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
daniel.kroening@gmail.com

ABSTRACT

Generic taint analysis is a pivotal technique in software security. However, it 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.

CCS CONCEPTS

• **Security and privacy → Systems security**; **Software and application security**;

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 certain runtime properties hold, e.g., to detect if 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 that contain 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 the 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 generic taint analysis can be delivered with a runtime overhead that is low enough for practical security applications.

We argue that a combination of two optimizations is able to deliver taint tracking that is both versatile and sufficiently fast. 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 4 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 expensive context-switching by limiting function calls. Section 5 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]. Our results show 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. The DBI platform passes every new basic block observed during runtime to the Taint Rabbit for instrumentation. 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, the 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 for the case when taint is present. If the Taint Rabbit encounters a set of *in* and *out* taint states that are 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 in the given case are safely elided from instrumentation. Therefore, fully-instrumented code is executed less often than in conventional approaches as, owing to the additional fast paths, control is not always blindly directed to it when taint is encountered. 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 for these states pays off.

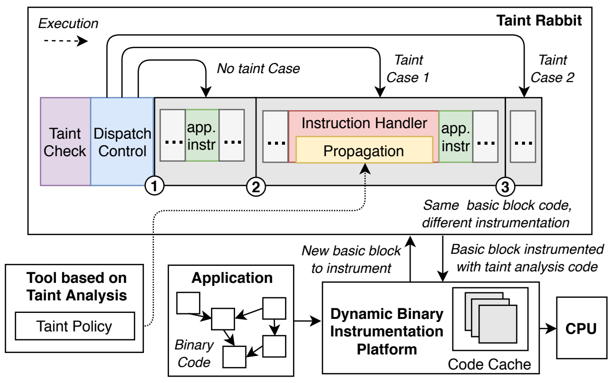

Execution
Taint Rabbit
No taint Case
Taint Case 1
Taint Case 2
Taint Check
Dispatch Control
app. instr
Instruction Handler
Propagation
app. instr
1
2
3
Same basic block code, different instrumentation
New basic block to instrument
Basic block instrumented with taint analysis code
Tool based on Taint Analysis
Taint Policy
Application
Binary Code
Dynamic Binary Instrumentation Platform
Code Cache
CPU

Figure 1: High-level design of the Taint Rabbit

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 of the internals of the engine.

3 BACKGROUND

3.1 Applications of Taint Analysis

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

Example 3.1. **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 status flags.

Example 3.2. **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 assigns the pointer status stored in the 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 *NO-TAINT*, *LIVE*, and *DANGLING*, and their merging cannot be appropriately done with the operation. Moreover, apart from the status of the pointer, 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.

Example 3.3. **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 as the input files of interest may be several kilobytes large. VUzzer therefore uses a bit array, which implies that the 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 [3, 12, 39]. Our approach is more versatile and suitable for all use cases.

3.2 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 the 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, combined 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.

4 THE TAINT RABBIT

Generic taint analysis enables user-defined taint policies. The support for custom merging of labels during propagation removes the need to change 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 considers the semantics of the instructions. This avoids tainting output locations unnecessarily, e.g., tainting stack pointers.

Generic Label Structure. The unit of meta-data that the Taint Rabbit uses as a label is a 32-bit word. The word may itself store tags or act as a pointer to a larger taint label data 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]. We do not label the x86 flag register to avoid *taint explosion* [56].

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

Three user-defined taint primitives are currently required for our supported instructions, and are informally defined as (1) $src \longrightarrow dst$, (2) $src, src \longrightarrow dst$, and (3) $src, src \longrightarrow_M meet$. 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 $src, src \longrightarrow dst$ primitive. Meanwhile, inspired by previous work [10], the third primitive computes the highest lower-bounded label for two given labels in a lattice.

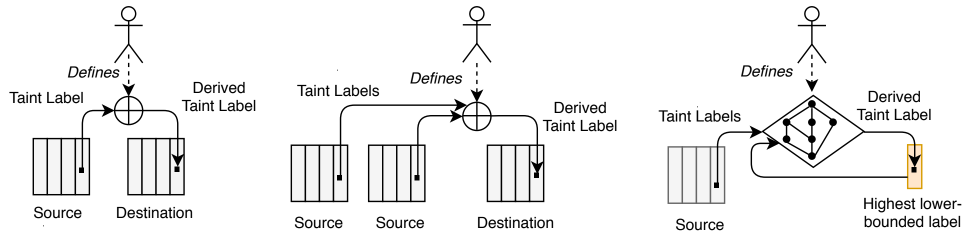
The taint primitives are the interface between the user-defined taint policy and the Taint Rabbit. We found that these three taint primitives are sufficient for our instruction handlers to track taint effectively and with reasonable precision, even for complex x86 instructions such as `punpckldq` and `pmaddwd`.

The primitives (2) and (3) serve very different purposes despite the fact that they have the same signature.

- The primitive $src, src \longrightarrow dst$ is used to combine the labels of two bytes stemming from different sources.

[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

[2]In contrast to the Taint Rabbit, Dytan [14] uses a bit vector as its label structure instead of a generic pointer. The number of bits is configurable at compile time.



(a) Primitive $src \longrightarrow dst$ **(b) Primitive** $src, src \longrightarrow dst$ **(c) Primitive** $src, src \longrightarrow_M meet$

Figure 2: User-defined primitives invoked by the Taint Rabbit to propagate taint

Algorithm 1: Computing the taint label for a two-operand instruction

```
Data: ID $dst$, ID $src_1$, ID $src_2$, Integer $opnd\_size$
1  $meet\_label_1 \leftarrow NULL$;
2  for $i \leftarrow 0$ to $opnd\_size - 1$ do
3      $label \leftarrow$ lookup_label($src_1 + i$);
4      $meet\_label_1 \leftarrow$ meet$_{primitive}$($meet\_label_1$, $label$)
5  end
6  $meet\_label_2 \leftarrow NULL$;
7  for $i \leftarrow 0$ to $opnd\_size - 1$ do
8      $label \leftarrow$ lookup_label($src_2 + i$);
9      $meet\_label_2 \leftarrow$ meet$_{primitive}$($meet\_label_2$, $label$)
10 end
11 for $i \leftarrow 0$ to $opnd\_size - 1$ do
12     $dst\_label \leftarrow$ src_src_dst$_{primitive}$($meet\_label_1$,
         $meet\_label_2$);
13     set_label($dst + i$, $dst\_label$);
14 end
```

- The primitive $src, src \longrightarrow_M meet$ is used to combine the labels of two bytes found within one source.

Essentially, the *meet* primitive provides means to compute a single label that summarizes the taint of a multi-byte operand. The two are incomparable. We recall the pointer tracking use-case (Example 3.2) to illustrate this point. Given two labels that represent a *LIVE* and *DANGLING* status, respectively, their *meet* is *DANGLING*, while the combination of two pointers is *NO-TAINT*.

Algorithm 1 specifies how the Taint Rabbit uses primitives (2) and (3) to compute the taint label for an instruction with two operands. The algorithm first iterates over the bytes of each operand separately and applies the *meet* primitive on these bytes. This yields one taint label for each of the two operands, denoted by $meet\_label_1$ and $meet\_label_2$, respectively.

In Line 11, the algorithm then iterates over the bytes of the destination operand, and uses the $src, src \longrightarrow dst$ primitive to combine the labels of the two source operands. The combined label is then assigned to the destination[3].

Optimization. For many x86 instructions, resulting bytes are independent. Instances of this are most transfer instructions (e.g. `mov`)

[3]We remark that the primitive may be stateful, and hence, its invocation is not hoisted out of the loop.

Algorithm 2: Optimized tainting for instructions with independent bytes

```
Data: ID $dst$, ID $src_1$, ID $src_2$, Integer $opnd\_size$
1  for $i \leftarrow 0$ to $opnd\_size - 1$ do
2      $src\_label_1 \leftarrow$ lookup_label($src_1 + i$);
3      $src\_label_2 \leftarrow$ lookup_label($src_2 + i$);
4      $dst\_label \leftarrow$ src_src_dst$_{primitive}$($src\_label_1$, $src\_label_2$);
5      set_label($dst + i$, $dst\_label$);
6  end
```

and many bit-manipulating instructions (e.g. `or`, `xor`). The semantics of these instructions guarantee that byte i of the result only depends on byte i of the first and byte i of the second operand. For this case, the two loops that merge the taint labels can be dropped. Algorithm 2 gives the resulting specialized instruction handler. Algorithm 2 is both faster than Algorithm 1 and produces a result that is more precise.

LibDFT uses the approach taken in Algorithm 2 even in cases when bytes may affect each other (e.g. `add`); it therefore under-approximates and may lose taint in return for a performance gain.

Algorithms 4 and 5, which illustrate the usage of the $src \longrightarrow dst$ 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`.

5 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.

5.1 Challenges

We address the following non-trivial challenges:

High Tracking Rate. Dynamic tainting 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 (excluding instructions such as `jmp` and `cmp`). We address this challenge in Section 5.2.

Expensive Context Switching. Unlike bitwise tainting, generic taint propagation is more complex, e.g., because of complex control flow. This leads to expensive context-switching incurred by clean calls. We address this challenge in Section 5.3.

5.2 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 given in Figure 10 in the appendix.

Truncation. When a new basic block is provided by the DBI platform, the Taint Rabbit begins by identifying 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 status cannot be checked prior to entering a fast path 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[4]. The cut-off code is no longer considered at this point, but is treated as a new separate basic block that undergoes its own analysis. The input and output operands are then retrieved and stored in a set by simply inspecting the remaining instructions found before the cut-off point.

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 maintain the one-exit-point property of basic blocks, control-flow instructions in the analyzed code of the application are not duplicated.

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. Fall-through implies that a new case is encountered, which is an opportunity for fast path generation. An unhandled case defaults to the execution of the fully instrumented basic block.

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 because of a large number of comparison instructions and cache pollution. The Taint Rabbit alleviates this issue by quickly checking registers via an over-approximation where a taint status bit is tracked for each register (as opposed to each byte in each register). Apart from conducting generic taint analysis, our instrumented paths also maintain these status bits. Therefore, a lot of the dispatcher's checking process is shifted down to paths that are less critical, away from the uninstrumented fast path. The idea of using over-approximate tags is similar to [42], but the Taint Rabbit cleverly uses the `pext` instruction [25] to construct the mask quickly. Although checks are imprecise owing to the higher granularity (sub-registers may considered as tainted when they are not), taint propagation is still performed by our byte-precise instruction handlers. Profiling done during development showed that the use of shared tags alone led to a speed-up of ~0.6x over native execution. Although shared tags are only associated to registers, the Taint Rabbit leverages SIMD instructions to efficiently test multiple labels simultaneously when taint checking memory.

[4]Our implementation reduces the impact of dynamic dependencies using constant propagation. For example, instead of truncating upon push and pop instructions, the Taint Rabbit patches operands with the offsets calculated by decrementing/incrementing the stack pointer.

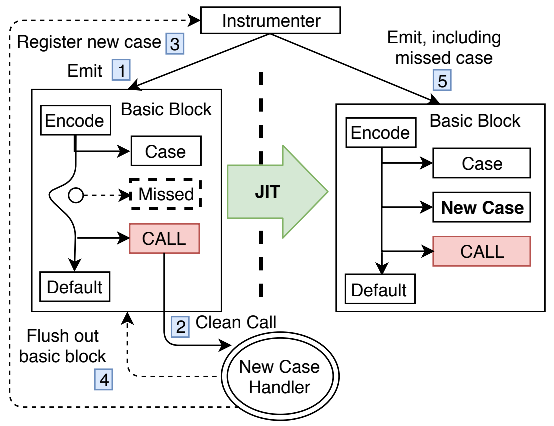


Figure 3: Dynamic Fast Path Generation

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 creating one fully instrumented basic block and maintaining another without any instrumentation at all. To handle other cases identified at runtime, forward data-flow analysis is performed on the basic block to determine which instructions deal with tainted operands. Such instructions propagate taint at runtime 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 that 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 the latency induced by flushing.

5.3 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 SIMD. We designed the code to follow known practices for optimization to the best of our abilities [25]. Iteration and branching are reduced with instruction handlers, amounting to over 970 in count, specialized not only to different opcodes but also to operand sizes and types. The loops in Algorithms 1 and 2 are unrolled.

Instruction handlers do not use a stack but rely on thread-local storage and registers for memory. Although Algorithm 1 is slower than Algorithm 2, owing to the *meet* primitives, it is only used for certain instructions, as described in Section 4. The taint primitives are given memory operands that refer to taint labels and two general purpose (GP) scratch registers for their implementation. Spillage is performed if the primitive requires additional registers.

5.4 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.

5.5 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 the code of basic blocks. In turn, `drbbdup` is used to implement fast path generation. The majority of drbbdup's code has been merged into DynamoRIO's repository[5]. Lastly, several tools, including Perf [18], were leveraged to profile the Taint Rabbit. Analysis results, visualized via flame graphs [22], are given in the appendix (Figure 8).

5.6 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 a like-for-like experimental comparison reduces the threat to validity. Moreover, the Taint Rabbit does not support some of the FPU instructions. Table 4 (in the appendix) 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.

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

Truncation of basic blocks removes the need for a static whole-program pointer analysis. However, this is not a perfect solution as the number of basic blocks 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 for data-flow analysis, and are always instrumented.

The Taint Rabbit uses additional memory, and the memory overhead may cause issues when analyzing large applications. The memory overhead is primarily caused by Taint Rabbit's 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.

6 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`: Do taint primitives enable generic taint analysis?

We ran the experiments on 32-bit Ubuntu 14.04 machines, each equipped with an 8 core 2.60 GHz Intel Core i7-6700HQ CPU and 32 GB RAM. Full results with numerical figures[6], along with scripts for running many of our experiments[7], are available online. The specific version of the Dynamic Rabbits that we used for our experiments is available as well[8].

6.1 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`.

[5]https://github.com/DynamoRIO/dynamorio/commit/6195c00

[6]https://docs.google.com/spreadsheets/d/1gAm7GJBB3Rl4bfTwWq-ITcNyVuRtTH2vQYYaS3n-OUk

[7]https://github.com/Dynamic-Rabbits/Taint-Evaluator/commit/e594963

[8]https://github.com/Dynamic-Rabbits/Dynamic-Rabbits/commit/56f9e2b9

We perform our experiments using two taint policies. The first policy (TR-ID) assigns a new numerical ID to each destination byte whenever taint propagation occurs. This policy could serve as a basis for a static single assignment trace generator. ID assignment is achieved by using *src* $\longrightarrow$ *dst* and *src*, *src* $\longrightarrow$ *dst* primitives that 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* $\longrightarrow$ *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, these two taint policies cannot be implemented with a bitwise taint engine.

6.2 Other Taint-Based Systems

To answer RQ1 and RQ2, 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 3.1415 alpha [28] inlines bitwise taint analysis, while Dytan[9][10] [14] performs user-defined operations on bit vectors that contain multiple tags. Triton 0.6 [41] is a dynamic binary analysis framework which is set up to use Pin 2.14 for tracing. DataTracker[11] [48] focuses on data provenance; a variant, named DataTracker-EWAH[12] [40], records input offsets to optimize fuzzing. We also ran BAP-PinTraces[13] [6], which generates execution logs of instructions that deal with taint. Its taint propagation routines are not implemented specifically to the semantics of the instructions, but instead leverage the IR of the DBI to determine the source and destination operands. DECAF[14] [24] is a QEMU-based taint tracker that inlines precise bitwise propagation into Tiny Code Generator (TCG) instructions, and Taintgrind 3.15.0 [29] is a taint engine built upon Valgrind [36]. Moreover, the Dr. Memory 2.1.17972[15] [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 to reduce the overhead. DBI overhead was also measured separately without taint analysis. We give results for Pin 2.12, DynamoRIO 7.1 and Valgrind 3.13.0, labeled as Pin-Null[16], DR-Null and Nullgrind, respectively. DECAF, just using its virtual machine introspection and with no taint analysis, is labeled as DECAF-VMI.

[9]https://github.com/dytan-taint-tracking/dytan-taint-tracking/commit/5211823d575

[10]We modified Dytan by removing failing assertions for unsupported x86 instructions.

[11]https://github.com/m000/dtracker/commit/dc729dca8

[12]https://github.com/vusec/vuzzer/commit/f6f7d593a

[13]https://github.com/BinaryAnalysisPlatform/bap-pintraces/commit/bed2b108

[14]https://github.com/decaf-project/DECAF/commit/1de4ed7c95

[15]https://github.com/DynamoRIO/drmemory/commit/5b988e31

[16]We ran the same Pin-Null tool provided by LibDFT 3.1415 alpha [28].

6.3 Performance

To answer RQ1, RQ2 and RQ3, we measure the Taint Rabbit's performance on benchmarks relating to data compression, PHP, image parsing, Apache and SPEC CPU. For these experiments, we configure 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 taint prevalence[17]. No taint is introduced when running other tools, which nevertheless instrument instructions even though no taint is propagated. However, our setup benefits instruction handlers that perform efficiently when dealing with untainted data only. There are exceptions, e.g., Dr. Memory, which automatically tags memory. We set BAP-PinTraces to taint command-line arguments to circumvent its fast-forward mechanism and measure the overhead of its taint analysis.

Data Compression. First, we evaluate the Taint Rabbit on well-known compression utilities, including gzip 1.6, bzip2 1.0.6, pigz 2.3 and pxz 4.999.9 beta. Results are given in Figures 4a–4d. As input, all applications were given a file that is 9.6M 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 implement taint primitives using a high-level programming language.

Comparing to 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 1.3x. 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 achieves 1.9x overhead, as opposed to 2.3x achieved by TR-RAW-BV-FP.

PHP. We have applied the Taint Rabbit to PHP 7.2.4, driven by PHPBench[18] 0.15.0 [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 enhance 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 suggests that the slowdown imposed by taint analysis

[17]Figure 11 in the appendix gives performance results when taint is sampled.

[18]https://github.com/phpbench/phpbench/commit/04a1a1b

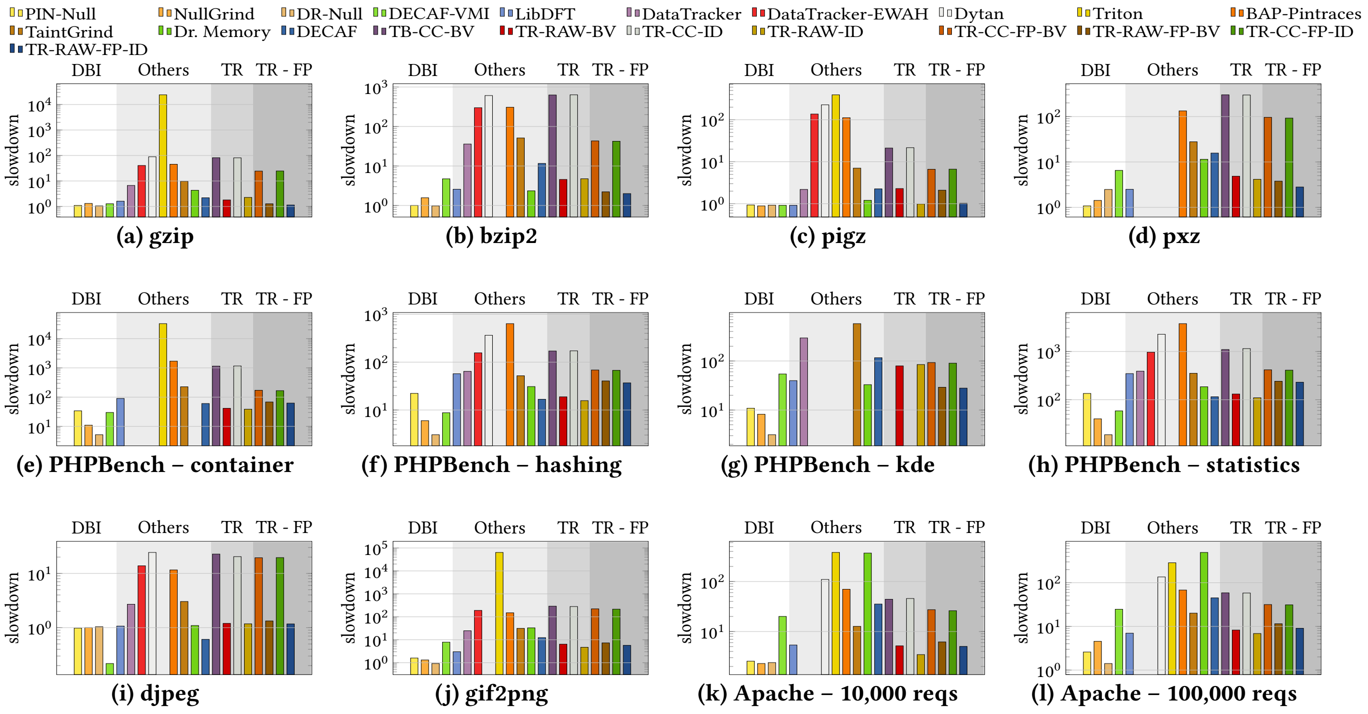


Figure 4: Results of the Taint Rabbit and other taint systems on command-line utilities, PHPBench and Apache. Missing entries imply that the corresponding taint engine timed-out or crashed.

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 increases the overhead from to 62.2x to 89.6x. Nevertheless, TR-RAW-ID-FP still outperforms all other existing generic taint analyzers. For example, it is faster than DataTracker, which incurs 250.1x overhead.

Image Parsing. We consider djpeg, version 9c, and gif2png 2.5.8 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 large amount of tainted 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

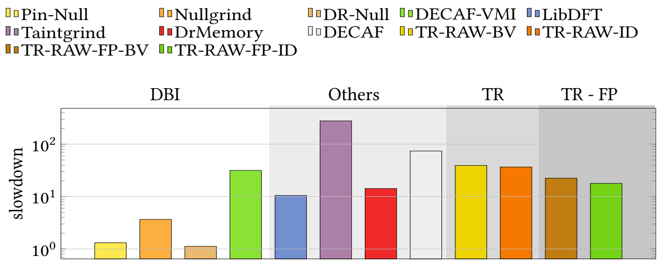


Figure 5: Average overheads on SPECrate 2017

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 incurs 7x overhead, while the latter achieves 6.1x. In our experiments, Apache primarily performs I/O and process forks to handle requests, and hence Triton is able to complete the experiment, albeit with 334x overhead. DataTracker and DataTracker-EWAH both time-out after 3 hours.

SPEC CPU 2017. The average overheads observed on the SPECrate 2017 Integer benchmark 1.0.2[19] are given in Figure 5. `TB-RAW-ID` achieves an overhead of 36.7x over native execution, which 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 and DECAF, which incur overheads of 278x and 74.1x respectively.

We excluded the `gcc` and `x264` benchmarks when calculating results because of known limitations of the Taint Rabbit. First, the Taint Rabbit ran out of memory on `gcc` for both TR-ID and TR-BV. This problem can be mitigated by using a 64-bit architecture. Second, `TB-RAW-BV` times-out on `x264` because of high overhead. Moreover, we terminated experimentation with `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.

6.4 Dynamic Fast Path Generation

Table 6 (in the appendix) gives insight about the benefit of fast path generation. We gathered these measurements mainly by executing the training sets of several SPEC CPU 2017 benchmarks. The second column gives the percentage of basic blocks where dynamic path generation is applicable. For instance, we exclude basic blocks consisting of just one instruction or those that do not contain data-flow. The third column details the average basic block size after truncation. The fourth column gives an approximation of the average number of instructions per basic block that elided instrumentation due to fast path generation. The fifth and sixth 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 the timelines when fast paths were generated and executed during runtime.

The results show that execution dominantly takes fast paths. Although the most commonly executed path is the *no taint* case, generated fast paths are executed frequently, particularly for `mcf`. As one would expect, the number of fast paths generated is negligible when compared to the number of times they are executed.

Static vs. Dynamic Fast Paths. Lift [39] does not generate fast paths just-in-time. It is fixed to only consider fast paths that never engage in taint propagation represented 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.

To answer `RQ1`, we quantify the performance benefits of dynamic fast path generation compared to the static variant by running the same set of experiments described in Section 6.3. Unfortunately, Lift is not publicly available. Therefore, we modeled similar functionality by modifying the Taint Rabbit and switching off dynamic

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

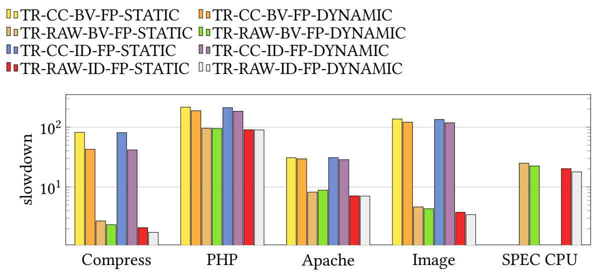


Figure 6: Static vs. Dynamic Fast Path Generation

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 obtained 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 performance with overheads of 2.3x and 22.4x.

Performance impact of the number of Fast Paths. In order to observe the relationship between performance and the number of possible fast paths that the Taint Rabbit can generate per basic block, we ran our experiments with varying limits. Once the limit is reached, the Taint Rabbit no longer monitors and attempts fast path generation for the block. Our results are in Figures 12a–12d (in the appendix). They indicate that the performance impact of fast paths highly depends on whether the costs of monitor checks and fast path generation are amortized. In particular, the generation of multiple fast paths gradually improves the performance of the Taint Rabbit on the compute-intensive SPEC CPU 2017 benchmark. However, applications, such as PHP, that incur heavy costs during the instrumentation process do not benefit.

6.5 Application-Specific Experiments

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

Control-Flow Hijacking Prevention. The first use case is the detection of control-flow hijacking attacks. We configure the Taint Rabbit to perform bitwise tainting similar to previous work [29]. The $src \longrightarrow dst$ primitive does a move operation, while the $src, src \longrightarrow dst$ and $src, src \longrightarrow_M meet$ 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 1.46 s and 0.97 s, 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 given as Example 3.2. These labels

Table 1: Results for detecting control-flow attacks

Application	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

Table 2: Time taken to detect UAF Vulnerabilities

Application	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
libzip	2017-12858	0.51 s	0.44 s
libwebm	2018-6548	1.19 s	2.47 s
libsass	2018-19827	2.88 s	3.68 s
imagemagic	2019-19952	57.29 s	6.12 s

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

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

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 given in Table 2. We validated the results by checking public bug reports and comparing alarms with those raised by Dr. Memory. In summary, UAF bug detection can be performed using our generic taint analyzer.

Although our results indicate 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. Dr. Memory also tracks addressable data via bitwise tainting.

Fuzzing. Although a thorough evaluation of the application of Taint Rabbit to fuzzing is beyond the scope of this paper, we demonstrate that our approach could be used for this purpose. We replaced VUzzer's taint engine with our own custom tool that provides the same output, i.e., a list of file offsets that affect `lea` and comparison instructions. The rest of VUzzer's code, e.g. its mutator, is left untouched. Table 3 gives the bug counts obtained on LAVA [19] by VUzzer and our version, `TR-Fuzz`, in 6 hour runs.

6.6 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 overhead of the Taint Rabbit is reduced from 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 reduce the overhead from 224x to 68.8x. The benefit of this optimization is its broad applicability as it can be used with clean call based instruction handlers, thus removing the need to write low-level assembly code. We observe positive synergy on these benchmarks when the two optimizations are used together: the overhead is further reduced to 3x. However, fast paths are mainly effective for long-running, CPU-bound applications where tainting does not comprehensively limit the execution of fast paths, unlike witnessed when parsing images. 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 has to perform heavier analyses to support custom taint propagation logic. We therefore cannot expect that the Taint Rabbit is faster than optimized bitwise tainting. On SPEC CPU, `TR-RAW-ID-FP` is slower than LibDFT with overheads of 17.8x 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 the CPU-bound benchmarks (compression and image parsing) that Dytan manages to complete, Dytan incurs 237x overhead. By contrast, LibDFT incurs 1.5x and the Taint Rabbit is only slightly slower with an overhead of 1.7x.

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

We argue that our proposed optimizations increase the performance of generic taint analysis to the point that real-world target applications can be analyzed. For instance, DataTracker fails to run SPEC CPU, while `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 outperforms DataTracker significantly, which incurs an overhead of 14.5x. Unfortunately, like the existing generic tools, we did encounter one case, namely SPEC CPU's `gcc` benchmark, where the Taint Rabbit crashed because of 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-bound applications.

RQ4: *Do taint primitives enable generic taint analysis?*

To answer `RQ4`, we demonstrate versatility by considering a variety of exemplars. 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.

6.7 Threats to Validity

Our experimental setup may have impacted our results. Particularly, we use old versions of Pin (i.e., 2.12 and 2.14) since LibDFT and Triton do not support newer, potentially faster, versions. To reduce this threat, we ran Pin-Null with a recent version of Pin (3.7, released in 2018) on SPEC CPU. The results (given in the appendix) indicate no major changes in performance (with an average overhead difference of 0.01x). Moreover, unlike many other tools that use Pin as a DBI framework, we use DynamoRIO. Therefore results cannot be compared directly. 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. We also built our own baseline, `TR-CC`, to mitigate this threat.

Moreover, there might exist a taint policy that cannot be implemented effectively using the interfaces between Taint Rabbit and the taint primitives. This would impact our claim that the Taint Rabbit delivers generic taint analysis. We are also exposed to the problem of benchmark bias, i.e., our findings might not generalize to further benchmarks. We have addressed these two threats by considering three different taint-based applications and a wide range of complex benchmarks.

7 RELATED WORK

There is a substantial body of work on improving the efficiency of bitwise taint analysis [3, 12, 39]. LibDFT includes carefully implemented routines so that they are automatically inlined by DBI tools. Minemu [3] reduces register spillage by sacrificing the SSE registers, which are assumed to be dead, to store taint status. Lastly, unlike our dynamic approach, Lift [39] uses static fast paths, and Davanian et al. [17] apply this approach system-wide. Overall, these works present performant bitwise solutions but lack the versatility of generic taint analysis.

Dytan [14] supports generic taint analysis that enables the user to define custom merging policies for multiple tags stored in bit vectors. Unfortunately, its 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 performs generic taint analysis that is also optimized.

Other works [27, 42] perform preliminary analysis to reduce runtime overhead. Jee et al. [27] avoid instrumentation by means of 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] uses threads that perform symbolic execution on code recently executed by the application until a concrete taint state is processed by a thread spawned earlier. 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. 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 the speed of DBI for JIT applications by using parallel memory mapping. Such approaches could further improve the Taint Rabbit.

8 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 a variety of taint policies.

While our results indicate that avoiding clean calls when executing instruction handlers delivers the highest performance improvements, fast paths also provides additional speed-ups once amortized. The total speed up is substantial: Dytan achieves an overhead of 237x on CPU-bound 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 about the DBI engine, and reviewing our pull requests. We also thank the creators of other taint engines for making their tools available, and the anonymous reviewers for their invaluable feedback. This work is supported by the EPSRC CDT in Cyber Security, VETSS and the Endeavour Scholarship Scheme (partly financed by the European Social Fund).

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. 2016. Intel 64 and IA-32 architectures software developer's manual. *Intel Corporation*. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.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.

[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*. Internet Society.

[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

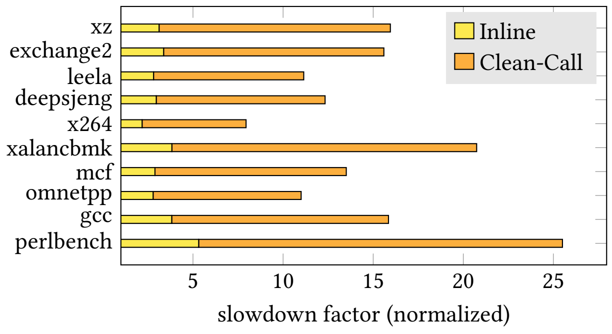


Figure 7: Peformance of clean call and inline instruction execution counters. The inline optimization is turned on/off via DynamoRIO's `-opt_cleancall` option.

Algorithm 3: Pointer tracking propagation [8]

```
Data: Taint labels $s_1$ and $s_2$
Result: Taint label $d$
if $s_1 = NULL$ and $s_2 = NULL$ then
    $d \leftarrow$ NULL;
else if $s_1 \neq NULL$ and $s_2 \neq NULL$ then
    $d \leftarrow$ NULL;
else if $s_1 \neq NULL$ then
    $d \leftarrow s_1$;
else
    $d \leftarrow s_2$;
end
return $d$;
```

Listing 1: An instruction handler of LibDFT. It propagates taint using a *bitwise or* for an instruction, e.g., `add`, where two registers are sources and one is also the destination.

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

Listing 2: An instruction handler of DataTracker. Propagation performs union operations on the sets associated with each source byte (lines 5–8). `tag_combine` calls `set_union` 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(), rhs.begin(), rhs.end(),
14                   std::inserter(res, res.begin()));
15  ...
```

Algorithm 4: Taint propagation for one source operand

```
Data: ID $dst$, ID $src_1$, Integer $opnd\_size$
$meet\_label_1 \leftarrow NULL$;
for $i \leftarrow 0$ to $opnd\_size - 1$ do
    $label \leftarrow$ lookup_label($src_1 + i$);
    $meet\_label_1 \leftarrow$ meet$_{primitive}$($meet\_label_1$, $label$)
end
for $i \leftarrow 0$ to $opnd\_size - 1$ do
    $dst\_label \leftarrow$ src_dst$_{primitive}$($meet\_label_1$);
    set_label($dst + i$, $dst\_label$);
end
```

Algorithm 5: Optimized taint propagation for one source operand with independent bytes

```
Data: ID $dst$, ID $src_1$, Integer $opnd\_size$
for $i \leftarrow 0$ to $opnd\_size - 1$ do
    $src\_label_1 \leftarrow$ lookup_label($src_1 + i$);
    $dst\_label \leftarrow$ src_dst$_{primitive}$($src\_label_1$);
    set_label($dst + i$, $dst\_label$);
end
```

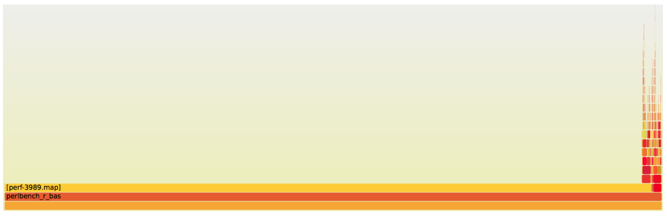


Figure 8: Perf [18], along with Flame Graphs [22], aided the profiling of the Taint Rabbit. The figure illustrates a recording of the Taint Rabbit on `perlbench`. Most of the flames are caused by basic block instrumentation. Meanwhile, the flat area represents execution in the DBI's code cache. Its dominance is a positive result as time is not 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, profiling helped discover a bottleneck related to shadow memory during development.

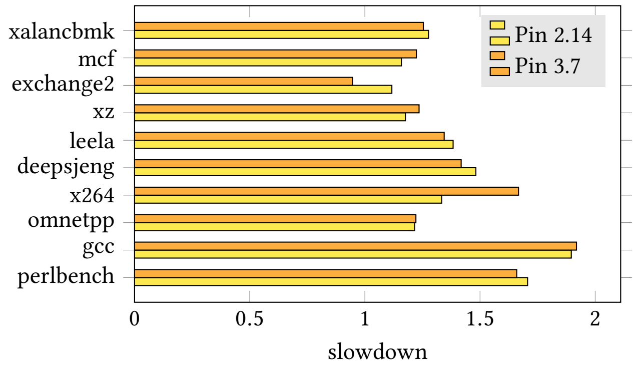


Figure 9: Performance of PinNull on Pin 2.14 and 3.7

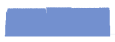
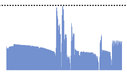
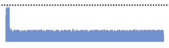
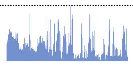
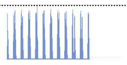
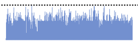
Table 4: The list of instructions that the Taint Rabbit supported at the time of experimentation. Instructions, such as `jmp` and `prefetchnta`, that have no effect on taint propagation are not listed. Most of the missing instructions relate to floating-point arithmetic and AVX. We plan to add support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

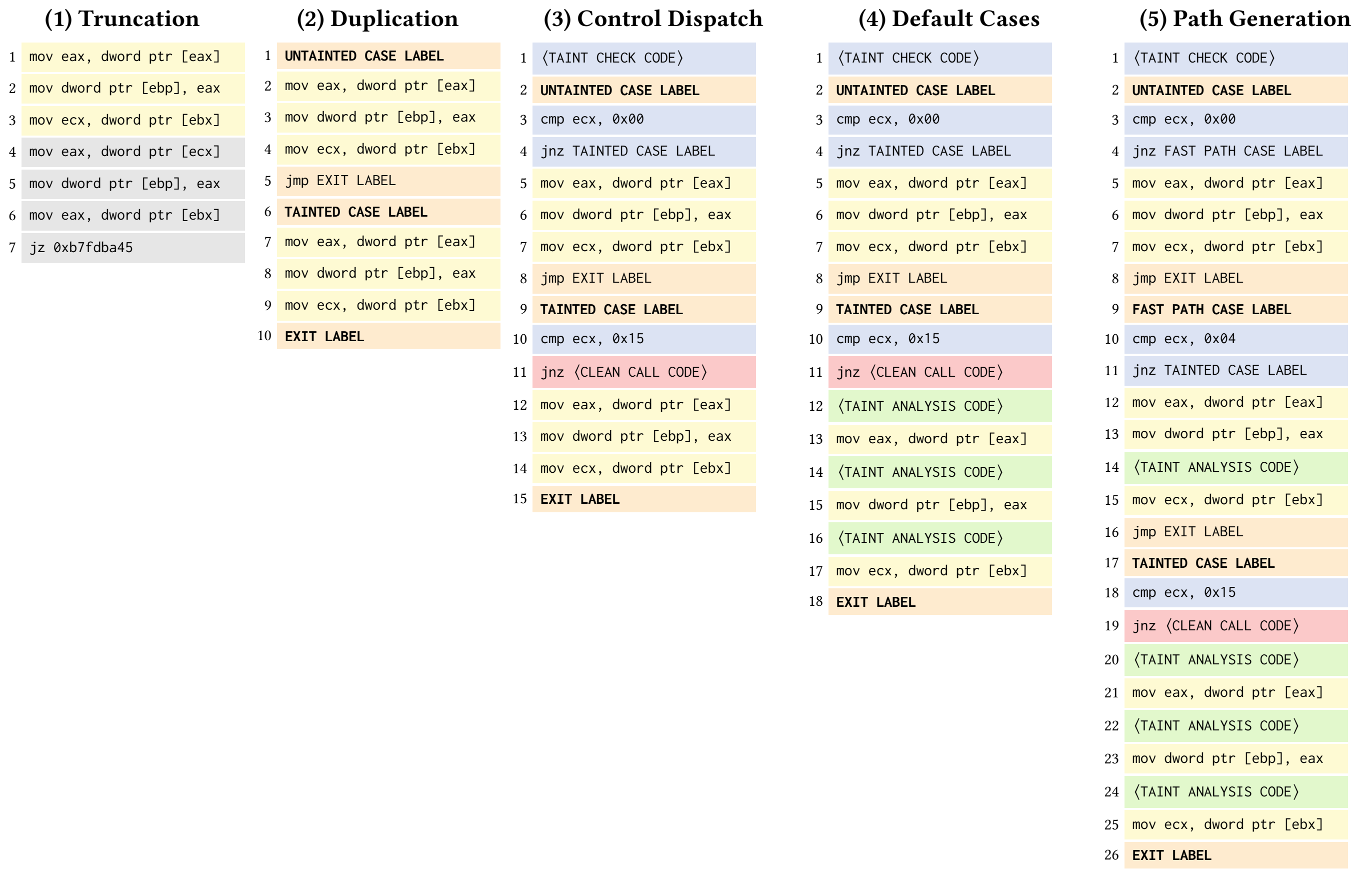
`add`	`or`	`adc`	`sbb`	`and`	`sub`	`xor`	`inc`	`dec`	`push`	`pop`
`imul`	`call`	`mov`	`lea`	`xchg`	`cwde`	`cdq`	`leave`	`rdtsc`	`cmovo`	`cmovno`
`cmovb`	`cmovnb`	`cmovz`	`cmovnz`	`cmovbe`	`cmovnbe`	`cmovs`	`cmovns`	`cmovp`	`cmovnp`	`cmovl`
`cmovnl`	`cmovle`	`cmovnle`	`punpcklbw`	`punpcklwd`	`punpckldq`	`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 considered in our experimental comparison. [a]BAP Pin-Traces assigns a special constant integer value to indicate merged taint.

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	Over	QEMU
Dr. Memory [4]	Byte	2 Bits	Bitwise	Under	DynamoRIO
Taint Rabbit	Byte	32-Bit Word	Generic	Over	DynamoRIO

Table 6: Statistics related to Dynamic Fast Path Generation. Generation and execution timelines are relative to each other. Generation counts are less than execution counts and therefore are hardly visible on the timeline.

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		
bzip2	87.4%	4	3	510	77	1.69E9	2.82E8	6.27E7		
djpeg	84%	4	1	141	73	1.49E8	6.42E8	7.33E8		



(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.

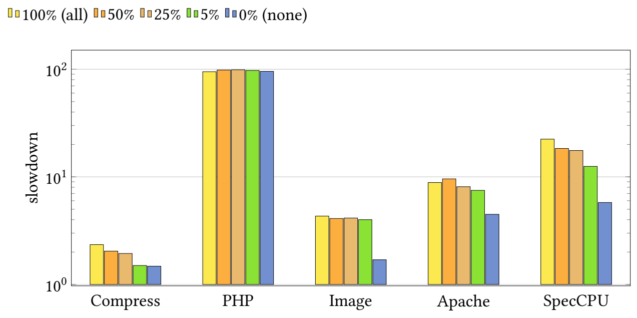


Figure 11: The Taint Rabbit achieves better performance when less taint is introduced because more fast paths are executed. To aid validate this claim, we ran the same experiments but randomly sampled taint introduction based on various probabilities. For instance, on SPEC CPU, overall overhead is reduces to 12.5x from 22.4x when the odds of introducing taint is set to $\frac{1}{25}$.

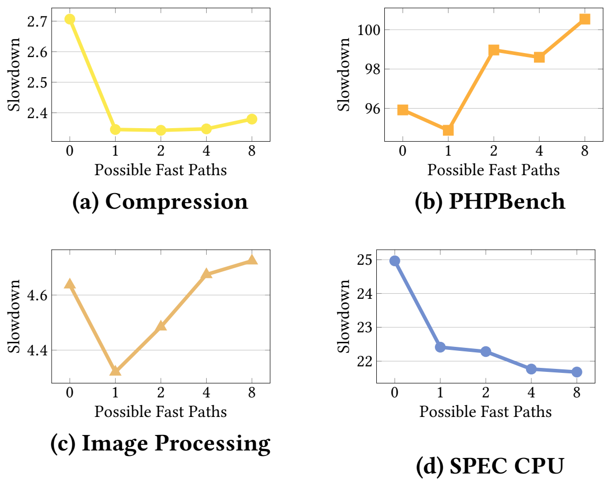


Figure 12: Overhead vs. Number of Possible Fast Paths