LATCH: A Locality-Aware Taint CHecker

Daniel Townley
Binghamton University
Binghamton, New York, USA
dtownle1@binghamton.edu

Khaled N. Khasawneh
George Mason University
Fairfax, Virginia, USA
kkhasawn@gmu.edu

Dmitry Ponomarev
Binghamton University
Binghamton, New York, USA
dponomar@binghamton.edu

Nael Abu-Ghazaleh
University of California Riverside
Riverside, California, USA
nael@cs.ucr.edu

Lei Yu
Binghamton University
Binghamton, New York, USA
lyu@cs.binghamton.edu

ABSTRACT

We present LATCH (short for Locality-Aware Taint CHecker), a generalizable architecture for optimizing dynamic information flow tracking (DIFT). LATCH exploits the observation that information flows under DIFT exhibit strong *temporal locality*, with typical applications manipulating sensitive data during limited phases of computation. This property allows *LATCH* to monitor significant spans of execution using lightweight, coarse-grained checks, invoking precise, computationally intensive tracking logic only during periods of execution that involve sensitive data. *LATCH* implements this policy without sacrificing the accuracy of DIFT.

We propose and evaluate three systems incorporating *LATCH*: *S-LATCH* to accelerate software-based DIFT on a single core; P-LATCH to accelerate multicore software-based DIFT, and *H-LATCH* to reduce the architectural complexity of hardware-based DIFT. We developed an FPGA prototype of the LATCH system, demonstrating that its advantages come with negligible impact on power and complexity and no effect on processor cycle time.

CCS CONCEPTS

• **Security and privacy** → **Intrusion detection systems**; **Security in hardware**; **Information flow control**.

KEYWORDS

dynamic information flow tracking, security in hardware

ACM Reference Format:
Daniel Townley, Khaled N. Khasawneh, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Lei Yu. 2019. LATCH: A Locality-Aware Taint CHecker. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3352460.3358327

MICRO-52, October 12–16, 2019, Columbus, OH, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358327

1 INTRODUCTION

Dynamic Information Flow Tracking (DIFT) is a general security and privacy protection mechanism that is based on marking and tracking data as it flows through the system [12, 48, 54]. In a typical security application, DIFT marks (taints) data originating from untrusted sources, such as the network or file system, and tracks this data and any data derived from it through registers and memory system during program execution. DIFT checks operations on tainted data for conflicts with declared policies, and generates security exceptions in response to violations. Though computationally intensive, DIFT provides effective protection against a wide range of attacks. For example, DIFT can detect buffer overflows and thus prevent a wide range of control-flow hijacking attacks, such as return-oriented programming and jump-oriented programming attacks [2, 46].

Despite these benefits, existing DIFT proposals face significant obstacles to widespread adoption. Though highly flexible and relatively easy to deploy, *software-based DIFT* [3, 8, 9, 19, 21, 30, 32, 33, 38, 39, 44, 55, 57] requires recompilation to insert additional DIFT instructions into the monitored program, imposing performance overheads ranging from 2X to as high as 100X, which are generally prohibitive for real-time security monitoring. *Hardware-based DIFT* [7, 11–13, 15, 31, 40, 43, 47, 48, 50, 53, 54] virtually eliminates these overheads, but requires additional processor logic to enforce propagation and data-use policies, as well as a dedicated *taint cache* or equivalent structures to check the taint status of each memory operand.

To address these challenges, we propose *LATCH* (short for Locality-Aware Taint CHecker), a versatile, lightweight hardware module that greatly mitigates the costs of deploying *both* hardware- and software-based DIFT. *LATCH* is motivated by a new analysis, presented in section 3 of this paper, that shows a high degree of *temporal locality* that is inherent in DIFT information flows. Specifically, most applications that we evaluated only manipulate tainted bytes during limited phases of execution. In general, there are two reasons for this behavior. First, the potential for taint explosion [32] is a strong motivation for security policy designers to exercise restraint in defining the set of untrusted inputs, thus contributing to taint locality. Second, programs have been shown to execute as a sequence of distinct phases [18], some of which may not manipulate tainted data.

During the periods of execution where no taint manipulation takes place, *LATCH* checks operands for the presence of taint using a coarse-grained taint checking module that entails minimal architectural complexity and imposes a negligible performance overhead. On detecting taint at a coarse granularity, *LATCH* temporarily invokes a byte-precise monitor to propagate taint tags and enforce data utilization rules. Due to the temporal locality of taint, this heavy-weight monitoring is rarely invoked, and can be significantly scaled down in proportion to a diminished workload.

LATCH performs passive taint checking by exploiting the *spatial locality* of tainted data, a property that has been established in previous works [29, 49]. *LATCH*'s coarse taint state divides memory into fixed length, multi-byte *taint domains*, and assigns taint to each domain that contains one or more tainted bytes. As we show in Section 4, this *coarse taint state* can be cached and checked using minimal hardware, and, due to the spatial locality of taint, closely approximates the byte-precise taint state. Figure 1 shows the intuition behind this approach. For elements in untainted domains (A), taint status always corresponds to that of the coarse taint domain. Tainted domains may contain untainted bytes (B), which are false positives for *LATCH*, as well as tainted bytes (C). In *LATCH*, false positives can be easily checked and dismissed when the byte-precise policy is invoked, allowing the combined system to preserve an accurate taint state. Similarly, because all possible taint events (false and true positives) are subject to fine-grained checks, the combine system offers precise taint checking without introducing false negatives

LATCH is a versatile and generalizable hardware module that can enhance both software- and hardware-based DIFT systems. When combined with software-based DIFT, *LATCH* controls the invocation of heavy-weight software monitoring, localizing the overhead of DIFT to program regions where taint is actively manipulated. The effects of this policy are illustrated in Figure 2. By running only the taint-handling epoch (b) in software mode, *LATCH* allows taint-free epochs (a) and (c) to be executed in hardware at near-native speeds.

When combined with hardware-based DIFT, *LATCH* reduces architectural complexity by screening accesses to a conventional, hardware-based taint cache. This filtering greatly reduces the number of evictions and cache misses experienced by the byte-precise taint cache. Consequently, the precise taint cache, which is the greatest contributor to architectural complexity in most hardware DIFT schemes, can be reduced in size without sacrificing performance.

To demonstrate the various applications of *LATCH*, we design and evaluate three DIFT systems that incorporate the *LATCH* module. The first system, called *S-LATCH*, uses *LATCH* to optimize software-based DIFT on a single core, achieving a 4X speedup over a baseline system ([32]) without losing precision. The second system, called *P-LATCH*, applies *LATCH* to optimize DIFT monitoring using a separate core. It achieves an estimated 2X speedup over the existing LBA scheme ([6, 7]). Our third system, called *H-LATCH*, applies *LATCH* to hardware-based DIFT, achieving a mean taint cache miss rate of less than 0.02% despite a taint cache capacity of less than 8% the size of a conventional implementation ([54]). In addition, we implemented the core *LATCH* architecture on an FPGA, and demonstrate negligible impacts on CPU power, cycle time, and architectural complexity.

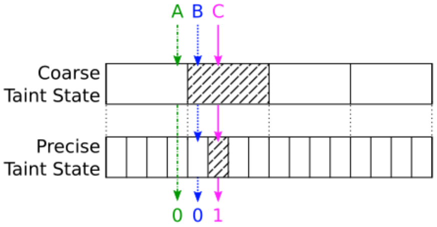


Figure 1: Coarse-grain Taint Domains with LATCH

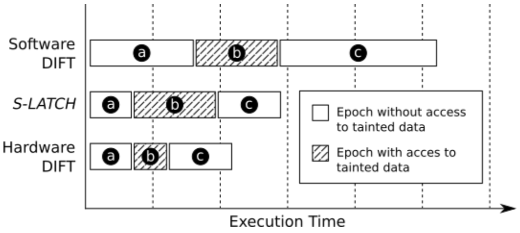

Software DIFT
S-LATCH
Hardware DIFT
a
b
c
Epoch without access to tainted data
Epoch with acces to tainted data
Execution Time

Figure 2: Exploiting Temporal Locality with LATCH

2 BACKGROUND AND SCOPE

DIFT operations can be abstracted into four main components: initialization, storage, propagation and validation (or checking). These operations are depicted in Figure 3. *Initialization* (1) occurs when data is tagged as tainted; such data may be data read from an external *taint source*, or sensitive data that we are interested in tracking to prevent its exposure. Typical taint sources include files, network sockets, or user inputs. A typical initialization scheme assigns each byte read from such a source a *taint tag* indicating its origin. The *storage* of taint tags (2) differs significantly among DIFT implementations. Software-based DIFT stores taint tags for registers and memory addresses in software-defined data structures. Hardware-based DIFT typically provides dedicated hardware to store tags for registers, and stores memory taint tags in designated memory accessible through a dedicated *taint cache* [48, 54]. DIFT applies taint not only to data initialized directly, but also *propagates* taint to data that is computed or modified using tainted inputs (3). Software-based DIFT schemes introduce instrumentation to check the input operands of each native instruction and generate the result according to specified rules. Hardware-based DIFT performs this process using dedicated logic. Finally, every DIFT system supports a *validation* (or checking) procedure to ensure that the use of tainted data is consistent with pre-defined security rules (4). In hardware-based systems, validation is generally implemented in hardware using rule-checking mechanisms similar to those employed for propagation. Software-based DIFT schemes perform validation using instructions.

Software-based DIFT is commonly implemented using *dynamic binary instrumentation* (DBI). In DBI, a software engine is injected into a monitored program's memory space and assumes control of its execution. The DBI engine extracts the program's basic blocks at runtime, rewrites them with interleaved instrumentation code, and virtually executes them in place of the original code. When used for DIFT, the instrumentation code includes logic to check and propagate taint. DBI can be applied at runtime, requires no access to

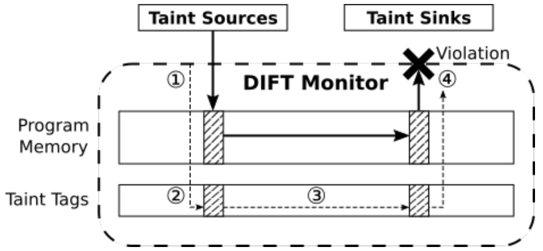


Figure 3: Overview of DIFT operations

source code, and can be accomplished using widely available DBI engines such as Intel's Pin [37] or the open-source dynamoRIO [5]. The software component of *S-LATCH* is based on DBI.

LATCH targets the implementations of DIFT described in this section. This DIFT policy has applications in detecting control-flow exploits and malicious data leakage, which are primary security threats. The next section focuses on locality characteristics of these policies. While *LATCH* is applicable in principle to any dynamic tagging policy with exploitable locality, our conclusions do not necessarily apply to all other tagged memory schemes. For instance, schemes that tag pointers to enforce their use with corresponding memory regions[10] are likely to require continual comparison between pointer and memory taint tags, and would not benefit from *LATCH* in its current form. Similarly, indirect tracking of taint through control flows poses significant challenges to conventional DIFT as well as to *LATCH* due to rapid taint explosion, and is an open problem for continuing research. The extension of *LATCH* optimizations to accommodate other tagging policies is beyond the scope of this paper.

3 UNDERSTANDING TAINT LOCALITY IN DIFT

In this section, we characterize the spatio-temporal properties of data flows that motivate *LATCH*'s design. Our analysis shows that the strong temporal locality of operations on tainted data allows the exclusive use of coarse-grained taint checks for extended periods of program execution. We also show that in most applications, DIFT dataflows exhibit a high degree of spatial locality, making coarse-grained tainting an effective, lightweight strategy, especially when paired with a mechanism to exploit temporal locality.

3.1 Experimental Framework

All evaluations were performed under the Debian 8 distribution of Linux. We generated DIFT data flows using *libdft* [32], an open-source taint tracking library for Intel's Pin dynamic binary instrumentation engine. We measured the temporal and spatial behavior of data flows for common network applications, in which taint is introduced through `socket` and `accept` system calls, and for desktop applications from the SPEC CPU 2006 benchmark suite, which receive untrusted data from files. The network benchmarks included two popular network clients, *wget* and *curl*, the *Apache* web server and the *mySQL* server. The evaluation of performance-oriented SPEC benchmarks in combination with a selection of security-critical network applications is a standard practice in DIFT research [28, 32, 49, 49]. Because programs experience varied phases of taint propagation over the course of their execution, we sought to run all of our benchmarks to completion, wherever possible. Except where noted, all results are derived from complete program runs. To complete evaluations in reasonable time under Pin instrumentation, we ran the SPEC 2006 benchmarks using the smaller input sets provided with the benchmarks. We omitted results for SPEC benchmarks that did not receive any file input. All of our evaluations apply the classical Dynamic Taint Analysis rules used by [32].

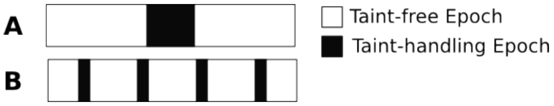


Figure 4: Examples of two program traces with long and short taint-free epochs

The general evaluation assumes a conservative policy that taints data from potentially untrustworthy network or file sources. However, for some network applications, more nuanced policies may better represent realistic use cases. Specifically, we note that in production environments a web server may handle requests both from trusted sources (such as clients on a local network), and untrusted sources (such as unknown remote clients). To evaluate these scenarios, we included three additional assessments of the *Apache* server in which subsets of requests were tainted as untrustworthy. This was accomplished within Pin by generating a random number for each `accept4` system call (used to read a single request by *Apache*), and tainting the data introduced by that call if the number exceeded a specified threshold. This resulted in additional evaluations of *Apache* in which approximately 25, 50, and 75 percent of incoming requests were marked as trusted, resulting in differing patterns of locality. The runs of both apache and mySQL included 1000 requests.

3.2 Temporal locality of accesses to tainted data

This section examines two aspects of the temporal locality of operations on tainted data: 1) the percentage of instructions that manipulate tainted data, and 2) the duration of taint-free epochs.

3.2.1 Percentage of taint-free instructions. This characteristic defines the upper bound on the cumulative length of exploitable taint-free epochs, and is thus a key measure of the efficiency of *LATCH*. Table 1 shows the percentage of instructions in each benchmark that touched tainted data. In 16 of the 20 SPEC benchmarks, fewer than 1% of instructions accessed tainted data. In the remaining programs, *astar*, *perl*, *soplex* and *sphinx*, between 2% and 22% of instructions were involved in manipulating tainted data. As illustrated in Table 2, taint-handling instructions were slightly more frequent in the network benchmarks, but did not exceed 2%. Not surprisingly, the amount of taint handled declined in a linear fashion when apache was executed with an increasing number of trusted connections.

3.2.2 Duration of taint-free epochs. In addition to the overall percentage of taint-free instructions, the duration of continuous taint-free epochs impacts *S-LATCH*'s potential effectiveness. Figure 4 illustrates this effect with two hypothetical applications, A and B, that have the same proportion of tainted and untainted instructions. In A, these instructions comprise a single, long epoch, whereas in B they are divided into several shorter epochs. In *S-LATCH*, executing

	astar	*bzip2*	*calculix*	*cactusADM*	*gcc*	*gobmk*	*gromacs*	*h264ref*	*hmmer*	*lbm*	*mcf*	*namd*	*omnetpp*	*perlbench*	*povray*	*sjeng*	*soplex*	*sphinx*	*wrf*	*Xalan*
Instructions touching tainted data (%)	21.73	0.01	0.28	0.01	0.08	0.01	0.19	0.01	0.01	0.14	0.29	0.17	0.01	2.67	0.21	0.01	7.69	13.53	0.28	0.11

Table 1: Percentages of instructions touching tainted data in SPEC 2006 benchmarks

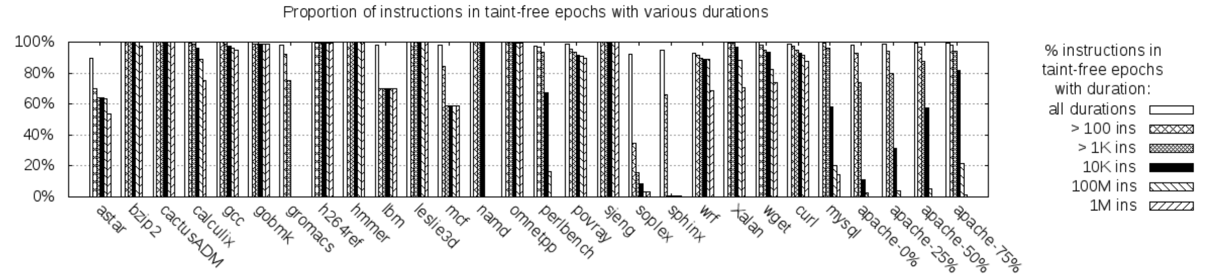


Figure 5: Percentage of instructions in taint-free epochs of various lengths

	curl	*wget*	*mySQL*	*apache*	*Apache-25*	*Apache-50*	*Apache-75*
Instructions touching tainted data (%)	1.13	0.15	0.19	1.94	1.49	0.95	0.45

Table 2: Percentages of instructions touching tainted data in network applications

taint-free epochs in accelerated mode incurs a constant overhead with every switch between the hardware and software layers. Consequently, while A must incur this overhead only twice, B experiences switching repeatedly, reducing cumulative acceleration relative to A.

To evaluate the duration of taint-free intervals in typical workloads, we ran each benchmark for 500 million instructions and detected all taint-free epochs. We assigned these taint-free epochs to sets based on the epoch length. We considered the sets that contained more than 100, 1K, 10K, 100K and 1M instructions. Note that under these sorting criteria, some epochs belong to multiple sets. Figure 5 shows the total number of instructions represented in each epoch size category as a percentage of all instructions executed. Programs with more instructions belonging to longer epochs resemble program A in Figure 4, and are likely to benefit most from acceleration.

Figure 5 shows that 13 of 20 benchmarks executed more than 80% of their instructions during taint-free epochs of 1K instructions or more, a pattern similar to that of program A in Figure 4. Benchmarks such as *lbm*, *mcf* and *gromacs* have fewer instructions in long taint-free epochs. However, our later evaluations show that these programs still include enough longer periods to allow significant acceleration. The four remaining benchmarks, *astar, sphinx, perl and soplex*, more closely resemble program B in Figure 4, in that a higher proportion of their taint-free instructions are divided among shorter epochs.

Among the network applications, the web clients *curl* and *wget* have a relatively high proportion of long taint-free epochs, while mySQL showed a moderately sharp drop-off in the length of exploitable epochs. Under the most restrictive DIFT policy, Apache Server experienced a severe fragmentation of taint free periods. However, under policies that included trusted clients, the duration of taint-free epochs increased significantly.

3.3 Spatial locality of tainted data

The spatial locality of tainted data underpins *LATCH*'s two-layer logic, and predicts the viability of locality-aware DIFT optimizations. The following experiments focus on two aspects of spatial locality that affect *LATCH*: 1) the distribution of taint across the memory space accessed by each application, and 2) the accuracy of coarse-grained tainting at various granularities, measured by the frequency of false positives.

3.3.1 Distribution of taint in the memory space. The distribution of tainted data in the memory space of each application is a fundamental measure of spatial locality that is critical to locality-aware taint caching. If tainted data occurs in a small fraction of memory regions, data in taint-free regions will frequently be accessed, allowing most DIFT checks to be resolved using the coarse taint state.

To evaluate the distribution of tainted data in the memory of each application, we recorded the number of 4KiB memory pages that received tainted data in the course of execution, and compared this value to the total number of pages accessed. The results of this evaluation for the SPEC benchmarks, presented in Table 3, show that for 17 out of 20 benchmarks, more than 90% of the accessed pages were completely free of taint. A minority of applications, specifically *astar*, *soplex*, and *sphinx* showed incidences of taint likely to interfere with locality-based optimizations.

Table 4 shows the results for network applications. Tainted pages occured somewhat more frequently in these evaluations, but occupied a minority of the memory space in all cases. The apache web server showed the highest occurrence of taint at the page granularity. Interestingly, the frequency of taint did not vary with the proportion

	astar	*bzip2*	*cactusADM*	*calculix*	*gcc*	*gobmk*	*gromacs*	*h264ref*	*hmmer*	*lbm*	*mcf*	*namd*	*omnetpp*	*perlbench*	*povray*	*sjeng*	*soplex*	*sphinx*	*wrf*	*Xalan*
Pages accessed	2344	52110	6199	806	2590	3981	3604	6861	182	104766	21481	11575	1786	203	725	44713	412	7133	25182	1634
Pages tainted	2001	70	1	9	213	1	17	183	5	2	2	3	14	22	24	3	84	4133	246	105
Accessed pages tainted (%)	85.37	0.13	0.02	1.12	8.22	0.03	0.47	2.67	2.75	0.01	0.01	0.03	0.78	10.84	3.31	0.01	20.39	57.94	0.98	6.43

Table 3: Distribution of taint at page granularity for SPEC 2006 benchmarks

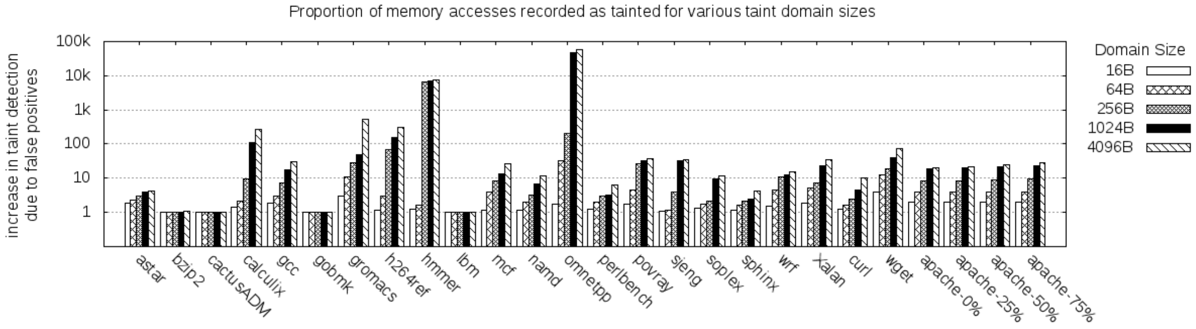


Figure 6: Increase in taint detection rates for coarse-granularity tainting policies using various multibyte taint domains. Values over 1 correspond to the ratio of false positives relative to the byte-precise baseline.

	curl	*wget*	*mysql*	*apache*	*apache-25*	*apache-50*	*apache-75*
Pages accessed	600	1591	10483	1113	1170	1101	1115
Pages tainted	33	44	435	238	260	231	238
Accessed pages tainted (%)	5.5	2.77	4.15	21.38	22.22	20.98	21.35

Table 4: Page granularity taint distribution for network applications

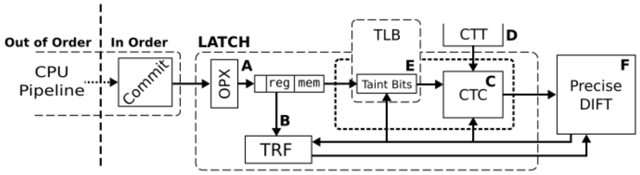


Figure 7: *LATCH* hardware organization

of untrusted requests. This behavior suggests that the same set of pages are used in sequence to store tainted and untainted requests, meaning that the evaluation, which covered the entire execution, did not capture changes in the taint contents of these pages over time.

3.3.2 Frequency of false positives under coarse taint checking. The incidence of false positives under coarse-grained taint checking is a more concrete predictor of *LATCH*'s effectiveness. In *LATCH*, false positives can occur when an accessed taint domain contains a mixture of tainted and untainted data, and result in unnecessary invocations of the precise propagation and checking logic. Smaller taint domains typically reduce the frequency of false positives, but require more complex hardware to check. The trade-off between taint-domain granularity and the frequency of false positives is thus critical to *LATCH*'s implementation.

Figure 6 shows how the number of accessed memory elements falsely reported as tainted increases in proportion to taint domain size. The number of taint detection events are plotted as a multiple of the number of tainted elements detected by byte-precise taint; for instance, a value of 10x means that the precise DIFT logic is invoked 10 times more frequently under a given domain granularity due to false positives. For most SPEC programs, taint detection accuracy degrades steadily as the domain size increases, but remains useful for most applications for domains of 64 bytes and in some cases even 256 bytes. The *bzip2*, *gobmk*, and *lbm* benchmark are notable in that the coarse-grained tainting policies produced few or no false positives. This suggests that in these programs, tainted data is closely aligned with page boundaries.

Though counterintuitive, the low rates of taint for benchmarks such as *bzip2* and the web clients are consistent with the operation of these programs. In both the compression algorithm used by *bzip2*, and the SSL and TLS cryptographic schemes used by the web clients, data from the taint source is replaced by untainted, precomputed values from a substitution table[45]. We note that in general, the taint policies used in this section are conservative in that they tag all external data as untrusted: to avoid taint explosion and excessive false positives, security policies are likely to define a subset of inputs as untrusted, rather than tagging all inputs indiscriminately, thus further increasing taint locality.

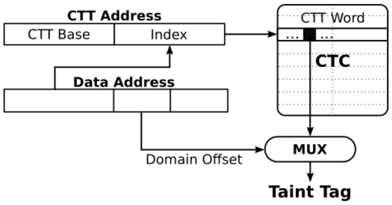


Figure 8: Taint cache address generation logic in *LATCH*

4 *LATCH* DESIGN

Motivated by the preceding analysis of taint locality, this section presents the design and operation of *LATCH*. The core *LATCH* logic, shown in Figure 7, consists of the following components:

- **Extraction Logic (A):** Extracts operands from committed instructions.
- **Taint Register File(TRF) (B):** Stores byte-level taint associated with the register file.
- **Coarse Taint Cache (CTC) (C):** Performs coarse taint checks on memory operands, evaluating taint status at the granularity of tens of bits.
- **Coarse Taint Table (CTT) (D):** In-memory data structure storing coarse taint tags for use by the CTC.
- **TLB Taint Bits (E):** Provides additional filtering at the granularity of kilobytes. This justification for this is elaborated in Section 4.2.
- **Precise DIFT mechanism (E):** Implements byte-granularity propagation, if required following CTC taint check.

Among the components described above, the operand extraction logic and the TRF are simple to implement and are analogous to structures described in existing hardware-based DIFT proposals such as [54]. *LATCH* provides alternate update mechanisms to support integration with hardware- and software based DIFT, and we defer discussion of these mechanisms to sections 5.1 and 5.3, which introduce the corresponding *LATCH* integrations. In the following sections we discuss the combined coarse tainting infrastructure, comprised of the CTT, CTC, and TLB Taint Bits, which underpin *LATCH*'s exploitation of taint locality.

4.1 Coarse taint representation and caching

The CTC and the underlying CTT constitute the core of *LATCH*'s coarse tag-checking infrastructure. The CTT uses single-bit tags to represent taint at the granularity of tens of bytes. For instance, by using one bit to convey the taint state for a 32-bit memory region, or *taint domain*, the CTT can hold coarse taint data for 1KB of memory in a single 32-bit word. Due to the high degree of compression inherent in the CTT, *LATCH*'s CTT requires comparatively few entries to hold the (approximate) taint state for a large set of addresses. Moreover, the temporal locality of taint allows these addresses to remain cached for long periods of time, allowing *LATCH* to achieve high hit rates using a small CTC. Consequently, the CTC can be implemented with minimal hardware while maintaining high hit rates.

Figure 8 details *LATCH*'s procedure for accessing the taint cache. To retrieve the index of the CTT word with a memory operand's taint bit, the taint-cache uses the corresponding high order bits of the extracted operand. *LATCH* combines this index with the base address of the CTT to obtain the absolute address of the CTT word in virtual memory.

4.2 TLB-Taint Bits

In the taint analysis in section 3.3, we observed that spatial locality is evident at the page and Kilobyte level, as well as at the level of taint domains consisting of tens of bytes. While the second type of locality is often needed to avoid excessive false positives, and is thus employed in the CTT, *LATCH*'s taint caching infrastructure also exploits the former type of locality by using page-level taint bits in each TLB entry to screen out checks to large, untainted regions before they reach the CTC. To provide page-level filtering, *LATCH* extends the page table and the TLB with a small number of *page taint bytes*, which divide each page into one or more multi-kilobyte *page-level taint domains*. The extension of the TLB with additional metadata is an approach used in several existing proposals [20, 41] to reduce accesses to main memory. Alternatively, the page taint data could be stored in a separate small cache, whose misses would coincide with those of the TLB. To simplify implementation, each page-level taint domain corresponds to a single word of CTT taint tags. When data in a tainted page is being manipulated, the CTC logic handles locality at the sub-page level.

5 *LATCH* INTEGRATIONS WITH DIFT SYSTEMS

In this section, we demonstrate the integration of *LATCH* into three variants of DIFT: single-core software-based DIFT, parallelized software-based DIFT on two cores, and hardware-based DIFT. We call these systems *S-LATCH*, *P-LATCH*, and *H-LATCH*, respectively. In *S-LATCH* and *P-LATCH*, *LATCH* greatly reduces execution overheads relative to the equivalent software-only policies; in *H-LATCH* , the *LATCH* module mitigates architectural complexity without sacrificing performance.

5.1 *S-LATCH*: Accelerating Software DIFT

S-LATCH is a hardware-software system in which *LATCH* controls the invocation of software-based DIFT on a single core. Using the minimal *LATCH* hardware base, *S-LATCH* combines the flexibility of software-based DIFT with performance properties that are closer to hardware implementations.

At a high level, *S-LATCH* consists of two parts: (1) a *software component*, which uses DBI to instrument application code with DIFT propagation and validation operations; and (2) a *hardware acceleration component*, which uses the core *LATCH* logic to continuously check that data accessed by the program under monitoring is not tainted. Figure 9 illustrates the high-level operation of the system. During normal execution, the hardware component continually checks memory operands against the *LATCH* coarse taint state, while register operands are checked against taints in a small taint register file (TRF) (1). Upon detecting taint, the hardware component traps to the software component (2), which checks the trapped instruction against the precise taint state to verify that the operation is indeed occurring on tainted data (3). The software component maintains a secondary image of the monitored process that is dynamically instrumented with DIFT instructions, and transfers control to this

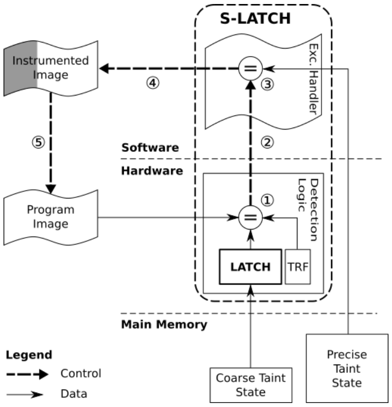


Figure 9: High-level Operation of *S-LATCH*

strf reg	set the TRF flags to the value in the register reg
stnt adr reg	update the taint status of memory address mem, to the value in reg
ltnt reg	load the address operand that caused the most recent *S-LATCH* exception into register reg

Table 5: Special *S-LATCH* instructions

image if the hardware exception is confirmed (4). After performing DIFT in software for a specified period, the instrumented image initiates a transfer of control back to the uninstrumented image (5). Mechanisms inspired by [1] and [52] could enhance the security of *S-LATCH*'s hardware and software components, respectively, in the presence of untrusted system software.

5.1.1 S-LATCH Components. By performing all precise taint checks in software, *S-LATCH* obviates hardware propagation logic and the precise taint cache, retaining only the core *LATCH* module, TRF, and operand extraction logic. The *S-LATCH* software layer consists of three distinct units: the *S-LATCH* exception handler, the DBI engine, and the taint initialization logic. The exception handler, discussed in section 5.1.2, is responsible for validating exceptions issued by the hardware layer against the precise taint state and, if an exception is confirmed, for transferring control to the DBI engine. The DBI engine (discussed in Section 5.1.3), is responsible for performing DIFT analysis and for returning control to hardware at the end of each taint-containing epoch. Additionally, *S-LATCH* introduces three dedicated ISA instructions shown in Table 5. The strf (set TRF) instruction updates the taint register file following a period of in-software propagation. The stnt (set taint) is a special memory write instruction that updates the CTT directly though the taint cache, rather than the regular data cache. Lastly, in the course of filtering false positives, the *S-LATCH* exception handler uses the ltnt (load taint) instruction to load the address that triggered the last *S-LATCH* hardware exception.

5.1.2 Transition from Hardware to Software Monitoring. When the hardware monitor detects a potential taint, the exception handler serves as a point of entry to *S-LATCH*'s software layer. The handler performs the following operations. First, it screens out any false positive operations issued by the hardware based on the precise taint information at the software layer. If the operation is indeed a false positive, the handler switches back to the hardware mode. If the operation is a true taint operation, the handler initiates a transfer of control from the native application code to the instrumented image. Finally, it passes control to the DBI software monitor. To evaluate the status of an address, the handler loads the address into a register using the ltnt instruction, checks the corresponding entry in the precise taint state, and, in accordance with the result of the check, transfer control to either the native or instrumented program image.

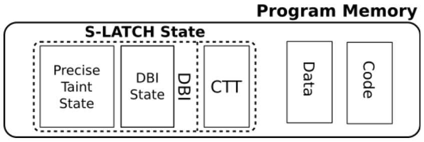


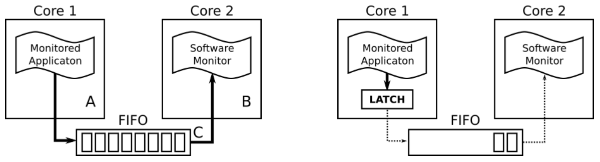
Figure 10: Memory layout under *S-LATCH* monitoring

5.1.3 Software Taint Checking and Propagation. *S-LATCH* uses a Pin-based program (Pintool) incorporating functionality from *libdft* library. However, other, potentially lighter-weight DBI engines could easily be substituted in the future. The instrumentation code run by this engine is modified to update the CTT and precise taint state directly using the stnt instruction, thus reducing the number of taint-related memory operations and avoiding the cache pollution. Figure 10 shows the integration of *S-LATCH*'s software components into program memory.

To make full use of hardware acceleration, the software component must relinquish control as quickly as possible following each epoch of active taint propagation. However, given our observations on the temporal locality of taint operations, if we return to the hardware monitor immediately, it is likely that other tainted data will be accessed soon, causing another switch and harming performance. Thus, we implemented a *timeout policy* in the software layer that determines when to switch back to the hardware monitoring mode. While a variety of timeout policies are possible, *S-LATCH* achieves strong performance using a simple timeout scheme that returns control to hardware after 1000 instructions have been executed without manipulating tainted data.

5.1.4 Updating the Coarse Taint State. In *S-LATCH*, the absence of a hardware taint cache necesitates an alternative method of synchronizing the coarse and precise taint states. Specifically, while the stnt instruction allows the *S-LATCH* software layer to synchronize taint *assertions* between the CTT and precise taint state, special measures must be taken to update the CTT when a taint domain is fully *cleared* of taint.

To track domains that might have been cleared, the CTC incorporates a *taint clear bit* for each taint domain bit. The taint clear bit is asserted whenever a byte in the corresponding domain is assigned a taint status of zero by a stnt instruction, and de-asserted when a non-zero taint status is written. Before returning control to hardware, the software component iterates through the precise representation of each domain whose clear bit is asserted, and updates the CTT entry for any domains that have been completely cleared of taint. To eliminate the need to store taint clear bits in memory, a check is also triggered (using a hardware exception) whenever a CTC word with asserted clear bits is evicted. This mechanism ensures that coarse



(a) Baseline parallel DIFT (b) P-LATCH organization

Figure 11: P-LATCH Architecture

taint is removed when the last precise taint bit in the corresponding domain is cleared.

5.2 *P-LATCH*: Accelerating Parallel Software DIFT

In addition to accelerating software DIFT on a single core, *LATCH* can provide significant performance gains to systems that perform DIFT in parallel. Figure 11-a shows a generalization of this scheme, in which two cores respectively host the monitored code (A) and the software security monitor (B). The monitored core includes simple logic to extract instructions from the hosted program and place them in a shared first-in-first-out (FIFO) queue (C). The monitor then reads events from the queue, and performs the requisite analysis and detection.

Parallelization for DIFT and other tag-based policies, similar to the baseline system described above, was introduced in the Log Based Architecture (LBA) proposal [6], and has been optimized in subsequent proposals using general-purpose cores and dedicated co-processors [6, 7, 14, 15, 25, 31, 34, 36, 39]. Existing systems significantly accelerate software-based DIFT, but still impose substantial (> 3x) performance overhead, or require complex, specialized hardware optimizations. This is because the shared queue can become saturated, forcing stalls [6]. As shown in Figure 11-b, *P-LATCH* addresses the problem of queue stalls by reducing the load on the shared queue. Instead of filtering invocations to the DBI engine, as in S-LATCH, the otherwise unmodified *LATCH* hardware module on the monitored core selectively enqueues instructions according to the coarse taint detection policy, ensuring that only instructions that might require monitoring populate the queue. Due to the temporal locality of taint-handling accesses, this policy ensures that the queue is empty – and thus stall-free – for significant periods of execution. Moreover, during periods of execution that contain numerous taint-handling instructions, *P-LATCH* can continue to provide acceleration by filtering operations that do not contain taint. In addition to reducing the queue stalls that cause overhead on the monitored core, this policy also increases computational throughput by freeing the monitoring core to execute other processes. *P-LATCH* implementations may require additional logic to prevent false negatives from arising due to outstanding CTT updates. This can be addressed by tracking the destination operands for queued events, and treating them as tainted until the coarse taint state is updated. A small FIFO-like structure could be used to track these operands. When taint is updated, a signal from the monitored core can pop the corresponding entries in the FIFO and invalidate any associated CTC lines if taint has been changed. Although this mechanism is not addressed in the current analytical model for *P-LATCH*, and could occasionally cause additional false positives, taint locality makes changes to the CTT rare, limiting the effects on performance.

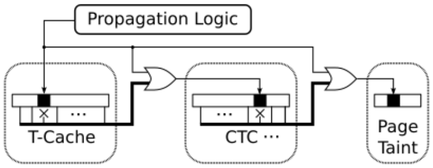


Figure 12: *H-LATCH* cache update logic

5.3 *H-LATCH*: Reducing Complexity in Hardware DIFT

In addition to providing performance benefits to software-based DIFT in single-core and paralleled contexts, *LATCH* can be integrated with hardware-based DIFT systems to significantly reduce architectural complexity. We refer to the resulting system as *H-LATCH*. *H-LATCH* is motivated by the observation that, in most hardware-based DIFT systems, the caching logic used for checking the taint status of memory operands is the single greatest contributor to architectural complexity of hardware-based DIFT. For instance, [54] requires a dedicated 4KB taint cache to perform word granularity checking with one-byte taint tags (or larger, if taint tags are extended). By resolving most taint checks using a lightweight, coarse-grained taint state, *H-LATCH* allows the precise taint cache to be significantly scaled down while maintaining high performance. *H-LATCH* consists of the core *LATCH* logic integrated into a generic DIFT architecture. For this implementation, we assume an architecture similar to [54], which performs all checking and propagation at the commit stage of the pipeline. This design avoids invasive modifications to the out-of-order portion of the pipeline, and is thus easily incorporated into existing architectures as a modular extension.

5.3.1 *Updating the Coarse Taint States.*

Similar to other *LATCH* implementations, *H-LATCH* must synchronize the coarse and precise taint states whenever a taint tag is updated. Figure 12 illustrates the logic that *H-LATCH* uses to ensure multi-granular taint-state integrity. At the beginning of each taint update, *H-LATCH* extracts the logical `and` of the taint bits in the pre-update taint state word, excluding the bits for the tag to be updated. The de-selection of the updated taint bit is performed by modified decoder logic using offset bits from the memory operand. When the new taint tag is computed, it is `and`ed with the masked word taint to produce the updated higher granularity taint bit. This operation can be chained for multiple levels of coarse tainting, allowing simultaneous updates to the CTC and page-level taint bits. This operation is comparable in delay to the peripheral logic for register file access. Since the operation occurs during the commit stage, it can be pipelined if necessary to remove it from the critical path. Similar to the logic described in Section 5.1.4, this mechanism ensures that a coarse-grain taint domain is marked as taint-free when the last taint tag within that domain is cleared.

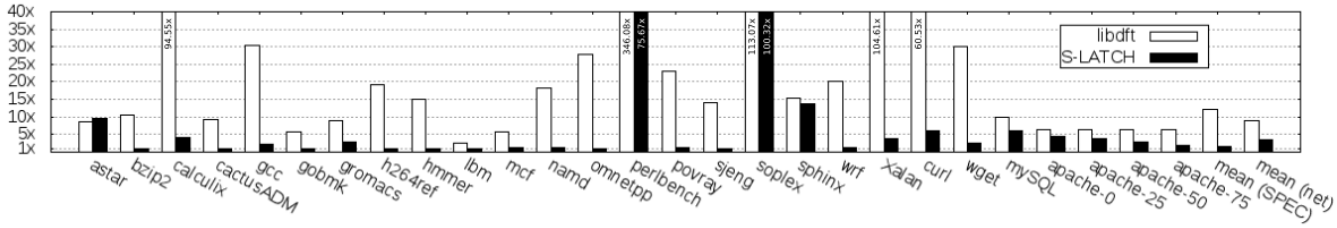


Figure 13: Performance overheads of software DIFT (libdft) and *S-LATCH* over native execution

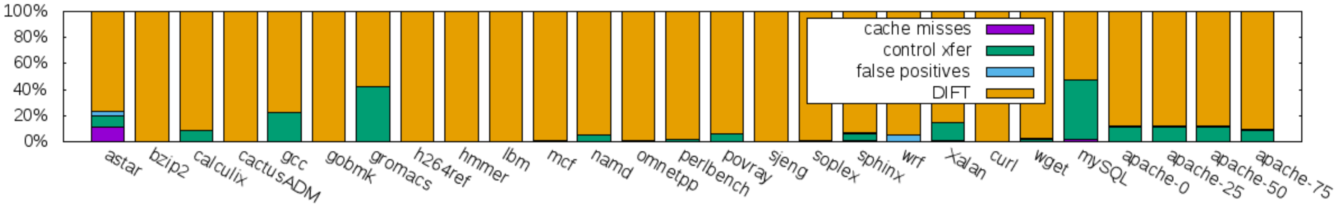


Figure 14: Sources of overhead in *S-LATCH*

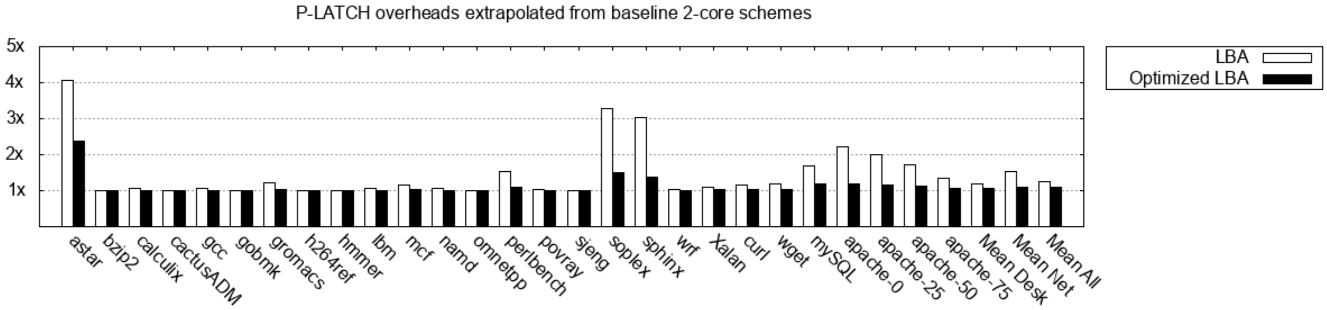


Figure 15: *P-LATCH* performance overheads relative to native execution

6 RESULTS AND DISCUSSION

This section evaluates performance and/or complexity advantages of *S-LATCH*, *P-LATCH*, and *H-LATCH*. Our performance evaluations were performed in Linux environment described in Section 3. Execution time estimates are based on a virtualized single-core system, with 3.4 GHz 32-bit x86 processor and 2GB of DRAM.

6.1 *S-LATCH* Evaluation

We evaluated *S-LATCH* using a simulation framework based on Intel Pin. Our simulator used libdft to propagate taint, and perfromed coase taint detection using a simulated CTC with a miss penalty of 150 cycles. Our simulator recorded the proportion of instructions executed under hardware and software monitoring, and assigned performance overheads accordingly. Overheads for software monitoring were based on the average slowdown for each benchmark when continuously monitored by a libdft-based Pintool implementing Dynamic Taint Analysis. Additionally, our framework included the overheads for reseting the coarse taint state, handling false positives, and transferring control between hardware and software modes. The transfer overheads arise from two sources: the storing and reloading of the native program context before and after each mode switch, and the cost of loading the current section (Pin *trace*) of the instrumented image from the Pin code cache. The former was based on timing information for the C functions `getcontext` and `setcontext`, which are used to store and load program context; the latter was determined for each benchmark based on the delay between the execution of two traces, which corresponds to Pin code-cache latency.

6.1.1 S-LATCH Performance for SPEC benchmarks. Figure 13 shows the overhead of *S-LATCH* and software-only libdft relative to execution without DIFT protections. When all SPEC benchmarks are considered, *S-LATCH* incurs a harmonic mean overhead of 60% across all evaluated benchmarks, with more than half (12 of 20) experiencing overheads of less than 50%. While direct comparison is challenging given the differing sets of benchmarks used, this overhead is significantly lower compared to most aggressively optimized software-based schemes [4, 44]. In addition, 8 benchmarks, experience overheads of less than 5%, which is close to the performance of hardware-based DIFT. Most of the programs that experienced higher overheads are those with poor temporal locality, as detailed in Section 3. When these programs are omitted, the mean overhead is 32%. On average, SPEC benchmarks experienced approximately a 4X speedup relative to software-based DIFT.

S-LATCH was able to accelerate both web clients by a factor of more than 10X relative to software-based DIFT. *mySQL* and baseline *Apache* experienced speedups of 63% and 47% respectively, relative to software-based DIFT. While the servers experienced relatively high overheads compared to execution without DIFT, [32] showed that network latency frequently masks DIFT overheads from the perspective of the client, and the speedups achieved represent a meaningful gain in the context of server-side resource utilization. Additionally, under the alternative tainting policies introduced in Section 3, Apache server achieved significantly higher speedups of up to 3.25X.

6.1.2 Sources of Overhead. Figure 14 shows that *libdft* instrumentation is the primary source of overhead in most programs, indicating that the hardware logic of *S-LATCH* performs taint detection and mode switches (shown as *control xfer*) efficiently. For a few programs, switches between the hardware and software layers contributed more to the overhead. Because *S-LATCH* accommodates a range of software-based DIFT policies, we note that *libdft* could be replaced with a more aggressively optimized scheme, such as LIFT or *minemu* [4, 44], to mitigate this source of overhead. In addition to hardware-software control transfers, false-positive checks and CTC misses contributed to *S-LATCH* overhead, but only exerted significant impacts on the performance of *astar*, in which poor spatial locality imposed unusual pressure on the coarse-grained tainting policy.

6.2 *P-LATCH* Evaluation

Next, we extended our performance model to evaluate the overhead of *P-LATCH* in a 2-core monitoring system. To assess *P-LATCH* overheads over native execution, we integrated the reported performance overheads of the baseline LBA 2-core monitor (obtained from the results presented in [7]) into our evaluation framework, and estimated performance with *LATCH* localizing the overheads to periods of active propagation, measured at 1000 instruction granularity. For comparison, we also modeled a *P-LATCH* integration with an optimized version of the LBA framework [7]. Since [6, 7] evaluated LBA on a different (older) benchmark suites, we used reported mean LBA overheads for our model's baseline. Though approximate, this model represents a conservative assessment of *S-LATCH* performance across the full set of applications, since it does not account for increased queue slack introduced by *P-LATCH* during periods of active taint propagation.

Figure 15 reports *P-LATCH* performance overheads for the applications in our evaluation set. For the simple integration with LBA, we observed mean overheads of 18.4% for the SPEC benchmarks and 52.4% for the network benchmarks, with a mean overhead of 25.7% across all applications. These values are significantly lower than the mean 3.38X overhead imposed by baseline LBA [7]. Overheads for the optimized schemes were still lower at 7.6% for SPEC 2006 benchmarks, 10.1% for network applications, and 0.8% overall, compared to 36% for the optimized scheme without *P-LATCH*([7]).

6.3 *H-LATCH* Evaluation

To evaluate *H-LATCH*, we extended the Pin-based framework used in the *S-LATCH* experiments with a simulated precise taint cache, and analyzed the effect of coarse-grained filtering on overall taint caching performance. Table 6 and Figure 16 present these results for SPEC 2006 benchmark suite. The cache miss rate of *H-LATCH* is the combined number of CTC and taint cache misses as a fraction of all memory accesses. This value did not exceeded 1% for any SPEC or network benchmark, except *astar* and *sphinx*. To contextualize these observations, we compared the results for *H-LATCH*'s filtered taint cache to a comparable taint cache without the benefits of *LATCH* filtering. As a result of filtering, *H-LATCH* eliminated over 89% of cache misses for SPEC benchmarks, and more than 98% for network applications. Indeed, for all programs except *astar* and sphinx, *H-LATCH* eliminated between 98 and 99.99% of all cache misses incurred by the taint cache without *LATCH* filtering.

To assess the internal dynamics of *H-LATCH*, we measured the number of memory accesses resolved by each element of *H-LATCH*'s taint caching stack: the TLB, CTC, and precise taint cache. Our results, presented in Figure 16, show varied distribution of access loads between the TLB and CTC. In most programs, the TLB deflected more than 90% of memory accesses. In *astar*, *gromacs*, *omnetpp*, and *Apache*, however, the CTC assumed a critical role in screening the precise taint cache. Unsurprisingly, *astar* and *sphinx* placed the heaviest burden on the taint cache, although in both cases *LATCH* logic screened the majority of memory accesses.

6.4 *LATCH* Complexity Analysis

In the *H-LATCH* integration presented above, *LATCH* uses a fully associative CTC with a 32 bit cache line, resulting in a capacity of 64 bytes. The precise taint cache used 32 bit blocks, with 4 ways and a 128 byte capacity. When combined with taint bits added to a 128-entry TLB, the *H-LATCH* caching system has a total capacity of 320 bytes to achieve byte-granular tainting. In the *S-LATCH* and *P-LATCH* integrations, *LATCH* uses a 16-entry (64 bytes of total capacity) fully-associative CTC, with a 64-byte taint domain, and the TLB is correspondingly extended with two page-level taint bits. The resulting system has a total capacity of 160B, including the CTC clear bits.

To assess impacts on the area, power, and time of a real architecture, we integrated the core *LATCH* hardware with an open source AO486 processor implemented on an FPGA. The AO486 processor is a 32-bit, in-order, pipelined, 33MHz implementation of the Intel 80486 ISA. We synthesized this combined architecture on a DE2-115 FPGA using Quartus II 17.1 software, with *LATCH* attached to the back-end stage of the core.

The results show that *LATCH* indeed has a low complexity. In terms of area, *LATCH* increases the total logic elements and total memory bits by 4% and 5%, respectively. In terms of power, it increases core dynamic and static power by 5% and 0.2%, respectively. Although it is difficult to measure power accurately, we applied the power analysis tool provided by Quartus to measure power after synthesis to get more accurate results. Moreover, since *LATCH* design is simple, it fits within the optimized frequency of the core. Thus, it has no effect on cycle time. While we used the in-order

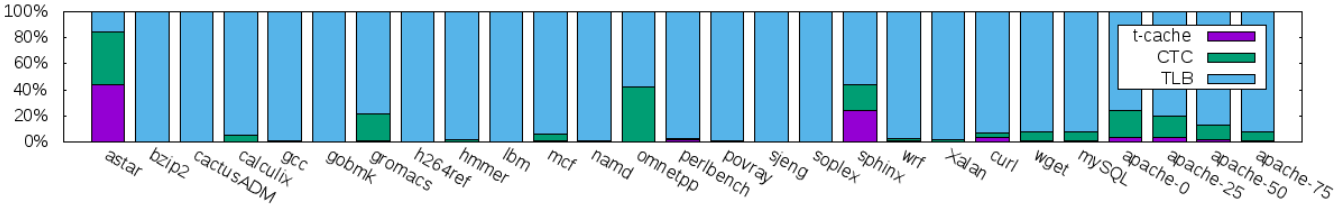


Figure 16: Percentage of memory accesses handled by each taint caching element in *H-LATCH*

	astar	*bzip2*	*cactusADM*	*calculix*	*gcc*	*gobmk*	*gromacs*	*h264ref*	*hmmer*	*lbm*	*mcf*	*namd*	*omnetpp*	*perlbench*	*povray*	*sjeng*	*soplex*	*sphinx*	*wget*	*wrf*	*Xalan*	*mean*
CTC miss percentage	2.622	0.0001	0.0001	0.0001	0.0008	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0034	0.0001	0.0001	0.0001	0.2872	0.0004	0.0035	0.0141	0.0001
t-cache miss percent in H-LATCH	2.8894	0.0001	0.0001	0.0025	0.0037	0.0001	0.0044	0.0002	0.0001	0.0026	0.0024	0.0008	0.0001	0.0469	0.0017	0.0001	0.0001	2.0087	0.0055	0.0274	0.0124	0.0003
combined miss percent in H-LATCH	5.5114	0.0001	0.0001	0.0025	0.0045	0.0001	0.0044	0.0002	0.0001	0.0026	0.0024	0.0008	0.0001	0.0503	0.0017	0.0001	0.0001	2.2959	0.0058	0.0309	0.0265	0.0003
t-cache miss percent without LATCH	7.9707	5.3137	25.364	10.3279	11.3298	11.3462	5.0965	6.9702	7.39	23.6281	35.6878	12.1935	12.3787	16.4413	10.0139	15.0817	13.5815	11.3727	7.0173	16.4611	13.4061	10.4956
percent misses avoided by H-LATCH	30.8541	99.9995	99.9999	99.9758	99.9604	99.9991	99.913	99.9977	99.9999	99.9891	99.9933	99.9932	99.9997	99.6939	99.9829	99.9999	99.9999	79.8126	99.9168	99.8125	99.8022	89.3475

Table 6: *H-LATCH* cache performance for SPEC 2006 benchmarks

	apache-0	*apache-25*	*apache-50*	*apache-75*	*curl*	*mySQL*	*wget*	*mean*
CTC miss percentage	0.0632	0.0454	0.0305	0.0141	0.0022	0.0722	0.0003	0.0018
t-cache miss percent in H-LATCH	0.1528	0.1365	0.0713	0.0371	0.0817	0.0544	0.0055	0.0262
combined miss percent in H-LATCH	0.2159	0.1818	0.1018	0.0511	0.0839	0.1266	0.0059	0.0306
t-cache miss percent without LATCH	10.6789	10.7884	10.7945	10.8036	5.8689	11.6442	6.9646	9.0745
percent misses avoided by H-LATCH	97.9779	98.3146	99.0569	99.5267	98.5707	98.9128	99.9157	98.8925

Table 7: *H-LATCH* cache performance for network applications

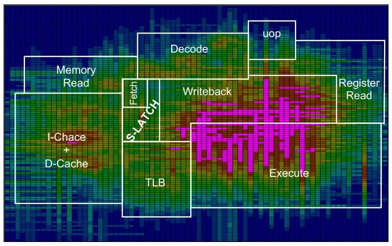


Figure 17: *LATCH* integrated into AO486 processor core

AO486 processor for demonstration purposes, *LATCH* can be integrated to out-of-order designs without additional complexity, as its extraction logic applies to committed instructions. These results are relative to the small AO486 pipelined core. The overheads will be much smaller if compared to a modern out-of-order superscalar core, because LATCH hardware will represent a much smaller percentage

7 RELATED WORK

In addition to using temporal locality to limit taint propagation and checks, *LATCH* exploits spatial locality of taint as part of its coarse-grained tainting policy. In this regard, *LATCH* advances a line of research previously developed by the RangeCache system [49]. In contrast to *LATCH*, which exploits spatial locality to act as a generalizable filter for a finer-grained tainting system, RangeCache is a hardware-based tagging scheme that uses range-based compression to reduce the size of a single-granularity taint cache. Although the proof-of-concept implementation of *H-LATCH* was evaluated in the context of a conventional taint cache, the use of multigranularity tainting to further reduce the complexity of RangeCache and other compressed caching schemes is a promising area of research for future work.

Another recent proposal applied a compression mechanism based on multi-granularity tainting to a single-granularity last-level taint cache [29]. Unlike *LATCH*, this system caches taint tags for the last-level cache, and thus required a significant investment of hardware. In addition to the 32KiB last-level taint-cache itself, this system required the extension data caches at each level with additional tag bits. Moreover, the implementation presented in [29] tracks taint tags at a coarse granularity of 256 bits, significantly more hardware would be required to implement byte-precise tainting.

To our knowledge, *LATCH* is the first practical system to exploit the inherent temporal properties of DIFT. The work of [26] used page granularity taint tagging to activate an emulation-based taint tracker, but relied on a modified VM to artificially enhance the locality of tainted data, and was evaluated for a restricted taint tracking policy that omitted DIFT propagation logic. The authors of [26] observed that reliance on virtualization and emulation still make the performance overhead too large to be practically usable. *LATCH* advances this line of research by defining and exploiting patterns of locality inherent to DIFT itself, and by specifying lightweight hardware logic to support practical locality-based DIFT acceleration with reasonable overhead. PAGARUS [42] is a recent work that pursues a different form a coarse-grained tainting that propagates taint from the inputs to the outputs of hardware accelerators, rather than at the granularity of individual instructions.

The work of [24] proposed a mechanism for triggering, disabling, or changing security monitoring based on high-level events, such as user authentication. In contrast, *LATCH* exploits intrinsic localities in DIFT information flows. The two mechanisms are thus orthogonal and complementary: for instance, *LATCH* can accelerate pre-authentication monitoring, until the technique of [24] disables DIFT entirely after authentication. The idea of approximate taint tracking was explored in [56], which proposed PIFT - a scheme that propagates taint through consecutive load and store operations without tracking taint through intermediate registers. In comparison, *LATCH* supports precise taint tracking, but can also enhance the taint-caching in approximate schemes such as PIFT.

S-LATCH shares the objective of accelerating software-based DIFT with several prior proposals. LIFT [44] and Minemu [4] apply optimizations within DBI itself, achieving mean overheads 4X and 2.6X, respectively, over native execution. Though impressive, these overheads remain a deterrent to production deployment. Minemu's aggressive software-based optimizations are orthogonal to *S-LATCH*, and require that the SSL registers are consistently available for storage of taint metadata, a strategy which becomes problematic as more programs take advantage of them. Software-based DIFT schemes developed for mobile platforms also exhibit impressive properties [22], but the overhead of running these systems is largely hidden because DIFT checks are built into a virtualized system, and would not pertain to natively executed code. Some proposals introduced DIFT logic statically at compile-time [33]. This approach improves performance but cannot be used with COTS binaries.

P-LATCH relates to recent efforts to parallelize software-based DIFT using general-purpose cores. Simple implementations of multiprocessor DIFT, as embodied in LBA[6, 7], impose overheads on DIFT in excess of 3X, while more complex systems require significant additional hardware [7]. A proposal by lo et al. [36] allows parallel taint tracking to be temporarily disabled to allow hard real-time systems to achieve deadlines, but provides no security guarantees during non-monitored periods. A related proposal [35] augments this selective monitoring approach by using limited (although spatially fine-grained) hardware checks to limit comprehensive monitoring, thus conserving the budget of cycles allocated for detailed enforcement. These approaches are ideal for applications with stringent limitations on monitoring overheads, but unlike *LATCH* implementations are not designed to support comprehensive monitoring.

Among existing proposals, FADE [23] most closely resembles *P-LATCH* in its use of hardware checks to filter security events processed by a second-layer monitor. However, the FADE hardware module is a fine-grained mechanism that is closer in complexity to traditional hardware systems, requiring, for instance, a 4KB metadata cache. Other related approaches involve specialized external modules or co-processors [16, 17, 31, 34] or a reconfigurable fabric [15]. These mechanisms are strongly optimized for operations on tagged metadata, but can not be re-purposed for general computation.

Existing hardware-based taint-tracking systems [12, 23, 27, 48, 51, 54] provide excellent performance, but incur substantial design complexity which *H-LATCH* seeks to address. Comparatively lightweight implementations such as [54] and [23] still require a 4KB taint cache to check taint tags for all operands at precise granularity, while other designs introduce additional complexities, such as the extension of memory buses and DRAM [12].

8 CONCLUDING REMARKS

This paper introduced *LATCH*, a lightweight hardware module that exploits temporal locality to reduce the architectural and performance overheads of DIFT. Following an in-depth analysis of the temporal and spatial features of DIFT dataflows, we described the *LATCH* architecture and applied it to three DIFT implementations. We demonstrated significant reductions in performance overhead for single-core and multi-core implementations of software DIFT(*S-LATCH* and *P-LATCH*), as well as a decrease in the architectural complexity needed to achieve acceptable taint-cache performance in hardware-based DIFT (*H-LATCH*). These examples demonstrate that locality-aware taint checking is a viable and generalizable optimization substrate, enabling more practical and efficient DIFT solutions.

ACKNOWLEDGMENTS

The work in this paper is partially supported by National Science Foundation grants CNS-1422401, CNS-1617915 and CNS-1619322.

REFERENCES

[1] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, Vol. 13.

[2] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 30–40.

[3] Erik Bosman, Asia Slowinska, and Herbert Bos. 2011. Minemu: The world's fastest taint tracker. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 1–20.

[4] Erik Bosman, Asia Slowinska, and Herbert Bos. 2011. Minemu: The World's Fastest Taint Tracker.. In *RAID*, Vol. 11. Springer, 1–20.

[5] Derek Bruening, Qin Zhao, and Saman Amarasinghe. 2012. Transparent Dynamic Instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE '12)*. ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/2151024.2151043

[6] Shimin Chen, Babak Falsafi, Phillip B Gibbons, Michael Kozuch, Todd C Mowry, Radu Teodorescu, Anastassia Ailamaki, Limor Fix, Gregory R Ganger, Bin Lin, et al. 2006. Log-based architectures for general-purpose monitoring of deployed code. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. ACM, 63–65.

[7] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B Gibbons, Todd C Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. 2008. Flexible hardware acceleration for instruction-grain program monitoring. In *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*. IEEE, 377–388.

[8] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. 2006. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Computers and Communications, 2006. ISCC'06. Proceedings. 11th IEEE Symposium on*. IEEE, 749–754.

[9] Andrey Chudnov and David A Naumann. 2015. Inlined information flow monitoring for javascript. In *2015ACM SIGSAC conference on Computer and Communications Security*.

[10] James Clause, Ioannis Doudalis, Alessandro Orso, and Milos Prvulovic. 2007. Effective memory protection using dynamic tainting. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 284–292.

[11] Jedidiah R Crandall and Frederic T Chong. 2004. Minos: Control data attack prevention orthogonal to memory model. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*. IEEE, 221–232.

[12] Michael Dalton, Hari Kannan, and Christos Kozyrakis. 2007. Raksha: a flexible information flow architecture for software security. In *ACM SIGARCH Computer Architecture News*, Vol. 35. ACM, 482–493.

[13] Michael Dalton, Hari Kannan, and Christos Kozyrakis. 2008. Real-World Buffer Overflow Protection for Userspace and Kernelspace.. In *USENIX Security Symposium*. 395–410.

[14] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. 2015. HAFIX: Hardware-assisted flow integrity extension. In *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 74.

[15] Daniel Y Deng, Daniel Lo, Greg Malysa, Skyler Schneider, and G Edward Suh. 2010. Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 137–148.

[16] Daniel Y Deng and G Edward Suh. 2012. High-performance parallel accelerator for flexible and efficient run-time monitoring. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*. IEEE, 1–12.

[17] Udit Dhawan, Nikos Vasilakis, Raphael Rubin, Silviu Chiricescu, Jonathan M Smith, Thomas F Knight Jr, Benjamin C Pierce, and André DeHon. 2014. Pump: a programmable unit for metadata processing. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 8.

[18] Ashutosh S Dhodapkar and James E Smith. 2003. Comparing program phase detection techniques. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 217.

[19] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. 2007. Dynamic Spyware Analysis. In *Procedings of the2007 USENIX Annual Technical Conference (ATC'07)*. USENIX Association, Berkeley, CA, USA, 18:1—-18:14. http://dl.acm.org/citation.cfm?id=1364385.1364403

[20] Jesse Elwell, Ryan Riley, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Iliano Cervesato. 2016. Rethinking memory permissions for protection against cross-layer attacks. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 4 (2016), 56.

[21] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Trans. Comput. Syst.* 32, 2 (jun 2014), 5:1—-5:29. https://doi.org/10.1145/2619091

[22] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 5.

[23] Sotiria Fytraki, Evangelos Vlachos, Onur Kocberber, Babak Falsafi, and Boris Grot. 2014. FADE: A programmable filtering accelerator for instruction-grain monitoring. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 108–119.

[24] Dimitris Geneiatakis, Georgios Portokalidis, Vasileios P Kemerlis, and Angelos D Keromytis. 2012. Adaptive defenses for commodity software through virtual application partitioning. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 133–144.

[25] Ingoo Heo, Minsu Kim, Yongje Lee, Changho Choi, Jinyong Lee, Brent Byunghoon Kang, and Yunheung Paek. 2015. Implementing an application-specific instruction-set processor for system-level dynamic program analysis engines. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 20, 4 (2015), 53.

[26] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. 2006. Practical taint-based protection using demand emulation. In *ACM SIGOPS Operating Systems Review*, Vol. 40. ACM, 29–41.

[27] Wei Hu, Jason Oberg, Ali Irturk, Mohit Tiwari, Timothy Sherwood, Dejun Mu, and Ryan Kastner. 2012. On the complexity of generating gate level information flow tracking logic. *IEEE Transactions on Information Forensics and Security* 7, 3 (2012), 1067–1080.

[28] Kangkook Jee, Vasileios P Kemerlis, Angelos D Keromytis, and Georgios Portokalidis. 2013. ShadowReplica: efficient parallelization of dynamic data flow tracking. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 235–246.

[29] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W Moore, Alex Bradbury, Hongyan Xia, Robert NM Watson, David Chisnall, Michael Roe, Brooks Davis, et al. 2017. Efficient Tagged Memory. In *Computer Design (ICCD), 2017 IEEE International Conference on*. IEEE, 641–648.

[30] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation.. In *NDSS*.

[31] Hari Kannan, Michael Dalton, and Christos Kozyrakis. 2009. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, 105–114.

[32] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. 2012. libdft: Practical dynamic data flow tracking for commodity systems. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 121–132.

[33] Lap Chung Lam and Tzi-cker Chiueh. 2006. A general dynamic information flow tracking framework for security applications. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, 463–472.

[34] Jinyong Lee, Ingoo Heo, Yongje Lee, and Yunheung Paek. 2015. Efficient dynamic information flow tracking on a processor with core debug interface. In *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 79.

[35] Daniel Lo, Tao Chen, Mohamed Ismail, and G Edward Suh. 2015. Run-time monitoring with adjustable overhead using dataflow-guided filtering. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 662–674.

[36] Daniel Lo, Mohamed Ismail, Tao Chen, and G Edward Suh. 2014. Slack-aware opportunistic monitoring for real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*. IEEE, 203–214.

[37] Chi-Keung Luk, B C Ed, F C G Hi, E D Q Rs, A Tu, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation - PLDI '05* 40, 6 (2005), 190. https://doi.org/10.1145/1065010.1065034

[38] James Newsome and Dawn Song. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. (2005).

[39] Edmund B Nightingale, Daniel Peek, Peter M Chen, and Jason Flinn. 2008. Parallelizing security checks on commodity hardware. In *ACM Sigplan Notices*, Vol. 43. ACM, 308–318.

[40] Meltem Ozsoy, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Tameesh Suri. 2011. SIFT: A low-overhead dynamic information flow tracking architecture for smt processors. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*. ACM, 37.

[41] Avadh Patel and Kanad Ghose. 2008. Energy-efficient mesi cache coherence with pro-active snoop filtering for multicore microprocessors. In *Proceedings of the 2008 international symposium on Low Power Electronics & Design*. ACM,

247–252.

[42] Luca Piccolboni, Giuseppe Di Guglielmo, and Luca P Carloni. 2018. PAGURUS: Low-Overhead Dynamic Information Flow Tracking on Loosely Coupled Accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2685–2696.

[43] Joël Porquet and Simha Sethumadhavan. 2013. WHISK: An uncore architecture for dynamic information flow tracking in heterogeneous embedded SoCs. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. IEEE Press, 4.

[44] F Qin, C Wang, Z Li, H s. Kim, Y Zhou, and Y Wu. 2006. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 135–148. https://doi.org/10.1109/MICRO.2006.29

[45] Julian Seward. [n. d.]. Technical Overview of bzip2. http://www.bzip.org.

[46] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. ACM, New York, NY, USA, 552–561. https://doi.org/10.1145/1315245.1315313

[47] Jangseop Shin, Hongce Zhang, Jinyong Lee, Ingoo Heo, Yu-Yuan Chen, Ruby Lee, and Yunheung Paek. 2016. A hardware-based technique for efficient implicit information flow tracking. In *Computer-Aided Design (ICCAD), 2016 IEEE/ACM International Conference on*. IEEE, 1–7.

[48] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. 2004. Secure program execution via dynamic information flow tracking. In *Acm Sigplan Notices*, Vol. 39. ACM, 85–96.

[49] Mohit Tiwari, Banit Agrawal, Shashidhar Mysore, Jonathan Valamehr, and Timothy Sherwood. 2008. A small cache of large ranges: Hardware methods for efficiently searching, storing, and updating big dataflow tags. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 94–105.

[50] Mohit Tiwari, Hassan MG Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T Chong, and Timothy Sherwood. 2009. Complete information flow tracking from the gates up. In *ACM Sigplan Notices*, Vol. 44. ACM, 109–120.

[51] Mohit Tiwari, Hassan MG Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T Chong, and Timothy Sherwood. 2009. Complete information flow tracking from the gates up. In *ACM Sigplan Notices*, Vol. 44. ACM, 109–120.

[52] Daniel Townley and Dmitry Ponomarev. 2019. SMT-COP: Defeating Side-Channel Attacks on Execution Units in SMT Processors. In *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques*. ACM.

[53] N Vachharajani, M J Bridges, J Chang, R Rangan, G Ottoni, J A Blome, G A Reis, M Vachharajani, and D I August. 2004. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*. 243–254. https://doi.org/10.1109/MICRO.2004.31

[54] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. 2008. Flexitaint: A programmable accelerator for dynamic taint propagation. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. IEEE, 173–184.

[55] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: capturing system-wide information flow for malware detection and analysis. In *Procedings of the14thACM SIGSAC conference on Computer and Communications Security*. ACM, 116–127.

[56] Man-Ki Yoon, Negin Salajegheh, Yin Chen, and Mihai Christodorescu. 2016. PIFT: Predictive Information-Flow Tracking. *SIGPLAN Not.* 51, 4 (March 2016), 713–725. https://doi.org/10.1145/2954679.2872403

[57] David Yu Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. 2011. TaintEraser: Protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review* 45, 1 (2011), 142–154.