we optimize taint checks by minimizing redundant shadow address translations when memory operands share the same base address. Third, space overhead is reduced by creating shadow memory on demand, with the first write, detected via special faults. Lastly, memory dereferences are minimized by using addressable thread local storage to access frequent fields, e.g., registers' shadow memory.

6.5 Implementation

The Taint Rabbit is the core of the Dynamic Rabbits, a suite of binary analysis libraries for building taint-based tools. The Dynamic Rabbits are built upon DynamoRIO and Dr. Memory. They consist of over 70,000 lines of C code (including tests) and their source is available at https://github.com/Dynamic-Rabbits/Dynamic-Rabbits. Furthermore, Dr. Memory's shadow memory library Umbra [55] was enhanced to handle 32-bit tags. We also implemented a new DynamoRIO library called `drbbdup`, which duplicates code of basic blocks. In turn, `drbbdup` is used to implement fast path generation. The majority of drbbdup's code has been merged to DynamoRIO's repository[4].

Several tools, including Perf [18], were leveraged to profile the Taint Rabbit. Analysis results, visualized via flame graphs [22], are shown in the appendix (Figure 8).

6.6 Limitations

Currently, the Taint Rabbit does not analyze 64-bit binaries. The main reason is that many existing engines, particularly LibDFT, only support 32-bit and direct experimental comparisons are vital to reduce threats to validity. Moreover, the Taint Rabbit supports many but not all x86 instructions (mainly floating point). In the appendix, Table 4 provides a comprehensive list of the supported instructions. When an unsupported instruction is encountered, all destinations are untainted to avoid false positives. To penalize the Taint Rabbit, this process is done via a clean call.

[4]https://github.com/DynamoRIO/dynamorio/issues/4134

Our approach is more versatile than bitwise tainting. However, while instruction handlers are call-free, user-defined taint primitives could break the optimization. These include primitives that perform a call to allocate dynamic memory. We mitigates this issue with an inline custom allocator that performs clean calls in a slow-path only when requesting additional memory for management.

Regarding fast path generation, truncation of basic blocks circumvents static whole-program pointer analysis. However, it is not a perfect solution as the number of basic block increases as a consequence. This, In turn, increases the number of taint checks done by the dispatcher. Moreover, `rep` instructions, which deal with many bytes, are not checked, as determining their *in* and *out* taint states could be expensive. Therefore, these instructions are treated as potential taint sources when conducting forward data-flow analysis, and are always instrumented.

Issues concerning high memory overheads may also arise when analyzing large applications. The main cause is due to the Taint Rabbit's scaled up shadow memory where a 32-bit pointer is mapped to each application byte. To address this challenge, we implemented a simple garbage collector that is triggered when memory is low. The collector iterates over shadow memory blocks and checks whether they store any tainted data. If an entire block is found to store only untainted data, i.e., NULL values, it is deallocated.

7 EVALUATION

We performed an experimental evaluation to answer the following research questions.

- RQ1: How much does call-avoiding instrumentation and dynamic fast path generation improve the performance of generic taint analysis?
- RQ2: With these techniques, is the performance of generic taint analysis comparable to the state of the art of bitwise taint analysis?
- RQ3: Can the improved generic taint analysis scale to real-world target-applications?
- RQ4: Does a primitive based approach enable generic taint analysis?

Experiments ran on 32-bit Ubuntu 14.04 LTS machines, each equipped with an 8 core 2.60 GHz Intel Core i7-6700HQ CPU and 32 GB RAM. Full results with numerical figures[5], along with scripts for running many of our experiments[6], are available online.

7.1 The Taint Rabbit Engines

The Taint Rabbit (TR) offers two generic taint engines. As a baseline, `TR-CC` has instruction handlers implemented in C and uses clean calls. The second engine, `TR-RAW`, has its instruction handlers implemented in assembly without clean calls. When combined with fast paths, these variants are referred to as `TR-CC-FP` and `TR-RAW-FP`. The engineering effort required to implement another taint engine, namely `TR-CC`, as our baseline was worthwhile to answer `RQ1`.

For our experiments, the Taint Rabbit considers two taint policies. The first policy (`TR-ID`) assigns a new numerical ID to each destination byte whenever taint propagation occurs. This policy could

[5]https://docs.google.com/spreadsheets/d/1gAm7GJBB3Rl4bfTwWq-ITcNyVuRtTH2vQYYaS3n-OUk/edit?usp=sharing

[6]https://github.com/Dynamic-Rabbits/Taint-Evaluator

serve as a basis for a static single assignment trace generator. ID assignment is achieved by having the $src \rightarrow dst$ and $src, src \rightarrow dst$ primitives increment a counter if any source is tainted. The 32-bit tags contain the IDs and are not used as pointers.

The second policy (TR-BV) propagates bit vectors similar to the multi-tag policy adopted by Dytan. Instead of mapping a separate bit vector to each tag, which results in high memory usage, our policy represents bit vectors concisely. We use a global reduced binary decision tree, similar to previous work [11]. However, the algorithms presented previously are recursive and would break our call-free optimization upon union operations. Therefore, we devised iterative variants where clean calls are done only when inserting a new allocated node to the tree. Through this memoization, inserted nodes only represent bit vectors that have not been encountered previously. The $src \rightarrow dst$ primitive simply transfers a source's pointer referring to a node in the tree, while the other primitives efficiently perform unions via inlined hash-lookups. Note, our two policies cannot be implemented with a bitwise taint engine, and therefore their consideration is vital to answer RQ1, RQ2 and RQ3.

7.2 Other Taint-Based Systems

To aid answer RQ2 and RQ3, we ran nine other taint analyzers on our benchmarks as baselines for comparison. Table 5 (in the appendix) gives a summary of their main features. LibDFT [28] inlines bitwise taint analysis, while Dytan [14] performs user-defined operations on bit vectors containing multiple tags. Triton [41] is another framework that uses Pin for tracing. DataTracker [48] focuses on data provenance; a variant, named DataTracker-EWAH [40], records input offsets to optimize fuzzing. We also ran BAP-PinTraces [6], which generates execution logs of instructions that deal with taint. Its taint propagation routines are not implemented specific to the semantics of instructions, but instead leverage the DBI's IR to determine source and destination operands. DECAF [24] is a QEMU-based taint tracker that inlines precise bitwise propagation to Tiny Code Generator (TCG) instructions, and Taintgrind [29] is a taint engine built upon Valgrind [36]. Moreover, the Dr. Memory [4] debugger builds on DynamoRIO to check the addressability of memory using bitwise tainting. Unfortunately, we are unable to assess its taint analysis separately as it is tightly coupled with other components. Therefore, its reported overhead also includes memory checks. However, we did remove code in DataTracker-EWAH and BAP-PinTraces that concerns logging to file for better evaluation. DBI overhead was also measured separately without taint analysis. We give results for Pin, DynamoRIO and Valgrind, labeled as Pin-Null[7], DR-Null and Nullgrind, respectively.

7.3 Performance

In order to answer RQ1, RQ2 and RQ3, we measure the performance of the Taint Rabbit on benchmarks relating to compression, PHP, image parsing, Apache and SPEC CPU. For these experiments, we set the Taint Rabbit to taint all data read from files, sockets, command-line arguments and environment variables. While we envisage better performance with less taint introduction, our methodology aims for a grounded evaluation, measuring the worst performance cases where taking optimal fast paths is difficult due to comprehensive tainting. No taint is introduced when running other tools, which nevertheless instrument instructions despite no taint propagation. However, our setup benefits instruction handlers which perform efficiently when dealing with untainted data only. There are exceptions, e.g., Dr. Memory, which automatically tags memory. We also set BAP-PinTraces to taint command-line arguments to circumvent its fast-forward mechanism and measure its overhead for conducting taint analysis.

Compression tools First, the Taint Rabbit is evaluated on well-known compression utilities, including gzip, bzip2, pigz and pxz. Results are given in Figures 4a–4d. As input, all applications were given a file that is 9.2 MB large and contains random data. We note that TR-CC-ID and TR-CC-BV are substantially slower than their RAW counterparts. For instance, TR-CC-BV and TR-RAW-BV incur overheads of 258x and 3.7x respectively compared to native runs. The use of fast paths further enhances performance: TR-RAW-BV-FP reduces the overhead from 3.7x down to 2.3x. Results also show the positive impact of fast paths with respect to expensive clean-call implementations; TR-CC-BV-FP achieves 42.6x overhead. Consequently, fast paths also benefit users who wish to implement taint primitives using a high-level programming language.

Regarding existing tools, Dytan incurs a 308x overhead and is significantly surpassed by TR-CC-BV-FP and TR-RAW-BV-FP. All other generic engines, including DataTracker, are also slower than TR-RAW-BV-FP. For instance, DataTracker and DataTracker-EWAH incur overheads of 6.7x and 40.5x on gzip. By contrast, TR-RAW-BV-FP only results in an overhead of 1.8x. Triton suffers from the heaviest slowdown; we aborted Triton's run on bzip2 after 60 hours and therefore no result is reported. Owing to efficient bitwise tainting, LibDFT is faster than TR-RAW-BV-FP on average. It obtains 1.9x overhead as opposed to 2.3x achieved by TR-RAW-BV-FP.

PHP. The Taint Rabbit also ran on PHP 7.2.4 using PHPBench [32], a framework that provides a collection of micro/macro benchmarks for measuring performance. We ran the following benchmarks with 10,000 revolutions each: container, hashing, kde and statistics. The results are presented in Figures 4e–4h. An improvement is achieved by TR-RAW-BV-FP when compared to TR-CC-BV; the former achieves 94.9x while the latter incurs a staggering 806.4x overhead relative to native execution time. Fast paths also enhances performance for the clean call implementation, with TR-CC-BV-FP resulting in 187.5x.

Several taint engines, including Dytan and TR-CC-BV, crashed on the kde benchmark. The crash is caused by an integer overflow error, and we suspect that the slowdown imposed by taint analysis is to blame. We do not encounter this error when benchmarking performant engines such as LibDFT, TR-RAW-BV and TR-RAW-ID.

Another observation is that TR-RAW-BV and TR-RAW-ID perform faster than TR-RAW-BV-FP and TR-RAW-ID-FP, despite the use of fast paths. The issue is that TR-RAW-ID-FP fails to amortize many of its initial overheads, such as those posed by the dispatcher and the generation of fast paths pertaining to untaint cases. Since the majority of the PHP benchmarks take less than a second to execute natively, the optimizations of TR-RAW-ID-FP do not have enough time to be effective. Overall, this leads the overhead to increase from to 62.2x to 89.6x. Nevertheless, TR-RAW-ID-FP still outperforms all

[7] We ran the same Pin-Null tool provided by previous work [28].

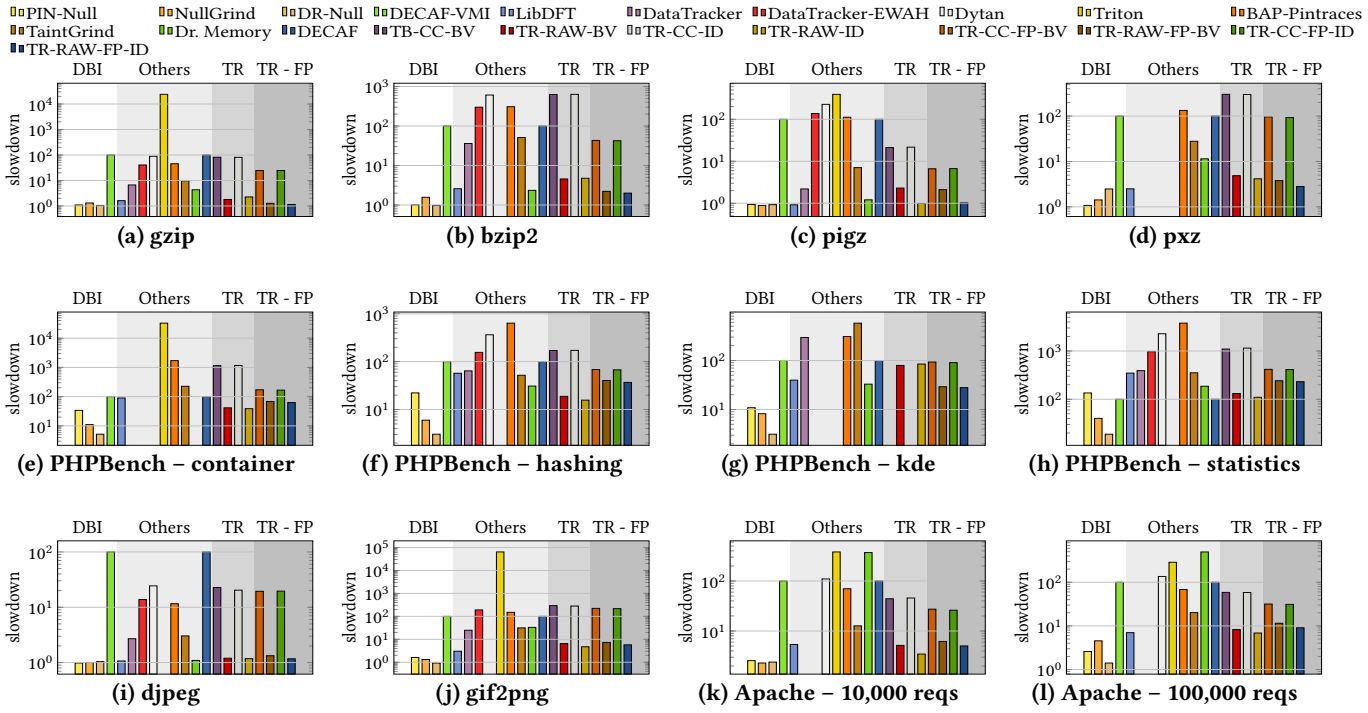


Figure 4: Results of the Taint Rabbit and other taint systems on command-line utilities, PHPBench and Apache. Missing entries imply that the corresponding taint engine timed-out or crashed.

other existing generic taint analyzers. For example, it is faster than DataTracker, which incurs 250.1x overhead.

Image Parsing. We consider djpeg and gif2png as two exemplars of image parsing. Results are given in Figures 4i–4j. Similar to the results presented so far, we again observe the high overheads incurred by existing generic taint engines. For instance, when running djpeg, DataTracker, DataTracker-EWAH, and Dytan achieve slowdowns of 2.7x, 13.8x and 24.3x respectively over native execution time. Meanwhile, `TR-RAW-BV-FP` yields better results with just an overhead of 1.3x. Interestingly, fast paths do not contribute to performance on this benchmark for RAW implementations as they are not significantly taken due to the high amount of taint data. The short one-second runtime of gif2png also renders generation difficult to amortize. Nevertheless, `TR-CC-BV-FP` achieves a lower overhead of 222.6x on gif2png in comparison to `TR-CC-BV`, which incurs 292x. Since clean call based instruction handlers are expensive, their elision is more effective in improving performance than those implemented in efficient assembly code.

Apache. Figures 4k–4l depict our results on Apache 2.4.33. The benchmark tool `ab` [21] was used to send 10,000 and 100,000 requests to Apache. Results again show that fast paths speed up taint engines implemented using clean calls. `TR-CC-BV-FP` results in 29.5x overhead over native runtime execution, which is less than the overheads of 51.4x and 122.8x incurred by `TR-CC-BV` and Dytan respectively. Moreover, `TR-RAW-ID-FP` is only slightly slower than LibDFT; the former obtains 7x overhead, while the latter achieves 6.1x. Since our experiments have Apache mainly deal with I/O,

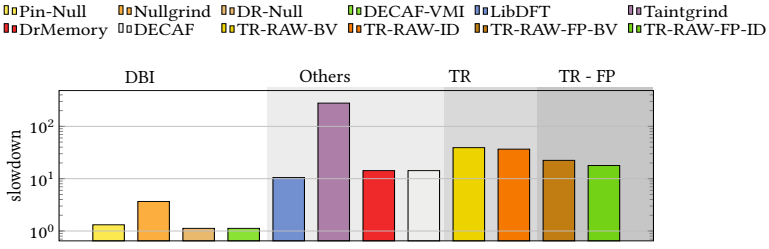


Figure 5: Average results on SPECrate 2017

Triton completes the experiment, albeit slowly with 334x overhead. DataTracker and DataTracker-EWAH both time-out after 3 hours.

SPEC CPU 2017. Average results obtained using the SPECrate 2017 Integer benchmark[8] are given in Figure 5. In particular, `TB-RAW-ID` achieves an overhead of 36.7x over native execution and is reduced to 17.9x when fast paths are enabled. It fails to out-perform specialized bitwise taint engines such as LibDFT, which achieves 10.5x. However, this is expected given its trade-off for versatility. Moreover, the Taint Rabbit is significantly faster than Taintgrind which obtains an overhead of 278x.

We excluded the gcc and x264 benchmarks when calculating results due to known limitations of the Taint Rabbit. First, the Taint Rabbit ran out of memory on gcc. This problem should be mitigated

[8]The new SPECint 2017 does not support 32-bit systems.

in the future when the Taint Rabbit supports 64-bit architectures. Secondly, TB-RAW-BV timed-out on x264 because of high overhead. Moreover, we terminated experimentation of TB-CC-ID as the duration of the first benchmark (i.e., perlbench) exceeded 24 hours. Because of the unmanageable overhead of the clean-call implementations of the taint analyzers, we focused on the optimized versions when running the SPECrate benchmark. DataTracker also faced issues as it crashed on all benchmarks except for one. The remaining benchmark, namely x264, exceeded 24 hours.

7.4 Dynamic Fast Path Generation

Table 6 found in the appendix provides statistics related to fast path execution. We gathered these statistics by running the training-sets of several SPEC CPU 2017 benchmarks. The first column specifies the percentage of basic blocks that were applicable for dynamic path generation. For instance, we exclude basic blocks consisting of just one instruction or those that do not contain data-flow. The second column details the average basic block size after truncation. The third column gives an approximation of the average number of instructions per basic block that elided instrumentation due to fast path generation. The fourth and fifth column indicate the total number of fast paths dynamically generated and reverts. The next three columns denote the execution counts of paths with no instrumentation, adaptive instrumentation, and full instrumentation respectively. Finally, the last two columns depict timelines relating to when fast paths were generated and executed during runtime.

Results show that execution dominantly takes fast paths. Although the most commonly executed path is that represented by the *no taint case*, generated fast paths are still notably taken. This is particularly the case for mcf. As one would expect, the number of fast paths generated is negligible to their number of executions.

Static vs. Dynamic Fast Paths. Lift [39] does not generate fast paths just-in-time. It is fixed to only consider fast paths that do not engage in taint propagation posed by the *no taint* case. If any input or output of a basic block is tainted, execution leads to the slow fully-instrumented path. We call this approach *static path generation*, because no other fast paths are constructed at runtime.

In relation to RQ1, we validate the performance benefits of dynamic fast path generation over the static variant by running again the same set of experiments described in Section 7.3. Unfortunately, Lift is not publicly available. Therefore, we modeled similar functionality by modifying the Taint Rabbit and switching off dynamic fast path generation. Average results are given in Figure 6. TR-CC-BV-FP-DYNAMIC outperforms TR-CC-BV-FP-STATIC on all considered benchmarks. For instance, results obtain using the compression benchmarks show that TR-CC-BV-FP-DYNAMIC achieves an overhead of 42.6x, while TR-CC-BV-FP-STATIC incurs 81.8x overhead relative to native execution time. Moreover, TR-RAW-BV-FP-DYNAMIC is faster than TR-RAW-BV-FP-STATIC on the compression benchmarks and SPEC CPU. TR-RAW-BV-FP-STATIC incurs 2.7x and 25x overheads on these benchmarks respectively. Meanwhile, TR-RAW-BV-FP-DYNAMIC improves with overheads of 2.3x and 22.4x.

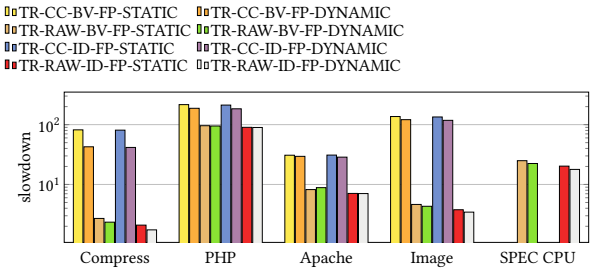


Figure 6: Static vs. Dynamic Fast Path Generation

Table 1: Results for detecting control-flow attacks

App.	CVE ID	TR-CHECK (CC)	TR-CHECK (RAW-FP)
RTF2Latex	2004-1293	3.75 s	0.8 s
RSync	2004-2093	0.26 s	0.63 s
Aeon	2005-1019	0.24 s	0.5 s
Nginx	2013-2028	1.6 s	1.94 s

7.5 Application-Specific Experiments

Apart from performance, we aim to validate the versatility of the Taint Rabbit. In particular, we show that it is feasible for our taint-primitive based approach to support three different taint policies to answer RQ4.

Control-Flow Hijacking Prevention. The first case is the detection of control-flow hijacking attacks. We use the Taint Rabbit to employ bitwise tainting similar to previous work [29]. The $src \rightarrow dst$ primitive does a move operation, while the $src, src \rightarrow dst$ and $src, src \rightarrow_M meet$ primitives perform a *bitwise or*. Table 1 presents the attacks detected by our tool called TR-CHECK. Although the short execution times of our benchmarks make amortization difficult, the Taint Rabbit has a faster mean detection time than our baseline. TR-CHECK-CC and TR-CHECK-RAW-FP result in average duration times of 1.46s and 0.97s respectively.

Use-After-Free Debugging. The second application, TR-UAF, uses taint analysis to track pointers and debug use-after-free vulnerabilities [8]. The 32-bit tags represent pointers to composite labels containing debugging information, and are propagated with primitives based on the policy described in Section 3. These labels are shared with tainted pointers derived from the same root address and reference counting is employed to manage their memory. Taint propagation is also turned off when entering allocation routines to prevent false triggers. Results are shown in Table 2. We validated the results by checking public bug reports and comparing alarms with those raised by Dr. Memory. Overall, it is feasible for taint primitives to support UAF bug detection.

Although our results suggest that Dr. Memory is faster than TR-UAF, the detection mechanisms which they employ are different and therefore performance cannot be directly compared. Unlike TR-UAF, Dr. Memory does not tag pointers to identify UAF vulnerabilities, but instead poisons freed memory regions.

Fuzzing. Although a thorough evaluation of the Taint Rabbit with respect to fuzzing is beyond the scope of this paper, we do show

Table 2: Time taken to detect UAF Vulnerabilities

App.	CVE ID	Dr. Memory	TR-UAF
V8	2017-5098	4.59 s	6.31 s
Yara	2017-5924	0.67 s	1.57 s
libzip	2017-12858	1.51 s	1.24 s
lrzip	2017-5924	1.84 s	2.47 s

Table 3: Bug counts achieved by VUzzer and TR-Fuzz

App.	Total Bugs	VUzzer	TR-Fuzz
base64	44	1	1
uniq	28	26	27
who	2136	33	55

that our approach could be used for such purposes. We replaced VUzzer's taint engine with our own custom tool that provides the same output, i.e., a list of file offsets which affect `lea` and comparison instructions. The rest of VUzzer's code, e.g., its mutator, is left untouched. Table 3 shows bug counts obtained on LAVA [19] by VUzzer and our version called `TR-Fuzz` in 6 hour runs.

7.6 Research Questions

RQ1: *How much does call-avoiding instrumentation and dynamic fast path generation improve the performance of generic taint analysis?*

Our results show that the Taint Rabbit brings its overhead of 224x down to 3.5x when call-avoiding propagation is enabled on benchmarks related to compression and image parsing. The optimization is effective because it essentially addresses the main bottleneck of expensive context-switching which is experienced by existing generic taint engines. Furthermore, fast paths alone reduces the overhead from 224x to 68.8x. The benefit of this optimization is its applicability as it can be employed to clean call based instruction handlers, thus avoiding the requirement of delving into low level assembly code. We observe positive synergy on these benchmarks when the two optimizations are used together bringing the overhead down to 2.7x. However, fast paths are mainly effective for long CPU-intensive applications where tainting does not comprehensively limit the execution of fast paths unlike witnessed for image parsing. On SPEC CPU, the overall overhead is reduced to 22.4x from 39.2x.

RQ2: *With these techniques, is the performance of generic taint analysis comparable to the state of the art of bitwise taint analysis?*

Inherently, specialized and generic taint engines have opposing performance and versatility trade-offs. The Taint Rabbit undergoes heavier analyses to support custom taint propagation logic. We therefore do not claim that the Taint Rabbit is faster than optimized bitwise tainting. On SPEC CPU, `TR-RAW-ID-FP` is slower than LibDFT with overheads of 17.6x and 10.5x respectively when compared to native runs. Nevertheless, the Taint Rabbit significantly reduces the performance gap that existed between the two types of analyses. On CPU-intensive benchmarks, related to compression and image parsing, which Dytan manages to run, we observe that it incurs 237x overhead. Meanwhile, LibDFT obtains 1.5x and the Taint Rabbit is slightly slower with an overhead of 1.7x.

RQ3: *Can the improved generic taint analysis scale to real-world target-applications?*

With our proposed optimizations, we show that generic taint analysis has advanced in performance in comparison to existing engines. For instance, DataTracker fails to run SPEC CPU, while the `TR-RAW-BV-FP` achieves an overhead of 22.4x. On smaller real-world benchmarks relating to compression and image parsing, the Taint Rabbit has an overhead of 2.8x, and therefore still outperforms DataTracker, which incurs a slower overhead of 14.5x. Unfortunately, like existing generic tools, we did encounter a case, namely SPEC CPU's gcc benchmark, where the Taint Rabbit crashed due to memory limitations. However, this limitation is exacerbated by our current implementation which is intended to analyze 32-bit software. Nevertheless, the Taint Rabbit provides a new opportunity to better scale expensive dynamic analyses when applied to large and CPU-intensive applications.

RQ4: *Does a primitive based approach improve the versatility of taint analysis?*

To answer `RQ4`, we demonstrate versatility through experimental validation. Our results indicate that it is feasible for user-defined primitives to support security applications concerning exploit detection, UAF debugging and fuzzing, all of which rely on different taint policies.

7.7 Threats to Validity

Our experimentation setup may have impacted our results. Particularly, we used an old version of Pin (i.e., 2.14) as LibDFT and Triton do not support newer, potentially faster, versions. To reduce this threat, we ran Pin-Null with the latest version of Pin (i.e., 3.7) on the SPEC CPU benchmark. Shown in the appendix, results indicate no major changes in performance (with an average overhead difference of 0.01x). Moreover, unlike many other tools which use Pin as a DBI framework, we instead use DynamoRIO. Therefore results cannot be directly correlated. However, the Taint Rabbit is faster than other generic taint engines with high margins which should exceed any performance benefits provided by the DBI framework. Nevertheless, we also implemented our own baseline, `TR-CC`, to mitigate the threat.

Moreover, while the Taint Rabbit is generic, there might exist a taint policy that is inappropriate for our taint-primitive-based algorithms. Our findings might also differ on other benchmarks. We addressed these threats by considering three different taint policies and a wide range of complex benchmarks.

8 RELATED WORK

Several works have focused on efficient bitwise-tainting [3, 12, 39]. LibDFT carefully implements its routines so that they are automatically inlined by DBI tools. Minemu [3] reduces register spillage by sacrificing SSE registers, which are assumed to being always dead, to store taint statuses. Lastly, Lift [39] proposes static fast paths with Davanian et al. [17] employing the approach system-wide. This is in contrast to our dynamic path generation. Such works present performant bitwise solutions but lack versatility.

Dytan [14] supports generic taint analysis that enables the user to define the custom merging of multiple tags stored in bit vectors. Unfortunately, routines are not optimized and suffer from high overhead. DECAF [24] performs bitwise-tainting inline to QEMU's TCG intermediate language, but maintains taint labels asynchronously via tracing at a slower pace. The Taint Rabbit aims towards generic taint analysis that is also optimized.

Other works [27, 42] perform preliminary analysis to reduce runtime overhead. Jee et al. [27] avoids instrumentation through code abstraction and TaintEraser [56] leverages taint summaries of standard API functions. These approaches are orthogonal to our work, and could further improve performance.

While the Taint Rabbit is an *online* taint tracker, other works [9, 15, 34, 35] propose *offline* variants where analysis is decoupled from the application's execution. FlowWalker [15] employs DBI to log traces and after runtime performs taint analysis. StraightTaint [34] takes a similar approach, but uses an efficient multi-threaded buffer to save data required for constructing the trace. Chabbi et al. [9] investigate taint analysis performed on a secondary shadow thread, which is in sync with the application's thread. Meanwhile, TaintPipe [35] has threads perform symbolic execution on code recently executed by the application until a concrete taint state is processed by an earlier spawned thread. Unlike the Taint Rabbit, these approaches face issues related to discrepancies in *time of attack* versus *time of detection*, or require expensive synchronization.

Iodine [1] also uses dynamic information to drive static analysis. Instrumentation is optimistically pruned such that it avoids rollbacks upon violations of likely runtime invariants. The Taint Rabbit instead uses dynamic information when performing forward dataflow analysis to generate fast paths. Iodine does not support binaries and depends on a prior profiling stage.

Similar to versatility, precision is also a trade-off for better performance, and in the context of pointer tracking, has fostered several discussions [16, 46, 47]. The Taint Rabbit works at the byte-level for speed, while Yadegari et al. [52] perform bit-level taint analysis to tackle obfuscation techniques.

Recently, Chua et al. [13] investigated synthesising propagation. The approach aims to reduce implementation effort, but the efficiency of the generated analyses remains unclear. Therefore, our work provides reciprocal benefits.

DBI Optimisations. Regarding DBI optimizations, Kleckner [30] reduces clean calls via partial inlining, while Wang et al. [51] extend the applicability of persistent code caching. Hawkins et al. [23] enhance DBI speed for JIT applications by using parallel memory mapping. Such approaches could further improve the Taint Rabbit.

9 CONCLUSION

In this work, we make several contributions towards generic taint analysis. First, call-avoiding instruction handlers and dynamic fast path generation are shown to be effective optimizations. Second, we demonstrate that our approach, based on taint primitives, is flexible enough to support different taint policies.

While our results indicate that the implementation of instruction handlers with the avoidance of clean calls delivers the highest performance gains, fast paths also provides additional speed-ups once amortized. The total speed up is substantial: Dytan achieves an overhead of 237x on CPU-intensive benchmarks concerning compression and image parsing when compared to native execution times, and our optimizations enable the Taint Rabbit to reduce that overhead to 1.7x. Overall, the techniques presented reduce the performance gap between generic and bitwise taint engines, and offer better scalability for difficult dynamic analyses.

ACKNOWLEDGEMENT

We would like to extend our sincere gratitude to Derek Bruening, Hendrik Greving and the rest of the DynamoRIO team for answering any queries we had on the DBI engine, and reviewing many of our pull requests. We also thank the researchers of other taint engines for making their tools available, and the anonymous reviewers for their invaluable feedback. This work is supported by EPSRC CDT in Cyber Security, VETSS, and the Endeavour Scholarship Scheme (partly financed by the European Social Fund (EU)).

REFERENCES

[1] Subarno Banerjee, David Devecsery, Peter M Chen, and Satish Narayanasamy. 2018. Iodine: Fast Dynamic Taint Tracking Using Rollback-free Optimistic Hybrid Analysis. In *Symposium on Security and Privacy*. IEEE.
[2] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. 2009. Scalable, behavior-based malware clustering. In *Network and Distributed System Security Symposium*. Internet Society, 8–11.
[3] Erik Bosman, Asia Slowinska, and Herbert Bos. 2011. Minemu: The World's Fastest Taint Tracker. In *International Workshop on Recent Advances in Intrusion Detection*. Springer LNCS, 1–20.
[4] Derek Bruening and Qin Zhao. 2011. Practical Memory Checking with Dr. Memory. In *Code Generation and Optimization*. IEEE, 213–223.
[5] Derek Bruening, Qin Zhao, and Saman Amarasinghe. 2012. Transparent Dynamic Instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*. 133–144.
[6] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. 2011. BAP: A binary analysis platform. In *International Conference on Computer Aided Verification*. Springer LNCS, 463–469.
[7] James Bucek, Klaus-Dieter Lange, et al. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *International Conference on Performance Engineering*. ACM, 41–42.
[8] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 2012. Undangle: Early Detection of Dangling Pointers in Use-after-free and Double-free Vulnerabilities. In *International Symposium on Software Testing and Analysis*. ACM, 133–143.
[9] Milind Chabbi, Somu Peritanayagam, Gregory Andrews, and Saumya Debray. 2007. *Efficient Dynamic Taint Analysis using Multicore Machines*. Master's thesis. The University of Arizona, Department of Computer Science.
[10] Walter Chang, Brandon Streiff, and Calvin Lin. 2008. Efficient and extensible security enforcement using dynamic data flow analysis. In *Computer and Communications Security*. ACM, 39–50.
[11] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *Symposium on Security and Privacy*. IEEE, 711–725.
[12] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. 2006. TaintTrace: Efficient flow tracing with dynamic binary rewriting. In *Computers and Communications,*. IEEE, 749–754.
[13] Zheng Leong Chua, Yanhao Wang, Teodora Baluta, Prateek Saxena, Zhenkai Liang, and Purui Su. 2019. One Engine To Serve'em All: Inferring Taint Rules Without Architectural Semantics. In *Network and Distributed System Security Symposium*. Internet Society.
[14] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: A Generic Dynamic Taint Analysis Framework. In *International Symposium on Software Testing and Analysis*. ACM, 196–206.
[15] Baojiang Cui, Fuwei Wang, Tao Guo, Guowei Dong, and Bing Zhao. 2013. FlowWalker: A Fast and Precise Off-Line Taint Analysis Framework. In *Emerging Intelligent Data and Web Technologies*. IEEE, 583–588.
[16] Michael Dalton, Hari Kannan, and Christos Kozyrakis. 2010. Tainting is not pointless. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 88–92.
[17] Ali Davanian, Zhenxiao Qi, and Yu Qu. 2019. DECAF++: Elastic Whole-System Dynamic Taint Analysis. In *RAID*. USENIX Association.
[18] Arnaldo Carvalho De Melo. 2010. The new Linux Perf tools. In *Slides from Linux Kongress*. https://pdfs.semanticscholar.org/16ca/fd05fa375dfe370274cd22b4c16c72d6c53b.pdf

[19] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-scale Automated Vulnerability Addition. In *Symposium on Security and Privacy*. IEEE, 110–121.
[20] DynamoRIO. 2017. Documentation: Clean Calls. http://dynamorio.org/docs/API_BT.html#sec_clean_call
[21] The Apache Software Foundation. [n. d.]. ab – Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html
[22] Brendan Gregg. 2016. The Flame Graph. *Commun. ACM* 59, 6 (2016), 48–57.
[23] Byron Hawkins, Brian Demsky, Derek Bruening, and Qin Zhao. 2015. Optimizing binary translation of dynamically generated code. In *Code Generation and Optimization*. IEEE, 68–78.
[24] Andrew Henderson, Lok Kwong Yan, Xunchao Hu, Aravind Prakash, Heng Yin, and Stephen McCamant. 2017. DECAF: A Platform-Neutral Whole-System Dynamic Binary Analysis Platform. *Transactions on Software Engineering* 43 (2017), 164–184. Issue 2.
[25] Intel. 2014. Intel 64 and IA-32 architectures optimization reference manual. *Intel Corporation*. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf
[26] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2010. An Empirical Study of Privacy-Violating Information Flows in JavaScript Web Applications. In *Computer and Communications Security*, Vol. 10. ACM, 270–283.
[27] Kangkook Jee, Georgios Portokalidis, Vasileios P Kemerlis, Soumyadeep Ghosh, David I August, and Angelos D Keromytis. 2012. A General Approach for Efficiently Accelerating Software-based Dynamic Data Flow Tracking on Commodity Hardware. In *Network and Distributed System Security Symposium*. Internet Society.
[28] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. 2012. LibDFT: Practical Dynamic Data Flow Tracking for Commodity Systems. In *ACM Sigplan Notices*, Vol. 47. ACM, 121–132.
[29] Wei Ming Khoo. 2012. Taintgrind: A taint-tracking plugin for the Valgrind memory checking tool. https://github.com/wmkhoo/taintgrind
[30] Reid Kleckner. 2011. *Optimization of Naïve Dynamic Binary Instrumentation Tools*. Master's thesis. Massachusetts Institute of Technology.
[31] David Korczynski and Heng Yin. 2017. Capturing Malware Propagations with Code Injections and Code-Reuse Attacks. In *Computer and Communications Security*. ACM, 1691–1708.
[32] Daniel Leech. 2015. PHPBench. https://phpbench.readthedocs.io/en/latest/index.html#
[33] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *ACM Sigplan Notices*, Vol. 40. ACM, 190–200.
[34] Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu. 2016. StraightTaint: Decoupled Offline Symbolic Taint Analysis. In *Automated Software Engineering*. ACM, 308–319.
[35] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. 2015. TaintPipe: Pipelined Symbolic Taint Analysis. In *USENIX Security Symposium*. 65–80.
[36] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM Sigplan Notices*. ACM, 89–100.
[37] James Newsome and Dawn Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Network and Distributed System Security Symposium*. Internet Society, 3–4.
[38] Pin. 2015. Pin 2.14 User Guide. https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html/
[39] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. 2006. Lift: A Low-overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Microarchitecture*. IEEE, 135–148.
[40] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Network and Distributed System Security Symposium*. Internet Society, 1–14.
[41] Florent Saudel and Jonathan Salwan. 2015. Triton: A Dynamic Symbolic Execution Framework. In *Symposium sur la sécurité des technologies de l'information et des communications*. SSTIC, 31–54.
[42] Prateek Saxena, R Sekar, and Varun Puranik. 2008. Efficient Fine-grained Binary Instrumentation with Applications to Taint-tracking. In *Code Generation and Optimization*. ACM, 74–83.
[43] Jonathan L Schilling. 2003. The Simplest Heuristics May be the Best in Java JIT Compilers. *ACM Sigplan Notices* 38, 2 (2003), 36–46.
[44] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might have been Afraid to Ask). In *Symposium on Security and Privacy*. IEEE, 317–331.
[45] Fermin J Serna. 2012. The Info Leak Era on Software Exploitation. *Black Hat USA*.
[46] Asia Slowinska and Herbert Bos. 2009. Pointless Tainting? Evaluating the Practicality of Pointer Tainting. In *European Conference on Computer systems*. ACM, 61–74.
[47] Asia Slowinska and Herbert Bos. 2010. Pointer Tainting Still Pointless (But We All See the Point of Tainting). *ACM SIGOPS Operating Systems Review* 44, 3 (2010), 88–92.
[48] Manolis Stamatogiannakis, Paul Groth, and Herbert Bos. 2014. Looking Inside the Black-box: Capturing Data Provenance using Dynamic Instrumentation. In *International Provenance and Annotation Workshop*. Springer LNCS, 155–167.
[49] Gang-Ryung Uh, Robert Cohn, Bharadwaj Yadavalli, Ramesh Peri, and Ravi Ayyagari. 2006. Analyzing Dynamic Binary Instrumentation Overhead. In *Workshop on Binary Instrumentation and Applications*.
[50] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2007. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Network and Distributed System Security Symposium*, Vol. 2007. Internet Society, 12.
[51] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Stephen McCamant. 2016. A General Persistent Code Caching Framework for Dynamic Binary Translation (DBT). In *USENIX Annual Technical Conference*. USENIX Association, 591–603.
[52] Babak Yadegari and Saumya Debray. 2014. Bit-level taint analysis. In *Source Code Analysis and Manipulation*. IEEE, 255–264.
[53] Heng Yin and Dawn Song. 2010. *TEMU: Binary Code Analysis via Whole-system Layered Annotative Execution*. Technical Report UCB/EECS-2010-3. EECS Department, University of California, Berkeley.
[54] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Computer and Communications Security*. ACM, 116–127.
[55] Qin Zhao, Derek Bruening, and Saman Amarasinghe. 2010. Umbra: Efficient and Scalable Memory Shadowing. In *Code Generation and Optimization*. ACM, 22–31.
[56] David Yu Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. 2011. TaintEraser: Protecting Sensitive Data Leaks using Application-level Taint Tracking. *ACM SIGOPS Operating Systems Review* 45, 1, 142–154.

APPENDIX

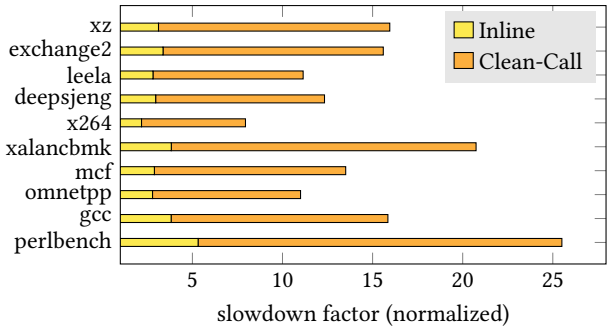


Figure 7: Peformance of clean call and inline instruction execution counters. The inline optimization is turned on/off via DynamoRIO's `-opt_cleancall` option.

Listing 1: An instruction handler of LibDFT. It propagates taint for an instruction, e.g., `add`, where two registers are sources and one is also the destination. Shadow memory, representing the bit flags of the registers, is stored in `vcpu`.

```
1 static void PIN_FAST_ANALYSIS_CALL
2 r2r_binary_opl(thread_ctx_t *thread_ctx, uint32_t dst, uint32_t
      ↪ src) {
3   thread_ctx->vcpu.gpr[dst] |= thread_ctx->vcpu.gpr[src];
4 }
```

Algorithm 3: Pointer tracking propagation [8]

Data: Taint labels s_1 and s_2
Result: Taint label d

```
if s1 = UNTAINT and s2 = UNTAINT then
    d ← 0;
else if s1 ≠ UNTAINT and s2 ≠ UNTAINT then
    d ← 0;
else if s1 ≠ UNTAINT then
    d ← s1;
else
    d ← s2;
end
return d;
```

Listing 2: An instruction handler of DataTracker. Propagation performs union operations on the sets associated with each source byte (lines 5–8). The function `set_union` is called via `tag_combine` at line 13.

```
1  static void PIN_FAST_ANALYSIS_CALL
2  r2r_binary_opl(thread_ctx_t *thread_ctx, uint32_t dst, uint32_t
       ↪ src)
3  {
4    ...
5    RTAG[dst][0] = tag_combine(dst_tag[0], src_tag[0]);
6    RTAG[dst][1] = tag_combine(dst_tag[1], src_tag[1]);
7    RTAG[dst][2] = tag_combine(dst_tag[2], src_tag[2]);
8    RTAG[dst][3] = tag_combine(dst_tag[3], src_tag[3]);
9  }
10
11 std::set<uint32_t> tag_combine(std::set<uint32_t> const & lhs,
       ↪ std::set<uint32_t> const & rhs) {
12   std::set<uint32_t> res;
13   std::set_union(lhs.begin(), lhs.end(),
14                  rhs.begin(), rhs.end(),
15                  std::inserter(res, res.begin()));
16 ...
```

Algorithm 4: Under-Approx. Tainting via $src \rightarrow dst$

Data: ID dst, ID src_1, Integer $opnd\_size$

```
for i ← 0 to opnd_size - 1 do
    src_label_1 ← lookup_label(src_1 + i);
    dst_label ← src_dst_primitive(src_label_1);
    set_label(dst + i, dst_label);
end
```

Algorithm 5: Over-Approx. Tainting via $src \rightarrow dst$

Data: ID dst, ID src_1, Integer $opnd\_size$

```
meet_label_1 ← UNTAINT;
for i ← 0 to opnd_size - 1 do
    label ← lookup_label(src_1 + i);
    meet_label_1 ← meet_primitive(meet_label_1, label)
end
for i ← 0 to opnd_size - 1 do
    dst_label ← src_dst_primitive(meet_label_1);
    set_label(dst + i, dst_label);
end
```

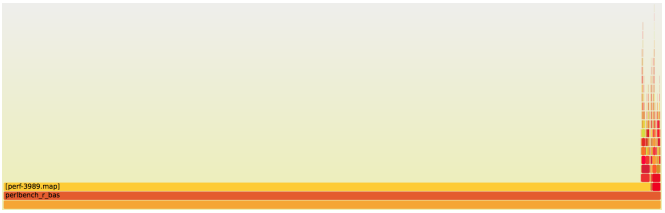


Figure 8: Perf [18], along with Flame Graphs [22], aided the profiling of the Taint Rabbit. Flame Graphs are stack-based diagrams designed to visualise frequent code traces where width represents the degree of presence in samples. The figure illustrates a recording of the Taint Rabbit on `perlbench`. Most of the flames are caused by basic block instrumentation. Meanwhile, the flat area represents execution in the DBI's code cache. Its dominance is a positive result as time is not being heavily spent on instrumentation (only 2% in this recording). Unfortunately, symbols required for generating the flames are not available as this code is JIT'ed. However, during development, the use of flame graphs helped discover a bottleneck related to faults and shadow memory. Moreover, we identified frequent code execution by mapping instruction addresses reported by perf with basic block information found in DynamoRIO debug logs as well as through disassembly via an attached debugger. This is not ideal, and therefore supporting code cache symbols and better profiling visualization pose as technical windows of opportunity for future work.

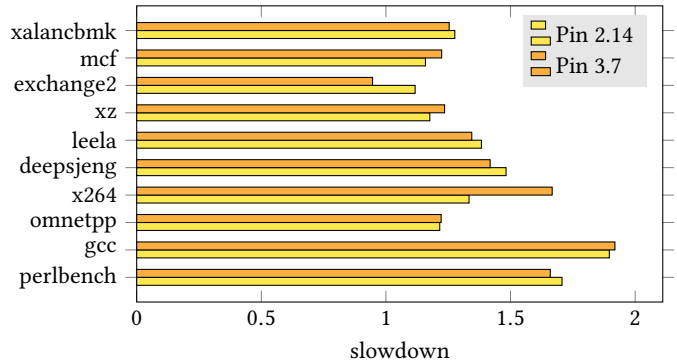


Figure 9: Performance of PinNull on Pin 2.14 and 3.7

Table 4: The list of instructions that the Taint Rabbit supported at the time of experimentation. Instructions, such as `jmp` and `prefetchnta`, which have no effect on taint propagation are not listed. Most of the missing instructions relate to floating-point and AVX. We plan to add support for additional instructions, and therefore this list may not reflect its current state.

add	or	adc	sbb	and	sub	xor	inc	dec	push	pop
imul	call	mov	lea	xchg	cwde	cdq	leave	rdtsc	cmovo	cmovno
cmovb	cmovnb	cmovz	cmovnz	cmovbe	cmovnbe	cmovs	cmovns	cmovp	cmovnp	cmovl
cmovnl	cmovle	cmovnle	punpcklbw	punpcklwd	punpckldq	packsswb	pcmpgtb	pcmpgtw	pcmpgtd	packuswb
punpckhbw	punpckhwd	punpckhdq	packssdw	punpcklqdq	punpckhqdq	movd	movq	movdqu	movdqa	pshufw
pshufd	pshufhw	pshuflw	pcmpeqb	pcmpeqw	pcmpeqd	seto	setno	setb	setnb	setz
setnz	setbe	setnbe	sets	setns	setp	setnp	setl	setnl	setle	setnle
shld	shrd	cmpxchg	movzx	bsf	bsr	movsx	xadd	pextrw	bswap	psrlw
psrld	psrlq	paddq	pmullw	pmovmskb	pminub	pand	pmaxub	pandn	psraw	psrad
pmulhuw	pmulhw	movntdq	pminsw	por	pmaxsw	pxor	psllw	pslld	psllq	pmaddwd
psubb	psubw	psubd	psubq	paddb	paddw	paddd	psrldq	pslldq	rol	ror
rcl	rcr	shl	shr	sar	not	neg	mul	div	idiv	movups
movupd	movlps	movlpd	movaps	andps	andpd	andnps	andnpd	orps	orpd	xorps
xorpd	movs	rep movs	stos	rep stos	lddqu	pshufb	palignr	lzcnt	pcmpeqq	movntdqa
packusdw	pcmpgtq	pminsd	pminuw	pminud	pmaxsb	pmaxsd	pmaxuw	pmaxud	pmulld	pextrb
pextrd	xgetbv	movq2dq	movdq2q	tzcnt	pext					

Table 5: Overview of the taint engines that are considered in our experimental comparison. [a]BAP Pin-Traces assigns a special constant integer value to handle taint merging.[b]In practice, the Taint Rabbit can overall be seen as an over-approximate taint engine, but it is a hybrid as instructions are handled precisely when possible.

Taint Engine	Granularity	Meta-Data	Union Operator	Approximation	DBI Platform
LibDFT [28]	Byte	Bit/Byte	Bitwise	Under	Pin
Triton [41]	Byte	Bool	Bitwise	Over	Pin
Dytan [14]	Byte	Bit-Vector	Generic	Under	Pin
DataTracker [48]	Byte	Set	Set Union	Under	Pin
DataTracker-EWAH [40]	Byte	Compressed Set	Set Union	Under	Pin
BAP-Pin Traces [6]	Byte	32-Bit Unsigned Offset	Set to *top*[a]	Over	Pin
Taintgrind [29]	Byte	Bit	Bitwise	Under	Valgrind
DECAF [24]	Bit	Bit	Bitwise	Precise	QEMU
Dr. Memory [4]	Byte	2 Bits	Bitwise	Under	DynamoRIO
Taint Rabbit	Byte	32-Bit Word	Generic	Hybrid[b]	DynamoRIO

Table 6: Dynamic Fast Path Generation

App.	% BB Instrum.	Avg. BB Size.	Avg. Instr Elided.	# FP Gen.	# Revert	# Exec. None	# Exec. FP	# Exec. Full	FP Gen. Timeline	Exec. FP Timeline
perlbench	81.0%	4	1	2009	1255	2.77E9	3.43E9	1.53E6		
mcf	82.3%	5	4	281	92	3.42E9	4.03E9	9.32E7		
xalancbmk	81.2%	4	3	213	57	3.51E9	3.14E9	1.43E9		
exchange2	81.2%	5	4	791	245	3.82E9	5.19E7	1.67E9		

(1) Truncation

```
1  mov eax, dword ptr [eax]
2  mov dword ptr [ebp], eax
3  mov ecx, dword ptr [ebx]
4  mov eax, dword ptr [ecx]
5  mov dword ptr [ebp], eax
6  mov eax, dword ptr [ebx]
7  jz 0xb7fdba45
```

(2) Duplication

```
1  UNTAINTED CASE LABEL
2  mov eax, dword ptr [eax]
3  mov dword ptr [ebp], eax
4  mov ecx, dword ptr [ebx]
5  jmp EXIT LABEL
6  TAINTED CASE LABEL
7  mov eax, dword ptr [eax]
8  mov dword ptr [ebp], eax
9  mov ecx, dword ptr [ebx]
10 EXIT LABEL
```

(3) Control Dispatch

```
1  ⟨TAINT CHECK CODE⟩
2  UNTAINTED CASE LABEL
3  cmp ecx, 0x00
4  jnz TAINTED CASE LABEL
5  mov eax, dword ptr [eax]
6  mov dword ptr [ebp], eax
7  mov ecx, dword ptr [ebx]
8  jmp EXIT LABEL
9  TAINTED CASE LABEL
10 cmp ecx, 0x15
11 jnz ⟨CLEAN CALL CODE⟩
12 mov eax, dword ptr [eax]
13 mov dword ptr [ebp], eax
14 mov ecx, dword ptr [ebx]
15 EXIT LABEL
```

(4) Default Cases

```
1  ⟨TAINT CHECK CODE⟩
2  UNTAINTED CASE LABEL
3  cmp ecx, 0x00
4  jnz TAINTED CASE LABEL
5  mov eax, dword ptr [eax]
6  mov dword ptr [ebp], eax
7  mov ecx, dword ptr [ebx]
8  jmp EXIT LABEL
9  TAINTED CASE LABEL
10 cmp ecx, 0x15
11 jnz ⟨CLEAN CALL CODE⟩
12 ⟨TAINT ANALYSIS CODE⟩
13 mov eax, dword ptr [eax]
14 ⟨TAINT ANALYSIS CODE⟩
15 mov dword ptr [ebp], eax
16 ⟨TAINT ANALYSIS CODE⟩
17 mov ecx, dword ptr [ebx]
18 EXIT LABEL
```

(5) Path Generation

```
1  ⟨TAINT CHECK CODE⟩
2  UNTAINTED CASE LABEL
3  cmp ecx, 0x00
4  jnz FAST PATH CASE LABEL
5  mov eax, dword ptr [eax]
6  mov dword ptr [ebp], eax
7  mov ecx, dword ptr [ebx]
8  jmp EXIT LABEL
9  FAST PATH CASE LABEL
10 cmp ecx, 0x04
11 jnz TAINTED CASE LABEL
12 mov eax, dword ptr [eax]
13 mov dword ptr [ebp], eax
14 ⟨TAINT ANALYSIS CODE⟩
15 mov ecx, dword ptr [ebx]
16 jmp EXIT LABEL
17 TAINTED CASE LABEL
18 cmp ecx, 0x15
19 jnz ⟨CLEAN CALL CODE⟩
20 ⟨TAINT ANALYSIS CODE⟩
21 mov eax, dword ptr [eax]
22 ⟨TAINT ANALYSIS CODE⟩
23 mov dword ptr [ebp], eax
24 ⟨TAINT ANALYSIS CODE⟩
25 mov ecx, dword ptr [ebx]
26 EXIT LABEL
```

Figure 10: A code example showing the instrumentation steps for fast path generation. The basic block is first truncated as the address in ecx is obtained via another memory access (at line 3). The remaining code is duplicated in the second step. Jumps and labels are also inserted. Taint checks and the control dispatcher are then inserted in step 3. At line 11, a clean call triggers path generation when control reaches the end of the compare and branch sequence. In step 4, the Taint Rabbit weaves analysis code for the two default cases (i.e. no taint and full taint instrumentation). Step 5 shows the inclusion of a generated path.