Optimizing a Primitive Based Approach to Generic Taint Analysis

John Galea

St Cross College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy

Michaelmas 2020

Abstract

Dynamic taint analysis is a fundamental technique in software security that tracks the flow of interesting or suspicious data during the execution of a target application. However, it is also known to suffer from excessively high runtime overhead which limits its overall practicality. The overhead is especially problematic when the taint analysis performed is *generic*. While this type of analysis is more powerful than standard taint status tracking, due to its support of user-defined custom taint policies, its versatility naturally implies that it cannot be optimized for a single task. Along these lines, this dissertation addresses the challenging problem of enhancing the performance of generic taint analysis.

Our research focuses on taint engines implemented using dynamic binary instrumentation. Essentially, the performance bottleneck of these systems stems from complex taint propagation code that is dynamically instrumented, at the level of machine instructions, into the target application. In order to facilitate the implementation of generic taint analysis, instrumentation is typically achieved by inserting transparent calls, which conveniently invoke propagation routines without disrupting the execution behaviour of the analysed application. However, these calls perform expensive context-switching, and therefore degrade overall runtime performance significantly.

We hypothesize that the overhead of generic taint analysis can be reduced in comparison to the slowdowns incurred by the analysis implemented with the use of transparent calls. This hypothesis is tested by investigating three optimizations. Firstly, call-free generic taint propagation is evaluated as a means to mitigate expensive context switching. Secondly, the idea of vectorized generic taint analysis is explored to determine the effectiveness of using SIMD features to propagate multiple taint labels simultaneously. Thirdly, dynamic fast path generation is proposed so that the execution of ineffective taint propagation can be adaptively elided. Our optimizations are presented with respect to a generic taint analysis driven by user-defined primitives, which act as building blocks for custom taint propagation. Crucially, this primitive based approach enables the user to focus on optimizing core pieces of functionality regarding the desired policy, without the need to delve into the complex internals of the system.

The optimizations are integrated into a generic taint tracker called the Taint Rabbit, and their effectiveness is evaluated on various real-world software and CPU-bound benchmarks. With average speed-ups of around 42% on SPEC CPU benchmarks, and as high as 99% on data compression tools, our results demonstrate the advancement of optimized and generic taint analysis.

Optimizing a Primitive Based Approach to Generic Taint Analysis

John Galea

St Cross College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Michaelmas 2020

The research work disclosed in this publication is partially funded by the
Endeavour Scholarship Scheme (Malta). Scholarships are part-financed
by the European Union - European Social Fund (ESF) -
Operational Programme II – Cohesion Policy 2014-2020
"*Investing in human capital to create more opportunities and promote the well-being of society*".

European Union – European Structural and Investment Funds
Operational Programme II – Cohesion Policy 2014-2020
"Investing in human capital to create more opportunities and promote the well-being of society"
Scholarships are part financed by the European Union - European Social Funds (ESF)
Co-financing rate: 80% EU Funds;20% National Funds

Acknowledgements

Undoubtedly, my deepest appreciation goes towards my parents; Anna and John, my brother Francesco, and my sister Ann Marie, for their unconditional love and support during my studies. To put it simply; I would not have managed to complete my DPhil without them.

I would like to extend my sincerest respect and gratitude towards my supervisor; Prof. Daniel Kroening. His academic expertise, exactness, and sound advice have been invaluable throughout the years.

It was a great pleasure to have Sean Heelan, Youcheng Sun and Martin Nyx Brain share office no. 313 with me. Over the years, we engaged in many interesting and intellectual conversations, both at academic and personal levels, which I will miss dearly.

I extend my thanks to Derek Bruening, Hendrik Greving, and the rest of the DynamoRIO team for answering any queries which I had had about the dynamic binary instrumentation engine. During my DPhil, I enjoyed contributing to the open-source project, and I greatly appreciated the maintainers for reviewing many of my GitHub pull-requests. While my DPhil comes to a successful close, I look forward with a keen interest to contributing further to DynamoRIO in the future.

During my first year of studies, I acquired essential knowledge in the field of Cyber Security thanks to the expertise provided by the Centre of Doctoral Training (CDT). I was fortunate to have had kind and remarkable members in my cohort whom ultimately all became great friends of mine. I am also deeply indebted to Prof. Andrew Martin, David Hobbs and Maureen York for giving me the opportunity to pursue my DPhil at the University of Oxford.

To conclude, I would also like to extend my appreciation to the funding bodies who financially aided my research, namely the Endeavour Scholarship Scheme, the Research Institute in Verified Trustworthy Software Systems (VeTSS) and EPSRC.

Abstract

Dynamic taint analysis is a fundamental technique in software security that tracks the flow of interesting or suspicious data during the execution of a target application. However, it is also known to suffer from excessively high runtime overhead which limits its overall practicality. The overhead is especially problematic when the taint analysis performed is *generic*. While this type of analysis is more powerful than standard taint status tracking, due to its support of user-defined custom taint policies, its versatility naturally implies that it cannot be optimized for a single task. Along these lines, this dissertation addresses the challenging problem of enhancing the performance of generic taint analysis.

Our research focuses on taint engines implemented using dynamic binary instrumentation. Essentially, the performance bottleneck of these systems stems from complex taint propagation code that is dynamically instrumented, at the level of machine instructions, into the target application. In order to facilitate the implementation of generic taint analysis, instrumentation is typically achieved by inserting transparent calls, which conveniently invoke propagation routines without disrupting the execution behaviour of the analysed application. However, these calls perform expensive context-switching, and therefore degrade overall runtime performance significantly.

We hypothesize that the overhead of generic taint analysis can be reduced in comparison to the slowdowns incurred by the analysis implemented with the use of transparent calls. This hypothesis is tested by investigating three optimizations. Firstly, call-free generic taint propagation is evaluated as a means to mitigate expensive context switching. Secondly, the idea of vectorized generic taint analysis is explored to determine the effectiveness of using SIMD features to propagate multiple taint labels simultaneously. Thirdly, dynamic fast path generation is proposed so that the execution of ineffective taint propagation can be adaptively elided. Our optimizations are presented with respect to a generic taint analysis driven by user-defined primitives, which act as building blocks for custom taint propagation. Crucially, this primitive based approach enables the user to focus on optimizing core pieces of functionality regarding the desired policy, without the need to delve into the complex internals of the system.

The optimizations are integrated into a generic taint tracker called the Taint Rabbit, and their effectiveness is evaluated on various real-world software and

CPU-bound benchmarks. With average speed-ups of around 42% on SPEC CPU benchmarks, and as high as 99% on data compression tools, our results demonstrate the advancement of optimized and generic taint analysis.

Contents

List of Figures

1 Introduction

We are living in an exciting high-tech era, and software has nowadays become essentially intertwined with our personal lives and that of society in general. It has provided for the worldwide population a phenomenal platform with ample social and economic opportunities. According to Gartner [1], global IT spending amounted to over 3.7 trillion U.S. Dollars in 2019. Yet, software has its downsides as it is infested with swarms of bugs which could bring about severely damaging effects and pose a constant threat to our security and privacy. Indeed, this year, a cyber-attack disrupted civil services in New Orleans, forcing the mayor to declare a state of emergency [2].

It is of paramount importance that we analyse software and fix vulnerabilities, but the sheer overgrowing number of large and complex software make such tasks arduous, let alone feasible, from the standpoint of manual intervention. If we are ever going to keep up and mitigate security threats, then the automation of processes to facilitate the testing, analysis and verification of software plays a crucial role.

Notably, the study of such automated approaches fall under the wide and prominent research area of program analysis. While the topic is vast, one type of analysis, based on the tracking of information flows, has demonstrated to be a pivotal and enabling technique in software security. Its typical applications include malware analysis [3–6], vulnerability discovery [7–12] and runtime attack detection[13–17]. In particular, this technique is none other than *dynamic taint analysis* [13, 17, 18].

The key feature of dynamic taint analysis is the tracking of memory locations and CPU registers that store "interesting" or "suspicious" data. This kind of data is called *tainted*. Initially, data is marked as tainted when it is introduced

by the target application from an interesting source, such as the reading of user input. Taint propagation is then performed so that new locations storing data, influenced by tainted sources as a result of computation, are also marked. Naturally, those locations which no longer store tainted data are *untainted*. In essence, this propagation enables the tracking of taint, and from the perspective of implementation, is achieved by instrumenting the target application's code. Finally, taint is checked at particular points during program execution to enforce rules, or *policies* [18], on how tainted data is used, and to determine whether certain runtime properties hold. For instance, taint analysis has been employed to detect whether the instruction pointer is alarmingly under the control of an attacker [13, 17, 19], or to identify which parts of the user input influence path conditions to optimize fuzzing [7, 11, 20, 21].

Typically, meta-data, referred to as a tag, is associated with tainted data for book-keeping purposes and maintained in supplementary storage known as shadow memory [22, 23]. Most taint analysers, e.g., LibDFT [24], implement single, byte-sized tags that act as flags to denote whether corresponding data is tainted. This setup supports efficient propagation of taint (using simple *bitwise or* operations to combine taint flags) and fast querying of a location's taint status. However, many interesting applications that build upon taint analysis require richer propagation logic, larger taint tags and complex taint labels. For instance, the taint-based fuzzing tool, VUzzer [7], propagates, via union operations, bit-vectors that denote offsets of interesting input bytes which influence difficult path conditions. Meanwhile, the use-after-free debugger, Undangle [10], tracks heap pointers, storing taint information to debug the use of dangling pointers in a composite data structure.

In order to support such use-cases which go beyond bitwise tainting, previous work proposed the extension of single tags and deliver what is called *generic taint analysis* [25, 26]. Generic taint engines track richer labels (say via larger pointer-sized tags) and support user-defined taint propagation. The first notable generic taint engine is Dytan [25].

However, the main problem is that the versatility of generic taint analysis comes at a price: the authors of Dytan report a staggering runtime overhead of ∼30x on `gzip` (a CPU bound application), which has led to the perception that generic taint analysis is, in essence, impractical. We challenge this perception, and aim to deliver generic taint analysis with a runtime overhead that is low enough for practical security applications. In other words, our goal is to optimize generic taint analysis. However, this goal is difficult to achieve; unlike specialized bitwise taint engines, generic trackers cannot have their propagation code tuned for a particular task, but instead must cater for custom user-defined taint policies. Therefore, addressing this performance problem is the focal-point of our work.

1.1 Scope

The scope of our research targets dynamic taint analysis on binary applications using runtime instrumentation. We describe further our research direction as follows:

- **Dynamic Analysis.** We focus on the dynamic variant of taint analysis where tracking is performed while the target application is being executed. Although static information flow analysis [27–29] has also undergone significant study, trade-offs exist between the two approaches. Static analysis is able to consider multiple program paths but suffers from precision and scalability issues. Meanwhile dynamic analysis is limited to a single execution trace, often requiring sophisticated input generation to gain code coverage. However, it is generally more precise and scales better for large real-world applications. Therefore, these benefits led us to focus on dynamic analysis in this work.

- **Binary Analysis.** Our focus is also placed on taint engines capable of analysing stripped x86 binaries without the requirement of source code or re-compilation. This implies that taint analysis delivered by compile-time instrumentation, such as LLVM's `DFSan` [30], are not considered. Despite their complexities, binary taint trackers are of notable benefit to end-users, particularly when targeting legacy or closed-source applications (including malware). Basically, the range of possible target applications is wider when binaries are supported. Moreover, binary analysis also avoids the *What You See Is Not What You eXecute* (WYSINWYX) phenomena [31] which arises due to the absence of low-level behavioural details when analysing source code (before compilation).

- **Dynamic Instrumentation.** In this work, we consider taint analysis built upon dynamic binary instrumentation [32–34], which is the standard technique adopted by the state-of-the-art. During runtime, code of the target application is decoded in the form of basic blocks, instrumented with taint propagation routines, and JITed into an intermediary code cache for execution. While static binary rewriting [35, 36] is an alternative approach where the target application has propagation code inserted not at runtime, it generally faces several challenges which makes its employment difficult in practice. Firstly, software that executes dynamically generated code (e.g., unpackers and JavaScript engines [37]) cannot be instrumented statically as the code is only available at runtime. Secondly, effort must be placed to ensure all libraries used by the target application are also instrumented upfront, otherwise a

typically unwanted partial analysis is performed[1]. Such library boundaries are practically non-existent with dynamic instrumentation [32]. The technique does not require modifying binaries on disk, as instrumentation is done within the application's code stream, on the fly, with the use of a code cache. Finally, dynamic binary instrumentation does not rely on (static) disassembly, which, despite its advances, is known to not always being accurate [38].

1.2 Taint Propagation as a Bottleneck

Fundamentally, one of the primary bottlenecks of taint analysis stems from propagation. Over the years, much work has been dedicated to optimizing taint analysis [24, 39–41], but many do not support extensible propagation logic and thus lack versatility. Meanwhile, existing generic taint [25, 42] engines incur prohibitively high runtime overheads.

Essentially, generic trackers, based on dynamic instrumentation, implement propagation code via the insertion of transparent calls [43], known as *clean calls* [44], prior to the instructions of the target application. Taint propagation code must be transparent in order to avoid inadvertently modifying the behaviour of the application while under analysis. In this respect, clean calls facilitate the development of complex generic taint propagation without concerns of disrupting transparency.

However, despite the benefits, clean calls are expensive in terms of performance. They achieve transparency by comprehensively spilling and restoring all general purpose registers, and swapping between dedicated stacks before invoking taint propagation routines. While bitwise tainting is simple enough to be automatically inlined by a DBI engine [24], the complexity of generic taint analysis makes the avoidance of clean calls difficult. The bottleneck is severe to the extent that the practical use of generic taint analysis is hindered. For instance, Dytan incurs an average overhead of 225.6x over native execution on our benchmarks, relating to data compression and image parsing, that it manages to analyse.

1.3 Research Hypothesis

In this dissertation, we hypothesize that the overhead of generic taint analysis can be reduced in comparison to the slowdowns incurred by the analysis implemented by using transparent calls. This hypothesis is tested by investigating

[1]For instance, such issues were raised for taint analysis leveraged by the Angora fuzzer, which builds upon compile-time instrumentation. https://github.com/AngoraFuzzer/Angora/issues/22

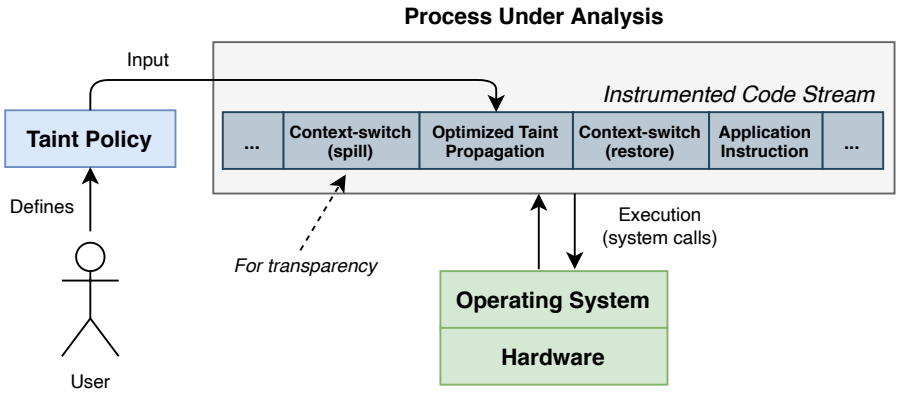


Figure 1.1: High-level approach of the Taint Rabbit based on DBI

three optimizations, which are incorporated within a novel generic taint tracker for x86 binaries called the Taint Rabbit. We evaluate our optimizations on relevant benchmarks and real-world software, as well as compare the Taint Rabbit against numerous state-of-the-art taint engines. Results demonstrate an enhancement of performance; for instance, the Taint Rabbit achieves an average speed-up of around 99% on data compression tools. This suggests that it is feasible for generic taint propagation to be delivered with substantially lesser runtime overhead than that provided by a clean-call-based implementation.

1.4 The Taint Rabbit

We introduce the Taint Rabbit, a framework for building security applications based on fast dynamic taint analysis. As shown in Figure 1.1, the key idea behind the Taint Rabbit's high performance is to speed up taint propagation for dynamic binary instrumentation. According to a taint policy implemented by the user, optimized and transparent propagation code is inserted prior to the application's instructions via DBI. At its core, the Taint Rabbit is based on two main characteristics:

- **The Taint Rabbit is Generic.** Notably, the processing of taint labels during propagation is not fixed by the engine but is user-defined. The Taint Rabbit maps a 32-bit word (or pointer) to every tainted byte in memory and CPU registers which, in turn, enables the storage of a reference to a custom taint label data structure. It propagates these pointers efficiently, and supports user-defined code to update the taint labels during the tracking process.

- **The Taint Rabbit is Optimized.** The Taint Rabbit takes advantage of various optimizations to reduce the overhead of generic taint analysis, ranging from the execution of call-free and vectorized propagation code to the generation of fast paths just-in-time. Ultimately, our work is the first significant effort to bridge the performance gap between specialized and generic taint trackers.

The Taint Rabbit is built upon the powerful DBI engine DynamoRIO [43], and along with various tools built upon it, is available as open-source[2]. In total, the framework consist of over 70,000 lines of C code (including tests).

1.5 Contributions

We propose effective optimizations to speed-up a generic taint analysis that supports user-defined propagation. In particular, our contributions are as follows:

- **A Primitive-based approach to generic taint analysis.** We propose a versatile generic taint analysis for x86 binaries that is based on the idea of *taint primitives*. At a high-level of abstraction, taint primitives are seen as building blocks to taint propagation. For instance, one type of taint primitive defines the way two given taint labels, each associated to a source operand of an instruction, are merged together to produce a resulting taint label that is then mapped to the destination. Taint primitives act as the principle interface between the user and the taint tracker. They are implemented by the user in accordance to the desired taint policy and are provided to the Taint Rabbit in order to perform taint propagation with respect to the semantics of x86 instructions. Crucially, details on the internals of the taint engine are hidden away from the user, who is therefore required to focus only on implementing optimized primitives. To demonstrate generic taint analysis via a taint-primitive-based approach, we consider three applications related to fuzzing, use-after-free debugging, and control-flow hijack detection, where each build upon different taint policies and taint label structure. Our flexible approach is extensively detailed in Chapter 3, while the rest of our contributions relate to its optimization.

[2]https://github.com/Dynamic-Rabbits/Dynamic-Rabbits

- **Optimizing generic taint analysis with call-free propagation.** With clean calls being the main, albeit slow, method for delivering generic taint analysis, we investigate call-free taint propagation implemented using optimized hand-crafted code to bypass their use. While a similar approach has already been adopted for specialized bitwise taint engines [45], generic taint analysis is more complex than simply using *bitwise or* operations, and therefore more challenging to optimize. Our results demonstrate that call-free propagation substantially reduces the runtime overhead. The optimization alone led to average speed-ups of 99% and 98% on data compression and image parsing benchmarks respectively. This optimization is described in Chapter 4.

- **Optimizing generic taint analysis with vectorization.** Vectorized generic taint propagation is also proposed in this work. The main idea is to leverage SIMD instructions so that propagation code processes multiple taint labels, pertaining to multi-byte-sized source operands, in parallel rather than one by one as done by existing scalar implementations. The crux of our approach is based on the observation that the instructions of an application operate on data sizes that are equivalent to the architecture's word. Furthermore, we also note that the pointer-sized taint tags of a word-sized operand conveniently fit within SIMD registers, thus enabling the vectorization of taint propagation. While other works, i.e., Minemu [41], have also adopted vectorization to enhance the performance of taint analysis, the overall approach was only proposed within the scope of taint analysis via bitwise propagation, without any consideration for the more complex generic variant. Our results indicate the feasibility to vectorize popular taint policies, and that the optimization can improve performance when compared to the scalar approach. Specifically, the optimization attains an average speed-up of 23% on data compression benchmarks. Vectorized generic taint analysis is presented in Chapter 5.

- **Optimizing generic taint analysis with dynamic fast path generation.** We contribute a JIT optimization called *dynamic fast path generation* which aims to adaptively elide the execution of unnecessary propagation code to improve performance. In a nutshell, the optimization constructs basic blocks with alternative instrumented versions that have propagation code sliced out according to frequent *in* and *out* taint states identified at runtime. Unlike traditional approaches, which instrument all instructions of the application with taint propagation code by default, the optimization generates fast paths where only those instructions that operate on tainted data are instrumented.

Furthermore, the optimization enables the construction of effective fast paths in comparison to existing trackers, particularly Lift [40], which also adopt the general technique. Specifically, Lift only supports one particular kind of fast path that has all of its instructions uninstrumented. Consequently, this fast path is taken at runtime only when the inputs and outputs of a basic block are all not tainted and propagation is thus completely unnecessary. By contrast, our optimization enables a variety of additional paths, constructed based on runtime information, allowing for further opportunities to avoid the execution of propagation code. Overall, dynamic fast path generation is observed to have synergy with other optimizations, and is effective at reducing overhead, with an average speed-up of 42.8% on SPEC CPU benchmarks. This technique is proposed in Chapter 6.

In addition, Chapter 7 details related work extensively, while Chapter 8 concludes this dissertation with key outcomes and future goals. Background information is also presented in the next chapter.

Finally, the work presented in this dissertation, including verbatim text, is based on the following research papers:

- Galea, John, and Kroening, Daniel. “The Taint Rabbit: Optimizing Generic Taint Analysis with Dynamic Fast Path Generation" ACM AsiaCCS. (2020).
- Galea, John, and Kroening, Daniel. “Vectorized and Generic Taint Analysis" Under Submission. (2020).

2 Background

In order to provide the reader with the necessary background to understand the rest of this dissertation, we start off by describing the overall technique of dynamic taint analysis, as well as the main design factors that impact the capabilities of a taint engine. We also delve into common software vulnerabilities and the ways taint tracking aid their discovery and analysis. Finally, dynamic binary instrumentation is detailed, since it is the underpinning technique used in this work to build the Taint Rabbit. We assume that the reader is already familiar with x86 assembly (a brief guide can be found online [46]).

2.1 Dynamic Taint Analysis

Dynamic taint analysis is a runtime technique that marks and tracks which memory locations are influenced by sources of interest, e.g., user input. The ability to track taint information is a powerful feature; it avoids an analyst from the cumbersome task of inspecting and following potentially long low-level execution traces to determine certain flows of data manually [47]. The analysis also supports the mapping of meta-data, known as *taint labels*, to tainted memory locations, enabling the building of powerful dynamic security tools.

Figure 2.1 illustrates the high-level approach of dynamic taint analysis. The analysis (shown in the `taint view`) is performed during the execution of the target application (shown in the `application view`). In this example, we consider taint analysis for information leak detection [48–50], which prevents confidential information from being exposed publicly. The application starts by reading a

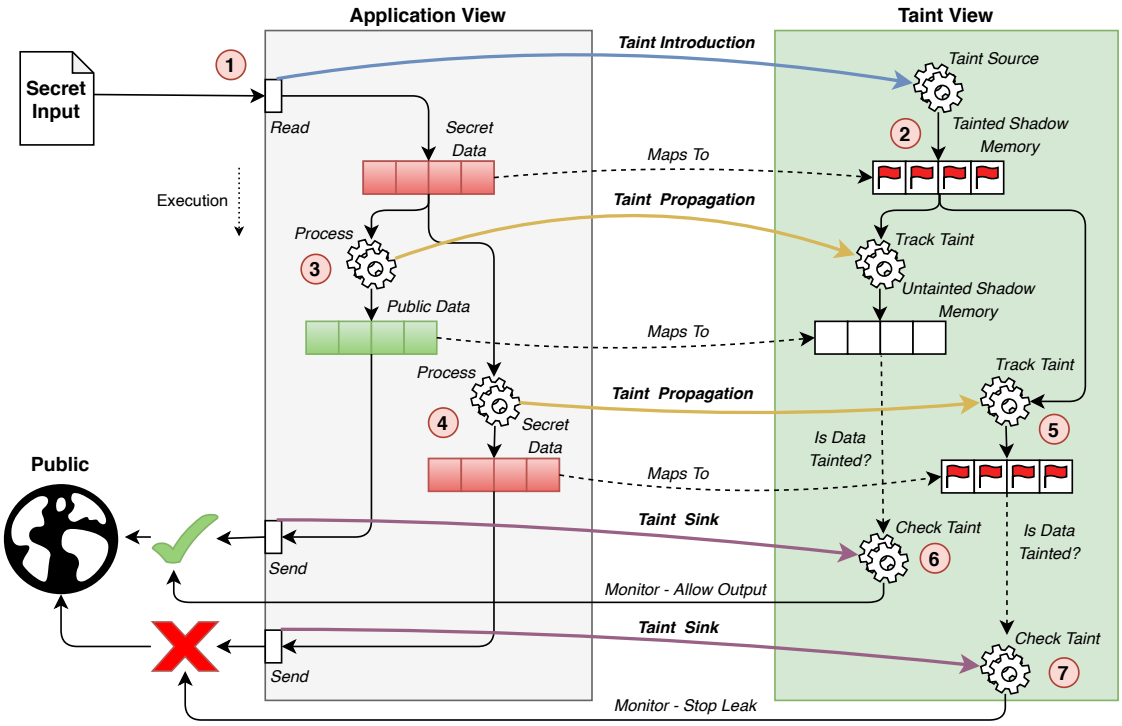


Figure 2.1: High-level approach of Dynamic Taint Analysis.

secret from a local file ①, and stores the content to a buffer in memory. From the view of taint analysis, this action implies that interesting data is introduced into the system. Therefore, the buffer storing the secret is flagged with "tainted" statuses maintained in shadow memory ②.

Next, as seen in the application view, execution proceeds by processing the secret to produce two results stored in separate buffers. One result does not disclose the secret, and therefore is not security-sensitive ③. However, the other result includes the secret and cannot be made public ④. Meanwhile, the taint engine tracks the data-flow of the secret, observing which locations store this sensitive information, as processed by the application. Accordingly, taint is propagated to corresponding shadow memory. In particular, the result buffer consisting of secret data is flagged ⑤.

Finally, the application then attempts to send the results. At this point, the taint engine checks whether the content to be sent is tainted so that the leakage of confidential information is prevented. Consequently, the insensitive data, stored in the untainted result buffer, is outputted ⑥. Meanwhile, the result based on secret data is blocked from being made public ⑦.

2.1.1 Taint Policy

Dynamic taint analysis is performed in accordance to a set of rules collectively referred to as a *taint policy* [18]. Essentially, a policy is implemented by a taint engine and is made up of three components:

- **Taint Introduction.** Taint introduction is a key function that taints initial data stemming from interesting sources. One responsibility of a taint policy is to define such sources which are fundamentally dependent on the considered use-case. For instance, a policy for detecting information leakage dictates that the loading of secret data, say via the `read` system call, is a taint source and the buffer storing the loaded secret must therefore be tainted.

- **Taint Propagation.** Taint trackers propagate taint in accordance to the operations done by the application at instruction-level granularity. A taint policy needs to define the ways taint is processed and propagated to and from shadow memory with respect to these operations, specifying the taint affects of source operands on destinations. While there exists a variety of taint engines implementing different policies, the classical propagation algorithm simply tracks whether memory locations (including CPU registers) are tainted using bit flags (generally referred to as *single* tags) [17]. For instance, the execution of the code statement $A := B$ directs the taint engine to set the current tag of program variable A, denoted as T_A, to the tag of program variable B (i.e., $T_A := T_B$).

  When two or more sources influence a destination, their tags are merged together to produce a combined tag, which is then mapped to the destination. For single tags, *taint merging* is implemented by using simple and fast *bitwise or* operations. This implies that if any source is tainted, then the destination is also tainted as a result. For example, propagation corresponding to the code statement $A := B + C$ is handled by performing $T_A := T_B \mid T_C$. Specifically, since A is affected by both B and C, its tag is assigned to the combined tag of the two sources. In this work, we describe this policy for propagating single tags as *bitwise* due to the principal use of bitwise operations.

  One limitation of single tags is that they are unable to differentiate between taint introduced by different sources (e.g, network traffic, file reading, keystrokes, etc.). In other words, the analysis cannot determine which sources influence a tainted memory location if many sources are considered. To address this issue, other analysis engines take a *multi-tag* approach [25, 26],

leveraging taint labels that consist of more than just one taint flag. Typically, the data-structure used to support a multi-tag is a bit vector, where each bit signifies whether a particular source affects tracked data. Moreover, taint merging is achieved by performing standard union operations on the bit vectors. While multi-tag policies are more extensive than a bitwise policy, the latter is usually sufficient for information leak detection as the analysis only requires querying whether a location is tainted. However, as explained further in Section 2.2, there are use-cases for taint analysis (e.g., taint-based fuzzing) where the support of extensive taint labels is necessary.

- **Taint Sink.** Finally, the last component of a taint policy concerns the querying of taint. In particular, a policy defines taint sinks which are points in a program where the taint of a location is checked to determine or enforce some runtime property. With respect to information leak detection, taint sinks are typically system calls, such as `send`, which output data. These sinks check that the message is not tainted to ensure no sensitive information is made public.

2.1.2 Taint Granularity

Granularity [51] concerns the mapping relationship between application data and taint tags. It denotes the primitive working size of application data that is associated by a taint tag. For instance, a bit-level granularity results in mapping a tag to every bit of application memory, while a byte-level granularity associates a tag to each byte. Generally, the coarser the granularity, the more imprecise is taint tracking (i.e., word-level granularity is less accurate than byte-level granularity, which in turn is less accurate than a bit-level granularity). With byte-level granularity, if a single bit is tainted, then the entire byte is considered tainted. Naturally, this is not the case with a bit-level granularity due to it being finer.

However, the consideration of a coarse granularity results in easier and faster implementation of taint propagation code; the task of modelling complex bit-level semantics of operations performed by the application for taint tracking is avoided. Moreover, a coarse granularity reduces shadow memory costs as less number of tags are maintained per application data.

2.1.3 Taint Approximation

Taint approximation concerns the precision of taint propagation. As detailed in Section 2.1.1, taint is tracked according to the execution of application instructions. This implies that the analysis code responsible for propagating taint for a particular instruction (referred to as an *instruction handler*) needs to follow the semantics of the instruction. However, precise instruction handlers are difficult to implement and lead to expensive taint propagation at runtime [52, 53]. In practice, instruction handlers *approximate* the semantics of instructions.

However, this raises other challenges concerning design choices of taint trackers. In particular, one challenge for a tracker is to decide how to propagate taint to bytes that are indirectly operated on by a handled instruction. While the instruction handler for a `mov` instruction is fairly simple to implement precisely (as there is a direct correlation between source and destination bytes), arithmetic operations may or may not influence destination bytes depending on the actual result of the computation. Figure 2.2 illustrates an example. On the x86 architecture, the `mul` instruction performs a multiplication on the source operand and `eax`. It stores the low-order bits of the result to `eax` and the high-order bits to `edx`. The `edx` register is only used if the entire result does not fit into `eax`. Otherwise, it is set to zero. From the perspective of taint propagation, emulating such instructions to determine precisely whether `edx` is significantly modified based on the values of source operands is often too expensive, and therefore avoided. Consequently, given a tainted source such as `eax`, instruction handlers are either implemented to always taint `edx`, potentially *over-tainting* the register, or skip propagation to the register and open up the possibility of *under-tainting*.

Since over-tainting blindly taints all destination bytes, even in cases where indirect bytes are not influenced by tainted data, it is seen as a conservative approach. In contrast, under-tainting misses out on potentially tainting bytes that actually need to be tainted, but instruction handlers based on this approximation are generally more performant. Ultimately, the accuracy of taint tracking is impacted differently based on the type of approximation chosen, and therefore engines adopt the approach that is best suited for their primary use-case.

2.1.4 Shadow Memory

Shadow memory [22, 23, 41] is a fundamental component of taint trackers as it stores the tags of corresponding application memory. With respect to the granularity of the tracker, the scale of shadow memory determines the size of a tag. Table

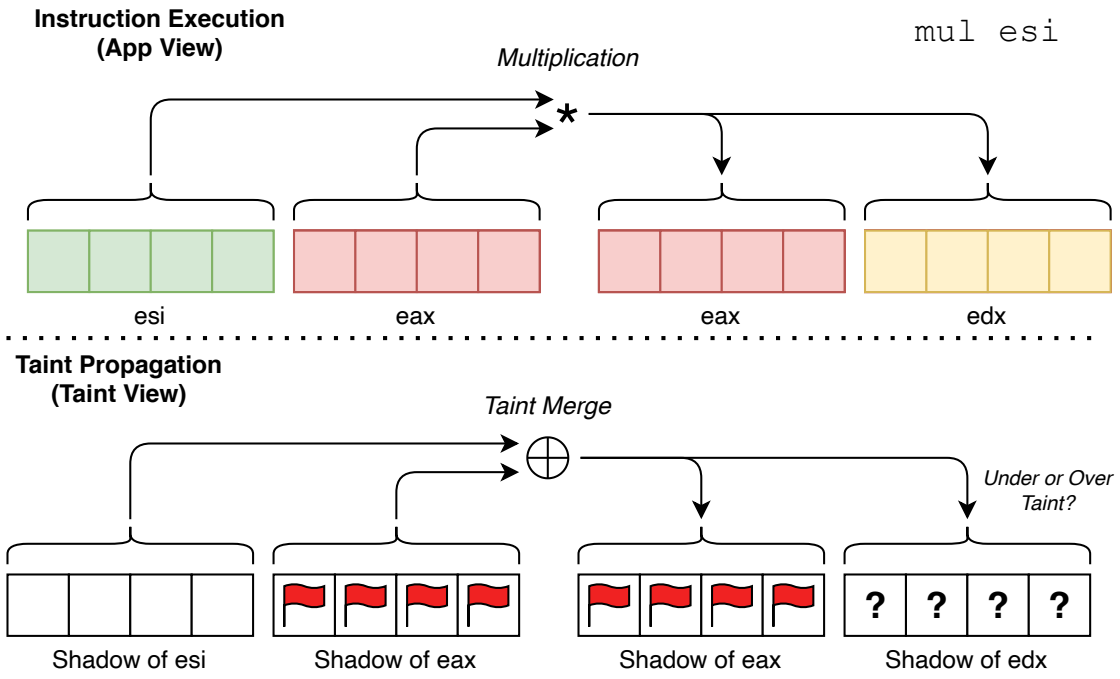


Figure 2.2: Approximating taint propagation for the `mul` instruction

Table 2.1: Various shadow memory scales supported by Umbra [22].

Umbra Setting	Scale	Granularity	Tag Size	Description
SCALE_DOWN_8X	Down	byte	1-bit	Eight application bytes to 1 shadow byte.
SCALE_DOWN_4X	Down	byte	2-bit	Four application bytes to 1 shadow byte.
SCALE_DOWN_2X	Down	byte	4-bit	Two application bytes to 1 shadow byte.
SCALE_SAME_1X	Same	byte	1-byte	One application bytes to 1 shadow byte.
SCALE_UP_2X	Up	byte	2-byte	One application bytes to 2 shadow byte.

2.1 details some of the scales supported by Umbra [22], a robust shadow memory library used by the Dr. Memory Debugger [45]. Assuming byte-level granularity, shadow memory may be scaled down, e.g., to map one shadow bit per application byte, or scaled up so that tags are made large. In this work, we build upon Umbra for shadow memory management.

Mapping Schemes

In order to access and modify taint information, a translation mechanism is required to obtain the shadow address from its corresponding application memory address [54]. The method of achieving this translation depends on the mapping scheme implemented for organizing shadow memory. Generally, *direct* and *segmented* mappings are the two primary schemes adopted:

Direct Mapping Scheme. Under a direct mapping scheme, shadow memory is set up as one whole region and placed at a fixed offset from application memory. Typically, access is achieved via a simple memory reference operand of type: `base + displacement`. Specifically, the base register holds the application address being translated, taking into account the scale of shadow memory, and the displacement is the offset between the two memory regions. Due to the simplicity of the scheme, its primary benefit is performance. However, it also requires a strict memory layout to establish the offset.

Segmented Mapping Scheme. A segment mapping tackles the problem of layout inflexibility for shadow memory through indirection. Rather than maintaining one whole area of shadow memory, this scheme allows the area to be split into smaller blocks located at different base addresses. When translating the application address, the scheme fetches the base address of the appropriate block, which contains the corresponding taint information, via a lookup table. Unfortunately, a segmented mapping scheme is usually more costly in terms of performance than direct mapping, due to these lookup operations.

However, a segment mapping elegantly enables an optimization similar to create-on-write in order to reduce space overhead imposed by shadow memory. Basically, rather than expensively creating all shadow memory segments upfront, the optimization creates segments on demand upon first writes of taint information. Indeed, popular debuggers such as Dr. Memory and Valgrind [55] adopt this technique.

Figure 2.3 illustrates the high-level details of the optimization. Initially, the scheme requires that only a single special block is created. The block has its content set to no-taint statuses, and all the entries of the translation lookup table are set to refer to it. The special block is shared to handle read accesses to shadow memory prior to any writes, thus reducing space overhead.

The mechanism used to detect first writes is via signal faults. The special block has its access permission set to read-only and therefore any attempted writes leads to a fault ①. This triggers a fault handler which dynamically creates an official shadow memory block to perform the write and registers the new entry in the translation lookup table. The handler also sets the base register of the write operation with the new translated address so that the write operation is retried. This time, the taint information is written to actual shadow memory and not the special block ②.

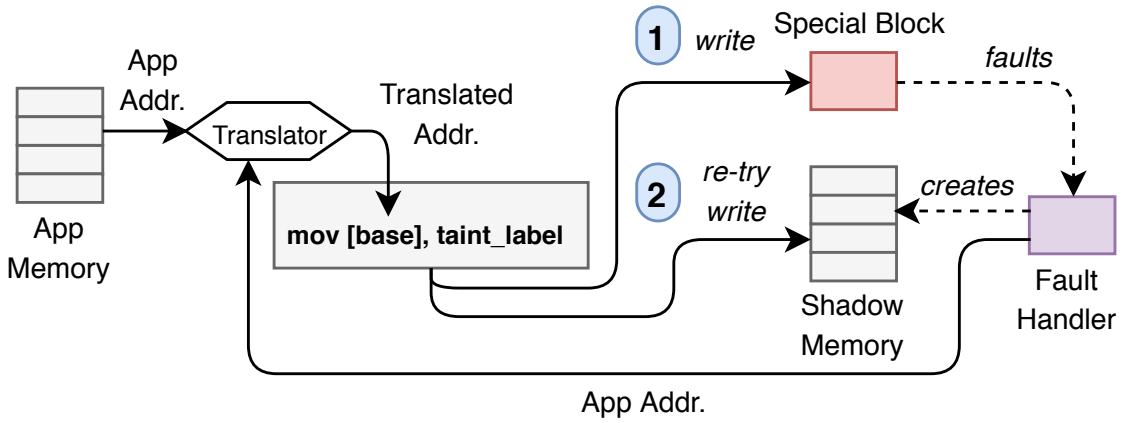


Figure 2.3: Creating shadow memory upon writes

2.2 Vulnerability Analysis

2.2.1 Software Vulnerabilities

Many software vulnerabilities [56] stem from memory-corruption-bugs introduced by developers when using unsafe programming languages such as C and C++. Their prevalence are of deep concern for security as attackers exploit them to cause mischief and harm. Vulnerability analysis is an arduous task and we need automated tools to facilitate this process [57]. Fortunately, dynamic taint analysis is demonstrated to be an effective technique [7–12, 20]. In this section, we explain various types of vulnerabilities and delve into the methods of using taint analysis to enable their discovery and exploit detection.

Buffer Overflow

A buffer overflow occurs when data is stored in a memory buffer of insufficient allocation size, resulting in adjacent content being corrupted due to out-of-bound writes. Often, the root cause originates from the lack of boundary checks on the length of user input prior to calling unsafe functions, such as LibC's `memcpy()` or `strcpy()`.

As an elementary example, Figure 2.4 illustrates a simplistic exploitation of a stack buffer overflow. The goal of an attacker is to smash the stack so that control-sensitive information, such as a saved return address, is overwritten with a malicious address that points to shell-code [58]. A string of 'A' characters, labelled as junk, is used to reach the saved instruction pointer, which is corrupted with the address `0x01AAF23A` set by the attacker. Typically, the malicious address refers to an appropriate trampoline instruction. Assuming countermeasures (e.g., ASLR and

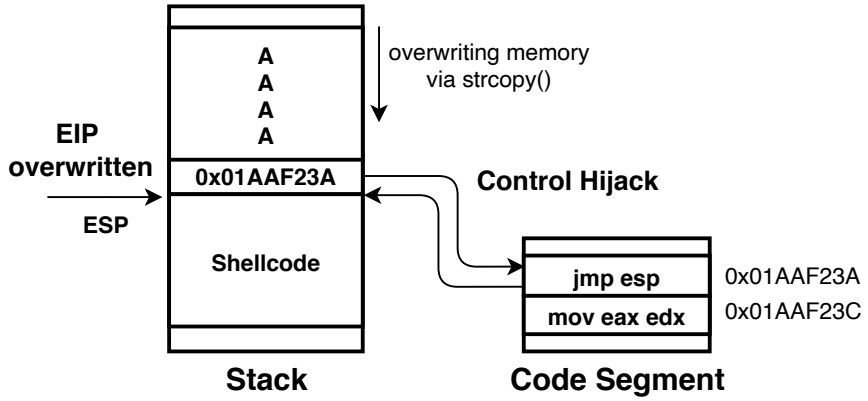


Figure 2.4: Exploiting a stack-buffer overflow, corrupting the saved return address

DEP) are disabled, this instruction, located in a non-randomized module of the application, is responsible for directing control to the shellcode reliably. In this example, `jmp esp` is considered as a suitable trampoline instruction since the stack pointer `esp` refers to the beginning of the shellcode. Ultimately, once shellcode is executed, the attacker hijacks control-flow.

Apart from buffer overflows stemming from the stack, their heap variants are also exploitable through the corruption of control data [59]. For instance, if a function pointer is stored somewhere alongside a heap buffer, it may be maliciously overwritten by exploiting an overflow. Consequently, when the application eventually performs a call via the function pointer, control-flow is directed to the attacker's code, rather than executing the legitimate function.

Taint analysis is able to detect such attacks by checking whether control data is tainted and therefore deemed as untrustworthy[13]. A taint policy for detecting control-hijacking attacks, such as those that originate from buffer overflows, involves: (1) introducing taint upon the reading of user input, (2) propagating taint to memory and registers when they store data derived from user input, and untainting them when they do not, and (3) checking that control data (e.g. a saved instruction pointer) is not corrupted with tainted data.

Buffer Over-read

Similar to a buffer overflow, an over-read vulnerability is also caused by an out-of-bounds memory error. However, while an overflow writes to memory beyond the buffer, an over-read continues to access further existing data (that is not intended to be read) [60]. As shown in Figure 2.5, the danger is that this data may be sensitive and not intended for access. Consequently, over-read vulnerabilities enable data-oriented attacks leading to the leakage of information, such as a process address

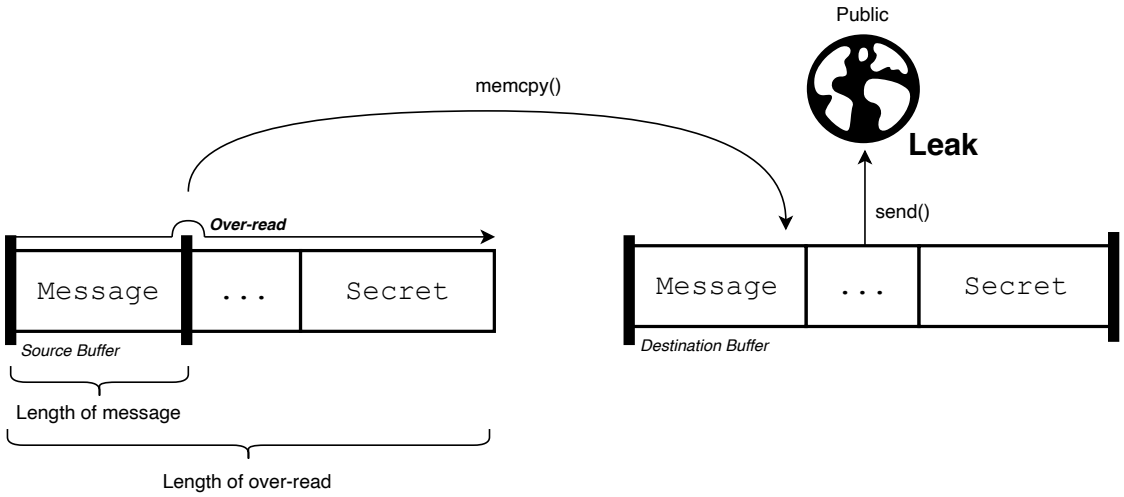


Figure 2.5: Leakage of secret data due to buffer over-read

useful for defeating ASLR. In fact, the infamous Heartbleed bug [61], which was found in the OpenSSL cryptographic library, was caused by such a vulnerability that ultimately allowed for the stealing of encryption-key-material. Since these exploits do not involve control-flow hijacking, current OS security defences, particularly CFI, are unable to prevent them. To detect information leak vulnerabilities before the release of software, taint analysis can be adopted to track the flow of secret data and ensure that the application never outputs such data, say via the network, e.g., `send()` or to the console, e.g., `printf()`. We considered this use-case as a means to explain taint analysis in the beginning of this chapter in Section 2.1.

Use-After-Free

Use-after-free (UAF) [62–65] is another vulnerability class often associated with the heap. These software bugs originate from incorrect dereferences of dangling pointers referring to freed memory. Unfortunately, UAF vulnerabilities are exploitable. By contriving a dangling pointer to refer to malicious data, the exploit is able to hijack control upon the pointer's erroneous use. For instance, as illustrated in Figure 2.6, an attacker typically achieves exploitation by having the dangling pointer refer to what appears to be a C++ object but with a corrupted virtual table pointer. In turn, this pointer refers to a fake virtual table that directs to shellcode. Indeed, the main challenge of the attacker is to make the dangling pointer refer to the payload. One technique to solve this problem is via heap spraying, where multiple heap objects containing the payload are allocated to increase the chances of the allocator reusing the freed memory referred to by the dangling pointer.

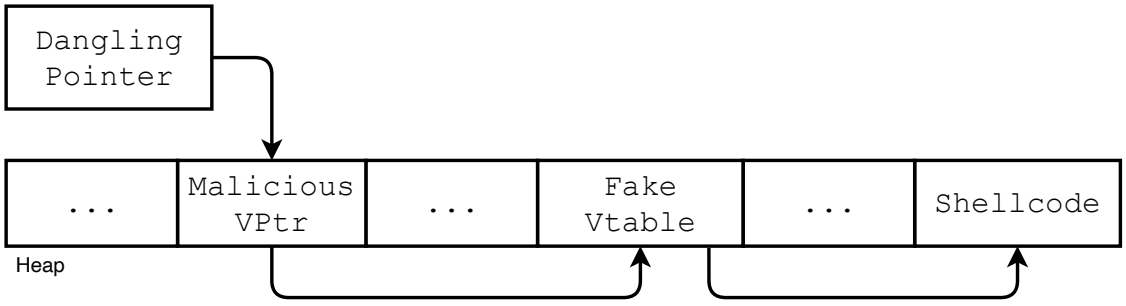


Figure 2.6: Exploiting UAF vulnerabilities, where the dangling pointer refers to a malicious vtable

In contrast to spatial errors, e.g., buffer overflows, use-after-free is particularly arduous to debug due to its temporal characteristics [66]. Specifically, the allocation and deallocation of heap memory, along with the erroneous use of a dangling pointer, occur at different stages in execution time.

To address this challenge, previous work [10, 67] adopted taint analysis to track pointers, where dangling statuses, acting as taint labels, aid vulnerability analysis. Specifically, taint is introduced to mark memory pointers, returned by heap allocation functions (e.g., `malloc`), as *LIVE* via their taint labels. Upon the execution of deallocation functions (e.g., `free()`), pointers still referring to freed memory have their taint labels updated to mark them as *DANGLING*. The policy considers memory dereferences as taint sinks, where UAF vulnerabilities are detected by checking whether the used pointer is marked as *DANGLING* according to the pointer's taint label.

2.2.2 Vulnerability Discovery

Several techniques have been proposed to facilitate the discovery of security vulnerabilities. Some also leverage taint analysis to improve their effectiveness. We now delve into two research topics relating to bug finding, namely fuzzing [68] and symbolic execution [69].

Fuzzing

As illustrated in Figure 2.7, fuzzing aims to crash the application and discover vulnerabilities by bombarding the applications with multiple runs driven by random input. Through the consideration of an initial set of test cases, referred to as seeds, a mutational fuzzer starts by generating new inputs via the modification of their content using several strategies, such as bit flips and arithmetic operations [70]. These inputs are then used to test the application, where feed-back, usually based

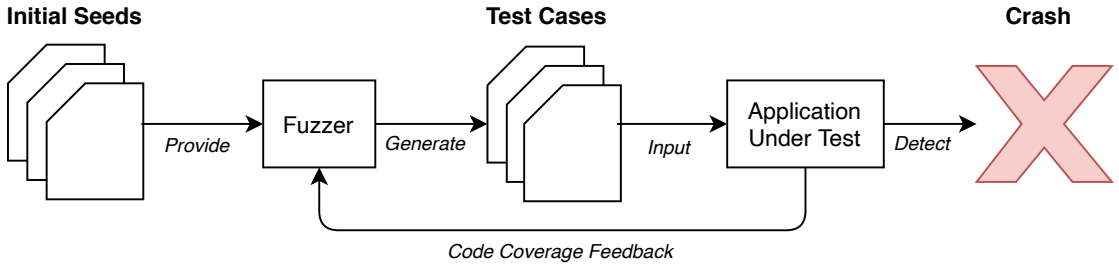


Figure 2.7: Evolutionary mutation-based fuzzing

on code coverage, is provided back to the fuzzer to incrementally test more code. Specifically, test cases which drive the application to execute new code blocks are placed in the seed pool so that they are mutated again in the next fuzzing iterations.

Taint-based-fuzzers, such as Angora [11] and VUzzer [7], leverage input-data-tracking to identify specific input bytes that influence difficult path conditions, blocking the exploration of new code. Naturally, taint introduction occurs upon the reading of input files, and each byte is associated with a multi-tag indicating its offset. In other words, the use of multi-tags, built upon bit vectors, enables each bit to represent whether a specific input byte is affecting the data of a tracked memory location. For instance, if a location's taint stems only from the first two bytes of user input, the bit vector, acting as its taint label, is in the form $< 1, 1, 0, 0, ... >$, where only the first two corresponding bits are set. With such taint information, a taint-based fuzzer checks the operands of `cmp` instructions so that it is able to scope mutation and randomize those particular influential input bytes during test case generation. This increases the chances of satisfying path conditions, enhancing code-coverage and ultimately finding bugs.

Symbolic Execution

Symbolic execution is a program analysis technique that reasons over program paths expressed in the form of logical formulae. Rather than running an application concretely, a symbolic execution engine emulates paths, treating values of program variables as unknown (i.e., symbolic). Notably, the use of symbolic variables, each acting as a wild-card to represent any possible value, enables the analysis of multiple concrete executions at the same time. Moreover, by using a constraint solver, a path formula may be conditioned to verify certain properties or generate inputs for normal execution. Over the years, a number of symbolic execution engines have been proposed which among others include KLEE [71], Triton [72], S2E [73], BitBlaze [74] and SAGE [75].

Listing 2.1: A C function that counts the number of even integers in an array.

```
1  int count_even(int *input)
2  {
3      int count = 0;
4
5      /* Iterate over integers. */
6      for (int i = 0; i < 2; i++){
7          /* If even, increment counter. */
8          if (input[i] % 2 == 0)
9              count++;
10     }
11
12     return count;
13 }
```

Process As detailed further in an existing survey [69], a symbolic execution engine analyses paths by maintaining so called *symbolic states*. In particular, a state consists of a symbolic store σ and path constraints π. σ maps program variables to expressions of symbolic variables α and concrete values. Meanwhile, π is a formula denoting the conditions stemming from branches on symbolic variables.

Listing 2.1 presents a simple function that returns the count of even numbers inside an array of two integers. Basically, it iterates over the array and checks whether the integer is even via the mod (%) operator. The function's symbolic execution tree, which shows the states at each execution point, is illustrated in Figure 2.8. We consider the two integers in the array as symbolic, and therefore σ is initialized accordingly with two program-variable-mappings ①. Moreover, since no path constraints are set, π is $TRUE$. As the program's code statements are analysed step by step, the symbolic state of a path is updated. For instance, when `counter` is initialized ②, the variable is mapped to the concrete value of zero in σ.

Furthermore, whenever a state branches according to symbolic data, the engine performs a fork producing two states which are then inserted in its state pool. One of the states represents the path that takes the $TRUE$ case of the branch, while the other represents the path that considers the $FALSE$ case. The symbolic variable conditioned by the branch statement is constrained for each state, depending on the path taken. In our example, a fork is performed when checking if an integer is even ③. In particular, one state has its path constraints π modified to $\alpha_a \% 2 \neq 0$ to denote that $input[0]$ is not even, and the other is conditioned to represent the alternative.

State Explosion Unfortunately, one major limitation of symbolic execution concerns scalability, imposed by the state explosion problem [69], where the number of states to explore is infeasibly large. Suppose the size of the array passed to the `count_even()` function is arbitrary, denoted as n, instead of two. This implies that

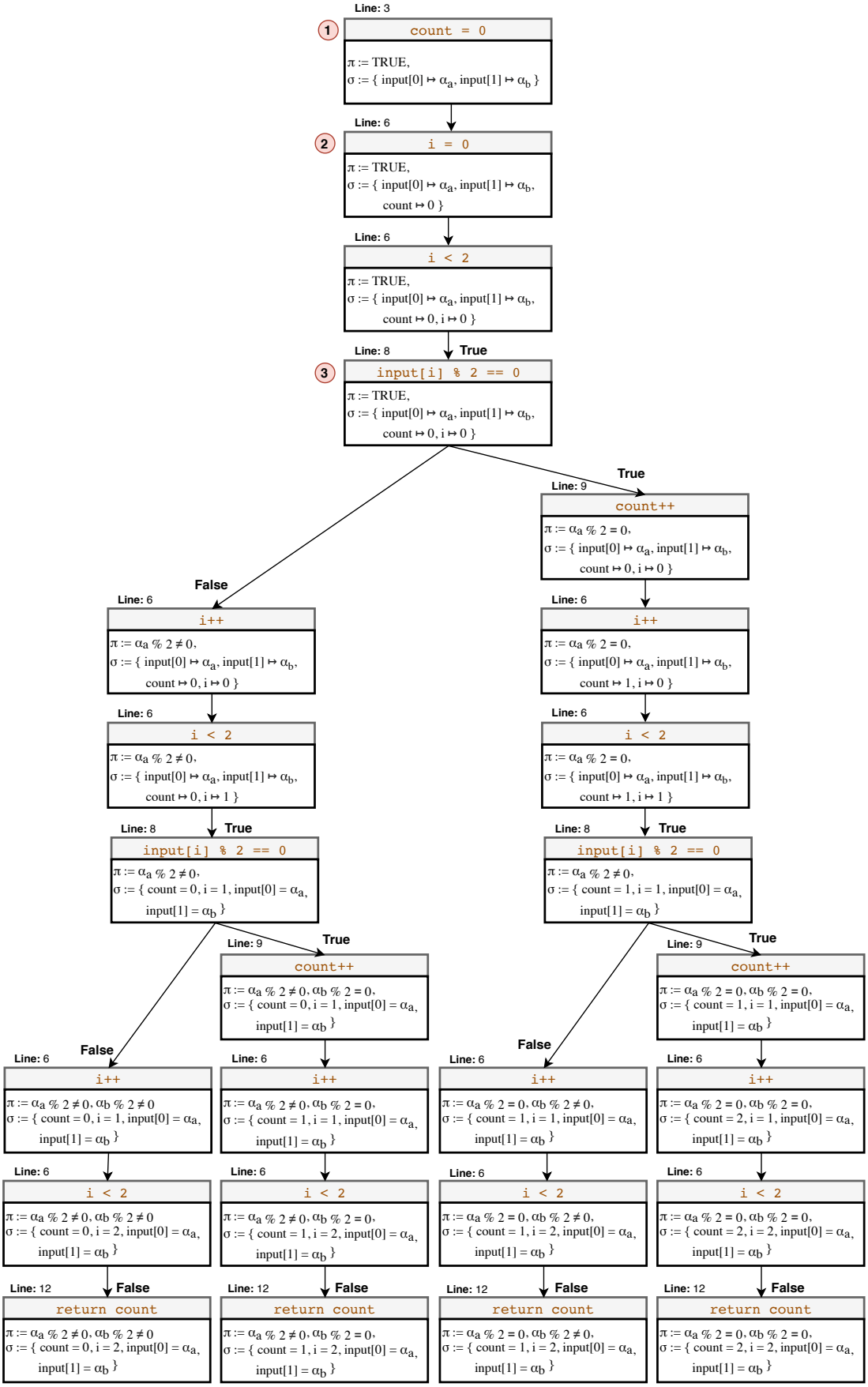


Figure 2.8: The symbolic execution tree of `count_even()`. Adapted from [69]

the number of states produced from forking is 2^n, resulting in an explosion due to exponential growth. Consequently, this arises the challenge of managing many states simultaneously, which leads to significant memory overhead. A variant of symbolic execution, known as concolic execution [76–78], scales analysis by considering a single path at a time. Specifically, concolic execution runs the application concretely under some test input while maintaining the corresponding symbolic state during runtime. To explore other branches of the program after execution, path conditions are negated and a constraint solver is used to generate additional test cases.

Augmentation with Taint Analysis. An integral component of symbolic execution is the chaining of symbolic data for building expressions. If program variable `A` is considered symbolic and assignment $B := A + 7$ is performed, then `B` is also symbolic.

Dynamic taint analysis serves as a suitable platform for such tracking of symbolic data required by concolic execution [47, 72]. The feature of taint labels enables the mapping of memory locations with symbol expressions. As illustrated in Figure 2.9, one way to achieve this is by labelling symbolic expressions as numerical IDs [47]. Whenever at least one source is tainted, the taint propagation policy generates new IDs and maps them to the destination bytes to signify that the bytes are associated to fresh variables. This helps build the use-def dependency chain abstractly in the form of static single assignment (SSA) [79][1], and lays some of the ground work to slice and represent the execution path for constraint solving.

2.3 Dynamic Binary Instrumentation

Similar to previous research [24, 25, 39, 40], we focus on taint analysis implemented based on dynamic binary instrumentation (DBI) [33, 43]. With DBI, taint trackers instrument the application under analysis with propagation code at runtime.

2.3.1 Architecture

Figure 2.10 illustrates the high-level architecture of a DBI system. Although it is primarily based on DynamoRIO [32], it shares many similarities with the designs of other DBI systems, such as Pin. In particular, the code cache is an integral component which ultimately provides the flexibility for dynamic analysis tools to instrument and manipulate code of the application.

[1]Of course, generating an SSA representation also requires defining the operations performed on the variables, but the key point being conveyed in this section is that taint analysis is a powerful technique which can be used to facilitate concolic execution.

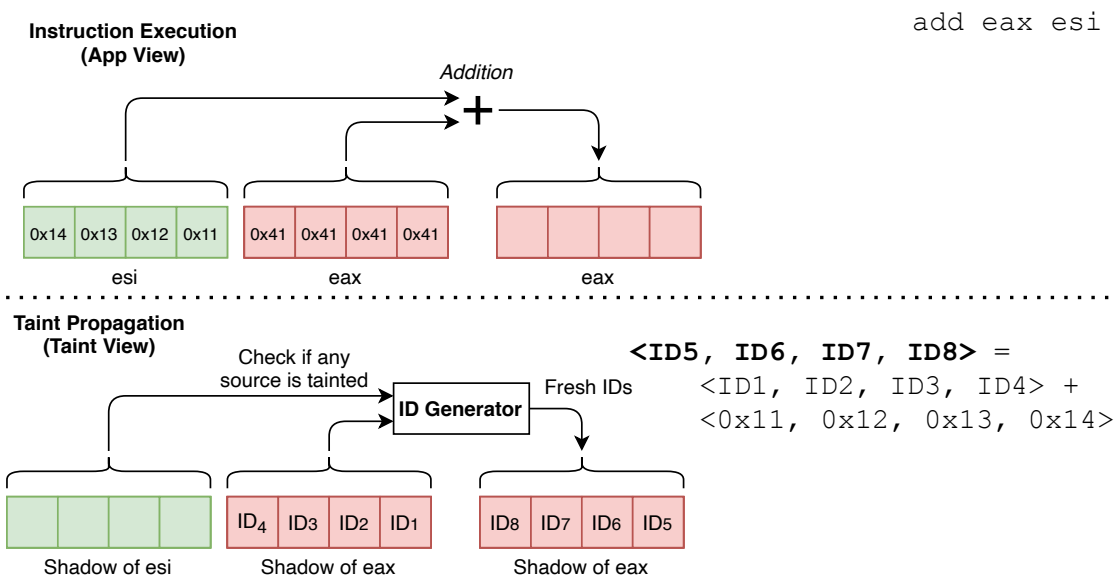


Figure 2.9: ID Propagation - Whenever taint is propagated from a source, the corresponding destination byte is mapped to a fresh ID.

Crucially, a DBI system has a complete view of the application's process and is particularly useful for gathering runtime information. With access to the machine state of the application, concrete values of registers and memory are easily obtained. A DBI system injects itself into the process (for instance, by using a private custom loader or the `LD_PRELOAD [80]` environment variable on Linux), and thus is within the same address space. Once injected, the DBI system directs control to the dispatcher to start the execution of the application's code ①. This is achieved by copying and decoding code ② in the form of basic blocks, where each block is a sequence of non-control-transferring instructions, with exception of the last instruction. Basic blocks are provided to the analysis tool for instrumentation ③, and then passed back to the DBI system ④ to be jitted into the code cache ⑤.

Essentially, execution takes place within the code cache ⑦. Therefore, the dispatcher directs control, entailing a context-switch, to the target basic block inside the cache ⑥. An interpreter-based approach, where the dispatcher acts as an intermediary between the execution of each basic block, is primarily avoided as this leads to intensive context-switching. Instead, basic blocks ending with direct branches are linked to their next target blocks so that control does not return back to the dispatcher but is maintained within the cache. For indirect branches, where the target is usually known only at runtime (i.e., is a non-static dependency), the DBI system relies on fast inlined hash look-ups [81] to translate the address and locate the next appropriate block for execution. Naturally, apart from the instructions of the applications, execution of a basic block also considers

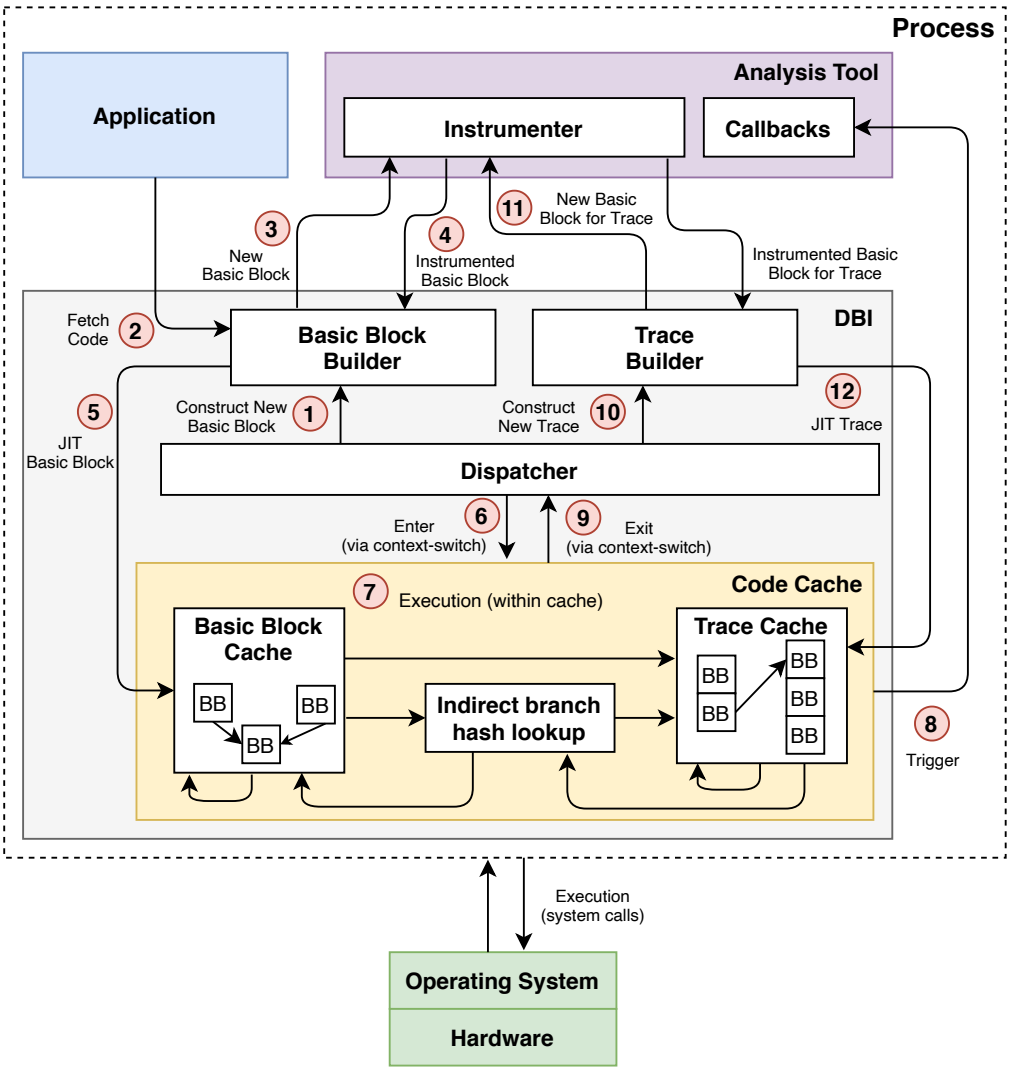


Figure 2.10: High-level approach of Dynamic Binary Instrumentation. Adapted from [32].

the instructions inserted by the analysis tool. This inserted code typically results in triggering callback functions, registered by the tool, in order to perform the actual analysis (e.g., taint propagation) ⑧.

If a basic block is not yet present in the code cache, control returns back to the dispatcher for its construction ⑨. Transitioning out of the cache also incurs overhead due to a context switch. However, the cost is usually limited as invoking the dispatcher is mainly done for code cache management. Furthermore, once the new basic block is jitted, existing blocks that directly target its code have their links updated *proactively* [32].

As an optimization, DBI engines also work with other units of code fragments apart from basic blocks. Specifically, they generate *traces* ⑩, which are sequences

Listing 2.2: Clean call code, with context-switches, generated by a DBI engine.

```
1  64 a3 00 00 00 00     mov    %eax -> %fs:0x00[4byte]
2  64 a1 10 00 00 00     mov    %fs:0x10[4byte] -> %eax
3  89 60 0c              mov    %esp -> 0x0c(%eax)[4byte]
4  8b a0 a8 02 00 00     mov    0x000002a8(%eax)[4byte] -> %esp
5  64 a1 00 00 00 00     mov    %fs:0x00[4byte] -> %eax
6  68 00 00 00 00        push   $0x00000000 %esp -> %esp 0xfffffffc(%esp)[4byte]
7  9c                    pushf  %esp -> %esp 0xfffffffc(%esp)[4byte]
8  60                    pusha  %esp %eax %ebx %ecx %edx %ebp %esi %edi -> %esp 0
       ↪ xffffffe0(%esp)[32byte]
9  68 02 00 00 00        push   $0x00000002 %esp -> %esp 0xfffffffc(%esp)[4byte]
10 e8 4f 38 4e 6c        call   $0xb7fd43e0 %esp -> %esp 0xfffffffc(%esp)[4byte]
11 8d 64 24 04           lea    0x04(%esp) -> %esp
12 61                    popa   %esp (%esp)[32byte] -> %esp %eax %ebx %ecx %edx %ebp %
       ↪ esi %edi
13 9d                    popf   %esp (%esp)[4byte] -> %esp
14 8d a4 24 5c 02 00 00  lea    0x0000025c(%esp) -> %esp
15 64 a3 00 00 00 00     mov    %eax -> %fs:0x00[4byte]
16 64 a1 10 00 00 00     mov    %fs:0x10[4byte] -> %eax
17 8b 60 0c              mov    0x0c(%eax)[4byte] -> %esp
18 64 a1 00 00 00 00     mov    %fs:0x00[4byte] -> %eax
```

of frequently executed basic blocks. The principle idea is to monitor for common execution patterns and attach basic blocks together, in order to elide branches that are normally required for transitioning between these blocks.

From the perspective of the analysis tool, traces are instrumented in a similar fashion as to basic blocks. The DBI system provides the basic blocks that form a trace to the tool so that trace-specific instrumentation can be performed (11). Finally, traces are then jitted to a separate code cache (12) where control can effectively flow between the execution of basic blocks and traces.

2.3.2 Transparent Calls

Since resources, particularly CPU registers, are shared by both taint propagation code and the application when employing DBI, analysis needs to be transparent so that it does not inadvertently affect the runtime behaviour of the application. To simplify tool development and alleviate the concern of transparency from the user, DBI frameworks allow the insertion of *clean calls* [44] to invoke analysis routines. Before a call, a context switch that comprehensively spills registers [82] is performed, essentially saving the machine state of the application. Once the analysis routine returns, another context-switch then restores the state.

Specifically, Listing 2.2 shows the code generated by DynamoRIO to perform a clean call. First, lines 1-5 perform a stack swap, switching from the stack used by the application to a dedicated, thread-local stack for the analysis routine. To achieve this swap, `eax` is spilled so that it is used as a scratch register and set to store the base address of the DBI context, (i.e., book-keeping data required for DBI), of the

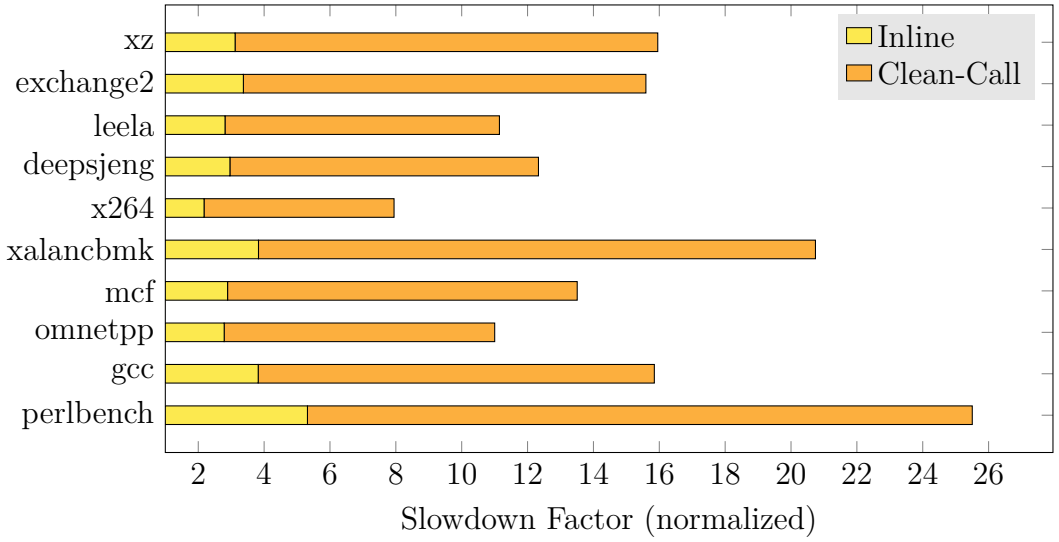


Figure 2.11: Performance of clean call and inline instruction execution counters.

executing thread. Next, the actual swap is performed and the value of `esp`, which stores the pointer to the application's stack, is saved and then set to point to the analysis stack. Lastly, `eax` is then restored back to its original app value.

At line 7, the code saves the value of the flags register by pushing it onto the stack with the `pushf` instruction. The values of all general purpose registers are then also spilled via `pusha`. As a result, the machine state of the application is saved, and preparation of the call follows. In particular, a parameter value is pushed onto the stack at line 9, and finally the analysis function is invoked at line 10.

Once the analysis function has been triggered and control returns, the parameter is first skipped by modifying `esp` via an `lea` instruction at line 11. The code proceeds by restoring the scratch and flag register (lines 12-13), and then performs another stack swap to set `esp` point again to the application's stack (lines 15-18). As a result, the machine state is restored, thus completing the clean call.

Performance and Optimization

Although clean calls bring about easy development of runtime tools, their expensive context switching incurs high overheads. Taint analysis requires instrumenting many instructions of the application to track tainted data, and therefore the intensive use of clean calls is problematic in terms of performance.

To quantify the overhead of clean calls, we ran the DynamoRIO tool `inscount`[2] which inserts such calls at the start of basic blocks to count the number of instructions executed by an application. As illustrated in Figure 2.11, we observe an average

[2]https://github.com/DynamoRIO/dynamorio/blob/master/api/samples/inscount.cpp

Listing 2.3: Code of an inlined clean call

```
1  64 89 1d 00 00 00 00 mov    %ebx -> %fs:0x00[4byte]
2  64 8b 1d 10 00 00 00 mov    %fs:0x10[4byte] -> %ebx
3  89 83 90 02 00 00    mov    %eax -> 0x00000290(%ebx)[4byte]
4  89 8b 8c 02 00 00    mov    %ecx -> 0x0000028c(%ebx)[4byte]
5  89 93 88 02 00 00    mov    %edx -> 0x00000288(%ebx)[4byte]
6  9f                   lahf    -> %ah
7  0f 90 c0             seto    -> %al
8  89 83 94 02 00 00    mov    %eax -> 0x00000294(%ebx)[4byte]
9  b8 02 00 00 00       mov    $0x00000002 -> %eax
10 89 83 98 02 00 00    mov    %eax -> 0x00000298(%ebx)[4byte]
11 31 d2                xor    %edx %edx -> %edx
12 b9 e8 43 fd b7       mov    $0xb7fd43e8 -> %ecx
13 81 c1 3c 3b 00 00    add    $0x00003b3c %ecx -> %ecx
14 8b 83 98 02 00 00    mov    0x00000298(%ebx)[4byte] -> %eax
15 01 81 24 01 00 00    add    %eax 0x00000124(%ecx)[4byte] -> 0x00000124(%ecx)[4byte]
16 11 91 28 01 00 00    adc    %edx 0x00000128(%ecx)[4byte] -> 0x00000128(%ecx)[4byte]
17 8b 83 94 02 00 00    mov    0x00000294(%ebx)[4byte] -> %eax
18 04 7f                add    $0x7f %al -> %al
19 9e                   sahf   %ah
20 8b 93 88 02 00 00    mov    0x00000288(%ebx)[4byte] -> %edx
21 8b 8b 8c 02 00 00    mov    0x0000028c(%ebx)[4byte] -> %ecx
22 8b 83 90 02 00 00    mov    0x00000290(%ebx)[4byte] -> %eax
23 64 8b 1d 00 00 00 00 mov    %fs:0x00[4byte] -> %ebx
```

slowdown of ∼15x on the SPEC CPU 2017 benchmarks. Consequently, DBI frameworks attempt to avoid clean calls and automatically inline analysis code with the application's instructions. The context switches only spill and restore a few registers, and the out-of-line execution that is naturally implied by the use of function calls is removed.

Listing 2.3 presents code of an inlined clean call. First, `ebx` is saved to addressable thread-local storage so that it is used as a scratch register. In particular, at line 2, the register is set to the base pointer of spill slots. This enables the saving of three other general purpose registers, namely `eax`, `ecx` and `edx`, between lines 3 and 5. The `eax` register is then used as an intermediary to store the current flags by using the `lahf` and `seto` x86 instructions (lines 6-7). The flag statuses are then saved in a slot at line 8. Next, lines 9-16 denote the inlined analysis code where, by definition, has call instructions elided. Finally, lines 17 to 23 are responsible for restoring the application's machine state, setting the scratch registers and the flags back to their original values.

Inlined instrumentation is cheaper than full clean calls because context-switching is not overly comprehensive. We ran `inscount` again but with the inline optimization enabled[3] and the optimization overall reduces the overhead down to ∼3.3x. Unfortunately, analysis routines are not always inlined by DBI frameworks but only if they are *simple*, meaning that they are small, avoid control-flow and perform no function calls themselves (i.e., they are leaf functions) [44, 83].

[3]Turning on/off the inline optimization is achieved via DynamoRIO's `-opt_cleancall` option.

Listing 2.4: Code responsible for deciding whether to inline a clean call. Adapted (shortened) from DynamoRIO 8.0.0's source code: core/arch/clean_call_opt_shared.c

```
1  static void
2  analyze_callee_inline(dcontext_t *dcontext, callee_info_t *ci)
3  {
4      bool opt_inline = true;
5
6      /* Callee cannot be inlined - num of instrs. */
7      if (ci->num_instrs > MAX_NUM_INLINE_INSTRS)
8          opt_inline = false;
9
10     /* Callee cannot be inlined - has control flow. */
11     if (ci->bwd_tgt != NULL || ci->fwd_tgt != NULL)
12         opt_inline = false;
13
14     /* Callee cannot be inlined - uses SIMD. */
15     if (ci->num_simd_used != 0)
16         opt_inline = false;
17
18     /* Callee cannot be inlined - uses mask register. */
19     if (ci->num_opmask_used != 0)
20         opt_inline = false;
21
22     /* Callee cannot be inlined - accesses TLS. */
23     if (ci->tls_used)
24         opt_inline = false;
25
26     /* Callee can be inlined. */
27     if (opt_inline)
28         ci->opt_inline = true;
29 }
30
31 static bool
32 analyze_clean_call_inline(dcontext_t *dcontext, clean_call_info_t *cci)
33 {
34     bool opt_inline = true;
35
36     /* Fail inlining clean call - number of args > 1 */
37     if (cci->num_args > 1)
38         opt_inline = false;
39
40     /* Fail inlining clean call - used slots > available slots. */
41     if (info->slots_used > CLEANCALL_NUM_INLINE_SLOTS)
42         opt_inline = false;
43
44     return opt_inline;
45 }
```

For instance, Listing 2.4 shows part of the actual code of DynamoRIO that determines whether to inline a clean call. At instrumentation-time, DynamoRIO decodes the callback for analysis, invoking `analyze_callee_inline()`, which returns a *true* value to activate inlining. The function mainly consists of conditional statements that check whether the callback: (1) does not have a high instruction count that exceeds a predefined threshold (line 7), (2) does not have any control-flow instructions (line 11), (3) does not use any SIMD registers (line 15), including k-mask registers introduced for AVX-512 (line 18), and (4) does not use thread-local-storage (line 23). DynamoRIO also inspects the caller; it does not perform inlining if more

than one parameter is passed (line 37), or if there are not enough scratch slots available to save all registers used by the callback function (line 41).

To improve the chances of inlining code, it is recommended that tools are compiled with optimization flags set (-O2). However, as discussed in Chapter 3, the complexity of rich taint propagation leads to inlining failure even when the taint engine is compiled with optimization flags.

2.4 Summary

This chapter provides the preliminary background information necessary to understand the main content of the dissertation. After introducing the integral aspects of dynamic taint analysis, we explain how the technique is useful for inspecting memory corruption vulnerabilities. Moreover, popular bug-finding approaches, such as fuzzing and symbolic execution, are also described. Finally, we delve into dynamic binary instrumentation, focusing particularly on how clean calls are performed to transparently execute analysis code. The clean calls' lack of performance and their inlining as an optimization are also highlighted.

3

A Primitive-Based Approach to Generic Taint Analysis

As detailed in Chapter 2, dynamic taint analysis is performed with respect to a policy that defines the ways taint is introduced, propagated and checked at runtime. Specialized taint engines are built to support a specific propagation policy that is often based on bitwise tainting [17]. The consideration of a specific policy brings about numerous advantages to deliver fast taint analysis. In particular, its allows the taint engine to adopt specialized optimizations and have its implementation tuned for the task at hand [24]. However, this also results in a trade-off in terms of versatility, as specialized taint engines are naturally unable to support other taint propagation policies off-the-shelf. To suit their needs, users are required to undergo the task of modifying the complex internals of the taint engine, or in the worst case scenario, build a new taint engine from scratch.

By contrast, generic taint engines [25, 26] aim to support user-defined taint policies, and therefore are more versatile than specialized trackers. In this work, we present the Taint Rabbit, a generic taint engine for x86 binaries. Two key design features positively impact the versatility of the Taint Rabbit. Firstly, a 32-bit word (or generic pointer) is mapped to every tainted byte, enabling the storage of a reference to a custom taint label data structure. Secondly, the Taint Rabbit supports custom code, provided by the user, to update the taint labels during propagation according to a desired taint policy. This is achieved with a programming interface, where the user supplies this custom propagation code in the form of *taint primitives*. At a high level, a taint primitive is seen as a building block, implementing a core piece of functionality (e.g., taint merging) for taint propagation. In turn, the Taint

Rabbit harnesses these primitives with respect to the semantics of x86 instructions. Crucially, by adopting such a primitive-based approach, the Taint Rabbit frees its users from the need to delve into its internal components and deal with the large and complex x86 instruction set in order to implement taint propagation. Therefore, from the user's perceptive, primary focus is placed on developing the required policy.

Through empirical evidence, we demonstrate that the delivery of generic taint analysis via a taint-primitive-based approach is feasible. Specifically, we consider three diverse policies and show their successful employment using the Taint Rabbit.

3.1 Applications of Taint Analysis

There are numerous use cases for taint analysis. However, specialized bitwise taint engines that use *bitwise or* operations to merge taint are unsuitable to employ them all; they fail to support richer and more complex policies. To emphasize this point, we consider three policies, each of which rely on different taint propagation logic and label structures.

Control-Flow Hijacking Detection. Previous work [13] has shown that taint analysis can detect control-flow hijacking attacks. Since the analysis only has to check whether control data is tainted, *bitwise or* operations suffice for merging taint status flags.

UAF Detection. Use-after-free (UAF) bugs are exploitable [63, 84]. Undangle [10] debugs such vulnerabilities by tracking heap pointers via taint analysis. Undangle monitors allocations and deallocations and assigns the pointer status stored in taint labels to *LIVE* and *DANGLING* accordingly. 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 (i.e., the destination is not a pointer), even though both of 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 *NOT-POINTER*, *LIVE*, and *DANGLING*, and their merging cannot be appropriately done with the operation. Moreover, apart from the status of the pointer, Undangle's taint labels also contain debugging data, e.g., PCs of the creation and dangling of root pointers, and thus are of composite type. Instead of using a *bitwise or*, Algorithm 1 gives an implementation for propagating such labels via conditional statements.

Algorithm 1: Pointer tracking propagation [10]

Data: Taint Label: s_1, s_2
Result: Taint label: d

```
// If both sources are not pointers (i.e., not tainted), then destination is also not a pointer.
if s1 = NULL and s2 = NULL then
    d ← NULL;
// If both sources are pointers, then destination is not a pointer.
else if s1 ≠ NULL and s2 ≠ NULL then
    d ← NULL;
// If first source is the only pointer, then propagate its label.
else if s1 ≠ NULL then
    d ← s1;
else
    // Second source is the only pointer; propagate its label.
    d ← s2;
end
return d;
```

Fuzzing. VUzzer [7] 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 as large as, if not more than, several kilobytes. VUzzer therefore uses a bit vector, which implies that a single *bitwise or* instruction is not suitable for taint merging, and the implementation of 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 [39–41]. Our approach is a *generic* taint analysis and supports all of the mentioned use cases.

3.2 Generic Taint Analysis

Generic taint analysis enables the employment of user-defined taint policies that go beyond the use of status flags that are conventionally offered by specialized bitwise taint engines. Its support of custom processing and merging of taint labels during propagation removes the need for the user to change the internals of a taint engine for a particular use case.

The first notable generic taint engine proposed in literature is Dytan [25]. The engine offers multi-tags, implemented by using bit vectors, as taint labels which

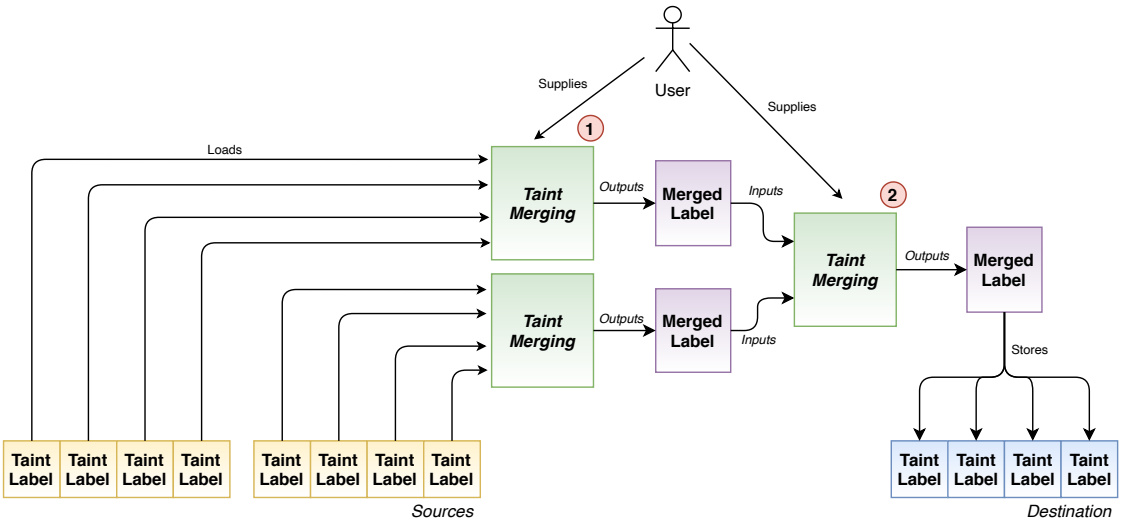


Figure 3.1: Generic taint propagation performed by Dytan

are associated to each byte of application memory[1]. It also enables propagation, particularly functionality related to taint merging, to be easily customized based on the desired policy. While the default approach is to merge bit vector labels via union operations, users can override Dytan's *merge* function to implement their own process for obtaining the resultant label from a set of source bit vectors.

Figure 3.1 illustrates taint propagation performed by Dytan when analysing an instruction that has one destination and two source `dword` sized operands. User-defined taint merging is leveraged in two ways to carry out the propagation. Firstly, for each of the two source operands, the labels of their bytes are merged together, resulting in the production of two intermediary combined labels that overall represent their taint information ①. Secondly, taint merging is performed again to combine these two intermediary labels and derive the final label that is associated to all destination bytes ②. Ultimately, through custom taint merging, the user is able to exert control over the processing of taint labels during taint propagation.

However, Dytan has some limitations in terms of design which impacts its versatility[2]. Firstly, Dytan is hard-coded to use bit vectors as taint labels, but policies, such as that of pointer tracking for UAF detection, require different data-structures, often of composite type. Therefore, a better solution is to support pointer-sized tags in order to allow references to user-defined taint labels. Secondly, although a single taint merging function simplifies policy implementation, its use to

[1]Dytan adopts byte granularity only for memory. Every register is actually mapped to one taint label, and not for each of their bytes.

[2]In practice, one major limitation of generic taint analysis is substantial runtime overhead, but we will address this issue in subsequent chapters. For now, we primarily focus on the design of the analysis.

derive both a combined label of a multi-byte-sized source operand and a label to associate with destination bytes limits the flexibility of custom propagation. Unlike those based on the union of bit vectors, certain policies might require different merging functionality for these two steps, but such finer control for the user is not supported by existing approaches.

For instance, recall Algorithm 1, which describes taint merging for pointer tracking. Specifically, this algorithm is intended only for merging labels stemming from two separate source operands (i.e., the merge indicated by ② in Figure 3.1). It is not suitable for merging labels of application bytes pertaining to the same operand, as otherwise the resultant intermediary label may incorrectly represent the *NOT-POINTER* state after combining the labels of an actual pointer's bytes.

While the first limitation concerning the support of large 32-bit tags is fairly easy to address, so much so, that such tags are already offered by other existing engines (e.g., DataTracker [7]), the second limitation is more challenging from a design's perspective. Ultimately, maximal flexibility for customizing propagation requires the user to delve deeper into the internals of the taint engine, and in the worst case implement propagation handlers specific to x86 instructions. However, this complicates the implementation of taint policies. Instead, this work investigates whether a primitive-based approach enables generic taint analysis while also offering a suitable programming interface for users to implement custom propagation at an appropriate level of abstraction.

3.3 The Taint Rabbit

We now detail the generic taint analysis delivered by the Taint Rabbit. The high-level architecture of the tracker is illustrated in Figure 3.2. To ease wide deployment, our analysis targets x86 binary applications and therefore does not require source code. Dynamic Binary Instrumentation (DBI) is the underpinning of the Taint Rabbit, where propagation code is inserted into the basic blocks of the target application's code at runtime.

From the standpoint of the user, effort is dedicated to building a dynamic analysis tool based on a taint policy. The tool uses the APIs of both the DBI engine and the Taint Rabbit to implement taint introduction and sinks via callback functions that are triggered upon runtime events relevant to the chosen policy. Upon taint introduction ①, the Taint Rabbit is informed by the tool, via its API, which locations are tainted, along with their starting labels. Meanwhile, taint checks, performed at sinks, query the Taint Rabbit to retrieve the label associated to a

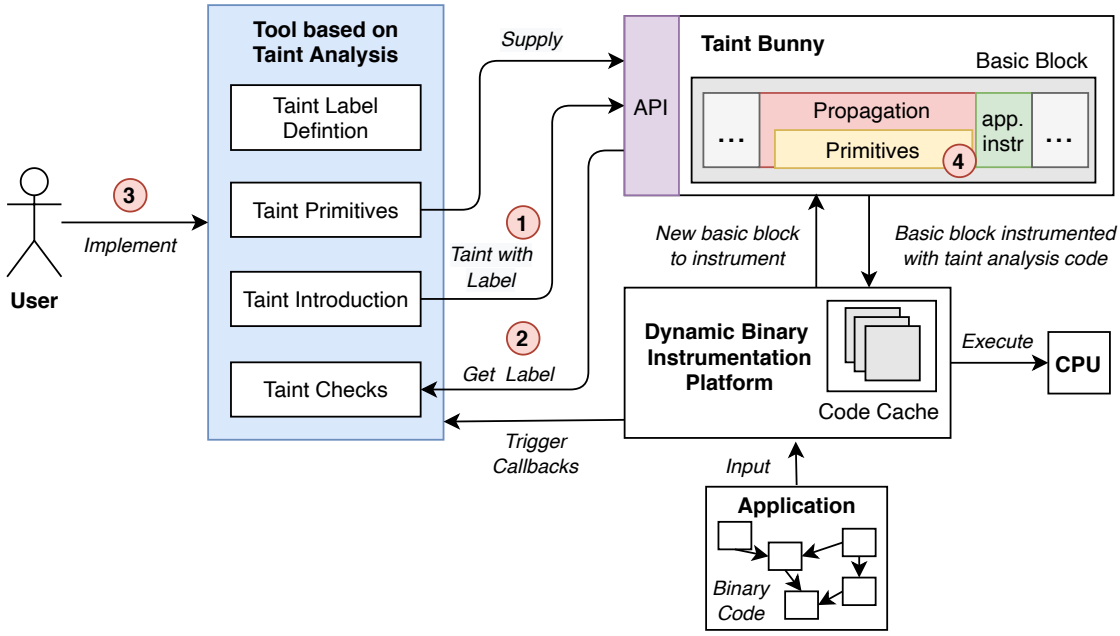


Figure 3.2: High-level architecture of the Taint Rabbit that employs propagation policies driven by user-defined primitives

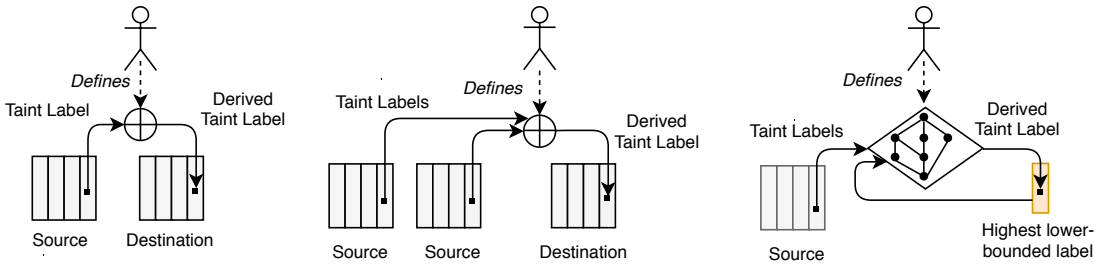
tainted memory location ②. Notably, the Taint Rabbit associates a pointer-sized tag to each tainted byte, enabling the reference of a taint label that is structured to satisfy the needs of the user.

In order to implement customized taint propagation, the user supplies code, referred to as taint primitives, to the Taint Rabbit ③. Essentially, a *taint primitive* is a core piece of functionality, acting as a building block, for taint propagation. It is responsible for processing and deriving a taint label from a set of source taint labels. During propagation, the Taint Rabbit fetches the labels of the source operands from shadow memory, applies the appropriate primitives with respect to the semantics of x86 instructions, and associates the resulting labels to the bytes of destination operands ④.

Ultimately, with the support of user-defined taint propagation and label structure, the Taint Rabbit aims to be a flexible framework for building sophisticated taint-based tools. We aim to allow users to focus on solely implementing and optimizing their taint policies, separating other concerns such as shadow memory management and the modelling of x86 instructions.

3.3.1 Design Features

The main design features of the Taint Rabbit are detailed as follows:



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

Figure 3.3: User-defined primitives invoked by the Taint Rabbit to propagate taint

- **Binary Analysis.** Our analysis is scoped to consider x86 binaries. The code that performs propagation considers the semantics of the target application's instructions. Naturally, we refer to such code responsible for handling the propagation of an instruction simply as an *instruction handler*. By taking into account how instructions operate, handlers avoid tainting output locations unnecessarily, e.g., tainting the stack pointer.

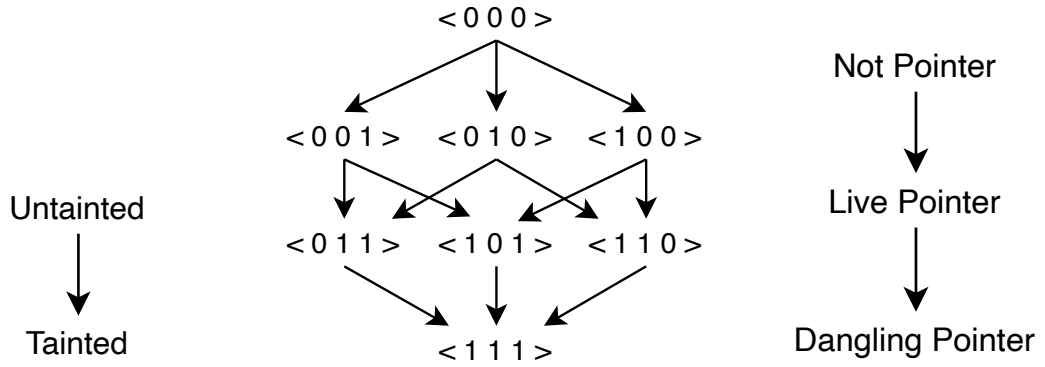
- **Generic Taint Label Structure.** The unit of meta-data that the Taint Rabbit uses as a label is a 32-bit word. The word may itself store tags or act as a generic pointer to a larger taint label data structure. A `NULL` value represents "no taint".

- **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 scaled-up shadow memory [22] where one application byte corresponds to four shadow bytes. We do not label the x86 flag register or track implicit flows to avoid *taint explosion* [85].

- **Over-Approximation.** The Taint Rabbit takes a conservative and over-approximate approach to taint propagation. In particular, all destination bytes which *could* be influenced by a tainted source are always marked as tainted by our instruction handlers.

3.3.2 Taint Primitives

As illustrated in Figure 3.3, taint labels are propagated via user-defined code called *taint primitives*. 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



(a) Status Flags [86] **(b)** Bit Vectors **(c)** Pointer Tracking [10]

Figure 3.4: Examples of taint lattices

with a destination byte from one and two source labels respectively. For example, Algorithm 1 is a $src, src \rightarrow dst$ primitive. Meanwhile, inspired by previous work [86], the third primitive computes the greatest lower bound, referred to as a meet label, of two given labels in a *taint lattice*. For instance, Figure 3.4 illustrates the taint lattices for the use cases detailed in Section 3.1. The meet label of two bit vectors $< 0, 1, 0 >$ and $< 1, 0, 1 >$ is $< 1, 1, 1 >$. Moreover, in the case of pointer tracking, two labels both representing $LIVE$ statuses result in their meet label to also indicate $LIVE$ in accordance to the lattice. Specifically, this meet operation is detailed by Algorithm 2

Algorithm 2: Meet operation for pointer tracking

Data: Taint label: s_1, s_2
Result: Taint label: m

```
// If both sources are not pointers (i.e., NULL), then meet label is NULL.
if s1 = NULL and s2 = NULL then
    m ← NULL;
// If any source is a dangling pointer, then meet label is a dangling status.
if s1 = DANGLING or s2 = DANGLING then
    m ← DANGLING;
else
    // A source must be a live pointer; meet is a live status.
    m ← LIVE;
end
return m;
```

Essentially, a taint primitive can be seen as a function that takes a set of taint labels and returns a label as a result. For example, a $src, src \rightarrow dst$ primitive is a mapping $f : T \times T \rightarrow T$, where T is the set of all possible taint labels.

While all of the taint primitives are user-defined, we place one restriction that prevents them from introducing taint. If all of the source labels passed to

a primitive indicate *no taint* (i.e., they are all `NULL` values), then the primitive must return `NULL` as a result. In other words, $f(NULL, NULL) = NULL$. The high-level intuition here is that primitives are intended to be used to propagate taint and not act as taint sources.

Overall, the taint primitives are the interface between the user-defined taint propagation 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`.

Meet vs. Merge. In Section 3.2, we explained that Dytan's support of a single merging function hinders flexibility of taint analysis. By contrast, the Taint Rabbit uses different taint primitives to address this limitation. Therefore, the $src, src \rightarrow dst$ and $src, src \rightarrow_M meet$ primitives serve very different purposes despite the fact that they have the same signature:

- The primitive $src, src \rightarrow dst$ is used to combine taint labels stemming from two different sources.
- The primitive $src, src \rightarrow_M meet$ is used to combine taint labels stemming from 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 to emphasize this point. Given two labels that represent a *LIVE* and *DANGLING* status, respectively, their *meet* is *DANGLING*, while the combination of two pointers is *NOT-POINTER*.

Another important difference is that $src, src \rightarrow_M meet$ cannot return a meet label based on any side-effects, but always in accordance to a taint lattice. This is in contrast to the other primitives, namely $src \rightarrow dst$ and $src, src \rightarrow dst$, which may be stateful (e.g. these primitives may use global variables to derive taint labels if required by the user).

3.3.3 Instruction Handlers

An instruction handler is responsible for propagating taint upon the execution of an application's instruction. It can be seen as a harness for taint primitives, which are invoked according to the semantics of the instruction being analysed. In this Section, we detail how the Taint Rabbit handles instructions with one destination and two source operands (e.g., `add`), as well as instructions with one destination and one source operand (e.g., `mov`). While handlers for other types of instructions differ, the algorithms presented highlight the crux of the Taint Rabbit's use of taint primitives.

Algorithm 3: Computing the taint label for a two-operand instruction

Data: ID: dst, src_1, src_2, Taint Label: $label$, $meet\_label_1$, $meet\_label_2$, Integer: $opnd\_size$

```
   // Get meet label of first source operand.
1  meet_label_1 ← NULL;
2  for i ← 0 to opnd_size − 1 do
3  |   label ← lookup_label(src_1 + i);
4  |   meet_label_1 ← meet_primitive(meet_label_1, label)
5  end
   // Get meet label of second source operand.
6  meet_label_2 ← NULL;
7  for i ← 0 to opnd_size − 1 do
8  |   label ← lookup_label(src_2 + i);
9  |   meet_label_2 ← meet_primitive(meet_label_2, label)
10 end
   // For every destination byte, derive a label by merging the meet labels.
11 for i ← 0 to opnd_size − 1 do
12 |   label ← src_src_dst_primitive(meet_label_1, meet_label_2);
13 |   set_label(dst + i, label);
14 end
```

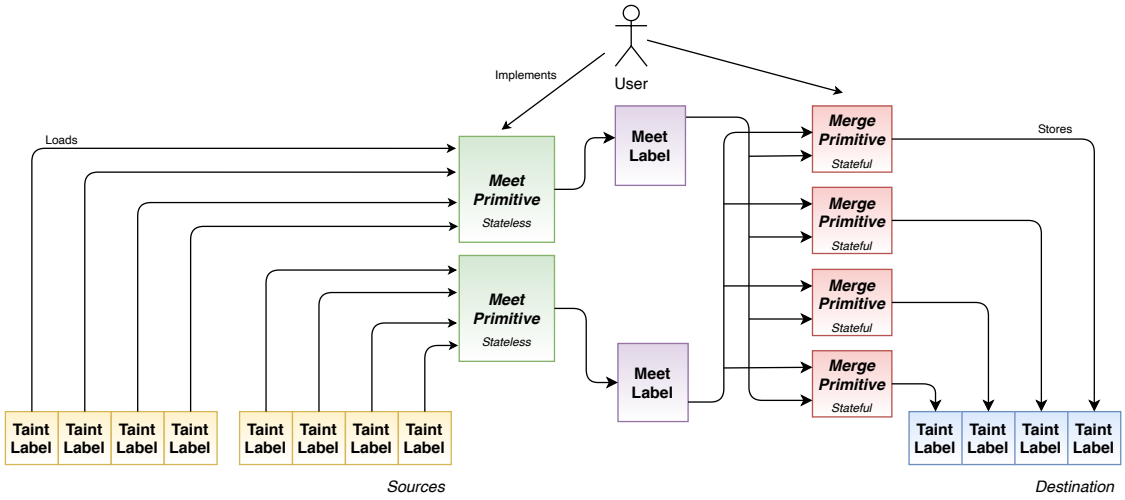


Figure 3.5: Generic taint propagation performed by the Taint Rabbit

One destination and two source operands. Algorithm 3 specifies how the Taint Rabbit uses the $src, src \rightarrow_M meet$ and $src, src \rightarrow dst$ primitives to compute the taint label for an instruction with two source operands. The propagation is also illustrated in Figure 3.5. The algorithm first iterates over the bytes of each operand separately and applies the *meet* primitive on these bytes. This yields one *meet* label for each of the two operands, denoted by $meet\_label_1$ and $meet\_label_2$, respectively.

At line 11, the algorithm then iterates over the bytes of the destination operand,

Algorithm 4: Optimized tainting for instructions with independent bytes

Data: ID: dst, src_1, src_2, Integer: $opnd\_size$

```
// For every destination byte, derive a label by merging the corresponding source labels.
```

1 **for** $i \leftarrow 0$ *to* $opnd\_size - 1$ **do**
2 $\quad src\_label_1 \leftarrow$ lookup_label($src_1 + i$);
3 $\quad src\_label_2 \leftarrow$ lookup_label($src_2 + i$);
4 $\quad dst\_label \leftarrow$ src_src_dst$_{primitive}$(src_label_1, src_label_2);
5 $\quad$ set_label($dst + i$, dst_label);
6 **end**

and uses the $src, src \rightarrow dst$ primitive to combine the meet labels of the two source operands. The combined label is then assigned to the destination byte. We remark that the primitive may be stateful, and hence, its invocation is not hoisted out of the loop.

For many x86 instructions, resulting bytes are independent. Instances of this are most transfer instructions (e.g. `mov`) 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. In this case, the two loops that meet the taint labels can be dropped. Algorithm 4 gives the resulting specialized instruction handler. Algorithm 4 is both faster than Algorithm 3 and produces a result that is more precise when handling such instructions. Some trackers, such as LibDFT, use the approach taken in Algorithm 4 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.

One destination and one source operand. Algorithms 5 and 6, illustrate the usage of the $src \rightarrow dst$ primitive. They are similar to Algorithms 3 and 4, but only accept one source operand. We implement them to propagate taint for instructions such as `mov`, `inc` and `bswap`.

Crucially, we enable users to customize propagation also when one source is considered and taint merging is not required. The Taint Rabbit does not take any conventional propagation short-cuts, e.g., those posed by *silent stores* [87]. For instance, bitwise taint engines do not instrument arithmetic/logic instructions that have an immediate source operand, e.g., `add eax, 5` as it does not affect the taint status of the destination. However, a generic taint engine requires instrumenting such instructions because user-defined primitives, which can be based on side-effects, may still derive new taint labels. In other words, a generic taint engine cannot be optimized for a particular taint policy in contrast to a specialized taint engine.

Algorithm 5: Taint propagation for one source operand

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

// Get meet label of source operand.

$meet\_label_1 \leftarrow NULL$;
for $i \leftarrow 0$ *to* $opnd\_size - 1$ **do**
  $label \leftarrow$ lookup_label($src_1 + i$);
  $meet\_label_1 \leftarrow \text{meet}_{primitive}(meet\_label_1, label)$
end
// For every destination byte, derive a label from the meet label.
for $i \leftarrow 0$ *to* $opnd\_size - 1$ **do**
  $dst\_label \leftarrow \text{src\_dst}_{primitive}(meet\_label_1)$;
  set_label($dst + i$, dst_label);
end

Algorithm 6: Optimized taint propagation for one source operand with independent bytes

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

// For every destination byte, derive a label from the corresponding source label.

for $i \leftarrow 0$ *to* $opnd\_size - 1$ **do**
  $src\_label_1 \leftarrow$ lookup_label($src_1 + i$);
  $dst\_label \leftarrow \text{src\_dst}_{primitive}(src\_label_1)$;
  set_label($dst + i$, dst_label);
end

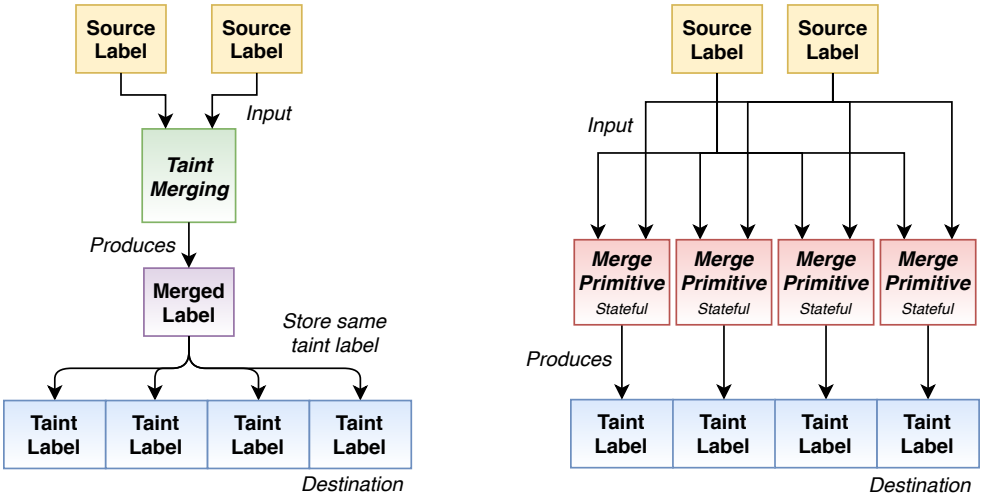
It needs to do *extra work* so that data-flows are comprehensively captured by our primitives and flexible propagation is achieved.

3.3.4 Limitations

One limitation of our design is that the tracking of implicit flows [88] is not considered. It is a well-known problem that tracking indirect flows leads to the taint explosion problem, where many memory locations are excessively over-tainted, resulting in impractical analyses. We therefore do not make any claims on the suitability of a taint-primitive-based approach for such data-flows.

The Taint Rabbit requires that the user places some further effort to implement taint policies when compared to existing solutions. Dytan only needs a single merging function to be implemented by the user, while the Taint Rabbit needs to be provided with three taint primitives in exchange for better flexibility.

Furthermore, the instruction handlers of the Taint Rabbit intensively use taint primitives to propagate taint. As illustrated in Figure 3.6, Dytan only invokes the merging function once to combine the labels stemming from two source operands.



(a) Dytan - store same merged label for all destination bytes

(b) The Taint Rabbit - invoke primitive for each destination byte

Figure 3.6: Differences in taint merging done by Dytan and the Taint Rabbit.

The same resulting label is then mapped to each destination byte. By contrast, the Taint Rabbit applies $src \rightarrow dst$ and $src, src \rightarrow dst$ on the source labels for each destination byte, as these primitives may be stateful and produce a different taint label per invocation. As a result, this enables better user customization of propagation than Dytan's approach but at the cost of more work.

3.3.5 Implementation

The Taint Rabbit is the core of the Dynamic Rabbits, a suite of binary analysis libraries for building taint-based tools targeting x86 applications. 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.

We chose DynamoRIO as a DBI engine due to its flexibility and performance. It is also an open-source project that is actively maintained. Meanwhile, the Dr. Memory framework, which is also built upon DynamoRIO, provides the shadow memory library Umbra [22]. We used Umbra to implement the Taint Rabbit and enhanced it to support 32-bit tags.

The Taint Rabbit provides an API to facilitate tool development. Listing 3.1 shows some declarations of notable functions offered by the API. For taint introduction, the tool calls `tb_taint_mem()` to taint a memory byte. Moreover, the retrieval of the taint label associated to a memory location is achieved via `tb_mem_get_taint_label()`. Finally, lines 18-42 show the type declarations of the three primitives required by the Taint Rabbit.

Listing 3.1: Declarations of some functions offered by the API of the Taint Rabbit.

```
/**
 *  Considers memory byte location, passed as parameter, as tainted.
 *
 *  @param addr The memory address to taint
 *  @param taint_label The label to associate with the addr. Note this value cannot
     ↪  be NULL
 *  as we use this value to denote if a location is not tainted.
 */
DR_EXPORT void tb_taint_mem(void *taint_bunny_ctx, uintptr_t addr, void *
    ↪ taint_label);

/**
 * Retrieves the taint label of an address
 *
 * @param addr The address to check.
 * @return Returns the taint label, or NULL if the passed address is not tainted.
 */
DR_EXPORT void* tb_mem_get_taint_label(void *taint_bunny_ctx, uintptr_t addr);

/**
 * Meet primitive.
 *
 * @param meet The meet label (updated by primitive).
 * @param src The source label.
 * @return Returns true if meet label reaches bottom of lattice.
 */
typedef bool (*tb_cc_meet_srcs_func_t)(void **meet, void *src);

/**
 * src -> dst primitive
 *
 * @param src The source label.
 * @return Returns the destination taint label.
 */
typedef void * (*tb_cc_propogate_1dst_1src_func_t)(void *src);

/**
 * src, src -> dst primitive
 *
 * @param src The first source label.
 * @param src2 The second source label.
 * @return Returns the destination taint label.
 */
typedef void * (*tb_cc_propogate_1dst_2src_func_t)(void *src, void *src2);
```

3.4 Evaluation

Our evaluation setup aims to validate the versatility of the Taint Rabbit. By using the Taint Rabbit to implement three different taint policies, we provide empirical evidence suggesting that a primitive-based approach enables generic taint analysis.

3.4.1 Control-Flow Hijacking Detection

The first use case is the detection of control-flow hijacking attacks. By using the Taint Rabbit, we developed a tool called `TR-CHECK` that performs bitwise tainting to track the taint statuses of locations similar to previous work [89]. Taint is introduced upon the reading of user input, and taint checks ensure that return addresses are

Table 3.1: Results for detecting control-flow attacks

Application	CVE ID	Taint Source	TR-CHECK
RTF2Latex	2004-1293	File	3.75 s
rsync	2004-2093	Environment Variable	0.26 s
Aeon	2005-1019	Command-line Argument	0.24 s
Nginx	2013-2028	Network	1.6 s

not corrupted. The $src \rightarrow dst$ primitive simply does a transfer operation, where the returned destination status is set to the source tag (i.e., an identity mapping). Meanwhile, both the $src, src \rightarrow dst$ and $src, src \rightarrow_M meet$ primitives perform a *bitwise or* to combine the statuses passed as arguments.

Table 3.1 presents the various buffer overflow attacks detected by our tool. We obtained our benchmarks from the online repository ExploitDB[3], and compiled them with exploit mitigations, such as stack canaries, turned off. The attacks stem from various sources; the payload targeting `Nginx` is sent via network traffic, while the attack on `rsync` starts from the reading of an environment variable. Overall, results suggest that the detection of such attacks is feasible with the use of taint primitives.

3.4.2 Use-After-Free Debugging

The second application, `TR-UAF`, uses taint analysis to track pointers and debug use-after-free vulnerabilities [10]. The 32-bit tags represent pointers to composite taint labels containing the statuses of tracked heap pointers (i.e., whether they are LIVE or DANGLING) as well as debugging information. These labels are shared with tainted pointers derived from the same root address, and reference counting is employed to manage their memory. The $src, src \rightarrow dst$ and $src, src \rightarrow_M meet$ primitives are based on Algorithms 1 and 2 respectively. Moreover, via the Taint Rabbit's API, the tool turns off taint propagation when analysing runtime code of allocation routines, in order to prevent false positives.

We ran `TR-UAF` on seven real-world applications vulnerable to a use-after-free and our results are given in Table 3.2. 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 analyser based on user-defined primitives.

[3]https://www.exploit-db.com/

Table 3.2: 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
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 3.3: 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

3.4.3 Taint-based Fuzzing

Although a thorough evaluation of the application of the Taint Rabbit to fuzzing is well beyond the scope of this work, 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. As detailed in Section 3.1, the policy uses bit vectors as taint labels. Both the $src, src \rightarrow dst$ and $src, src \rightarrow_M meet$ primitives perform union operations while $src \rightarrow dst$ simply returns the source's bit vector. Table 3.3 gives the bug counts obtained on LAVA [90] by VUzzer and our version, `TR-Fuzz`, in 6 hour runs.

3.4.4 Discussion

We aim to investigate whether a taint-primitive-based approach enables generic taint analysis. To test this research hypothesis, we demonstrate the versatility of the Taint Rabbit by considering three case studies. 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. We observe that for exploit detection and fuzzing, the $src, src \rightarrow dst$ and $src, src \rightarrow_M meet$ primitives have equivalent implementations. However, this is not the case for the policy of pointer tracking, thus highlighting the effectiveness of multiple taint primitives, as opposed to a single merge function.

3.5 Summary

In this chapter, we present a generic taint tracker called the Taint Rabbit. Our approach enables users to define custom taint propagation logic and label structures, allowing for powerful dynamic analyses. Therefore, versatility is the primary characteristic of the Taint Rabbit. Our design is based on the concept of taint primitives, which act as user-defined building blocks to perform taint propagation. Essentially, taint primitives are the main interface that bridge the user and the Taint Rabbit to employ desired taint policies. Our evaluation results suggest it is feasible to facilitate generic taint analysis via a primitive-based approach. In particular, we successfully demonstrate the employment of policies relating to fuzzing, UAF debugging and control-flow hijack detection, all of which are based on different propagation logic.

4

Optimizing Generic Taint Analysis with Call-Free Propagation

In the previous chapter, we presented a primitive-based approach to generic taint analysis, a powerful technique that enables the employment of various policies by using the same underlying taint engine. We now address the analysis's major limitation related to performance. In particular, such engines suffer from staggeringly high runtime overheads, as unlike specialized trackers, they cannot be optimized for a particular task. Overheads can reach as high as 30x on gzip as reported by the authors of Dytan [25] and hinder the analysis's practicality. Essentially, user-defined taint propagation implies that the analysis goes beyond just tracking status flags with the use of fast *bitwise or* operations, and therefore is a notable source of slowdown.

With the goal of enhancing the performance of binary taint trackers, we observe that dynamic binary instrumentation is the primary approach taken for their implementation. At runtime, instructions of the application are instrumented with taint propagation code and JIT-compiled to an intermediary code cache for execution. The transparency of the analysis is a strong requirement, as otherwise the execution of propagation code may inadvertently change the behaviour of the application. In order to alleviate the burden of maintaining transparency by tools of users, DBI frameworks support a clean call mechanism that conveniently performs context-switches before and after invoking an analysis routine.

However, as we showed in Chapter 2, the context-switches of clean calls are expensive as they comprehensively spill and restore registers, and perform stack swaps. Therefore, in order to improve performance, DBI engines also attempt to mitigate clean calls through inlining, but this optimization fails if

an analysis routine involves control-flow and is considered overall complex. While the simplicity of bitwise tainting is sufficient to trigger inlining, the rich propagation delivered by generic taint analysis result in such an optimization to be unsuitable. Consequently, since instructions are intensively instrumented with clean calls to conduct propagation, existing generic trackers, such as Dytan and DataTracker, are forced to incur severe slowdowns.

In this work, we mitigate clean calls by implementing the Taint Rabbit to use call-free propagation code. We ensure that no calls are done by manually building instruction handlers using raw, hand-crafted assembly instructions. Performance is enhanced as only registers required by the instruction handler are spilled/restored and the use of a stack is avoided. While a similar approach has been adopted by previous work [45], the propagation code implemented for existing solutions perform bitwise tainting and therefore simply use *bitwise or* instructions to merge taint. By contrast, we are the first to explore call-free propagation for generic taint analysis which is challenging due to the analysis's support of rich policies.

Although clean calls simplify implementation of propagation routines with the ability to code using high-level programming languages, e.g., C/C++, the low-level engineering required for call-free instrumentation pays off as runtime overhead is substantially reduced. Dytan achieves an average overhead of 225.6x on benchmarks concerning compression and image parsing when compared to native execution times, and the call-free optimization enables the Taint Rabbit to incur only 2.5x when configured to employ a multi-tag policy. Crucially, our taint primitive-based approach facilitates the employment of the optimization as users focus only on building call-free primitives, without the need to delve into the implementation of instruction handlers. Ultimately, optimizing generic taint analysis with call-free propagation is effective because it addresses the main implementation bottleneck of existing engines posed by the intensive use of clean calls.

4.1 Taint Propagation via DBI

Generic taint trackers built upon DBI leverage clean calls to trigger propagation routines. In order to improve performance, DBI engines try to automatically inline instrumentation code and mitigate the use of clean calls. However, as explained in Chapter 2, routines are only inlined if they are *simple* and do not include, for instance, any branches or calls to sub-routines [44, 83].

LibDFT [24] exploits the inline optimization. Listing 4.1 shows one of its taint propagation routines. Essentially, propagation is done by bitwise tainting, which

Listing 4.1: 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.

```
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 }
```

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 [42] 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, particularly the adding of support for multi-tag propagation, DataTracker is used by VUzzer. However, the changes made in DataTracker break the original inline optimization. Listing 4.2 gives the instruction handler that corresponds to the one in listing 4.1. 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.

4.2 Taint Propagation without Clean Calls

Building a generic taint engine with a high-level programming language renders taint propagation code produced by off-the-shelf compilers too complex to be automatically inlined by a DBI tool. Consequently, unlike bitwise tainting, generic taint propagation implemented by using clean calls leads to expensive context-switching. Therefore, the Taint Rabbit's instruction handlers are built using hand-crafted x86 assembly code to ensure no calls are used. Although Dr. Memory [45] take a similar approach, their instruction handlers are coded for bitwise tainting, using simple *bitwise or* operations to merge taint flags. Therefore, in our work, we explore the effectiveness of this call-free optimization particularly for generic taint analysis.

Listing 4.2: 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 ...
```

4.2.1 Mitigating Clean Calls in Taint Primitives

In order to gain performance, the user is required to supply optimized taint primitives written in assembly, avoiding the use of clean calls as much as possible in the dominant execution path of the primitives. Although implementing optimized taint primitives can be challenging for the user, depending on the desired taint policy, the effort pays off significantly in terms of performance, as indicated by our results in Section 4.3. Furthermore, the primitive-based approach adopted by the Taint Rabbit hides internal complexities, such at the implementation of instruction handlers, from the user who therefore can solely focus on optimizing taint primitives. In other words, the design of the Taint Rabbit facilitates the adoption of call-free taint propagation.

The taint primitives are given memory operands that refer to source taint labels in shadow memory, and two general purpose (GP) scratch registers for their implementation. Further register reservation can be performed if the primitive requires additional registers.

4.2.2 Optimizing Instruction Handlers

The instruction handlers of the Taint Rabbit harness the optimized primitives supplied by the user and are also implemented in x86 assembly. Many instructions, including SIMD, are supported by our handlers. Table 4.1 provides a comprehensive list of the supported instructions.

We designed the code to follow known practices for optimization to the best of our abilities [91, 92]. Iteration and branching are reduced with instruction

Table 4.1: 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.

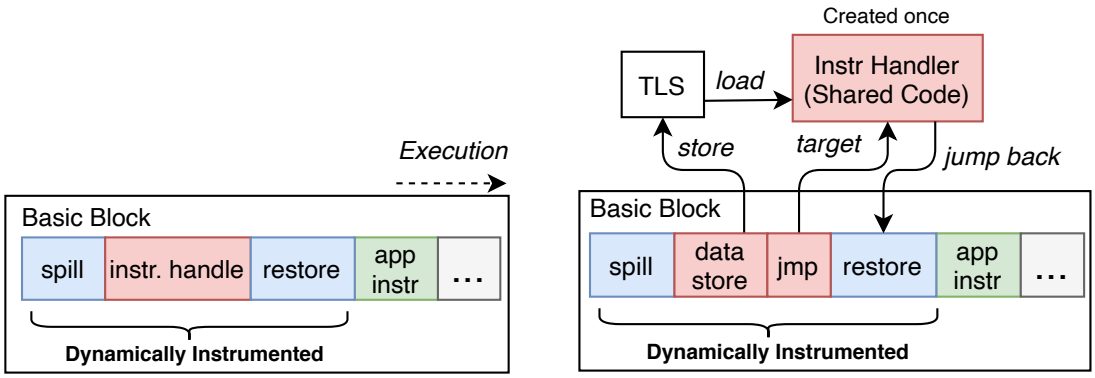
add	or	adc	sbb	and	sub	xor	inc	dec	push	pop
imul	call	mov	lea	xchg	cwde	cdq	leave	rdtsc	cmovo	cmovno
cmovb	cmovnb	cmovz	cmovnz	cmovbe	cmovnbe	cmovs	cmovns	cmovp	cmovnp	cmovl
cmovnl	cmovle	cmovnle	punpcklbw	punpcklwd	punpckldq	packsswb	pcmpgtb	pcmpgtw	pcmpgtd	packuswb
punpckhbw	punpckhwd	punpckhdq	packssdw	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					

handlers, amounting to over 970 in count, specialized not only to different opcodes but also to operand sizes and types. For instance, a dedicated instruction handler is available for each variant of the `mov` instruction, in order to handle cases when the instruction copies 1, 2 and 4 bytes and operates on memory, registers and immediate values. Furthermore, loops, such as those in Algorithms 3 and 4 (presented in Chapter 3), are unrolled. Finally, instruction handlers do not use a dedicated stack but rely on thread-local storage and registers for memory. Therefore, unlike clean calls, stack swaps are not necessary.

4.2.3 Instrumentation

Apart from clean calls, there are two other main approaches to instrument application instructions with propagation code. As illustrated in Figure 4.1, these approaches are inlining and outlining via a jump-and-link mechanism [93]. With inlining, propagation code is inserted directly into basic blocks along with the application's instructions. On the other hand, outlining directs control to shared code caches to perform taint propagation. Specifically, after spilling scratch registers, control jumps to the code cache of an instruction handler located outside basic blocks, and jumps back after execution, restoring scratch registers.

Importantly, jump-and-link does not use clean calls to invoke the execution of outlined propagation code, but simple trampoline instructions. Moreover, it reduces fragment sizes as propagation code is shared among instrumentation across basic blocks. However, this brings about a drawback concerning overhead stemming from the thread-local storage of operand data, e.g. register IDs, and the jump address for



(a) Inline **(b)** Outline

Figure 4.1: Inline and Outline Instrumentation

backwards link. Moreover, since the exit of shared code involves an indirect jump, the CPU's branch predictors are pressured, potentially degrading performance.

Inlining avoids these limitations but runs the risk of severe code bloating and large code fragments to be managed by the DBI engine. Unfortunately, the intensive instrumentation of application code required to track taint exacerbates the problem. Consequently, since the Taint Rabbit is a generic taint engine and cannot make strict assumptions on user-defined taint primitives, instrumentation is achieved via a jump-and-link approach by default. However, if needed by the user, inlining is also supported and enabled simply via a configuration flag.

4.2.4 Other Optimizations

We also adopt previously proposed optimizations [24, 40, 45]. Firstly, live register analysis is done to only spill/restore register values that are relied upon by subsequent application instructions. Therefore, dead registers do not need spilling. Secondly, 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.

4.2.5 Limitations

Currently, the Taint Rabbit does not analyse 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. 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 specific issue with an inline custom allocator that performs clean calls in a slow-path only when requesting additional memory for management.

Lastly, the Taint Rabbit uses additional memory, and the memory overhead may cause issues when analysing 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.

4.3 Evaluation

Our evaluation of the Taint Rabbit aims to answer the following three main research questions:

- `RQ1`: How much does call-free propagation improve the performance of generic taint analysis?
- `RQ2`: By leveraging call-free propagation, how does the performance of generic taint analysis compare to the existing state-of-the-art trackers?
- `RQ3`: Does call-free propagation aid in scaling generic taint analysis to real-world target applications?

To answer these questions, we measure the Taint Rabbit's performance on real-world software relating to data compression, PHP, image parsing, and Apache as well as the SPEC CPU 2017 benchmarks. Experiments ran 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[1], along with scripts for running many of our experiments[2], are available online. The specific version of the Taint Rabbit that we used for our experiments is available as well[3].

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

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

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

To measure performance, we consider the elapsed times taken to execute our benchmarks while under taint analysis and proceed by calculating resulting overheads with respect to native executions. Usually, the real overhead of a taint engine is considerably apparent when using CPU bound applications as they enforce taint propagation due to the high computation. Nevertheless, we also consider IO bound programs, such as Apache, with the rationale that our evaluation benefits from running various types of benchmarks, thus reducing threats to validity that concerns the generalisability of our results.

4.3.1 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. The engineering effort required to implement another taint engine, namely `TR-CC`, as our baseline was worthwhile to answer `RQ1`.

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 and facilitate the development of concolic execution, as detailed in Chapter 2. ID assignment is achieved by using $src \rightarrow dst$ and $src, src \rightarrow dst$ primitives that increment a global counter if any source is tainted (recall that these primitives can de stateful as explained in Chapter 3). 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, nodes about to be inserted 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, these two taint policies cannot be implemented with a bitwise taint engine.

4.3.2 Other Taint-Based Systems

To answer RQ2, we ran nine other taint analysers on our benchmarks as baselines for comparison. Table 4.2 gives a summary of their main features. LibDFT 3.1415 alpha [24] inlines bitwise taint analysis, while Dytan[4][5] [25] performs user-defined operations on bit vectors that contain multiple tags. Triton 0.6 [72] is a dynamic binary analysis framework which is set up to use Pin 2.14 for tracing. DataTracker[6] [42] focuses on data provenance; a variant, named DataTracker-EWAH[7] [7], records input offsets to optimize fuzzing. We also ran BAP-PinTraces[8] [94], 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[9] [53] is a QEMU-based taint tracker that inlines precise bitwise propagation into Tiny Code Generator (TCG) instructions, and Taintgrind 3.15.0 [89] is a taint engine built upon Valgrind [33]. Moreover, the Dr. Memory 2.1.17972[10] [45] 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, labelled as `Pin-Null`[11], `DR-Null` and `Nullgrind`, respectively. DECAF, only using its virtual machine introspection and with no taint analysis, is labelled as `DECAF-VMI`.

4.3.3 Performance

To measure performance, we configure the Taint Rabbit to taint all data read from files, sockets, command-line arguments and environment variables so that instruction handlers are made to process and propagate taint labels. No taint is introduced when running other tools, which nevertheless instrument instructions even though no taint is propagated. However, our setup benefits existing tools that perform

[4] https://github.com/dytan-taint-tracking/dytan-taint-tracking/commit/5211823d575
[5] We modified Dytan by removing failing assertions for unsupported x86 instructions.
[6] https://github.com/m000/dtracker/commit/dc729dca8
[7] https://github.com/vusec/vuzzer/commit/f6f7d593a
[8] https://github.com/BinaryAnalysisPlatform/bap-pintraces/commit/bed2b108
[9] https://github.com/decaf-project/DECAF/commit/1de4ed7c95
[10] https://github.com/DynamoRIO/drmemory/commit/5b988e31
[11] We ran the same Pin-Null tool provided by LibDFT 3.1415 alpha [24].

Table 4.2: 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 [24]	Byte	Bit/Byte	Bitwise	Under	Pin
Triton [72]	Byte	Bool	Bitwise	Over	Pin
Dytan [25]	Byte	Bit-Vector	Generic	Over	Pin
DataTracker [42]	Byte	Set	Set Union	Under	Pin
DataTracker-EWAH [7]	Byte	Compressed Set	Set Union	Under	Pin
BAP-Pin Traces [94]	Byte	32-Bit Unsigned Offset	Set to *top*[a]	Over	Pin
Taintgrind [89]	Byte	Bit	Bitwise	Under	Valgrind
DECAF [53]	Bit	Bit	Bitwise	Over	QEMU
Dr. Memory [45]	Byte	2 Bits	Bitwise	Under	DynamoRIO
Taint Rabbit	Byte	32-Bit Word	Generic	Over	DynamoRIO

efficiently 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 assess 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 Figure 4.2. 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 average overheads of 258x and 3.4x respectively compared to native runs. One of our best results is on `bzip2`, where `TR-RAW-BV` achieves a speed-up 99% over `TR-CC-BV`.

Call-free taint propagation also results in the Taint Rabbit to outperform all other existing generic taint engines that we evaluated. The overheads incurred by Dytan and DataTracker are 292.7x and 15x respectively and are significantly higher than that obtained by `TR-RAW`. Moreover, Triton suffers from the heaviest slowdown; we aborted Triton's run on `bzip2` after 60 hours and therefore no result is reported. However, the Taint Rabbit is slower than bitwise tainting, particularly that delivered by LibDFT. Results show that `TR-RAW-ID` incurs an average overhead of 3x, which is more than LibDFT's overhead of 1.9x.

PHP. We have applied the Taint Rabbit to PHP 7.2.4, driven by PHPBench[12] 0.15.0 [95], a framework that provides a collection of micro/macro benchmarks for

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

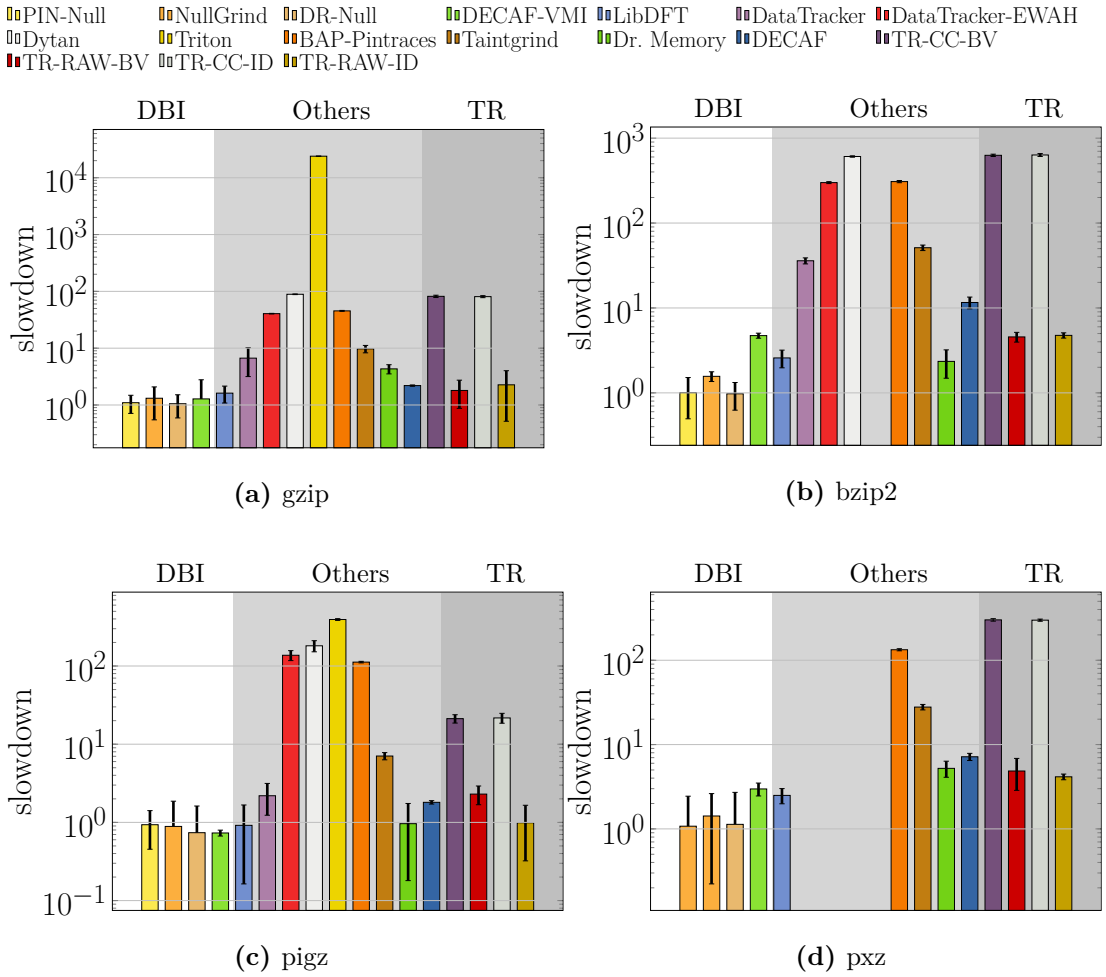


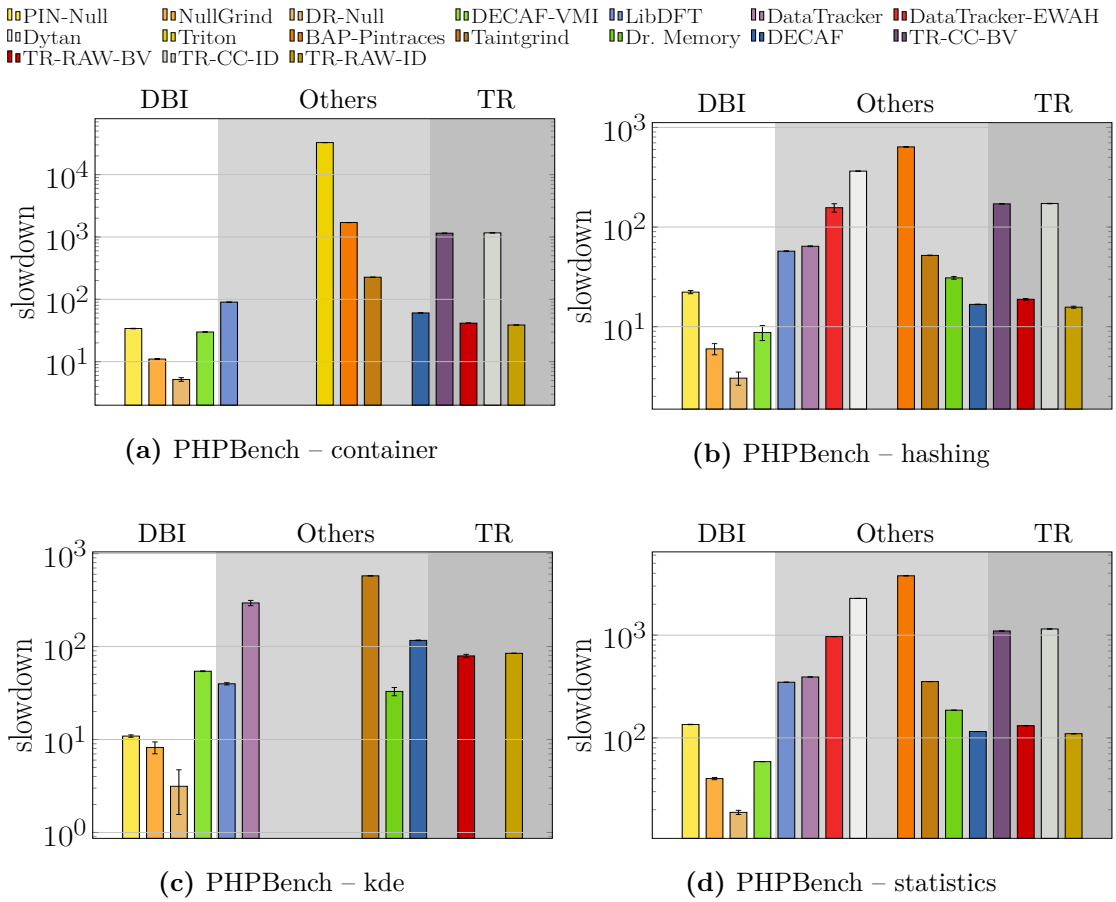
Figure 4.2: Results of call-free taint propagation on data compression benchmarks. Missing entries imply that the corresponding taint engine timed-out or crashed.

measuring performance. We ran the following benchmarks with 10,000 revolutions each: `container`, `hashing`, `kde` and `statistics`.

Our results on PHPBench are illustrated in Figure 4.3. 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 blame is due to the slowdown imposed by taint analysis. We do not encounter this error when benchmarking performant engines such as LibDFT, `TR-RAW-BV` and `TR-RAW-ID`.

Overall, the avoidance of clean calls greatly improves the performance of generic taint analysis on PHP. On the benchmarks `TR-CC-ID` managed to run (i.e., excluding `kde`), `TR-RAW-ID` obtains an average speed-up of 93%.

Image Parsing. We consider `djpeg`, version 9c, and `gif2png` 2.5.8 as two exemplars of image parsing. Results are given in Figure 4.4. Similar to the



(a) PHPBench – container

(b) PHPBench – hashing

(c) PHPBench – kde

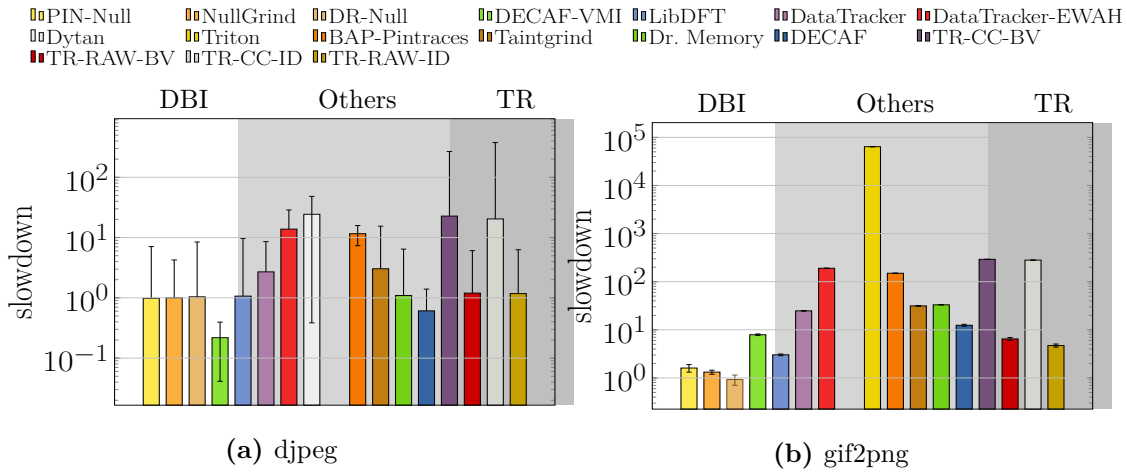
(d) PHPBench – statistics

Figure 4.3: Results of call-free taint propagation on PHPBench. Missing entries imply that the corresponding taint engine timed-out or crashed.

results presented so far, we again observe the high overheads incurred by existing generic taint engines. For instance, DataTracker, DataTracker-EWAH and Dytan obtain average slowdowns of 2.7x, 13.8x and 24.3x respectively on `djpeg` over native execution time. Meanwhile, `TR-RAW-BV` incurs a lower overhead of 1.2x. We also observe again that the specialized bitwise tainting provided by LibDFT is faster, albeit slightly, with an overhead of 1.1x.

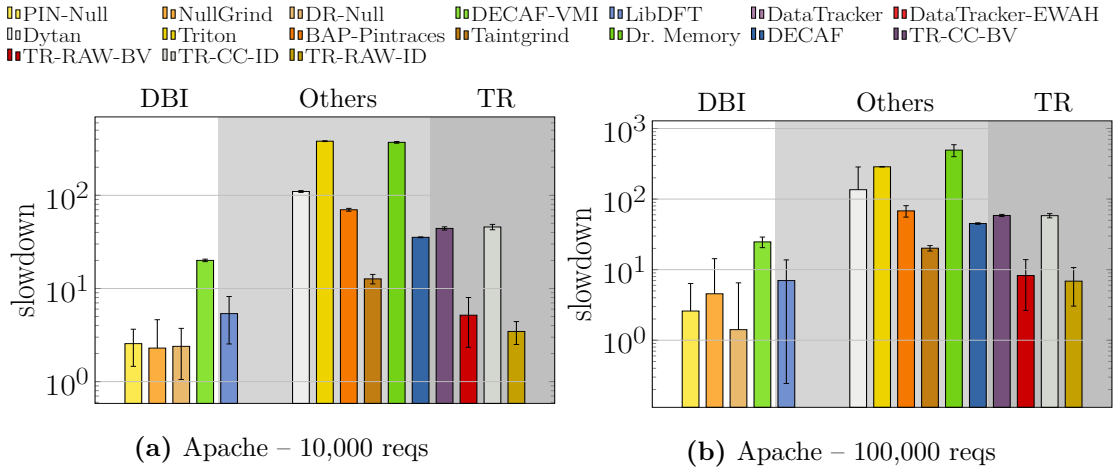
Apache. Figure 4.5 depicts our results on Apache 2.4.33. The benchmark tool `ab` [96] was used to send 10,000 and 100,000 requests to Apache while under the analysis of our taint trackers. `TR-RAW-BV` and `TR-RAW-ID` perform faster than their clean-call based alternatives, with average speed-ups of 87% and 90% respectively.

With call-free propagation, the Taint Rabbit also outperforms existing trackers such as Dytan, DECAF, Taintgrind and BAP-Pintraces. Furthermore, since Apache primarily performs I/O and process forks to handle requests, Triton is able to



(a) djpeg **(b)** gif2png

Figure 4.4: Results of call-free taint propagation on image parsing applications. Missing entries imply that the corresponding taint engine timed-out or crashed.



(a) Apache – 10,000 reqs **(b)** Apache – 100,000 reqs

Figure 4.5: Results of call-free taint propagation on Apache. Missing entries imply that the corresponding taint engine timed-out or crashed.

complete the experiment, albeit with 334x overhead. Unfortunately, DataTracker and DataTracker-EWAH both stalled and timed-out after 3 hours.

SPEC CPU 2017. The average overheads observed on the SPEC-rate 2017 Integer benchmark 1.0.2[13] are given in Figure 4.6. We excluded the `gcc` and `x264` benchmarks when calculating results because of known limitations of the Taint Rabbit. Firstly, the Taint Rabbit ran out of memory on `gcc` for both TR-ID and TR-BV. This problem might be mitigated in future work by using a 64-bit

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

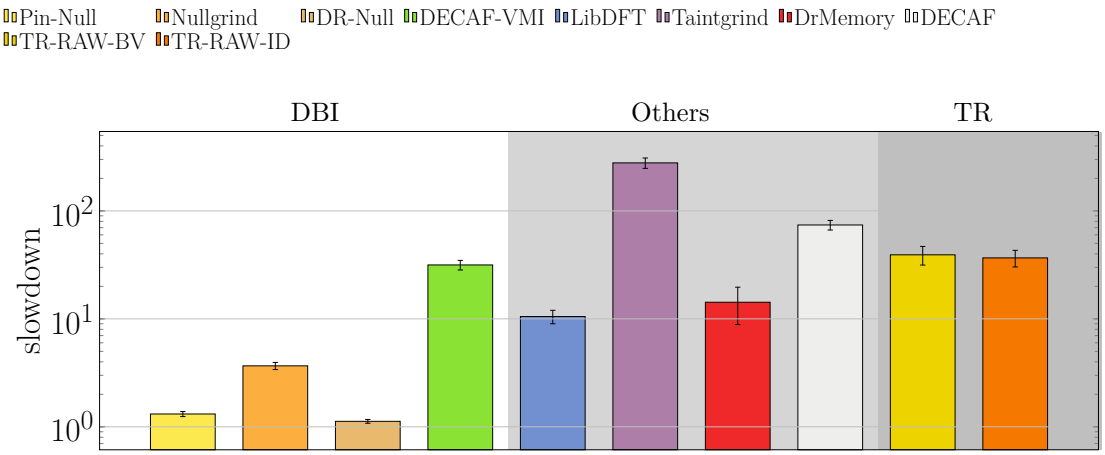


Figure 4.6: Performance results of inlined taint propagation on SPECrate 2017 (excluding `gcc` and `x264`)

architecture. Secondly, `TR-RAW-BV` times-out on `x264` because of high overhead. Moreover, we terminated experimentation with `TR-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 analysers, we focused on the optimized versions when running the SPECrate benchmark. DataTracker also faced issues as it crashed on all benchmarks except just one. The remaining benchmark, namely `x264`, exceeded 24 hours.

Our results indicate that on average, the Taint Rabbit is faster than Taintgrind on the SPEC CPU benchmark. However, our taint engine, with call-free taint propagation, still fails to surpass the inlined bitwise tainting performed by LibDFT and Dr Memory. In particular, `TR-RAW-ID` incurs an overhead of 36.7x over native execution, while LibDFT and Dr. Memory obtain overheads of 10.6x and 19.8x respectively.

4.3.4 Research Questions

RQ1: *How much does call-free propagation improve the performance of generic taint analysis?*

On all of our benchmarks, the Taint Rabbit performs substantially faster when clean calls are avoided for taint propagation. Although the user is required to implement taint primitives using assembly instructions to enable the optimization, the effort significantly pays off as the optimization essentially addresses the main bottleneck of expensive context switching. For instance, 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.

RQ2: *By leveraging call-free propagation, how does the performance of generic taint analysis compare to the existing state-of-the-art trackers?*

Inherently, specialized and generic taint engines have opposing performance and versatility trade-offs. The Taint Rabbit has to perform heavier analyses to support custom and complex taint propagation logic. We therefore cannot expect that the Taint Rabbit is faster than optimized bitwise tainting. With this in mind, our results on SPEC CPU suggest that even when employing the call-free optimization, Taint Rabbit is slower than LibDFT and Dr. Memory. However, our optimization led the Taint Rabbit to be the fastest generic taint engine among those we evaluated. For instance, on `bzip2`, the Taint Rabbit has an overhead of 4.6x over native execution, and substantially outperforms DataTracker and Dytan, which incur overheads of 36x and 607.9x respectively.

RQ3: *Does call-free propagation aid in scaling generic taint analysis 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 better analysed. For instance, DataTracker fails to run SPEC CPU, while `TR-RAW-ID` achieves an average overhead of 36.7x. We also observed similar scaling issues faced by existing engines on Apache and PHP. However, 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 analyse 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.

4.3.5 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, shown in Figure 4.7, indicate no major changes in performance (with an average overhead difference of $\sim$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.

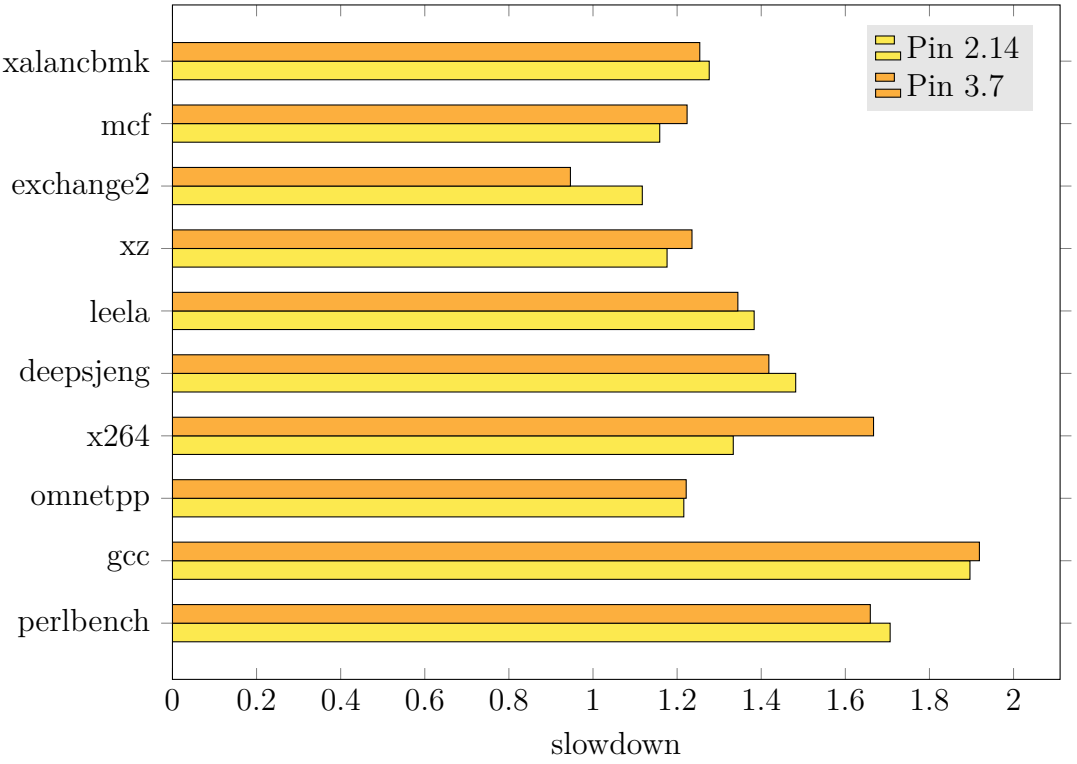


Figure 4.7: Performance of PinNull on Pin 2.14 and 3.7

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 different taint policies and a wide range of complex benchmarks.

4.4 Summary

Dynamic binary instrumentation is the underpinning technique used to implement standard taint trackers. To ease development, especially in relation to maintaining transparent analysis, trackers leverage clean calls to implement propagation. However, this comes with the cost of expensive context switching. While DBI engines attempt to automatically inline analysis code to avoid clean calls, the built-in optimization fails when the code is complex. Given the complexity of generic taint analysis, the overhead of clean calls are therefore imposed upon existing trackers, thus creating a severe performance bottleneck.

In this chapter, we investigate the performance benefits of manually optimizing generic taint analysis with call-free propagation. In particular, we build instruction

handlers using hand-crafted x86 code to minimize the use of clean calls. While existing solutions adopt a similar approach to avoid clean calls through inlined taint analysis, the overall approach was only explored within the context of specialized bitwise tainting.

This chapter also delves deeply into our evaluation results, which overall show that ID and multi-tag propagation perform substantially faster when clean calls are avoided. For instance, on compression benchmarks, the Taint Rabbit incurs average overheads of 258x and 3x when conducting ID propagation with and without clean calls respectively. The optimization is highly effective as it addresses the main implementation bottleneck of existing approaches - so much so, that it led the Taint Rabbit to achieve the best performance results when compared to other generic (non-bitwise) taint trackers, such as Dytan and DataTracker.

5

Optimizing Generic Taint Analysis with Vectorized Propagation

Dynamic taint analysis is a well-established technique in software security. However, its practical value is often limited owing to excessive performance overheads. The primary bottleneck is the propagation of taint labels during runtime, where its negative impact is exacerbated if the analysis performed is generic. In Chapter 4, we showed that the mitigation of clean calls used to conduct propagation substantially reduces overheads. We now address the performance problem even further by investigating the hypothesis whether vectorization is an effective means to accelerate generic taint propagation.

In a nutshell, vectorization operates on multiple data elements, stored in large registers, simultaneously with single instructions [92]. Along these lines, the key idea of our work is to process and propagate taint labels in parallel rather than sequentially as done in Chapter 4. Our research into vectorization is propelled by the existing support for SIMD processing in commodity CPUs. Therefore, unlike other efficient solutions which depend on specialized hardware [17, 97–99], we avoid limitations concerning the wide deployment of our optimizations.

It is worth mentioning that previous work, particularly Minemu [41], has already investigated using SIMD to speed up taint analysis. However, there still remain several research challenges that need to be addressed. Firstly, Minemu performs bitwise tainting, and no notable attention has been given to designing vectorized *generic* taint analysis. It simply uses the vector registers as fast storage for taint labels. Secondly, no register liveness analysis is performed to spill and restore the vector registers used by Minemu even though these registers are shared with the

target application. Therefore, to avoid the clobbering of taint information, Minemu requires that the target application never uses vectorization for its own processing; an assumption that quickly breaks when analysing any modern software. In turn, this limitation prevents a thorough evaluation of the overall approach.

To this end, we extend the Taint Rabbit by implementing our design of vectorized generic taint analysis. As a result, users are able to invest in building taint primitives using SIMD instructions in exchange for a potential gain in performance. Specifically, assuming byte-granularity, vectorized taint primitives process four taint labels associated to four bytes of a `dword` operand simultaneously. This is in contrast to a scalar implementation, which loads and propagates these four labels one by one.

In this chapter, we describe the implementation of vectorized taint primitives from the perspective of the user with the consideration of two use-case, namely ID and multi-tag propagation. We also explain how these primitives are leveraged internally by the instruction handlers of the Taint Rabbit to accelerate the analysis.

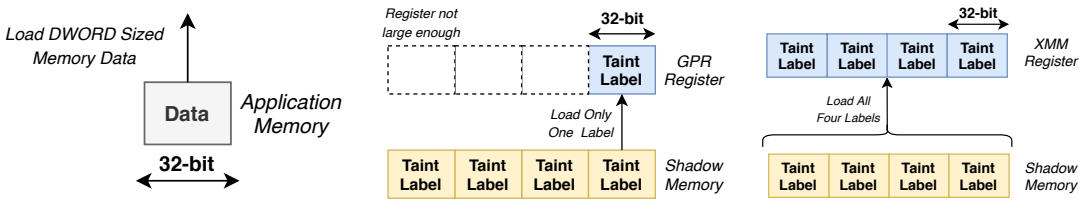
The performance of vectorized generic propagation is evaluated by considering benchmarks used in Chapter 4. These benchmarks include SPEC CPU 2017, PHP, Apache and various data compression and image parsing tools. For example, results of data compression benchmarks demonstrate that our optimization is effective in comparison to the scalar baseline as average slowdown is reduced from 3.4x to 2.6x over native execution, when employing a multi-tag taint policy. We argue that many applications of dynamic taint analysis in security are suitable for vectorization and can therefore benefit from our method.

5.1 Vectorized Taint Propagation

At its core, the Taint Rabbit is designed to be generic and to support a broad variety of user-defined taint policies. It tracks a pointer for each tainted byte of memory and registers. The pointer acts as a tag directly or as a reference to a custom taint label data structure (e.g., a bit vector), as required by the policy.

State-of-the-art generic taint trackers use scalar (non-vectorized) implementations for propagating taint labels. Consequently, since both the tags and the registers used to implement propagation are both pointer-sized, taint propagation code has to process the labels of multi-byte operands sequentially, one by one.

Therefore, this work proposes vectorization as a means to speed-up taint propagation. We observe that the instructions of an application often use operand sizes that are equivalent to the word size of the architecture. Furthermore, assuming byte-level taint granularity [51], we also note that a SIMD vector register is typically



(a) Application Load **(b)** Loading to GP reg **(c)** Loading to SIMD reg

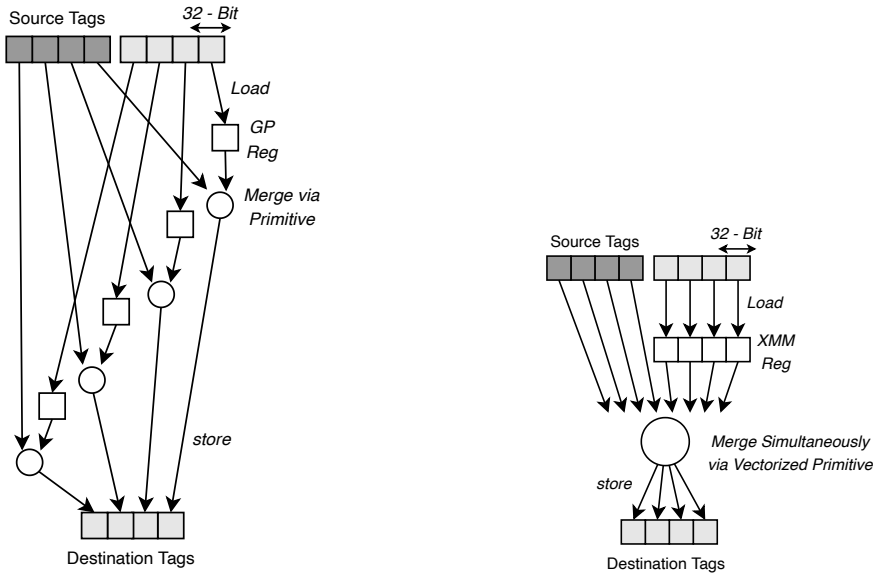
Figure 5.1: Unlike a general purpose register, a SIMD register is large enough to store all labels pertaining to a `dword` sized operand of an application's instruction

wide enough to conveniently store the tags of all bytes of the word. This is illustrated in Figure 5.1 for the x86 32-bit architecture. While an x86 general purpose register (GPR) stores only one 32-bit tag, a single XMM vector register is large enough to store four 32-bit tags. We have not implemented vectorized taint analysis for other architectures but we remark that these observations are true on other popular platforms as well, e.g., ARM and PowerPC. Moreover, on x86_64 architectures, one AVX-512 register can store the eight pointer-sized tags pertaining to a `quadword` sized operand.

As a result of the convenience of vector registers, the Taint Rabbit's key contribution is its ability to support *vectorized* taint primitives. In particular, the Taint Rabbit allows users to invest in implementing taint primitives using SIMD instructions so that multiple taint labels are processed simultaneously. While scalar taint primitives accept single source labels as parameters, vectorized primitives instead take *vectors* of source labels. The benefit is illustrated in Figure 5.2. The scalar implementation of the $src, src \rightarrow dst$ primitive would be invoked multiple times to derive a label for each byte of a `dword` sized destination operand. Meanwhile, a vectorized implementation is invoked only once for all bytes. This leads to a reduction in shadow memory accesses and less instrumented code, thus improving performance.

5.2 Applicability

It is well understood that not all algorithms can be effectively vectorized, often due to control-flow divergence, nested data-structures or non-contiguous memory access [100]. Therefore, we overtly assume that the taint policy considered by the user is indeed vectorizable to benefit from the Taint Rabbit's optimizations. Unfortunately, this assumption impacts our claim on versatility, as contrarily, policies which cannot be vectorized are not supported. To address this issue, we gauge the significance of our assumption by considering several policies employed



(a) Scalar Propagation (Non-Vectorized) **(b)** Vectorized Propagation

Figure 5.2: Optimizing generic taint propagation through vectorization

in previous work and investigate whether their propagation, such as taint merging, can be vectorized. In total, we have studied twenty papers to determine the potential impact of vectorized taint propagation, and our results are summarized in Table 1 (found in the Appendix).

We observe that out of the twenty policies, fifteen could benefit from vectorization. Specifically, six works rely on taint status tracking, which is a policy that benefits from vectorization, as already demonstrated by previous work [41]. Moreover, we also note that other applications [7, 101–105] require the capability of tracking multiple tags. Unlike other vectorized taint engines, this policy is also supported by the Taint Rabbit when employing vectorized taint propagation. Finally, five policies [106–110] appear not to be supported. Overall, this is due to the following reasons:

1. **Granularity.** Our approach to vectorized taint propagation is unsuitable for policies which rely on coarse taint granularities. For instance, such a policy may deal with specific program source concepts, e.g. Javascript [106], or have tags associated with high-level modules [107]. The main problem is that these policies do not naturally consider multiple source taint labels during propagation. By contrast, assuming byte-level granularity, multi-byte-sized source operands result in the propagation and processing of all their tags, thus being ideal for vectorization.

2. **No Taint Propagation.** Certain policies cannot be vectorized because they actually do not perform any taint propagation. For instance, previous work [108] takes a multi-faceted approach which detects policy violations not by propagating taint labels but via output comparisons.

3. **Nested Data Structures.** Nested data-structures are problematic for vectorization. We encountered such an issue when considering the taint policy employed by Lin et al. [109]. Essentially, a taint label is defined as a compound data-structure, which in turn includes a set of references to other taint labels. Propagation requires accessing this set for multiple source tags, and its vectorization is difficult to fan out and access all of the data points simultaneously.

5.3 Vectorized Taint Primitives

When employing vectorized and generic taint analysis, the Taint Rabbit uses the same kind of primitives, namely (1) $src \rightarrow dst$, (2) $src, src \rightarrow dst$, and (3) $src, src \rightarrow_M meet$, as explained in Chapter 3. However, vectorized primitives operate over vectors so that multiple labels are processed simultaneously. Therefore, to differentiate from scalar primitives, we denote corresponding vectorized primitives as (1) $src_{vector} \rightarrow dst_{vector}$, (2) $src_{vector}, src_{vector} \rightarrow dst_{vector}$, and (3) src_{vector}, $src \rightarrow_M meet$[1].

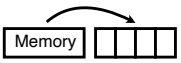
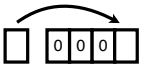
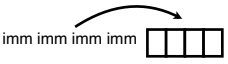
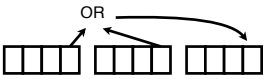
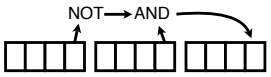
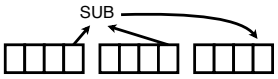
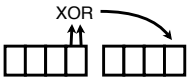
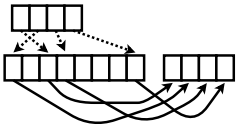
5.3.1 Examples of Taint Primitives

In order to describe vectorized taint primitives, we consider two use-cases that a user might implement. We present our propagation algorithms by adopting a notation that resembles Intel intrinsic instructions to denote SIMD operations. While the reader is encouraged to refer to Intel's guide[2][111] for further details, Table 5.1 summarizes several intrinsic instructions used by our algorithms.

[1]Apart from a source vector, the $src_{vector}, src \rightarrow_M meet$ also takes a single label as input that indicates the *current* meet value. Internally, this parameter enables the Taint Rabbit to chain the primitive to compute the meet label of a source operand that is associated with more tags than can fit in a single vector register. Section 5.4 provides further details.

[2]https://software.intel.com/sites/landingpage/IntrinsicsGuide/

Table 5.1: Summary of several Intel Intrinsic Instructions [111]

Intrinsic Instruction	Instruction	Description	Visual
store_si128	`movdqa`	Store 128-bit data to memory	
load_si128	`movdqa`	Load 128-bit data from memory	
cvtsi32_si128	`movd`	Copy 32-bit data to first lane, clearing all upper lanes	
set_epi32	*Sequence*	Set lanes with immediate integers	
or_si128	`por`	Perform `OR` on 128-bit operands	
andnot_si128	`pandn`	Perform `NOT` on first 128-bit source, and `AND` result with second source	
sub_epi32	`psubd`	For each lane, perform a subtraction	
cmpeq_epi32	`pcmpeqd`	For each lane, compare elements, and if equal set result to all 1s	
setzero_si128	`pxor`	Perform `XOR` with same 128-bit source to clear register with zeros	
unpackhi_epi32	`punpckhdq`	Interleave high 32-bit elements	
shuffle_epi32	`pshufd`	Based on a control mask, shuffle data to different lanes. If control is zero, broadcast first element to all lanes	
gather_epi32	`vpgatherdd`	Load 32-bit elements from non-contiguous memory specified by an index vector	

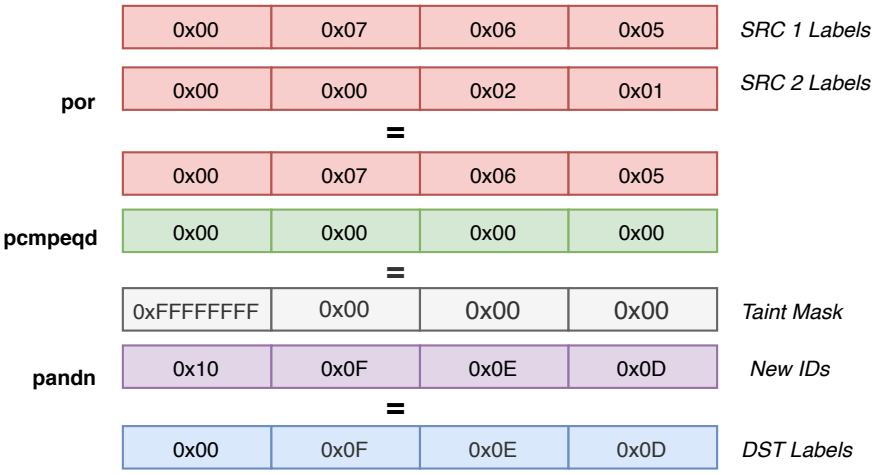


Figure 5.3: Vectorized ID Assignment

ID Propagation

The first considered propagation policy generates a unique numerical ID for every tainted destination byte upon the execution of an application's instruction. It checks whether a source byte of an instruction is tainted, and in such cases, increments a global counter to generate a new ID. This ID is then mapped to the corresponding destination byte that is influenced by the tainted data. We already considered the same policy in Chapter 4, but now present its vectorized variant for better performance. Indeed, the scalar approach expensively generates IDs for each tainted byte of a destination operand sequentially. Instead, our vectorized variant, also supported by the Taint Rabbit, generates and maps multiple fresh IDs to destination bytes simultaneously.

Vectorizing Taint Merging. We focus on the $src_{vector}, src_{vector} \rightarrow dst_{vector}$ primitive as an example. The vectorized primitive is detailed in Algorithm 7 and illustrated in Figure 5.3. It takes two source vectors, each containing four taint tags in the form of numerical IDs, as input and produces a vector of fresh IDs to associate to four consecutive destination bytes. Note, the consideration of four labels as a count stems from the assumption that the pointer size of the architecture, e.g., x86, is four bytes.

The algorithm begins by incrementing a global counter by four to reserve the next new IDs (line 2), where the result is then stored in the starting lane of a vector called *counter_vector* (line 3). It proceeds by broadcasting the value to all of the vector's four lanes using the `pshufd` SIMD instruction at line 4. Next, we subtract the vector by a constant vector, called *template_vector*, which is set up

Algorithm 7: The $src_{vector}, src_{vector} \rightarrow dst_{vector}$ primitive for ID propagation. New IDs are generated and assigned to tainted bytes of a destination.

Data: Vector: src_vector_1, src_vector_2, $counter\_vector$, $mask\_vector$, $template\_vector$, $zero\_vector$, Integer: $counter$
Result: Vector: dst_vector, Integer: $counter$

```
// Increment counter by 4 to prepare for the new IDs.
1  pointer_size ← 4;
2  counter ← counter + pointer_size
   // Broadcast the counter.
3  counter_vector ← cvtsi32_si128(counter);
4  counter_vector ← shuffle_epi32(counter_vector, 0);
   // Generate ID for each vector lane.
5  template_vector ← set_epi32(0, 1, 2, 3);
6  counter_vector ← sub_epi32(counter_vector, template_vector);
   // Create mask.
7  mask_vector ← or_si128(src_vector1, src_vector2);
8  zero_vector ← setzero_si128();
9  mask_vector ← cmpeq_epi32(mask_vector, zero_vector);
   // Set IDs only to tainted lanes.
10 dst_vector ← andnot_si128(mask_vector, counter_vector);
11 return dst_vector, counter;
```

with decrementing values starting from 3 down to 0. This results in generating a new ID at each lane of *counter_vector* at line 6.

However, the policy requires that IDs are only associated to bytes that are affected by tainted sources. In other words, if both source vectors hold "no taint", i.e., NULL values, at a given lane, then no ID should be assigned to the destination vector at that same lane. The primitive achieves this propagation by merging the two sources, via a *bitwise or*, at line 7. A "no taint" mask, indicating which lanes do not deal with tainted data, is then created by comparing the result with a vector of zeros using the `pcmpeqd` instruction (line 9). This mask is then applied to the *counter_vector* via an *and-not* operation (i.e., the `pandn` instruction) so that generated IDs only map to tainted lanes. The result is assigned to *dst_vector* at line 10, thus concluding the vectorized primitive.

Although the presented algorithm abstracts away from certain implementation details for simplicity, it is important to note that our primitive checks for integer overflows when incrementing the global counter at line 2. Since "no taint" is represented as a `NULL` value, `ID 0` cannot be assigned as a valid ID. Consequently, if an overflow occurs, the value of the counter wraps around but starts at `ID 4`, thus skipping `ID 0`.

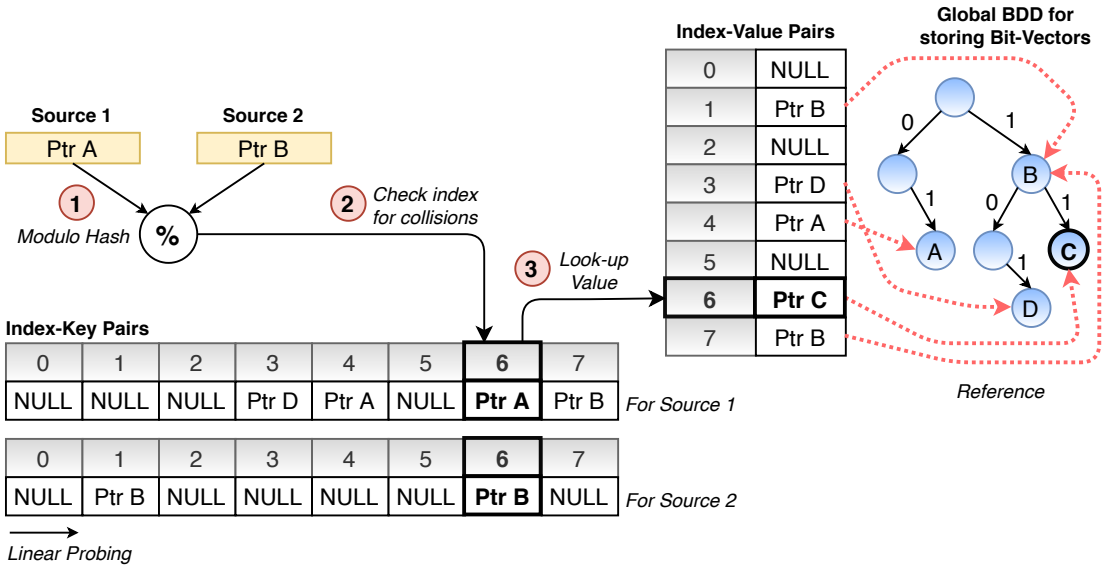


Figure 5.4: Efficient unions of bit vectors via hash lookups

Multi-Tag Propagation

The second propagation policy that we consider relates to multi-tags. This policy is implemented with the use of bit vectors for maintaining tags, and achieves taint merging via union operations. Often, taint-based fuzzers [7, 11] employ multi-tags to track which input bytes influenced memory locations.

As explained in Chapter 4, we concisely represent bit vectors using a global tree and have the 32-bit tags provided by the Taint Rabbit point to nodes in the tree. Figure 5.4 illustrates a design of the propagation. In order to perform a lookup of the union of two nodes, a combined key is first constructed by shifting and adding the two source node pointers ①. A modulo hash is then performed on the combined key to derive the corresponding index. In turn, the index is used to access the two key-store arrays of the map and check whether the retrieved keys match with the two initial source node pointers ②. In the case of a collision, linear probing is done and the index is incremented to check the next adjacent slot. Linear probing is carried out until the keys eventually match or up to an increment threshold. If matching keys are found, the same index is then used to access the value-store array and retrieve the pointer to the union node ③.

We now present vectorized taint primitives for multi-tag propagation. The crux of the primitives is the vectorization of hash lookups which are used to fetch multiple node pointers referring to the unions of source bit vectors simultaneously. Inspired by previous work related to databases and image processing [112, 113], we implement such lookups by using gather SIMD operations. Essentially, this instruction enables

Algorithm 8: Node pointers contained in two source vectors are combined and hashed to derive a vector of starting indexes for accessing the map

Data: Vector: src_vector_1, src_vector_2, $temp\_vector_1$, $combined\_vector$, $mask\_vector$, Integer: $mask$, map_size
Result: Vector: $index\_vector$

```
// Shift node pointers in first source vector by 8 bits to the left.
1 temp_vector1 ← slli_epi32 (src_vector1, 8);
// Combine node source pointers by adding them.
2 combined_vector ← add_epi32(temp_vector1, src_vector2);
// Obtain indexes through a mod operation that is implemented using a bit mask.
3 mask ← map_size - 1;
4 mask_vector ← set_epi32(mask, mask, mask, mask);
5 index_vector ← and_si128(combined_vector, mask_vector);
6 return index_vector;
```

the access of non-contiguous memory indicated by a vector of indexes, and therefore is suitable to load multiple data entries from the map's key or value arrays in parallel.

Vectorizing Taint Merging. We begin by explaining the src_{vector}, $src_{vector} \rightarrow dst_{vector}$ primitive, which takes two source vectors, each containing four 32-bit node pointers, and derives a vector of node pointers referring to union bit vectors. As detailed in Algorithm 8, the primitive begins by first combining the vectors' pointers of each lane and then hashing the result to obtain the starting indexes needed to access the key-store arrays of the map. In particular, the first vector has its pointers shifted by 8 bits to the left (line 1). The results are then added to the pointers in the second vector at line 2, thus forming a vector of combined keys required for the vectorized hash lookup.

To that end, the primitive does a modulo operation on the keys to derive the corresponding starting indexes for linear probing at line 5. The operation is implemented by using a simple *bitwise and* instruction, based on the decremented value of the hash table's size, which is defined to be a power of two.

Next, the primitive accesses the key-store arrays, according to the indexes derived from the modulus operation, and compares gathered keys with those in source vectors. Matching keys imply that index vector also corresponds to node pointers referring to union bit vectors in the value-store array. Crucially, the `vpgatherdd` SIMD instruction is used to load the keys stored non-contiguously. Its support of a mask operand, specifying which lanes to consider when gathering keys, also facilitates the implementation of linear probing for handling collisions. When collisions occur

at particular lanes, a mask is constructed so that re-attempts of the gather operation load adjacent keys only for those lanes. As a result, found indexes are left untouched.

Algorithm 9 describes the lookup of keys in more detail. At lines 5-8, a taint mask is constructed based on the two source vectors via an instruction sequence of `pcmpeqd` and `pxor`. If any of the vectors store an actual node pointer that is not NULL at a lane, then all of the bits of the mask's 32-bit element at that same lane are set to 1s. This mask, denoted as *gather_mask_vector*, is used by the gather operations executed in the first iteration of the linear probe.

At line 10, the gather operation accesses the key store array, denoted by key_base_1, loading the keys specified by *index_vector* to $fetch\_vector_1$ only for those lanes masked by *gather_mask_vector*. For inactive lanes indicated by the mask, the operation instead copies the corresponding elements from src_vector_1. A similar operation is also done to gather keys from the second key-store array at line 11.

The primitive proceeds by comparing the gathered keys with node pointers in the source vectors. It then performs a *bitwise and* operation to combine the results, and constructs an overall mask indicating which lanes have matching keys. If all keys match, then the bits of the mask's element at every lane are set to 1s. Such a case implies that the lookups are all successful. Therefore, to detect this condition, a smaller byte mask is produced via `pmovmskb` and then tested at line 16. The *success* flag is set to true and control breaks out of the loop.

Otherwise, if not all keys match, the primitive continues to try and resolve the collisions. Firstly, the *gather_mask_vector* is updated to stop the gathering of keys at lanes where matches are already successful. This is achieved by simply negating *mask_vector*. Secondly, lines 21-25 are responsible for incrementing indexes so that adjacent keys are considered in the next iteration of the loop. Naturally, these increments are only done at lanes where collisions are present. Consequently, before performing the actual addition at line 24, the primitive clears out the increment value for those lanes by bit masking with *gather_mask_vector*.

When control exists the loop, the *success* flag denotes whether all source keys match with gathered keys. Failure implies that a new node needs to be added to the global tree, along with a respective entry to the map. This is done by executing a special function, which is triggered by a clean call, and not by our vectorized propagation code. However, success means that the returned indexes direct to the node pointers of union bit vectors found in the value-store array.

Specifically, Algorithm 10 details the fetching of the node pointers from the value-store array using the indexes. Once again, a gather SIMD operation is leveraged to load the pointers as denoted at line 6.

Algorithm 9: Matching keys in source vectors with keys gathered from the map's key-store arrays.

Data: Vector: src_vector_1, src_vector_2, $starting\_index\_vector$, $gather\_mask\_vector$, $fetch\_vector_1$, $fetch\_vector_2$, cmp_vector_1, cmp_vector_2, $mask\_vector$, $increment\_vector$, $zero\_vector$, Key-Store Array: key_base_1, key_base_2, Integer: $probe\_length$, $scale$ Byte: $mask$

Result: Vector: $index\_vector$, Bool: $success$

```
1  success ← false;
2  scale ← 4;
3  index_vector ← starting_index_vector;
4  zero_vector ← setzero_si128();
   // Prepare mask indicating tainted lanes.
5  gather_mask_vector ← or_si128 (src_vector1, src_vector2);
6  mask_vector ← cmpeq_epi32(gather_mask, zero_vector);
7  gather_mask_vector ← cmpeq_epi32(gather_mask, gather_mask_vector);
8  gather_mask_vector ← xor_epi32(gather_mask, mask_vector);
   // Begin fetching and matching keys with linear probing.
9  for i ← 0 to probe_length - 1 do
       // Gather keys.
10     fetch_vector1 ← mask_i32gather_epi32 (src_vector1, key_base1,
         index_vector, gather_mask_vector, scale);
11     fetch_vector2 ← mask_i32gather_epi32 (src_vector2, key_base2,
         index_vector, gather_mask_vector, scale);
       // Compare gathered keys with keys in source vectors.
12     cmp_vector1 ← cmpeq_epi32(fetch_vector1, src_vector1);
13     cmp_vector2 ← cmpeq_epi32(fetch_vector2, src_vector2);
       // Create mask indicating which lanes have matched keys.
14     mask_vector ← and_si128( cmp_vector1, cmp_vector2);
15     mask ← movemask_epi8(mask_vector);
       // Check if all keys have been matched.
16     if mask = 0xFFFF then
17         success ← true;
18         break;
19     end
       // For linear probing, negate mask to consider only those lanes where keys have not matched.
20     gather_mask_vector ← cmpeq_epi32( mask_vector, zero_vector);
       // Increment indexes to consider next slot for linear probing.
21     if i ≠ probe_length - 1 then
22         increment_vector ← set_epi32(1, 1, 1, 1);
           // Only increment indexes at lanes where keys have not yet been found.
23         increment_vector ← and_si128(increment_vector,
             gather_mask_vector);
24         index_vector ← add_epi32(index_vector, increment_vector);
25     end
26 end
27 return index_vector, success;
```

Algorithm 10: Fetching values via gather operation

Data: Vector: src_vector_1, src_vector_2, $index\_vector$, $gather\_mask\_vector$, $mask\_vector$, $zero\_vector$, Integer: $scale$
Result: Vector: $value\_vector$

```
// Initialize.
1 scale ← 4;
2 zero_vector ← setzero_si128();
// Prepare gather mask.
3 gather_mask_vector ← or_si128 (src_vector1, src_vector2);
4 mask_vector ← cmpeq_epi32(gather_mask, gather_mask);
5 gather_mask_vector ← xor_epi32(gather_mask, mask_vector);
// Fetch values via gather operation.
6 value_vector ← mask_i32gather_epi32 (zero_vector, value_base,
    index_vector, gather_mask_vector, scale);
7 return value_vector;
```

It is worth mentioning that the vectorization of the $src_{vector} \rightarrow dst_{vector}$ primitive is simpler than $src_{vector}, src_{vector} \rightarrow dst_{vector}$ as it mainly consists of copying tags from the source vector to the destination vector. No union operations are needed. Therefore, we now focus on vectorizing the *meet* primitive.

Vectorizing Meet. A vectorized src_{vector}, $src \rightarrow_M meet$ primitive updates the meet label based on a vector of source tags. For this particular policy, the primitive derives the node pointer representing the union bit vector of all the source tags and the passed meet label. Similar to $src_{vector}, src_{vector} \rightarrow dst_{vector}$, we optimize this primitive by using vectorized table lookups. However, the main difference between the two primitives is that the meet primitive needs to fetch the union bit vector by taking into account all of the pointers present across all lanes in the passed source vector. Therefore, to perform the SIMD operations of the lookup, the tags in the vector need to be separated into two sub vectors. As illustrated in Figure 5.5, the primitive solves this problem by using unpacking instructions.

As specified by Algorithm 11, the primitive first interleaves the lower and upper tags of the source vector with a vector of NULL values using the `punpckldq` and `punpckhdq` instructions respectively. This produces two vectors, namely *low_vector* and *high_vector*, which have source tags present at the first and third lanes. Accordingly, two union node pointers are then fetched by using a vectorized lookup at line 4, where the result is stored in *sub_union_vector*. These union nodes are then merged by essentially repeating the process. The unpacking instructions are leveraged again to produce two vectors, each containing one of the nodes at the

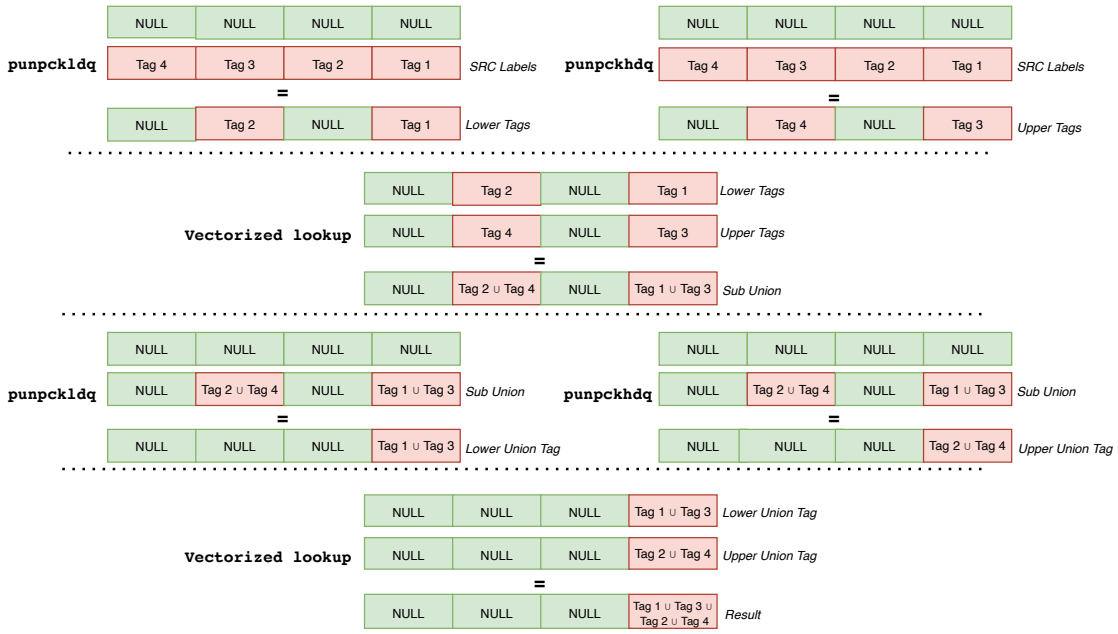


Figure 5.5: The high-level approach of obtaining the union bit vector node pointer of a source vector

first lane and NULL values at the remaining lanes. Their union node is then fetched with another vectorized lookup. Finally, at line 9, the resulting node is merged in a scalar fashion with the current meet label, which is also passed to the primitive, thus completing the primitive's process.

5.4 Support of Vectorized Primitives

The previous section describes the implementation of vectorized taint primitives from the perspective of the user. We now explain the internals of the Taint Rabbit and focus particularly on how instruction handlers leverage vectorized primitives to propagate taint.

5.4.1 Vectorized Instruction Handlers

The Taint Rabbit implements instruction handlers specially designed to harness the benefits of vectorized taint primitives. Like in previous chapters, we demonstrate our approach by considering two instruction handlers. These handlers are designed for instructions with one destination and two source operands, and instructions with one destination and one source operand. Essentially, the functionality of these instruction handlers is equivalent to those that harness scalar primitives. The crucial difference is that vectors are used to propagate multiple labels and accelerate performance.

Algorithm 11: A vectorized meet primitive for the multi-tag policy.

Data: Vector: *src_vector*, *low_vector*, *high_vector*, *sub_union_vector*, *union_vector*, Taint Label: *union_label*
Result: Taint Label: *meet_label*

```
// Initialize.
1  zero_vector ← setzero_si128();
// Get union of upper and lower tags.
2  low_vector ← unpacklo_epi32(src_vector, zero_vector);
3  high_vector ← unpackhi_epi32(src_vector, zero_vector);
4  sub_union_vector ← vectorized_union_lookup(low_vector, high_vector);
// Get union of two resulting sub unions.
5  low_vector ← unpacklo_epi32(sub_union_vector, zero_vector);
6  high_vector ← unpackhi_epi32(sub_union_vector, zero_vector);
7  union_vector ← vectorized_union_lookup(low_vector, high_vector);
// Get union of current meet and source tags in a scalar fashion.
8  union_label ← cvtsi128_si32(union_vector);
9  meet_label ← union_lookup(union_label, meet_label);
10 return meet_label;
```

One destination and two source operands. Algorithm 12 details the handler for an instruction (e.g., `add`) which has one destination and two source operands. It begins by deriving the meet label of each source via separate loops. However, unlike scalar implementations, four labels are considered per iteration, up to the size of the operand. Specifically, at line 4, four 32-bit tags pertaining to the first source are loaded from shadow memory simultaneously into a vector referred to as src_vector_1. Since taint labels are stored consecutively in shadow memory, a single load instruction is sufficient to retrieve the four tags[3]. At line 5, these tags are then passed to the $src_{vector}, src_{vector} \rightarrow dst_{vector}$ primitive in order to update the source's meet label, namely $meet\_label_1$. Between lines 8 and 11, the same process is done to derive the meet label of the second source.

Next, the handler continues by processing the meet labels and associating the resulting labels to destination bytes. This is achieved by first broadcasting the meet labels to all lanes of two vectors, namely $meet\_vector_1$ and $meet\_vector_2$. These uniform vectors are then passed to the vectorized $src_{vector}, src_{vector} \rightarrow dst_{vector}$ primitive at line 17 to perform taint merging and obtain four destination labels stored in *dst_vector*. Finally, the labels are stored to shadow memory, where only a single instruction is needed to perform the store (line 18). This is in contrast to a scalar implementation, which requires a sequence of store instructions.

[3]There are some edge-cases, such as dealing with cross shadow block accesses, which are explained in Section 5.4.2

Algorithm 12: Vectorized propagation for instructions with one destination and two source operands. Operand size is a multiple of four bytes.

Data: Shadow Memory: dst_mem, src_mem_1, src_mem_2, Vector: dst_vector, src_vector_1, src_vector_2, $meet\_vector_1$, $meet\_vector_2$, Label: $meet\_label_1$ $meet\_label_2$, Integer: $opnd\_size$

1 $pointer\_size \leftarrow 4$;
2 $meet\_label_1 \leftarrow NULL$;
3 **for** $i \leftarrow 0$ **to** $opnd\_size$ **by** $pointer\_size$ **do**
    // Load four taint labels of the first source.
4 $src\_vector_1 \leftarrow \text{load\_si128}(src\_mem_1 + (i * pointer\_size))$;
    // Update meet label using primitive.
5 $meet\_label_1 \leftarrow \text{meet}_{primitive}(meet\_label_1, src\_vector_1)$;
6 **end**
7 $meet\_label_2 \leftarrow NULL$;
8 **for** $i \leftarrow 0$ **to** $opnd\_size$ **by** $pointer\_size$ **do**
    // Load taint labels of the second source.
9 $src\_vector_2 \leftarrow \text{load\_si128}(src\_mem_2 + (i * pointer\_size))$;
    // Update meet label using primitive.
10 $meet\_label_2 \leftarrow \text{meet}_{primitive}(meet\_label_2, src\_vector_2)$;
11 **end**
    // Broadcast the first meet label.
12 $meet\_vector_1 \leftarrow \text{cvtsi32\_si128}(meet\_label_1)$;
13 $meet\_vector_1 \leftarrow \text{shuffle\_epi32}(meet\_vector_1, 0)$;
    // Broadcast the second meet label.
14 $meet\_vector_2 \leftarrow \text{cvtsi32\_si128}(meet\_label_2)$;
15 $meet\_vector_2 \leftarrow \text{shuffle\_epi32}(meet\_vector_2, 0)$;
16 **for** $i \leftarrow 0$ **to** $opnd\_size$ **by** $pointer\_size$ **do**
    // Process labels for destination using primitive.
17 $dst\_vector \leftarrow \text{src\_src\_dst}_{primitive}(meet\_vector_1, meet\_vector_2)$;
    // Store resulting taint labels to destination.
18 $\text{store\_si128}(dst\_mem_1 + (i * pointer\_size), dst\_vector)$;
19 **end**

It is important to note that in order to employ a vectorized instruction handler, we require that the operand size is a multiple of the architecture's word size. In our case, we consider the x86 architecture and therefore the pointer size is four bytes (line 1). This ensures that vector registers are made to use their full capacity at each iteration.

One destination and one source operand. Algorithm 13 details the instruction handler for instructions with one destination and one source operand. Similar to the previous instruction handler detailed in Algorithm 12, it considers four labels at a time to obtain the overall meet label (lines 3-6), which is then broadcast to each

Algorithm 13: Vectorized propagation for instructions with one destination and one source operand. Operand size is a multiple of four bytes.

Data: Shadow Memory: *dst_mem*, *src_mem*, Vector: *dst_vector*, *src_vector*, *meet_vector*, Taint Label: *meet_label*, Integer: *opnd_size*, *pointer_size*

1 $pointer\_size \leftarrow 4$;
2 $meet\_label \leftarrow NULL$;
3 **for** $i \leftarrow 0$ **to** *opnd_size* **by** *pointer_size* **do**
    // Load four taint labels of the first source.
4 $src\_vector \leftarrow \text{load\_si128}(src\_mem + (i * pointer\_size))$;
    // Update meet label using primitive.
5 $meet\_label \leftarrow \text{meet}_{primitive}(meet\_label, src\_vector)$;
6 **end**
  // Broadcast the first meet label.
7 $meet\_vector \leftarrow \text{cvtsi32\_si128}(meet\_label)$;
8 $meet\_vector \leftarrow \text{shuffle\_epi32}(meet\_vector, 0)$;
9 **for** $i \leftarrow 0$ **to** *opnd_size* **by** *pointer_size* **do**
    // Process labels for destination using primitive.
10 $dst\_vector \leftarrow \text{src\_dst}_{primitive}(meet\_vector)$;
    // Store resulting taint labels to destination.
11 $\text{store\_si128}(dst\_mem + (i * pointer\_size), dst\_vector)$;
12 **end**

lane of the vector denoted as *meet_vector*. Labels that are needed to be associated to destination bytes are then derived via the $src_{vector} \rightarrow dst_{vector}$ primitive (line 10) and are stored in shadow memory simultaneously (line 11).

Untainting. Untainting is performed when handling an instruction that sets destination bytes to data known always to be safe/trusted. This is often encountered, for instance, due to instructions that move an immediate value to a register, i.e., `mov eax, 5`. The Taint Rabbit also benefits from vectorization to perform fast untainting. In particular, Algorithm 14 describes the vectorization of untainting. Basically, the Taint Rabbit clears a scratch vector register (using `pxor`) at line 2, and stores the resulting NULL values in the register to the destination bytes' shadow memory. A similar approach is adopted by Memcheck [23] to efficiently clear all address tags stored in a range of shadow memory.

5.4.2 Cross-Boundary Accesses to Shadow Memory

In Chapter 2, we describe various schemes for managing shadow memory. One approach is to directly map application memory to supplementary memory via an

Algorithm 14: Untainting multiple destination bytes via vectorization. Operand size is a multiple of four bytes.

```
Data: Shadow Memory: dst_mem Vector: clear_vector, Integer:
      opnd_size, pointer_size
1 pointer_size ← 4;
  // Get a clear vector consisting of NULL values.
2 clear_vector ← setzero_si128();
3 for i ← 0 to opnd_size by pointer_size do
      // Store in shadow memory.
4     store_si128(dst_mem + (i * pointer_size), clear_vector);
5 end
```

established offset. Although the approach leads to fast translations from application to shadow addresses, it typically mandates that all shadow memory is allocated upfront and set to a fixed location. Another more flexible method is to divide shadow memory into manageable blocks where translation is achieved through indirection using a lookup table.

Moreover, a vectorized approach loads multiple taint labels via a single memory access. In particular, four 32-bit pointers are loaded for four application bytes. Since the Taint Rabbit adopts a segmented scheme for managing shadow memory and blocks are bounded in size, unaligned accesses done during taint propagation could lead to surpassing the boundary and partially filling an XMM register with garbage data. Roughly speaking, it resembles the notorious issue of cache line splitting in CPUs. Scalar approaches do not encounter this issue, as a translation to shadow memory is performed per application byte. By contrast, loading multiple labels simultaneously relies on a single translation.

Our approach to detect this problem is illustrated in Figure 5.6. The Taint Rabbit inserts faulty red-zones at the start and end of each shadow memory block. Consequently, a fault occurs when propagation code attempts to access beyond a block due to unaligned accesses ①. This in turn triggers a fault handler which emulates the access to shadow memory, taking into account the two blocks that concern the split ②.

While unaligned accesses are uncommon due to padding added by compilers automatically, we still have to consider such edge-cases [114] to maintain the robustness of the Taint Rabbit. Ultimately, this is just one example that highlights the intricate challenges of engineering a low-level binary taint tracker.

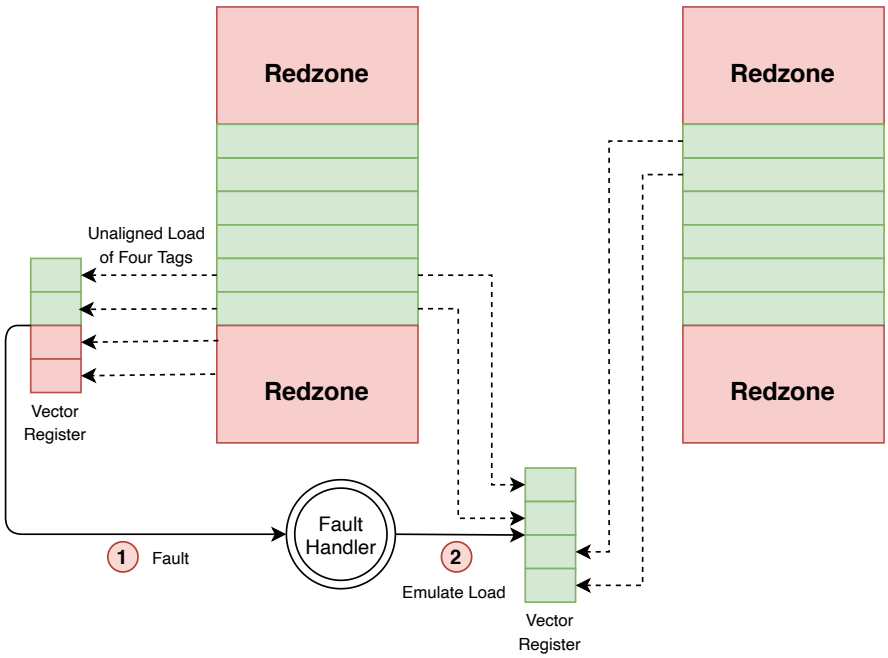


Figure 5.6: Detecting cross-boundary accesses to shadow memory via faulty redzones

5.4.3 Limitations

One limitation is that vectorized taint propagation is not performed for instructions, e.g., `movzx`, that have operand sizes less than the architecture's word size. The reason behind this design choice is primarily to avoid the performance penalty caused by partial register stalls [92], which occur when part of a register, e.g., the lower 64-bits of an XMM register, is modified. Since the number of taint labels associated with operand sizes of two or less is not sufficient to fill vector registers to their full capacity, the Taint Rabbit propagates these taint labels in a scalar fashion.

Other main limitations can be resolved with further engineering. While the Taint Rabbit supports many x86 instructions, it does not yet have taint propagation handlers for some floating-point instructions. Furthermore, our implementation cannot analyse 64-bit target applications. Unfortunately, our focus on 32-bit architectures bring forth performance challenges as they typically have a limited number of SIMD vector registers (just eight on x86), resulting in high register pressure. This limitation could be addressed in future work by employing existing techniques relating to cross-ISA translation [115–117], where 64-bit hardware features are made available to 32-bit applications.

Moreover, our research focuses particularly on Intel x86 and its instruction-set. Consequently, our research outcomes could differ on other architectures, e.g., ARM Neon.

5.4.4 Implementation

We implemented vectorized taint propagation by extending the Taint Rabbit, which in turn builds upon DynamoRIO [32]. While this chapter presents our vectorized algorithms with a notation heavily influenced by Intel's intrinsic instructions, our implementation actually uses the DynamoRIO's low-level API to encode corresponding SIMD instructions. These instructions are based on Intel's AVX and AVX2 extended instruction-set and not on SSE. We avoided SSE instructions to build vectorized propagation code since DynamoRIO internally makes use of AVX instructions, e.g., to implement context-switching, and mixing the use of these extensions leads to overhead penalties by the CPU [92].

Moreover, we improved DynamoRIO's live register analysis (`drreg` [118]) to support efficient spilling and restoring of SIMD registers. This enables our analysis to share SIMD registers with target applications that also use vectorization (and therefore also use the SIMD registers) for their own processing. Other works, such as Minemu, do not perform such spilling and are unable to analyse applications that also perform vectorization. Finally, we enhanced Umbra to support faulty red-zone pages for the detection of shadow access splits.

5.5 Evaluation

Our experimental setup aims to answer whether vectorization improves the performance of generic taint propagation in comparison to scalar implementations. We measure the execution times of the Taint Rabbit while performing vectorized taint propagation on benchmarks used in Chapter 4. We refer to this variant of the Taint Rabbit as `TR-VECT`. Furthermore, acting as our baseline, we consider the Taint Rabbit's call-free taint propagation with SIMD features disabled. Since the baseline is built specifically for our design of generic taint analysis, it is suitable for measuring *directly* the performance benefits of the optimization and reducing threats to the validity of our results. We consider the policies based on ID and multi-tag propagation, as described in Section 5.1.

We run the taint engines on a diverse set of benchmarks, many of which have also been used in previous work [24, 25, 119]. In particular, we consider SPEC CPU 2017, Apache, PHPBench [95], and various other applications related to data compression and image parsing. Like in previous chapters, all of our experiments are conducted on 32-bit Ubuntu 14.04 LTS machines, each installed with an 8 core, 2.60 GHz Intel i7-6700HQ CPU.

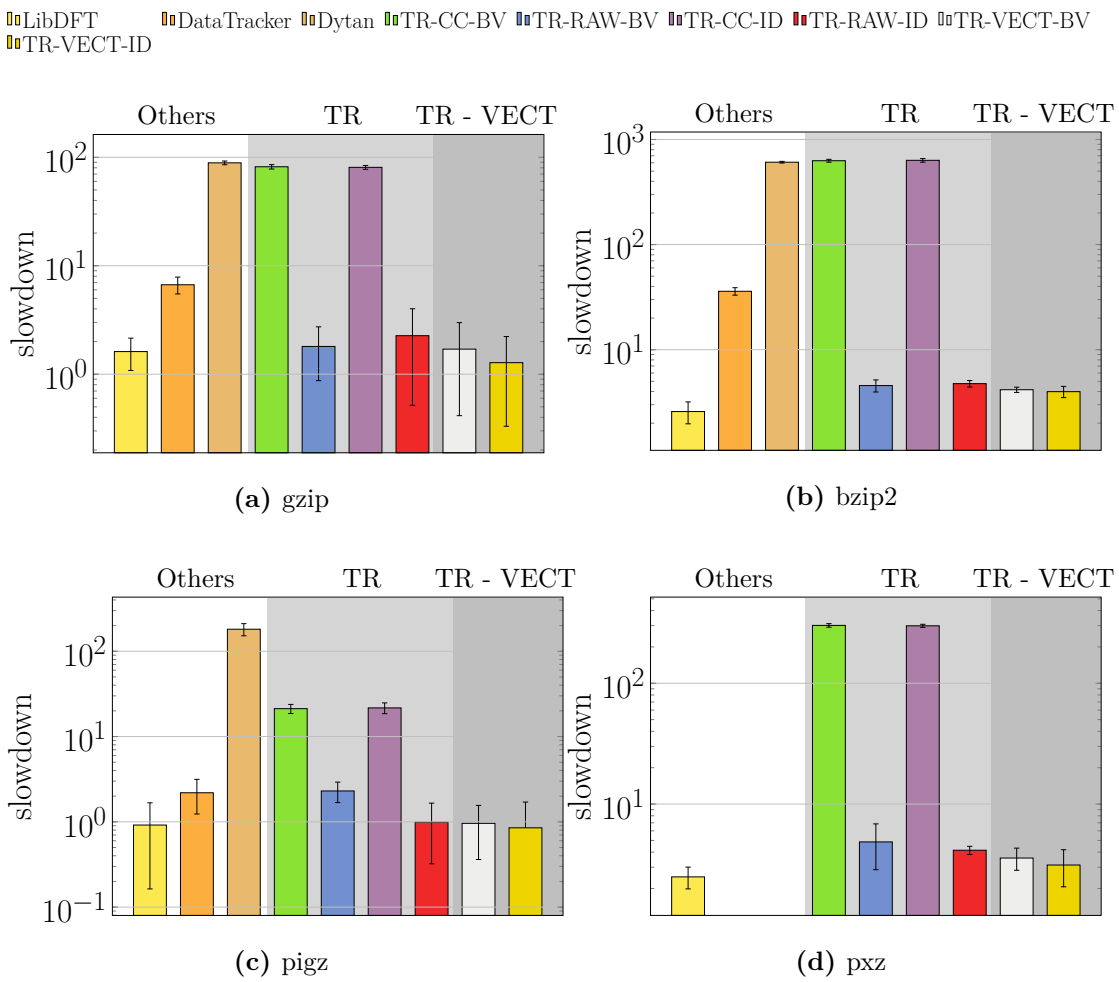


Figure 5.7: Results of vectorized generic taint propagation on data compression benchmarks. Missing entries imply that the corresponding taint engine timed-out or crashed.

5.5.1 Performance

Data Compression. Figure 5.7 illustrates the performance overheads of vectorized taint propagation on data compression tools. We observe that `TR-VECT` is faster than `TR-RAW` on all of the benchmarks. This result is obtained for both ID and multi-tag propagation. `TR-RAW-BV` obtains an average overhead of 3.4x, while `TR-VECT-BV` incurs a better overhead of 2.6x over native execution. When ID propagation is considered, `TR-VECT-ID` achieves an average speed-up of 24% with respect to `TR-RAW-ID`. As expected, `TR-VECT` also outperforms `TR-CC` since the propagation conducted is both call-free and vectorized.

PHP. As shown in Figure 5.8, `TR-VECT` performs faster than `TR-RAW` on PHPBench when performing ID propagation, but is slower when multi-tags are considered.

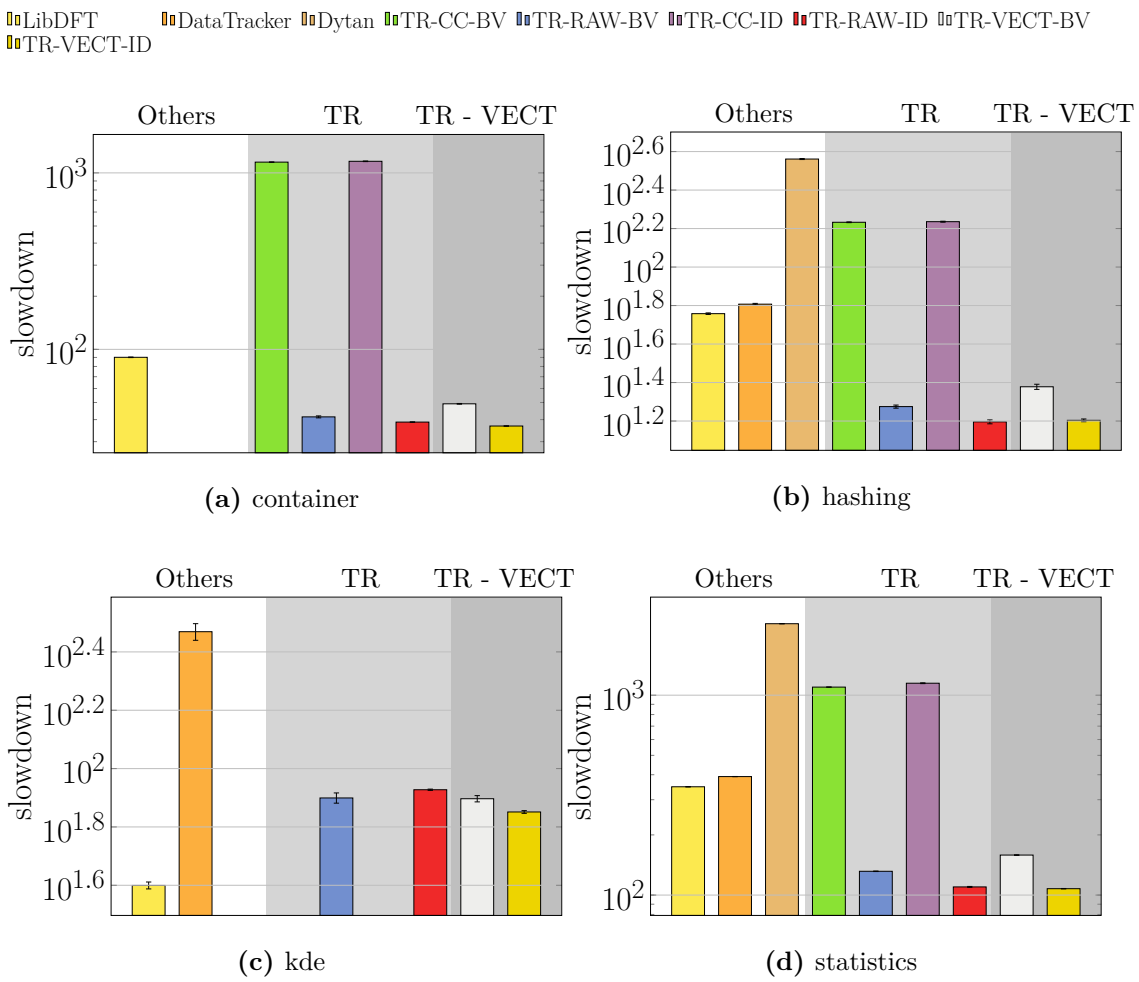
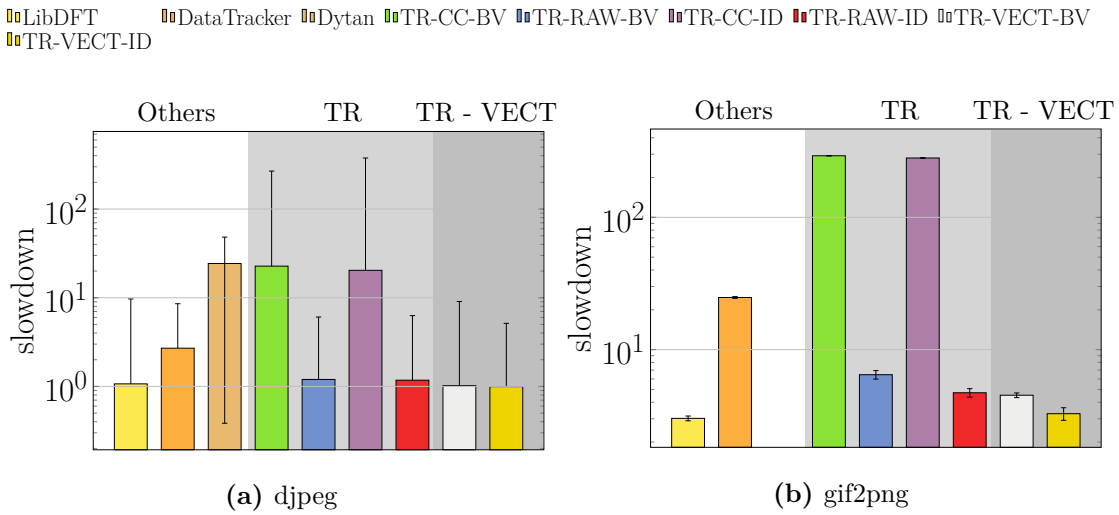


Figure 5.8: Results of vectorized taint propagation on PHPBench. Missing entries imply that the corresponding taint engine timed-out or crashed.

Specifically, `TR-VECT-ID` achieves an average speed-up of 7% over `TR-RAW-ID`, while `TR-VECT-BV` incurs a slowdown of 14%. Out of the four benchmarks, `TR-VECT-BV` only outperforms `TR-RAW-BV` (albeit slightly) on `kde` with overheads of 78.8x and 79.3x over native execution respectively.

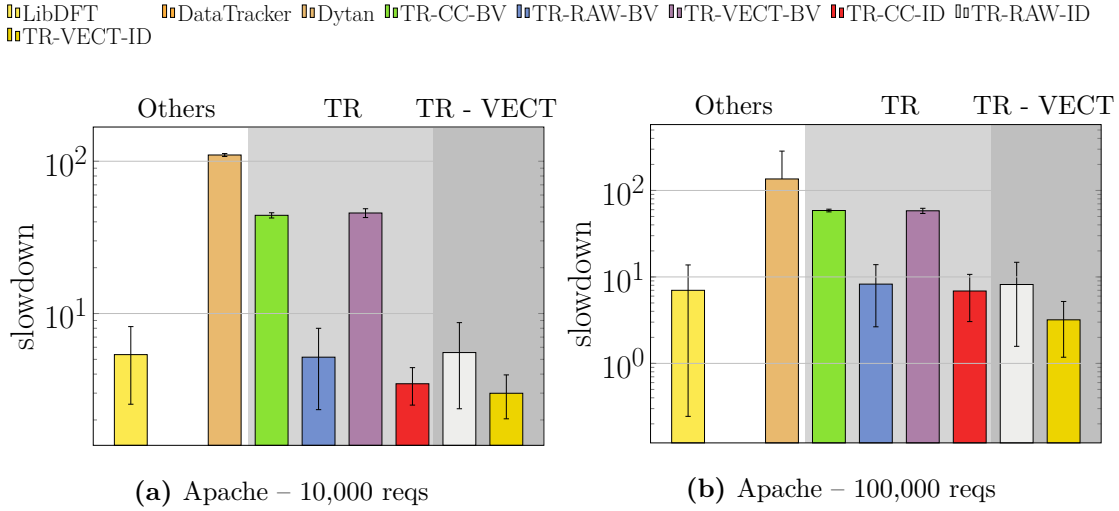
Image Parsing. When employing vectorization, the Taint Rabbit obtains positive results on image parsing tools for both taint policies, as shown in Figure 5.9. Specifically, `TR-VECT-BV` and `TR-VECT-ID` achieve average speed-ups of 27% and 23% respectively with respect their scalar counter-parts.

Apache. The Taint Rabbit obtains positive performance results on Apache for ID propagation, as illustrated in Figure 5.10. When compared to its scalar counter-part, `TR-VECT-ID` achieves an average speed-up of 40%. However, for the multi-tag policy,



(a) djpeg **(b)** gif2png

Figure 5.9: Results of vectorized taint propagation on image parsing applications. Missing entries imply that the corresponding taint engine timed-out or crashed.



(a) Apache – 10,000 reqs **(b)** Apache – 100,000 reqs

Figure 5.10: Results of vectorized taint propagation on Apache. Missing entries imply that the corresponding taint engine timed-out or crashed.

`TR-VECT` only marginally outperforms the scalar implementation in the experiment that sends 100,000 requests, with overheads of 8.3x and 8.2x respectively.

SPEC CPU 2017. Finally, performance results of vectorized taint propagation on the SPEC CPU 2017 benchmarks [120] are shown in Figure 5.11. Our results demonstrate that vectorized taint analysis performs significantly faster than a scalar implementation on CPU intensive benchmarks. On average, `TR-VECT-ID` and `TR-RAW-ID` incur slowdowns of 28.9x and 36.7x respectively. Meanwhile, `TR-`

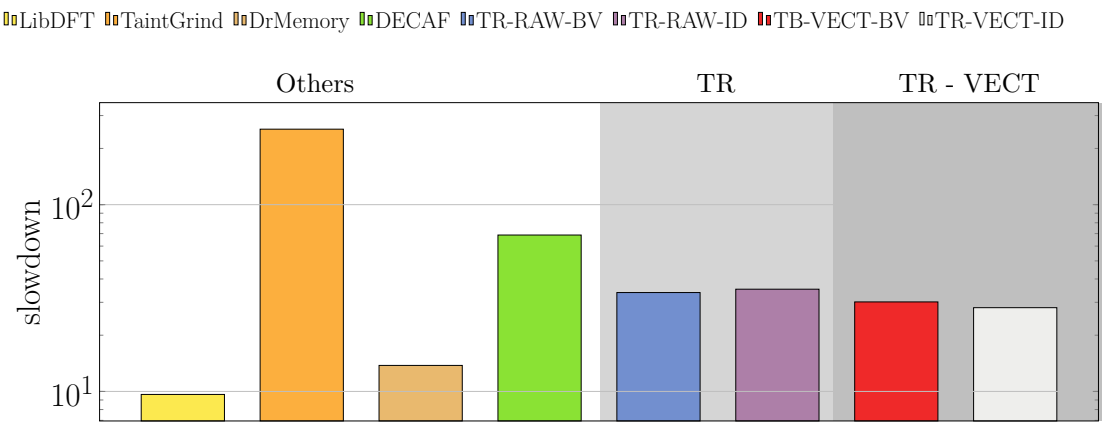


Figure 5.11: Performance results of vectorized taint propagation on SPECrate 2017 (excluding `perlbench`, `gcc` and `x264`)

`VECT-BV` outperforms `TR-RAW-BV` by 11%. However, `TR-VECT-BV` failed to analyse the `perlbench`, `gcc` and `x264` benchmarks due to out-of-memory issues.

5.5.2 Discussion

The scope of our evaluation is to validate the performance of vectorized and generic taint analysis. Across all of our considered benchmarks, the Taint Rabbit performs ID propagation consistently faster when the optimization is enabled. One of our most significant results is on `bzip2`, where a speed-up of 16% is observed.

However, when considering multi-tag propagation, vectorization does not improve performance for certain benchmarks such as the majority of programs presented by PHPBench. Upon further analysis of our results, we note that the overhead of PHP predominantly stems from instrumentation. Our current implementation of vectorized multi-tag propagation uses a significant amount of SIMD registers, where code for their spillage and restoration, driven by live register analysis, is done during the instrumentation of each basic block. This number of registers is typically seven, but can reach eight in the worst case when handling propagation for complex instructions such as `div`. We measured the overhead caused by the spillage, without actually performing taint propagation, on PHPBench in order to further investigate the problem. Indeed, the slowdown amounts to 17.4% with respect to the overhead of spillage required by the scalar approach.

Furthermore, one distinction between `TR-VECT-BV` and `TR-VECT-ID` is that the primitives of the latter work directly on the 32-bit tags in source vectors, without needing to dereference non-contiguous memory via expensive gather instructions. While there is still room for further optimizations, `TR-VECT-BV` still outperforms

its corresponding scalar implementation on other benchmarks, including `kde`, and all considered data compression and image parsing tools.

Overall, our results suggest that the vectorization of taint propagation can improve performance over scalar implementations. The optimization is especially effective when vectorized primitives compute directly the taint tags stored in vectors, avoiding memory accesses as much as possible.

5.6 Summary

Although versatile, generic taint analysis incurs significant slowdowns due to intensive and complex taint propagation. To address this performance problem, this chapter presents vectorized generic taint analysis, which enables users to implement vectorized propagation algorithms so that multiple taint labels are processed simultaneously. Two taint policies are considered to demonstrate how SIMD operations, such as gather instructions, are used to implement vectorized primitives. Results indicate that our optimization lessens overheads from 36.7x down to 28.9x with respect to native execution, on SPEC CPU benchmarks, for ID propagation. While efficient taint analysis remains a difficult open challenge, our research suggests that vectorization is a potential step forward to enhancing its performance, particularly over scalar implementations.

6

Optimizing Generic Taint Analysis with Dynamic Fast Path Generation

Conventional taint engines rely on intensively instrumenting instructions of the target application with analysis code (i.e., instruction handlers) to propagate taint at runtime. Naturally, this leads to a high execution rate of instruction handlers, and a substantial slowdown for generic engines. Existing work [40] have proposed the use of fast paths to address this performance problem, where the execution of unnecessary instruction handlers is safely elided. Specifically, if a basic block's execution is determined not to propagate taint and therefore not modify the current taint state, then an uninstrumented version, with no inserted analysis code, is executed instead to gain a speed-up. However, current solutions only consider this one particular fast path, which is taken when all inputs and outputs of a basic block are not tainted at point of entry. Other possible fast paths that elide propagation for *some* of the application's instructions within a basic block are not leveraged. Instead, if any input or output is found to be tainted, the slow path is used by default, thus missing opportunities for optimization.

In this chapter, we propose a novel adaptive optimization, called *dynamic fast path generation*, that is employed by the Taint Rabbit for generic taint analysis. In addition to the fast path used by the state of the art, the optimization establishes other fast paths in a just-in-time fashion based on frequent *in* and *out* taint states of basic blocks observed at runtime. We consider the same benchmarks as described in Chapter 4, and our evaluation outcomes suggest that fast path generation significantly reduces the overhead for intensive CPU-bound applications. However, our results also indicate that the optimization does not pay off for short

running applications, since not enough execution time is available to leverage constructed fast paths.

6.1 Fast Paths

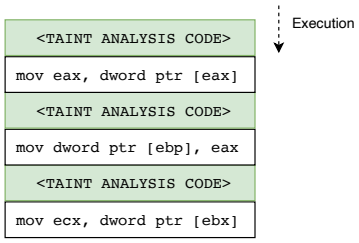
Figure 6.1 (a) illustrates the typical slow path for analysing a basic block. Essentially, each instruction of the basic block is instrumented with taint propagation code. With the consideration of such intensive instrumentation points, the slow path places maximal runtime overhead for taint tracking. Regardless of whether the execution of an instruction does correspondingly modify the taint state of the analysis, propagation is always performed. An instruction handler is still triggered even when all operands of an instruction do not map to tainted data and its execution essentially results in a `no op` to the taint state. Perhaps, it could include taint checks to jump over expensive processing code, such as the invoking of taint primitives, but these checks placed at instruction-level still incur slowdowns. Besides, context-switches to and from taint propagation still need to be performed nonetheless.

Unfortunately, the slow path results in a high execution rate of instruction handlers, causing a significant bottleneck for dynamic taint analysis. Indeed, on a preliminary test run, we measured that at least 73% (over 8 billion) of the instructions executed by `bzip2` conventionally require corresponding propagation code to be executed[1] so that taint is tracked.

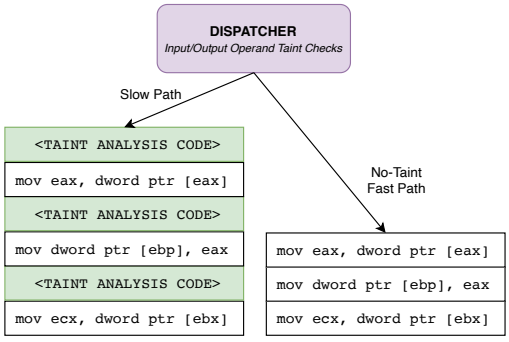
Fast paths aim to reduce overhead by offering other ways to analyse a basic block with different instrumentation. Essentially, a fast path can be seen as a short-circuit for tracking taint where the execution of unnecessary instruction handlers are sliced out. As illustrated in Figure 6.1 (b), existing taint engines, such as Lift [40], generate an alternative version of the same basic block, but leaves the block uninstrumented. As a result, a fast path is established. The uninstrumented basic block is executed when all of its input and output registers/memory are not tainted at runtime. Otherwise, the slow path, implementing full-blown taint analysis, is taken. The fast path can be safely selected since operands do no deal with any tainted data and therefore correctly keeps the taint state unchanged. The dispatcher, invoked at basic block entry, is responsible for performing the necessary dynamic taint checks on *in* and *out* operands and selecting which path to take.

Yet, current solutions do not consider fast paths that lie between the two traditional paths, i.e., the fully-instrumented slow path and the no-taint fast path.

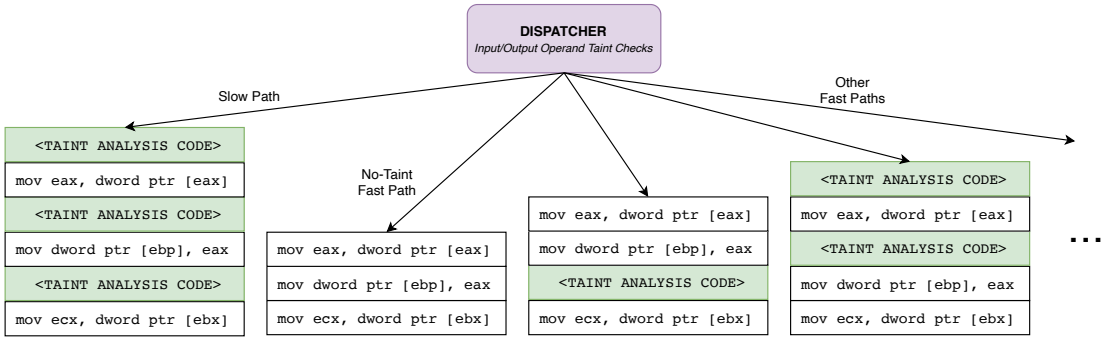
[1] We exclude control instructions such as `jmp` and `cmp` which do not modify the taint state explicitly.



(a) Slow Path - Fully Instrumented



(b) Fast Path - No Instrumentation



(c) Multiple Fast Paths - Different Instrumentation

Figure 6.1: User-defined primitives invoked by the Taint Rabbit to propagate taint

Figure 6.1 (c) shows such fast paths that have taint propagation code elided for *some* of the instructions in the basic block. Since existing approaches simply take the slow path upon any tainted operand and therefore miss opportunities for optimization, we aim to improve the state of the art by leveraging these additional fast paths. However, one of the main challenges concerns the decision of which other fast paths need to considered by the Taint Rabbit. It is impractical in terms of performance to generate all of the possible paths for each basic block via DBI. For instance, there are eight possible paths, each with a different combination of instrumentation points, which would need to be constructed just for the single basic block shown in the figure. Therefore, our solution is for the Taint Rabbit to construct significant fast paths

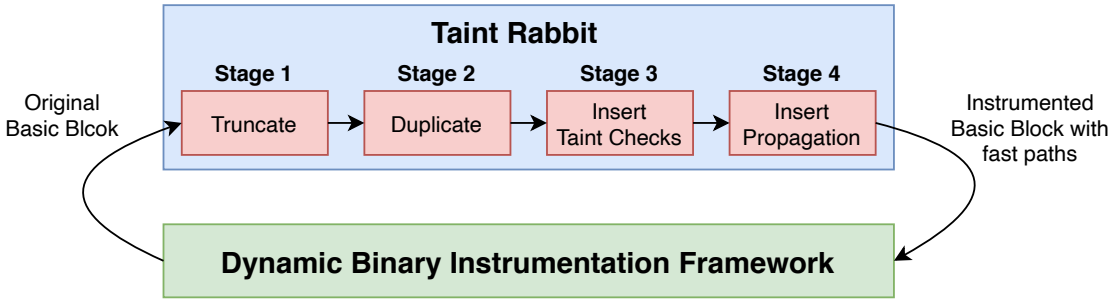


Figure 6.2: Different paths of instrumentation

just-in-time according to runtime behaviour. We refer to this technique as *dynamic fast path generation*, which is investigated thoroughly in the rest of this chapter.

6.2 Dynamic Fast Path Generation

Although the basic variant of using non-instrumented versions of basic blocks as fast paths has been proposed previously and implemented in Lift, the Taint Rabbit can achieve more: it also **dynamically** generates fast paths for the case when taint is present. If the Taint Rabbit encounters a set of tainted inputs and outputs of a basic block frequently executed at runtime, the basic block is duplicated again to create another path and instrumented specifically for the particular case just-in-time. Irrelevant instructions that do not deal with taint in the given case (determined via data-flow analysis) are safely elided from instrumentation. Therefore, fully-instrumented code encompassing the slow path is executed less often than in conventional approaches. Due 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 *in* and *out* taint states. Therefore, the cost of generating fast paths for these states pays off in the long run.

Figure 6.2 illustrates the high-level process of basic block instrumentation to enable fast path generation. At instrumentation-time, the original basic block is provided by the DBI framework to the Taint Rabbit and undergoes various modification and code-insertion stages. Once completed, the basic block is then passed back to the DBI framework for management, such as its encoding to the DBI's code cache. We now detail these stages that enable dynamic fast path generation, using the code example given in Figure 6.3 as an aid for explanation.

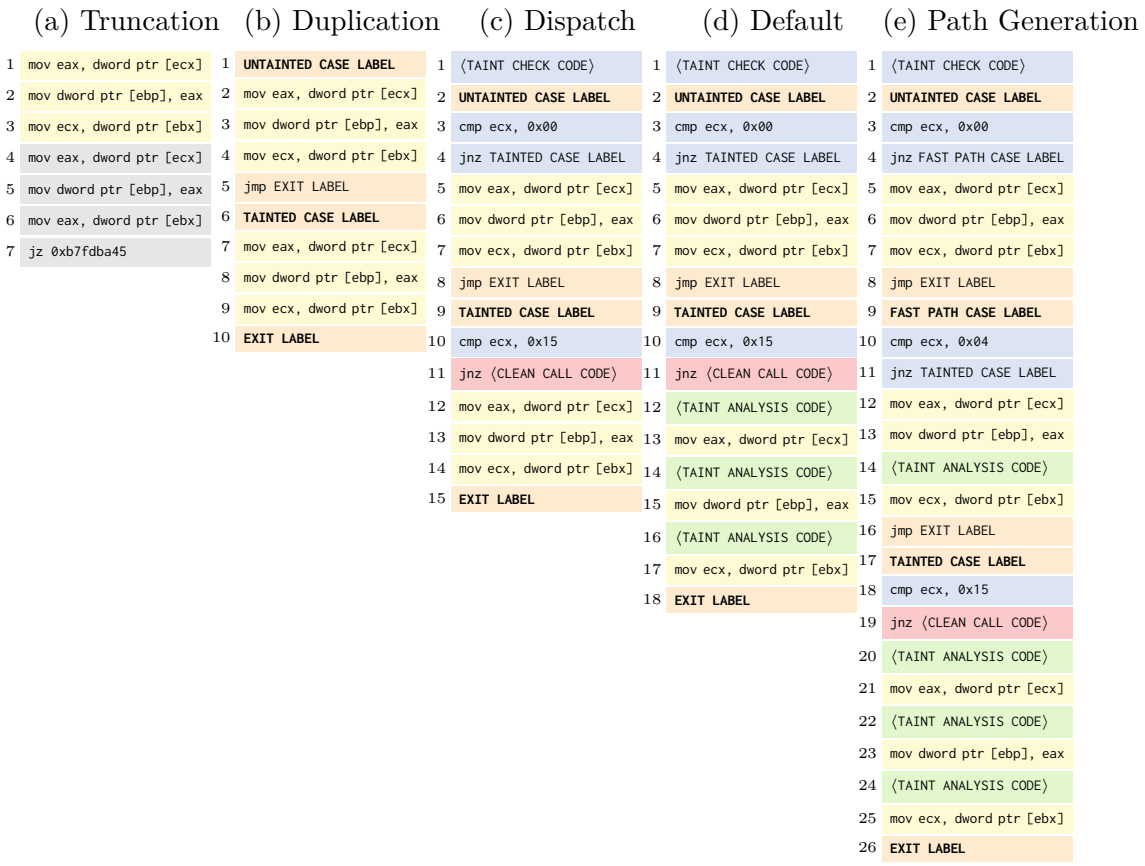


Figure 6.3: A code example showing the instrumentation steps for fast path generation

6.2.1 Truncation

When a new basic block is provided by the DBI platform for instrumentation, the Taint Rabbit begins by identifying any memory reference operands that have addresses indeterminable, due to non-static dependencies, at the start of the basic block during runtime. Such addresses are problematic as their taint statuses cannot be checked by the dispatcher, (which is placed at the beginning of the basic block), prior to entering a fast path. For instance, Figure 6.3 (a) shows a memory access at line 4 where the base register `ecx` has a value that is inconsistent with its start. In particular, the address stored in `ecx` is obtained by a previous instruction, via `dword ptr [ebx]`, found in the middle of the basic block at line 3.

Although code emulation is one way for the dispatcher to compute the future address stored in `ecx`, this approach is expensive and difficult to always perform accurately. Instead, we ensure that all inputs and outputs of a basic block are determinable at the start by simply truncating the block till the first instruction that uses a memory operand relying on a modified register value. Therefore, in our

example, the cut-off code, highlighted in gray, starts at line 4. This code is no longer considered at this stage, but is treated as a new separate basic block. As a result, it undergoes its own analysis for instrumentation, and crucially is now structured such that the memory address in `ecx` is reachable from its start at runtime.

Algorithm 15 describes the process of truncation in more detail. According to use-def chains, we cut the basic block at the instruction that dereferences a memory address generated from a register having its initial (implicit) definition *killed* (i.e., unreachable). The flow-sensitive algorithm iterates over the basic block, instruction by instruction, in a sequential manner (line 3), and checks whether address generation registers (i.e., base and index registers) of memory operands are modified by previous instructions (lines 6 - 12). This is achieved with the use of a register set, denoted as s_r, that keeps track of inconsistent registers. In particular, if a register is written by an instruction, it is inserted to s_r (line 16). The crux of the algorithm is found at line 9, where a shortened basic block, specified as bb_t, is returned immediately upon encountering the use of a problematic memory operand.

After truncation, the set of inputs and outputs is then retrieved by inspecting the remaining instructions found before the cut-off point. As denoted by Algorithm 16, the Taint Rabbit simply iterates over the instructions of the basic block (line 2), and stores all unique source and destination operands in an ordered set *io* (lines 4 and 5). Indeed, for the considered code example (i.e., Figure 6.3), the set of inputs and outputs consists of: `eax`, `ecx`, `dword ptr [ecx]`, `dword ptr [ebx]` and `dword ptr [ebp]`.

We organise *io* according to a pre-defined order, placing register operands before memory references. In turn, registers are ordered based on the following arbitrary sequence: $eax \prec ecx \prec edx \prec ebx \prec esp \prec ebp \prec esi \prec edi$[2]. Moreover, memory operands in the form `[base + disp]` are sequenced according to the base register, followed by their displacement values in ascending order. Such memory operands precede other complex operands that use index registers and absolute addresses. Most importantly, while the order of the set is in itself significantly arbitrary, it is required nonetheless so that the encoding of taint states (which is explained in Section 6.2.3) can be corresponded to the input and output operands contained in the set.

[2]Coincidentally, this order is the same as the `enum` ordering of registers implemented in DynamoRIO.

Algorithm 15: Truncation of basic blocks so that all inputs and outputs of a basic block are determinable at the start

Data: Basic Block bb of size bb_{size}, Integer i, Instruction $instr$, Register Set s_r, Operand $opnd$, Register r
Result: Basic Block bb_t

```
1  s_r ← ∅;
2  bb_t ← [];
   // Iterate over instructions of the basic block.
3  for i ← 0 to bb_size − 1 do
4      instr ← bb[i];
5      foreach opnd ∈ get_opnds(instr) do
6          if is_mem_opnd(opnd) then
               // Consider registers used for generating memory addresses.
7              foreach r ∈ get_addr_gen_regs(opnd) do
                   // Check whether the register's value has been modified.
8                  if r ∈ s_r then
                       // Return truncated block, not including next instructions.
9                      return bb_t;
10                 end
11             end
12         end
13     end
14     foreach opnd ∈ get_dst_opnds(instr) do
           // Consider writes to registers
15         if is_reg_opnd(opnd) then
               // Kill initial definition of register and record modification by updating set.
16             s_r ← s_r ∪ get_reg(opnd);
17         end
18     end
       // Add instruction to truncated basic block.
19     append(bb_t, instr);
20 end
21 return bb_t;
```

6.2.2 Code Duplication

Next, path construction starts with the basic block copied to produce multiple adjacent instances of it. A global map $\mathcal{M}$ associates a basic block ID with meta-data specifying the different paths of instrumentation; ergo, the number of paths determines the total number of basic block instances. By default, this meta-data is initialized with two defined paths. One represents the fully-instrumented slow path, while the other corresponds to the uninstrumented fast path which is taken when none of the basic block's inputs and outputs are tainted at runtime. Consequently, as shown in Figure 6.3 (b), the Taint Rabbit considers two successive instances

Algorithm 16: Gets the set of a basic block's inputs and outputs

Data: Basic Block bb of size bb_{size}, Integer i, Instruction $instr$
Result: Operand Set io

```
   // Initialize Operand Set.
1  io ← ∅;
   // Iterate over instructions of the basic block.
2  for i ← 0 to bb_size − 1 do
3  |   instr ← bb[i];
   |   // Add source operands.
4  |   io ← io ∪ get_src_opnds(instr);
   |   // Add destination operands.
5  |   io ← io ∪ get_dst_opnds(instr);
6  end
7  return io;
```

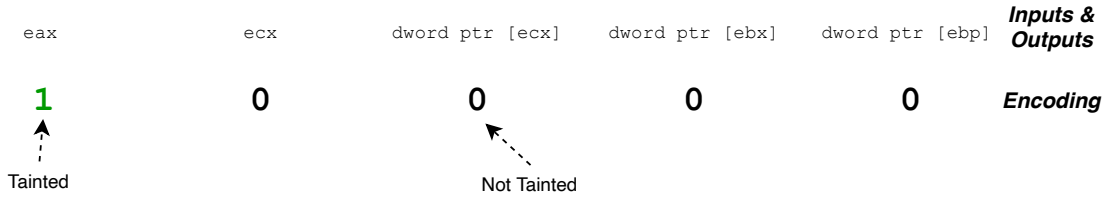


Figure 6.4: An example of an encoding of *in* and *out* taint states

of the same basic block. An entry label is inserted prior to each instance (lines 1 and 6) and, if needed, direct jumps are inserted at the ends to skip over the code of other paths and reach the `EXIT` label (line 5).

To maintain the one-exit-point property of basic blocks, control-flow instructions, such as `call`, in the analysed code of the application are not duplicated for each path. Instead, they are left at the end of basic blocks, after the `EXIT` labels, and are always considered for instrumentation.

6.2.3 Taint Checks and Control Dispatch

The Taint Rabbit proceeds by inserting taint checking code to determine the *in* and *out* runtime taint states of a basic block at point of entry. Naturally, this taint checking code is placed at the start of the basic block, as shown in Figure 6.3 (c) (line 1). The result is encoded as a mask, where each bit indicates whether an input/output is tainted. For instance, the encoding that represents all inputs and outputs as not tainted is essentially a sequence of zero bits. Figure 6.4 illustrates another example, which indicates that the register `eax` is the only tainted input/output.

The simple encoding scheme is described by Algorithm 17. Upon the execution of a basic block, all inputs and outputs, stored in the ordered set *io*, are taint checked. Accordingly, corresponding bits in the encoding e are set to 1 if they are tainted (line 4), and 0 otherwise (line 6).

Algorithm 17: Encoding *in* and *out* taint states

```
   Data: Operand Set io, Operand opnd
   Result: Taint Encoding e
   // Initialize encoding.
1  e ← [];
   // Iterate over inputs and outputs of the basic block.
2  foreach opnd ∈ io do
3      if is_tainted(opnd) then
           // Set bit to indicate taint.
4          append(e, 1);
5      else
           // Do not set bit to indicate no taint.
6          append(e, 0);
7      end
8  end
9  return e;
```

Compare and branch code sequences are inserted to check the encoded mask with the masks of the defined paths (which are stored and retrieved via $\mathcal{M}$) and direct control to appropriate basic block instances. Specifically, as shown in the code example (Figure 6.3 (c)) at lines 3 and 4, if a match fails, execution jumps to the comparison instruction gating the next path, and so on, thus forming a chain for path selection. Conversely, in the case that a match is found, execution does not jump, but instead falls through and enters the path. Moreover, if the encoding fails to match with any of the masks of existing paths at runtime, it implies that a new case is encountered. This is an opportunity for dynamic fast path generation, which is triggered at the end of the selection chain, at line 11 (dynamic fast path generation is explained in Section 6.2.5).

6.2.4 Data-flow Analysis and Instrumentation

The basic block versions are then instrumented with taint propagation code. Essentially, as shown in Figure 6.3 (d), the two default paths are established by fully instrumenting one of the basic block versions and leaving the other without any instrumentation at all.

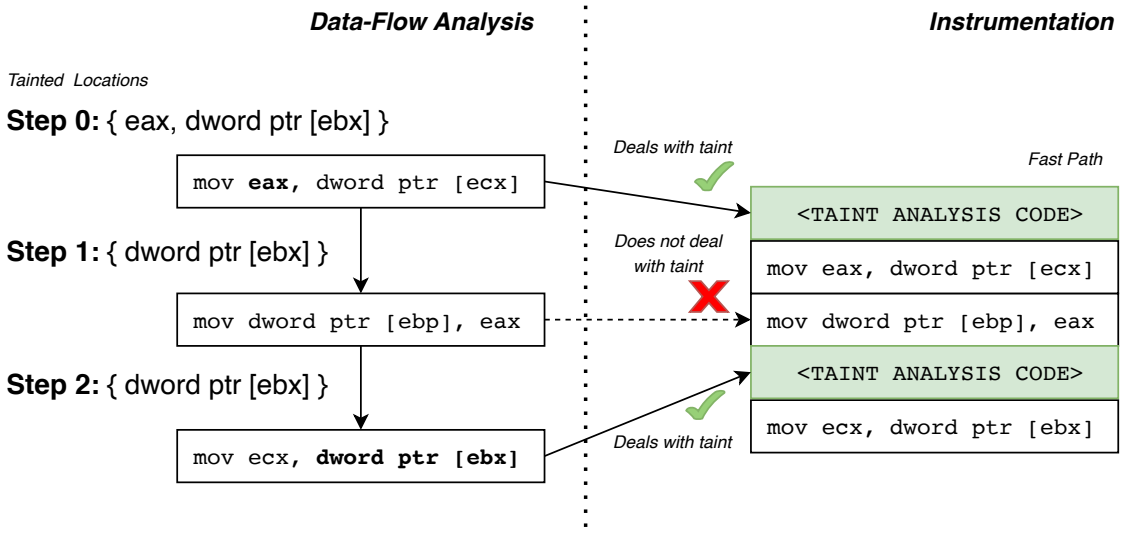


Figure 6.5: Sliced instrumentation driven by data-flow analysis

In order to instrument other prominent paths, forward data-flow analysis is performed on the basic block to determine which instructions deal with tainted operands. Such instructions may propagate taint at runtime and are therefore instrumented, while others have instrumentation elided. Figure 6.5 illustrates the high-level instrumentation process driven by data-flow analysis. Naturally, the initial in-set for data-flow analysis includes the *in* and *out* taint states of the particular path. In our example, `eax` and `dword ptr [ebx]` are the only inputs and outputs known to be tainted at the start of the basic block. Other locations, such as the memory referred by `dword ptr [ecx]`, are not tainted. Since the first instruction writes to `eax`, which is considered tainted, it is instrumented with taint propagation code. After the analysis step of the instruction, `eax` is no longer marked as tainted because at that point it stores untainted data according to data-flow analysis. Consequently, the next instruction does not operate on any tainted data, and therefore its propagation code is elided. However, `dword ptr [ecx]` is still known to be tainted, and therefore the third instruction is instrumented.

Algorithm 18 details further the instrumentation of fast paths. The analysis state is first set up with initial *facts* according to the taint encoding, specifying which inputs/outputs are tainted (lines 1 - 8). If an operand's corresponding encoding bit has a value of 1, then it is considered tainted (done via *mark_as_taint*()).

Next, instructions of the path's basic block instance are considered sequentially. Both the source and destination operands of each instruction are checked to determine whether any of them *may* be tainted (line 12). If this is the case, the instruction is instrumented with propagation code at line 13. Transfer steps

required to conduct flow-sensitive data-flow analysis are performed at line 17 so that the state is updated over-approximately with respect to the instructions. Locations which may be tainted are added to the data-flow set, while others which certainly would not be tainted are removed.

Algorithm 18: Insertion of propagation code based on data-flow analysis.

Data: Basic Block bb of size bb_{size}, Integer i, Instruction $instr$, Operand $opnd$, Data-flow State s, Operand Set io of size io_{size}, Taint Encoding e

```
   // Initialize data-flow state with initial facts.
1  for i ← 0 to io_size − 1 do
2  |  opnd ← io[i];
   |  // Mark input and outputs as tainted according to encoding.
3  |  if e[i] = 1 then
4  |  |  mark_as_taint(s, opnd);
5  |  else
6  |  |  mark_as_untaint(s, opnd);
7  |  end
8  end
   // Iterate over instructions of the basic block.
9  for i ← 0 to bb_size − 1 do
10 |  instr ← bb[i];
   |  // Iterate over operands of the instruction.
11 |  foreach opnd ∈ get_opnds(instr) do
12 |  |  if may_be_tainted(s, opnd) then
   |  |  |  // Instruction may deal with taint at runtime; insert propagation code.
13 |  |  |  insert_propagation_code(instr);
   |  |  |  // Stop operand iteration and move on to next instruction.
14 |  |  |  break;
15 |  |  end
16 |  end
   |  // Update data-flow state with respect to the instruction.
17 |  do_update_step(s, instr);
18 end
```

6.2.5 Generating Additional Fast Paths Just-In-Time

Figure 6.6 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 encountered during runtime ①. The called routine retrieves the encoding of the unhandled path and registers it by updating $\mathcal{M}$ ②. The existing code

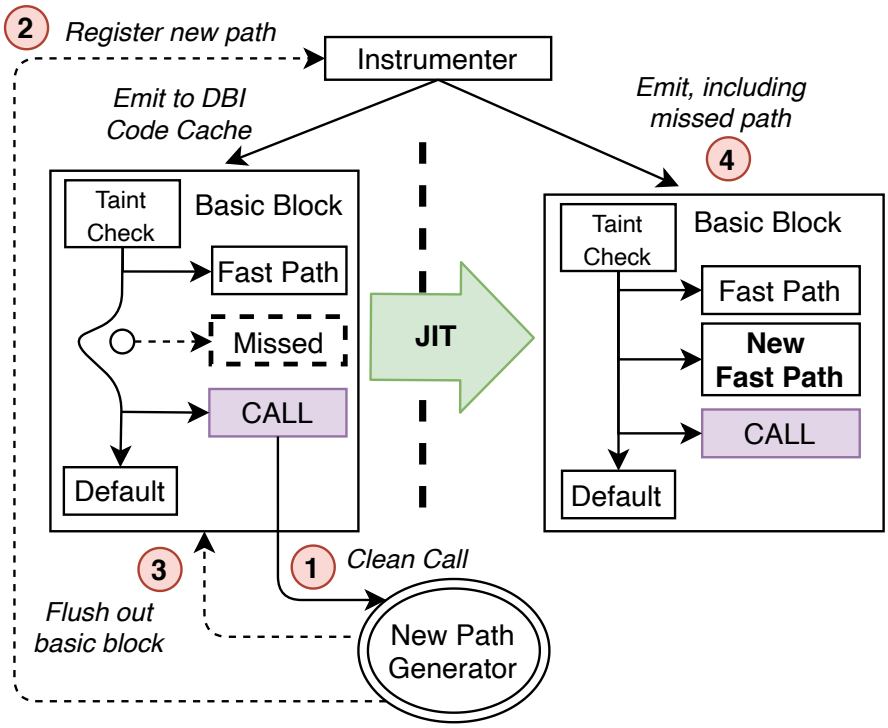


Figure 6.6: Dynamic Fast Path Generation

fragment is then flushed out from the DBI cache ③ [121], and its instrumentation is re-triggered with the inclusion of the missed path ④.

Rather than immediately triggering dynamic fast path generation upon a new set of *in* and *out* taint states, we employ conventional JIT heuristics [122], based on execution count, to reduce the latency induced by flushing. Therefore, a new fast path is generated for a basic block only when a configurable threshold, based on the number of times execution fails to take a fast path, is reached. If fast path generation is not triggered, the unhandled case defaults to the execution of the fully-instrumented slow path.

The Taint Rabbit also has a stopping mechanism that prevents basic blocks from attempting generation if unhandled input/output taint states do not actually result in paths eliding any instructions. The code fragment is still flushed but instead of adding a new path, its re-instrumentation adapts analysis code by removing the monitor and trigger (including the clean call) for fast path generation.

6.3 Limitations

Overhead of Taint Checking. Taint checking is critical with regards to performance, as it is placed in the common path and always executed at the start of

basic blocks. Essentially, the dispatcher *must* direct control fast to appropriate paths. However, determining an input/output's taint status by inspecting all of its pointer-sized tags, one-by-one, is costly because of a large number of comparison instructions as well as cache pollution.

The Taint Rabbit alleviates this limitation 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). Instead of inspecting the tags of each byte, the Taint Rabbit simply tests the status bit. Apart from conducting generic taint analysis, our instrumented paths, not including the no-taint fast path, also maintain and update these status bits. Therefore, a lot of the dispatcher's checking process is shifted down to paths that are less performance critical, away from the uninstrumented fast path. The idea of using over-approximate shared tags is similar to [123], but the Taint Rabbit cleverly uses the `pext` instruction [92, 124] to extract status bits and aid the quick construction of the encoding. Although checks are imprecise, owing to the higher granularity (sub-registers may be seen 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 difference of ∼0.6x over native execution on the SPEC CPU benchmarks. While shared tags are only associated to registers, the Taint Rabbit uses SIMD instructions, e.g., `ptest`, to efficiently test multiple labels simultaneously when taint checking memory.

Moreover, the operands of `rep` instructions, which refer to ranges of bytes, are not taint checked as the process could be expensive for the dispatcher. Therefore, these instructions are treated as potential taint sources for data-flow analysis, and are always instrumented with taint propagation code.

Increase in Basic Blocks due to Truncation. The truncation of basic blocks enables dispatching code to accurately taint check memory referred to by addresses that are unreachable from the start of basic blocks, due to non-static dependencies. While a static whole-program pointer analysis on binary applications is an alternative solution, it yields imprecise results and not as convenient as our approach. However, truncation 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 Taint Rabbit and thus adds overhead.

Increase in instrumentation code due to Inlining. As mentioned in Chapter 4, inlining instrumentation code, which is based on user-defined taint primitives, may result in large code fragments that stresses the CPU's instruction cache and the emitting of code to the DBI cache. This limitation is exacerbated by the instrumentation of duplicated basic blocks required for fast path generation. As a mitigation, the Taint Rabbit outlines instruction handlers to shared code caches by default - inlining can also be used instead at the user's discretion. Note that outlining does not use clean calls but simple trampolines to link with shared code.

Generation of Duplicate Fast Paths. Different encodings, specifying which inputs/outputs of a basic block are tainted, may nevertheless lead data-flow analysis to identify the same instrumentation points for taint propagation, and therefore result in the generation of duplicate fast paths. At the moment, the Taint Rabbit has no general mechanism that allows the mask of a single fast path to match with different encodings despite the path's suitability. While we aim to address this limitation in future work with further engineering, the Taint Rabbit currently only prevents the generation of paths which are equivalent to the slow path, as described in Section 6.2.5.

6.4 Implementation

We implemented dynamic fast path generation into the Taint Rabbit's code-base. The data-flow analysis, used to determine instrumentation points when constructing fast paths, was built from scratch, and roughly consists of 1,500 lines of C code. Moreover, our adaptive optimization uses the flushing capabilities provided by DynamoRIO to flush basic blocks from the code cache. Finally, we also implemented a new DynamoRIO library called `drbbdup`, which duplicates the code of basic blocks. In turn, `drbbdup` is used by the Taint Rabbit to generate fast path. The core functionality of `drbbdup` has been merged into DynamoRIO's repository[3] and is available to end-users in release version 8.0.0.

Software Profiling. Several tools, including `Perf` [125], were leveraged to profile the Taint Rabbit during development and improve the efficiency of generating fast paths at runtime. Figure 6.7 illustrates profiling results of the Taint Rabbit on `perlbench` via a flame graph [126]. Flame graphs are stack-based diagrams designed to visualise frequent code traces where width represents the degree

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

Figure 6.7: Perf [125], along with Flame Graphs [126], aided the profiling of the Taint Rabbit.

of presence in samples. Most of the flames shown in the figure 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 such as generating fast paths (only ∼2% in this recording). Unfortunately, symbols required for generating the flames are not available as this code is JITed to the DBI code cache. However, profiling helped discover a bottleneck related to on-demand creation of shadow memory during development. Moreover, we identified frequent execution of code by mapping instruction addresses reported by `Perf` with basic block information found in DynamoRIO logs and 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. Lastly, profilers built into DynamoRIO itself were also used to garner additional information related to performance.

6.5 Evaluation

Our evaluation aims to answer the following research questions:

- `RQ1`: How much does dynamic fast path generation improve the performance of generic taint analysis?
- `RQ2`: Is there synergy between dynamic fast path generation and other optimizations?
- `RQ3`: With dynamic fast path generation, is the performance of generic taint analysis comparable to the state of the art of bitwise taint analysis?

To answer RQ1, RQ2 and RQ3, our experiments evaluate three additional variants of the Taint Rabbit, referred to as TR-CC-FP and TR-RAW-FP. These variants are similar to those detailed in Chapters 4 and 5, but use fast paths as an optimization. Furthermore, we adopt a similar experimental setup as used in the previous chapters to measure performance, considering again the same set of benchmarks and taint policies, as well as configuring the Taint Rabbit to introduce taint on all read data. While we envisage better performance with less taint introduction, our methodology aims for a grounded evaluation. We aim at measuring the worst performance cases where taking optimal fast paths is difficult due to taint prevalence.

6.5.1 Performance

Data Compression. Our results, given in Figure 6.8, suggest that the use of fast paths enhances the performance of generic taint analysis on data compressors: When compared to TR-RAW-BV, TR-RAW-BV-FP reduces the average overhead from 3.4x down to 2.3x over native execution. The best result is attained on bzip2, where the overhead is brought down from 4.6x to 2.2x. We also observe a positive impact of fast paths with respect to expensive clean call implementations. In particular, TR-CC-BV incurs a 258x slowdown over native execution, while TR-CC-BV-FP performs significantly faster with an overhead of 42.6x. Therefore, fast paths benefit users who prefer to implement taint primitives using a high-level programming language.

Comparing to existing tools, Dytan incurs a 292.7x overhead and is significantly surpassed by TR-CC-BV-FP and TR-RAW-BV-FP. Other generic engines, including DataTracker, are also slower than TR-RAW-BV-FP. For instance, DataTracker incurs an overhead of 6.7x on gzip. By contrast, TR-RAW-BV-FP only results in an overhead of 1.3x. Owing to efficient bitwise tainting, LibDFT is faster than TR-RAW-BV-FP on average. It obtains 1.9x average overhead, as opposed to 2.3x achieved by TR-RAW-BV-FP.

PHP. The performance of fast paths on PHPBench is presented in Figure 6.9. Notably, an improvement is achieved by TR-RAW-BV-FP when compared to TR-CC-BV; the former obtains 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 on all programs, with TR-CC-BV-FP attaining 187.5x on average.

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 amortise many of its initial overheads, such as those posed by the dispatcher and the generation of fast paths pertaining to untaint cases. Since

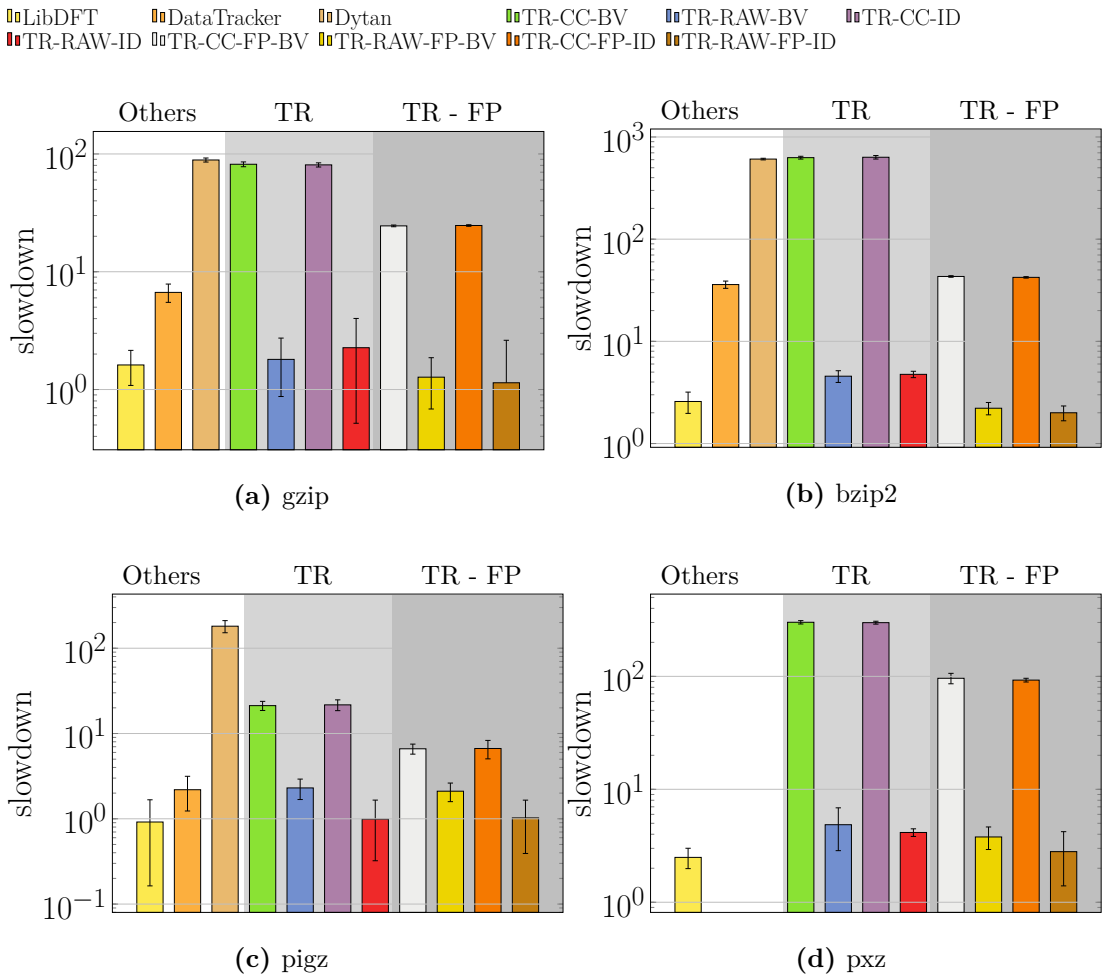


Figure 6.8: Results of fast path generation on data compression benchmarks. Missing entries imply that the corresponding taint engine timed-out or crashed.

the majority of the PHP benchmarks take less than a second to execute natively, the JIT optimizations done by `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 analysers. For example, it is faster than DataTracker, which incurs 250.1x overhead.

Image Parsing. We also evaluate the use of fast paths on image parsing applications. Results are shown in Figure 6.10. With an average overhead of 4.3x, `TR-RAW-BV-FP` achieves better performance than existing taint engines such as DataTracker which incurs 13.7x.

However, we note that fast paths do not lead to a speed-up 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

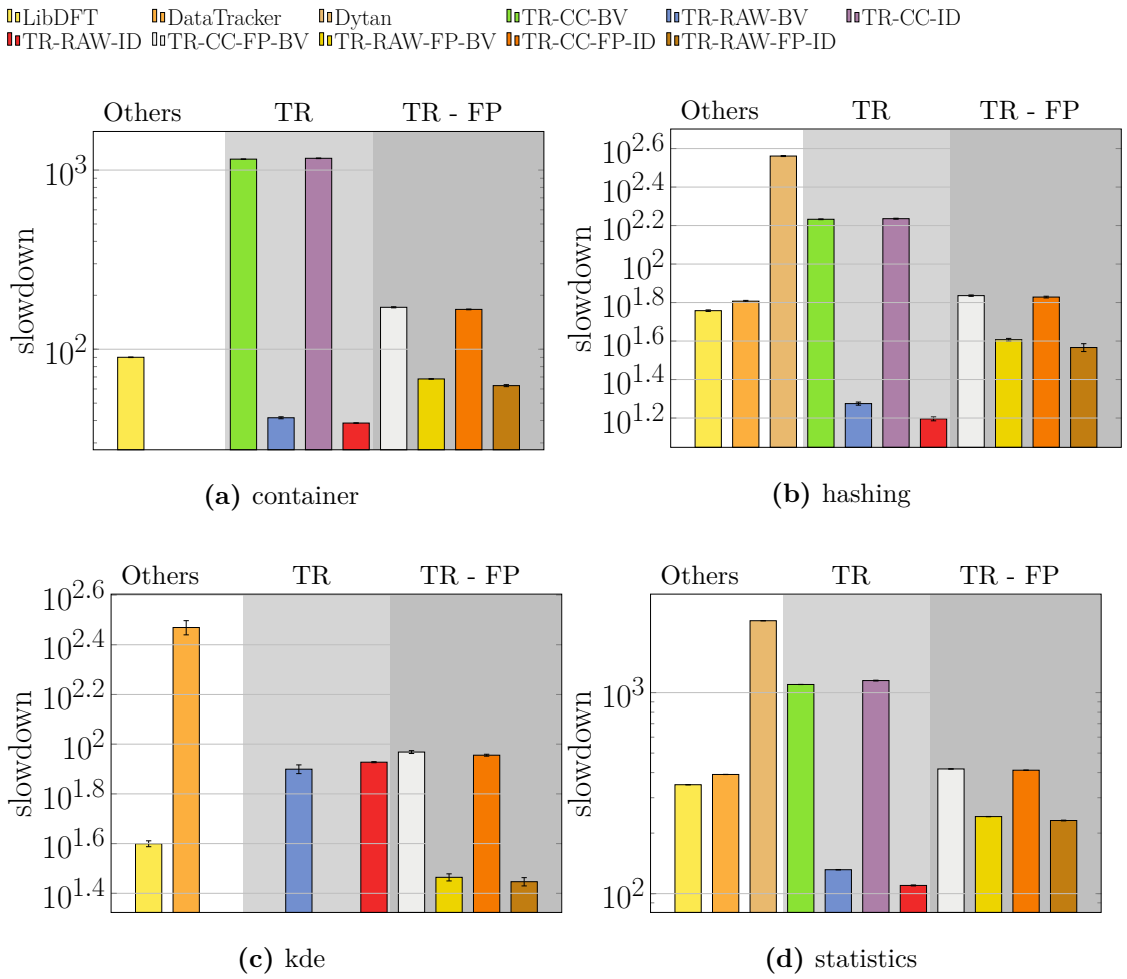
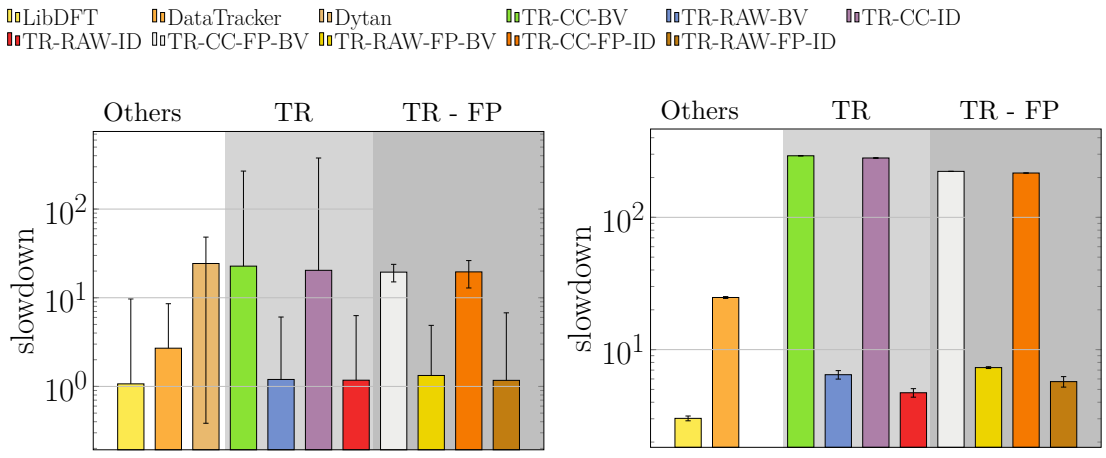


Figure 6.9: Results of fast path generation on PHPBench. Missing entries imply that the corresponding taint engine timed-out or crashed.

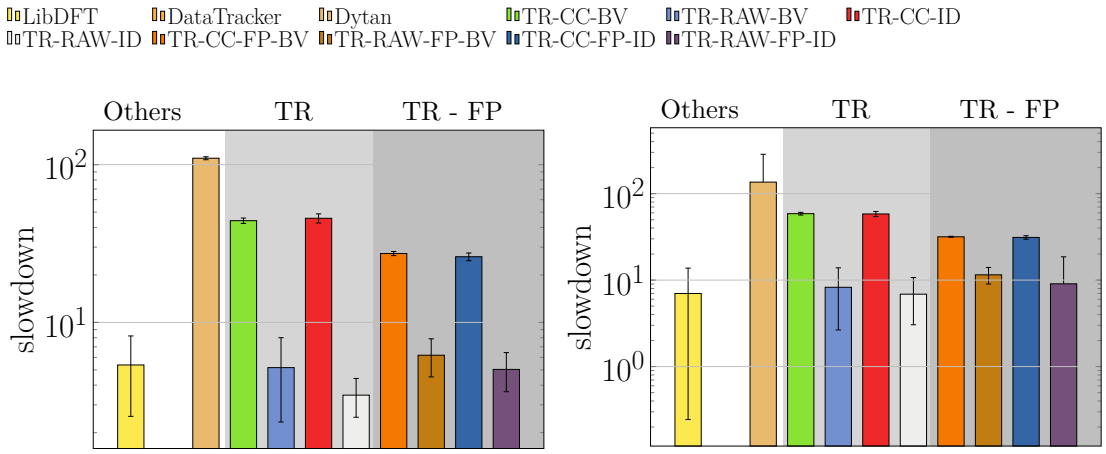
difficult to amortise. For instance, `TR-RAW-BV-FP` results in an average slowdown of 12% when compared to `TR-RAW-BV` as a baseline. Nevertheless, `TR-CC-BV-FP` obtains a lower overhead of 222.6x on `gif2png` than `TR-CC-BV`, which incurs 292x. Since clean call based instruction handlers are expensive, their elision is more effective in improving performance than those implemented in efficient assembly code.

Apache. Figure 6.11 depict our performance results on Apache. The results again show that fast paths speed up taint engines implemented by 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.



(a) djpeg **(b)** gif2png

Figure 6.10: Results of fast path generation on image parsing applications. Missing entries imply that the corresponding taint engine timed-out or crashed.



(a) Apache – 10,000 reqs **(b)** Apache – 100,000 reqs

Figure 6.11: Results of fast path generation on Apache. Missing entries imply that the corresponding taint engine timed-out or crashed.

SPEC CPU 2017. The average overheads observed on the SPEC-rate 2017 Integer benchmark are given in Figure 6.12. `TR-RAW-ID` incurs an overhead of 36.7x over native execution, which is significantly reduced down to 17.9x when fast paths are enabled. We also observe a similar outcome for `TR-RAW-BV-FP`, where a speed-up of 42% is attained when compared to `TR-RAW-BV`. Despite the use of fast paths, the Taint Rabbit 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.

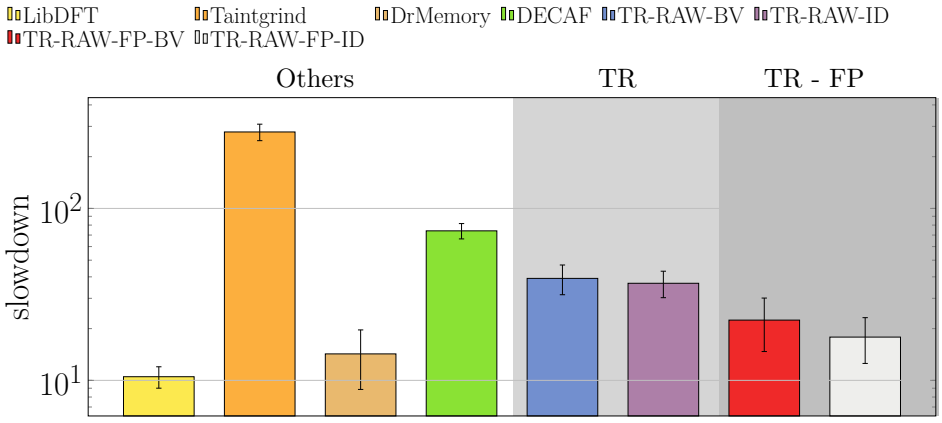


Figure 6.12: Performance results of fast path generation on SPECrate 2017 (excluding `gcc` and `x264`)

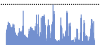
6.5.2 Dynamic Fast Path Generation

Table 6.1 gives insight about the benefit of fast path generation to give an insightful answer to `RQ1`. 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 prevented respectively. The next three columns denote the execution counts of paths with no instrumentation, adaptive instrumentation, and full instrumentation respectively. Finally, the last two columns show timelines when fast paths are generated and executed at 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. Otherwise, performance costs arise due to amortization failure.

Static vs Dynamic Fast Paths. Lift [40] does not generate fast paths just-in-time. It is fixed to only take 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.

Table 6.1: 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		

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 6.5.1. Unfortunately, Lift is not publicly available. Therefore, we modelled similar functionality by modifying the Taint Rabbit and switching off dynamic fast path generation. Average results are given in Figure 6.13. `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 with respect 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 given in Figure 6.14. They indicate that the performance impact of fast paths highly depends on whether the costs of monitor checks and fast path generation are amortised. In particular, on the compute-intensive SPEC CPU 2017 benchmark, the generation of fast paths improves the performance of the Taint Rabbit as the limit increases. However, applications, such as PHP, do not benefit as they incur heavy costs during the instrumentation process.

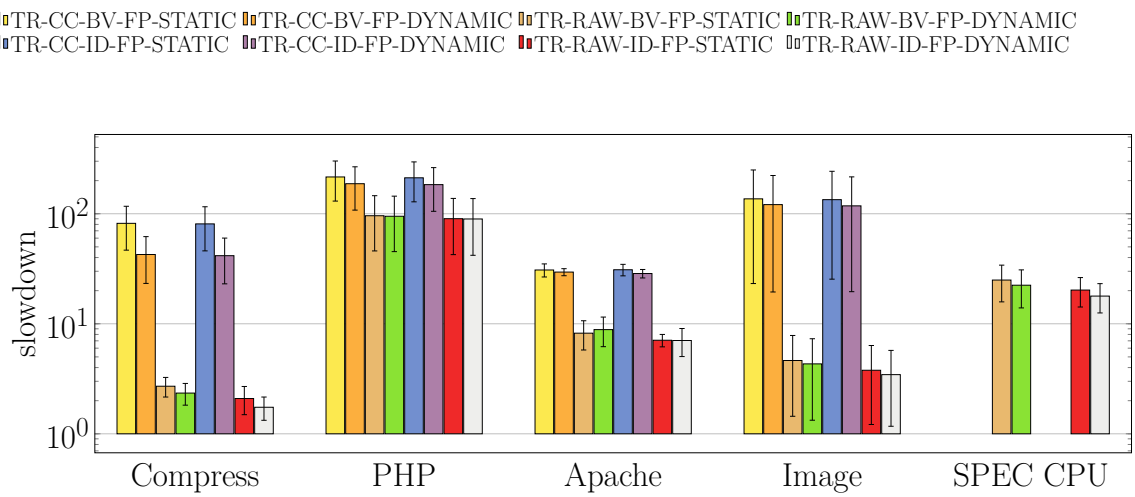


Figure 6.13: Static vs. Dynamic Fast Path Generation

Often, a tipping point is present which if surpassed results in diminishing returns in terms of performance. On PHPBench and benchmarks related to image parsing, this tipping point is reached once the prominent additional fast path is dynamically generated. Consequently, we set the Taint Rabbit to consider the limit of dynamic fast path generation to 1 path by default (but this limit can easily be changed and configured depending on the application under analysis).

Partial Tainting. The Taint Rabbit achieves better performance when less taint is introduced because more optimal fast paths are executed. To validate this claim, we ran the same experiments using `TR-RAW-BV-FP` but randomly sampled taint introduction based on various probabilities. Our results are illustrated in Figure 6.15, where the most indicative outcome is observed on SPEC CPU. Specifically, when the probability tainting read data is set to $\frac{1}{25}$, the average overhead is reduced to 12.5x from 22.4x. Moreover, since most of the overhead is spent on instrumentation costs when analysing PHPBench, sampling taint has negligible effect on performance.

6.5.3 Research Questions

RQ1: *How much does dynamic fast path generation improve the performance of generic taint analysis?*

Generic DBI taint trackers insert expensive taint propagation code per instruction of the target application. This leads to severe overheads at runtime. Our evaluation results suggest that dynamic fast path generation is an effective optimization to address this problem. Fast paths improves the performance of the Taint Rabbit, when using clean calls for taint propagation (i.e., `TR-CC-BV-FP` and `TR-CC-ID-FP`), on all of our benchmarks. For instance, `TR-CC-ID-FP` achieves a high average

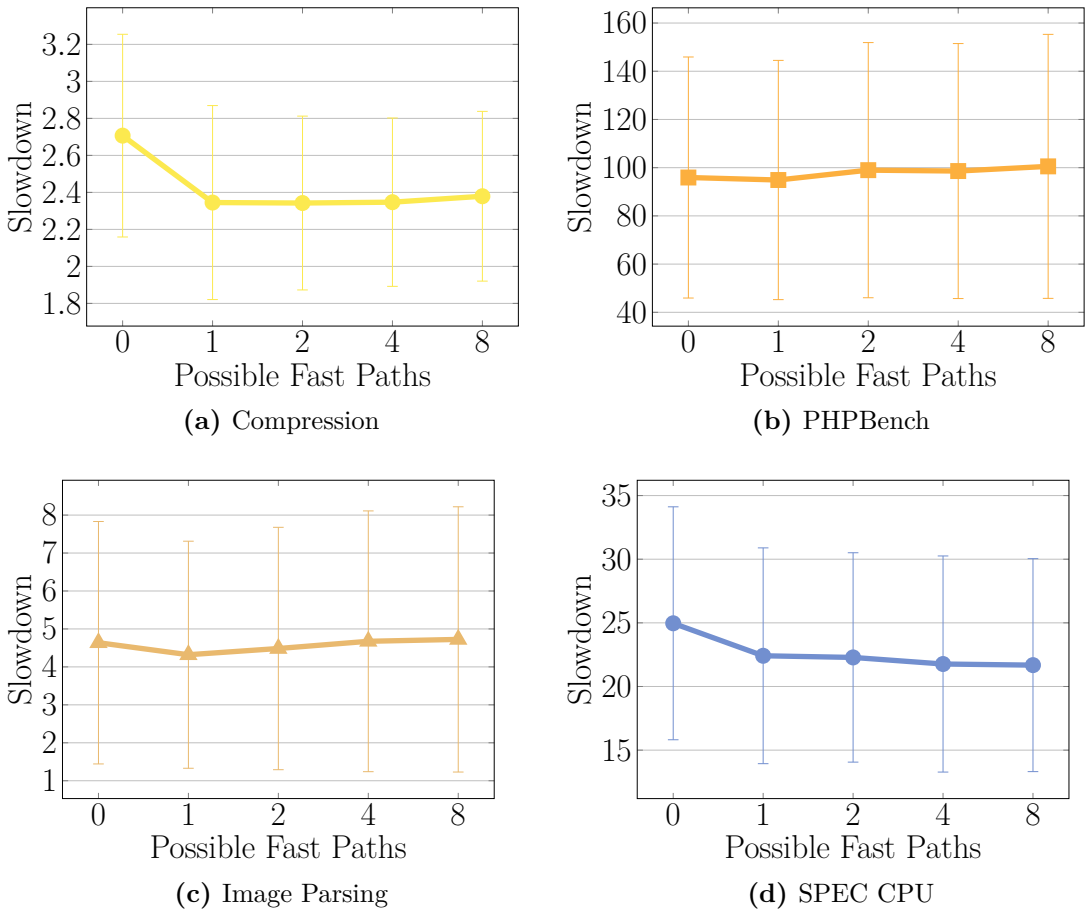


Figure 6.14: Overhead vs. Number of Possible Fast Paths

speed-up of 84% when analysing data compressors. Since clean calls are expensive, their elision through the use of fast paths significantly reduces overhead. Another benefit of fast paths is broad applicability as the technique can be easily adopted with clean-call-based instruction handlers, thus avoiding the need for the user to write low-level assembly code, unlike the other proposed optimizations in this work.

Our optimization also yields faster taint analysis than the traditional approach of fast paths similarly adopted by Lift, where only the uninstrumented no-taint fast path is considered. In particular, on SPEC CPU, `TR-RAW-BV-FP` achieves a speed-up of 10.2% compared to the traditional approach, suggesting that the additional fast paths generated dynamically by our approach contributes to reducing overheads.

RQ2: *Is there synergy between dynamic fast path generation and other optimizations?*

On data compression benchmarks, a positive synergy is observed when dynamic fast path generation is used together with other optimizations, such as call-free

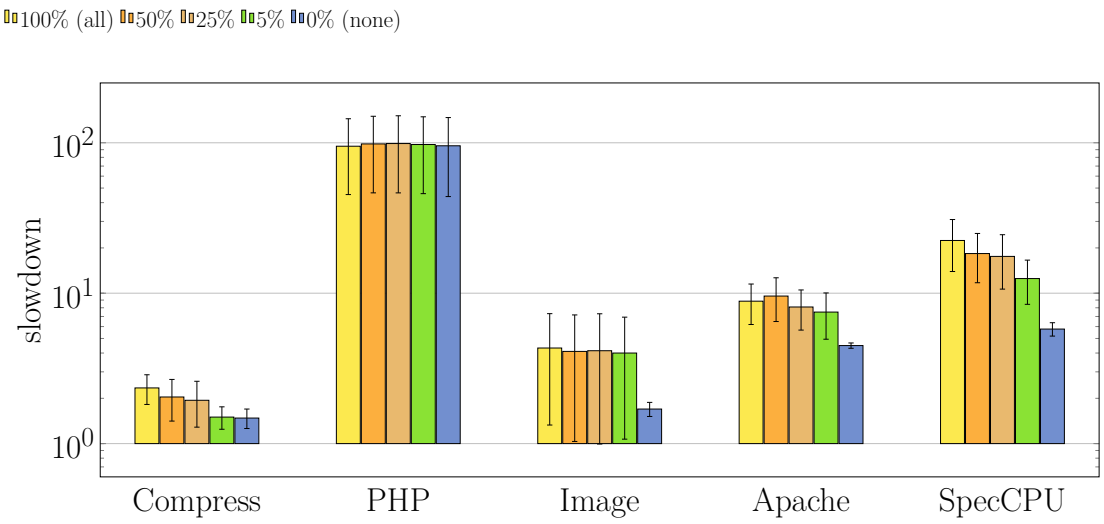


Figure 6.15: Performance results achieved with fast paths when randomly sampling taint introduction

taint propagation. For example, the average overhead attained by `TR-RAW-ID` is 3x, which is reduced further down to 1.7x by `TR-RAW-ID-FP`. Moreover, on SPEC CPU, `TR-RAW-BV-FP` achieves an average speed-up of 42.8% over `TR-RAW-BV`. However, fast paths are mainly effective for long-running CPU-bound applications, where tainting does not comprehensively limit the execution of fast paths. The effectiveness of dynamic fast path generation is also negatively impacted when analysing short-running applications that execute little re-used code as the construction of fast paths fails to pay off.

RQ3: *By enabling the dynamic fast path generation, is the performance of generic taint analysis comparable to the state of the art of bitwise taint analysis?*

On SPEC CPU, `TR-RAW-ID-FP` is slower than LibDFT with overheads of 17.8x and 10.5x respectively. Nevertheless, the Taint Rabbit greatly reduces the performance gap that existed between the two types of analyses. On benchmarks relating to compression and image parsing that Dytan manages to complete, 225.6x overhead is incurred. By contrast, LibDFT incurs 1.5x and the Taint Rabbit is only slightly slower with an overhead of 1.7x.

6.6 Summary

In this chapter, we contribute a new adaptive optimization called dynamic fast path generation, where sliced versions of instrumented basic blocks, acting as fast paths, are constructed just-in-time to reduce the execution of instruction handlers

responsible for performing generic taint propagation. The results presented indicate that fast paths improve performance once generation is amortised. On SPEC CPU 2017, the Taint Rabbit achieves an average speed-up of 42.8% when compared to only taking traditional fully-instrumented slow paths. However, we observe that the optimization struggles to reduce overheads for short running applications as the effort placed to construct fast paths does not pay off. We therefore conclude that the dynamic fast path generation is effective for generic taint analysis when applied particularly on CPU-bound target applications.

7 Literature Review

Dynamic taint analysis [17, 18] is a well-established technique in software security and consequently a resourceful body of knowledge has been developed on the topic over the years. In this chapter, we review related work and highlight the differences between existing taint engines and the Taint Rabbit. While the application of taint analysis, such as for detecting runtime attacks, vulnerability analysis, and information leakage, is extensive in literature, this chapter focuses particularly on the delivery and optimization of the technique.

7.1 Specialized and Generic Taint Analysis

There is a substantial body of work on improving the efficiency of bitwise taint analysis [39–41]. For instance, TaintTrace [39] and LibDFT [24] adopt DBI to inline efficient bitwise propagation with the application's code. Crucially, the simplicity of bitwise propagation enables DBI frameworks to achieve this inlining automatically. Another specialized optimization is adopted by Hayes et al. [127] for conducting GPU taint analysis. The optimization stores 32 taint status flags of 32 registers all within a single register to perform fast taint checks. The end result of such specialized trackers is high performance, but this benefit comes with a cost in terms of versatility. Due to their lack of support of custom taint label structures and propagation logic, they are unable to facilitate the employment of rich policies that go beyond just checking whether a location is tainted. For instance, the optimization proposed by Hayes et al. [127] is unsuitable for generic taint analysis as larger pointer-sized tags need to be tracked per register byte and all these large tags cannot fit in a single register.

While more versatile taint engines, e.g. Dytan [25] and DataTacker [42] have also been presented, they incur staggeringly high overheads. Dytan supports generic taint analysis, enabling the user to define custom merging policies for multi-tags stored in bit vectors. However, its routines require expensive context-switching imposed by transparent calls. A parallel can be drawn between Dytan and TEMU [26], which is a generic taint engine built upon QEMU that also relies on expensive function calls to perform taint tracking. Its successor, DECAF [53], tries to address the performance problem by inserting fast taint status tracking inline to QEMU's Tiny Code Generator (TCG) intermediate language and therefore relates to the approach taken by LibDFT. However, taint labels are maintained at a slower pace asynchronously via tracing. Rather than side-lining the propagation of complex taint labels, the Taint Rabbit optimizes the process with dynamic fast path generation and the support of call-free and vectorized user-defined taint primitives.

Compile-time approaches such as DFSan [30] avoid runtime instrumentation costs but require that the target application's source-code is available. This differs from the Taint Rabbit, which is capable of analysing x86 binaries.

Other works [123, 128] perform preliminary analyses to reduce overheads. Jee et al. [128] avoid instrumentation by means of code abstraction, and TaintEraser [85] takes advantage of taint summaries of standard API functions. Meanwhile, Ruwase et al. [129] take an alternative approach, where higher-level code blocks, particularly hot traces, are considered instead of simple basic blocks in order to make instrumentation code more efficient. We envisage that such optimizations can be adopted to further improve the performance of generic taint analysis, and therefore are considered orthogonal to our work.

7.2 Optimizing Taint Analysis with Fast Paths

Like the Taint Rabbit, other works also leverage fast path optimizations as a means to reduce the execution of taint propagation code. Typically, these optimizations perform coarse-grained taint checks to determine which instrumentation path to take during runtime. Notably, Lift [40] checks whether any input or output of a basic block is tainted, and in cases where no taint is present dispatches control to a non-instrumented version of the basic block. The optimization is also adopted by Davanian et al. [130] but is employed system-wide, and not just for single user space applications. PTT [131] switches between standard virtual emulation and taint tracking by using a modified hypervisor, while LATCH [132] dispatches control to fast paths by efficiently performing the coarse-grained taint checks via hardware.

In contrast to existing techniques, the Taint Rabbit performs dynamic fast path generation. Rather than just taking the non-instrumented fast path, additional paths are constructed just-in-time to enhance performance.

Iodine [133] creates a fast path for taint tracking by optimistically slicing instrumentation based on likely invariants discovered via a prior profiling stage. Crucially, the slicing is done in a way that avoids expensive roll-backs, enabling Iodine to simply switch to the traditional slow path upon failure of the invariants at runtime. Similar to Iodine, the Taint Rabbit uses dynamic information when performing forward data-flow analysis to construct fast paths. However, Iodine does not support binaries. It also depends on a prior profiling stage, whereas the Taint Rabbit employs an adaptive optimization that generates fast paths on the fly.

7.3 Decoupling Taint Analysis

While the Taint Rabbit is an *online* taint tracker, other works [134–137] propose *offline* variants where analysis is decoupled from the application's execution. FlowWalker [136] employs DBI to log traces, and after runtime performs taint analysis. StraightTaint [137] takes a similar approach, but uses an efficient multi-threaded buffer to save data required for constructing the trace. Chabbi et al. [135] investigate taint analysis performed on a secondary shadow thread, which is in sync with the application's thread. Meanwhile, TaintPipe [134] 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.

7.4 Precision of Taint Tracking

Similar to versatility, precision is also a trade-off for better performance, and in the context of pointer tracking, has fostered several discussions [138–140]. The Taint Rabbit works at the byte-level for speed, while Yadegari et al. [51] perform bit-level taint analysis to tackle obfuscation techniques.

We do not envisage the Taint Rabbit to be used for heavy malware analysis as its scope is process-centric and DBI frameworks may easily be escaped[1] [141, 142]. Complex malware often stealthily interact with multiple processes [4], as a means

[1] `https://github.com/lgeek/dynamorio_pin_escape`

of avoiding detection, and therefore tools, such as TaintDroid [49], Decaf [53] and Panda [143], that capture a whole system view for taint analysis are more suitable.

Moreover, Dytan and DTA++ [88] support control-flow based taint analysis to track implicit data flows [144]. In this work, we have not addressed this problem as conservative measures based on over-tainting are known to suffer from inaccuracy posed by the taint explosion problem.

Recently, Chua et al. [145] 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.

7.5 Hardware Solutions to Fast Taint Analysis

Several works also propose hardware-based solutions to achieve efficient taint tracking. However, their dependence on specialized hardware also hinders their wide deployment. Since generic taint analysis aims to facilitate researchers employ various policies for different use-cases, we avoided hard requirements on specialized hardware. For instance, Suh et al. [17] and Minos [97] integrate support for taint analysis in processors, extending registers with a supplementary bit, widening the data bus, and inserting additional caches all for tag management. Similarly, Loki [146] uses special registers and instructions, along with a custom *permission* cache, to enforce tag-based policies. Moreover, the use of a dedicated hardware buffer is proposed by Nagarajan et al. [98] to speed-up communication with a separate CPU core dedicated for taint tracking, while Kannan et al. [99] adopted a decoupled coprocessor to avoid significant changes to the main CPU.

Previous work also presented hardware accelerators such as FlexiTaint [147] and FADE [148], where the latter employs a non-blocking programmable filter to avoid the handling of irrelevant monitor events not relating to taint checks. SIFT [149] performs taint analysis on a separate thread which has its instructions for tracking taint generated during the commit stage of the processor's pipeline. Furthermore, Yoon et al. [150] contributes a hardware-based scheme for conducting predictive taint analysis. Finally, Chen et al. [151] propagate taint by appropriating mechanisms intended for speculative execution. Unlike many of these approaches, the Taint Rabbit avoids dedicated hardware. Instead, it leverages SIMD processing which is widely supported on commodity CPUs.

Perhaps, Minemu [41] relates mostly to our work on vectorized taint analysis. Similar to Taint Rabbit, Minemu uses SIMD registers to carry out propagation. However, it does not conduct register liveness analysis and therefore assumes that

SIMD registers are not used at all by the target application. Minemu is also designed particularly for bitwise tainting and does not consider vectorizing generic taint analysis. This limitation also concerns Raksha [152], which is a customisable data tracking system built on hardware, but it only enables users the option of merging taint flags via `or`/`and` operations. In contrast, our approach allows the user to implement different vectorized propagation algorithms using taint primitives.

7.6 Optimizing Dynamic Binary Instrumentation

DBI is leveraged by many binary taint trackers, including the Taint Rabbit. Several works have focused on optimizing the powerful technique in general. For instance, Kleckner [153] reduces clean calls via partial inlining, while Wang et al. [154] extend the applicability of persistent code caching. Hawkins et al. [155] enhance the speed of DBI for JIT applications by using parallel memory mapping. Such approaches could further improve the performance of the Taint Rabbit.

7.7 Summary

This chapter provides a literature review on optimizations for dynamic taint analysis. It first delves into the different approaches of specialized and generic taint engines and highlights their respective limitations regarding versatility and performance. Notably, our work aims to bridge the gap between these two limitations and deliver optimized generic taint analysis. The chapter also explains existing techniques that take fast paths to reduce the execution of taint propagation code. In this regard, the main contribution of our work is the adaptive optimization that we call dynamic fast path generation. Furthermore, a review is then presented on alternative optimizations based on decoupling taint analysis from runtime execution. However, unlike the online taint analysis performed by the Taint Rabbit, these techniques often face issues on timely runtime detection and therefore in theory limit certain use-cases, such as information leak prevention. In the chapter, we also detail works related to precision of taint tracking, and highlight the trade-offs made to maintain reasonable performance. Finally, the chapter concludes with a brief outlook on general optimizations for DBI, which underpins the implementation of the Taint Rabbit.

8 Conclusion

This dissertation investigates various optimizations for generic taint analysis targeting x86 binaries. While existing online generic taint engines leverage clean calls to benefit from the easy development of taint propagation, they suffer from severe runtime overheads. Our work is the first notable research to demonstrate that *optimized* and *generic* taint analysis is feasible, despite its complexity. Specifically, we claim that the runtime overhead of generic taint analysis can be reduced in comparison to that delivered by the extensive use of clean calls as offered by the current state of the art. A summary of our main results is shown in 8.1. Ultimately, our research led the Taint Rabbit to significantly close the performance gap between existing generic and specialised taint trackers.

As detailed in Chapter 3, our approach to generic taint analysis is based on user-defined taint primitives. Essentially, taint primitives act as the main interface between the user and the taint engine. They are implemented by the user according to the desired taint policy and leveraged by the Taint Rabbit as building blocks to handle taint propagation with respect to the semantics of x86 instructions. Crucially, it is not required that the user delves into specific details regarding the internals of the taint engine or the x86 instruction-set. By adopting a taint-primitive-based approach, the Taint Rabbit supports powerful custom taint propagation logic with suitable performance.

For our first optimization, we investigate thoroughly the overhead of generic taint propagation implemented using clean calls, and evaluate the effectiveness of call-free propagation as a means to improve performance. We build fast taint propagation using hand-crafted assembly code to manually mitigate the use of

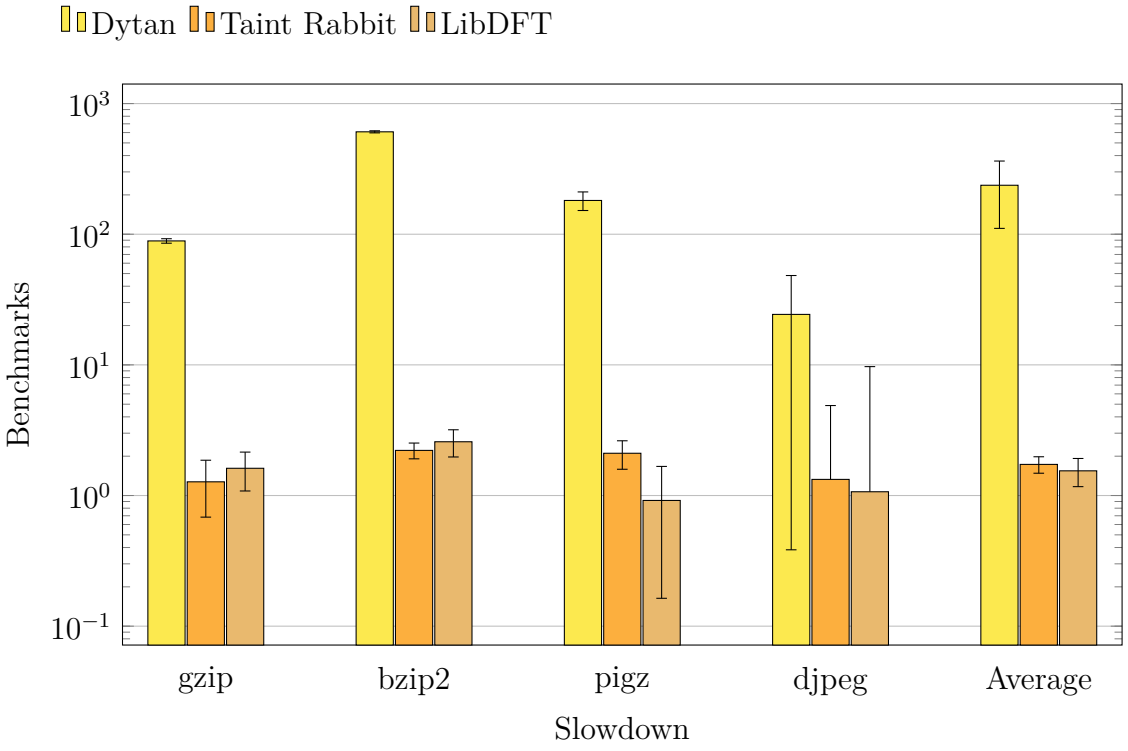


Figure 8.1: Summary of Performance Results for Data Compression and Image Parsing Benchmarks that Dytan manages to Run

clean calls. To measure overhead, a wide variety of real-world software and CPU-intensive benchmarks are considered. Nine existing x86 binary taint systems are also employed as baselines for comparison. Our results, presented in Chapter 4, demonstrate that call-free propagation is a significant optimization and worth the implementation effort, resulting in the Taint Rabbit to become the fastest generic taint engine among those we evaluated.

Chapter 5 delves into the idea of vectorized taint analysis. Essentially, we propose the taint propagation of multiple taint labels, pertaining to operand bytes, to be done in parallel via SIMD processing. This is in contrast to existing slow approaches which process taint associated to bytes, one by one, in a scalar fashion.

Finally, in Chapter 6, we also contribute dynamic fast path generation as an adaptive optimization. Our approach monitors for frequent taint states based on instruction operands at the granularity of basic blocks, and accordingly generates fast paths just-in-time. These fast paths essentially elide instrumentation of propagation code for instructions that do not operate on tainted data specifically for the observed taint state.

8.1 Key Takeaways

Taint-Primitive-Based Approach. Empirical evidence suggests that an interface between the user and the taint engine based on taint primitives is a versatile means to support a variety of taint policies while abstracting away from the internals of the taint system. Notably, results show that a taint-primitive-based approach is suitable to support policies relating to fuzzing, use-after-free vulnerability analysis and runtime attack detection. These policies leverage different taint label structures and taint propagation logic.

Call-free Generic Taint Propagation. Results demonstrate that clean calls are a major bottleneck for generic taint analysis. For instance, when the Taint Rabbit uses clean calls to analyse benchmarks related to compression and image parsing, it incurs an average overhead of 224x over native execution. By contrast, call-free taint propagation achieves a substantially lower overhead of 3.5x. We therefore claim that the effort of optimizing taint primitives to be call-free pays off significantly.

Vectorized Generic Taint Propagation. Our results suggest that vectorization can be an effective optimization to process multiple labels simultaneously when conducting generic taint propagation. We considered two taint propagation policies, namely those based on IDs and multi-tag, and observe speed ups of 24% and 23% in comparison to their respective scalar implementations on compression benchmarks.

Dynamic Fast Paths Generation. Conventionally, taint trackers instrument each instruction of the target application with propagation code even in unnecessary cases that do not result in updating the taint labels of memory locations and registers. To elide instrumentation for such cases, we propose a novel JIT optimization that dynamically generates fast paths for taint propagation. Our results demonstrate that the optimization, built into the Taint Rabbit, performs faster than the conventional fully-instrumented slow path; the Taint Rabbit achieves an average speed-up of 42.8% on SPEC CPU 2017. Although we also observe that dynamic fast path generation may fail to amortize when analysing short running benchmarks, our research suggests that it is highly effective on intensive CPU-bound applications.

Static and Dynamic Analyses. We highlight the usefulness of combining both dynamic and static analyses, particularly when generating fast paths at runtime. The Taint Rabbit identifies which taint states are frequently encountered dynamically and uses this information to drive static data-flow analysis on basic blocks at instrumentation-time. In turn, static data-flow analysis plays an imperative role to slice instrumentation of complex propagation code and reduce runtime overhead.

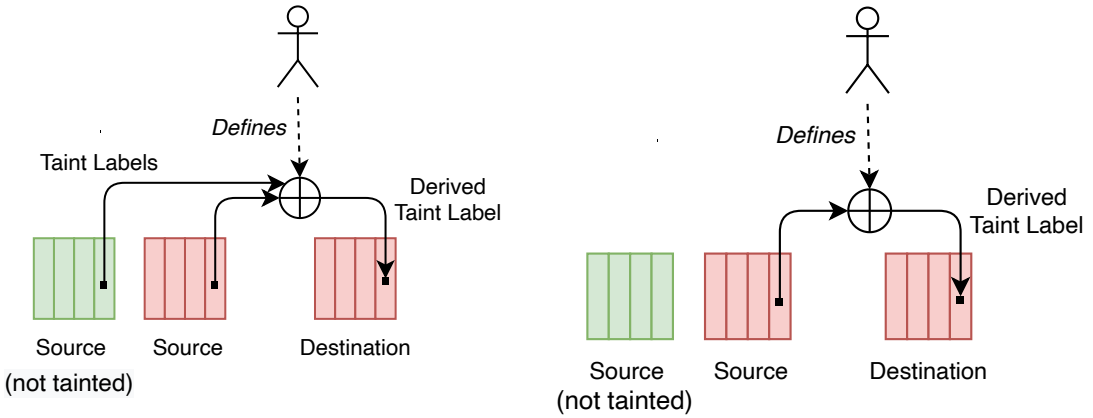
Orthogonality of Optimizations. Finally, another research outcome is the synergy of our optimizations. Our results suggest that the Taint Rabbit performs better when dynamic fast path generation is combined with call-free taint propagation in comparison to deploying these optimizations separately. Notably, our optimizations are orthogonal. While our call-free approach improves the performance of actual propagation code (i.e., instruction handlers), the elision of instrumentation brought by fast paths reduces the number of times such instruction handlers are executed.

8.2 Future Work

Generating fast paths that go beyond elision. The fast paths generated by the Taint Rabbit elide propagation for instructions that do not operate on any tainted data at runtime. In future work, we aim to optimize our fast paths further by not having them only elide the instrumentation of instructions completely. According to the data-flow analysis performed on basic blocks during fast path generation, the idea is to optimize instruction handlers based on which operands are tainted This enables the Taint Rabbit to improve its selection of taint primitives for instruction handling as shown in Figure 8.2. For instance, if only one of the two source operands of an `or` instruction can deal with tainted data, then the generic propagation code only needs to process taint labels of the tainted source operand and safely ignore the other. In this way, the Taint Rabbit uses the $src \rightarrow dst$ primitive instead of the more complex $src, src \rightarrow dst$ primitive that is primarily used to merge taint.

Profile Guided Optimizations. As described in Chapter 6, the Taint Rabbit generates fast paths dynamically for taint cases frequently encountered at runtime. The monitoring done to identify such *hot* cases incurs overhead which the use of fast paths needs to pay off in order for the approach to be effective.

Another method worthy for investigation is to drive fast path generation via profile guided optimization (PGO) [156]. This approach aims to identify which fast paths should be generated for basic blocks according to a preliminary profiling stage



(a) Current Approach - $src, src \rightarrow dst$ **(b)** Optimized Approach - $src \rightarrow dst$

Figure 8.2: Optimizing the use of taint primitives based on which operands are tainted. If only one source of an instruction is tainted, then propagation code uses the $src \rightarrow dst$ primitive instead of the $src, src \rightarrow dst$ primitive.

rather than leveraging solely on runtime monitors. As a result, the overhead incurred by monitors during the main analysis could be reduced or omitted completely.

However, limitations do exist. In particular, a comprehensive set of inputs is needed to conduct profiling. While a test suite of a target application could be suitable, as demonstrated in previous work [133], for profile guided optimization, its availability is not always viable, e.g. when analysing closed-source applications or malware. Future work would delve into measuring whether the gains in speed outweigh meeting this requirement.

Persistent Code Caches. The Taint Rabbit builds upon dynamic binary instrumentation, and naturally this type of instrumentation contributes to runtime overhead. When analysing applications with short running traces covering significantly new code, DBI systems are known to be excessively slow. The JIT compilation process that is integral to DBI is not amortized by the lack of significant code reuse. Indeed, the instrumentation of the Taint Rabbit exacerbates the problem since, compared to other simple DBI tools (e.g., instruction counting), it undergoes more complex processing, including data-flow analysis. Previous work [154, 157] have explored the use of persistent code caches to address this problem. The general idea is that once a basic block is instrumented, it is saved to disk so that it is reused during future analysis, instead of instrumenting the same basic block again. The use of persistent caches by the Taint Rabbit is an interesting research direction in effort to reduce overall overhead.

Shadow Memory. Our results show that the Taint Rabbit has high memory overhead. Recall that in Chapter 4, we observe that the Taint Rabbit ran out of memory when analysing the gcc SPEC CPU benchmark. The root cause of the issue is likely to be scaled-up shadow memory; for every application byte, the Taint Rabbits maps a 32-bit tag. In future work, we aim to address this research challenge by investigating further the idea of associating tags to address ranges, instead of just single bytes [158]. The approach relies on the basic heuristic that contiguous memory or all bytes in a register are usually tagged with the same taint label, and therefore only this label needs to be stored.

However, the challenge is that tagging ranges may not be an effective optimization for certain policies, e.g., ID propagation. In particular, since each tainted byte is associated to a unique identifier, the sharing of tags is not possible. Perhaps, one way of addressing this issue is to store a single primary tag and a conceptual sequence generator that is capable of deriving the n^{th} label from the tag on the fly.

Multi-level JIT Optimization. In order to address the failure of amortizing fast paths for short running applications, one could investigate a multi-level JIT approach. In particular, rather than duplicating and instrumenting each basic block immediately, this process would be shifted to a later stage particularly for hot basic blocks. Further optimization could be broken down to separate JIT stages as well. For instance, rather than immediately inline generic taint propagation code, the Taint Rabbit could start off with the use of clean calls and inline only for hot basic blocks. In this way, the instrumentation costs of encoding long sequences of inlined analysis code are reduced and execution is done as soon as possible for short running applications. Such a multi-staged JIT approach is widely employed in JavaScript engines [37, 159], where the point of contention revolves around deciding whether to interpret or compile code.

Applications of Taint Analysis. Apart from investigating techniques to further scale dynamic taint analysis, future work could also leverage the Taint Rabbit as a framework to advance taint-based applications. Since the Taint Rabbit is a generic tracker, we could explore various research topics in software security such as fuzzing, concolic execution and light malware analysis. Crucially, it would be compelling to investigate the new practical opportunities that are raised by the improvement in performance of taint analysis resulting from this work.

Synthesizing Taint Policies. One key feature of the Taint Rabbit is its support of user-defined custom taint policies. The benefit of this feature is that it removes the requirement from users to extensively modify an existing specialized tracker or building a new one from scratch just to employ their policy. However, while the distribution of the Taint Rabbit is packaged with various implementations of taint propagation policies (e.g., multi-tag and ID propagation), the user still needs to put in effort towards building policies which are not supplied by default. In particular, the user needs to code functionality for taint introduction, taint propagation (i.e., taint primitives) and taint sinks.

One research direction that has the potential to alleviate the burden of this task from the user is the automatic synthesis of taint policies. Inspired by existing work [145], we aim to test the hypothesis whether the synthesis of taint policies, using benign and faulty execution traces, is feasible. Although we understand that program synthesis is an overall difficult and ambitious challenge to solve [160], advancements to the automatic generation of policies could open doors to more usable security.

Appendices

Table 1: Applications of taint analysis, detailing their taint policies and whether they appear to benefit from vectorization.

Application	Goal	Label	Propagation	Vectorizable
Rawat et al. [7]	Fuzzing	Multi-tag	Union	✓
Zhang et al. [161]	Detect Integer Overflow to Buffer Overflow	2-Bit Meta-Data	Include the “May Overflow" flag upon merge	✓
Zhou et al. [106]	Access Control	World ID (Integer)	Assign World ID of script to dynamically modified nodes	✗
Mui et al. [162]	Web Attack Prevention	Status Flag	Union	✓
Caselden et al. [101]	Track flow between buffers	Multi-tag	Status Transfer	✓
Jia et al. [107]	Information Leakage	Three attributes: - Set of secrecy tags - Set of integrity tags - Set of Declassifications and Endorsements	Merge when component invoke other components	✗
Schoepe et al. [108]	Information Leakage	Status Flag	N/A	✗
Yin et al. [163]	Hook Detection	Status Flag	Status Transfer	✓
dAnubis [164]	Malware Analysis	Status Flag	Status Transfer	✓
Slowinska et al. [102]	Code Deobfuscation	Multi-tag	Union	✓
Polino et al. [103]	Analyze API Links	Multi-tag	Union	✓
RAMBO [104]	Optimize Symbolic Execution	Multi-tag	Union	✓
Salwan et al. [165]	Code Deobfuscation	Taint Status Flag	Status Transfer	✓
Minemu [41]	Control-flow Hijacking	Taint Status Flag	Status Transfer	✓
Kawakoya et al. [166]	Malware Analysis	Three types of tags: - API tags - Malware tags - Benign tags.	Status Transfer	✓
Pappas et al. [167]	Taint as Service	32-bit IDs	Generate new ID upon merge	✓
Caballero et al. [168]	Function Extraction	Semantic Tag	Mark as unknown if two different tags merge	✓

Lin et al. [109]	Type Inference	Three components: - A set of memory locations that should have the same types - A set of types associated to the location - A timestamp of the variable's birth	Unifies the labels of the destination and source operands. Tags of variables which should have the same type are also updated.	✗
Chen et al. [105]	Identify protocol fields passed to API functions for fuzzing	Multi-tag	Union	✓
Steffens et al. [110]	Detection of Persistent Client-Side XSS	Taint Status Flag	Status Transfer at JavaScript level	✗

References

[1] *Gartner Says Global IT Spending to Grow 3.7% in 2020*. 2019. URL: https://www.gartner.com/en/newsroom/press-releases/2019-10-23-gartner-says-global-it-spending-to-grow-3point7-percent-in-2020 (visited on 11/22/2020).

[2] *New Orleans Declared A State of Emergency And Takes Down Servers After Cyber Attack*. 2019. URL: https://time.com/5750242/new-orleans-cyber-attack/ (visited on 11/22/2020).

[3] Heng Yin et al. "Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis". In: *Computer and Communications Security*. ACM, 2007, pp. 116–127.

[4] David Korczynski and Heng Yin. "Capturing Malware Propagations with Code Injections and Code-Reuse Attacks". In: *Computer and Communications Security*. ACM. 2017, pp. 1691–1708.

[5] Andreas Moser, Christopher Kruegel, and Engin Kirda. "Exploring multiple execution paths for malware analysis". In: *Symposium on Security and Privacy*. IEEE, 2007, pp. 231–245.

[6] Ulrich Bayer et al. "Scalable, behavior-based malware clustering". In: *Network and Distributed System Security Symposium*. Internet Society, 2009, pp. 8–11.

[7] Sanjay Rawat et al. "VUzzer: Application-aware Evolutionary Fuzzing". In: *Network and Distributed System Security Symposium*. Internet Society. 2017, pp. 1–14.

[8] Istvan Haller et al. "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations". In: *USENIX Security Symposium*. 2013, pp. 49–64.

[9] Sofia Bekrar et al. "A taint based approach for smart fuzzing". In: *Software Testing, Verification and Validation*. IEEE. 2012, pp. 818–825.

[10] Juan Caballero et al. "Undangle: Early Detection of Dangling Pointers in Use-after-free and Double-free Vulnerabilities". In: *International Symposium on Software Testing and Analysis*. ACM. 2012, pp. 133–143.

[11] Peng Chen and Hao Chen. "Angora: Efficient fuzzing by principled search". In: *Symposium on Security and Privacy*. IEEE. 2018, pp. 711–725.

[12] Mateusz Jurczyk. "Detecting Kernel Memory Disclosure with x86 Emulation and Taint Tracking". In: 2018. URL: https://j00ru.vexillium.org/papers/2018/bochspwn_reloaded.pdf.

[13] James Newsome and Dawn Song. "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software". In: *Network and Distributed System Security Symposium*. Internet Society, 2005.

[14] Wei Xu, Sandeep Bhatkar, and Ramachandran Sekar. "Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks". In: *USENIX Security Symposium*. 2006, pp. 121–136.

[15] Dongseok Jang et al. "An Empirical Study of Privacy-Violating Information Flows in JavaScript Web Applications". In: *Computer and Communications Security*. Vol. 10. ACM. 2010, pp. 270–283.

[16] Philipp Vogt et al. "Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis". In: *Network and Distributed System Security Symposium*. Internet Society, 2007.

[17] G. Edward Suh et al. "Secure Program Execution via Dynamic Information Flow Tracking". In: *Architectural Support for Programming Languages and Operating Systems*. ACM, 2004, pp. 85–96.

[18] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. "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. 2010, pp. 317–331.

[19] Sean Heelan. "Automatic generation of control flow hijacking exploits for software vulnerabilities". MA thesis. University of Oxford, 2009.

[20] Vivek Jain et al. "TIFF: Using input type inference to improve fuzzing". In: *Annual Computer Security Applications Conference*. 2018, pp. 505–517.

[21] Tielei Wang et al. "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection". In: *ISecurity and Privacy*. IEEE. 2010, pp. 497–512.

[22] Qin Zhao, Derek Bruening, and Saman Amarasinghe. "Umbra: Efficient and Scalable Memory Shadowing". In: *Code Generation and Optimization*. ACM. 2010, pp. 22–31.

[23] Nicholas Nethercote and Julian Seward. "How to shadow every byte of memory used by a program". In: *Virtual Execution Environments*. ACM. 2007, pp. 65–74.

[24] Vasileios P Kemerlis et al. "LibDFT: Practical Dynamic Data Flow Tracking for Commodity Systems". In: *ACM Sigplan Notices*. Vol. 47. 7. ACM. 2012, pp. 121–132.

[25] James Clause, Wanchun Li, and Alessandro Orso. "Dytan: A Generic Dynamic Taint Analysis Framework". In: *International Symposium on Software Testing and Analysis*. ACM. 2007, pp. 196–206.

[26] Heng Yin and Dawn Song. *TEMU: Binary Code Analysis via Whole-system Layered Annotative Execution*. Tech. rep. UCB/EECS-2010-3. EECS Department, University of California, Berkeley, 2010.

[27] Steven Arzt et al. "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps". In: vol. 49. ACM. 2014, pp. 259–269.

[28] Zhemin Yang and Min Yang. "Leakminer: Detect information leakage on Android with static taint analysis". In: *Third World Congress on Software Engineering*. IEEE. 2012, pp. 101–104.

[29] Xinran Wang et al. "Still: Exploit code detection via static taint and initialization analyses". In: *Annual Computer Security Applications Conference*. IEEE. 2008, pp. 289–298.

[30] The Clang Team. *DataFlowSanitizer Design Document*. URL: `https://clang.llvm.org/docs/DataFlowSanitizerDesign.html` (visited on 11/22/2020).

[31] Gogul Balakrishnan and Thomas Reps. "WYSINWYX: What You See Is Not What You eXecute". In: *ACM Transactions on Programming Languages and Systems* 32.6 (2010), pp. 1–84.

[32] Derek Bruening and Saman Amarasinghe. "Efficient, transparent, and comprehensive runtime code manipulation". PhD thesis. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2004.

[33] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *ACM Sigplan Notices*. ACM. 2007, pp. 89–100.

[34] Chi-Keung Luk et al. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: *ACM Sigplan Notices*. Vol. 40. 6. ACM. 2005, pp. 190–200.

[35] Manish Prasad and Tzi-cker Chiueh. "A Binary Rewriting Defense Against Stack based Buffer Overflow Attacks." In: *USENIX Annual Technical Conference*. 2003, pp. 211–224.

[36] Sushant Dinesh et al. "Retrowrite: Statically instrumenting COTS binaries for fuzzing and sanitization". In: *IEEE Symposium on Security and Privacy*. IEEE. 2020, pp. 1497–1511.

[37] Filip Pizlo. *Introducing the WebKit FTL JIT*. 2014. URL: `https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/`.

[38] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. "Disassembly of executable code revisited". In: *Working Conference on Reverse Engineering*. IEEE. 2002, pp. 45–54.

[39] Winnie Cheng et al. "TaintTrace: Efficient flow tracing with dynamic binary rewriting". In: *Computers and Communications,* IEEE. 2006, pp. 749–754.

[40] Feng Qin et al. "Lift: A Low-overhead Practical Information Flow Tracking System for Detecting Security Attacks". In: *Microarchitecture*. IEEE. 2006, pp. 135–148.

[41] Erik Bosman, Asia Slowinska, and Herbert Bos. "Minemu: The World's Fastest Taint Tracker". In: *International Workshop on Recent Advances in Intrusion Detection*. Springer LNCS. 2011, pp. 1–20.

[42] Manolis Stamatogiannakis, Paul Groth, and Herbert Bos. "Looking Inside the Black-box: Capturing Data Provenance using Dynamic Instrumentation". In: *International Provenance and Annotation Workshop*. Springer LNCS. 2014, pp. 155–167.

[43] Derek Bruening, Qin Zhao, and Saman Amarasinghe. "Transparent Dynamic Instrumentation". In: *Conference on Virtual Execution Environments*. 2012, pp. 133–144.

[44] DynamoRIO. *Documentation: Clean Calls*. 2017. URL: http://dynamorio.org/docs/API_BT.html#sec_clean_call.

[45] Derek Bruening and Qin Zhao. "Practical Memory Checking with Dr. Memory". In: *Code Generation and Optimization*. IEEE. 2011, pp. 213–223.

[46] *Guide to x86 Assembly*. URL: https://www.cs.virginia.edu/~evans/cs216/guides/x86.html (visited on 11/22/2020).

[47] Sean Heelan and Agustin Gianni. "Augmenting vulnerability analysis of binary code". In: *Annual Computer Security Applications Conference*. 2012, pp. 199–208.

[48] Hyung Chan Kim et al. "Capturing information flow with concatenated dynamic taint analysis". In: *Availability, Reliability and Security*. IEEE. 2009, pp. 355–362.

[49] William Enck et al. "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones". In: *Transactions on Computer Systems* (2014), p. 5.

[50] Jim Chow et al. "Understanding data lifetime via whole system simulation". In: *USENIX Security Symposium*. USENIX, 2004, pp. 321–336.

[51] Babak Yadegari and Saumya Debray. "Bit-level taint analysis". In: *Source Code Analysis and Manipulation*. IEEE. 2014, pp. 255–264.

[52] Lok Kwong Yan and Heng Yin. "SoK: On the Soundness and Precision of Dynamic Taint Analysis". In: URL: http://www.cs.ucr.edu/~heng/teaching/cs260-winter2017/formaltaint.pdf.

[53] Andrew Henderson et al. "DECAF: A Platform-Neutral Whole-System Dynamic Binary Analysis Platform". In: *Transactions on Software Engineering* 43 (2 2017), pp. 164–184.

[54] Qin Zhao, Derek Bruening, and Saman Amarasinghe. "Efficient memory shadowing for 64-bit architectures". In: *Proceedings of the 2010 international symposium on Memory management*. 2010, pp. 93–102.

[55] Julian Seward and Nicholas Nethercote. "Using Valgrind to Detect Undefined Value Errors with Bit-Precision". In: USENIX Association. 2005, pp. 17–30.

[56] Laszlo Szekeres et al. "SoK: Eternal war in memory". In: *2013 IEEE Symposium on Security and Privacy*. IEEE. 2013, pp. 48–62.

[57] Yan Shoshitaishvili et al. "SoK:(state of) the art of war: Offensive techniques in binary analysis". In: *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2016, pp. 138–157.

[58] One Aleph. "Smashing the stack for fun and profit". In: (1996).

[59] Mark Daniel, Jake Honoroff, and Charlie Miller. "Engineering Heap Overflow Exploits with JavaScript." In: (2008).

[60] Shuo Chen et al. "Non-Control-Data Attacks Are Realistic Threats." In: *USENIX Security Symposium*. Vol. 5. 2005.

[61] Marco Carvalho et al. “Heartbleed 101”. In: *IEEE Security & Privacy* 12.4 (2014), pp. 63–67.

[62] Wen Xu et al. “From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel”. In: *Computer and Communications Security*. 2015, pp. 414–425.

[63] Wei Wu et al. “FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities”. In: *USENIX Security Symposium*. USENIX Association. 2018, pp. 781–797.

[64] Project Zero. *Project Zero: Racing MIDI messages in Chrome*. 2016. URL: `https://googleprojectzero.blogspot.com/2016/02/racing-midi-messages-in-chrome.html` (visited on 11/22/2020).

[65] Stephen Bradshaw. *Heap Spray Exploit Tutorial: Internet Explorer Use After Free Aurora Vulnerability*. URL: `http://thegreycorner.com/2010/01/24/heap-spray-exploit-tutorial-internet.html` (visited on 11/22/2020).

[66] Sam Ainsworth and Timothy M Jones. “MarkUs: Drop-in use-after-free prevention for low-level languages”. In: *Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 860–860.

[67] John Galea and Mark Vella. “SUDUTA: Script UAF Detection Using Taint Analysis”. In: *Security and Trust Management*. Springer LNCS. 2015, pp. 136–151.

[68] Valentin Jean Marie Manès et al. “The Art, Science, and Engineering of fuzzing: A Survey”. In: *IEEE Transactions on Software Engineering* (2019).

[69] Roberto Baldoni et al. “A survey of symbolic execution techniques”. In: *ACM Computing Surveys* 51.3 (2018), pp. 1–39.

[70] Michał Zalewski. *Technical "whitepaper" for afl-fuzz*. URL: `https://lcamtuf.coredump.cx/afl/technical_details.txt`.

[71] Cristian Cadar, Daniel Dunbar, and Dawson Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In: *Operating Systems Design and Implementation*. USA, 2008, 209–224.

[72] Florent Saudel and Jonathan Salwan. “Triton: A Dynamic Symbolic Execution Framework”. In: *Symposium sur la sécurité des technologies de l’information et des communications*. SSTIC. 2015, pp. 31–54.

[73] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. “The S2E platform: Design, implementation, and applications”. In: *ACM Transactions on Computer Systems* 30.1 (2012), pp. 1–49.

[74] Dawn Song et al. “BitBlaze: A New Approach to Computer Security via Binary Analysis”. In: *Information Systems Security*. 2008.

[75] Patrice Godefroid, Michael Y Levin, and David Molnar. “SAGE: Whitebox fuzzing for security testing”. In: *Queue* 10.1 (2012), pp. 20–27.

[76] Insu Yun et al. “QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing”. In: *USENIX Security Symposium*. Aug. 2018, pp. 745–761.

[77] Sebastian Poeplau and Aurélien Francillon. “Symbolic execution with SymCC: Don’t interpret, compile!” In: *USENIX Security Symposium*. 2020, pp. 181–198.

[78] Nick Stephens et al. “Driller: Augmenting Fuzzing Through Selective Symbolic Execution.” In: *NDSS*. Vol. 16. 2016. 2016, pp. 1–16.

[79] Ron Cytron et al. “Efficiently computing static single assignment form and the control dependence graph”. In: *ACM Transactions on Programming Languages and Systems* 13.4 (1991), pp. 451–490.

[80] Kevin Pulo. “Fun with LD_PRELOAD”. In: *linux. conf. au*. Vol. 153. 2009.

[81] Amanieu d’Antras et al. “Optimizing indirect branches in dynamic binary translators”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 13.1 (2016), pp. 1–25.

[82] Gang-Ryung Uh et al. “Analyzing Dynamic Binary Instrumentation Overhead”. In: *Workshop on Binary Instrumentation and Applications*. 2006.

[83] Pin. *Pin 2.14 User Guide*. 2015. URL: `https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html/`.

[84] Fermin J Serna. “The Info Leak Era on Software Exploitation”. In: 2012.

[85] David Yu Zhu et al. “TaintEraser: Protecting Sensitive Data Leaks using Application-level Taint Tracking”. In: vol. 45. 1. ACM, 2011, pp. 142–154.

[86] Walter Chang, Brandon Streiff, and Calvin Lin. “Efficient and extensible security enforcement using dynamic data flow analysis”. In: *Computer and Communications Security*. ACM. 2008, pp. 39–50.

[87] Kevin M Lepak and Mikko H Lipasti. “Silent stores for free”. In: *International Symposium on Microarchitecture*. IEEE. 2000, pp. 22–31.

[88] Min Gyung Kang et al. “DTA++: Dynamic Taint Analysis with Targeted Control-flow Propagation”. In: *Network and Distributed System Security Symposium*. Internet Society. 2011.

[89] Wei Ming Khoo. *Taintgrind: A taint-tracking plugin for the Valgrind memory checking tool*. 2012. URL: `https://github.com/wmkhoo/taintgrind`.

[90] Brendan Dolan-Gavitt et al. “LAVA: Large-scale Automated Vulnerability Addition”. In: *Symposium on Security and Privacy*. IEEE, 2016, pp. 110–121.

[91] Agner Fog. “Optimizing subroutines in assembly language: An optimization guide for x86 platforms”. In: Copenhagen University College of Engineering. 2008.

[92] Intel. “Intel 64 and IA-32 Architectures Optimization Reference Manual”. In: vol. 1. 2019.

[93] Saman Amarasinghe, Derek Bruening, and Qin Zhao. “Tutorial: Building dynamic instrumentation tools with DynamoRIO”. In: *Code Generation and Optimization*. 2011.

[94] David Brumley et al. “BAP: A binary analysis platform”. In: *International Conference on Computer Aided Verification*. Springer LNCS. 2011, pp. 463–469.

[95] Daniel Leech. *PHPBench*. 2015. URL: `https://phpbench.readthedocs.io/en/latest/index.html#`.

[96] The Apache Software Foundation. *ab – Apache HTTP server benchmarking tool*. URL: `https://httpd.apache.org/docs/2.4/programs/ab.html`.

[97] Jedidiah R Crandall and Frederic T Chong. "Minos: Control data attack prevention orthogonal to memory model". In: *International Symposium on Microarchitecture*. IEEE. 2004, pp. 221–232.

[98] Vijay Nagarajan et al. "Dynamic information flow tracking on multicores". In: *Interaction between Compilers and Computer Architectures*. 2008.

[99] Hari Kannan, Michael Dalton, and Christos Kozyrakis. "Decoupling dynamic information flow tracking with a dedicated coprocessor". In: *Dependable Systems & Networks*. IEEE. 2009, pp. 105–114.

[100] Ralf Karrenberg. *Automatic SIMD Vectorization of SSA-based Control Flow Graphs*. Springer, 2015.

[101] Dan Caselden et al. "HI-CFG: Construction by Binary Analysis and Application to Attack Polymorphism". In: *European Symposium on Research in Computer Security*. Springer. 2013, pp. 164–181.

[102] Asia Slowinska et al. "Data structure archaeology: scrape away the dirt and glue back the pieces!" In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2014, pp. 1–20.

[103] Mario Polino et al. "Jackdaw: Towards automatic reverse engineering of large datasets of binaries". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2015, pp. 121–143.

[104] Xabier Ugarte-Pedrero et al. "RAMBO: Run-time packer analysis with multiple branch observation". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2016, pp. 186–206.

[105] Jiongyi Chen et al. "IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing". In: *NDSS*. 2018.

[106] Yuchen Zhou and David Evans. "Protecting private web content from embedded scripts". In: *European Symposium on Research in Computer Security*. Springer. 2011, pp. 60–79.

[107] Limin Jia et al. "Run-time enforcement of information-flow properties on Android". In: *European Symposium on Research in Computer Security*. Springer. 2013, pp. 775–792.

[108] Daniel Schoepe et al. "Let's face it: Faceted values for taint tracking". In: *European Symposium on Research in Computer Security*. Springer. 2016, pp. 561–580.

[109] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. "Automatic reverse engineering of data structures from binary execution". In: *Proceedings of the 11th Annual Information Security Symposium*. 2010.

[110] Marius Steffens et al. "Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild". In: *NDSS*. 2019.

[111] Intel. "Intel C++ Compiler for Linux Intrinsics Reference". In: 2006.

[112] Tobias Behrens et al. "Efficient SIMD Vectorization for Hashing in OpenCL". In: *International Conference on Extending Database Technology*. OpenProceedings.org, 2018, pp. 489–492.

[113] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. "Rethinking SIMD Vectorization for In-Memory Databases". In: *International Conference on Management of Data*. ACM, 2015, 1493–1508.

[114] Daniel Drake and Johannes Berg. *Unaligned Memory Accesses*. 2020. URL: https://www.kernel.org/doc/Documentation/unaligned-memory-access.txt (visited on 11/22/2020).

[115] Mathias Payer, Enrico Kravina, and Thomas R Gross. "Lightweight memory tracing". In: *USENIX Annual Technical Conference*. 2013, pp. 115–126.

[116] Jianjun Li, Chenggang Wu, and Wei-Chung Hsu. "Dynamic Register Promotion of Stack Variables". In: *Code Generation and Optimization*. IEEE, 2011, pp. 21–31.

[117] Leonid Baraz et al. "IA-32 Execution Layer: A Two-phase Dynamic Translator Designed to Support IA-32 Applications on Itanium®-based systems". In: *International Symposium on Microarchitecture*. IEEE. 2003, p. 191.

[118] DynamoRIO. *The drreg DynamoRIO Register Management Extension*. URL: http://dynamorio.org/docs/page_drreg.html.

[119] John Galea and Daniel Kroening. "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*. Taipei, Taiwan: Association for Computing Machinery, 2020. URL: https://doi.org/10.1145/3320269.3384764.

[120] James Bucek, Klaus-Dieter Lange, et al. "SPEC CPU2017: Next-Generation Compute Benchmark". In: *International Conference on Performance Engineering*. ACM. 2018, pp. 41–42.

[121] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. "An infrastructure for adaptive dynamic optimization". In: *Code Generation and Optimization*. IEEE. 2003, pp. 265–275.

[122] Jonathan L Schilling. "The Simplest Heuristics May be the Best in Java JIT Compilers". In: *ACM Sigplan Notices* 38.2 (2003), pp. 36–46.

[123] Prateek Saxena, R Sekar, and Varun Puranik. "Efficient Fine-grained Binary Instrumentation with Applications to Taint-tracking". In: *Code Generation and Optimization*. ACM. 2008, pp. 74–83.

[124] Yedidya Hilewitz and Ruby B Lee. "Fast bit compression and expansion with parallel extract and parallel deposit instructions". In: *Application-specific Systems, Architectures and Processors*. IEEE. 2006, pp. 65–72.

[125] Arnaldo Carvalho De Melo. "The new Linux Perf tools". In: *Slides from Linux Kongress*. 2010. URL: https://pdfs.semanticscholar.org/16ca/fd05fa375dfe370274cd22b4c16c72d6c53b.pdf.

[126] Brendan Gregg. "The Flame Graph". In: *Communications of the ACM* 59.6 (2016), pp. 48–57.

[127] Ari B. Hayes et al. "GPU Taint Tracking". In: *USENIX Annual Technical Conference*. USENIX Association, 2017, pp. 209–220.

[128] Kangkook Jee et al. "A General Approach for Efficiently Accelerating Software-based Dynamic Data Flow Tracking on Commodity Hardware". In: *Network and Distributed System Security Symposium*. Internet Society. 2012.

[129] Olatunji Ruwase et al. “Decoupled lifeguards: Enabling path optimizations for dynamic correctness checking tools”. In: *ACM Sigplan Notices* 45.6 (2010), pp. 25–35.

[130] Ali Davanian, Zhenxiao Qi, and Yu Qu. “DECAF++: Elastic Whole-System Dynamic Taint Analysis”. In: *RAID*. USENIX Association, 2019.

[131] Andrey Ermolinskiy et al. “Towards practical taint tracking”. In: UCB/EECS-2010-92 (2010).

[132] Daniel Townley et al. “LATCH: A Locality-Aware Taint CHecker”. In: *International Symposium on Microarchitecture*. ACM, 2019, pp. 969–982.

[133] Subarno Banerjee et al. “Iodine: Fast Dynamic Taint Tracking Using Rollback-free Optimistic Hybrid Analysis”. In: *Symposium on Security and Privacy*. IEEE. 2018.

[134] Jiang Ming et al. “TaintPipe: Pipelined Symbolic Taint Analysis”. In: *USENIX Security Symposium*. 2015, pp. 65–80.

[135] Milind Chabbi et al. “Efficient Dynamic Taint Analysis using Multicore Machines”. MA thesis. The University of Arizona, Department of Computer Science, 2007.

[136] Baojiang Cui et al. “FlowWalker: A Fast and Precise Off-Line Taint Analysis Framework”. In: *Emerging Intelligent Data and Web Technologies*. IEEE. 2013, pp. 583–588.

[137] Jiang Ming et al. “StraightTaint: Decoupled Offline Symbolic Taint Analysis”. In: *Automated Software Engineering*. ACM, 2016, pp. 308–319.

[138] Asia Slowinska and Herbert Bos. “Pointless Tainting? Evaluating the Practicality of Pointer Tainting”. In: *European Conference on Computer Systems*. ACM. 2009, pp. 61–74.

[139] Michael Dalton, Hari Kannan, and Christos Kozyrakis. “Tainting is not pointless”. In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 88–92.

[140] Asia Slowinska and Herbert Bos. “Pointer Tainting Still Pointless (But We All See the Point of Tainting)”. In: *ACM SIGOPS Operating Systems Review* 44.3 (2010), pp. 88–92.

[141] Julian Kirsch et al. “PwIN–Pwning Intel piN: Why DBI is Unsuitable for Security Applications”. In: *European Symposium on Research in Computer Security*. Springer LNCS. 2018, pp. 363–382.

[142] Daniele Cono D’Elia et al. “SoK: Using dynamic binary instrumentation for security (and how you may get caught red handed)”. In: *Asia Conference on Computer and Communications Security*. ACM. 2019, pp. 15–27.

[143] Brendan Dolan-Gavitt et al. “Repeatable reverse engineering for the greater good with Panda”. In: *Columbia University Academic Commons* (2014).

[144] Lorenzo Cavallaro, Prateek Saxena, and R Sekar. “Anti-taint-analysis: Practical evasion techniques against information flow based malware defense”. In: 2007, pp. 1–18.

[145] Zheng Leong Chua et al. “One Engine To Serve’em All: Inferring Taint Rules Without Architectural Semantics”. In: *Network and Distributed System Security Symposium*. Internet Society. 2019.

[146] Nickolai Zeldovich et al. “Hardware Enforcement of Application Security Policies Using Tagged Memory”. In: *OSDI*. Vol. 8. 2008, pp. 225–240.

[147] Guru Venkataramani et al. “FlexiTaint: A Programmable Accelerator for Dynamic Taint Propagation”. In: *High Performance Computer Architecture*. IEEE. 2008, pp. 173–184.

[148] S. Fytraki et al. “FADE: A Programmable Filtering Accelerator for Instruction-grain Monitoring”. In: *High Performance Computer Architecture*. IEEE, 2014, pp. 108–119.

[149] Meltem Ozsoy et al. “SIFT: A low-overhead dynamic information flow tracking architecture for SMT processors”. In: *International Conference on Computing Frontiers*. ACM. 2011, p. 37.

[150] Man-Ki Yoon et al. “PIFT: Predictive Information-Flow Tracking”. In: *Architectural Support for Programming Languages and Operating Systems*. ACM, 2016, pp. 713–725.

[151] H. Chen et al. “From Speculation to Security: Practical and Efficient Information Flow Tracking Using Speculative Hardware”. In: *International Symposium on Computer Architecture*. IEEE, 2008, pp. 401–412.

[152] Michael Dalton, Hari Kannan, and Christos Kozyrakis. “Raksha: A Flexible Information Flow Architecture for Software Security”. In: *International Symposium on Computer Architecture*. ISCA ’07. ACM, 2007, pp. 482–493.

[153] Reid Kleckner. “Optimization of Naïve Dynamic Binary Instrumentation Tools”. MA thesis. Massachusetts Institute of Technology, 2011.

[154] Wenwen Wang et al. “A General Persistent Code Caching Framework for Dynamic Binary Translation (DBT)”. In: *USENIX Annual Technical Conference*. USENIX Association. 2016, pp. 591–603.

[155] Byron Hawkins et al. “Optimizing binary translation of dynamically generated code”. In: *Code Generation and Optimization*. IEEE. 2015, pp. 68–78.

[156] Rajiv Gupta, Eduard Mehofer, and Youtao Zhang. *Profile Guided Compiler Optimizations*. 2002.

[157] Vijay Janapa Reddi et al. “Persistent Code Caching: Exploiting code reuse across executions and applications”. In: *Code Generation and Optimization*. IEEE. 2007, pp. 74–88.

[158] Mohit Tiwari et al. “Complete Information Flow Tracking from the Gates Up”. In: *Architectural Support for Programming Languages and Operating Systems*. ACM, 2009, pp. 109–120.

[159] Lin Clark. *A crash course in just-in-time (JIT) compilers*. 2017. URL: https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/.

[160] Cristina David and Daniel Kroening. “Program synthesis: challenges and opportunities”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375.2104 (2017), p. 20150403.

[161] Chao Zhang et al. “IntPatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time”. In: *European Symposium on Research in Computer Security*. Springer. 2010, pp. 71–86.

[162] Raymond Mui and Phyllis Frankl. “Preventing web application injections with complementary character coding”. In: *European Symposium on Research in Computer Security*. Springer. 2011, pp. 80–99.

[163] Heng Yin et al. “Hookscout: Proactive binary-centric hook detection”. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2010, pp. 1–20.

[164] Matthias Neugschwandtner et al. “dAnubis - Dynamic device driver analysis based on virtual machine introspection”. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2010, pp. 41–60.

[165] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. “Symbolic deobfuscation: From virtualized code back to the original”. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2018, pp. 372–392.

[166] Yuhei Kawakoya et al. “API Chaser: Anti-analysis resistant malware analyzer”. In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2013, pp. 123–143.

[167] Vasilis Pappas et al. “CloudFence: Data flow tracking as a cloud service”. In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2013, pp. 411–431.

[168] J. Caballero et al. “Binary Code Extraction and Interface Identification for Security Applications”. In: *NDSS*. 2010.