LDX: Causality Inference by Lightweight Dual Execution

Yonghwi Kwon[1] Dohyeong Kim[1] William N. Sumner[2] Kyungtae Kim[1]
Brendan Saltaformaggio[1] Xiangyu Zhang[1] Dongyan Xu[1]

[1]Department of Computer Science, Purdue University, USA
[2]School of Computing Science, Simon Fraser University, Canada
[1]{kwon58,kim1051,kim1798,bsaltafo,xyzhang,dxu}@purdue.edu [2]wsumner@sfu.ca

Abstract

Causality inference, such as dynamic taint anslysis, has many applications (e.g., information leak detection). It determines whether an event *e* is causally dependent on a preceding event *c* during execution. We develop a new causality inference engine LDX. Given an execution, it spawns a slave execution, in which it mutates *c* and observes whether any change is induced at *e*. To preclude nondeterminism, LDX couples the executions by sharing syscall outcomes. To handle path differences induced by the perturbation, we develop a novel on-the-fly execution alignment scheme that maintains a counter to reflect the progress of execution. The scheme relies on program analysis and compiler transformation. LDX can effectively detect information leak and security attacks with an average overhead of 6.08% while running the master and the slave concurrently on seperate CPUs, much lower than existing systems that require instruction level monitoring. Furthermore, it has much better accuracy in causality inference.

Keywords Causality Inference, Dual Execution, Dynamic Analysis

1. Introduction

Causality inference during program execution determines whether an event is causally dependent on a preceding event. Such events could be system level events (e.g., input/output syscalls) or individual instruction executions. A version of causality inference, *dynamic tainting*, is widely used to detect *information leak*, namely, sensitive information is undesirably disclosed to untrusted entities, and *runtime attacks*, in which exploit inputs subvert critical execution state such as stack and heap (Qin et al. 2006; Song et al. 2008; Kemerlis et al. 2012; Clause et al. 2007; Bosman et al. 2011).

Most existing causality inference techniques are based on program dependences, especially data dependences. There is data dependence between two events if the former event defines a variable and the later event uses it. These techniques have a few limitations. *First*, they have difficulty in handling control dependence. There is control dependence between a predicate and an instruction if the predicate directly determines whether the instruction executes. The challenge lies in that control dependences sometimes lead to strong causality, but some other times lead to very weak causality that cannot be exploited by attackers and hence should not be considered. Most existing solutions (McCamant and Ernst 2008; Kang et al. 2011; Cox et al. 2014) rely on detecting syntactic patterns of control dependences and hence are incomplete. *Second*, existing techniques are expensive (e.g., a few times slow-down (Kemerlis et al. 2012)), as memory accesses need to be instrumented to detect data dependences. *Third*, the complexity in implementation is high. Dependence tracking logic needs to be defined for each instruction, which is error-prone for complex instruction sets. Instrumenting third party libraries, various languages and their runtimes, is very challenging.

We observe that these limitations root at tracking causality by monitoring program dependencies. We propose to directly infer causality based on its defintion. In (Lewis 1973), *counterfactual causality* was defined as follows. An event *e* is causally dependent on an earlier event *c* if and only if the absence of *c* also leads to the absence of *e*. Program dependence tracking in some sense just approximates counterfactural causality. Our technique works as follows. It perturbs the program state at *c* (the *source*) and then observes whether there is any change at *e* (the *sink*). There are a number of challenges. (1) We need at least two executions to infer causality. Thus, we must prune the differences caused by nondeterminism such as different external event orders. (2) Meaningful comparison of states across executions requires execution alignment. Due to perturbation, the event *e* may occur at different locations. Naive approaches such as using program counters hardly work due to path differences (Xin et al. 2008). (3) The second execution is not a simple replay of the first one, as the perturbation may cause path differences and then input/output syscall differences. (4) Ideally, the two executions should proceed in parallel. Otherwise, the execution time is at least doubled.

The core of our technique is a novel runtime engine LDX, which stands for *Lightweight Dual eXecution*. Its execution model is similar to *Dual Execution* (DualEx) (Kim et al. 2015). Given an original execution (the *master*), a new execution (the *slave*) is derived by mutating the source(s). Later, by comparing the output buffer contents of the two executions at the sink(s), we can determine if the sink(s) are causally dependent on the source(s). The master and the slave are coupled and run concurrently. The slave tries to reuse syscall and nondeterministic instruction outcomes (e.g., `rdtsc`) from the master to avoid nondeterminism. To avoid side effects, the slave often ignores output syscalls. Since perturbation may cause path differences and hence syscall differences, an on-the-fly execution alignment scheme is necessary. DualEx has a very expensive alignment scheme based on *Execution Indexing* (Xin et al. 2008). The slow-down reported in (Kim et al. 2015) is three orders of magnitude. In contrast, LDX features a novel lightweight *on-the-fly alignment scheme* that maintains a *counter* that reflects the progress of execution. The counter is computed in such a sophisticated way

ASPLOS '16 April 2–6, 2016, Atlanta, Georgia, USA.
Copyright © 2016 ACM 978-1-4503-4091-5/16/04. . . $15.00
DOI: http://dx.doi.org/10.1145/2872362.2872395

that an execution with a larger counter value must be ahead of another with a smaller one. The slave blocks if it reaches a syscall earlier than the master. If different paths are taken in the executions, the scheme can detect them and instructs the executions to perform their syscalls independently. It also allows the executions to realign by ensuring that they have the same counter value at the join point of the different paths. *Without such fine-grained alignment, when the slave encounters a syscall different from that in the master, it cannot decide if the master is running behind (so that it can simply wait) or the two are taking different paths so that the syscall will never happen in the master.*

Our contributions are summarized in the following.

- We study the limitations of program dependence based causality and propose counterfactual causality instead.
- We develop a lightweight dual execution engine that enables practical counterfactual causality inference.
- We develop a novel scheme that computes a counter cost-effectively at runtime using simple arithmetic operations. The counter values from multiple executions indicate their relative progress, facilitating runtime alignment. The scheme handles language features such as loops, recursion, and indirect calls.
- Our evaluation shows that LDX outperforms existing program dependence based dynamic tainting systems LIBDFT (Kemerlis et al. 2012) and TAINTGRIND (Khoo 2013). In the effectiveness aspect, LIBDFT and TAINTGRIND can only detect 31.47% and 20% of the true information leak cases and attacks detected by LDX. Also, LDX does not report any false warnings. In the efficiency aspect, the overhead of LDX is 6.08% to the original execution while it requires running the master and the slave concurrently on two separate CPUs. In contrast, the other two cause a few times slowdown although they do not require the additional CPU and memory. Note that the counter scheme allows aligning and continuing executions in the presence of path differences, which makes LDX superior to TIGHTLIP (Yumerefendi et al. 2007), which often terminates when it detects misaligned syscalls.

Limitations. LDX requires access to source code. Specifically, the target application should be compiled with LLVM because our analysis and instrumentation techniques are implemented in a LLVM pass. LDX occupies more resources than a single execution. In the worst case scenario, it may double the resource consumption on memory, processor, and external resources such as files on disk. Our performance evaluations assume that the machine has enough capacity to accomodate such resource duplication. In practice, if the slave and the master executions are coupled most of the time, only the processor and memory consumptions are doubled because the slave can shares most external I/Os with the master. LDX may have false positives. For example, low level data races that are not protected by any locks may induce non-deterministic state differences and eventually lead to undesirable output differences. However, for shared memory accesses protected by locks, LDX ensures the same synchronization order across the master and the slave. Furthermore, heap addresses are non-deterministic across the two runs, if heap pointer values are emitted as part of the output, LDX reports causality even though the two pointers may be semantically equivalent. However, In our experience, pointer values are rarely printed as part of the outputs at sink points.

LDX may also have false negatives. The current implementation may not capture causality through covert channels. For example, information can be disclosed through execution time and file metadata (e.g. last accessed time). We will leave it to our future work. Furthermore, program execution may run into extremal conditions (e.g., running out of disk/memory space), the current implementation of LDX does not handle such conditions.

2. Counterfactual Causality

Counterfactual causality (CC) (Hume 1748; Lewis 1973) is the earliest and the most widely used definition of causality: an event e is causally dependent on an event c if and only if, if c were not to occur, e would not occur. Later, researchers also introduce the notion of causal strength: c is a *strong cause* if and only if it is the necessary and sufficient condition of e (Miller and Johnson-Laird 1976; Kushnir and Gopnik 2005; Cheng 1997). Otherwise, c is a weak cause.

We adapt the definition in the context of program and program execution as follows. *Given an execution, we say a variable y at an execution point j is causally dependent on a variable x at an earlier point i, if and only if mutating x at i will cause change of y at j.* The causality is strong if and only if any change to x must lead to some change of y. We call this causality a *one-to-one mapping*. The causality is weak if multiple x values lead to the same y value. We call it a *many-to-one mapping*. The strength of the causality is determined by how many x values map to the same y.

Most existing causality inference techniques including dynamic tainting are based on tracking program dependences, especially data dependencies. Two events are causally related if there is a dependence path between them during execution. As we discussed in Section 1, these techniques have inherent limitations because *program dependences are merely approximation of CC*. Next, we discuss the relation between CC and dynamic program dependences to motivate our design.

(1) *Most Data Dependences Are Essentially Strong CCs.* Consider Fig. 1 (a). There is a strong CC between s at the source (line 1) and x at the sink (line 4) as any change to s leads to some change at the sink, and there is a data dependence path $4 \rightarrow 3 \rightarrow 1$ between the two. Other data dependences have similar characteristics, which implies that conventional dynamic tainting (based on data dependence) tracks strong CCs. On the other hand, if there is a technique that infers all strong CCs, it must subsume dynamic tainting.

(2) *Control Dependences Induce Both Strong and Weak CCs*. In Fig. 1 (b), assume the true branch is taken and $x = 1$. We can infer that s must be 10; there is strong causality between x and s. This strong CC is induced by the control dependence $14 \rightarrow 13$, together with data dependences $15 \rightarrow 14$ and $13 \rightarrow 11$. If control dependence is not tracked (like in most existing dynamic tainting techniques), the CC is missed. However in many cases, control dependences only lead to weak CC. In case (c), assume $s = 50$ and hence $x = 1$. There is a dependence path $25 \rightarrow 24 \rightarrow 23 \rightarrow 21$ if control dependence $24 \rightarrow 23$ is tracked. However, the casuality between x at 25 and s at 21 is weak as many values of s lead to the same $x = 1$. Such weak causality is very difficult for the attacker to exploit. For example with $x = 1$, the adversary can hardly infer s's value, even with the knowledge of the program. Moreover in code injection attacks, the attacker can hardly manipulate the sink (e.g. function return address) by changing the source. According to (Bao et al. 2010), if control dependences are not tracked, 80% strong CCs are missed; if all control dependences are tracked, strong CCs are never missed but 45% of the detected causalities are weak. In some large programs, an output event is causally dependent on almost all inputs with 90% of them being weak causalities that cannot be exploited. In summary, control dependences are a poor approximation of strong CCs.

(3) *Tracking both Data and Control Dependences May Still Miss Strong CCs.* Fig. 1 (d) presents such a case. Assume $s = 10$ and hence the else branch is executed. As such, x is not updated. However, the fact that x is not updated (and hence has the value of 0) allows the adversary to infer $s = 10$. It is a strong CC: any change

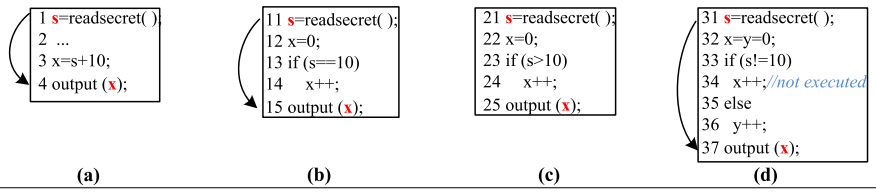


Figure 1. Examples to illustrate the comparison of counterfactual causality and program dependences. Arrows denote strong causalities between x at the sink and s at the source. Case (a) shows strong CC by data dependence; (b) shows strong CC by control dependence; (c) shows control dependence does not imply (strong) CC; (d) strong CC missed by both data and control dependences.

to s makes x have a different value. Unfortunately, such strong CC cannot be detected by tracking program dependences as line 37 is only data dependent on line 32 as the true branch is not executed. More cases are omitted due to the space limitations. They can be found in our technical report (tr).

The above discussion suggests that program dependences are a poor approximation of strong CCs. Hence, we propose LDX, a cost-effective technique that allows us to directly infer strong CCs, strictly following the definition.

3. Overview and Illustrative Example

We use an example to illustrate LDX. Here we are interested in information leak detection. We mutate the secret inputs. If output differences are observed at the sinks, there are strong CCs between the sinks and the secret inputs, and hence leaks.

Specifically, given the master execution, LDX creates a slave and runs the two concurrently in a closely coupled fashion. The master interacts with the environment and records its syscall outcomes. In most cases, the slave does not interact with the environment, but reuses the master's syscall outcomes, to eliminate state differences caused by nondeterministic factors such as external event orders. The slave mutates the sources, which potentially leads to path differences and hence syscall differences. A novel feature of LDX is to tolerate syscall differences in a cost-effective manner. It maintains a counter for each execution that indicates the progress. Execution points (across runs) with the same counter value and the same PC are guaranteed to *align* (in terms of control flow). An execution with a larger counter value is ahead of another with a smaller value. Aligned syscalls can share their outcomes; if the slave encounters a syscall with a counter larger than that in the master, the slave blocks until the master catches up; if the slave encounters an input syscall that does not have an alignment in the master, it will execute the syscall independently. The counter is computed as follows. It is incremented by 1 at each syscall. When two executions take different branches of a predicate –since the branches may have different numbers of syscalls– the values added to the counter may be different. The technique compensates the counter in the branch that has a smaller increment so that the counter must have the same value when the join point of the branches is reached. As such, the executions are re-synchronized.

Cnt	Master	Cnt	Slave	Action
1	1. read();	1	1. read();	M exec, S. copies
2	2. open();	2	2. open();	M exec, S. copies
3	12. open();	3	12. open();	M and S exec
4	13. read();	4	13. read();	M and S exec
		5	17. write();	M waits, S exec
		6	7. read();	M waits, S exec
7	11. send();	7	11. send();	Compare

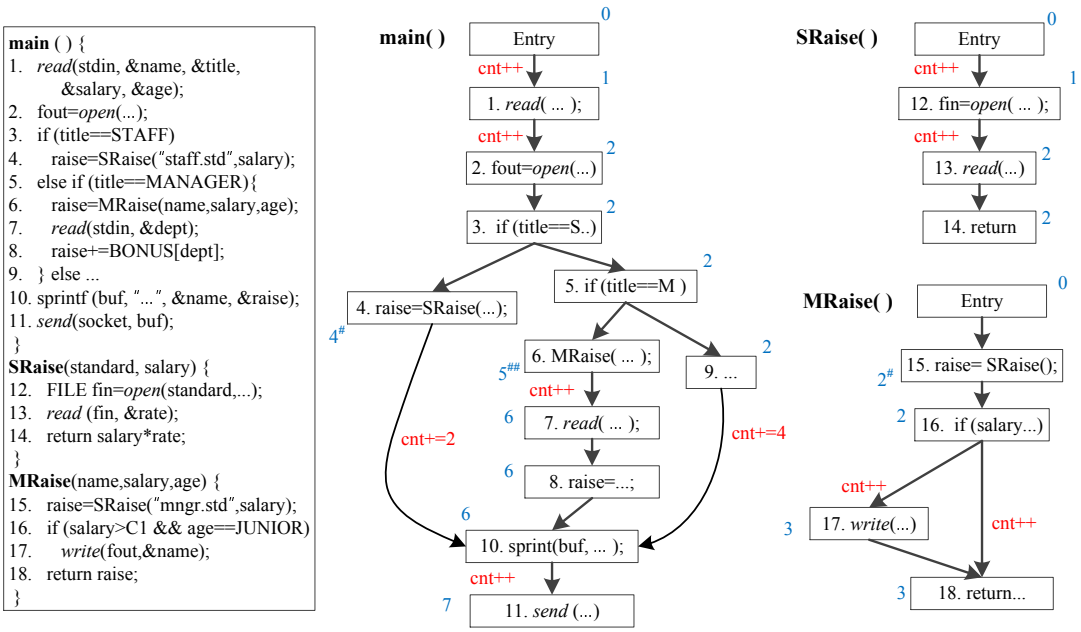
Figure 3. Syscall traces and the synchronization action sequence by LDX for the example in Fig. 2 with `title` the secret. The shaded entries are aligned.

Example. Consider the program in Fig. 2 (a). It reads information of an employee, computes his/her raise and sends it to a remote site. If the employee is a regular staff, function `SRaise()` is called to compute the raise (line 4). If he/she is a manager, function `MRaise()` is called (line 6). Moreover, the program reads the department information to compute the bonus for the manager. Finally (lines 10 and 11), the name and the raise are reported to a remote site. `SRaise()` opens and reads a contract file that describes the rate of raise. `MRaise()` calls `SRaise()` to compute the basic raise, using a different contract file. Furthermore, it saves all the junior managers with a salary higher than `C1` to a local file.

The control flow graphs (CFGs) and their instrumentation for counter computation (i.e. code along CFG edges) are shown in Fig. 2 (b). The number beside a node denotes the counter value at the node, computed by the instrumentation starting from the function entry. It can be intuitively considered as the maximum number of syscalls encountered along a path from the entry to the node. In `SRaise()`, the counter is incremented twice along edges $Entry \rightarrow 12$ and $12 \rightarrow 13$ before the two syscalls. The total increment is hence 2, as shown beside the exit node. In `MRaise()`, the counter value of line 15 is 2, although the edge is not instrumented. This is because of the increments inside `SRaise()`. The true branch of line 16 has an increment of 1 due to the `write` syscall. To ensure identical counter values at the join point, the false branch (i.e., edge $16 \rightarrow 18$) is compensated with $+1$. As a result, the total increment of `MRaise()` is 3 along any path. Similarly in `main()`, the path $3 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 10$ has an increment of 4, due to the three syscalls inside `MRaise()` and the syscall at line 7. As such, we compensate the edges $4 \rightarrow 10$ and $9 \rightarrow 10$ by $+2$ and $+4$, respectively.

Assume `title=STAFF` is the secret. In the slave, it is mutated to `MANAGER`. Also assume `age=JUNIOR`. Fig. 3 shows the syscall sequences of the two executions and the corresponding counter values. The first two entries are the syscalls at lines 1 and 2 in both executions, and they align due to the same counter value. Hence, the slave *copies* the syscall results from the master. The two executions diverge at line 3 and different syscalls are encountered. In particular, the master executes two syscalls inside `SRaise()` and the slave *executes* the two syscalls inside `SRaise()` in a different context, followed by the `write` at line 17 and the `read` at 7. Since these syscalls do not align, both the master and the slave execute them separately. Assume the master finishes its (true) branch first and continues to the `send` syscall at line 11. At this time, the counter is 7 in the master and larger than the slave's. The master blocks until the slave's counter also reaches 7, at which the two syscalls (at line 11) align again. Since the syscall is a sink, LDX compares the outputs and identifies differences. It hence reports a leak. Note that even though there is no direct data flow from `title` to `raise`, the value of `raise` still leaks the secret `title` through control dependences. Many existing techniques cannot detect such causality. □

One may notice in Fig. 3 that the third and the fourth syscalls in both executions have the same counter. In fact, both are syscalls



(a) Program **(b) Control Flow Graphs and Instrumentation**

Figure 2. Illustrative Example. The code along control flow edges represents instrumentation. #cnt+=2 inside **SRaise()**; ##cnt+=3 in **MRaise()**.

in `SRaise()`. To recognize syscalls that are different but have the same counter value and the same PC, LDX compares their parameters.

Fixed versus Dynamically Computed Counter values. One may also be curious that why LDX does not assign a fixed counter value to each syscall. This is because a function may be invoked under different contexts such that the counter value computed for a syscall inside the function may vary.

Use of LDX. LDX is fully automated during production runs. It has a predefined configuration of sources (e.g., socket receives) and sinks (e.g., file writes). The user can also choose to annotate the sources and sinks in the code during instrumentation. At runtime, all the specified sources are mutated. If output differences are observed at any sink, LDX considers that there is strong causality between the sink and some source(s) and reports an exception. It does not require running multiple times for individual sources.

4. Basic Design

The basic design consists of two components. The first is for counter computation and the second is for synchronizing the executions and sharing syscall results. For now, we assume programs do not have loops, recursion, or indirect calls. They are discussed in later sections (loops/recursion in Section 5 and indirect calls in Section 6).

4.1 Counter Computation

In LDX, each execution maintains a counter to allow progress comparison across runs. The basic idea of counter computation is to ensure that the current counter value represents the maximum number of syscalls along a path from the beginning of the program to the current execution point. If the program does not have any loops, recursion, or indirect calls, such a number can be uniquely computed. Hence, our instrumentation compensates the paths other than the one that has the maximum number of syscalls, by incrementing the counter, to make sure the counter must have the same value (i.e. the maximum number of syscalls) along any path. Intuitively, when the two executions take different branches of a predicate, the counter computation ensures that they align when the branches join again, because the counter will have the same value regardless of the branch taken.

Algorithm 1 Basic Counter Instrumentation Algorithm.

Input: The CFGs of the m functions of a program P, denoted as $\langle N_1, E_1\rangle$, ..., $\langle N_m, E_m\rangle$
Output: Instrumented CFGs
1: **function** INSTRUMENTPROG
2: **for** $\langle N_i, E_i\rangle$ in reverse topological order of the call graph **do**
3: INSTRUMENTFUNC ($\langle N_i, E_i\rangle$)
4: **end function**
Input: The CFG of a function F, denoted as $\langle N, E\rangle$
Output: The instrumented CFG
5: **function** INSTRUMENTFUNC
6: **for** each node $n \in N$ **do**
7: $cnt[n] \leftarrow 0$
8: **for** node $n \in N$ in topological order **do**
9: $cnt[n] \leftarrow max_{p \to n \in E}(cnt[p])$
10: **if** n is a syscall **then**
11: $cnt[n] \leftarrow cnt[n] + 1$
12: **for** each edge $p \to n \in E$ **do**
13: **if** $cnt[p] \neq cnt[n]$ **then**
14: instrument $p \to n$ with "$cnt+=$"$\cdot cnt[n] - cnt[p]$
15: **if** n is a call to user function F_x **then**
16: $cnt[n] \leftarrow cnt[n] + FCNT[F_x]$
17: $FCNT[F] \leftarrow cnt$[exit node of F]
18: **end function**

The instrumentation procedure is presented in Algorithm 1. It consists of two functions: INSTRUMENTPROG() that instruments the program and INSTRUMENTFUNC() that instruments a function. INSTRUMENTPROG() instruments functions in the reverse topological order. As such, when a function is instrumented, all its callees must have been instrumented. In INSTRUMENTFUNC(), $cnt[n]$ contains the number of maximum syscalls along a path from the function entry to n. In lines 6-7, $cnt[]$ is initialized to 0. Then in the loop from lines 8-16, the algorithm traverses the CFG nodes in the topological order and computes $cnt[]$. In particular, $cnt[n]$ is first set to the maximum of $cnt[p]$ for all its predecessors p (line 9). It is fur-

ther incremented by one if n is a syscall (lines 10-11). Then for any incoming edge $p \rightarrow n$, the algorithm instruments it with a counter increment of $cnt[n] - cnt[p]$, ensuring the counter value must be $cnt[n]$ along all edges (lines 12-14). After that, if n denotes a function call to F_x, $cnt[n]$ is incremented by the counter of the function $FCNT[F_x]$, which denotes the maximum number of syscalls that can happen inside F_x along any path (line 15-16). Note that this increment does not cause any instrumentation on n because the increment denoted by $FCNT[F_x]$ is realized inside F_x. At the end, $FCNT[F]$ is set to the computed counter value for the exit node. It will be used in counter computation in the callers of F.

Example. In Fig. 2, the algorithm first instruments `SRaise()`. The $cnt[]$ values are showed beside the nodes. Observe that $cnt[12] = 1$ and $cnt[13] = 2$, which lead to the instrumentation on $entry \rightarrow 12$ and $12 \rightarrow 13$. $FCNT[\texttt{SRaise}] = cnt[14] = 2$. `MRaise()` is instrumented next. Due to $FCNT$ [`SRaise`], $cnt[15] = 2$. Note that node 15 is not instrumented. Node 18 has two predecessors and thus $cnt[18] = max(cnt[17], cnt[16]) = 3$, which entails the instrumentation on $16 \rightarrow 18$. At last, function `main()` is instrumented. $cnt[10] = max\ (cnt[8], cnt[4], cnt[9]) = cnt[8] = 6$, causing the instrumentation on $4 \rightarrow 10$ and $9 \rightarrow 10$. □

Algorithm 2 Syscall Wrapper for Master.

Input: Syscall id *sys_id* and parameters *args*.
Output: Syscall return value.
Definition: Q_m the syscall outcome queue maintained by the master; O_s the latest sink syscall by the slave; cnt_m and cnt_s the local counters in master and slave, respectively; $ready_m$ the counter value in master exposed to the slave; similarly, $ready_s$ the counter value in the slave exposed to the master.

```
1:  function SYSCALLWRAPPER(sys_id, args)
2:      if sys_id denotes a sink syscall then
3:          while cnt_m > ready_s do
4:              {}
5:          if cnt_m < ready_s ∨ O_s.sys_id ≠ sys_id ∨ O_s.args ≠ args then
6:              report causality
7:      r ← SYSCALL (sys_id, args)
8:      Q_m.enq(⟨cnt_m, sys_id, args, r⟩)
9:      ready_m ← cnt_m
10:     return r
11: end function
```

4.2 Dual Execution Facilitated by Counter Numbers

To support dual execution, LDX intercepts syscalls to perform synchronization and syscall outcome sharing. In the master, when a syscall is encountered, if it is not a sink, LDX executes the syscall and saves the outcome for potential reuse by the slave. Otherwise, it waits for the slave to reach the same sink so that their parameters can be compared. In the slave, upon a syscall, it first checks whether it is ahead of the master. If so, it waits until the master finishes the corresponding syscall so that it can copy the master's result. If the corresponding syscall does not appear in the master (due to path differences), which can be detected by the counter scheme, the slave executes the syscall.

Execution Control in the Master. Algorithm 2 shows the controller of the master. It is implemented as a syscall wrapper. Each syscall in the master must go through the controller. Inside the controller, cnt_m and cnt_s denote the current counter values in the master and the slave, respectively. They are local to their execution and invisible to the other execution. It also uses two shared variables $ready_m$ and $ready_s$ to facilitate synchronization. They are assigned the values of cnt_m and cnt_s when the master and the slave are ready to disclose the effects of the current syscall to the other party.

Lines 2-6 handle a sink syscall. At line 3, the master spins until the slave catches up. Note that the value of $ready_s$ is the same as cnt_s when the state of the slave's syscall denoted by cnt_s becomes visible. There are four possible cases after the master gets out of the spin loop.
(1) $cnt_m < ready_s$. This happens when there is not a syscall denoted by the value of cnt_m in the slave. For example in Fig. 2, assume the master takes the false branch at line 3 and is now at line 7 with $cnt_m = 6$ while the slave takes the true branch and now it just returns from the call to `SRaise()` at line 4 with $cnt_s = ready_s = 4$. Assume we make line 7 a sink. Then the master will wait at line 7. However, the next time $ready_s$ is updated (in the slave) is at line 11, at which $ready_s = 7$, larger than $cnt_m = 6$.
(2) $cnt_m \equiv ready_s$ but the syscall in the slave represented by $ready_s$ is different from the sink syscall in the master. This is due to path differences.
(3) $cnt_m \equiv ready_s$ and both the master and the slave align at the same sink syscall. However, their arguments are different.
(4) The counters, syscalls, and arguments are all identical.

The first three cases denote causality between the source and the sink, suggesting leak or exploit. The last case is benign. In the first two cases, there is causality because the sink (in the master) disappears in the slave with the input perturbation. The three comparisons at line 5 correspond to the first three cases, respectively.

If the current syscall is not a sink, lines 7-8 in the algorithm perform the real syscall and enqueue the syscall and its outcome, which may be reused by the slave. At last (line 9), $ready_m$ is set up-to-date, indicating the syscall outcome for cnt_m is ready (for the slave).

Execution control in the slave is similar. Details can be found in our technical report (tr).

Syscall Handling. LDX's policy of handling syscalls is similar to that in *dual execution* (DualEx) (Kim et al. 2015). For most input/output syscalls, the slave simply reuses the master's syscall outcome if their alignments in the master can be found. Otherwise, it *executes* the syscall. To avoid undesirable interference, the slave may need to construct its own copy of the system state before executing the syscall. For example, before the slave executes a file read, the file needs to be cloned, opened, and then seeked to the right position. Some special syscalls are always executed independently such as process creation. Since the policy is not our contribution, we refer the interested reader to (Kim et al. 2015).

Dual Execution Model Comparison between LDX and DualEx (Kim et al. 2015). Similar to LDX, DualEx also has the master and the slave. However, its synchronization and alignment control is through a third process called the *monitor*. Both the master and the slave simply send their executed instructions to the monitor, which builds a tree-like execution structure representation called *index* and aligns the executions based on their indices. The monitor also determines if a process needs to be blocked, achieving lockstep synchronization. As such, its overhead is very high (i.e., 3 orders of magnitude). In contrast, LDX is much more lightweight. It is based on counter values and uses spinning to achieve synchronization.

5. Handling Loops

The basic design assumes programs without loops. Handling loops is challenging because the number of iterations for a loop is unknown at compile time. The master and the slave may iterate different numbers of times due to the perturbation at sources, leading to different increments to the counters and hence difficulty in alignment. Our solution is to synchronize two corresponding loops at the iteration level. In particular, it aligns the ith iteration of the master with the ith iteration of the slave by synchronizing at the backedges, i.e. the edge from the end of the loop body back to the loop head. It is analogous to having a barrier at the end of each iteration. Along the backedge, LDX also resets the counter to the value before it en-

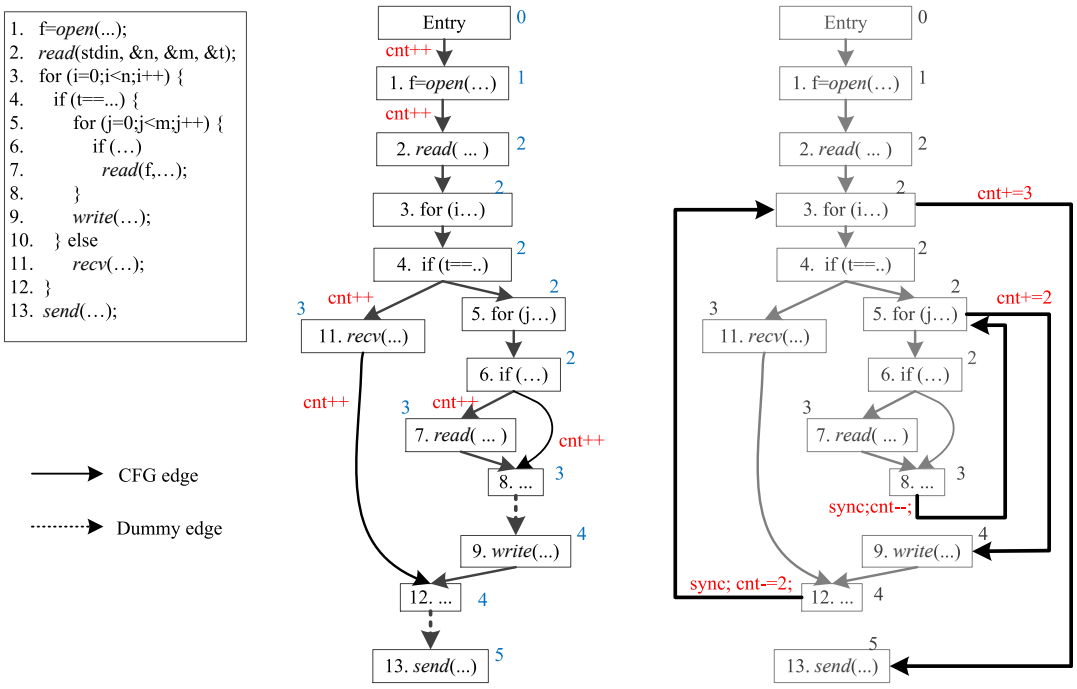


Figure 4. Loop Example.

tered the loop. Doing so, the value of the counter is bounded and does not grow with the number of iterations. If an execution gets out of the loop, its counter is incremented by the maximum number of syscalls along any path inside the loop. As such, a counter value beyond the loop is larger than any counter values within the loop, correctly indicating that the execution beyond the loop is ahead of the one in the loop.

Algorithm 3 Counter Instrumentation with Loops.

Input: The CFG of a function F, denoted as $\langle N, E\rangle$
Output: The instrumented CFG

```
1:  function InstrumentFuncWithLoop
2:      for each back edge t → h ∈ E do
3:          Let h → n be the exit edge of the loop
4:          E ← E − {t → h, h → n}      ▷ Remove loop exit and back edges
5:          E ← E ∪ {t → n}             ▷ Add dummy edge
6:      InstrumentFunc(⟨N, E⟩)
7:      remove all dummy edges and their instrumentation
8:      restore all the removed edges in the original CFG
9:      for each original back edge e : t → h do
10:         instrument e with "sync(); cnt- ="·cnt[t] − cnt[h]
11:     for each original loop exit edge e : h → n do
12:         instrument e with "cnt+ ="·cnt[n] − cnt[h]
13: end function
```

Algorithm 3 presents the instrumentation algorithm for a function with loops. It transforms the CFG to an acyclic graph by removing loop edges. As such, the $cnt[]$ values in the acyclic graph are statically computable. The computed $cnt[]$ values are then leveraged to construct the instrumentation, including that for the original loop edges. Particularly, the algorithm first removes all the backedges and the loop exit edges (line 2-5). A loop exit edge is from the loop head h to the next statement n beyond the loop. A *dummy edge* is inserted from the end of the loop body t to the next statement n beyond the loop. Our discussion focuses on `for` and `while` loops, `do-while` loops can be similarly handled.

At line 6, the acyclic graph is instrumented through InstrumentFunc(). After that, the dummy edges and their instrumentation are removed as they do not denote real control flow (line 7). The backedges and loop exit edges are then restored. Lines 9-10 instrument the backedges. For a backedge $t \to h$, the instrumentation first calls a barrier function `sync()`, which is similar to lines 3-4 in Algorithm 2, to synchronize with the backedge of the same iteration in the other execution. It then resets the counter to the value at h such that the counter increment of the next iteration has a fresh start. Lines 11-12 instrument the loop exit edges. For a loop exit $h \to n$, the instrumentation increments the counter by the difference between $cnt[n]$ and $cnt[h]$. Intuitively, it raises the counter to the value of $cnt[n]$.

Example. Fig. 4 (a) shows a loop example. There are two loops: the i loop and the j loop. Their iteration numbers are determined by the inputs from line 2. Figure (b) shows the transformed CFG and part of the instrumentation generated by InstrumentFunc() in the basic design. Observe that the backedges $8 \to 5$ and $12 \to 3$, the loop exit edges $3 \to 13$ and $5 \to 9$ are removed. Dummy edges $8 \to 9$ and $12 \to 13$ are added. They do not represent real control flow, but allow $cnt[9]$ to be computed as $cnt[8]+1$ and $cnt[13] = cnt[12]+1$. Figure (c) shows the instrumentation for backedges and loop exit edges. Note that the CFG in (c) is the original CFG. The backedge $8 \to 5$ is instrumented with the call to the barrier function and the decrement of the counter by $cnt[8] - cnt[5] = 1$. The loop exit edge $5 \to 9$ is instrumented with the counter increment of $cnt[9] - cnt[5] = 2$, which makes the counter value of node 9 always larger than those within loop j. The instrumentation for loop i is similar.

Fig. 5 shows the dual execution when the loop bounds n and m are the sources. Assume the master executes with $n = 1$ and $m = 2$ and the slave executes with $n = 2$ and $m = 1$. Along the syscall sequences, we also show the loop iterations to facilitate understanding. The first three syscalls (up to inside the first iteration of j) align in the two executions. At Ⓐ, the two executions are

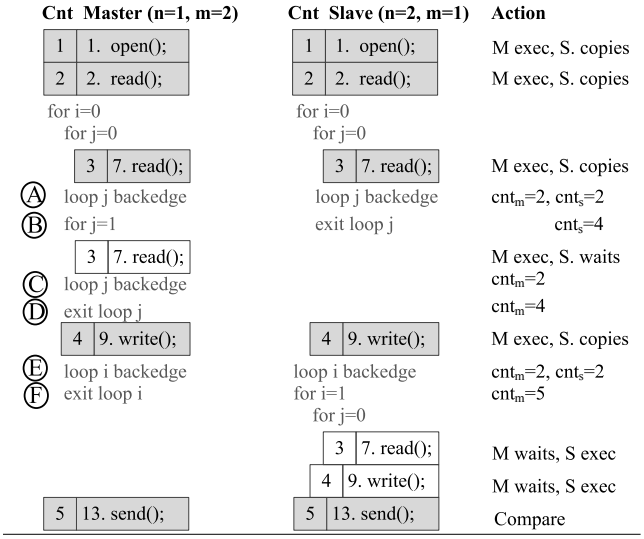


Figure 5. Syscalls and the sequence of synchronizations by LDX for the example in Fig. 4 with `n` and `m` the sources. The shaded entries are aligned. The indentation shows the loop nesting.

synchronized and counters are reset to 2. However at Ⓑ, the slave exits loop *j* while the master continues to the second iteration of *j*. As such, the slave's counter becomes 4, which blocks its execution. At Ⓒ, the master finishes the second iteration of *j* and its counter is reset to 2. At Ⓓ, the master also exits loop *j* and its counter is incremented to 4, which aligns the two syscalls at line 9. At Ⓔ, the two runs are synchronized at the backedge of loop *i* and their counters are reset to 2 due to the instrumentation on $12 \rightarrow 3$. At Ⓕ, the master exits the *i* loop; its counter becomes 5 due to the instrumentation on $3 \rightarrow 13$, which blocks its execution as the master needs the parameters of the `send()` from the slave to infer causality. In contrast, the slave executes the remaining *i* iteration before it reaches the aligned sink (line 13). □

Recursive functions are handled similarly. Also note that we only need to instrument loops that include syscalls. Hot loops are usually computation intensive and should not have syscalls. Therefore, they are unlikely to be instrumented.

6. Handling Indirect Function Calls

The challenge for handling indirect calls is that the call targets are usually unknown at compile time. As a result, we cannot use the counter values in the callee(s) to compute those in the caller. To handle indirect calls, LDX saves a copy of the current counter to the stack when an indirect call is encountered, and resets the counter to 0 such that the two executions start a fresh alignment from the indirect call site. When the executions return from the indirect call, the counter value is restored. As such, we do not need to know the precise counter increment inside the indirect call to support alignment in the caller. LDX supports components that cannot be instrumented such as third party libraries and dynamic loaded libraries by synchronizing at their interface. `Longjmp` and `setjmp` are ignored during the CFG analysis. They are supported at runtime by saving a copy of the counter stack at the `setjmp` which will be restored upon the `longjmp`. Moreover, an artificial sink is inserted before the `longjmp` so that if one process `longjmp`s but the other does not, LDX reports exception. More details can be found in (tr).

7. Handling Concurrency and Library Calls

LDX supports real concurrency, which is completely different from DualExec (Kim et al. 2015). Threads have their own counters. Threads in the master and the slave are paired up. LDX treats pthread library calls as syscalls. The two executions hence synchronize on those calls and share the outcomes of lock acquisitions and releases. Note that sharing synchronization outcomes induces very similar thread schedules in the two executions. However, path differences may lead to synchronization differences which may in turn lead to deadlocks in LDX if not handled properly. We taint locks that have encountered differences and avoid sharing synchronization outcomes for those locks. Moreover, low-level data races that are not protected by any locks may induce non-deterministic state differences, leading to false positives in strong CC inference. In Section 8, our experiment shows that false positives rarely happen (for the programs we consider). Intuitively, non-determinism during computation may not lead to non-determinism at the sinks.

Light-weight Resource Tainting. In our current implementation, a file/directory is considered a resource. Taint metadata is associated with each resource. When an operation for a resource is misaligned, the resource is tainted to indicate state differences so that any future syscalls on the resource cannot be coupled. When a tainted resource is accessed by the other execution, LDX will create a copy of the related resource(s) so that the master and the slave operate on their own copies, without causing interference. For example, if the master creates a directory while the slave does not, the directory is tainted. When the slave tries to access the directory later, it gets into the de-coupled mode. The slave's syscall will be performed on a clone of the parent directory without the created directory. Similarly, if a file is renamed or removed from a directory in one execution but not the other, the file is tainted. Any following acceses to the file lead to de-coupled execution.

Handling Library Calls. Regarding local file outputs, the slave does not perform any outputs to the disk if they are aligned. Instead, it skips the calls or buffer the output values for causality inference if local file outputs are considered sinks. The slave ignores its own signals and receives its signals from the master. Upon a signal, LDX allows the slave to execute the signal handler. Handler invocations are handled similar to indirect calls. Note that the slave may invoke system calls to cause different signals or events such as creating threads or processes different from the master. LDX buffers such different system calls and all the system calls caused by such signals and events for causality inference. The threads and processes unique to either execution run in the de-coupled mode.

Handling UI Library Calls. LDX is intended to be transparent to the user. Hence, it is undesirable to have two (almost identical) user interfaces. Therefore, LDX allows the master to handle all the UI library calls as usual. The slave does not have its own interface. It tries to reuse the UI library call outcomes from the master as much as possible. Misaligned UI library calls, if they are input related, return random values to the slave. Misaligned output UI calls are ignored, or buffered for causality inference if the outputs are considered sinks.

8. Evaluation

LDX is implemented in LLVM 3.4. We evaluate its runtime performance, the capability of handling misaligned syscalls, and the effectiveness of causality inference with two applications: information leak detection and attack detection. Experiments are on a machine with Intel i7-4770 3.4GHz CPU (4 cores), 8GB RAM, and 32-bit LinuxMint 17.

Benchmark Programs. We used 28 programs as shown in Table 1. They include four different subsets: SPECINT2006 (the first 12); the network and system related set for information leak detection (the next 5), the vulnerable program set for attack detection (the next 6), and the concurrency set (the last 5) for evaluation of concurrency control. The detailed introduction of these programs can be found in (tr).

Table 1. Benchmarks and Instrumentation.

Program	LOC	Instrumented instances				Syscalls		Max Cnt.	Dyn. Cnt.		Mutated inputs
		Inst.	Loop	Recur.	FPTR	Sinks	Total		Value	Stack*	
400.perlbench	128K	5540 (1.56%)	10233	634	852	4	62	72K	3392	2.91/7	Input perl source file
401.bzip2	5739	43 (0.24%)	360	0	57	4	10	7	4.5	0/1	Input data file
403.gcc	385K	791 (0.07%)	45702	2928	463	3	31	424	96.1	0.11/5	Input C source file
429.mcf	1579	27 (1.32%)	44	1	0	3	11	8	4.3	0/0	Input data file
445.gobmk	157K	235 (0.22%)	7910	74	47	3	15	37	1.7	1.68/4	Input data file
456.hmmer	20K	1762 (3.59%)	1611	11	13	4	25	281	83.2	0/1	Input file and arguments
458.sjeng	10K	26 (0.13%)	978	10	1	4	12	6	2.7	0.07/1	Input data file
462.libquantum	2611	52 (1.08%)	153	11	0	3	17	8	1	0/0	Input arguments
464.h264ref	36K	102 (0.09%)	1994	38	362	4	20	101	26.4	0.26/2	Configuration file
471.omnetpp	26K	121 (0.09%)	6102	46	838	2	22	20	4.5	2.3/6	Configuration file
473.astar	4285	56 (0.47%)	224	0	1	1	18	51	32.8	0.12/1	Configuration file
483.xalancbmk	266K	116 (0.01%)	28381	312	10265	5	25	5	1.5	1.34/9	Input XML file
Firefox	14M	83 (0.01%)	21	0	9	3	26	71	41.2	0.09/1	nsIURI object accesses
lynx	204K	13157 (6.92%)	6799	109	1179	6	132	15M	578K	0.3/6	Cookie, network packet
nginx	287K	4672 (4.27%)	1541	21	850	6	110	518	17.9	3.8/7	Configuration file
tnftp	152K	2452 (6.31%)	1093	17	210	8	125	5878	2623	0.01/1	Input arguments
sysstat	29K	811 (6.94%)	271	0	1	3	47	365	70.7	0.01/1	Returns of lib. calls
gif2png	16K	246 (7.76%)	62	0	0	7	36	76	18.2	0/0	Input image file
mp3info	9252	205 (8.34%)	91	0	0	3	31	88	6.4	0/0	Input mp3 file
prozilla	13K	1116 (8.19%)	285	0	14	5	67	5680	713	0/0	Network packet
yopsweb	1961	282 (5.93%)	97	0	1	4	44	24	3.7	0/1	Network packet
ngircd	66K	1052 (6.70%)	417	24	1031	4	62	2863	1524	0/1	Network packet
gocr	54K	2801 (5.48%)	2581	4	2	3	24	23K	2182	0/1	Input image file
Apache	208K	640 (0.61%)	2700	23	183	6	126	89	43.7	1.56/4	Input HTML file
pbzip2	4527	735 (6.74%)	226	0	3	4	49	1997	578.83	0/0	Input data file
pigz	5766	996 (5.85%)	434	2	15	6	54	9288	432.82	0.99/1	Input data file
axel	2583	342 (8.24%)	162	1	3	6	35	271	73.66	0/0	Network packet
x264	98K	2071 (1.30%)	2218	1	2295	8	49	881	76.58	15K/18K	Input video file

* It shows avg/max

Instrumentation Details. Table 1 shows the instrumentation details. Columns 3-6 describe the numbers of instrumented instructions (and their percentage), instrumented loops, instrumented recursive functions, and instrumented indirect calls. The next two columns show the number of sinks and syscalls instrumented. For programs that have network connections, we use the outgoing networking syscalls as sinks. For other programs, we treat the local file outputs as sinks. The "max cnt." column shows the maximum counter value in a program. It denotes the largest number of syscalls along some static program path. For `firefox`, we were not able to instrument the whole program as LLVM failed to generate the whole program bitcode (supposedly larger than 600MB). We identified the source files for event processing and the JS engine and only instrumented those. The resulted object files are then linked with the rest.

We have a few observations. (1) We have some large and complex programs such as `lynx`, `403.gcc`, and `apache`. (2) The percentage of instrumented instructions is low (3.44% on average). (3) Some programs (e.g., `403.gcc` and `400.perlbench`) have a large number of recursive functions and indirect calls. LDX handles all of them.

The last column of Table 1 shows the source mutations. For the SPEC and network/system programs, we mutate the data files and the configuration files. For the vulnerable program set, we mutate the inputs from untrusted sources and detect whether differences are observed at function return addresses (for buffer overflow attacks) and at parameters of memory management functions (for integer overflow attacks). We perform off-by-one mutations. In order to avoid invalid mutations, we only mutate data fields, not magic values or structure related values.

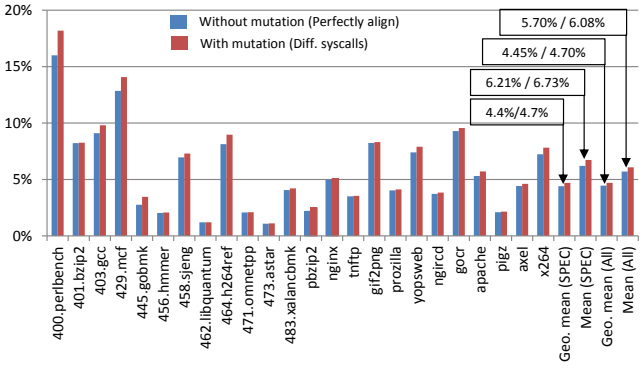


Figure 6. Normalized overhead of LDX.

8.1 Performance

We study the performance of LDX using SPECINT2006 and programs that are not interactive and have non-trivial execution time. For server programs such as `nginx` and `apache`, we run the server and send 10,000 requests, and then measure the throughput. For web servers such as `apache`, we use `ApacheBench` to provide the requests. `Firefox` and `lynx` are omitted because they are interactive. `Sysstat` and `mp3info` are also excluded as their running time is trivial (¡0.01sec). We use the reference inputs for SPEC. We run each program twice. In the first run, we do not mutate the input so that the master and the slave perfectly align. The overhead is thus for counter maintenance and syscall outcome sharing. In the second run, the master and the slave execute with different inputs. Since they can take different paths and have different syscalls, the overhead includes that for synchronization and realignment. The results are shown in Fig. 6. The baseline is the

native execution time for the uninstrumented programs with the original inputs. The geometric means of the overhead are 4.45% and 4.7%, while the arithmetic means are 5.7% and 6.08%. Observe that the overhead of LDX is very low. We have also measured the overhead of LIBDFT (Kemerlis et al. 2012), one of the state-of-the-art dynamic tainting implementations that works by instruction level monitoring. Its slow-down over native executions is roughly 6X on average. LDX is also three orders of magnitude faster than *dual execution* (Kim et al. 2015).

Another observation is that the input differences and hence the syscall differences do not cause much additional overhead. As we will show later, the syscall differences are not trivial. This is because our alignment scheme allows the misaligned syscalls to execute separately and concurrently. The "dyn. cnt." columns in Table 1 show the runtime characteristics of the counter values. Observe that the average counter values are much smaller than the maximum values (column 9). The maximum depth of the stack is also small, meaning that we rarely encounter nesting indirect calls.

Table 2. Dual Execution Effectiveness.

Program	Input 1 / Input 2		# of syscall diffs	
	LDX	TightLip	Input 1	Input 2
lynx	O / X	O / O	1801 (4.13%)	1272 (3.0%)
nginx	O / X	O / O	202 (13.92%)	181 (13.02%)
tnftp	O / X	O / O	2443 (19.19%)	381 (15.74%)
sysstat	O / X	O / O	53 (7.42%)	58 (19.21%)
gcc	O / X	O / O	38161 (24.99%)	3590 (3.11%)
xalancbmk	O / X	O / O	102 (2.60%)	91 (2.32%)
gobmk	O / X	O / O	345 (1.68%)	114 (0.55%)
perlbench	O / X	O / O	17 (7.08%)	11 (4.58%)
bzip2	O / X	O / O	53 (54.63%)	49 (50.51%)
mcf	O / X	O / O	20 (0.01%)	17 (0.01%)
sjeng	O / X	O / O	729 (45.45%)	132 (8.22%)
h264ref	O / X	O / O	141 (31.68%)	12 (2.69%)
hmmer	O / -	O / -	2 (0.03%)	-
libquantum	O / -	O / -	1 (12.5%)	-
omnetpp	O / -	O / -	0	-
astar	O / -	O / -	11 (73.33%)	-

8.2 Effectiveness of Dual Execution

In this experiment, we answer the question why we need to align the master and the slave. The experiment is in the context of detecting information leak. For each program, we construct two input mutations with the following goal: one input mutation leads to sink differences (and hence leakage) and the other does not. Both mutations may trigger syscall differences. We also compare LDX with TIGHTLIP, which does not align executions and often has to terminate at syscall differences, reporting leakage. Table 2 presents the results. Symbol 'O' denotes that leakage is reported and 'X' denotes normal termination without any warning. The last two columns show the syscall differences before the sink difference and their percentage over the total number of dynamic syscalls. We have the following observations. (1) LDX correctly identifies that one input mutation causes leakage while the other one does not (except for the last four cases), whereas TIGHTLIP reports leakage for both input mutations. Note that a lot of syscall differences are not output related. (2) The syscall differences caused by input mutations are not trivial and are sometimes substantial. LDX can properly handle all such differences. (3) For numerical computation oriented programs (i.e., the last four in the table), we were not able to construct the input mutation that does not cause leakage as any input mutation always leads to sink differences.

8.3 Effectiveness of Causality Inference

Comparison with Dynamic Tainting. We first compare LDX with TAINTGRIND (Khoo 2013) and LIBDFT (Kemerlis et al. 2012) [1].

[1] We have tried DECAF (formerly TEMU), but encountered build problems.

Table 3. Comparison with Dynamic Tainting

Program	# of tainted sinks			Total # of sinks
	LDX	TAINTGRIND	LIBDFT	
gcc	3	0	0	146
perlbench	1	0	0	5
bzip2	7	0	0	20
mcf	12	4	3	36
gobmk	68	39	39	84
hmmer	17	4	4	29
sjeng	83	8	6	112
libquantum	4	2	2	7
h264ref	28	3	3	37
omnetpp	24	4	2	52
astar	16	3	3	53
xalancbmk	45	21	0	419
lynx	5	3	1	8
nginx	10	5	0	22
tnftp	5	2	0	32
sysstat	6	3	0	12
gif2png	1	1	1	7
mp3info	1	1	1	8
prozilla	1	1	1	100799
yopsweb	1	1	0	41
ngircd	1	1	1	597
gocr	1	1	1	5
total	340	107	68	-

We compare the number of tainted sinks for all the benchmarks. For the set of programs with vulnerabilities, their sinks include function returns and memory management library calls. The results are shown in Table 3. The three columns in the middle report the number of tainted sinks. The last column shows the total number of sinks encountered during execution.

We have the following observations. (1) The tainted sinks reported by TAINTGRIND and LIBDFT and are only 31.47% and 20% of those reported by LDX. This is because the other two are based on tracking data dependences. As we discussed in Section 2, data dependences are essentially strong causalities. Hence, LDX can detect what the other two detect. In addition, LDX can detect strong causalities induced by control dependences. We have validated that all the sinks reported by LDX have one-to-one mappings with the tainted inputs (i.e., no false positives). (2) The tainted sinks reported by TAINTGRIND are a superset of those reported by LIBDFT. Further inspection shows that LIBDFT does not correctly model taint propagation for some library calls. This indeed illustrates a practical challenge for instruction tracking based causality inference, which is to correctly model taint behavior for the large number of instructions and libraries. The last six rows show the results for the vulnerable program set. Observe that LDX can detect the attacks by correctly inferring the causality between the untrusted inputs and the critical execution points.

Effectiveness for Concurrent Programs. LDX supports real concurrency by sharing the thread schedule as much as possible between the two executions (Section 7). However, low level races may introduce non-deterministic state differences, leading to false positives in causality inference. In this experiment, we collect 5 concurrent programs. For each program, we mutate the input and dual execute it 100 times. We used the standard inputs provided with the programs. As shown in column 3 of Table 4, the number of tainted sinks rarely changes, whereas syscall differences do change (column 2) due to low level races. However, the syscall difference changes are not substantial because LDX was able to enforce the same schedule for most cases. This supports the effectiveness of the concurrency control of LDX (for the programs we consider). The tainted sink changes for `x264` are caused by the ex-

ecution statistics report (e.g., the bits processed per sec.). Although LDX forces the master and the slave to share the same schedule and the same timestamps, the number of bits processed per unit time is non-deterministic *across tests* and beyond control. The tainted sink changes for `axel` are because the program makes Internet connections in each run, which are non-deterministic.

Table 4. Effectiveness of concurrent programs.

Program	# of syscall diffs (Min/Max/Std. Dev.)	# of tainted sinks (Min/Max/Std. Dev.)
Apache	114 / 123 / 1.66	39 / 39 / 0
pbzip2	288 / 332 / 11.59	8 / 8 / 0
pigz	490 / 546 / 18.50	14/ 14 / 0
axel	1173 / 1252 / 25.39	813 / 834 / 6.5
x264	854 / 1211 / 89.38	350 / 353 / 0.3

Input Mutation. LDX performs off-by-one mutation on sources, which must detect any strong CCs as proved in (tr). However in some rare cases it may also detect weak causalities. We conduct an experiment to study different mutation strategies. We observe that other strategies do not supercede off-by-one. Details can be found in (tr).

8.4 Case Studies

403.gcc. In this study, we use the source code of `nginx` as input. Fig. 7 shows part of input code on the left. We specify the configuration `NGX_HAVE_POLL` as the source. The master has `NGX_HAVE_POLL` defined but the slave does not. As such, the master includes `poll.h` while the slave does not. This corresponds to 7_2, 8_2, and 10_2 (Fig. 7) occurring in the master but not in the slave. Later on, both executions re-align at 216_1 and run in the coupled mode. In fact, 216 and 217 are in an output loop that emits the preprocessed code. Due to the earlier differences, the preprocessed code is different. The differences manifest as parameter differences during executions of 216_i, 217_i in the master and 216_j, 217_j in the slave. The leak is reported. Note that the causality is strong as one can infer from the preprocessed code the value of `NGX_HAVE_POLL`.

Other tools such as LIBDFT and TAINTGRIND are not able to detect the causality as it is induced by control dependences, Fig. 7 shows the relevant `gcc` code on the right. At line 472, `gcc` reads the value of `NGX_HAVE_POLL` and stores it. Later, when the preprocessor reaches the "`#if NGX_HAVE_POLL`" statement inside `do_if()`, it reads the stored value and compares it with 0. The outcome is stored to `skip` at line 1329. Then, the variable is copied to `pfile->state.skipping` (line 1331), which later determines if the code block guarded by the if statement should be skipped or not. Note that although there are data dependences $472 \rightarrow 1329$ and $1329 \rightarrow 1331$, the connection between `pfile->node->value` and `skip` at line 1329 is control dependence, which breaks the taint propagation in LIBDFT and TAINTGRIND.

Firefox. In this case, we detect information leak in a `firefox` extension `ShowIP 1.2rc5` that displays the IP of current page. It sends the current url to a remote server. LDX instruments the event handling component and part of the JS engine in `firefox` to align JS code block executions that correspond to page loading and user event handling. It successfully detects the leak whereas TAINTGRIND and LIBDFT fail because the leak goes through control dependences. Details can be found in (tr).

9. Related Work

Dual Execution. LDX is closely related to dual execution (Kim et al. 2015). The main differences are the following. (1) LDX is very lightweight (6.08% overhead) whereas (Kim et al. 2015) relies on the expensive execution indexing (Xin et al. 2008), causing 3 orders of magnitude slowdown. (2) LDX allows threads to execute concurrently whereas (Kim et al. 2015) does not. (3) The applications are different. The low overhead of LDX makes it a plausible causality inference engine in practice. (4) Their dual execution models are different as explained in Section 4.2. TIGHTLIP (Yumerefendi et al. 2007) also uses the master-and-slave execution model to detect inforamtion leak. It uses a window to tolerate syscall differences. The simple approach can hardly handle nontrivial differences.

Execution Replication and Replay. Execution replication has been widely studied (Birman 1985; Chereque et al. 1992; Tulley and Shrivastava 1990; Black et al. 1998; Castro et al. 2003; Berger and Zorn 2006; Vandiver et al. 2007; Chun et al. 2008; Hosek and Cadar 2015). The premise is similar to n-version programming (Chen and Avizienis 1995), which runs different implementations of the same service specification in parallel. Then, voting is used to produce a common result tolerating occasional faults. There are many security applications (Cox et al. 2006; Bruschi et al. 2007; Lvin et al. 2008; Salamat 2009; McDermott et al. 1997; Yumerefendi et al. 2007) of execution replication by detecting differences among replicas. There are also works in execution replay (Hower and Hill 2008; Montesinos et al. 2009; Narayanasamy et al. 2006; Sorrentino et al. 2010; Park et al. 2009; Veeraraghavan et al. 2012; Chandra et al. 2011; Goel et al. 2005; Kim et al. 2010). In contrast, LDX align different paths during execution. RAIL (Chen et al. 2014) re-runs applications with previous inputs to identify information disclosure after a vulnerability is fixed. To handle state divergence between the original and replay executions, it requires developers to annotate the program. DORA (Viennot et al. 2013) is a replay system that records execution beforehand to replay with a modified version of the application. Instead, LDX runs two executions of an application with input perturbation to infer causality at real-time. LDX focuses on aligning two executions accurately using a counter algorithm, while (Viennot et al. 2013) relies on heuristics to tolerate non-determinism.

Dynamic Taint Tracking. Most dynamic tainting techniques (Song et al. 2008; Kemerlis et al. 2012; Clause et al. 2007; Bosman et al. 2011; Qin et al. 2006; Attariyan and Flinn 2010) work by tracking instruction execution and hence are expensive. They have difficulty handling control dependences (McCamant and Ernst 2008). Some have limited support by detecting patterns (Kang et al. 2011) or handling special dependences (Bao et al. 2010). In particular, (Attariyan and Flinn 2010) identifies and handles a subset of important control dependencies using several heuristics. LDX provides a solution to such problems by detecting strong CC based on the definition of causality instead of program dependencies. Approaches for quantifying information flow (McCamant and Ernst 2008; Heusser and Malacaria 2010; Backes et al. 2009; Mardziel et al. 2014) aim to precisely ascertain figures like the number of sensitive bits of information that an attacker may infer, the number of attack attempts required, or strategies for identifying secrets. Hardware based solutions (Tiwari et al. 2009b,a; Li et al. 2011; Tiwari et al. 2011) have been proposed to speed up or improve accuracy of taint analysis.

Secure Multiple Execution (SME). SME (Devriese and Piessens 2010; Austin and Flanagan 2012; Capizzi et al. 2008) splits an execution into multiple ones for different security levels: the low execution does the public outputs and the high execution does the confidential outputs. SME can enforce the non-interference policy. It blocks or terminates when the two executions diverge, which is intended for non-interference. In comparison, LDX focuses on causality inference and tolerates execution divergence.

Statistical Fault Localization (SFL). Recent approaches in SFL (Bai et al. 2015; Baah et al. 2010; Shu et al. 2013) use causal inference methodology in order to mitigate biases such as confoundings. In particular, suspiciousness scores that guide to locate faults can be distorted by such biases, producing inaccurate results. They run a

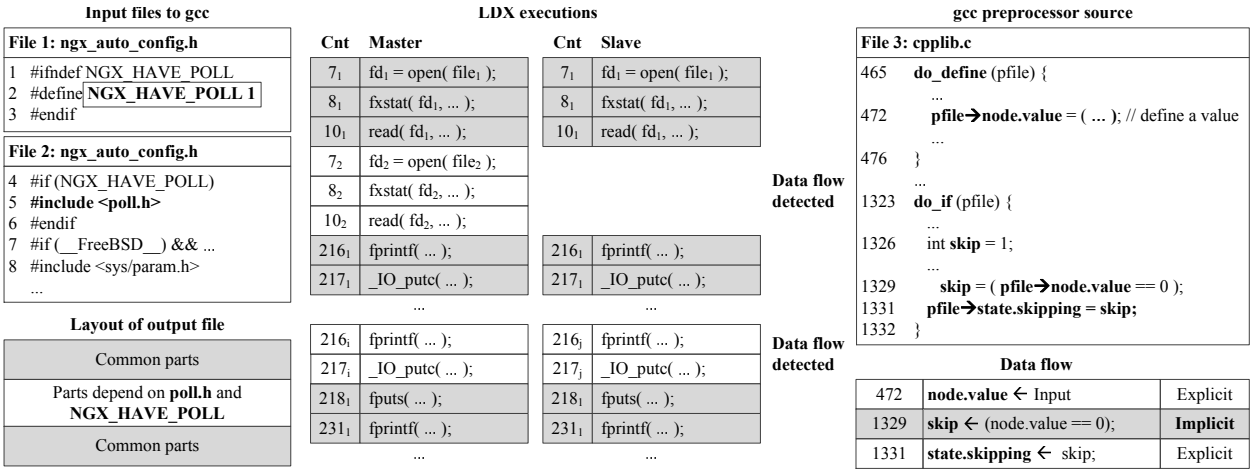


Figure 7. Case study on `403.gcc`. Input files on the left; relevant gcc code on the right; dual execution in the middle.

program over a set of inputs repeatedly to identify the causal effect of a statement on program failures. Such causal effect is then used to improve the performance and accuracy of SFL by reducing confounding bias. Instead, LDX infers causality by running multiple executions concurrently while tolerating execution divergence caused by the input purturbation.

10. Conclusion

We present LDX, a causality inference engine by lightweight dual execution. It features a novel numbering scheme that allows LDX to align executions. LDX can effectively detect information leak and security attacks. It has much better accuracy than existing systems. Its overhead is only 6.08% when executing both the master and the slave concurrently on separate CPUs. This is much lower than systems that work by instruction level tracing although they do not require the additional CPU and memory.

11. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. This research was supported, in part, by DARPA under contract FA8650-15-C-7562, NSF under awards 1409668 and 0845870, ONR under contract N000141410468, and Cisco Systems under an unrestricted gift. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

Lightweight dual-execution engine project website. https://sites.google.com/site/ldxprj.

M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–11, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1924943.1924960.

T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *POPL*, 2012.

G. K. Baah, A. Podgurski, and M. J. Harrold. Causal inference for statistical fault localization. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 73–84, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-823-0. doi: 10.1145/1831708.1831717. URL http://doi.acm.org/10.1145/1831708.1831717.

M. Backes, B. Kopf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 141–153, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3633-0. doi: 10.1109/SP.2009.18. URL http://dx.doi.org/10.1109/SP.2009.18.

Z. Bai, G. Shu, and A. Podgurski. Numfl: Localizing faults in numerical software using a value-based causal model. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–10, April 2015. doi: 10.1109/ICST.2015.7102597.

T. Bao, Y. Zheng, Z. Lin, X. Zhang, and D. Xu. Strict control dependence and its effect on dynamic information flow analyses. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 13–24, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-823-0. doi: 10.1145/1831708.1831711. URL http://doi.acm.org/10.1145/1831708.1831711.

E. D. Berger and B. G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 158–168, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: 10.1145/1133981.1134000. URL http://doi.acm.org/10.1145/1133981.1134000.

K. P. Birman. Replication and fault-tolerance in the isis system. *SIGOPS Oper. Syst. Rev.*, 19(5):79–86, Dec. 1985. ISSN 0163-5980. doi: 10.1145/323627.323636. URL http://doi.acm.org/10.1145/323627.323636.

D. Black, C. Low, and S. K. Shrivastava. The voltan application programming environment for fail-silent processes. *Distributed Systems Engineering*, 5(2):66–77, 1998.

E. Bosman, A. Slowinska, and H. Bos. Minemu: The world's fastest taint tracker. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, pages 1–20, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23643-3. doi: 10.1007/978-3-642-23644-0_1. URL http://dx.doi.org/10.1007/978-3-642-23644-0_1.

D. Bruschi, L. Cavallaro, and A. Lanzi. Diversified process replic for defeating memory error exploits. *Performance, Computing, and Communications Conference, 2002. 21st IEEE International*, 0:434–441, 2007. ISSN 1097-2641. doi: http://doi.ieeecomputersociety.org/10.1109/PCCC.2007.358924.

R. Capizzi, A. Longo, V. N. Venkatakrishnan, and A. P. Sistla. Preventing information leaks through shadow executions. In *ACSAC*, 2008.

M. Castro, R. Rodrigues, and B. Liskov. Base: Using abstraction to improve fault tolerance. *ACM Trans. Comput. Syst.*, 21(3):236–269, Aug. 2003. ISSN 0734-2071. doi: 10.1145/859716.859718. URL http://doi.acm.org/10.1145/859716.859718.

R. Chandra, T. Kim, M. Shah, N. Narula, and N. Zeldovich. Intrusion recovery for database-backed web applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, 2011.

H. Chen, T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek. Identifying information disclosure in web applications with retroactive auditing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 555–569, Broomfield, CO, Oct. 2014. USENIX Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chen_haogang.

L. Chen and A. Avizienis. N-version programminc: A fault-tolerance approach to rellablllty of software operatlon. In *Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth International Symposium on*, pages 113–, Jun 1995. doi: 10.1109/FTCSH.1995.532621.

P. Cheng. From covariation to causation: A causal power theory. *Psychological Review, 104*, pages 367–405, 1997.

M. Chereque, D. Powell, P. Reynier, J.-L. Richier, and J. Voiron. Active replication in delta-4. In *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, pages 28–37, July 1992. doi: 10.1109/FTCS.1992.243618.

B.-G. Chun, P. Maniatis, and S. Shenker. Diverse replication for single-machine byzantine-fault tolerance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 287–292, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1404014.1404038.

J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-734-6. doi: 10.1145/1273463.1273490. URL http://doi.acm.org/10.1145/1273463.1273490.

B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: A secretless framework for security through diversity. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267336.1267344.

L. P. Cox, P. Gilbert, G. Lawler, V. Pistol, A. Razeen, B. Wu, and S. Cheemalapati. Spandex: Secure password tracking for android. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 481–494, San Diego, CA, Aug. 2014. USENIX Association. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/cox.

D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *S&P*, 2010.

A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The taser intrusion recovery system. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05. ACM, 2005.

J. Heusser and P. Malacaria. Quantifying information leaks in software. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 261–269, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0133-6. doi: 10.1145/1920261.1920300. URL http://doi.acm.org/10.1145/1920261.1920300.

P. Hosek and C. Cadar. Varan the unbelievable: An efficient n-version execution framework. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 339–353, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2835-7. doi: 10.1145/2694344.2694390. URL http://doi.acm.org/10.1145/2694344.2694390.

D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 265–276, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3174-8. doi: 10.1109/ISCA.2008.26. URL http://dx.doi.org/10.1109/ISCA.2008.26.

D. Hume. An enquiry concerning human understanding. 1748.

M. G. Kang, S. McCamant, P. Poosankam, and D. Ong. DTA++: Dynamic taint analysis with targeted control-flow propagation. In A. Perrig, editor, *NDSS 2011, 18th Annual Network & Distributed System Security Symposium*, Washington, DC, USA, Feb. 2011. Internet Society. URL http://www.isoc.org/isoc/conferences/ndss/11/pdf/5_4.pdf.

V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 121–132, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1176-2. doi: 10.1145/2151024.2151042. URL http://doi.acm.org/10.1145/2151024.2151042.

W. M. Khoo. wmkhoo/taintgrind - github, Nov. 2013. URL https://github.com/wmkhoo/taintgrind/.

D. Kim, Y. Kwon, W. N. Sumner, X. Zhang, and D. Xu. Dual execution for on the fly fine grained execution comparison. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 325–338, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2835-7. doi: 10.1145/2694344.2694394. URL http://doi.acm.org/10.1145/2694344.2694394.

T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek. Intrusion recovery using selective re-execution. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10. USENIX Association, 2010.

A. Kushnir and A. Gopnik. Young children infer causal strength from probabilities and interventions. *Psychological Science, 16 (9)*, pages 678–683, 2005.

D. Lewis. Counterfactuals. *Oxford: Blackwell*, 1973.

X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf. Caisson: A hardware description language for secure information flow. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 109–120, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993512. URL http://doi.acm.org/10.1145/1993498.1993512.

V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn. Archipelago: Trading address space for reliability and security. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 115–124, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-958-6. doi: 10.1145/1346281.1346296. URL http://doi.acm.org/10.1145/1346281.1346296.

P. Mardziel, M. S. Alvim, M. Hicks, and M. R. Clarkson. Quantifying information flow for dynamic secrets. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 540–555, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-4686-0. doi: 10.1109/SP.2014.41. URL http://dx.doi.org/10.1109/SP.2014.41.

S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 193–205, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375606. URL http://doi.acm.org/10.1145/1375581.1375606.

J. McDermott, R. Gelinas, and S. Ornstein. Doc, wyatt, and virgil: prototyping storage jamming defenses. In *Computer Security Applications Conference, 1997. Proceedings., 13th Annual*, pages 265–273, Dec 1997. doi: 10.1109/CSAC.1997.646199.

G. Miller and P. N. Johnson-Laird. Language and perception. *Cambridge: Cambridge University Press*, 1976.

P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: A software-hardware interface for practical deterministic multiprocessor replay. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 73–84, New York, NY, USA, 2009. ACM. ISBN 978-1-

60558-406-5. doi: 10.1145/1508244.1508254. URL http://doi.acm.org/10.1145/1508244.1508254.

S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using strata. *SIGPLAN Not.*, 41(11):229–240, Oct. 2006. ISSN 0362-1340. doi: 10.1145/1168918.1168886. URL http://doi.acm.org/10.1145/1168918.1168886.

S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. Pres: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 177–192, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629593. URL http://doi.acm.org/10.1145/1629575.1629593.

F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 135–148, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2732-9. doi: 10.1109/MICRO.2006.29. URL http://dx.doi.org/10.1109/MICRO.2006.29.

B. Salamat. *Multi-variant Execution: Run-time Defense Against Malicious Code Injection Attacks*. PhD thesis, Irvine, CA, USA, 2009. AAI3359500.

G. Shu, B. Sun, A. Podgurski, and F. Cao. Mfl: Method-level fault localization with causal inference. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 124–133, March 2013. doi: 10.1109/ICST.2013.31.

D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, pages 1–25, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-89861-0. doi: 10.1007/978-3-540-89862-7_1. URL http://dx.doi.org/10.1007/978-3-540-89862-7_1.

F. Sorrentino, A. Farzan, and P. Madhusudan. Penelope: Weaving threads to expose atomicity violations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 37–46, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-791-2. doi: 10.1145/1882291.1882300. URL http://doi.acm.org/10.1145/1882291.1882300.

M. Tiwari, X. Li, H. Wassel, F. Chong, and T. Sherwood. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 493–504, Dec 2009a.

M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. Complete information flow tracking from the gates up. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 109–120, New York, NY, USA, 2009b. ACM. ISBN 978-1-60558-406-5. doi: 10.1145/1508244.1508258. URL http://doi.acm.org/10.1145/1508244.1508258.

M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood. Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 189–200, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0472-6. doi: 10.1145/2000064.2000087. URL http://doi.acm.org/10.1145/2000064.2000087.

A. Tulley and S. Shrivastava. Preventing state divergence in replicated distributed programs. In *Reliable Distributed Systems, 1990. Proceedings., Ninth Symposium on*, pages 104–113, Oct 1990. doi: 10.1109/RELDIS.1990.93956.

B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine Faults in Transaction Processing Systems Using Commit Barrier Scheduling. In *ACM SOSP*, Stevenson, WA, October 2007.

K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. *ACM Trans. Comput. Syst.*, 30(1):3:1–3:24, Feb. 2012. ISSN 0734-2071. doi: 10.1145/2110356.2110359. URL http://doi.acm.org/10.1145/2110356.2110359.

N. Viennot, S. Nair, and J. Nieh. Transparent mutable replay for multicore debugging and patch validation. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 127–138, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1870-9. doi: 10.1145/2451116.2451130. URL http://doi.acm.org/10.1145/2451116.2451130.

B. Xin, W. N. Sumner, and X. Zhang. Efficient program execution indexing. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 238–248, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375611. URL http://doi.acm.org/10.1145/1375581.1375611.

A. R. Yumerefendi, B. Mickle, and L. P. Cox. Tightlip: Keeping applications from spilling the beans. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation*, NSDI'07, pages 12–12, Berkeley, CA, USA, 2007. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1973430.1973442.