Purdue University
Purdue e-Pubs

Open Access Dissertations | Theses and Dissertations

8-2018

Combatting Advanced Persistent Threat via Causality Inference and Program Analysis

Yonghwi Kwon
Purdue University

Follow this and additional works at: https://docs.lib.purdue.edu/open_access_dissertations

Recommended Citation
Kwon, Yonghwi, "Combatting Advanced Persistent Threat via Causality Inference and Program Analysis" (2018). *Open Access Dissertations*. 1989.
https://docs.lib.purdue.edu/open_access_dissertations/1989

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

COMBATTING ADVANCED PERSISTENT THREAT VIA CAUSALITY INFERENCE AND PROGRAM ANALYSIS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Yonghwi Kwon

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2018

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Xiangyu Zhang, Chair

Department of Computer Science

Dr. Dongyan Xu

Department of Computer Science

Dr. Aniket Kate

Department of Computer Science

Dr. Ninghui Li

Department of Computer Science

Approved by:

Dr. Voicu S. Popescu by Dr. William J. Gorman

Head of the Departmental Graduate Program

To my beloved family members

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor Professor Xiangyu Zhang for his invaluable guidance and generous support throughout my PhD. Research discussions that we had for 6 years enlightened me and formed me as a decent computer scientist. From very detailed technical discussions to high-level research directions, he taught and influenced me very much. I have learned how to scientifically and systematically analyze problems and construct fundamental solutions through his invaluable training. Also, I am deeply grateful to my co-advisor Professor Dongyan Xu for his generous guidance and support for my PhD. He provided me a number of valuable opportunities and guidance in presenting and communicating research ideas. He taught and showed me the importance of conveying intuitions in addition to details of research projects. He was always very supportive of my career and helped me to have more experience outside of the lab. I am deeply grateful to my advisor and co-advisor for allowing me to gain various invaluable lessons I could not have without them.

In addition, I would like to sincerely thank my committee members: Professor Aniket Kate and Professor Ninghui Li for serving committee members of my dissertation and providing many insightful comments to improve the dissertation. Their suggestions also helped me frame my research in a more accessible way so that it would appeal to a broader audience.

I would like to acknowledge my collaborators and friends. First, Professor Kyu Hyung Lee helped and taught me in various ways throughout my PhD. I have learned a lot about how to develop research topics and projects. I will remember the time we spent on research discussions. Dr. Yunhui Zheng inspired me and showed how to be a strong and independent researcher. In particular, he showed me how to collaborate with others. He has been always supportive, resourceful, and sincere. I also thank Professor Brendan Saltaformmagio for being my best friend throughout my PhD. I will remember the great time we had in Chicago

and New Orleans. Spending 6 years in the small college town of Indiana would be difficult without my great lab mates: Weihang Wang, Fei Wang, Dohyeong Kim, I Luk Kim, Chunghwan Kim, Zhongshu Gu, Taegyu Kim, and Wei You. I learned a lot from them throughout projects and discussions. I feel very fortunate that I had a chance to get to know them. Last but not least, I would like to thank Kristine Johnson for being a great friend throughout my PhD and providing the best brownies.

More importantly, I would like to thank my parents, Sejung Kwon and Aeran Kim, unconditionally support me and my PhD. They taught me to stay positive and focused when negative rules. They patiently waited for me to finish my degrees and always support me without any doubt. Their unconditional support and love helped me get through whenever I got frustrated.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABBREVIATIONS

API	Application Program Interface
APT	Advanced Persistent Threat
AST	Abstract Syntax Tree
ASLR	Address Space Layout Randomization
CC	Counterfactual Causality
CFG	Control Flow Graph
CFI	Control Flow Integrity
CG	Causal Graph
CLOC	C lines of source code
CPU	Central Processing Unit
CSS	Cascading Style Sheets
DARPA	Defense Advanced Research Projects Agency
DOM	Document Object Model
DPI	Deep Packet Inspection
DE	Decoding and Encoding
DF	Decoding Frontier
DU	Definition-Use
ETW	Event Tracing for Windows
FP	False positive
FN	False negative
GB	Gigabyte
HB	Happens-before
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol

IR	Intermediate Representation
IRC	Internet Relay Chat
IT	Information Technology
JS	JavaScript
LLVM	The LLVM compiler infrastructure project which is a collection of modular and reusable compiler and toolchain technologies
Max-SAT	Maximum Satisfiability Problem
OS	Operating System
PC	Program Counter
ROP	Return Oriented Programming
SAT	Satisfiable
SMT	Satisfiability Modulo Theories
TB	Terabyte
TC	Transparent Computing Project (by DARPA)
UI	User Interface
UNSAT	Unsatisfiable
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	EXtensible Markup Language
Z3	An SMT Solver by Microsoft

ABSTRACT

Yonghwi, Kwon PhD, Purdue University, August 2018. Combatting Advanced Persistent Threat via Causality Inference and Program Analysis. Major Professors: Xiangyu Zhang and Dongyan Xu.

Cyber attackers are becoming more and more sophisticated. In particular, Advanced Persistent Threat (APT) is a new class of attack that targets a specific organization and compromises systems over a long time without being detected. Over the years, we have seen notorious examples of APTs including Stuxnet which disrupted Iranian nuclear centrifuges and data breaches affecting millions of users. Investigating APT is challenging as it occurs over an extended period of time and the attack process is highly sophisticated and stealthy. Also, preventing APTs is difficult due to ever-expanding attack vectors.

In this dissertation, we present proposals for dealing with challenges in attack investigation. Specifically, we present LDX which conducts precise counter-factual causality inference to determine dependencies between system calls (e.g., between input and output system calls) and allows investigators to determine the origin of an attack (e.g., receiving a spam email) and the propagation path of the attack, and assess the consequences of the attack. LDX is four times more accurate and two orders of magnitude faster than state-of-the-art taint analysis techniques. Moreover, we then present a practical model-based causality inference system, MCI, which achieves precise and accurate causality inference without requiring any modification or instrumentation in end-user systems.

Second, we show a general protection system against a wide spectrum of attack vectors and methods. Specifically, we present A2C that prevents a wide range of attacks by randomizing inputs such that any malicious payloads contained in the inputs are corrupted. The protection provided by A2C is both general (e.g., against various attack vectors) and practical (7% runtime overhead).

1 INTRODUCTION

Cyber attackers are becoming more and more sophisticated. In particular, Advanced Persistent Threat or APT is a special kind of attacks that leverages most stealthy and advanced attack methods. They lurk in victim systems for a long time (e.g., from weeks to months) without being detected while exfiltrating secrets and/or disrupting systems. We have seen many high-profile APT attacks including STUXNET which targets the most dangerous infrastructure, nuclear plants, and compromised more than hundreds of thousands of systems through multiple steps. It lurked in the systems for years while silently updating, installing backdoors, and exfiltrating information. It was commented that the attack could have caused a nuclear disaster more catastrophic than *Chernobyl*. Unfortunately, combatting APT attacks is particularly difficult because (1) the attacks happen for a long time hence even tracking and understanding what attackers did is challenging and (2) they leverage zero-day vulnerabilities which are not disclosed hence proactive prevention of APT is challenging.

This dissertation presents fundamental approaches that systematically prevent and analyze APT attacks. Specifically, we analyze state-of-the-art attack prevention and analysis techniques and identifies advantages and disadvantages. To this end, we realize the original concept of counter-factual causality which was first introduced by David Hume in the 18^{th} century can be effective in APT attack investigation and existing techniques are approximations of the counter-factual causality. In addition, we identify that existing attack prevention approaches are mostly attack-vector specific hence they are often ineffective in preventing zero-day exploits. As a result, we develop a novel causality inference technique that can precisely identify causal relationships between processes, files, and network addresses. Also, we develop a novel attack vector agnostic exploit injection attack prevention technique to thwarts zero-day exploits. By leveraging those fundamental techniques, we

can achieve the complete protection and analysis of APT attacks which happen over a long time and leverage stealthy techniques.

In particular, this dissertation includes (1) LDX [1], a novel counter-factual causality inference, which strictly follows the original definition of counter-factual causality first introduced by David Hume in 18th century, (2) MCI [2], a novel model-based causality inference technique built on top of LDX, that infers causality for enterprise systems without any instrumentation and modification of underlying systems such as kernel, and (3) A2C [3], a novel exploit injection attack prevention technique, that can prevent zero-day exploits which is not known hence existing attack vector specific techniques cannot prevent.

1.1 Dissertation Statement

Accurate attack investigation and general protection against advanced and sophisticated attacks can be achieved by leveraging causality inference and fundamental weaknesses of the attacks.

1.2 Contributions

The contributions of this dissertation are as follows:

- We develop a practical causality inference system, LDX [1] that can conduct a faithful counterfactual causality inference to determine dependencies between system calls (e.g., between input and output system calls) and allow investigators to determine the origin of an attack (e.g., receiving a spam email) and assess the consequences of the attack. LDX is *4 times more accurate and 2 orders of magnitude faster* (6% runtime overhead) than state-of-the-art taint analysis techniques.

- Expanding beyond LDX, we have proposed a model-based causality inference system, MCI [2]. MCI is practical as it does not require any modification or instrumentation to end-user systems, and it is more accurate and precise (0.1% FP/FN) than

the previous state-of-the-art technique BEEP [4] which does require instrumentation (12.8% FP/0.3% FN).

- We have designed a novel software protection system, A2C [3], that prevents a wide range of attacks by randomizing inputs such that *any malicious payloads contained in the inputs are corrupted*. The protection provided by A2C is both general (e.g., against various attack vectors including buffer-overflow, integer-overflow, use-after-free, type-confusion, and ROP) and practical (7% runtime overhead).

1.3 Dissertation Organization

This dissertation includes three fundamental primitives for the investigation and prevention of advanced cyber-attacks: LDX which proposes a novel causality inference technique (Chapter 2), MCI which develops a novel model-based causality inference technique for enterprise environment (Chapter 3), and A2C which is an attack vector agnostic exploit injection attack prevention technique (Chapter 4).

1.4 Dissertation Overview

Prior to my work, the two most widely used state-of-the-art techniques for attack investigation were taint analysis and audit-logging. Taint analysis tracks program dependencies by monitoring the data propagation of individual instructions. Audit-logging focuses on dependencies between syscalls exposed through explicit syscall arguments (e.g., file handles). For example, they consider syscalls on the same file or within the same process causally related. Unfortunately, taint analysis suffers from significant performance overhead as it needs to monitor every instruction. Moreover, both taint analysis and audit-logging are inaccurate as taint analysis has difficulty handling control dependencies and the assumptions made in audit-logging (e.g., all output syscalls are causally related to all input syscalls within a process) are too coarse-grained, leading to a large number of bogus dependencies.

Counter-factual causality, first introduced in the 18^{th} century by David Hume [5], can be used to describe the desired causal analysis in an attack investigation. Specifically, given two events, *a latter event is causally dependent on a preceding event if changes at the preceding event lead to state differences in the latter event*. To this end, we realized that the limitations of taint analysis and audit-logging stem from their *imprecise approximations* of counter-factual causality. My research pioneered building techniques that implement precise counter-factual causality for cyber attack investigation.

In addition, to build general protection against ever-evolving cyber attacks, my research breaks a common critical step of most attacks: *malicious payload injection and execution*. In particular, we exploit a fundamental characteristic of malicious payloads: they are designed with strict semantic assumptions about the execution environment (e.g., platform or architecture), hence they are *particularly brittle* to any mutation.

1.4.1 Conducting Faithful Counter-factual Causality

We take a fundamental approach: *adapting the original counter-factual causality concept in the context of program and program execution*. LDX [1] conducts faithful counter-factual causality inference on computer systems via *dual execution*. Specifically, it runs two executions in parallel — the original execution and its mutated version with mutations on input syscalls. Then, it observes differences at output syscalls. Any difference indicates causality between the mutated input syscalls and the output syscalls. Due to the mutation LDX introduces, the mutated execution may take a different path, leading to a different sequence of executed syscalls, when compared with the original execution. Hence, a fundamental challenge of LDX is to align the two executions so that they can be compared at the same execution point, because comparing executions at misaligned points leads to incorrect causality (i.e., FP/FN). To this end, we designed a novel runtime counter derived from program structure. The counter is not a simple logic timestamp, but rather denotes execution points by ensuring an important key property: The counter value indicates the relative progress of executions, meaning that an execution with a larger counter value must

be ahead of another execution with a smaller counter value with respect to program structure. The counter facilitates alignment of two executions, enabling precise and efficient causality inference. Evaluation on a large set of real-world applications, including *Apache web server*, shows that LDX is *4 times more accurate* and *2 orders of magnitude faster* than state-of-the-art taint analysis techniques.

1.4.2 Model-based Causality Inference for Practical Attack Provenance

The primary hindrance of existing techniques, including LDX, for attack provenance is their requirement of changing end-user systems such as program instrumentation and kernel modification. In contrast, existing automata-based techniques do not require instrumentation. They work by identifying program behaviors (e.g., file downloading) from a concrete log (e.g., a syscall log). They construct automata that represent the behaviors. Then, they parse a log generated from an execution with the automata to determine whether the behaviors are exhibited in the log. However, they do not take dependencies into account; for instance, they may detect two behaviors that are "download a file" and "send a message," while the causal relationship between these two behaviors is not exposed.

MCI [2] is a model-based causality inference technique for attack provenance that directly works on syscall logs *without requiring any end-user program instrumentation or kernel modification*. For each program, it uses LDX to acquire precise *causal models* for a set of primitive operations (e.g., opening a file). A causal model is a sequence of syscalls annotated with *inter-dependencies (causality) between the syscalls within the model*, where some of the inter-dependencies are caused by memory operations and hence implicit at the syscall level. During deployment, MCI parses the existing audit-logs into concrete model instances to derive causality. To this end, parsing syscall logs with causal models with implicit dependency information leads to two prominent challenges: *language complexity* and *ambiguity*. First, to express complex inter-dependencies annotated in causal models, expressive grammar is required while more expressive grammar describes more complex language (e.g., context-free or context-sensitive) and hence leads to higher cost in pars-

ing. Second, some syscalls can be parsed by multiple models that share common parts (e.g., common prefixes). In such cases, it is difficult to decide which model is the right one. As different causalities are derived from different models, the ambiguity problem may lead to incorrect causality (i.e., FP/FN). To solve these challenges, we designed a novel model parsing algorithm called *segmented parsing* that can handle multiple model complexity levels (e.g., regular, context-free, and context-sensitive) and substantially mitigate the ambiguity problem by leveraging *explicit dependencies* that can be directly derived from the log (e.g., dependencies caused by file handles). Specifically, MCI first obtains a *model skeleton* of each causal model. A model skeleton consists of syscalls with explicit dependencies. The skeleton partitions a model into *model segments* that can be described and parsed by automata. Without requiring any changes to end-user systems, MCI recovers causality with close to 0% FP/FN for most applications (the worst case: 8.3% FP and 5.2% FN). More importantly, causal models have *composability* such that models for primitive operations can be composed together to describe complex system-wide attack behaviors. For example, primitive models for "Edit", "Copy", "Paste", and "Save" can compose a new model that represents a complex user behavior "Edit→Copy→Edit→Paste→Edit→Save" (e.g., potential information exfiltration). Evaluation on *attack cases created by security professionals in the DARPA TC program* shows that attack causal graphs generated by MCI are more precise than those generated by the previous state-of-the-art system BEEP [4] that requires instrumentation.

1.4.3 Corrupting Malicious Payloads via Input Perturbation

A2C [3] exploits *the brittleness* of malicious payloads to provide general protection. It corrupts malicious payloads by encoding all inputs from untrusted sources at runtime. However, the encoding may break program execution on benign inputs as well. To assure that the program continues to function correctly when benign inputs are provided, We developed a static analysis technique that identifies all the places that read and process inputs and selectively inserts decoding logic at some of those places. Specifically, decoding only

occurs when the use of the inputs cannot be exploited. For instance, when inputs in a byte array are copied to an integer array, each byte of the inputs is padded with 3 zero bytes (as an integer is 4 bytes on 32-bit machines) before it is stored into the integer array. Constructing a meaningful payload with 3 zero bytes in every four bytes is extremely difficult, if not impossible. To this end, we proposed a novel *constraint solving algorithm* which identifies operations that make inputs no longer exploitable, such as the copy operation from a byte array to an integer array. The operations essentially divide the state space of a program into *exploitable* and *post-exploitable* sub-spaces because the program state before the operation is exploitable, but no longer so after the operations. Therefore, A2C decodes the mutated values only when they are transmitted from the exploitable space to the post-exploitable space. Notably, the exploitable space is much smaller than the post-exploitable space — making A2C highly efficient. A2C successfully achieves general protection for a large set of real-world programs, including *Apache web server* against a variety of attacks (e.g., heap spraying, use-after-free, buffer-overflow, integer-overflow, and type-confusion) with low overhead (6.94%).

2 LDX : CAUSALITY INFERENCE BY LIGHTWEIGHT DUAL EXECUTION

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 non-determinism, 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.

2.1 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 [6–10].

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 [11–13] rely on detecting syntactic patterns of control dependences and hence are incomplete. *Second*, existing techniques are expensive (e.g., a few times slow-down [8]), 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 [14], *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 non-determinism 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 [15]. (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) [16]. 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* [15]. The slow-down reported in [16] 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 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 [8] and TAINTGRIND [17]. 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 [18], 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.

2.2 Counterfactual Causality

Counterfactual causality (CC) [14, 19] 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 [20–22]. 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.

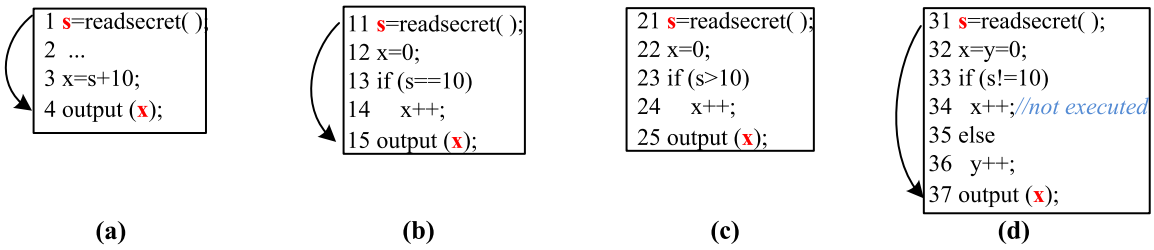


Figure 2.1.: Examples to illustrate the comparison of counterfactual causality and program dependences. Arrows denote strong causalities between x at the sink and s at the source.

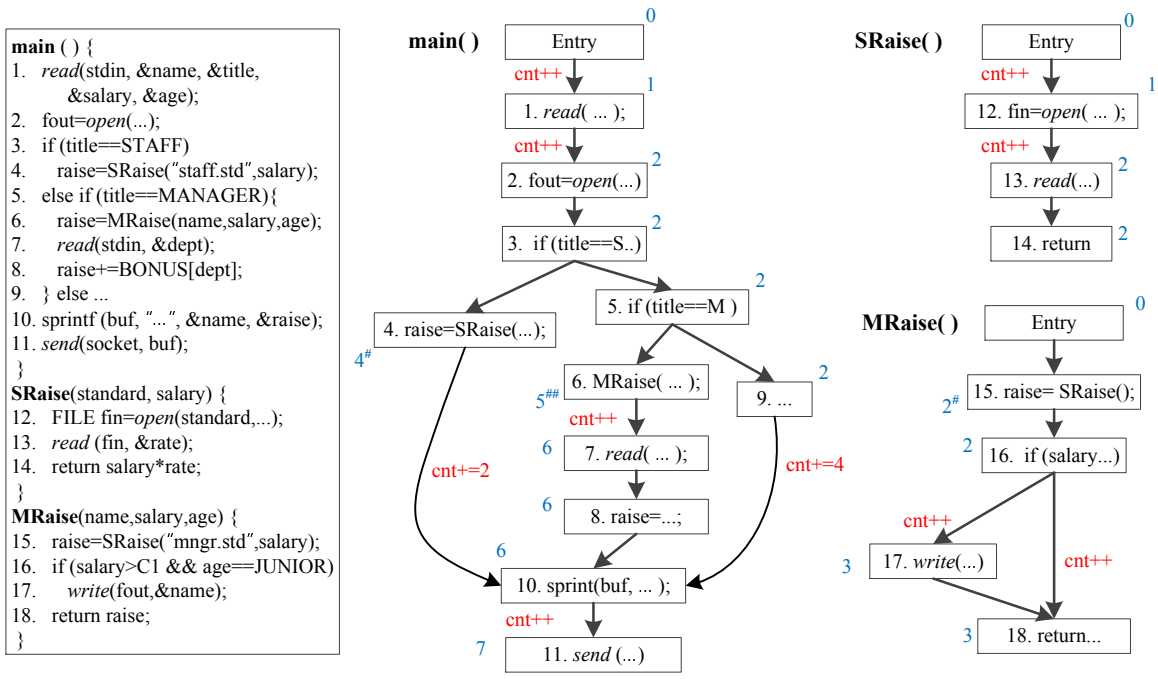
Most existing causality inference techniques including dynamic tainting are based on tracking program dependences, especially data dependencies. Two events are causally related if there is a dependence path between them during execution. As we discussed in

Section 3.1, 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. 2.1 (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. 2.1 (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 [23], 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. 2.1 (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



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

Figure 2.2.: Illustrative example. The code along control flow edges represents instrumentation. #cnt+=2 inside **SRaise()**; ##cnt+=3 in **MRaise()**.

the adversary to infer $s = 10$. It is a strong CC: any change 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 [24].

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.

2.3 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 2.3.: Syscall traces and the synchronization action sequence by LDX for the example in Fig. 2.2 with `title` the secret. The shaded entries are aligned.

Example. Consider the program in Fig. 2.2 (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. 2.2 (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. 2.3 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. 2.3 that the third and the fourth syscalls in both executions have the same counter. In fact, both are syscalls 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.

2.4 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 2.5 and indirect calls in Section 2.6).

2.4.1 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.

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 further 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

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

function INSTRUMENTPROG
  for $\langle N_i, E_i \rangle$ in reverse topological order of the call graph **do**
    INSTRUMENTFUNC ($\langle N_i, E_i \rangle$)
  end for
end function

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

function INSTRUMENTFUNC
  for each node $n \in N$ **do**
    $cnt[n] \leftarrow 0$
  end for
  for node $n \in N$ in topological order **do**
    $cnt[n] \leftarrow max_{p \to n \in E}(cnt[p])$
    if n is a syscall **then**
      $cnt[n] \leftarrow cnt[n] + 1$
    end if
    for each edge $p \to n \in E$ **do**
      if $cnt[p] \neq cnt[n]$ **then**
        instrument $p \to n$ with "$cnt+=$"$\cdot cnt[n] - cnt[p]$
      end if
    end for
    if n is a call to user function F_x **then**
      $cnt[n] \leftarrow cnt[n] + FCNT[F_x]$
    end if
  end for
  $FCNT[F] \leftarrow cnt[$exit node of $F]$
end function

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. 2.2, 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:          end while
6:          if cnt_m < ready_s ∨ O_s.sys_id ≠ sys_id ∨ O_s.args ≠ args then
7:              report causality
8:          end if
9:      end if
10:     r ← SYSCALL (sys_id, args)
11:     Q_m.enq(⟨cnt_m, sys_id, args, r⟩)
12:     ready_m ← cnt_m
13:     return r
14: end function
```

2.4.2 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. 2.2, 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 [24].

Syscall Handling. LDX's policy of handling syscalls is similar to that in *dual execution* (DualEx) [16]. 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 [16].

Dual Execution Model Comparison between LDX and DualEx [16]. 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.

2.5 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

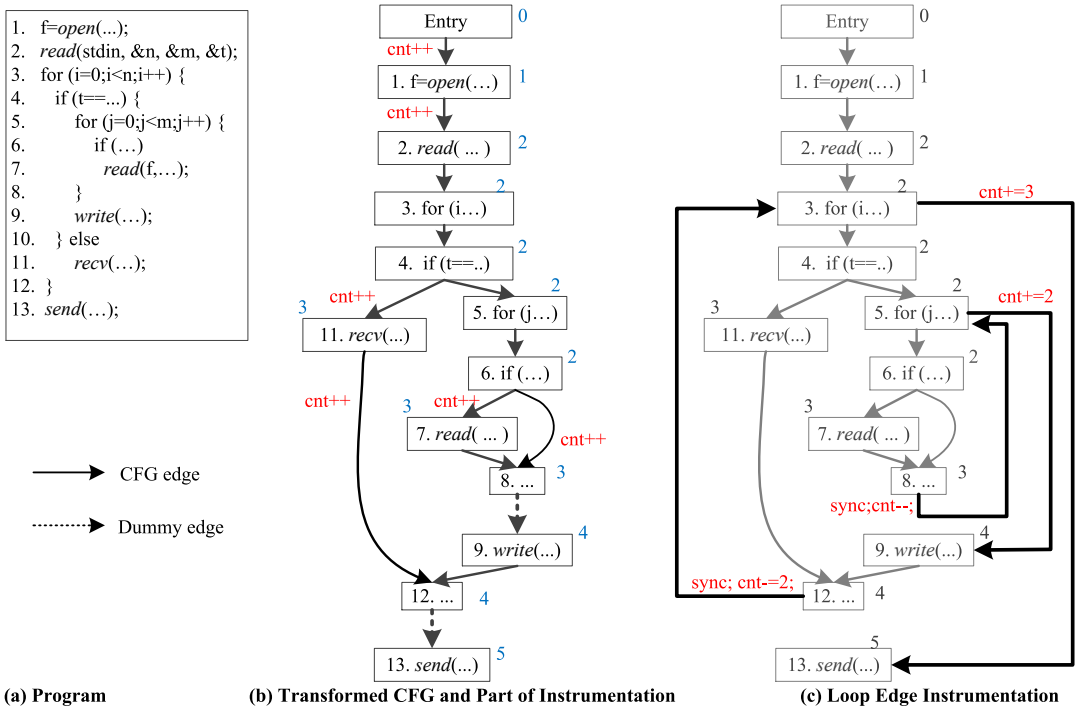


Figure 2.4.: Loop example

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 *i*th iteration of the master with the *i*th 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 entered 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 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,

Algorithm 3 Counter instrumentation with loops

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

function INSTRUMENTFUNCWITHLOOP
  for each back edge $t \to h \in E$ **do**
    Let $h \to n$ be the exit edge of the loop
    $E \leftarrow E - \{t \to h,\ h \to n\}$ ▷ Remove loop exit and back edges
    $E \leftarrow E \cup \{t \to n\}$ ▷ Add dummy edge
  end for
  INSTRUMENTFUNC($\langle N, E \rangle$)
  remove all dummy edges and their instrumentation
  restore all the removed edges in the original CFG
  for each original back edge $e : t \to h$ **do**
    instrument e with "$sync(); cnt -= $"$\cdot cnt[t] - cnt[h]$
  end for
  for each original loop exit edge $e : h \to n$ **do**
    instrument e with "$cnt += $"$\cdot cnt[n] - cnt[h]$
  end for
end function

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 $t \rightarrow h$, 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 $h \rightarrow n$, 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. 2.4 (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 $8 \rightarrow 5$ and $12 \rightarrow 3$, the loop exit edges $3 \rightarrow 13$ and $5 \rightarrow 9$ are removed. Dummy edges $8 \rightarrow 9$ and $12 \rightarrow 13$ 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 $8 \rightarrow 5$ 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 $5 \rightarrow 9$ 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. 2.5 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.

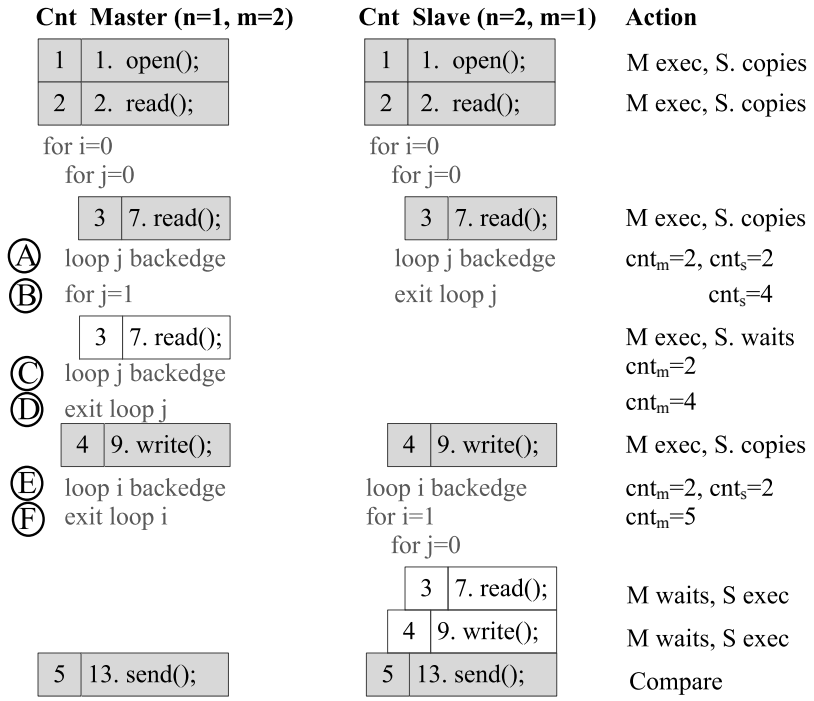


Figure 2.5.: Syscalls and the sequence of synchronizations by LDX for the example in Fig. 2.4 with `n` and `m` the sources. The shaded entries are aligned. The indentation shows the loop nesting.

At A, the two executions are synchronized and counters are reset to 2. However at B, 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 C, the master finishes the second iteration of j and its counter is reset to 2. At D, the master also exits loop j and its counter is incremented to 4, which aligns the two syscalls at line 9. At E, 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 F, 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.

2.6 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 [24].

2.7 Handling Concurrency and Library Calls

LDX supports real concurrency, which is completely different from DualExec [16]. 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 4.6, 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.

Table 2.1.: Benchmarks and Instrumentation

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

* It shows avg/max

2.8 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 2.1. 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 [24].

Instrumentation Details. Table 2.1 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 2.1 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 dif-

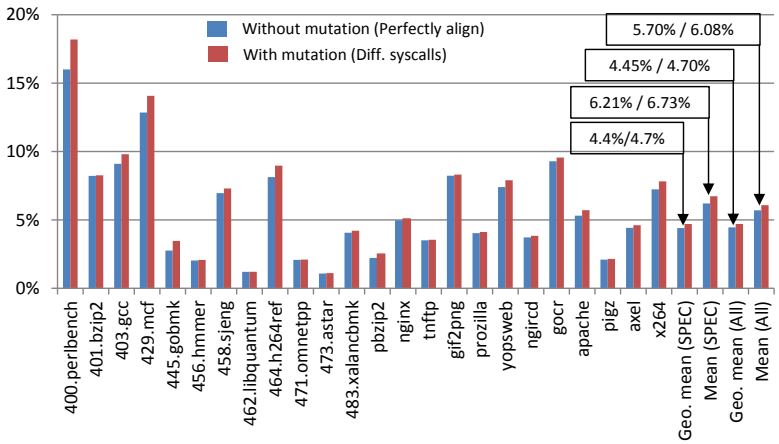


Figure 2.6.: Normalized overhead of LDX

ferences 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.

2.8.1 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 throughtput. 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. 2.6. 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 [8], 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* [16].

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 2.1 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 2.2.: 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%)	-

2.8.2 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 2.2 presents the results. Symbol '0' 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.

2.8.3 Effectiveness of Causality Inference

Comparison with Dynamic Tainting. We first compare LDX with TAINTGRIND [17] and LIBDFT [8] [1]. 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 2.3. The three columns in the middle report the number of tainted sinks. The last column shows the total number of sinks encountered during execution.

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

Table 2.3.: 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 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 2.2, 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 2.7). 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 2.4, 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 execution 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 2.4.: Effectiveness for 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 [24]. 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 [24].

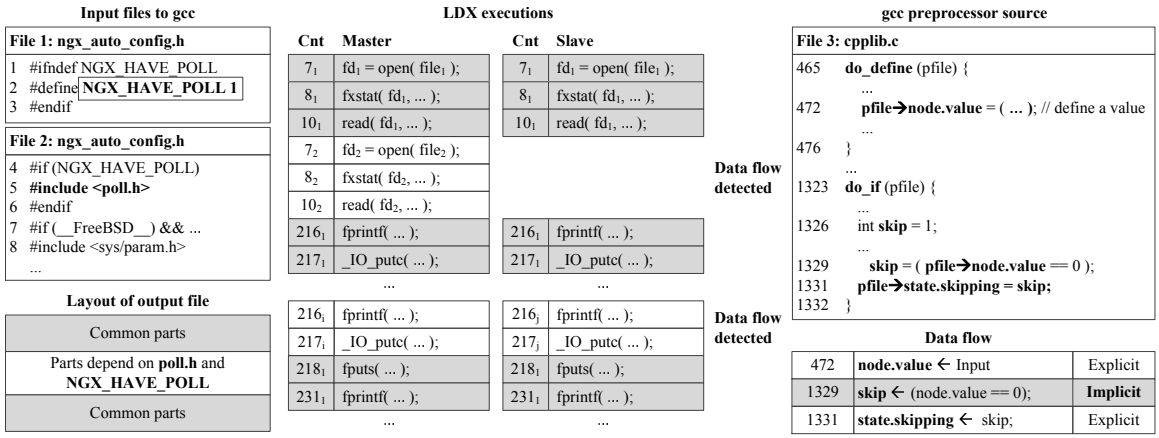


Figure 2.7.: Case study on 403.gcc. Input files on the left; relevant gcc code on the right; dual execution in the middle.

2.8.4 Case Studies

403.gcc. In this study, we use the source code of `nginx` as input. Fig. 2.7 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. 2.7) 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. 2.7 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 de-

pendences 472 → 1329 and 1329 → 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 [24].

2.9 Related Work

Dual Execution. LDX is closely related to dual execution [16]. The main differences are the following. (1) LDX is very lightweight (6.08% overhead) whereas [16] relies on the expensive execution indexing [15], causing 3 orders of magnitude slowdown. (2) LDX allows threads to execute concurrently whereas [16] 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 2.4.2. TIGHTLIP [18] 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 [25–33]. The premise is similar to n-version programming [34], 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 [18,35–38] of execution replication by detecting differences among replicas. There are also works in execution replay [39–47]. In contrast, LDX align different paths during execution. RAIL [48] 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 [49] 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 [49] relies on heuristics to tolerate non-determinism.

Dynamic Taint Tracking. Most dynamic tainting techniques [6–10, 50] work by tracking instruction execution and hence are expensive. They have difficulty handling control dependences [11]. Some have limited support by detecting patterns [12] or handling special dependences [23]. In particular, [50] 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 [11,51–53] 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 [54–57] have been proposed to speed up or improve accuracy of taint analysis.

Secure Multiple Execution (SME). SME [58–60] 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 [61–63] 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 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.

3 MCI : MODELING-BASED CAUSALITY INFERENCE IN AUDIT LOGGING FOR ATTACK INVESTIGATION

In this chapter, we develop a model-based causality inference technique for audit logging that does not require any application instrumentation or kernel modification. It leverages a recent dynamic analysis, dual execution (LDX), that can infer precise causality between system calls but unfortunately requires doubling the resource consumption such as CPU time and memory consumption. For each application, we use LDX to acquire precise causal models for a set of primitive operations. Each model is a sequence of system calls that have inter-dependences, some of them caused by memory operations and hence implicit at the system call level. These models are described by a language that supports various complexity such as regular, context-free, and even context-sensitive. In production run, a novel parser is deployed to parse audit logs (without any enhancement) to model instances and hence derive causality. Our evaluation on a set of real-world programs shows that the technique is highly effective. The generated models can recover causality with 0% false-positives (FP) and false-negatives (FN) for most programs and only 8.3% FP and 5.2% FN in the worst cases. The models also feature excellent composibility, meaning that the models derived from primitive operations can be composed together to describe causality for large and complex real world missions. Applying our technique to attack investigation shows that the system-wide attack causal graphs are highly precise and concise, having better quality than the state-of-the-art.

3.1 Introduction

Cyber-attacks are becoming increasingly targeted and sophisticated [64]. A special kind of these attacks, called Advanced Persistent Threat (APT), can infiltrate into target systems in stages and reside inert for a long time to remain undetected. It is important

to trace back attack steps and understand how an attack unfolds [65]. In the mean time, identifying the entry point of the attack and understanding the damage to the victim can be critical to recovering the victim system from the intrusion and also preventing future compromises.

Causality analysis techniques [46, 47, 66–68] are widely used in attack investigation. They analyze audit logs generated by operating system level audit logging tools (e.g., Linux Audit [69], Event Tracing for Windows [70], and DTrace [71]) and correlate system events, e.g., system calls (syscalls) to identify causal relations between system subjects (e.g., processes) and system objects (e.g., files, network sockets). Such capability is particularly important in cyber-attack investigation where causality of malicious events reveals attack provenance. For example, when an attacker exploits vulnerabilities and executes malicious payloads, causality analysis can identify such vulnerable interfaces including input channels that accept malicious inputs from the user or the network. Moreover, given a set of malicious or suspicious events, it can identify all the events that are causally related to the given set of events. Essentially, these events depict the source of the attack and/or the damage induced by the attacker. However, syscall based analysis has a major limitation: dependence explosion [4]. For a long-running process, an output event (e.g., creating a malicious file) is assumed to be causally related to all the preceding input events (e.g., file read and network receive). This conservative assumption causes significant false causal relations.

Some recent works [4, 72–74] focus on collecting enhanced information at run-time to avoid dependence explosion and enable accurate attack investigation. For instance, BEEP [4] and ProTracer [72] train and instrument long-running applications to capture information of fine-grained execution units in addition to syscalls. MPI [74] asks the user to annotate important data structures in applications' source code to enable semantic aware execution partitioning. Additionally, Bates at el. [75] propose a general provenance-aware framework called Linux Provenance Module (LPM) that allows users to define custom provenance rules. The major hindrance of these techniques in practice is their require-

ments of changing end-user systems, such as instrumenting user applications, installing new runtime support, kernel modules, and even changing the kernel itself.

Taint analysis [8, 17, 76] is another approach that can track causal relations (e.g., information flow) between system components (e.g., memory objects, files, and network sockets). However, whole system tainting is too computationally expensive (over 3x slow down [77, 78]) to be deployed on production systems. Additionally most taint analysis techniques cannot handle implicit flow, resulting in false-negatives.

In this chapter, we propose MCI, a novel causality inference technique on audit logs. Our technique *does not require any changes on the end-user system, nor any special operations during system execution*. The end-user only needs to turn on the audit logger shipped with the operating system (e.g., Linux Audit, Event Tracing for Windows, and DTrace). If the user detects a security incident, she only needs to provide the syscall log and program binaries from the victim system (or a disk image) to a forensic expert.

In off-line attack investigation, which is often done by the forensic expert, MCI precisely infers causality from a given system call log by constructing causal models and parsing the log with the models. Fig. 3.1 shows a high level overview of how MCI works. MCI consists of two phases: (1) causality annotated model generation, and (2) model parsing. First, MCI generates causal models by leveraging LDX [1] which is a dual-execution based system that can infer causality by mutating input syscalls and then observing output changes. In this phase, MCI takes two inputs: a program binary and typical workloads. MCI's model constructor automatically runs LDX and analyzes its results to construct models. Models are expressive and capable of representing fine-grained dependencies including invisible at the syscall level (e.g., dependencies induced by memory operations). The models can be pre-generated (for widely used applications) or generated on demand after an incident. Second, during investigation MCI identifies causal relations between events in a given syscall log collected from a victim system by parsing the log with the models. The derived precise dependencies are critical for attack investigation.

In summary, we make the following contributions:

- We propose a novel technique for precise causality inference that directly works on audit logs without requiring any changes or setup on end-user systems. We only require program binaries and the audit log from the victim system after the incident.
- We perform a comparative study using a real-world example to illustrate the merits and limitations of existing approaches.
- We propose to leverage LDX [1] to identify fine-grained causality from program execution. Using the generated causality information, we construct causal models annotated with fine-grained dependencies. We study the model complexity needed to describe causalities in audit logging.
- We develop a novel model parsing algorithm that can handle multiple model complexity levels and substantially mitigate the ambiguity problem inherent in model-based parsing.
- We perform thorough evaluation of MCI on a set of real-world applications. The results show that the generated models can recover causality with close to 0% FP and FN for most applications and the worst FP rate 8.3% and the worst FN rate 5.2%. Model construction and model parsing have reasonable overhead and scale to week-long and even month-long workloads. Applying MCI to attack investigation shows that our models have very nice composibility such that small models can be composed together to describe complex system-wide attack behaviors. Our attack causal graphs are even more precise than those generated by a state-of-the-art system [4].

3.2 Background and Motivation

In this section, we use an insider information leak attack case to illustrate the limitations of existing attack provenance analysis techniques, and then to motivate our work.

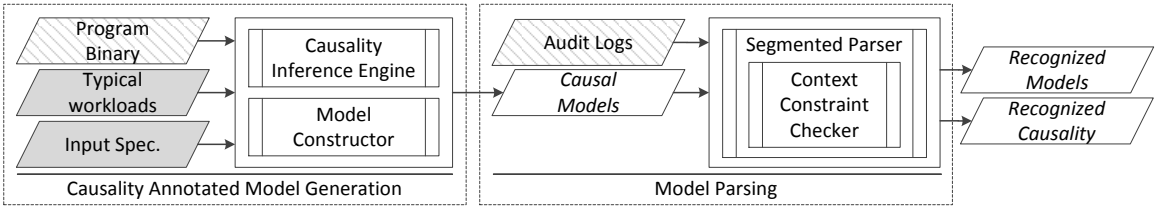


Figure 3.1.: Overview of MCI's off-line causality inference. Audit Logs and Program Binaries are provided from the end-user, workloads and input specifications are generated by an attack investigator (e.g., a forensic expert), and other components are automatically generated by MCI.

3.2.1 Motivating Example

We use a data exfiltration of confidential company data by an employee. Insider attacks are the dominant reason for data breach incidents in 2016 [79, 80].

Assume John is a project manager who has access to confidential data. John was bribed by a competitor company and attempts to breach some confidential data. However, John's company forbids copying data to removable media such as USB stick. Furthermore, the company inspects all incoming/outgoing network traffic via deep packet inspection (DPI) [81–83] to prevent exfiltration of confidential data and to block malicious network traffic from outside. To bypass the packet inspection, John decides to use the GPG encryption algorithm [84] to encrypt data before sending it.

GnuPG Vim plug-in. To use GPG encryption, John installed a `Vim` plug-in `GnuPG` [85], which enables transparent editing of gpg encrypted files. When he opens a file encrypted by gpg [84] which is an encryption utility supported by most operating systems with the GNU library (e.g., Linux, FreeBSD, and MacOS), the `GnuPG` plug-in automatically decrypts and passes the decrypted data to `Vim` so that the user can edit the contents of the encrypted file. The plug-in automatically encrypts the contents when the user saves the gpg file.

Attack Scenario. John uses `Vim` equipped with the `GnuPG` plug-in to open three confidential files, *data1, data2*, and *data3*. He also opens *out.gpg* in order to store confidential data in an encrypted format. Then he copies a few lines from *data2* using the `Vim` command

`'v'` to select characters and `'y'` to copy them to the clipboard buffer (i.e., `Vim`'s default register). Then he finds out the information in *data3* is more up-to-date. He thus copies lines from *data3* that overwrite the contents from *data2*. Later, he pastes the copied lines to *out.gpg*, saves the file in an encrypted format and terminates `Vim`. Note that, when he saves *out.gpg*, the `GnuPG` plug-in actually creates a new file (inode:8) and renames it to *out.gpg* so that the original *out.gpg* file (inode:4) is replaced by a new file (inode:8). Observe that the inode numbers of the original *out.gpg* file and the new file are different. Finally, he sends the encrypted *out.gpg* to a server outside the enterprise network.

This data breach incident is later detected, and a forensic analysis team starts to investigate the incident. Now, we introduce existing causal analysis based forensic techniques and discuss how they work on this attack.

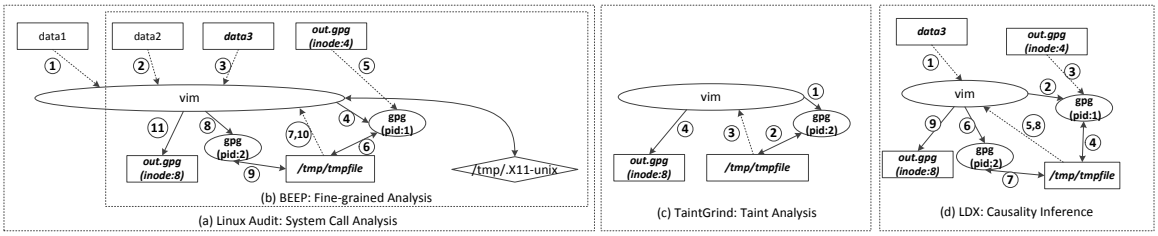


Figure 3.2.: Motivating example: Insider theft breaches confidential data using Vim and GPG

.

3.2.2 Existing Approaches and Limitations

System Call based Analysis. Most causal analysis techniques use syscall logging tools to record important system events at runtime and then analyze recorded events to identify causal relations between system subjects (e.g., process) and system objects (e.g., file or network socket). Syscall logging tools are shipped with most operating systems. For example, Linux Audit [69] is a default package in Linux and MacOS distributions, DTrace [71] is available in FreeBSD, and Event Tracing for Windows (ETW) [70] comes with Windows.

Syscall based analysis has been studied in a number of works [46, 47, 66–68]. For instance, BackTracker [66] and Taser [46] propose backward and forward analysis techniques in order to analyze syscall logs and construct causal graphs for effective attack investigation. The constructed causal graphs show system subjects and objects that involved in attacks, and their causal relations.

Fig. 3.2-(a) shows a provenance graph generated from the syscall log collected during the data breach incident discussed in the previous section. To understand the incident in detail, a security analyst first identifies the *out.gpg* file (inode:8) which contains confidential data. Then the analyst finds the system components that are causally related to the file from the graph in the backward direction (time-wise). Observe that it was `Vim` that wrote the file (⑪). Before that, `Vim` read */tmp/tmpfile* (⑩) which was written by "gpg" (⑨). The "gpg" process (pid:2) was forked by `Vim` (⑧). Before the fork, the `Vim` process reads */tmp/tmpfile* (⑦) which was written by another "gpg" process (pid:1) (⑥). "gpg" previously read the original *out.gpg* file with a different inode number (inode:4) (⑤) and the "gpg" process (pid:1) was forked by `Vim` (④) as well. There are also other files that `Vim` read, including *data3* (③), *data2* (②), and *data1* (①).

Note that Fig. 3.2-(a) contains many false dependencies such as dependencies between the `Vim` process and files *data1*, *data2*, and */tmp/.X11-unix* which is a socket for XWindow. The coarse-granularity of processes leads to this false dependency problem as it simply considers an output event is dependent on all the preceding input events in the process.

Execution Unit based Analysis. False dependencies in syscall based analysis are a major obstacle for attack investigation as it often causes the dependency explosion problem [4], which is a problem of having an excessive number of dependencies, with most of them being bogus. It makes investigation challenging, often leading to wrong conclusions. To address the problem, BEEP [4] and ProTracer [72] propose to divide a long-running process to autonomous execution units. In this way, an output event is only dependent on the preceding input events within the same execution unit. BEEP and ProTracer also detect inter-unit dependencies introduced via memory objects. ProTracer is a variant of BEEP that can significantly reduce runtime and space overhead while the effectiveness of attack

analysis remains the same because they share the same mechanism to partition a long process.

Unfortunately, BEEP and ProTracer require complex binary program analysis in order to instrument a target application for execution partitioning at runtime. To detect the inter-unit dependencies, they need to identify memory dependencies across units by analyzing training runs, and instrument the target program to monitor the relevant memory accesses in production runs. Note that identifying all relevant memory accesses that induce dependencies across execution units in complex binary programs via training is challenging. Missing memory accesses in training leads to false-negatives in attack investigation. They also generate a large number of additional syscalls to denote unit boundaries and memory accesses, increasing the storage pressure.

In addition, while BEEP can prune out some false dependencies as shown in Fig. 3.2-(b) (e.g., between *data1* and `Vim`) by leveraging fine-grained execution units, there are still false dependencies such as those involving *data2* and */tmp/.X11-unit*. This is because, in this example, BEEP considers each file read/write event as a separate unit and detects dependencies between units through memory objects. For example, BEEP considers units that read *data2* (②) and *data3* (③) are causally related to a unit that writes *out.gpg* (⑪) as texts from *data2* and *data3* are copied into a buffer for copy-and-paste in `Vim`. However, the cross-unit dependency between the unit with *data2* (②) and another unit with *out.gpg* (⑪) is bogus because the contents copied from *data2* are not pasted to *out.gpg*. The bogus dependency is introduced because BEEP simply detects `memory read` and `memory write` events with a same memory address without checking if there is true information flow between the two. In short, while BEEP can narrow down the scope of investigation, there are still unnecessary files and events in the graph.

Taint Analysis. Taint analysis techniques [8, 17, 76] track information flow between a set of system components (e.g., file, memory, and network), called taint sources, to another set of system components, called taint sinks. Given a set of input related system components to track, taint analysis keeps track of how data from the specified input components are consumed and propagated by individual instructions that operate on the data, in order to iden-

tify how they impact other system components. However, most taint tracking approaches including the state-of-the-art tools such as TaintGrind [17] and libdft [8] are expensive as they monitor each instruction to track information flow. Furthermore, they are often not able to track implicit flows caused by control dependencies, introducing false-negatives.

To illustrate the merits and limitation of taint analysis techniques, we use a state-of-the-art open source tool, TaintGrind, to analyze the aforementioned incident. Fig. 3.2-(c) shows the result from TaintGrind. In this example, TaintGrind fails to identify the dependency between the *data3* and */tmp/tmpfile*. Note that the most important part of the attack (i.e., the leaked confidential data) is not revealed in the attack investigation due to the missing dependency.

```
1  int tripledes_ecb_crypt(..., const byte* from, ...) {
2    ...
3    work = from ^ *subkey++;
4    to ^= sbox8[  work      & 0x3f ];
5    to ^= sbox6[ (work>>8)  & 0x3f ];
6    to ^= sbox4[ (work>>16) & 0x3f ];
7    to ^= sbox2[ (work>>24) & 0x3f ];
8    ...
9  }
```

Figure 3.3.: Information flow through a table look-up in GPG

We investigate the case in depth, and find that `GPG` decrypts values through a table lookup operation. Unfortunately, TaintGrind is not able to handle information flow through the table lookup, resulting in missing dependencies. Fig. 3.3 shows a code snippet extracted from `GPG`. Specifically, the function argument `from` contains an piece of encrypted text. At line 3, the encrypted text is used to calculate the value of `work`, and TaintGrind successfully propagates taint information to the variable. However, at lines 4-7, `work` is used to look-up a table `sbox2-8`, and TaintGrind loses track of taint information at this point because it does not handle information flow via array indexing. Note that most taint analysis techniques do not track information flow through array indexing to avoid the over-tainting problem. Specifically, the over-tainting problem often leads to an excessive number of taint tags, resulting in false-positives. Hence, most taint analysis tools decide not to track such

information flow. In addition to table look-up, explicit data flows through computations (e.g., bitwise and arithmetic) and implicit data flows caused by control dependency are often disregarded to avoid the over-tainting problem. Moreover, the significant overhead of taint analysis prohibits its application in practical forensic analysis that requires always-on monitoring to capture attacks in-the-wild.

Causality Inference. Recently, Kwon et al. propose a light-weight causality inference technique LDX [1] using a dynamic analysis called *dual execution*. For a given original execution, LDX derives a slave execution in which it mutates values of input source(s). It then compares the corresponding outputs from the original execution and the slave execution to determine whether the outputs are causally dependent on the source(s). Specifically, if the two executions have different values for an output, LDX considers that the output is causally dependent on the mutated input source(s). To address execution path divergence caused by input perturbation, LDX leverages its novel on-the-fly execution alignment scheme. Unlike dynamic taint analysis techniques (e.g., TaintGrind [17] and libdft [8]), LDX can detect explicit and implicit information flow and has much lower runtime overhead (about 6%).

Fig. 3.2-(d) shows the graph generated by LDX. Note that it contains only the objects and events related to the attack, without any false dependences. While LDX produces concise and accurate graphs, it requires the dual-execution framework available on the end-user system which doubles the consumption of computational resources (e.g., CPU and memory).

Table 3.1.: Comparison of Causality Analysis Approaches

	Syscall Analysis [46, 66, 67]	Fine-grained Analysis			Taint Analysis [8, 17, 76]	Causality Inference: LDX [1]	MCI
		BEEP [4]/ProTracer [72]	MPI [74]	WinLog [73]			
Space overhead	Low	Mid	Low	Low	High	Low	**Low**
Runtime overhead	Low	Low	Low	Low	High	Low	**Low**
Resource overhead	Low	Low	Low	Low	High	Mid	**Low**
False-positive	High	Mid	Low	Mid	Low	Low	**Low**
False-negative	Low	Low	Low	Low	Low-Mid	Low	**Low**
Granularity	Coarse	Mid	Fine	Mid	Fine	Fine	**Fine**
End-user requirements	None	Training/instrumentation	Code annotation	None	Tainting framework	Dual-execution framework	**None**

3.2.3 Goals and Our Approach

Table 3.1 presents merits and limitations of existing causality analysis approaches. In summary, syscall analysis techniques suffer from high false-positive rates due to dependence explosion. While BEEP and ProTracer mitigate the dependence explosion problem, they require complex static, dynamic binary analysis and instrumentation and incur non-trivial space overhead. MPI is efficient and effective, but requires access to source code and domain knowledge for annotation. Taint analysis techniques generally incur significant runtime and space overhead and suffer from the over-/under-tainting problems. LDX requires the dual-execution framework in production run that doubles computational resource consumption.

Our Goal. The goal of this chapter is to provide a causality analysis technique with the same accuracy as LDX, but *does not require any changes of end-user systems*, such as instrumenting user applications, modifying the kernel or installing special runtime. Specifically, the end-user only needs to turn on the default audit logging tool that comes with their system, such as Linux Audit, Event Tracing for Windows, and DTrace to collect syscall logs. Upon a security incident, MCI can generate precise causal graphs from the raw log to explain attack causality and assess system damages. We believe such a design would substantially improve applicability.

Our Approach. As shown in Fig. 3.1, the key idea of MCI is to use causal models to parse raw logs to derive precise causality information. Specifically, in the offline phase, we use LDX [1] as the causality inference engine to construct models for the applications that will be deployed on an end-user system. A causal model is essentially a sequence of inter-dependent syscalls and their causal relations. Such causalities/dependencies can be induced by system objects, called *explicit dependencies*, as they can be determined by analyzing syscalls alone, or induced by memory operations and control dependences, called *implicit dependences*, which are not visible by analyzing syscall events. Note that LDX can detect both explicit and implicit dependencies.

During deployment, given a syscall log collected from the incident, MCI can precisely infer causality between events in the log by parsing the log using the pre-generated models.

3.2.4 MCI on Motivating Example

We demonstrate the effectiveness of MCI on investigating the incident. Assume the causal models of applications have been derived offline. Note that generating models does not require any particular expert knowledge on target programs, but rather the typical user level workloads. Model generation is a one-time effort such that models generated for a program can be used for all installations of the program.

Fig. 3.4-(a), (b), (c), (d), and (e) show the graphical representations of some models from `Vim`. A node is denoted by a letter which represents a syscall, with a superscript (*) representing a sequence of syscalls. A subscript represents the (symbolic) system object (e.g., file or socket) operated by the syscall. For example, model (a) is for the behavior of opening and decrypting a gpg file. Specifically, as shown in the legend in Fig. 3.4, the first node of (a) r_α indicates a `read` syscall on α which is *stdin*. Note that each model has its own legend for the subscript. The first node is a syscall that causes the entire behavior. Intuitively, the model represents reading from a command line that loads a gpg file. The second node, s_β, represents a `stat` syscall on a file β (output file). The `GnuPG` plug-in uses a temporary file to store decrypted contents and then informs `Vim` to open. Subscript β symbolizes the temporary file which contains decrypted contents. The second node essentially checks whether the file exists. After that it loads a key file to prepare decryption which is represented as a third node (r^*_γ). Then, it checks (`stat`) the output file again (s^*_β). Finally, the fifth node (r^*_δ) represents reading a gpg file which is an encrypted file. The sixth node (w_β) indicates that the decrypted contents are written onto the output file (β). Then, the `GnuPG` plug-in sends a notification to `Vim` via a pipe which is shown in the last node (w_ε). Note that symbols in subscript (e.g. α, β) can be instantiated to any concrete file handler during parsing. The same subscript β in s_β and the later nodes s^*_β and w_β dictate that these syscalls must operate on the same file. The third and fifth nodes

are denoted by a superscript *, representing a sequence of `read` system calls (`read`*) on different files γ and δ.

The directed edges between nodes represent the causality/dependency between syscalls, with the solid and dotted edges representing the explicit and implicit dependencies, respectively. For example, in (a), there are explicit dependences from s_β to w_β and implicit dependencies from r^*_γ and r^*_δ. The implicit dependencies are caused by memory operations that copy values from a crypto key file (γ) to encrypted contents δ that are detected and modeled by MCI.

Fig. 3.4-(f) illustrates a syscall log collected during the incident by the default Linux Audit tool [69]. Given the syscall log and the models, MCI automatically parses the log and hence derives the corresponding dependencies. Each box in (f) denotes a model instance with the letter annotated on the box representing the model id. Note that we use different background colors for boxes to represent nodes belong to different models. We omit the dependences in the model instances for readability. For readability, we use superscripts to denote event timestamps.

The model instances essentially tell us that the user first opened a gpg file (i.e., *out.gpg*) by model (a), opened and copied a file (i.e., *data2*) without pasting by model (b), and opened, copied, and pasted another file (i.e., *data3*) by model (c). Observe that there are events that belong to multiple models, which allow us to determine causality across models and hence *compose* the whole attack path. For instance, event s_5^{11} belongs to both models (c) and (d) (i.e., the node in the two boxes in blue and green), suggesting that the contents from *data3* are copied to the previous gpg file. The subscript 5 corresponds to file `viminfo` that is used to indicates the state of editing. Note that model (c) does not have explicit dependencies with other models. Hence, without model (d), causality between model (c) and other models is difficult to reveal. After a few editing operations by model (d), the user finally saved the contents to a new gpg file by model (e). The event s_5^{11} belonging to models (c) and (d) indicates that the new gpg file contains information from *data3* (confidential data). Note that the matched instance of model (b) does not have any overlapping nodes with other model instances nor explicit dependencies, and hence no causal relations with

others. This indicates that *data2* is not involved in the incident. The final causal graph is shown Fig. 3.2-(d), which is accurate and concise, without any missing or bogus dependencies.

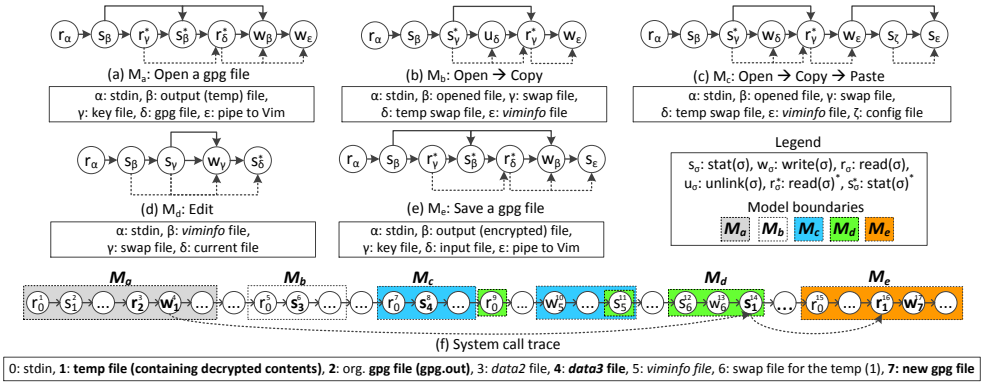


Figure 3.4.: MCI on the motivating example

3.3 Problem Definition

In this section, we introduce a number of formal definitions and the problem statement for MCI.

3.3.1 Definitions

Causal Model. Fig. 3.5 shows definitions for a causal model. Specifically, *SysName* represents syscall names such as *open* and *read*. *Repetition* indicates how many times a term or node repeats. It could be a constant number, a variable such as n or m, or $*$ representing any number of repetition. Variables are needed to to denote repetition constraints across syscall events. *ResourceSymbol* represents a symbol for a resource handler that a system call operates on (e.g., file handler). A *Term* is a sequence of *Node*s that could be annotated with the number of repetitions. A node N is a syscall annotated with a set of parameters denoted by *SymbolicResource*. A symbolic resource can be instantiated to different concrete resources

during parsing. Two nodes with the same symbolic resource indicates that they have explicit dependency. An *Edge* denotes dependency/causality between two nodes N_{from} and N_{to}. Finally, a causal model is defined as a 3-tuple $< \overline{T},\ \mathbb{P}(E)_{implicit},\ \mathbb{P}(E)_{explicit} >$ where $\overline{T}$ is a sequence of terms, $\mathbb{P}(E)_{implicit}$ is the set of implicit dependency edges and $\mathbb{P}(E)_{explicit}$ is the set of explicit dependency edges. The definitions of two kinds of edges can be found in Sec. 3.2.

SyscallName	*SysName* ::=	`open` | `read` | `write` | ...
Repetition	R ::=	1 | 2 | 3 | ... | n | m | $*$
SymbolicResource	S ::=	$\{\alpha,\ \beta,\ \gamma, ...\}$
Term	T ::=	$N \mid NT \mid (T)^R$
Node	N ::=	$SysName_{\mathbb{P}(S)}$
Edge	E ::=	$< N_{from}, N_{to} >$
Model	M ::=	$< \overline{T},\ \mathbb{P}(E)_{implicit},\ \mathbb{P}(E)_{explicit} >$

Figure 3.5.: Definition of causal model

For example, the model in Fig. 3.4 (a) can be represented as follows. First, $\overline{T}$ can be represented by a sequence: $read_\alpha, stat_\beta, read^*_\gamma, stat^*_\beta, read^*_\delta, write_\beta, write_\varepsilon$. Implicit dependencies (dotted edges below nodes) are denoted as follows: $\{\langle read^*_\gamma, read^*_\delta\rangle,\ \langle read^*_\delta, write_\beta\rangle,\ \langle read^*_\delta, write_\varepsilon\rangle\}$. Explicit dependencies (solid edges above nodes) are the following: $\{\ \langle stat_\beta, stat^*_\beta\rangle,\ \langle stat_\beta, write_\beta\rangle\ \}$. Observe the nodes in an explicit edge have the same resource symbol, indicating that they operate on the same resource. In this chapter, we will use the more concise graphical representations when possible.

Syscall Trace. As shown in Fig. 3.6, a system call trace T is a sequence of trace entries $\overline{TE}$ where a trace entry is a system call name annotated with a set of *ConcreteResource* that represents concrete resource handlers, and a number $\mathbb{N}$ that represent an index of *TE* in T. Note that it does not contain any dependency information. The first 6 entries in Fig. 3.4 (f) are represented as $\overline{TE} = (read^1_0, stat^2_1, ..., read^3_2, write^4_1, ...)$. Note that the subscripts represent concrete resource handlers and the superscripts represents indexes.

ConcreteResource	$C ::=$	$\mathbb{N}$
TraceEntry	$TE ::=$	$SysName^{\mathbb{N}}_{\mathbb{P}(C)}$
SyscallTrace	$T ::=$	$\overline{TE}$

Figure 3.6.: Definition of syscall trace

3.3.2 Problem Statement

We aim to infer fine-grained causality from a syscall trace by parsing it with models. This procedure can be formally defined as a function of T and $\mathbb{P}(M)$:

$$T \times \mathbb{P}(M) \mapsto (TE \mapsto \mathbb{P}(N \times M))$$

Specifically, given a syscall trace T and a set of models $\mathbb{P}(M)$, the function generates a mapping, in which a trace entry is mapped to a set of nodes N in model M. It is a set because a trace entry can be present in multiple models as shown in the motivation example in Sec. 3.2. With the mapping, the dependencies between trace entries can be derived from the dependencies between the matched nodes in the models. For example, parsing the trace in Fig. 3.4 (f) using the models in (a)-(d) generates the following mapping. The first 4 events are mapped to model (a): $(read_0^1 \mapsto < read_\alpha, M_a >), (stat_1^2 \mapsto < stat_\beta, M_a >), (read_2^3 \mapsto < read_\delta^*, M_a >), (write_1^4 \mapsto < write_\beta, M_a >)$. Moreover, $stat_5^{11}$ belongs to two models, resulting in two mappings: $(stat_5^{11} \mapsto < stat_\varepsilon, M_c >), (stat_5^{11} \mapsto < stat_\beta, M_d >)$. It entails the following concrete dependency edges $< read_2^3, write_1^4 >$ (from model edge $< read_\delta^*, write_\beta >$ in (a)) and $< stat_5^{12}, stat_1^{14} >$ (from model edge $< stat_\beta, stat_\delta^* >$ in (d)). The first edge indicates implicit dependency between the original gpg file (out.gpg) and a temp file containing its decrypted contents, and the second edge implies that the copy and paste action is related to the temp file containing the decrypted contents of the original gpg file (*out.gpg*). Such dependency edges lead to a causal graph as that in Fig. 3.2-(d).

The mapping may not be total, depending on the comprehensiveness of the models. An important feature of MCI is *model composibility*, meaning that a complex behavior can be composed by multiple models sharing some common nodes. For instance, a complex user behavior in `Vim` such as "open file, edit, copy, edit, paste, save, reopen" can be decomposed

to multiple primitive models. As such, the number of models needed for regular workload is limited as shown in Sec. 4.6.

The key challenge of MCI *lies in parsing the trace that does not contain any dependencies with models that contain dependency information*, which entails solving two prominent technical problems discussed next.

3.3.3 Technical Challenges: Complexity and Ambiguity

Language Complexity

According to our definition, a trace is a string in the trace language that does not contain dependency information, our problem is essentially to parse the string to various model instances. In the following, we use the classic language theory to understand the complexity of our problem. Note that although it seems that we could consider models as graphs and leverage the sub-graph isomorphism theory to understand our problem, there are places that can hardly be formulated in the graph theory. For instance, our trace is not a graph because it does not have implicit dependency information. Furthermore, our model may have constraints among the numbers of event repetitions (e.g., the number of `close` matches with the number of `open` while the number of repetitions may vary). Such constraints can hardly be represented in graphs.

The classical Chomsky hierarchy [86, 87] defines four classes of languages characterized by the expressive power of their defining grammars: *regular*, *context-free*, *context-sensitive*, and *recursively enumerable*. More expressive grammar can describe more complex language but requires higher cost in parsing. We study some of representative causal model types observed in real-world programs. For each type, we show a sample grammar and discuss the complexity of the grammar as well as scalability of the corresponding parser.

Regular Model. Fig. 3.7 shows a model from `ping` [88], representing a behavior "*resolving a network address, sending a packet, and receiving a response.*"

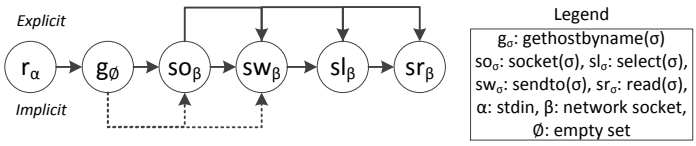


Figure 3.7.: Regular model from ping [88]

Observe that the explicit dependencies (solid edges) are caused by the socket (β). The implicit dependencies (dotted edges) are introduced because `gethostbyname()` decides whether to execute `socket()` and `sendto()` meaning that they have control dependences. In particular, if `gethostbyname()` returns an error, the program immediately terminates. Also, `sendto()` is dependent on the return value of `gethostname()` (e.g., IP address) as the `ping` program composes and sends Internet Control Message Protocol (ICMP) packets that contain the returned IP address. Such dependencies are not visible at the syscall level. Note that in any model, the first node, which is always an input syscall, has dependencies leading to all other nodes. Recall that a model is acquired from LDX that mutates an input syscall and observes changes at output syscalls (e.g., the first node in Fig. 3.7 is a syscall that reads an option from the command line that leads to all the other syscalls in the model).

The model in Fig. 3.7 can be simplified by a regular grammar (e.g., regular expression) which is the simplest one in Chomsky hierarchy. A regular language parser has very good scalability. From our experience, most models (53 out of 56 models in our evaluation) fall into this type.

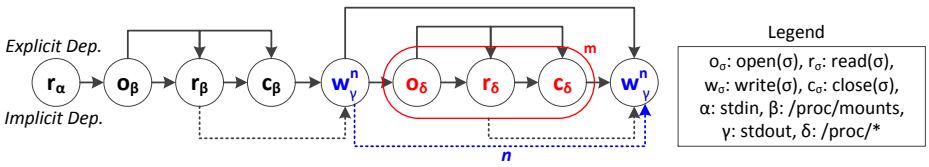


Figure 3.8.: Context-free model from procps [89]

Context-free Model. There are cases that the models need to be context-free. Fig. 3.8 shows such a model extracted from `procps` [89]. The model represents "*retrieving file*

system information." It first reads a file that contains information about the list of file systems. It then uses an outer loop to emit the information for individual file systems. For each file system, an inner loop is used to collect information about the file system from multiple places (e.g., different disks).

As shown in Fig. 3.8, three symbols from the 2^{nd} to the 4^{th} (o_β, r_β, c_β) have explicit dependencies due to the file containing the list of file systems (β). The 5^{th} symbol w_γ^n is to emit the header information for each file system, causing the implicit dependency between the 3^{rd} symbol r_β and the 5^{th}. The superscript n denotes that there are n file systems. The 6^{th}, 7^{th}, and 8^{th} symbols (o_δ, r_δ and c_δ) form a term, corresponding to the inner loop that reads m places to collect information for the n file systems. Note that m may not equal to n as multiple files may be accessed in order to collect information for a file system. After that, the 9^{th} symbol w_γ^n emits the collected information for the n file systems. Note that the number of writes in the 5^{th} and the 9^{th} symbols need to be identical (n times). The constraints on the numbers render the model cannot be transformed to an automaton that handles a regular language. It is essentially context-free. The parser for a context-free language requires some push-down mechanism, incurring higher complexity. We have encountered 2 context-free models in our evaluation.

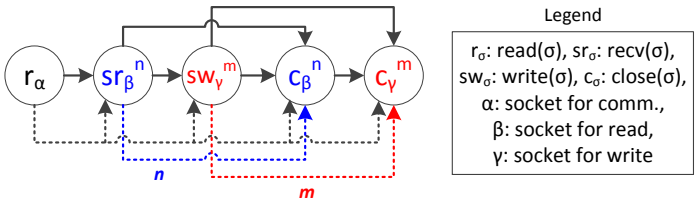


Figure 3.9.: Context-sensitive model from raft [90]

Context-sensitive Model. In some rare cases, even context-free models are not sufficiently expressive. Fig. 3.9 shows a model from [91] which is a distributed voting application that implements the Raft consensus protocol [90]. The program can exchange network messages between different number of users to get a consensus. The model describes a

voting procedure. Specifically, it receives network messages from n users (n iterations of `read()`), and sends network messages to m users (m iterations of `write()`). Later, it closes the sockets for n users and then m users. The crossing-constraints between m and n (r_2^n, c^n) and (w^m, c^m) require a context-sensitive language. However, a parser for a context-sensitive language is prohibitively expensive in general (PSPACE complexity [92]). We have not encountered any models more complex than context-sensitive languages. The various language complexities pose a prominent challenge: since syscall events belonging to multiple models interleave and are often distant from each other, we cannot know which model an event belongs to until reaching the end of the model. As such, we do not know which complexity class shall be used to parse individual events. As we will show later, we develop a uniform parsing algorithm for multiple complexity classes that leverages the special characteristics of causal models.

Ambiguity

The strings (of syscalls) parsed by multiple models may share common parts (e.g., common prefixes). In the worst case, multiple models may accept the same string, although we have not encountered such cases for models within the same application. As a result during trace parsing, given a syscall, there may be multiple models that it can be attributed to and MCI does not know which model(s) are the right ones. We call it the *ambiguity problem*.

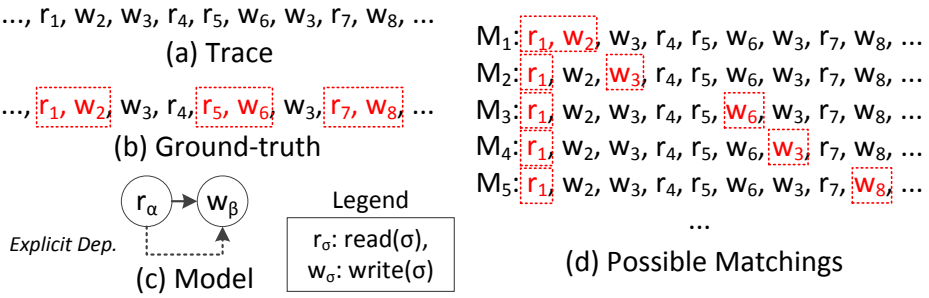


Figure 3.10.: Ambiguity problem

For instance, consider a trace, the ground-truth causality of the trace, and a model shown in Fig. 3.10-(a), (b), and (c), respectively. Observe that the model has a socket read followed by a file write. The two have implicit dependency but not explicit dependency visible at the syscall level. The three boxes in Fig. 3.10-(b) denote the three real model instances.

When the model is used to parse the trace, due to the lack of dependencies between the two syscalls in the model, there are many possible matchings as shown in Fig. 3.10-(d). Note that except M_1, the other matchings are incorrect even though they all appear possible at the syscall level. In practice, such incorrect matchings introduce false causalities which hinder attack investigation. Moreover, ambiguity may cause excessive performance overhead because MCI has to maintain numerous model instances at runtime. The root cause of the problem is that the trace does not have sufficient information. Hence, we develop a method that leverages explicit dependences to mitigate the problem. Details can be found in Sec. 3.4.2.

3.4 System Design

MCI consists of two phases: model construction and model parsing. The former is offline and the latter is meant to be deployed for production run.

3.4.1 Model Construction

Given an application, the forensic analyst provides a set of regular workloads. The application is executed on the LDX system with the workloads. The dependences detected by LDX, including explicit and implicit dependences, are annotated on the syscall events in the audit logs. The annotated logs are analyzed to extract inter-dependent subsequences, which are further symbolized (i.e., replacing concrete resource handlers with symbolic ones). The sequences of symbolic syscalls with dependences constitute our causal models.

In the following, we use a program snippet in Fig. 3.11 to illustrate how MCI constructs causal models. It first reads a network message (line 1) and encrypts the received message

(line 2). Later, it stores the encrypted message to a local file (line 3) and sends a notification to a GUI component (line 5).

```
1  while( (len = read(socket, buf, 1024)) != -1 ) {
2     ebuf = encrypt(buf);
3     write( file, ebuf, 4096 );
4  }
5  sendmsg( wnd, “Update: ” + ebuf ... );
```

Figure 3.11.: Example program

Dependencies Identification by LDX

The program is executed with a typical workload on LDX [1] to collect a system call log T. To identify dependencies, LDX mutates the value of input syscall `read()` in the slave execution. By contrasting the values of the following syscalls (e.g., the `write()` and `sendmsg()`) in the two executions, LDX identifies all the dependencies between syscalls.

```
1  // SR: means that the system call is a source system call.
2  // CD: means that the system call is causally dependent on the input
3  [SR] 100 = read( 0x11/*file handle*/, "AAAA..."/*Mutation: BBBB...*/, 1024 );
4  [CD] 4096 = write( 0x12/*file handle*/, "Contents to be written", 4096 );
5  [CD] -1 = read( 0x11/*file handle*/, "Returned buffer", 1024 );
6  [CD] 20 = sendmsg( ...., "Content to be written", 4096 );
```

Figure 3.12.: Causally dependent system calls from LDX

Fig. 3.12 shows the output generated by LDX. It includes two `read()`s (lines 3 and 5), one `write()` (line 4) and one `sendmsg()` (line 6) which are causally dependent on the source (i.e., `read()` at line 2). More specifically, the `write()` at line 4 and `sendmsg()` at line 6 are (implicitly) dependent on the source by variables `buf` and `ebuf`, and the `read()`s at lines 2 and 4 are explicitly dependent on the source due to the socket handler `0x11`.

The generated sequence of syscalls includes all the syscalls causally dependent on the source (line 3). We hence leverage them as a sample of the model. Note that LDX also returns dependences between syscalls inside the sequence such as the dependence between lines 3 and 4.

Symbolization

The collected sequence of syscalls cannot be directly used as a model due to the concrete arguments. For instance, in Fig. 3.12, syscalls have concrete values (e.g., handlers `0x11` and `0x12`) which may differ across executions. Hence, we symbolize concretes values in syscalls by replacing with symbols (e.g., α and β). For instance, if two syscalls share the same argument, they are assigned the same symbol.

If the application supports repeated workload, there must be repetitions in the syscalls that need to be modeled (such as n and m in Fig. 3.5). To do so, MCI duplicates the workload a few times and feeds the new workloads to LDX again. Subsequences that have a constant number of repetitions across workloads are annotated with the constant. Those that have varying numbers of repetitions across workloads are annotated with '*'. If there are correlations between the repetition numbers of multiple subsequences (inside the same model), variables n/m are used to model the number of repetition, such as the previous example Fig. 3.8 in Sec. 3.3.3.

```
1  SUCCESS = read( fd1 /* file handle*/, *, * );
2  SUCCESS = write( fd2 /* file handle*/, *, * );
3  FAILURE = read( fd1 /* file handle*/, *, * );
4  SUCCESS = sendmsg( *, *, * );
```

Figure 3.13.: Symbolized system calls

Fig. 3.13 shows a symbolized log. For example, 0x11 in `read()` in Fig. 3.12 is replaced by a new symbol fd_1 and 0x12 in `write()` in Fig. 3.12 is generalized to another symbol fd_2. 0x11 in the second `read()` is replaced by the previously assigned symbol fd_1 as it

already appeared before. Moreover, as shown in Fig. 3.13, all concrete return values are symbolized as either SUCCESS or FAILURE. They are part of the models in our system although our formal definitions did not describe them for brevity. The constructed model is shown in Fig. 3.14. The formal model construction algorithm is elided due to the space limitations.

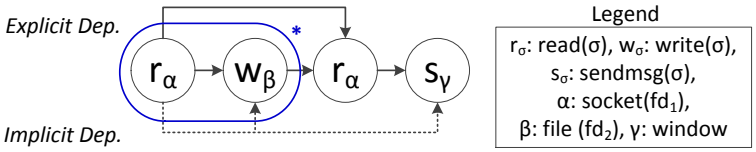


Figure 3.14.: Constructed model from the example

3.4.2 Trace Parsing with Models

In this section, we describe how MCI parses an audit log with models. As we described in Sec. 3.3.3, if we simply consider an audit log as a string of the trace language, we need to consider three language classes in the Chomsky hierarchy, namely, regular, context-free, and context-sensitive languages. Recursively enumerable languages are never encountered in our experience. A more expressive language requires more expensive parser. For instance, context-free language can describe almost all causal models we have encountered but context free parsers have a time complexity of n^3 where n is the length of a string (the number of events in audit log in our case), thus they are too expensive to handle real-world logs that can grow in the pace of gigabytes per day [93] (corresponding to millions of events). Context-sensitive parsers have even higher computational complexity. Furthermore, our parser needs to be able to substantially mitigate the ambiguity problem in which MCI does not know which models an event should be attributed to.

Segmented Parsing. Our proposal is *not to consider a trace as a simple string, but rather a sequence of symbols with explicit inter-dependences.* Note that explicit dependences can be

directly derived from the trace. The basic idea is hence to leverage explicit dependences to partition the sequence of terms/nodes in a model into *segments*, delimited by terms/nodes that are involved in some explicit dependences. Therefore, all the terms/nodes inside each segment are a string in some regular language. The essence is to leverage explicit dependences to reduce language complexity. During parsing, we first recognize (from the trace) the explicit dependences that match those of the model. These dependences partition the trace into sub-traces. Then automata are used to recognize model segment instances from the sub-traces. Since string parsing is only carried out within small sub-traces instead of the lengthy whole trace, ambiguity can be substantially suppressed. We call the technique *segmented parsing*.

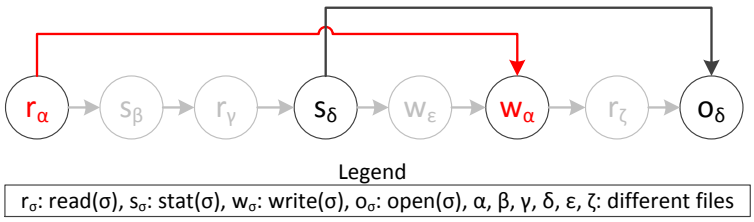


Figure 3.15.: Example for segmented parsing

Next, we use an example to illustrate the basic idea and then explain the algorithm. Fig. 3.15 shows a sample model. Observe that there are explicit dependences between the 1^{st} and the 6^{th} nodes (r_α and w_α), and between the 4^{th} and the 8^{th} nodes (s_δ and o_δ). The sequence of terms/nodes involved in explicit dependences form the *model skeleton*. In our example, it is r_α-s_δ-w_α-o_δ. The skeleton partitions the model into *sub-models*. A sub-model is a sub-sequence of nodes/terms of the model that are delimited by explicit dependences but themselves do not have any explicit dependences. In Fig. 3.15, three sub-models are obtained as follows: s_β-r_γ delimited by r_α and s_δ, w_ε delimited by s_δ and w_α, and r_ζ delimited by w_α and o_δ.

During parsing, we first find instances of the model skeleton. For each skeleton instance, we try to identify instances of sub-models within the trace ranges determined by

the skeleton instance. Any mismatch in any sub-model indicates this is not a correct model instance and the corresponding data structures are discarded. In our example, we first locate the possible positions of r_α, s_δ, w_α, o_δ in the trace, and then look for the instances of s_β-r_γ in between the positions of r_α and s_δ, and so on. Partitioning a model to a skeleton and a set of sub-models is straightforward. Details are hence elided. Given a trace, to facilitate segmented parsing, we extract a number of *trace indexes*, each containing all the nodes related to the same system object (e.g., a file) and the position of the nodes in the raw trace. These indexes allow our parser to quickly locate skeleton instances in the trace. Fig. 3.16 shows an example of index extraction from a trace. Observe that all the nodes in an index have explicit dependences.

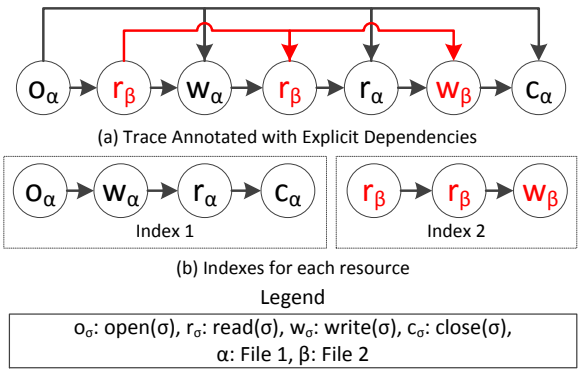


Figure 3.16.: Trace preprocessing

Algorithms. The parsing procedure consists of three major steps. The first one is to preprocess trace to extract indexes, which has been intuitively explained before. The second step is to locate skeleton instances in the trace and the third is to parse sub-models. In the following, we explain the algorithmic details of steps two and three.

The algorithm of locating skeleton instances is shown in Alg. 4. It takes the trace T, the indexes I that can be accessed by the concrete resource id (e.g., file handler), and a model skeleton S, and identifies all the possible instances of the skeleton. The result is stored in P. Each instance is a pair $\langle map, seq \rangle$ with map projecting each symbolic resource (e.g., α

Algorithm 4 Locating Skeletons

Input: trace T, indexes I, model skeleton S
Output: a set of skeleton instances P, each consisting of a mapping that maps a symbolic resource to a concrete one, and a sequence of positions

```
1: procedure LocateSkeleton(T, I, S)
2:     for all node N_α ∈ S do
3:         if P ≡ {} then
4:             P ← {⟨{α → h}, i⟩ | for all T[i] = N_h}
5:         else
6:             for all ⟨map, seq⟩ ∈ P do
7:                 Let the last position in seq be i
8:                 if map[α] ≠ nil then
9:                     pos ← findbeyond(N,i,I[map[α]])
10:                    if pos ≠ −1 then
11:                        seq ← seq · pos
12:                    else
13:                        P.remove(⟨map, seq⟩)
14:                    end if
15:                else // scan all indexes to find N_h syscalls that are beyond i
16:                    ... // and instantiate α to h.
17:                end if
18:            end for
19:        end if
20:    end for
21:    return P
22: end procedure
```

Algorithm 5 Model Parsing

Input: trace T, skeleton instances P, sub-models S
Output: the concrete syscall entries that correspond to the sub-models in the temporal order

```
1: procedure ParseSubModels(T, P, S)
2:     for all ⟨map, seq⟩ ∈ P do
3:         for i from 0 to |S| − 1 do
4:             instance[i] ← parse(T[seq[i], seq[i + 1]], S[i])
5:         end for
6:         if all instance[0 − (|S| − 1)] are not nil then
7:             if none of the concrete syscalls in instance[0(−|S| − 1)] share the same resource id then
8:                 output instance[0 − (|S| − 1)]
9:             end if
10:        end if
11:    end for
12: end procedure
```

and β) in the skeleton to some concrete handler and *seq* storing the trace positions of the individual nodes in the skeleton. To simplify our discussion, we assume the skeleton does not have repetitive nodes or terms. The algorithm can be easily extended to handle such cases.

The main procedure iterates over each node N_α in the skeleton (line 2) with N the syscall and α the symbolic resource. For the first node (indicated by an empty result set P), the algorithm considers each syscall of the same type N, in the form of N_h at location i in the trace, may start an instance of the skeleton, and hence instantiates α to the concrete handler h and records its position i (lines 3 and 4). If N_α is not the first node, the algorithm iterates over all the skeleton candidates in P in the inner loop (lines 6-18) to check if it can find a matching of the node for these candidates. If not, the skeleton candidate is invalid and hence discarded. Specifically, for each skeleton candidate denoted as $\langle map, seq \rangle$, line 7 identifies the trace position of latest node i. This is needed as the algorithm looks for the match of N_α in trace entries beyond position i. The condition at line 8 separates the processing to two cases with the true branch denoting the case that α has been instantiated before, that is, a node of the same symbolic resource was matched before (e.g., w_α in Fig. 3.15), the else branch otherwise (e.g., s_δ in Fig. 3.15). In the first case (lines 9-11), the algorithm looks up the index of the concrete handler associated with α, i.e., $I[map[\alpha]]$, to find a concrete syscall N beyond position i (line 9). If such a syscall is found, we consider the algorithm has found a match and the new position *pos* is appended to *seq* (line 11). Otherwise, the skeleton candidate is not valid and removed (line 13). Here, we have another simplification for ease of explanation. Line 9 may return multiple positions in practice while in the algorithm we assume it only returns one. The extension is straightforward.

In the else branch, the node has a new symbolic resource, the algorithm has to go through all indexes to find all instances of N and instantiate the symbolic resource accordingly. This may lead to the expansion of the candidate set P. Details are elided. To reduce search space, we use time window and other syscall arguments to limit scopes.

Given a set of skeleton instances for a model M, Alg. 5 parses the sub-models of M. In particular, the outer loop (lines 2-11) iterates over all the skeleton candidates identified

in the previous step. If matches can be found for all sub-models regarding a skeleton instance, the matches are emitted. Otherwise, it is not a legitimate instance and discarded. Specifically, the inner loop in lines 3 and 4 iterates over individual sub-models in order. In the i^{th} iteration, it uses automata to parse sub-model $S[i]$ in the trace range identified by the i^{th} segment identified by the skeleton candidate, which is from $seq[i]$ to $seq[i+1]$ (line 4). Automata based parsing is standard and elided. After such parsing, line 6 checks if we have found matches for all sub-models. If so, line 7 further checks that none of the concrete syscall entries that are matched with some node in a sub-model do not share the same resource (and hence have explicit dependences). This is because the model specifies that there are not explicit dependences between the corresponding nodes. Line 8 outputs the parsing results.

Handling Threaded Programs. Threading does not pose additional challenges to MCI in most cases because syscalls from different threads have different process ids so that models can be constructed independently for separate threads. Explicit dependences across threads can be easily captured by analyzing audit logs. Some programs such as `Apache` and `Firefox` use in-memory data structures (e.g., work queues) to communicate across threads, causing implicit dependences. However, it is highly complex to model and parse behaviors across threads due to non-deterministic thread interleavings. We observe that these data structures are usually protected by synchronizations, which are visible at the syscall level, and the synchronizations should follow the nature of the data structures, such as first-in-first-out for queues. Hence, MCI constructs models for individual threads including the dispatching thread and worker threads. The models include the synchronization behaviors. It then leverages the FIFO pattern to match nodes across threads. It works nicely for most of the programs we consider except `transmission`, whose synchronization is not visible at the system level (Sec. 4.6).

3.5 Evaluation

In this section, we evaluate MCI with a set of real-world programs in order to answer the following research questions.

RQ 1. How many models are required to infer causality for these programs in production runs (Sec. 3.5.1), and how much efforts are required to construct models? (Sec. 3.5.1)

RQ 2. How effective is MCI for system wide causality inference including multiple long-running programs and various activities? (Sec. 3.5.2)

RQ 3. How effective is MCI for realistic attack investigation? (Sec. 3.5.3)

RQ 4. Is MCI scalable on large workloads for long-running programs? (Sec. 3.5.3)

Table 3.2.: Details on Model Construction

Program	**Model Description**	**Size**[1]	**D**$_{exp}$[2]	**D**$_{imp}$[3]	**Lang.**[4]
Firefox	Tab Open/Switch/Close	7/9/5	2/2/1	3/4/3	Reg.
	Load a URI	12	2	4	Reg.
	Download (Save)	15	3	5	Reg.
	Click a link	9	2	3	Reg.
Apache	HTTP(S) resp.	17 (21)[5]	3 (4)[5]	8 (11)[5]	Reg.
	CGI resp.	26 (33)[5]	4 (5)[5]	11 (14)[5]	Reg.
Lighttpd	HTTP(S) resp.	8 (11)[5]	2 (3)[5]	4 (6)[5]	Reg.
	CGI resp.	16 (19)[5]	3 (4)[5]	7 (9)[5]	Reg.
nginx	HTTP(S) resp.	14 (17)[5]	3 (4)[5]	6 (9)[5]	Reg.
	CGI resp.	21 (24)[5]	4 (5)[5]	8 (11)[5]	Reg.

Continued on next page

Table 3.2: Details on Model Construction (cont.)

Program	**Model Description**	**Size**[1]	**D**$_{exp}$[2]	**D**$_{imp}$[3]	**Lang.**[4]
CUPS	Add printers	6	1	3	Reg.
	Remove printers	5	1	3	Reg.
	Modify printers	6	1	3	Reg.
	Print a doc.	7	2	4	Reg.
vim	Open	8	1	5	Reg.
	Edit	10	1	4	Reg.
	Save	13	2	4	Reg.
	Save As	15	3	6	Reg.
	Copy and Paste	14	3	6	Reg.
	Copy	11	1	5	Reg.
	Plug-in (gpg)	21	2	6	Reg.
elinks	Browse	11	3	6	Reg.
	Save	6	2	5	Reg.
	Upload	7	2	5	Reg.
alpine	Send emails	10	2	6	Reg.
	Send files	13	3	7	Reg.
	Download emails	9	2	6	Reg.
	Download files	11	2	5	Reg.
	Open a link	8	2	4	Reg.
zip	Compress file(s)	16	8	5	C.F.
	Use encryption	6	4	3	Reg.

Continued on next page

Table 3.2: Details on Model Construction (cont.)

Program	**Model Description**	**Size**[1]	**D**$_{exp}$[2]	**D**$_{imp}$[3]	**Lang.**[4]
transmission	Download	17	4	8	Reg.
	Add a torrent file	6	3	3	Reg.
	Add a magnet	12	3	7	Reg.
proftpd/ lftp/yafc	Login	5/4/6	1/1/2	4/3/4	Reg.
	Create directory	4/4/4	2/2/2	3/3/3	Reg.
	Delete directory	3/4/4	1/2/2	3/3/3	Reg.
	List directory	3/3/3	1/1/1	3/3/3	Reg.
	Upload	7/8/18	2/2/3	5/5/9	Reg.
	Download	6/7/16	2/2/4	5/6/9	Reg.
wget	Download (HTTP(S))	7 (15)[5]	2 (4)[5]	5 (8)[5]	Reg.
ping	Option -f	6	2	5	Reg.
	Option -r	5	2	5	Reg.
procps	Get file system info.	6	3	4	C.F.
raft [91]	Voting	5	2	6	C.S.
	Leader Election	7	2	7	Reg.
Average	-	10.2	2.4	5.4	-

[1]: # of nodes in a model. [2]: # of explicit dependencies (edges) in a model.

[3]: # of implicit dependencies (edges) in a model. [4]: Language Class of a model.

[5]: for HTTPS.

Experiment Setup. We evaluate our approach on 17 real-world programs. Table 3.2 shows the programs and models we constructed. Note that 15 out of the 17 programs (except `zip` and `Vim`) are network related which is a popular channel for cyber-attacks. For each program, we construct models offline. We use typical workloads briefly described in the second column of Table 3.2. Specifically, if there are available test inputs for a program, we use them as the typical workloads. Otherwise, we construct inputs by inspecting program

manuals and identifying options and commands that can trigger different functionalities, such as for `proftpd`, `CUPS`, and `zip`.

3.5.1 Model Construction

Table 3.2 shows the constructed models for each program. Columns 1 and 2 show programs and model description. Column *Size* shows the number of nodes in each model. The numbers in/out parentheses are for the same behaviors with/without `HTTPS`. The next two columns show the number of explicit and implicit dependencies in each model. The last column (Lang.) shows the language class of each model (Regular (Reg.), Context-free (C.F.), or Context-Sensitive (C.S.)).

We have the following observations from the results. First, the size of model is relatively small (on average 10.2 nodes) and there are on average 2.4 explicit dependencies (more than 4 nodes) for each model. The strong presence of explicit dependencies allows MCI to perform segmented parsing effectively. Second, we observe three language complexity classes and most models fall into the regular class. It supports our design choice of integrating regular parsers (i.e., automata) with explicit dependency tracking.

of Models Required

The constructed models listed in Table 3.2 are sufficient to infer causality for logs from realistic scenarios described in Sec. 3.5.3 including the motivation example in Sec. 3.2. The number of models for each program ranges from 3 to 12 which is fairly small and not difficult to obtain in practice. We observe that the primary reason why MCI is effective with a small number of models is model composibility, namely, primitive models can be used to compose complex behaviors. For instance, models for “Edit” and “Save” can compose a new model “Edit and Save”.

Efforts on Model Construction

To construct models, a program is executed repeatedly on LDX. The number of runs required to construct a model depends on the number of events in the model. Specifically, we first run a program with a workload on LDX to identify all the events causally dependent on the workload. Note that the detected events constitute the bulk of the model. Assume there are n such events (nodes). For each node in the model, MCI mutates the value of the corresponding syscall to determine dependencies on the node inside the model. To figure out the repetition factors of the node (Sec. 4.4), MCI runs k times for the node, each execution repeats the workload for different times. In total, we run a program $(k*n)+1$ times to construct a model. In our experiments, $k = 10$. On average, the machine time to construct a model, including LDX execution time and model extraction time, takes 4 minutes (253 seconds).

Table 3.3.: Results for System-wide Causality Inference

Program	**# of events**	**# of causality**	**# of matched models**	**FP**	**FN**
Firefox	2,313 M	11 M	549 K	8.3%	3.2%
Apache	296 M	6.6 M	435 K	0%	0%
Lighttpd	125 M	3.3 M	275 K	0%	0%
nginx	187 M	3.8 M	246 K	0%	0%
proftpd	49 M	2.1 M	179 K	0%	0%
CUPS	25 M	918 K	88 K	0%	0.8%
vim	43 M	4 M	219 K	0%	0%
elinks	38 M	3.6 M	145 K	0%	0%
alpine	116 M	4.7 M	231 K	0%	0.3%
zip	5 M	634 K	36 K	0%	0%
transmission	250 M	6.9 M	479 K	3.8%	5.2%
lftp	11 M	438 K	54 K	0%	0%
yafc	9 M	616 K	43 K	0%	0%
wget	627 K	71 K	5.4 K	0%	0%
ping	2.4 k	1.3 K	241	0%	0%
procps	4 M	1 M	176 K	0%	0%

3.5.2 System-wide Causality Inference

In this experiment, we apply MCI to infer causality on a system wide syscall trace collected for the system execution of a week, to demonstrate the effectiveness of causality inference for realistic programs with production runs. The trace includes syscall logs from multiple programs including those in Table 3.2. Specifically, we enable Linux Audit and use the programs in Table 3.2 with typical workloads for a week. Given the collected trace, we identify all the inputs that appear in the trace (e.g., file reads, command line arguments, user interactions). Then, we build a forward causal graph from each input, i.e., identifying all other syscalls depending on the input, using MCI and compare it with the ground truth by LDX. During the experiment, we record all inputs used for the programs. Then, we re-execute the program with the recorded inputs to reproduce the same execution. To do so, we develop a lightweight record and replay system similar to ODR [94]. LDX is run on top of the replay system to derive the ground truth. Note that due to the limitation of the replay system, the replayed execution may differ from the original execution. Such differences are counted as false-positives/negatives for conservativeness.

The collected log consists of syscalls from multiple programs and the size of the log is around 732 GB (without compression) containing 3707 million events. We first separate the log into smaller logs per process.

Table 3.3 shows results of the experiment. The second column shows # of events (syscalls) in the log for each program. The third and forth columns represent # of dependencies detected and # of models matched by MCI. For the # of dependencies, we count all those inferred by MCI via matched models and those explicit dependencies across matched models. The last column shows false-positive and false-negative rates.

For most programs, MCI precisely identifies causality with not measurable false positives and negatives. There are a few exceptions: `Firefox`, `CUPS`, `alpine`, and `transmissions`. We manually inspect a subset of these false-positives/negatives and have the following observations. Our Firefox models are intended to describe browser behaviors such as following a hyperlink and opening a tab. However, logs contain a lot of syscalls generated

by the page content. Some of them are not much distinguishable from browser-intrinsic behaviors, leading to mismatches. For `CUPS`, we identify new behaviors during the experiment which are variations of the existing models. `Transmission` is a threaded program with memory based synchronizations that are invisible to MCI. Hence, MCI misses some thread interdependences via memory.

Table 3.4.: Comparison with BEEP

	System subjects	**System objects**	**Edges**	**FP / FN**
BEEP	9.23	33.71	74.21	12.8% / 0.3%
MCI	9.18	25.38	62.87	0.1% / 0.1%

Comparison with BEEP. To evaluate the effectiveness of MCI when compared with BEEP, we randomly select 100 system objects (e.g., files or network connections) accessed in the week-long experiment. For each selected system object, we construct a causal graph by BEEP and by MCI, and compare the two. Table 3.4 shows the results. First of all, we observe that MCI has fewer false-positives and false-negatives. Again, we use LDX as the ground truth. Especially, MCI reduces the false-positive rate significantly. We investigate some of the cases that BEEP introduces false-positives, and find that many system objects accessed in a unit are included in the causal graphs while they are not causally related. Also, BEEP causes slightly more false-negatives due to missing inter-unit dependencies. We analyze the cases and find that the missing inter-unit dependencies were due to incomplete instrumentation caused by the difficulty of binary analysis in BEEP. We also manually investigate false-positive and false-negative cases from MCI. It turns out they are mostly caused by concurrent executions in `transmission`.

Runtime/memory Overhead. We also measure runtime overhead and memory overhead of MCI. Specifically, we report how long MCI takes to parse the audit log collected from the one week experiment which contains 3707 millions events. As we discussed in Sec. 3.4.2, we preprocess an audit log to extract indexes so that the parser can quickly locate skeleton instances. We measure the runtime performance and memory consumption of the trace preprocessor. It takes *4 hours 47 minutes* to preprocess (index) the entire log. The prepro-

cessor occupies around *2.8 GB of memory* on average. The parser first locates segments of the traces and launches automata within the identified segments. We find that the parser spend more time on parsing within the segments. In particular, the parser takes more time when it parses a wrong segment and eventually fails. Note that we parallelize the parsing within a segment to exploit multi-core processors. To parse the log, it takes *around 4 days (95 hours 43 minutes)*, and the parser consumes around *6.2 GB of memory* on average. We consider such one-time efforts reasonable given the huge log size. We leave performance optimization to our future work.

3.5.3 Case Studies

In this section, we present a few case studies to demonstrate the effectiveness of our approach in attack investigation.

Phishing email and camouflaged FTP server case

In this case, we use a scenario adapted from attack cases that were created by security professionals in a DARPA program [95], to demonstrate how MCI can effectively infer causality in a real-world security incident that happens across multiple programs including `PINE` and `Firefox`.

Attack Scenario. The user regularly uses `PINE` to send and receive emails. At some point, the user receives a phishing email, and she opens it, finds a hyperlink that looks interesting, and hence clicks the hyperlink. `PINE` automatically spawns the `Firefox` browser and the browser navigates to the given hyperlink. The hyperlink leads her to a web-page that contains an FTP server program. As she thinks the program is useful, she downloads the program. Before she closes the Firefox browser, she navigates a few more websites and downloads other files as well. Specifically, she opened 2 more tabs and downloaded 3 more programs.

After she closed the browser, she checked a few more emails and then opened a terminal to execute the downloaded FTP server program. The FTP server is a camouflaged trojan [96].

It normally behaves as a benign FTP server, serving remote FTP requests properly. However, it contains a backdoor which allows a remote attacker to connect and execute malicious commands on the victim computer. After she ran the trojan FTP server program, it served tens of benign FTP user requests with hundreds of FTP commands. A few hours later, the attacker connects to the machine through the backdoor, and modifies an important file (e.g., financial report). Later, the company identifies that the contents of the important file is changed and then hires a forensic expert to investigate the case to identify the origin of the incident.

Investigation. Given the causal models listed in Table 3.2 and a system-wide trace collected from the user's system, the forensic expert uses MCI to infer causal relations from the changed file. By matching models over the trace, MCI successfully identifies causality from the initial phishing email to the attacker's connection in the camouflaged trojan. The investigator further identifies that the important file is touched by the FTP server process. However, the file operation does not belong to any model instance. Interestingly, this indicates that the file is not part of regular behaviors, indicating that the FTP server may be trojaned. The investigator then tries to identify how the FTP server is downloaded and executed in the system. MCI reveals that a `Firefox` process downloaded the FTP server binary via y.y.y.y:80 through "LoadURI" and "Download a file" models. MCI further identifies that the `Firefox` process was launched by a `PINE` process when the user clicked a link from an email stored at */var/mail/.../94368.5222* downloaded from x.x.x.x.

We also investigate the same incident with BEEP, and find out that a causal graph generated by BEEP has a number of false-positives. Specifically, as shown in Fig. 3.17, the causal graph includes n.n.n.n:53 which is resolving the domain name, several other IP addresses from the `Firefox` process, which are from different tabs. Moreover, the causal graph contains other files downloaded from other tabs (*../file1* and *../file2*), two more sockets for internal messaging system (*unix socket*) and XWindow system (*/tmp/.X11-unix*), as well as some database files for storing browsing history (*/.../places.sqlite*).

In contrast, as MCI leverages accurate models generated by LDX, the graph generated by MCI is more accurate and precise without bogus dependencies. We also note that BEEP

requires training and binary instrumentation on the end-user site while MCI has no requirements on the end-user site.

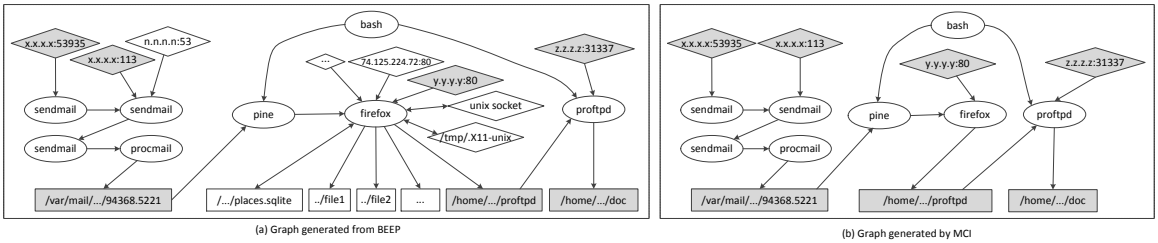


Figure 3.17.: Causal graphs generated from BEEP and MCI for the camouflaged FTP server case

Information Theft via InfoZip (zipsplit)

In this case, we use another insider attack to demonstrate the effectiveness of MCI. Specifically, an attacker in this case intentionally uses `zipsplit` to obstruct the investigation of the case as it reads and writes multiple input and output files where dependences between them are difficult to capture by existing approaches. We show how MCI can accurately identify the information flow through the program.

Attack Scenario. In this case, an insider tries to leak a secret document to a competitor company. However, the attacker's company forces all computer systems to enable audit logging system to monitor any attempts to exfiltrate important information. To avoid being exposed, he decides to use `zipsplit` before sending out the secret. Specifically, he understands that the `zipsplit` program can compress *n* files into *m* compressed files, and traditional audit logs are able to accurately identify causal relations if an input file is compressed to a *single* output file. Hence, the attacker used `zipsplit` to compress a secret document, *secret.pdf*, as well as two non-secret files, *1.pdf* and *2.pdf*, and generates four output files, *c1.zip*−*c4.zip*. In this example, the secret file is compressed and distributed into *c1.zip* and *c2.zip*, whereas *c3.zip* and *c4.zip* only contain non-secrets. Then he attached

all output files to an email, but before he sent it to the competitor company, he removed *c3.zip* and *c4.zip* from the email and only sent the other two that contain the secret. After that, he deleted all emails histories and compressed files.

A few days later, the company found suspicious behaviors from the attacker's computer. They identified that the secret document was accessed by `zipsplit`, and some files that may contain the secret were sent out. However, the attacker claimed that the secret document was mistakenly included in `zipsplit` and he only sent the zip files that contain non-secrets. At this point, the company started to investigate the attacker's machine to identify the source of outgoing files. Note that the investigator is not able to inspect the compressed files or email history as the attacker already deleted them.

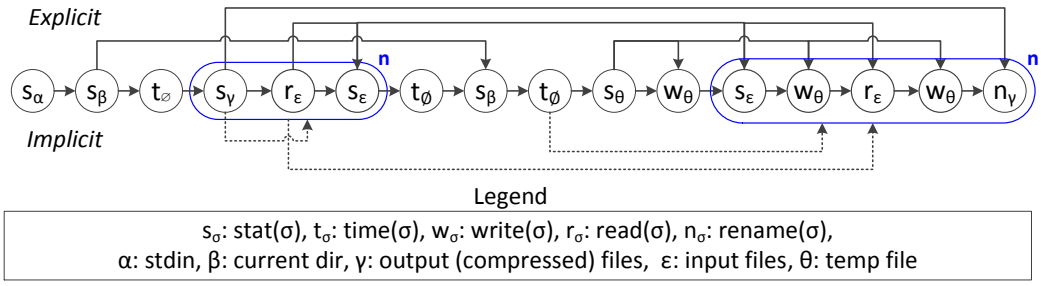


Figure 3.18.: Context-free model from zipsplit

Investigation. A forensic expert utilizes MCI to construct causal models for `zipsplit` and `PINE`. A related model for `zipsplit` is presented in Fig. 3.18, corresponding to the "read *n* files and compress to an output file" behavior. Note that it is context-free as there are two groups of nodes (from the 4^{th} to the 6^{th} and from the 12^{th} to the 16^{th}) that have the same number of repetition. The first group is for reading the meta information of the *n* input files and the second group is for reading the contents of the files and write to an output file.

MCI matches the models over the audit log collected from the attacker's machine, and it accurately reveals the causality between the secret document and the outgoing message. Fig. 3.19-(b) presents a causal graph generated by MCI. It shows that the *c1.zip* and *c2.zip* are derived from *secret.pdf*, and they are sent out via `PINE`. In contrast, Fig. 3.19-(a) shows a

causal graph generated by BEEP but it contains many false-positives as BEEP was not able to identify such removed attachments nor causal relations between inputs and outputs of `zipsplit`. We manually inspect the program to identify the root cause of false-positives. It turns out that `zipsplit` first compresses input files into a temporary file, then splits it into multiple output files. Hence, BEEP considers the temporary file is dependent on all input files, and the output files are dependent on the temporary file. In other words, BEEP considers all output files are dependent on all input files. Instead, MCI infers precise causality between each input and output file via implicit dependencies annotated in the model.

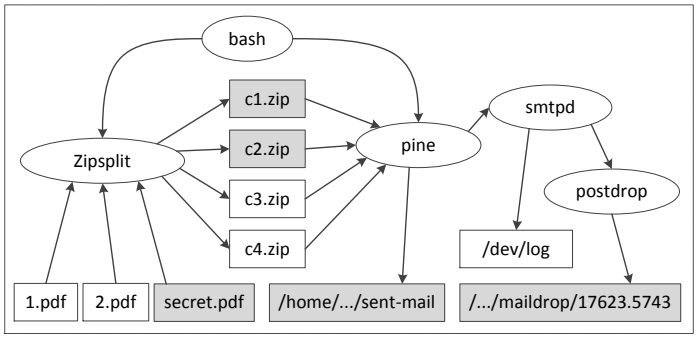


Figure 3.19.: Causal graphs for the zipsplit case

Table 3.5.: Evaluation on Long Running Executions

Access Log	**# of req. (unique)**	**Elapsed Time**	**FP / FN**
NASA-HTTP [97]	3.4M (36K)	19 hrs 41mins	3.9% / 0.2%
Our institution	5.6M (4.2M)	40 hrs 13mins	1.1% / 0.1%

Long running real world applications

In the last experiment, we evaluate MCI on large scale real world workloads. In particular, we use 2 months of NASA HTTP server access logs obtained from [97] as well as 3 months of our institution's HTTP server access logs (from Nov. 2015 to Jan. 2016).

To obtain audit logs from the HTTP access logs, we first emulate the web server environment by crawling all the contents of the original servers. Then, we create a script which connects and accesses the web server according to the access log so that the audit logging system on our server can regenerate logs for our analysis.

Table 3.5 shows the result. First, our parser takes 19 hours and 40 hours to parse the logs from [97] and our institution, respectively. Considering the size of the logs, we argue that our parser is reasonably scalable. For the accuracy test, we have 3.9% and 1.2% false-positives for the two respective logs. We analyze such cases and find that the NASA-HTTP log includes much more CGI requests than our institution's log. We find that most of the false-positive cases are from those CGI requests (e.g., PHP) that introduce noises. That is, some of the CGI behaviors are similar to the server behaviors and hence confuse our parser. We also have 0.2% and 0.1% false-negative rates. We manually analyze such cases and find out that they are mainly caused by CGI requests and suspicious requests embedding binary payloads, which crash the web-server during the experiment. Overall, the result shows that MCI is scalable to identify causality over large scale logs.

3.6 Related Work

Causality Tracking. There exists a line of work in tracking causal dependences for system-level attack analysis [46,47,66–68,98]. BackTracker [66] and Taser [46] propose backward and forward analysis techniques to identify the entry point of an attack and to understand the damage happened to the target system. Recently, a series of works [4, 72, 74] have proposed to provide accurate and fine-grained attack analysis. Dynamic taint analysis techniques [8, 76, 99] track information flow between taint sources and taint sinks. SME [100] detects information flows between different security levels by running a program multiple

times. LDX [1] proposes a dual execution based causality inference technique. When a user executes a process, LDX automatically starts a slave execution by mutating input sources. It identifies causal dependences between input source and outputs by comparing the outputs from the original and slave executions.

These approaches have limitations, for instance, syscall-based techniques suffer from imprecisions that cause false-positives and false-negatives, unit-based techniques require training or instrumentation on the end-user site, and dynamic taint analysis techniques cause too much runtime overhead. We discussed details of strengths and limitations of those techniques in Section 3.2 and compare them with MCI.

Program Behavior Modeling. Constructing program models that represent program's internal structures (e.g., control flow) or behaviors (e.g., system call invocations) have been extensively studied, especially in anomaly detection techniques [101–106]. Specifically, they train benign program executions to get models which are abstraction of the program behavior. Then, they use various ways such as DFA [102], FSA [101, 103], push-down automaton (PDA) [104], hidden Markov models [105], and machine learning [106, 107]. However, their models are mostly control flow models that do not have dependency information. Having dependences (acquired from LDX) in our models on one hand allows us to use models in attack provenance investigation, on the other hand poses a number of new technical challenges. Due to the difficulty of static binary dependency analysis, generating precise models using static analysis is highly challenging.

3.7 Discussion

Kernel-level Attack. We trust audit logs collected at the victim system. Most audit logging systems including Linux Audit and Windows ETW collect and store audit logs at the kernel level, and a kernel-level attack could disable the logging system or tamper with the log. One possible solution is to integrate with LPM-Hifi [75] that provides stronger security guarantees.

Limitations by LDX [1]. In our off-line analysis, we leverage LDX to construct causal models, hence, the limitations in LDX are also inherited by MCI. LDX doubles the resource consumption such as memory, processor and disk storage in order to run a slave execution along with original execution. However, we argue that the limitations only apply to the off-line analysis and do not apply the end-user.

Model Coverage. MCI relies on causal models generated by training with typical workloads. If an audit log includes behaviors that cannot be composed by the models in the provided workloads, MCI may not be able to infer causality precisely and could cause false-positives/negatives. Also, the FPs and FNs caused by missing models may cascade throughout the remaining MCI's parsing process. However, the cascading effect is mostly limited within a unit (e.g., each request in a server program) because MCI nonetheless starts a new model instance when it encounters an input syscall that matches with the model. Moreover, we can detect matching failures due to the incomplete models while MCI is parsing the audit log. For instance, missing models often lead to causal graphs lacking important I/O related system-objects (e.g., files/sockets), hence they are a strong indicator. Then we can enhance the model to resolve the situation by training with more workloads. Furthermore, we can fall back to a conservative strategy to assume unmatched events have inter-dependencies.

Although we mitigate the ambiguity problem (Sec. 3.3.3), as some models may not have enough dependencies to segment traces, ambiguity is still a challenge. We plan to investigate using irrelevant events as delimiters to further partition the trace and suppress ambiguity.

Signal and Exception Handler. Signals and exceptions can be delivered to a predefined handler at anytime, interrupting a normal execution flow. Unfortunately, it is possible that system calls in the handler may affect our parser. However, we observe that in practice our models are robust enough to handle the additional system calls caused by such handlers. This is because system calls invoked in a signal or exception handler are generally distinctive from the system calls in our causal models, hence our parser is able to filter them out. Moreover, in many programs such as `Lighttpd`, handlers functions often do not invoke

any system call. In the future, we plan to extend MCI to construct proper models for signal and exception handlers. As such, we can identify handler models from the audit log and extract them before we apply MCI's model parsing process.

4 A2C : SELF DESTRUCTING EXPLOIT EXECUTIONS VIA INPUT PERTURBATION

Malicious payload injection attacks have been a serious threat to software for decades. Unfortunately, protection against these attacks remains challenging due to the ever increasing diversity and sophistication of payload injection and triggering mechanisms used by adversaries. In this chapter, we develop A2C, a system that provides general protection against payload injection attacks. A2C is based on the observation that payloads are highly fragile and thus any mutation would likely break their functionalities. Therefore, A2C mutates inputs from untrusted sources. Malicious payloads that reside in these inputs are hence mutated and broken. To assure that the program continues to function correctly when benign inputs are provided, A2C divides the state space into exploitable and post-exploitable sub-spaces, where the latter is much larger than the former, and decodes the mutated values only when they are transmitted from the former to the latter. A2C does not rely on any knowledge of malicious payloads or their injection and triggering mechanisms. Hence, its protection is general. We evaluate A2C with 30 real-world applications, including `apache` on a real-world work-load, and our results show that A2C effectively prevents a variety of payload injection attacks on these programs with reasonably low overhead (6.94%).

4.1 Introduction

Attacks which exploit software vulnerabilities are among the most prevalent cybersecurity threats to date. This is due, in part, to many complex combinations of potential attack vectors: Buffer overflow attacks, Return-to-libc attacks [108], ROP [109], Jump-oriented programming (JOP) [110], and Heap spraying [111, 112] to name just a few. Unfortunately, this ever expanding variety of exploit attack vectors has led to a constant "cat and mouse game" of building defenses as each new attack is released.

In light of this, many existing protection mechanisms focus on specific attack vectors and become less effective (or even completely ineffective) for others. For example, non-executable stack and heap have difficulty preventing code reuse (e.g., ROP) attacks because the executable payload is constructed from the original code of the application. Shellcode detection techniques are only effective against injection of binary executable code and are often bypassable [113–116]. Control Flow Integrity [117–120] prevents attacks which exhibit certain abnormal control flows within a victim program. Further, some defense techniques may entail non-trivial overhead (e.g., [121]) or require hardware support (e.g., [122]), which affects their application in practice. Based on this trend of *attack-specific defense*, we are motivated to look for an entirely new, more fundamental weakness of software exploits to provide an *attack vector independent* protection mechanism.

It turns out that all software exploit attacks invariably have two common characteristics: First, they all need to inject an exploit payload into the target application. This payload could be a piece of executable code (shellcode) or information that allows constructing the malicious instruction sequence at runtime (e.g., a ROP chain that contains the entry addresses of gadgets). Second, these payloads are famously *brittle*. Specifically, exploit payloads are designed with very strict semantic assumptions about the environment (e.g., memory layout, libraries, or known binary instructions) which require *each byte of the payload to be carefully tailored to a victim*.

In this chapter, we will show that these invariant characteristics of exploit attacks make it possible to protect applications from exploit injections *independent* of the attack vector they use. Specifically, we leverage the observation that *exploit payloads (regardless of their attack vector) are so brittle that any mutation would break their execution* — i.e., cause the execution to crash. For example, even simple mutation of x86 shellcode results in invalid instructions. Similarly, most sequences of ROP addresses no longer form an executable gadget chain if even a single byte is changed. Secondly, since these exploit payloads must be injected into a victim application, their behavior eventually diverges from that of the application's legitimate inputs. Therefore, we propose that exploit payloads may be easily disabled via a "shoot first and ask questions later" policy, whereby all input to a victim

program is immediately mutated and only those that are beyond the control of the adversary are decoded.

Based on the above observations, we have developed the A2C (or "Attack to Crash") technique. A2C naturally exploits the brittleness of attack payloads by setting these attacks on track to crash before malicious logic is executed. First, any buffer inputs from untrusted sources are securely encoded using A2C's *One-Time Dictionary*, which varies for each input buffer to prevent memory disclosure/value guessing based attacks. Since all the untrusted inputs are mutated, malicious payloads that reside in these inputs are also mutated, resulting in broken payloads which will induce crashes when executed. Later, A2C must undo the mutation in the buffer inputs, when the program begins using these inputs to compute new values, so that our mutation does not cause any exceptions for legitimate input.

Our evaluation shows that A2C is able to protect a variety of applications against a wide spectrum of exploit attacks regardless of their injection methods, without affecting the normal functionalities of the program. Further, because A2C requires no knowledge of the specific attacks (only leveraging the two invariant characteristics mentioned above) it may even prevent currently unknown injection attack types in the future. The detailed threat model considered in this chapter is presented in Section 4.5.

Our contributions are summarized in the following:

- We propose the novel idea of partitioning program state space into the *exploitable* and *post-exploitable* sub-spaces so that we only need to protect the smaller exploitable sub-space, which is critical to A2C's efficiency and effectiveness.

- We develop a novel constraint solving based approach that can determine the boundary of the two sub-spaces. This serves as the basis to compute the execution points where the mutation can be safely undone.

- We develop a flow-, context-, and field-sensitive static analysis to identify the places at which A2C needs to undo the mutation so that execution on legitimate inputs is not affected.

- We develop an efficient runtime that leverages a *One-Time Dictionary*, which projects a value to another unique value. The dictionary varies for each input buffer to prevent memory disclosure based attacks. A2C also features efficient calling context encoding to support undoing input mutation.

- We develop a prototype A2C. The evaluation results show that A2C effectively prevents a number of known payload injection attacks with low overhead (6.94%).

4.2 System Overview

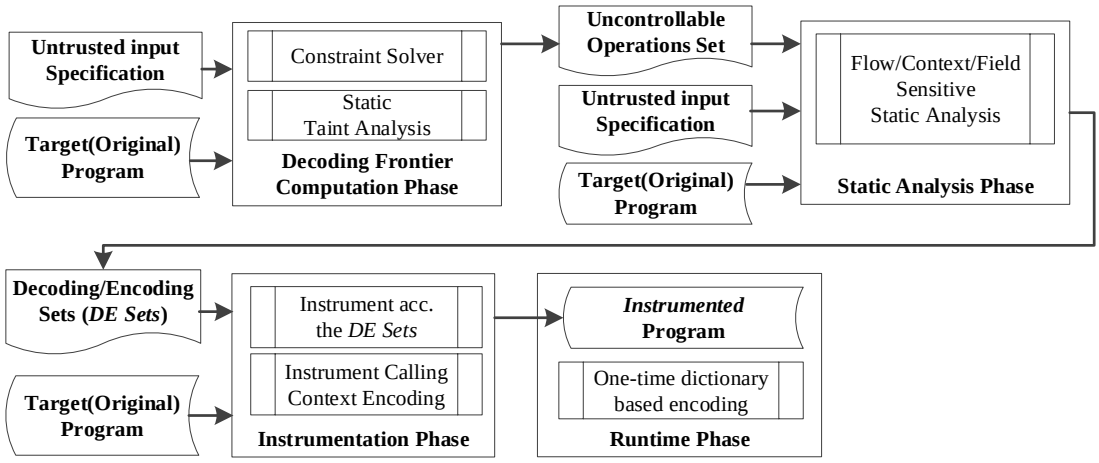


Figure 4.1.: Overall procedure of A2C

In this section, we present an overview of A2C, which is based on the following two observations. (1) *Most malicious payloads reside in buffers and they only go through copy operations or simple transformations before the attack is launched.* It is very rare for these payloads to undergo complex transformations in the victim program before being executed. This is due to the difficulty in controlling the transformations (in the victim program) to generate meaningful payloads. (2) *Malicious payloads are very fragile. Any mutation often leads to an unsuccessful attack.* For example, changing a few bits at the beginning of a shellcode can easily throw off the sequence of executed instructions, leading to a crash.

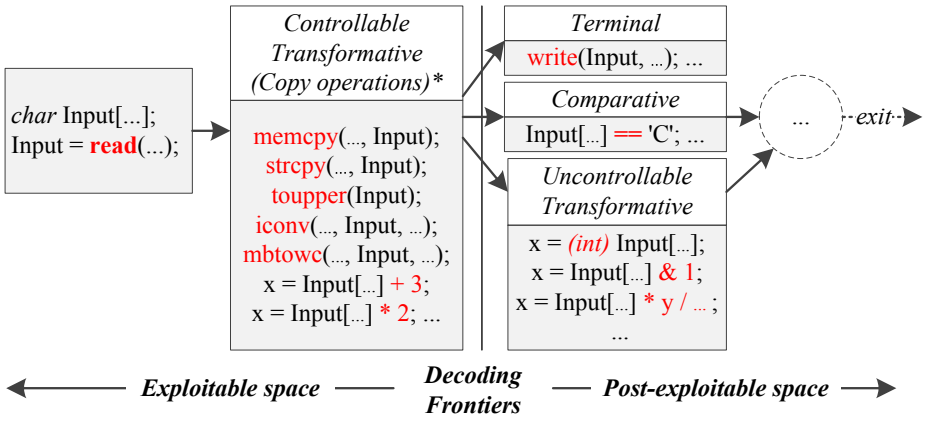


Figure 4.2.: Decoding frontiers

The overarching idea of A2C is to *protect a program from malicious injection attacks by perturbing or encoding inputs from untrusted sources*. However, inputs from untrusted sources (e.g., packets from remote IPs) are not necessarily malicious. We need to ensure that our perturbation does not fail executions based on non-exploit inputs. According to observation (1), we aim to undo the perturbation when the buffer data goes beyond copy operations/simple transformations and starts being used in benign computation.

In the following, we use the diagram in Fig. 4.2 to illustrate the life cycle of buffer data and hence the intuition behind A2C. After the buffer data are loaded through input functions, they may undergo a number of transformations, including copy operations (e.g., **memcpy**() and **strcpy**()) that copy a buffer to another target buffer, constant table lookup (e.g., in **iconv**(), **toupper**(), **mbtowc**(), and **wctomb**()), and simple transformative operations (e.g., additions with a constant). Then, the buffer data will eventually encounter one of the following three kinds of operations: (1) *Comparative operations*, in which elements in the buffer are used in comparisons; (2) *Terminal operations*, in which the buffer data are passed to output library functions (e.g., **write**(), **send**(), and **printf**()); (3) *Uncontrollable transformative operations*, in which elements in the buffer undergo transformations that disallow the attacker to control the values beyond these transformations to construct meaningful payloads. For instance, *type widening* copies a value of smaller type (e.g., char) to an array element of larger type (e.g., integer) so that each element in the array is padded

with leading 0's. As such, the resulting byte sequence denoted by the array cannot serve as a meaningful payload.

We call these three kinds of operations the *decoding frontier* (DF) because A2C should undo the perturbation for the buffer elements involved before executing the operations. Intuitively, we consider the space before the frontier the *exploitable space* where the malicious payloads are supposed to take effect and *without* perturbation would successfully exploit the program. Therefore, we use perturbation to achieve protection in this space. The space after the frontier is referred to as the *post-exploitable space*. This is because controlling the payload becomes infeasible if it has gone through these benign transformations conducted by the victim program. Therefore, it is safe to undo our perturbation before the decoding frontier so that benign inputs can be used in computation as usual[1]. The core technical challenge for A2C is hence to identify the DF of a subject program and perform instrumentation accordingly. More discussion about the decoding frontier can be found in Section 4.4.1.

Another interesting observation that makes our solution feasible is that the exploitable space is usually much smaller than the post-exploitable space as most computation happens in the post-exploitable space. As such, the frontier tends to be small and shallow and as explained above, operations beyond the frontier do not need our attention.

Overall Procedure. Fig. 4.1 shows the complete procedure of A2C. There are four phases: *constraint solving based decoding frontier computation*, *static analysis for determining encoding and decoding places which are a superset of the decoding frontier*, *instrumentation*, and *runtime*.

First, we leverage constraint solving to determine the uncontrollable operations. These operations, together with the comparative and terminal operations, form the decoding frontier. This phase simply marks all the operations on the frontier.

Second, a flow-, context-, and field-sensitive analysis is applied to determine the places to instrument. It takes three inputs: the LLVM IR of the program, the decoding frontier from the first phase, and the untrusted input specification that identifies a set of library

[1]Here we assume that output library functions are hardened and thus cannot be exploited by the decoded buffers.

functions that read inputs, such as **recv**() for network inputs and **read**() for file streams. In this phase, A2C produces two outputs. Specifically, the *decoding set* is a superset of the decoding frontier and the *encoding set* contains the statements to encode (input) values, such as **recv**() in network programs. Interestingly, the encoding set may also contain instructions that load constant values. Explanations about why we need to encode constants can be found in Section 4.4.3. The computation of decoding and encoding sets (DE sets for short) is iterative as new elements on encoding sets may introduce additional decoding operations.

Third, the instrumentation phase statically instruments the program according to the DE sets. An important observation is that the decoding frontier is context sensitive. Different inputs may lead to different calling contexts of a function invocation. The membership of a statement in the DE set may change with those contexts. As such, upon the execution of a statement in the DE set, we need to know the current calling context to determine if the instrumented version or the original version of the statement should be executed. Therefore, part of the instrumentation phase handles the problem of efficiently tracking the current calling context.

Lastly, the runtime supports execution of the instrumented program. It features encoding based on a *One-Time Dictionary*, which projects a plaintext value to a unique encoded value. Different input buffers use different dictionaries to prevent memory exposure based exploits.

4.3 Illustrative Example

In this section, we use a real-world example to illustrate A2C's operation. We use the `nginx` 1.4.0 web-server as the subject program. It has two known heap buffer overflow and integer overflow vulnerabilities, which can be triggered by providing crafted HTTP requests containing malicious payloads. Fig. 4.3 shows two code snippets with part of the original `nginx` program on the left and the corresponding instrumented version on the right. The column in the middle shows how the two code snippets process the request differently.

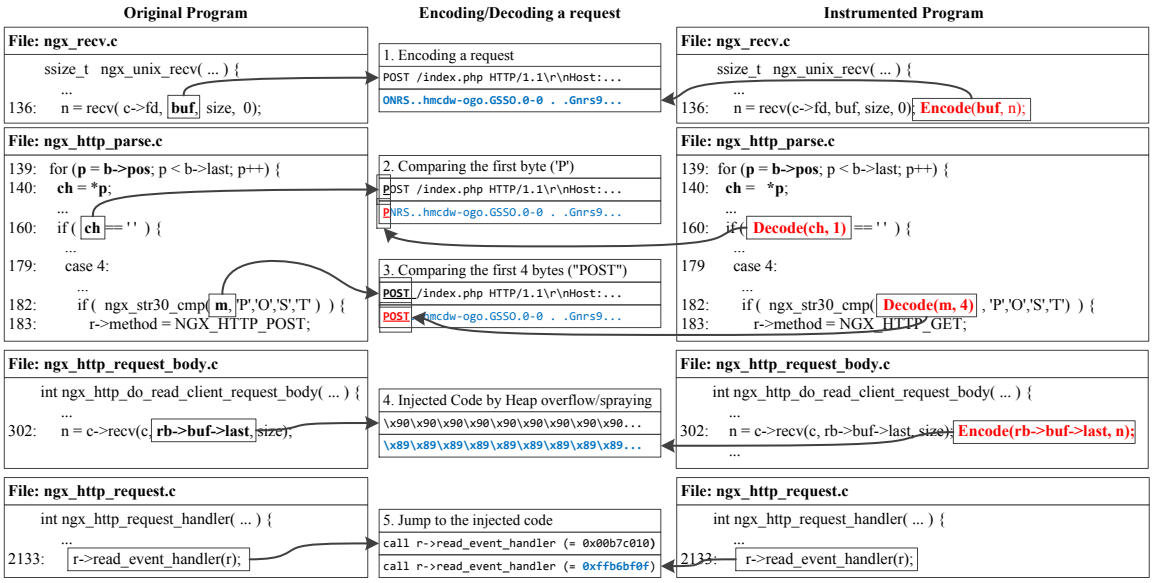


Figure 4.3.: Original and instrumented programs of demonstrative example

First, both programs receive a POST request at Line 136 in `ngx_recv.c`. Since the request is from an untrusted source, the instrumented program encodes the buffer. For simplicity of discussion, the encoding here is to subtract 1 from every byte. **Encode**() denotes this modification. The HTTP request "`POST /index.php HTTP/1.1\r\nHost: ...`" is hence encoded as "`ONRS..hmcdw-ogo.GSSO.0-0..Gnrs9...`". The request is parsed at Lines 160 and 182 in `ngx_http_parse.c`, which contain comparative operations on some buffer data and are hence part of the decoding frontier. Therefore, the instrumented program calls **Decode**() to undo the perturbation so that the program can parse and process the request correctly. Note that it only decodes a few bytes (of fixed length) at a time so that *the decoded data cannot be run as any meaningful payload*. Also observe that the original buffer remains encoded. This is achieved by only decoding the values after they are loaded into variables of primitive types (e.g., bytes and words).

Next, the `ngx_http_do_read_client_request_body()` function stores the contents of the request into a different heap buffer. Notice that without A2C this becomes vulnerable to heap spraying attacks which can be further leveraged to launch attacks such as ROP. Also, the same function has a heap buffer overflow vulnerability that allows overwriting a

function pointer, `read_event_handler`, which will be called inside `ngx_http_request_handler()`. However, since the instrumented program encodes all external requests, the payload at Line 302 and the address accessed at Line 2133 are mutated. Assume the malicious shellcode contains a sequence of nop instructions (`0x90*n`) for the *nop-sled* portion of a heap spray attack and the malicious address injected is `0x00b7c010`. In the instrumented program, the nop instructions (`0x90*n`) are encoded to "`0x89*n`", which denotes a sequence of mov instructions that write to invalid memory locations (e.g. `mov ecx, ecx(-76767677h)`). At this point, even though the shellcode is successfully injected, due to the mutation, it crashes upon execution. Similarly, the injected function pointer at Line 2133 is also broken. Note that if the request is valid, despite it being encoded by the instrumented program, it will be decoded at the frontier and will not affect normal execution.

4.4 Design

4.4.1 Decoding Frontier Computation via Constraint Solving.

The first phase of A2C is to determine the decoding frontier that will be used to identify the encoding and decoding sets in the next analysis phase. As we will see in the next section, A2C needs to decode at more places than input related buffers.

According to the definition in Section 4.2, the decoding frontier consists of three kinds of operations: comparative, terminal, and uncontrollable. While the identification of the first two is straightforward, we focus on the third in this section.

We first define *controllable operations* as follows: *if valid payloads can be generated in a memory region (e.g., a buffer) right after a set of operations by manipulating program inputs, these operations are controllable*. An example of a controllable operation is the `toupper()` transformation that turns a lower case character into its upper case. Assume an application transforms a text input buffer A into another buffer B using `toupper()`. The attacker can carefully prepare the input so that after the transformation, buffer B contains the intended payload. It was indeed reported that existing operations in a program could be leveraged to compute/decode payloads [123].

We further formulate the determination of controllable operations as a constraint solving problem. We consider program inputs as symbolic variables. We further model the operations that compute the values for a memory region (at a given program point) from the program inputs as a set of constraints. We then assert the values (of the memory region) to be some valid payload and query a solver if there is a satisfying (SAT) solution. If so, one may be able to manipulate the input (e.g., using the SAT solution generated by the solver) to induce the given payload. While it is difficult to precisely define what constitutes a valid payload, we use the following procedure to determine if operations are controllable.

Procedure to Determine Decoding Frontier. Given a program to protect, A2C identifies all memory regions larger than or equal to 16 bytes that can be affected by inputs (through a standard static taint analysis). These regions include buffers, consecutive local variables (on stack), consecutive global variables (in data section), as well as structures. For example, four consecutive local integer variables related to inputs constitute a region for testing. For these regions, A2C creates constraints according to the operations that compute the values in the regions from program inputs. Other variables that are not related to inputs are considered as free variables. This makes our analysis a conservative one as free variables can take any values during constraint solving, whereas in practice these variables may have various restrictions. After we generate the constraints, we use the Z3 solver [124] to test whether payloads can be generated through these operations. In particular, we collected 1.4GB binary codes, 200MB shellcode, and 200MB ROP gadgets from Internet [125–129]. We also generate 1.0G random numbers. We further break the data sets down to sequences based on the size of the region under testing. If the size is unknown, we use 16-byte sequences. We then assert the values of the region equal to each of these sequences one by one. If the constraint solver yields SAT, TIMEOUT, or UNKNOWN for *any* of the sequences, which implies that an attacker may be able to construct some malicious payload through the operations, then the operations are considered controllable. If the constraints are UNSAT for *all* these sequences, the operations that define the values of the memory region are considered uncontrollable.

Essence. Intuitively, we use the large pool of binary code and shell code snippets to model the distribution of executable payloads and the large pool of ROP gadget subsequences to model the distribution of address-based payloads (for code reuse attacks). We further use a large set of random number sequences to model the distribution of other *arbitrary* payloads. Since we only consider operations uncontrollable when *all* these sequences yield UNSAT results, A2C provides strong probabilistic guarantees that the values beyond these operations are not exploitable.

Note that for complex programs, it may be difficult to model the entire data flow from program inputs to the memory region of interest due to various reasons such as unmodeled library calls and uncertainty of data flow caused by aliasing. A2C leverages backward slicing, starting from the memory region of interest and traverses backward along data dependencies until the traversal becomes infeasible (e.g., due to unmodeled library calls). If program inputs cannot be reached by the traversal, A2C treats the farthest variables that it can reach as free variables. Note that this yields an over-approximation, which is safe. The decoding frontier analysis marks all the operations on the decoding frontier. Since the algorithms in this phase are standard, details are omitted.

In the following, we use a number of examples to facilitate understanding of decoding frontier.

Uncontrollable Operation Example One. Fig. 4.4 shows a code snippet from 464.h264ref (i.e., a video decoding program) in SPEC 2006.

(a) Code snippet from 464.h264ref

```
// Declarations (Data Types)
1. unsigned int      m7[...][...];
2. unsigned short    img[...][...];
3. unsigned short    mpr[...][...];
   ...
// Transformative Operations
4. for (int x = 0; ...; x++ )
5.   for (int y = 0; ...; y++ )
6.     m7[x][y] = img[...][...] - mpr[...][...];
```

(b) Constraints from the code snippet

```
; Constraints for Operations (img - mpr)
7.  m7[0,1,2,3] = img[0,1,2,3] - mpr[0,1,2,3]            /\

; Constraints for the range of unsigned short
8.  0 <= img[0,1,2,3] /\ 0 <= mpr[0,1,2,3]               /\
9.  img[0,1,2,3] <= 65535 /\  mpr[0,1,2,3] <= 65535 /\

; Constraints for Payloads (i will select a payload)
10. m7[0,1,2,3] = payload[i, i+1, i+2, i+3]
```

Figure 4.4.: Uncontrollable operations due to type widening in 464.h264ref

Fig. 4.4 (a) shows three arrays `m7`, `img`, and `mpr` with `m7` a temporary array that stores intermediate values during encoding, `img` holding raw input values and `mpr` calculated by the program and not related to inputs. Observe that `m7` is an *int* array whereas the other two are arrays of *short int*. Fig. 4.4 (b) shows the constraints generated. Lines 7-9 denote the constraints representing the operations. Line 7 denotes the subtraction at Line 6. Line 9 denotes the range constraints of `img` and `mpr`. We use "0,1,2,3" to represent that the same constraint applies to four respective elements. Line 9 denotes the payload assertion. We iterate this test with *i* from 0 to the number of sequences in our test data set.

The test result shows that the constraints are always UNSAT. This is mainly because the assignment of *short* to *int* (called *type widening*) requires payloads to have two zero bytes in every four bytes. As such, Line 6 is on the decoding frontier. Type widening is one of the major reasons for uncontrollability. Another popular form of type widening is through bit operations, namely, only a few bits of a word are set. Examples are omitted.

Uncontrollable Operation Example Two. Another common kind of uncontrollable operation is one that induces intensive correlations between values. For example, Fig. 4.5 (a) shows a code snippet from 429.mcf in SPEC.

(a) Code snippet from 429.mcf	**(b) Constraints from the code snippet**

// Declaration (Data Types)
1. typedef struct network{
2. **long** n, n_trips, max_m, m;
...
3. } network_t;
...
4. network_t* net;
5. in[2] = read(*InputFile*);
// Transformative Operations
6. net->n_trips = in[0];
...
7. net->n = (in[0]+in[0]+1);
8. net->m = (in[0]+in[0]+in[0]+in[1]);
9. *if (...)* net->max_m = net-> m;
10. *else* net->max_m = 0xA10001;
	; Constraints for Operations
	11. net[0] = (2 * in[0] + 1) ∧
	12. net[1] = in[0] ∧
	13. ((net[2] = (3 * in[0] + in[1])) ∨
	14. (net[2] = 0xA10001)) ∧
	15. net[3] = (3 * in[0] + in[1]) ∧
	
	; Constraints for Payloads
	; (*i* will select a payload to test)
	16. net[0] = payload[i] ∧
	17. net[1] = payload[i+1] ∧
	18. net[2] = payload[i+2] ∧
	19. net[3] = payload[i+3]

Figure 4.5.: Uncontrollable operations in 429.mcf program

Fields n, n_trips, max_m, and m are consecutive in the structure network and they are all related to inputs (in[0] and in[1]). As such, A2C needs to test if the operations on these fields are controllable. The constraints are shown in Fig. 4.5 (b). Observe that the net→max_m (i.e., net[3] in the constraint) and net→m (i.e., net[4]) are identical except when net→max_m has a constant value 0xA10001. The other 8 bytes are also closely correlated through in[0] and in[1]. Consequently, the solver returns UNSAT for all the payload tests.

Controllable Operation Examples. Most controllable operations are straightforward, such as copy operations. Method toupper() is another example of a controllable operation. The solver returns SAT for many payload sequences, such as consecutive 0x90's, which represent the NOP instructions (nop-sled) in exploits. A2C also determines unicode conversion functions (e.g., mbtowc()) as controllable. This is because while unicode conversion translates an ASCII character to two bytes with an additional byte (0x00), it also translates two byte characters such as Chinese, Japanese, and Korean characters to two bytes [130], making payload construction feasible. Our results echo the message conveyed in [123] that Unicode conversion function can be leveraged to construct payloads. In fact, all the data conversion/encryption/decryption/encoding via table lookup (e.g., iconv(), mbtowc(), wctomb(), and Inflate (Huffman Coding) Algorithm) are recognized as controllable by A2C.

Interestingly, we also observe that some operations of complex types and performing complex computations are determined as controllable by our analysis. Consider the following example that leverages existing floating point operations to construct malicious payloads. According to the IEEE-754 floating point representation standard, even a very small floating point value can affect all the 4 bytes of its presentation. For example, a floating point variable 0.0001 is encoded as 0x38d1b717 in memory. Fig. 4.6 shows FNorm() in 456.hmmer from SPEC. It first adds all elements in v into sum using FSum(), and then each element is divided by the sum if the sum is not 0.0. If the sum is 0.0, all the elements in v have 1.0 / n where n is the size of v. Note that when there are multiple definitions of a variable (e.g., v[x]), A2C disjoins the constraints for these definitions, which are rep-

(a) Code snippet from 456.hmmer

```
// Declarations (Data Types)
1. float v[...], sum;
2. int x, n;
// Transformative Operations
3.   sum = FSum(v, n);
     // FSum returns a sum of all elements.
4.   if (sum != 0.0)
5.     for (x = 0; x < n; x++)
6.       v[x] /= sum;
7.  else
8.     for (x = 0; x < n; x++)
9.       v[x] = 1. / n;
```

(b) Constraints from the code snippet

```
; Constraints for Operations
10.  sum = v_old[0] + v_old[1] + v_old[2] + v_old[3] /\
11.  (v_new[0] = (v_old[0] / sum) or (1.0 / n)) /\
12.  (v_new[1] = (v_old[1] / sum) or (1.0 / n)) /\
13.  (v_new[2] = (v_old[2] / sum) or (1.0 / n)) /\
14.  (v_new[3] = (v_old[3] / sum) or (1.0 / n)) /\
; Constraints for Payloads
; (i will select a payload to test)
15. v_new[0] = payload[i]      /\
16. v_new[1] = payload[i+1]    /\
17. v_new[2] = payload[i+2]    /\
18. v_new[3] = payload[i+3]
```

Figure 4.6.: Controllable operations in 456.hmmer program

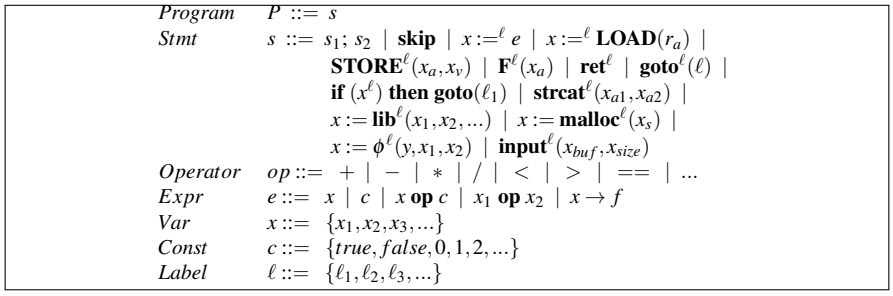
resented in the SSA form. The solver returns SAT for the constraints. The exploit input is a sequence of values (e.g., $-12068, -18966, -14108, -13991, ...$) whose binary representations do not denote any meaningful payload. But they are transformed to a meaningful payload by the operations in Fig. 4.6. The payload issues a system call through `int 0x80` with arguments.

4.4.2 Static Analysis to Compute Decoding and Encoding Sets

In this section, we discuss the second phase, i.e., the computation of decoding and encoding sets.

Language. A2C works on the Single Static Assignment (SSA) LLVM IR, which is generated from program source code. To facilitate precise discussion, we introduce a simplified language which models the LLVM IR in Fig. 4.7.

Memory loads and stores are denoted by **LOAD**(x_a) and **STORE**(x_a, x_v), respectively, with x_a holding the address and x_v the value. The address of a field access is explicitly computed by $x := x_{base} \rightarrow f$ with x_{base} the base pointer and f the field. Array accesses can be considered as a special kind of field accesses. $\mathbf{F}(x_a)$ models a call to function **F** with x_a the actual argument and x_f the formal argument. Function return is modeled by **ret**.



Program	$P ::= s$	
Stmt	$s ::= s_1; s_2 \mid \textbf{skip} \mid x :=^{\ell} e \mid x :=^{\ell} \textbf{LOAD}(r_a) \mid$	
	$\textbf{STORE}^{\ell}(x_a, x_v) \mid \textbf{F}^{\ell}(x_a) \mid \textbf{ret}^{\ell} \mid \textbf{goto}^{\ell}(\ell) \mid$	
	$\textbf{if } (x^{\ell}) \textbf{ then goto}(\ell_1) \mid \textbf{strcat}^{\ell}(x_{a1}, x_{a2}) \mid$	
	$x := \textbf{lib}^{\ell}(x_1, x_2, ...) \mid x := \textbf{malloc}^{\ell}(x_s) \mid$	
	$x := \phi^{\ell}(y, x_1, x_2) \mid \textbf{input}^{\ell}(x_{buf}, x_{size})$	
Operator	$op ::= + \mid - \mid * \mid / \mid < \mid > \mid == \mid ...$	
Expr	$e ::= x \mid c \mid x \textbf{ op } c \mid x_1 \textbf{ op } x_2 \mid x \rightarrow f$	
Var	$x ::= \{x_1, x_2, x_3, ...\}$	
Const	$c ::= \{true, false, 0, 1, 2, ...\}$	
Label	$\ell ::= \{\ell_1, \ell_2, \ell_3, ...\}$	

Figure 4.7.: Language

Conditional or loop statements are not directly modeled. Instead we define jumps using **goto** and guarded **goto**. Conditional and loop statements can be constructed by combining jumps and guarded jumps. **strcat**(x_{a1}, x_{a2}) denotes a function that concatenates two strings. It appends the second string denoted by pointer x_{a2} to the first string x_{a1}. We define **lib**$(x_1, x_2, ...)$ to model library calls. It takes several x_n's as arguments and returns a value in another variable. Function **input**(x_{buf}, x_{size}) models library calls that read inputs such as `read()` and `recv()`. The $x := \phi(y, x_1, x_2)$ denotes the ϕ function in SSA that determines the value of a variable at the joint point of two branches. In particular, if y is true, $x := x_1$ otherwise $x := x_2$. We also explicitly model heap allocation through the **malloc**() function.

Operator denotes uncontrollable (computed by the previous phase) or comparative operations. Each statement is annotated with a label, which can be intuitively considered as the line number of the statement in the program.

4.4.3 Static Analysis Phase

We formulate the static analysis as an abstract interpretation process. Intuitively, abstract interpretation can be considered as "*executing*" the program on the *abstract domain* instead of the concrete domain. The abstract domain is specific to an analysis. In abstract interpretation, it is often the case that branch outcomes cannot be statically determined. Therefore, it assumes all branches are possible. In the presence of loops, the interpretation may go through the loop bodies multiple times until a fix point is reached. If the abstract domain is well designed, the interpretation procedure is guaranteed to terminate.

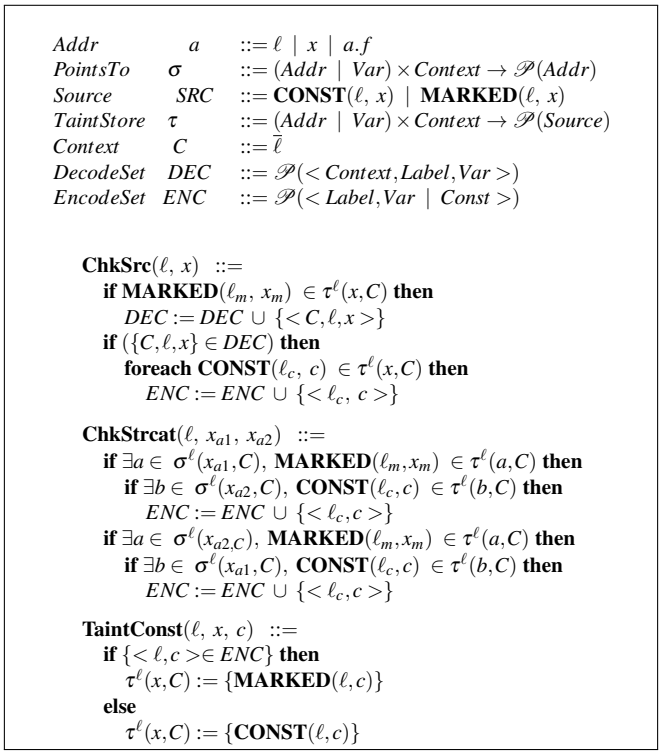


Figure 4.8.: Definitions for abstract interpretation rules

Before the abstract interpretation, constants are propagated during preprocessing using an existing LLVM pass (e.g., $x_1 * x_2$ is rewritten to $x_1 * c$ if x_2 is determined to hold a constant c). During the analysis, A2C iteratively goes through program statements following the control flow and updating the corresponding abstract states (e.g., the decoding set) until a fix point is reached. Specifically, A2C taints input buffers from untrusted sources. The taints are propagated through controllable operations, which may be conducted through library functions (e.g., `memcpy()`, `toupper()`, and `iconv()`), linear operations (e.g., $y = x$ and $y = 3 * x$), and so on. If a tainted value reaches an operation on the decoding frontier computed in the previous phase, which includes comparative, uncontrollable, and terminal operations, taint propagation is terminated and the operation is added to the decoding set. However, the decoding set may be context-sensitive and path-sensitive. To handle such cases, statements that load constant values may need to be considered as sources and hence

encoded. As a result, more statements may be added to the encoding set and the decoding set.

Definitions. To facilitate discussion, we introduce a few definitions in Fig. 4.8. Our analysis computes four kinds of abstract information: the points-to set, the taint set, and the encoding and decoding sets. The points-to set σ is a mapping from an abstract address a (representing some memory location) or a variable x, together with the calling context, to a set of abstract addresses denoting the memory locations that may be pointed-to by a or x. Abstract address *Addr* is denoted by some variable representing an abstract global/stack array/buffer or a label denoting an abstract heap buffer, followed by a sequence of fields. Intuitively, one can consider it as the reference path to some abstract memory location. The role of abstract addresses in our static analysis is similar to that of concrete addresses in dynamic analysis (e.g., to look up taint values). Since our analysis is context-sensitive and field-sensitive, context is part of the mapping and fields are explicitly modeled in abstract addresses.

Source represents the (taint) source of a value. There are two types of *Source*: **CONST** and **MARKED**, meaning a constant value and an untrusted input source, respectively. We use the term **MARKED** to indicate that a value originates from some input buffer and has only gone through controllable operations. Hence it is in the exploitable space (Section 4.2). Such values shall be in their encoded form at runtime. We track the **MARKED** value propagation through our analysis. *TaintStore* τ stores the (taint) source information for abstract addresses and variables. Both σ and τ are flow-sensitive, meaning that A2C computes separate σ and τ for different program locations (i.e., labels). For example, we use τ^{ℓ} to denote the abstract taint mapping computed at ℓ. It is implicit in the rest of the chapter for simplicity in discussion.

If **MARKED** values reach an operation on the decoding frontier, the operation is inserted to the *DecodeSet DEC*. The *EncodeSet ENC* contains the set of statements at which the (input) values ought to be encoded. *Context C* is denoted by a sequence of labels (ℓ's) that models a call stack. Each element in the *DEC* set includes a *Context*, suggesting that we decode input buffers depending on the calling context. For example, $\langle C, \ell, x \rangle \in DEC$

Table 4.1.: Abstract Interpretation Rules

Statement	Interpretation Rule	Name
$\textbf{input}^{\ell}(x_b, x_s)$	**foreach** $a \in \sigma^{\ell}(x_b, C)$ $\quad \tau^{\ell}(a, C) := \textbf{MARKED}(\ell, x_b)$; $\quad ENC := ENC \cup \{\langle \ell, x_b \rangle\}$;	INPUT
$x :=^{\ell} x_1$ $(x =^{\ell} x_1 \textbf{ op } c)$	$\sigma^{\ell}(x, C) := \sigma^{\ell}(x_1, C)$; $\tau^{\ell}(x, C) := \tau^{\ell}(x_1, C)$;	NON-DF-OP
$x :=^{\ell} \textbf{LOAD}(x_a)$	$\sigma^{\ell}(x, C) := \bigcup_{\forall a \in \sigma^{\ell}(x_a, C)} \sigma^{\ell}(a, C)$ $\tau^{\ell}(x, C) := \bigcup_{\forall a \in \sigma^{\ell}(x_a, C)} \tau^{\ell}(a, C)$	LOAD
$\textbf{STORE}(x_a, x_v)$	$\forall a \in \sigma^{\ell}(x_a, C) : \sigma^{\ell}(a, C) \cup := \sigma^{\ell}(x_v, C)$ $\forall a \in \sigma^{\ell}(x_a, C) : \tau^{\ell}(a, C) \cup := \tau^{\ell}(x_v, C)$	STORE
$x :=^{\ell} x_1 \textbf{ op } x_2$	$\sigma^{\ell}(x, C) := \perp$; $\textbf{ChkSrc}(\ell, x_1)$; $\textbf{ChkSrc}(\ell, x_2)$;	DF-OP
$x :=^{\ell} x_1 \rightarrow f$	$\sigma^{\ell}(x, C) := \{a \cdot f \mid \forall a \in \sigma^{\ell}(x_1, C)\}$	FIELD
$x :=$ $\textbf{lib}^{\ell}(x_1, x_2, ..)$	**for each** $x_i \in \{x_1, x_2, ...\}$ $\quad \textbf{ChkSrc}(\ell, x_i)$;	DF-TERM
$x :=^{\ell} c$	$\textbf{TaintConst}(\ell, x, c)$;	CONST
$\textbf{strcat}^{\ell}(x_{a1}, x_{a2})$	$\textbf{ChkStrCat}(\ell, x_{a1}, x_{a2})$;	STRCAT
$\textbf{F}^{\ell}(x_a)$	$C_0 := C$; $C := C \cdot \ell$; // x_f formal arg $\sigma^{\ell}(x_f, C) := \sigma^{\ell}(x_a, C_0)$; $\tau^{\ell}(x_f, C) := \tau^{\ell}(x_a, C_0)$; **foreach** buffer var $y \in F$: $\quad \sigma^{\ell}(y, C) = \{y\}$;	CALL
ret	$C := C - \textbf{last}(C)$;	RET
$x := \phi^{\ell}(y, x_1, x_2)$	$\sigma^{\ell}(x, C) := \sigma^{\ell}(x_1, C) \cup \sigma^{\ell}(x_2, C)$; $\tau^{\ell}(x, C) := \tau^{\ell}(x_1, C) \cup \tau^{\ell}(x_2, C)$;	PHI
$x := \textbf{malloc}^{\ell}(x_s)$	$\sigma^{\ell}(x, C) := \ell$;	HEAP

suggests that when the statement denoted by ℓ is encountered under context C at runtime, A2C will decode the variable x.

Decoding Set is Context-Sensitive and Path-Sensitive. The membership of a statement in the decoding set may change with the context. Fig. 4.9 shows an example in `ngircd`, an Internet Relay Chat (IRC) daemon program. In this example, we treat all network functions as untrusted input sources. Thus, the input data from these functions are encoded while data from files are not. `ngt_TrimStr()` is a utility function for trimming a string. It is invoked at different places. For instance, `Read_Config()` calls it with a string from the configuration file, which is not encoded. On the other hand, `Parse_Request()` also calls

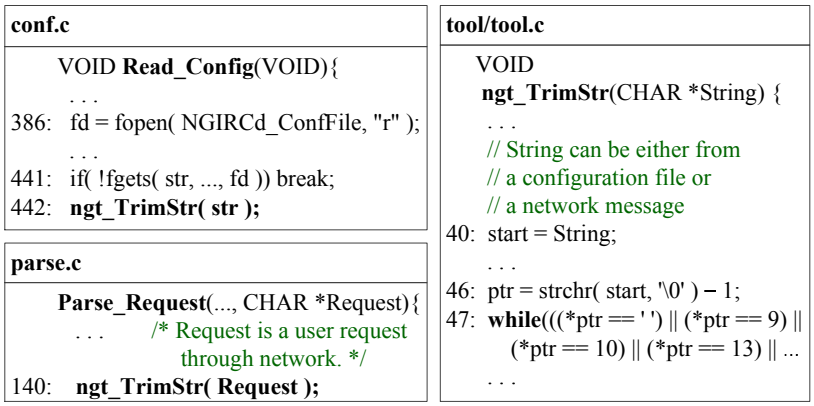


Figure 4.9.: An example of context sensitive code

it, but with a string from the network. The string is encoded this time. Hence, A2C may or may not decode the value in `*ptr` at Line 47, depending on the context. Therefore, each statement in the *DEC* set is annotated with a context such that decoding is only performed when the same context is encountered at runtime.

The decoding set is also path-sensitive. Consider the example in Fig. 4.10 (a), which contains code snippets from `unrtf`, a program for converting documents in Rich Text Format (RTF) to other formats such as HTML and LaTeX. At 2 and 3 , `str` may hold a constant value or a tainted value `ch`. At 4 and 5 , `str` is inserted to a hash map. Strings in the hash map are loaded and used at 6 . Depending on whether 2 or 3 is executed, Line 336 may or may not belong to the decoding set. In other words, if `tmp` holds a constant string at 336, it does not need to be decoded. Note that in this case, the context of Line 336 cannot be used to distinguish the different behaviors of the line. We cannot afford to track paths at runtime either. Hence, our solution is to identify the related constant strings, such as that at Line 326, and treat them as input sources so that they will be encoded as well. As a result, the behavior at Line 336 becomes path insensitive, always requiring decoding. □

Abstract Interpretation Rules. The interpretation procedure is formulated by the rules in Table 4.1, which specify how the abstract information is updated upon each statement. Specifically, when the program reads data from untrusted input sources through **input**(x_b, x_s) with x_b the buffer address and x_s the size, the *TaintStore* of all the abstract

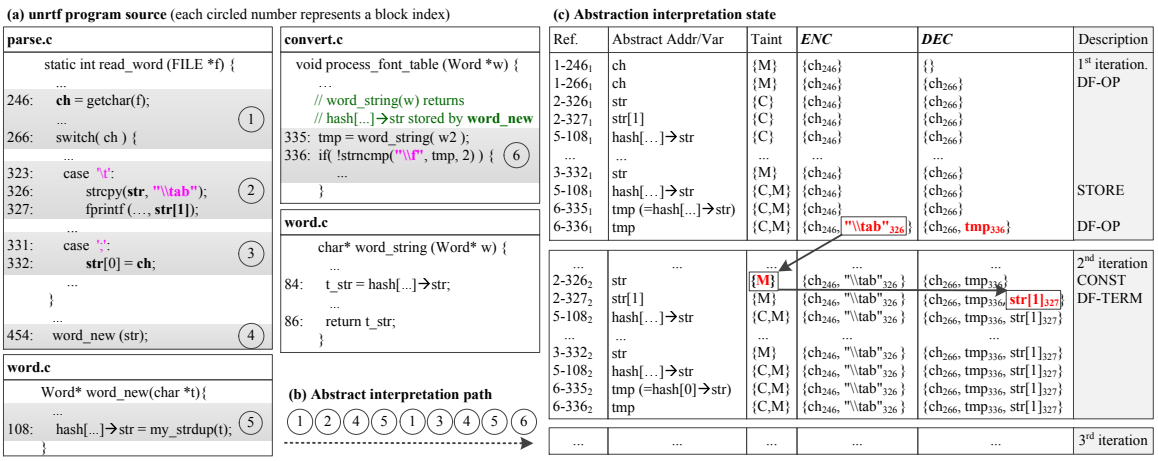


Figure 4.10.: An example of the iterative interpretation procedure on unrtf

memory locations pointed to by x_b are set to **MARKED** (Rule INPUT). Note that using the context C makes our analysis context sensitive. The encoding set is also updated. Rule NON-DF-OP describes the interpretation of an operation that is not on the decoding frontier, i.e., controllable operation such as copy. In this case, A2C copies the points-to set and the abstract taint set. Rule LOAD describes that for a load instruction, the resulting points-to/taint set is the union of all the points-to/taint sets of all abstract memory locations pointed-to by the address x_a. Similarly, for a store statement, the points-to/taint set of the value variable x_v is added to the points-to/taint set of any abstract memory location pointed to by x_a. A2C only propagates taints for controllable operations. Rules DF-OP handles an uncontrollable operation or a comparative operation. It first resets the taint. It then calls function **ChkSrc**(ℓ, x) that checks if variable x is tainted as **MARKED**. If so, the statement together with the current context and the variable are inserted to the decoding set *DEC*. The context and variable information is needed to indicate which variable should be decoded and under what context. The function further tests if the statement is already in *DEC* and the variable is currently tainted as **CONST**, suggesting that the statement sometimes uses a value from untrusted input and sometimes uses a constant. This corresponds to the case in which the decoding set is path sensitive. To eliminate such path sensitivity, A2C adds the

source of the constant to *ENC*, indicating that the source should be tainted as **MARKED** in the next round of abstraction interpretation.

Rule DF-TERM handles the other kind of operations in the decoding frontier: the terminal operations.

Rule CONST handles constant assignment, including constant string assignment. It tests if the constant assignment has been inserted to the *ENC* set (by Rules DF-OP or DF-TERM), indicating that the constant should be encoded so that we need to figure out its decoding places. In this case, it sets the taint as **MARKED**, otherwise **CONST**. Rule STRCAT handles string concatenations. When a string from an untrusted source is concatenated with a constant string, we add the constant string to the *ENC* set to indicate that the string shall be encoded. Such concatenation happens frequently when a program uses string formatting functions such as **sprintf**(). Rule CALL updates the current context. It further propagates the points-to and taint sets from the actual argument to the formal argument. At the end, it sets the points-to sets of all the local buffer variables to contain themselves. The RET rule pops the last entry in the context. The PHI rule specifies that since x takes the value of either x_1 or x_2, its abstract sets are the union of those of x_1 and x_2. A2C does not model path conditions so that it essentially considers all paths are feasible and computes merged results along various paths. Rule HEAP describes that we use the label of the allocation statement to denote the abstract heap region allocated. In addition, the σ and τ entries computed at a location are also propagated to its control flow successors. The rules are omitted as they are standard. The abstract interpretation is iterative until a fix point is reached. It is easy to infer that our analysis must terminate as all the abstract domains are finite.

Example. Fig. 4.10 shows how the analysis works for `unrtf` that reads an RTF file and transforms it to various formats. Fig. 4.10 (a) shows some code snippets of the program. The description of them can be found at the beginning of Section 4.4.3. The program is simplified and slightly changed from its original version for illustration.

The abstract interpretation procedure is equivalent to traversing the path in Fig. 4.10 (b). The real interpretation order inside A2C is slightly different due to the ϕ functions that are omitted for easy explanation, although the outcome is identical. In the path, the

two branches of the `switch` are traversed in two sub-paths: 1 2 4 5 and 1 3 4 5 . They insert strings to the hash table and the strings are later accessed at 6 .

Fig. 4.10 (c) shows the abstract states computed by A2C in multiple rounds. Each round follows the path in (b) during interpretation and corresponds to a sub-table in (c). The first column shows the block, line and round numbers of each statement. For instance, 2-326_1 means Line 326 inside 2 in the first round of interpretation. Here, we only show the statements related to our analysis. The next two columns present the abstract address or variable that each statement accesses and its taint set. C means the **CONST** type and M denotes the **MARKED** type. The next two columns show the contents of *ENC* and *DEC*. The last column presents the rules applied.

First Round. *ENC* and *DEC* sets are empty at the beginning. At $1-246_1$, since `ch` is loaded from an input source, we add $\texttt{ch}_{246}$ to *ENC* to indicate that we should encode `ch` at Line 246. Then, `ch` is used in a comparison at $1-266_1$, thus we add $\texttt{ch}_{266}$ to *DEC*, meaning that we should decode `ch` at Line 266. For simplicity, we ignore the contexts in the *DEC* set. At $2-326_1$, a constant string is copied to `str`, and part of it is printed at $2-327_1$. Since `str` has a constant taint at this point, it does not need to be decoded. Later it is stored into the hash table at $5-108_1$. Then, a character from a file is copied to `str` at $3-332_1$, and is then stored in the hash table at $5-108_1$. Since A2C cannot distinguish if the hash table write and the previous write access different (abstract) memory locations, it unions the two taints so that the hash table is tainted with both **CONST** and **MARKED**, according to Rule STORE.

Later, at $6-335_1$ and $6-336_1$, the stored string is loaded and compared with a constant string "`\\f`". According to Rule DF-OP, since Line 336 is comparative and `tmp` is tainted with **MARKED**, it shall be decoded. An entry is hence inserted to the *DEC* set. Also according to the second `if` statement inside `ChkSrc()`, which is invoked by Rule DF-OP, the constant string at Line 326 is added to *ENC*, meaning that the constant string shall be encoded.

Second and Third Rounds. The second round traverses the same path. At $2-326_2$, the constant string is **MARKED** as it is in *ENC*, meaning that we should track its propagation

to figure out the decoding places (Rule CONST). As a result, `str[1]` at Line 327 is added to *DEC* according to Rule DF-TERM. The rest is similar to the first round. In the third round, none of the abstract sets are updated, a fix point is reached. The analysis terminates.

From the final *ENC* and *DEC* sets, we should encode at Lines 246 and 326, and decode `ch`, `str[1]` and `tmp` at Lines 266, 327 and 336, respectively. □

4.4.4 Runtime

Supporting Context Sensitivity. Once the analysis phase is finished, we have the *DEC* and *ENC* sets. Since both *DEC* and *ENC* are context sensitive, meaning that decoding and encoding should be performed only under certain calling contexts, the instrumentation needs to compare at runtime if the current context matches with that in *DEC*/*ENC* in order to perform decoding/encoding.

A straightforward way to obtain the current context is to perform stack walking. However, it incurs significant overhead. Furthermore, the resulting contexts are verbose and difficult to compare. To address the problem, we adopt a precise calling context encoding algorithm [131]. The algorithm maintains an *id* which is a *unique* number for each context. Given a program and its call graph, the algorithm automatically determines a unique *id* for each context. It further instruments the program in such a way that the instrumentation (at call sites) guarantees to produce the corresponding *id* when a context is reached. The instrumentation only requires simple (and low-cost) additions and subtractions before and after a subset of call sites. Context comparison becomes simple id comparison. Since the encoding algorithm is not our contribution, details are elided.

Encoding Based on One-Time-Dictionary. Simple encodings such as subtract-by-one are easy for the adversary to reverse engineer. He/she can prepare the exploit accordingly so that the exploit inputs become the plain-text payloads after our encoding. To address the problem, we use one-time-cipher. In particular, A2C has a large number of pre-generated random one-to-one mappings that project a byte to another unique byte. Whenever the program reads inputs from an untrusted source, A2C selects a mapping to encode the buffer.

Since the dictionary for each untrusted input buffer is different from others, knowing previous mappings (e.g., through memory disclosure) does not help in launching subsequent attacks. More discussion can be found in Section 4.5. Another thing we want to point out is that A2C mutates every byte from an untrusted sources. As such, none of the instructions from the original payload can be properly executed.

Using different dictionaries for different buffers requires A2C to track the dictionaries for individual buffers so that decoding can be properly performed. This is achieved by adding runtime taint propagation logic for controllable operations in the exploitable space. For controllable operations that are not simple copies (e.g., $y = 3 * x$), A2C decodes the source operand(s), performs the operation, and encodes the resulting operand using the same mapping. Since the exploitable space is very small, the entailed runtime overhead is low (see Section 4.6).

4.5 Threat Model

A2C assumes the subject program is benign but the inputs may be malicious. The user specifies which part of the inputs cannot be trusted such as network inputs and/or local file reads. It trusts the kernel. It also trusts that the low level *output* libraries are free of vulnerabilities, as it decodes the buffer values before calling these libraries. If they cannot be trusted, we can mitigate the problem by postponing the decoding to before output syscalls, which requires instrumenting libraries. Note that we do not trust all library functions. For example, we do not decode inputs for functions that copy data such as `strcpy` and `memcpy`. In practice, such functions are commonly exploited by attackers whereas output library functions such as `write` and `send` are not.

A2C aims to protect against payload injection attacks. It cannot handle other attacks that do not inject payload. It also requires the payload injection go through explicit input channels, which is true for most attacks. A2C currently only supports C/C++ programs and hence cannot deal with payload injections for programs in other languages such as JavaScript, although the idea is general.

Attacks In the Post-exploitable Space. A2C leverages constraint solving and a large pool of payload test cases that models the distribution of valid payloads to determine the decoding frontier with strong probabilistic guarantees. However, it may still be possible to construct some payloads via the very limited controllability of those uncontrollable operations on the decoding frontier. We argue that such payloads will have very limited functionalities. Moreover, we only protect against payloads that are larger or equal to 16 bytes. While it may be possible to construct payloads smaller than that, we again argue that such payloads will have very limited functionalities. Note that if a primitive value of four bytes is related to input, the attacker could inject a four byte payload to that primitive if there existed one. Protecting against such small payloads is almost impossible and unnecessary. In practice, we have not seen any examples of these payloads.

Memory Disclosure. Memory disclosure vulnerabilities can reveal memory contents of a process. Attackers can access memory pages that contain the encoded values and thus reverse engineer dictionaries. For example, he/she can manipulate the input by providing a sequence of unique values and then search in the disclosed memory for regions that have a sequence of unique values of the same length. By contrasting the two, the dictionary can be revealed. However, since A2C uses different dictionaries for individual input buffers, disclosing previous dictionaries does not help in subsequent attacks. Since A2C uses a random dictionary each time, it is really difficult to guess the next dictionary even knowing the previous dictionaries (i.e., 1 out N with N the number of pre-generated dictionaries). We use $N = 10^6$ in this chapter.

4.6 Evaluation

A2C is implemented on LLVM [132]. We evaluate A2C on 18 different real world programs shown in Table 4.2. All the experiments were done on a machine with Intel Core i7 3.4GHz, 8GB RAM, and 32-bit LinuxMint 17.

We searched `exploit-db.com` to choose target programs. We tried the listed programs with reported exploits and selected those which we could reproduce. We have 6 network

Table 4.2.: Evaluation Results for Analysis

Program	Size	$\|ENC\|$	$\|DEC\|$	CS[1]	CCE[2]	Analysis Time	
						DF Comp.[3]	SA[4]
mupdf	483K	598	2283	241	172	1h 5m	12m 11s
prozilla	54K	98	754	391	104	9m 49s	2m 43s
stftp	18K	42	144	42	37	6m 51s	1m 58s
yops	9,215	49	153	4	12	24s	13s
nginx	335K	151	1005	37	72	34m 14s	17m 22s
ngircd	119K	123	391	113	249	7m 39s	10m 1s
unrar	99K	36	239	44	164	17m 21s	7m 11s
mcrypt	36K	83	278	40	35	12m 41s	4m 20s
gif2png	16K	32	129	28	22	8m 19s	1m 38s
mp3info	17K	33	91	23	19	6m 9s	2m 17s
fcrackzip	48K	18	37	23	11	8m 17s	2m 58s
chemtool	176K	100	388	27	39	20m 35s	7m 41s
vfu	180K	64	129	49	318	12m 51s	8m 21s
unrtf	25K	31	220	291	178	14m 5s	2m 43s
rarcrack	1,364	7	19	39	9	0s	5s
make	124K	106	719	125	94	31m 14s	1h 40m
Xerces-C	415K	121	1137	102	213	1h 28m	6h 21m
apache	208K	364	1586	98	63	1h 56m	5h 41m

[1]# of Context Sensitive Statements.
[2]# of instrumentations for Calling Context Encoding.
[3]Decoding Frontier Computation Phase. [4]Static Analysis Phase

Table 4.3.: Evaluation Results for Attack Prevention

Program	# of Inputs (Mal./Benign)	# of Vulnerabilities	# of Payloads (Shellcode/ROP)	# of Crashes (Mal./Benign)	# of ins. exec. in Payloads	# of ROP Gadgets Exec. in Payloads	Precision/Recall
mupdf	10 / 20	1 (CVE-2014-2013)	50 / 50	1000 / 0	3.62	0.1	100% / 100%
mcrypt	10 / 20	2[1]	50 / 50	1000 / 0	3.62	0.18	100% / 100%
sftp	10 / 20	1 (EDB-ID: 9264)	50 / 50	1000 / 0	3.6	0.08	100% / 100%
yops	10 / 20	1 (EDB-ID: 14976)	50 / 50	1000 / 0	3.62	0.05	100% / 100%
nginx	10 / 20	1 (CVE-2013-2028)*	50 / 50	1000 / 0	3.62	0.09	100% / 100%
ngircd	10 / 20	2[2]	50 / 50	1000 / 0	3.62	0.11	100% / 100%
unrar	10 / 20	1 (EDB-ID: 17611)	50 / 50	1000 / 0	3.62	0.18	100% / 100%
prozilla	10 / 20	2[3]	50 / 50	1000 / 0	3.6	0.09	100% / 100%
gif2png	10 / 20	1 (CVE-2009-5018)	50 / 50	1000 / 0	3.62	0.09	100% / 100%
mp3info	10 / 20	1 (CVE-2006-2465)	50 / 50	1000 / 0	3.62	0.05	100% / 100%
fcrackzip	10 / 20	1 (EDB-ID: 14904)	50 / 50	1000 / 0	3.62	0.05	100% / 100%
chemtool	10 / 20	1 (EDB-ID: 36024)	50 / 50	1000 / 0	3.6	0.18	100% / 100%
vfu	10 / 20	1 (EDB-ID: 35450)	50 / 50	1000 / 0	3.61	0.18	100% / 100%
unrtf	10 / 20	1 (CVE-2004-1297)	50 / 50	1000 / 0	3.62	0.18	100% / 100%
rarcrack	10 / 20	2[4]	50 / 50	1000 / 0	3.62	0.05	100% / 100%
make	10 / 20	1 (EDB-ID: 34164)	50 / 50	1000 / 0	3.62	0.18	100% / 100%
Xerces-C	10 / 20	1 (CVE-2015-0252)	50 / 50	1000 / 0	3.62	0.07	100% / 100%
apache#	10 / 20	2[5]	50 / 50	1000 / 0	3.6	0.13	100% / 100%

[1](CVE: 2012-4409, 2012-4527) [2](CVE: 2005-0226, 2005-0199) [3](CVE: 2005-0523, 2004-1120)
[4](EDB-ID: 15062, 15054) [5](CVE: 2004-0940, 2006-3747) *This CVE includes multiple vulnerabilities #Version 1.3.31

programs, with two client programs: `prozilla` and `stftp`, and four server programs: `apache`, `nginx`, `yops`, and `ngircd`. We have 12 user applications. `mupdf` reads and displays pdf documents. `unrar` is a decompressor program. `mcrypt` encrypts and decrypts files. `gif2png` converts gif to png. `unrtf` converts RTF files to other formats such as

HTML. `mp3info` reads and modifies meta tags of MP3 files. `rarcrack` and `fcrackzip` recover passwords of compressed files (e.g., `zip` and `rar` files) using different strategies. `vfu` is a text-mode file manager. `chemtool` is a GUI program for drawing chemical structures. `Xerces-C` is an XML parser. Among these programs, we have two GUI programs that require user interactions: `mupdf`, and `chemtool`. `vfu` requires text-based user interactions.

The first two columns of Table 4.2 show the programs and their size in C source code lines (CLOC). The third and fourth columns present the number of entries in *DEC* and *ENC* computed by our analysis. They are essentially LLVM IR statements annotated with contexts. The fifth column shows the number of statements in *DEC* that behave differently depending on the context. One such statement has multiple entries in the *DEC* set (for different contexts). The sixth column represents the number of instrumented IR statements for calling context encoding. The last two columns show the time spent on computing the decoding frontier, and the static analysis for DEC/ENC set computation and instrumentation, respectively. The overhead of decoding frontier computation includes the running time of Z3 constraint solver. We use one minute as the timeout threshold. We also avoid testing identical payload sequences.

From the table, we have the following observations. A2C can handle large and complex programs such as `mupdf` and `apache`. The number of entries in *ENC*/*DEC* is small with respect to the program size. This supports our speculation that the exploitable space is small. The data in the fifth column also supports that context sensitivity is needed. Finally, the analysis overhead is acceptable. Some large programs take a few hours. However, we argue that this is one-time cost.

4.6.1 Performance

Performance for Programs with Vulnerabilities (i.e., those in Table 4.2). To evaluate the runtime overhead of A2C, we run both the original program and the instrumented version 10 times and take the average. We use large inputs. For example, we use document files

that are larger than 10MB to test file processing programs `unrtf`, `Xerces-C`, and `gif2png`. As such, the native executions usually last for more than a few seconds. For the programs that require user interactions, we force them to quit after they load, process, and render the inputs, and before they take any user interactions. We manually identify the locations in the source files that indicate such status (e.g., before calling a function to change the status bar to show the input is successfully loaded and rendered) and insert `exit()` to these locations. We then measure the overhead for these shortened executions. Note that, this usually leads to over-approximation of the overhead as our instrumentation largely lies in the initial input loading and parsing logic.

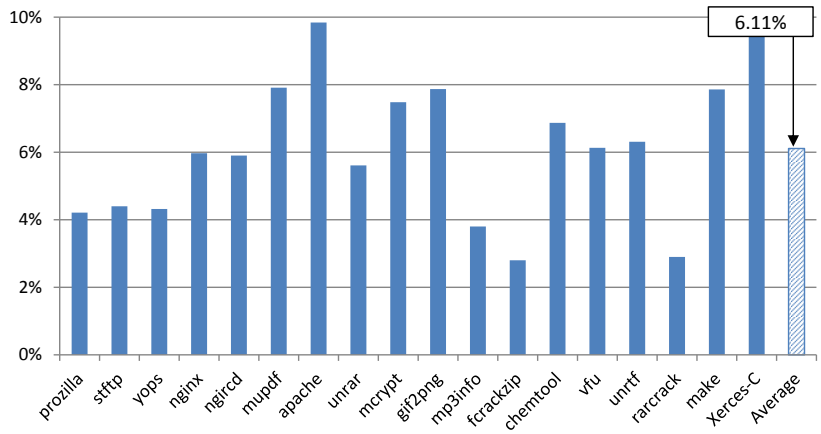


Figure 4.11.: Normalized overhead on programs in Table 4.2

Fig. 4.11 shows the result. The average overhead is 6.11%. In most cases, the overhead is less then 6%. There are a few exceptions. Programs dedicated to processing and parsing input files such as `make`, `Xerces-C`, `unrtf`, and `gif2png` have relatively higher overhead. This is because the instrumented statements are being executed throughout the execution. Also, the programs that require interactions, e.g., `mupdf`, `chemtool`, and `vfu`, have relatively higher overhead. This is because of the way we measure the overhead. `apache` has the highest overhead (9.84%) due to the complex structure of input filters that leads to many constant strings being encoded.

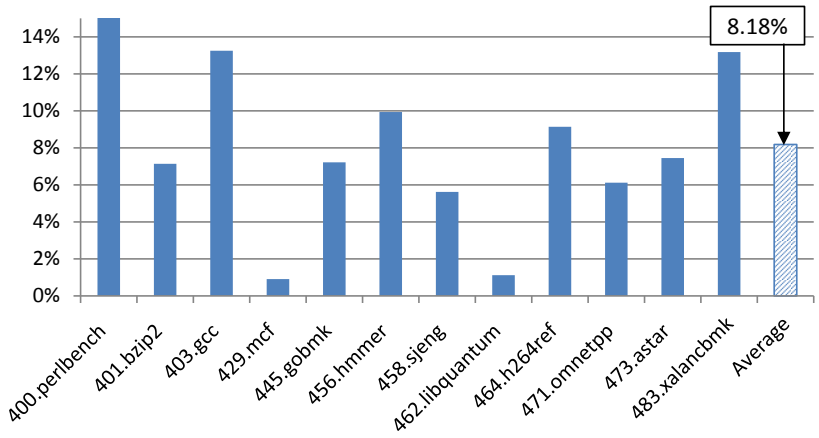


Figure 4.12.: Normalized overhead on SPEC CPU2006 programs

SPEC CPU2006. We also evaluate the performance of A2C on SPEC CPU2006. We run both the original and instrumented programs 10 times using the reference inputs. Fig. 4.12 shows the result. The average overhead is 8.18%. `401.perlbench`, `403.gcc`, and `483.xalancbmk` have relatively higher overhead because they process inputs intensively. `456.hmmer` has 9.94% overhead as it processes inputs even during the execution of its main algorithm. `429.mcf` and `462.libquantum` have extremely low overhead, less than 1.5%. This is because they process inputs once at the very beginning. As such, A2C only needs to decode at the beginning and the rest of the execution does not cause any overhead. The average overhead for all 30 programs including programs in Table 4.2 and SPEC CPU2006 is 6.94% and the geometric mean is 5.94%.

4.6.2 Effectiveness

To evaluate the effectiveness of A2C in preventing attacks and allowing benign executions, for each program, we prepare 10 exploits and 20 other benign inputs. For each exploit input, we prepare 100 different malicious payloads, including 50 shellcodes and 50 ROP payloads.

The shellcodes are generated from [127], and we use ROP attack creators [128, 129] to generate 50 different ROP payloads for each vulnerable application. Thus, we have 1,000 attack executions and 20 benign executions for each program. Note that, as shown in Table 4.3 Column 3, some programs have more than one vulnerability, which require unique exploit inputs. The table also shows the results. Observe in the fifth column, A2C successfully crashes all the attacks and allows all the benign inputs to proceed to normal termination and produce the expected outcomes. The next two columns show the average number of payload/gadget instructions that got executed before crashing. They are all in very small numbers. As such, they can hardly cause any damage to the system.

Decoding Frontier (DF) Operation Classification. We further analyze the DF operations for all the subject programs and classify them into a few categories. Fig. 4.13 shows the results, from which we have the following observations.

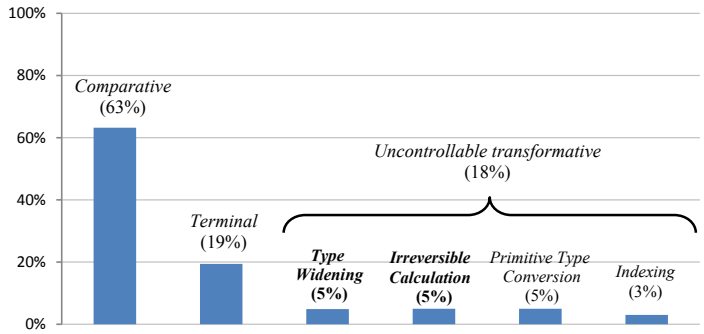


Figure 4.13.: Different types of decoding frontiers

First, 63% operations on DFs are *Comparative Operations*. Note that comparative operations are mostly conducted on individual buffer elements (of primitive types), A2C only decodes the element needed by the operation. The decoded value is dead (e.g., overwritten) right after the operation. Such DF operations cannot be exploited. Second, 19% DF operations are *Terminal Operations*. For a terminal operation, A2C first copies the original buffer to a temporary buffer, and then decodes the temporary buffer. Also, after the terminal operation, A2C releases the temporary buffer to minimize the attack window. Third, we also identify a few kinds of *Uncontrollable Transformative Operations*. In particular,

Type Widening expands each element in a buffer by padding it with some specific byte(s) such as `0x00`. Note that we use the constraint solver to determine whether each case of type widening is controllable as not all type widening cases are uncontrollable. In fact, casting a one-byte data type to a two-byte data type is solvable in many cases. Note that some binary operations (e.g., multiplication) of values with smaller types yield a value of a large type. These are not type-widening as the bits in the resulting value are often fully/largely controllable. *Irreversible Calculation* means arithmetic transformations that cause intensive correlations among values so that the solver returns UNSAT for all tests. An example can be found in Section 4.4.1. *Primitive type conversion* means that a buffer element is converted to a value of primitive type (e.g., `atof()`) and this value is not stored to any array/buffer. Since single primitive values can hardly be exploited to inject payloads due to the size, decoding is safe. Note that A2C protects consecutive primitive values if they can form a region larger than 16 bytes. *Indexing* means that an encoded value is used to index a non-constant array. It is safe to decode the value because the decoded value is of a primitive type and soon dies after the operation. The entire buffer is never decoded.

Decoding Frontier (DF) Computation. Table. 4.4 shows the evaluation results of decoding frontier computation. The first column shows the programs. The next three columns show the numbers of controllable operations, uncontrollable operations, and their sum, respectively. The last column shows the average number of constraints for each memory region under test. Recall that if the solver returns SAT, TIMEOUT or UNKNOWN for a constraint in any payload sequence test, the corresponding operations are considered controllable.

We make the following observations. First, in most cases, there are more UNSAT cases than SAT cases. This means that most input related computations are not controllable. There are a few exceptions. `gif2png`, `apache`, and `chemtool` have more SAT cases as our modeling of the external library calls is not complete and the modeling of floating point functions is conservative. For example, we assume `exp()` function can return any positive floating point values while the parameter of the `exp()` function may have constraints, hence it may not be able to produce some floating point values. Note that such a conser-

Table 4.4.: Results for Decoding Frontier Computation

Program	# of Operations			Avg. # of Constraints
	Controllable	Uncontrollable	Total	
mupdf	9	141	150	16.4
Prozilla	4	20	24	15.9
stftp	2	8	10	11.5
yops	0	1	1	8
nginx	4	41	45	17.2
ngircd	2	12	14	14.1
unrar	6	33	39	14.2
mcrypt	4	24	28	18.3
gif2png	13	10	23	16.9
mp3info	4	9	13	15.3
fcrackzip	4	4	8	13.6
chemtool	29	22	51	14.1
vfu	3	25	28	15.5
unrtf	2	22	24	14.5
rarcrack	0	0	0	0
make	9	53	62	15.4
Xerces-C	14	75	89	14.8
apache	145	129	274	17.7
Average	**14.1**	**34.9**	**49**	**14.05**

vative assumption only causes over-approximation. Second, the total number of operations for testing is not large (`apache` has the largest number 274). This is because the controllability classification for most operations is straightforward (e.g., comparative operations and copy operations) and hence does not require constraint solving. Third, the average number of constraints in our tests is not large, suggesting that controllable operations are often shallow in the data flow, meaning that they are close to program inputs. This supports our assumption that most computation happens in the post-exploitable space. Note that we do not need to test controllability of operations if their operands are not controllable.

4.6.3 Case Studies

Running Web Servers on Real-world Traffic. To further evaluate the robustness of A2C, we run the instrumented web servers on a real-world traffic log. We obtained our institu-

tion's server access log from November 2015 to January 2016. The log contains 5.6 million requests with 4.2 million unique requests, including some suspicious requests with binary payloads (about 100 of them). We also randomly inject 300 exploit inputs to the access log. We ran three servers (`apache`, `nginx`, and `yops`) with these requests. The results show that the instrumented versions produce the same expected results as the original versions except for the attacks. All attacks are prevented. The throughput is only reduced by 8.83%, 7.37%, and 5.49%, respectively.

Code Injection Through Benign Functions and Payload Triggered Through Integer Overflow. In this case study, we show how a payload can be injected through benign and non-vulnerable program logic and later triggered by an integer overflow vulnerability. Such a combination makes it difficult for traditional defense techniques. Fig. 4.14 shows code snippets of the victim program, `mupdf`. First, observe that the `xps_read_dir_part()` function reads a file. It opens a file at Line 455, then gets the size of file at Line 458. Later, it reads the file and puts it into a heap buffer (`part->data`) at Line 462. Note that the function `xps_read_dir_part()` is not vulnerable. But still, the attacker can provide a crafted `xps` file that contains a malicious payload. The payload will be injected through the normal file read in the benign function. Thus, most existing protection schemes including CFI, DFI, ASLR, and boundary checkers cannot prevent such injection. While malicious payload detection methods can identify the injected shellcode by scanning the input file at the `fread` function, the attacker can use obfuscation techniques to circumvent such detection.

To trigger the payload, the attacker exploits an integer overflow vulnerability. The integer overflow happens as follows. It reads input from a file at Line 91 in `lex_number()`. Then the input is propagated to Line 97 where the integer overflow occurs. The program assumes the input `c` is between '0' to '9', and converts it into an index (`i`). At Line 106, the converted index is stored into `buf->i`. Later, the index is used to write elements into a structure (at Lines 176-178 in `pdf_repair_obj_stm()`). Note that the earlier index is propagated to variable `n` which is also used as an index. This integer overflow can be leveraged to overwrite some critical data fields such as function pointers in order to change

control flow of the program to the injected shellcode. Note that the exploit may not be detected by address sanitizers as the attacker can manipulate the offset `n` to directly overwrite the target memory addresses that may fall into other legitimate memory regions, without overwriting the canaries.

In contrast, A2C defeats the attack by breaking its weakest link, which is the injected payload itself. In particular, A2C mutates the input including the shellcode at the `fread` in Line 462. The original shellcode is shown in Fig. 4.14 (a), and the corresponding mutated shellcode in Fig. 4.14 (b). Observe that the mutated shellcode is broken and not executable.

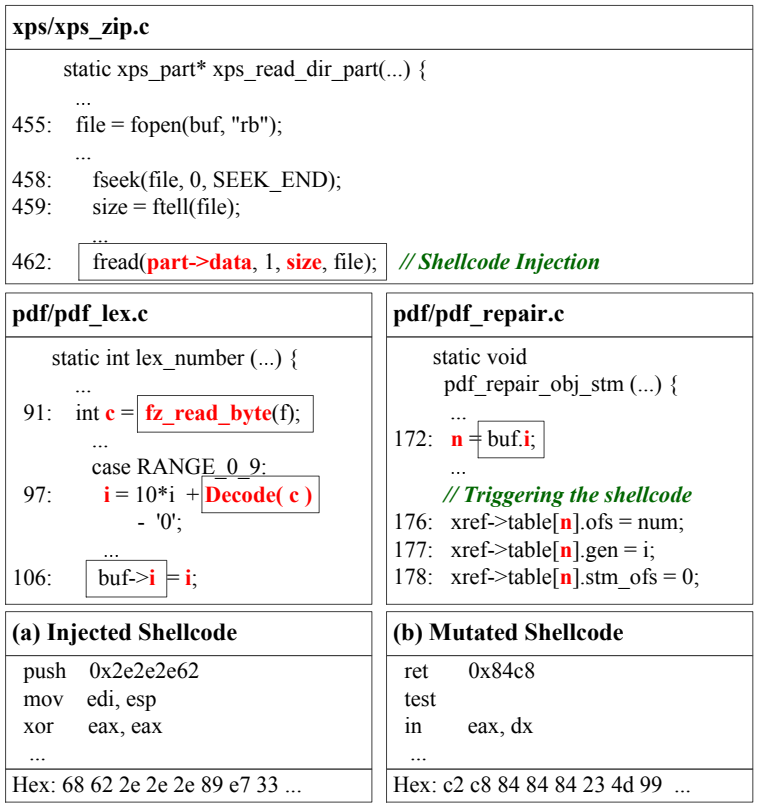


Figure 4.14.: Integer overflow in mupdf

Note that A2C does not prevent the integer overflow. Even through it encodes the input value at Line 91, it decodes the value right before the overflow (at Line 97) because that is an operation of primitive type. In other words, the attacker can still exploit integer overflow

vulnerabilities. However, when the control flow of the program is redirected to the injected shellcode, the execution crashes almost immediately as the first instruction of the mutated shellcode is "`ret 0x84c8`", which does not have a valid return address.

One might think the attacker can exploit the integer overflow to direct the control flow to some buffer in the post-exploitable space. However, as we pointed out in Section 4.5, the transformations performed by the subject programs are complex enough that the attackers cannot generate plain-text payloads in the post-exploitable space.

Preventing ROP attacks. As DEP (Data Execution Prevention) becomes more and more popular, attackers now use ROP to bypass such protection. In this case study, we show how A2C prevents ROP attacks using an example.

convert.c

```
      void process_font_table (...) {
        ...
331:  char name[255];
        ...
341:  while (w2) {
342:    tmp = word_string(w2);
343:    if ( tmp &&
              Decode( tmp[0] ) != '\\' )
344:      strcat( name, tmp );
```

(a) Injected ROP gadgets

Address	Instructions
0x804d820	mov ebx,0x0; ret
0x804ec7d	mov eax,0x806275c; ret
...	...

(b) Mutated ROP gadgets

Address	Instructions
0xa2ae728a	Invalid address
0xa2ae46d7	Invalid address
...	...

Figure 4.15.: Stack buffer overflow in unrtf

Fig. 4.15 shows `unrtf` which has a stack buffer overflow vulnerability. It can be leveraged to inject a malicious payload that allows constructing a ROP gadget chain. The program first gets a user provided string at Line 342. Then, it compares the string with a constant at Line 343. As it is a comparative operation, A2C decodes the value, allowing proper comparison. The buffer overflow happens when the program copies the user provided buffer (`tmp`) to a local buffer `name` at Line 344 in `process_font_table()`. Observe that the size of `name` is only 255. Thus, providing a long enough input to the `tmp` buffer will result in a stack overflow.

Fig. 4.15 (a) shows the injected ROP payload and the corresponding gadgets. The address column shows the payload that contains the raw addresses of the ROP gadgets. The instructions column shows the instructions from the ROP gadgets. Observe that they all end with a `ret` instruction. These chains of instructions are essentially the ones that get executed once the attack is launched. Fig. 4.15 (b) shows the mutated payload. For demonstration purpose, we use a simple encoding/decoding scheme even though our implementation uses one-time-dictionary. In particular, the mutation is to `xor` a value with `0xAA`. Observe that all the addresses in the original payload are encoded and point to invalid addresses. Hence, the attack fails. Note that since A2C prevents attacks by mutating payloads, the injection methods do not affect our protection.

Preventing English Shellcode. As a counter attack to shellcode detection techniques, Mason et al. proposed an automatic way to generate shellcode which is similar to English prose [115]. Such technique can be used to avoid existing shellcode identification techniques [133–136].

English Shellcode and Mutated English Shellcode		
Assembly	Opcode	ASCII
push esp push 0x20657265 ...	54 68 65 72 65 20 ...	There is a majorcenter of economic activity, ...
inc dl **iret** ...	fe c2 cf ...	No ASCII character found

Figure 4.16.: English shellcode example

Fig. 4.16 shows an example of English Shellcode presented in [115]. As shown in the ASCII column, the shellcode is an English statement. The corresponding assembly instructions are listed in the first column. While we are just showing one example, in practice attackers also use other various shellcode obfuscation and compression techniques [137, 138] to avoid shellcode identification. A2C mutates all untrusted inputs including shellcodes as they are part of the inputs. The mutated English Shellcode includes those shaded in

Fig. 4.16. For demonstration, we again apply the `xor` with `0xAA` mutation. Observe that the mutated shellcode is completely different from the original shellcode. While the first instruction is executable, it does not help attackers to achieve anything useful. More importantly, the second instruction is `iret`, which can only be executed in a kernel mode. Executing `iret` results in a segmentation fault. One interesting observation is that the first a few instructions in the mutated shellcode are often executable. The fifth column of Table 4.3 shows the average number of instructions executed in the mutated payload is very small (<4). It is also important to note that such a few (mutated) instructions do not have the same semantics as the original malicious logic. They often immediately lead to crashes and do not cause any damage to the system.

Buffer Overflow In Structure. AddressSanitizer [139] is an important technique to prevent various buffer overflow attacks including heap and stack overflows. It works by placing canaries before and after a buffer. One of the limitations of the technique is that it cannot handle buffer overruns within a structure.

Program.c

```
    void process(RECORD* p) {
1:    fread( p->name, ... );
2:    printf("Name: %s\n",
            Decode( p->name ));
3:    p->handler( p->privilege );
```

Program.h

```
typedef struct tag_RECORD {
   char name[255];
   void (*handler)(int);
   int privilege;
} RECORD;
```

Figure 4.17.: Buffer overrun in structure

Fig. 4.17 shows a buffer overflow vulnerability in a structure. Specifically, buffer `name` in the structure `RECORD` can affect adjacent data fields including a function pointer `handler`. At Line 1, it reads a file to fill the `name` buffer. By providing an input string longer than 255 bytes, it can overwrite `handler`. Note that A2C mutates the input in `fread` at Line 1, the `handler` is overwritten with a mutated address. Then, the program calls `printf` to display the name on the screen. As `printf` is an external call, A2C decodes the input buffer `name`. Specifically, in our implementation of the decoding function,

when A2C decodes a buffer for a library call, it allocates a new buffer, copies the original encoded buffer, and then decodes it in the new buffer before passing it. Since A2C does not decode the original buffer, the injected malicious payload remains mutated. At Line 3, the program calls `handler`. Although it is overwritten, the function pointer no longer points to the injected shellcode. Note that the `privilege` field can also be overwritten to launch non-control data attacks [140]. A2C mitigates the attacks by encoding the inputs from untrusted sources. As a result, the attacker cannot control the overwritten value.

4.7 Related Work

Control-flow Integrity (CFI). Recent advances in control-flow integrity have developed very robust systems for preventing malicious/abnormal control flows within a victim program. These typically monitor execution to enforce pre-determined control flow paths [117–120,141–146]. In contrast, A2C provides protection by corrupting input payloads, which is a perspective orthogonal to the enforcement of a program's legitimate control flow graph. Therefore, A2C is complementary to and can be deployed alongside CFI, e.g., to prevent exploit injection attacks that may employ indirect calls or not violate control flow integrity [146–153].

Malicious Payloads Detection. In [133] and [134], researchers proposed analyzing inputs to detect malicious payloads with little runtime overhead. However, Fogla at el. [154] demonstrated that polymorphism techniques can defeat these approaches. Dynamic analysis using emulation [155, 156] have been proposed to uncover polymorphic payload injection attacks, but they cause non-negligible performance penalty. A2C mutates all input buffers from untrusted sources and thus is resilient to polymorphism. It does not require emulation and causes low overhead. Nozzle [157] proposed a novel technique to detect heap spraying attacks at runtime. It uses runtime interpretation and static analysis to analyze suspicious objects in the heap. While Nozzle focuses on detecting heap spraying on JavaScript, A2C takes a more general approach to prevent a wider range of input injection attacks.

Randomization Approaches. Address space layout randomization (ASLR) is one of the most widely deployed defense mechanism to mitigate payload injection and triggering. ASLR randomizes the memory layout of a program when the OS loads the binary and dynamic libraries. ASLR is already a default defense mechanism in most operating systems including Linux, MacOS, BSD, and Windows. Address space layout perturbation [158] and fine-grained randomization techniques [159–164] have been developed to provide higher entropy. Instruction set randomization [122, 165, 166] aims to change the underlying instruction set to prevent executing injected code. However, it was shown recently that randomization could be evaded by brute-force attacks [108, 167], memory disclosure attacks [168–170], and just-in-time code reuse attacks [171]. In [172], researchers presented a novel defense technique to mitigate counterfeit object-oriented programming (COOP) attacks [151]. They randomize the layout of the code pointer table and plant booby-traps to prevent brute-force attacks. Compared to these techniques, A2C provides protection by working from the input perspective, which is complementary to randomization. Data randomization [121, 173] dynamically decrypts a buffer upon each buffer access and encrypts it again after the access. It encrypts all buffers including those not related to inputs. It also uses different keys for various buffers. A2C shares a similar idea of buffer encoding with data randomization. The differences lie in that A2C focuses on input related buffers; it encodes only once for each input and decodes only at the decoding frontier. As such, A2C has relatively lower overhead. PointGuard [174] encrypts pointer values at runtime.

Bounds Checking. Stackguard [175] inserts a secret value (canary) before each return address and frame pointer. However, it can be defeated through information leak attacks that reveal a canary value [176, 177]. Compile-time code analysis [178, 179] have been proposed to detect unsafe array and pointer accesses. However, they often generate many false positives and focus on specific kinds of vulnerabilities. Cling [180] and AddressSanitizer [139] provide pointer safety to prevent exploiting pointer related bugs such as use-after-free. However, as shown in our case study, they can hardly handle advanced attacks [181]. In contrast, A2C aims to break the weakest link of attacks, which is the payload itself.

5 CONCLUSION

As cyber-attacks are becoming more and more persistent and sophisticated, investigating and preventing advanced cyber-attacks such as APTs is of the utmost importance. In this dissertation, we present three fundamental primitives for the investigation and prevention of advanced cyber-attacks. Specifically, we adopt the original concept of counterfactual causality in the context of program and program execution in order to precisely infer causality between system call events. Moreover, we proposed a model-based causality inference technique that can precisely infer causality without any modification on end-user systems. Finally, we develop a novel attack prevention technique which can prevent unknown zero-day exploits by perturbing inputs. In other words, we showed that accurate attack investigation and general protection against advanced and sophisticated attacks can be achieved by leveraging causality inference and fundamental weaknesses of the attacks.

In particular, 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.

Second, we propose MCI, a novel causality inference algorithm that directly works on audit logs provided from commodity systems. MCI does not require any special efforts (e.g., training, instrumentation, code annotation) or framework (e.g., enhanced logging, taint tracking) on the end-user. Our off-line analysis precisely infers causality from a given system call log by constructing causal models and identifying the models in a given audit log. We implemented a prototype of MCI and our evaluation results show that MCI is scalable to cope with large scale log from long-running applications. We also demonstrate that MCI can precisely identify causal relations in realistic attack scenarios.

Finally, we describe A2C that provides general protection against a wide spectrum of payload injection attacks. It mutates all input buffers from untrusted sources to break malicious payloads. To assure the program functions correctly on legitimate inputs, it decodes them right before they are used to produce new values. A2C automatically identifies such places at which it needs to decode using a novel constraint solving based approach and a sophisticated static analysis. Our experiments on a set of real-world programs show that A2C effectively prevents known payload injection attacks on these programs with reasonably low overhead (6.94%).

REFERENCES

REFERENCES

[1] Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. LDX: Causality inference by lightweight dual execution. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 503–515, New York, NY, USA, 2016. ACM.

[2] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela Ciocarlie, Ashish Gehani, and Vinod Yegneswaran. MCI: Modeling-based causality inference in audit logging for attack investigation. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium, NDSS '18, San Diego, California, USA, February 18-21, 2018*. The Internet Society.

[3] Yonghwi Kwon, Brendan Saltaformaggio, I Luk Kim, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. A2C: Self destructing exploit executions via input perturbation. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium, NDSS '17, San Diego, California, USA, February 26-March 1, 2017*. The Internet Society.

[4] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium, NDSS '13, San Diego, California, USA, February 24-27, 2013*. The Internet Society.

[5] David Hume. *Enquiry Concerning Human Understanding*. Clarendon Press, 1904.

[6] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng 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.

[7] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek 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.

[8] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos 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.

[9] James Clause, Wanchun Li, and Alessandro 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.

[10] Erik Bosman, Asia Slowinska, and Herbert 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.

[11] Stephen McCamant and Michael 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.

[12] Min G. Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Ong. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium, NDSS '11, San Diego, California, USA, February 6-9, 2011*, Washington, DC, USA. The Internet Society.

[13] Landon P. Cox, Peter Gilbert, Geoffrey Lawler, Valentin Pistol, Ali Razeen, Bi Wu, and Sai Cheemalapati. SpanDex: Secure password tracking for Android. In *Proceedings of the 23rd Conference on USENIX Security Symposium*, USENIX-SS '14, pages 481–494, San Diego, CA, August 2014. USENIX Association.

[14] D. Lewis. Counterfactuals. *Oxford: Blackwell*, 1973.

[15] Bin Xin, William N. Sumner, and Xiangyu 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.

[16] Dohyeong Kim, Yonghwi Kwon, William N. Sumner, Xiangyu Zhang, and Dongyan Xu. Dual execution for on the fly fine grained execution comparison. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 325–338, New York, NY, USA, 2015. ACM.

[17] Wei Ming Khoo. Taintgrind. `https://github.com/wmkhoo/taintgrind`, 2017.

[18] Aydan R. Yumerefendi, Benjamin Mickle, and Landon 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.

[19] David Hume. *Enquiry Concerning Human Understanding*. Clarendon Press, 1904.

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

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

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

[23] Tao Bao, Yunhui Zheng, Zhiqiang Lin, Xiangyu Zhang, and Dongyan 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.

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

[25] Kenneth P. Birman. Replication and fault-tolerance in the ISIS system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, SOSP '85, pages 79–86, New York, NY, USA, 1985. ACM.

[26] M. Chereque, D. Powell, P. Reynier, J.-L. Richier, and J. Voiron. Active replication in Delta-4. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, FTCS '92, pages 28–37, July 1992.

[27] A Tulley and S.K. Shrivastava. Preventing state divergence in replicated distributed programs. In *Proceedings of the 9th Symposium on Reliable Distributed Systems*, SRDS '90, pages 104–113, Oct 1990.

[28] Dave Black, C. Low, and Santosh K. Shrivastava. The Voltan application programming environment for fail-silent processes. *Distributed Systems Engineering*, 5(2):66–77, 1998.

[29] Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions on Computing Systems*, 21(3):236–269, August 2003.

[30] Emery D. Berger and Benjamin 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.

[31] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating Byzantine-faults in transaction processing systems using commit barrier scheduling. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 59–72, New York, NY, USA, 2007. ACM.

[32] Byung-Gon Chun, Petros Maniatis, and Scott Shenker. Diverse replication for single-machine Byzantine-fault tolerance. In *Proceedings of the 2008 USENIX Annual Technical Conference*, ATC '08, pages 287–292, Berkeley, CA, USA, 2008. USENIX Association.

[33] Petr Hosek and Cristian Cadar. VARAN the unbelievable: An efficient N-version execution framework. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 339–353, New York, NY, USA, 2015. ACM.

[34] Liming Chen and A Avizienis. N-version programming: A fault-tolerance approach to rellablllty of software operatlon. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, FTCS '95, pages 113–122, Jun 1995.

[35] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *Proceedings of the 15th Conference on USENIX Security Symposium*, USENIX-SS '06, Berkeley, CA, USA, 2006. USENIX Association.

[36] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. Diversified process replic for defeating memory error exploits. *Performance, Computing, and Communications Conference, 2002. 21st IEEE International*, 0:434–441, 2007.

[37] Vitaliy B. Lvin, Gene Novark, Emery D. Berger, and Benjamin 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 '08, pages 115–124, New York, NY, USA, 2008. ACM.

[38] J. McDermott, R. Gelinas, and S. Ornstein. Doc, Wyatt, and Virgil: prototyping storage jamming defenses. In *Proceedings of the 13th Annual Computer Security Applications Conference*, ACSAC '97, pages 265–273, New York, NY, USA, 1997. ACM.

[39] Derek R. Hower and Mark 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.

[40] Pablo Montesinos, Matthew Hicks, Samuel T. King, and Josep 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 '09, pages 73–84, New York, NY, USA, 2009. ACM.

[41] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. Recording shared memory dependencies using strata. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '06, pages 229–240, New York, NY, USA, 2006. ACM.

[42] Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. PENELOPE: Weaving threads to expose atomicity violations. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 37–46, New York, NY, USA, 2010. ACM.

[43] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan 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.

[44] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. DoublePlay: Parallelizing sequential logging and replay. *ACM Transactions on Computing Systems*, 30(1):3:1–3:24, February 2012.

[45] Ramesh Chandra, Taesoo Kim, Meelap Shah, Neha Narula, and Nickolai Zeldovich. Intrusion recovery for database-backed web applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 101–114, New York, NY, USA, 2011. ACM.

[46] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The Taser intrusion recovery system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, SOSP '05, pages 163–176, New York, NY, USA, 2005. ACM.

[47] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Intrusion recovery using selective re-execution. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI '10, pages 89–104, Berkeley, CA, USA, 2010. USENIX Association.

[48] Haogang Chen, Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Identifying information disclosure in web applications with retroactive auditing. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, pages 555–569, Broomfield, CO, USA, October 2014. USENIX Association.

[49] Nicolas Viennot, Siddharth Nair, and Jason Nieh. Transparent mutable replay for multicore debugging and patch validation. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 127–138, New York, NY, USA, 2013. ACM.

[50] Mona Attariyan and Jason 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.

[51] Jonathan Heusser and Pasquale 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.

[52] Michael Backes, Boris Kopf, and Andrey Rybalchenko. Automatic discovery and quantification of information leaks. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, SP '09, pages 141–153, Washington, DC, USA, 2009. IEEE Computer Society.

[53] Piotr Mardziel, Mario S. Alvim, Michael Hicks, and Michael 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.

[54] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy 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 '09, pages 109–120, New York, NY, USA, 2009. ACM.

[55] M. Tiwari, Xun Li, H.M.G. Wassel, F.T. Chong, and T. Sherwood. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '09, pages 493–504, Dec 2009.

[56] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben 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.

[57] Mohit Tiwari, Jason K. Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy 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.

[58] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 109–124, Washington, DC, USA, 2010. IEEE Computer Society.

[59] Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 165–178, New York, NY, USA, 2012. ACM.

[60] Roberto Capizzi, Antonio Longo, V. N. Venkatakrishnan, and A. Prasad Sistla. Preventing information leaks through shadow executions. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, pages 322–331, Washington, DC, USA, 2008. IEEE Computer Society.

[61] Zhuofu Bai, Gang Shu, and A. Podgurski. NUMFL: Localizing faults in numerical software using a value-based causal model. In *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation*, ICST '15, pages 1–10, April 2015.

[62] George K. Baah, Andy Podgurski, and Mary Jean 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.

[63] Gang Shu, Boya Sun, A. Podgurski, and Feng Cao. MFL: Method-level fault localization with causal inference. In *Proceedings of the 6th IEEE International Conference on Software Testing, Verification, and Validation*, ICST '13, pages 124–133, March 2013.

[64] Quarterly Threat Report. `https://www.solutionary.com/threat-intelligence/threat-reports/quarterly-threat-reports/sert-threat-report-q4-2016/`.

[65] Trends from the years's breaches and cyber attacks. `https://www.fireeye.com/current-threats/annual-threat-report/mtrends.html`, 2017.

[66] Samuel T. King and Peter M. Chen. Backtracking intrusions. *ACM Transactions on Computing Systems*, 23(1):51–76, February 2005.

[67] Samuel T. King, Zhuoqing Morley Mao, Dominic G. Lucchetti, and Peter M. Chen. Enriching intrusion alerts through multi-host causality. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium, NDSS '05, San Diego, California, USA, February 3-4, 2005*. The Internet Society.

[68] Srinivas Krishnan, Kevin Z. Snow, and Fabian Monrose. Trail of bytes: Efficient support for forensic analysis. In *Proceedings of the 17th ACM SIGSAC Conference on Computer and Communications Security*, CCS '10, pages 50–60, New York, NY, USA, 2010. ACM.

[69] Steve Grubb. RedHat Linux Audit. https://people.redhat.com/sgrubb/audit/.

[70] Microsoft. Event tracing for windows. https://msdn.microsoft.com/en-us/library/windows/desktop/bb968803(v=vs.85).aspx, 2017.

[71] dtrace.org. DTrace. http://dtrace.org/blogs/, 2017.

[72] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium, NDSS '16, San Diego, California, USA, February 21-24, 2017*. The Internet Society.

[73] Shiqing Ma, Kyu Hyung Lee, Chung Hwan Kim, Junghwan Rhee, Xiangyu Zhang, and Dongyan Xu. Accurate, low cost and instrumentation-free security audit logging for windows. *ACSAC'15*.

[74] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. MPI: Multiple perspective attack investigation with semantic aware execution partitioning. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, USENIX SS '17, pages 829–844, Berkeley, CA, USA, 2017. USENIX Association.

[75] Adam Bates, Dave Tian, Kevin R. B. Butler, and Thomas Moyer. Trustworthy whole-system provenance for the linux kernel. In *Proceedings of the 24th USENIX Conference on Security Symposium*, USENIX SS '15, pages 319–334, Berkeley, CA, USA, 2015. USENIX Association.

[76] Kangkook Jee, Georgios Portokalidis, Vasileios P. Kemerlis, Soumyadeep Ghosh, David I. August, and Angelos D. Keromytis. A general approach for effciently accelerating software-based dynamicdata flow tracking on commodity hardware. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium, NDSS '12, San Diego, California, USA, February 5-8, 2012*. The Internet Society.

[77] Kangkook Jee, Vasileios P. Kemerlis, Angelos D. Keromytis, and Georgios Portokalidis. ShadowReplica: Efficient parallelization of dynamic data flow tracking. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 235–246, New York, NY, USA, 2013. ACM.

[78] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. TaintPipe: Pipelined symbolic taint analysis. In *Proceedings of the 24th USENIX Conference on Security Symposium*, USENIX SS '15, pages 65–80, Berkeley, CA, USA, 2015. USENIX Association.

[79] Insider threat spotlight report, 2016. http://crowdresearchpartners.com/wp-content/uploads/2016/09/Insider-Threat-Report-2016.pdf.

[80] Ponemon Institute. 2016 cost of data breach study. https://app.clickdimensions.com/blob/softchoicecom-anjf0/files/ponemon.pdf.

[81] Shijin Kong, Randy Smith, and Cristian Estan. Efficient signature matching with multiple alphabet compression tables. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Networks*, SecureComm '08, pages 1–10, New York, NY, USA, 2008. ACM.

[82] Vern Paxson. Bro: A system for detecting network intruders in real-time. *Comput. Netw.*, 31(23-24), December.

[83] Martin Roesch. Snort. https://www.snort.org/, 2016.

[84] Werner Koch. The GNU privacy guard. https://gnupg.org/, 2017.

[85] Markus Braun. GNUPG vim plugin. https://github.com/jamessan/vim-gnupg/blob/master/plugin/gnupg.vim, 2017.

[86] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, September 1956.

[87] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2:137–167, June 1959.

[88] Mike Muuss. Ping C program. http://ws.edu.isoc.org/materials/src/ping.c.

[89] Albert Cahalan. procps. http://procps.sourceforge.net/, 2009.

[90] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference*, ATC '14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[91] Willem. C implementation of the raft. https://github.com/willemt/raft.

[92] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[93] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. LogGC: garbage collecting audit log. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security*, CCS '13, pages 1005–1016, New York, NY, USA, 2013. ACM.

[94] Gautam Altekar and Ion Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 193–206, New York, NY, USA, 2009. ACM.

[95] DARPA. Transparent Computing. https://www.darpa.mil/program/transparent-computing, 2015.

[96] proftpd-1.3.3c-backdoor. https://www.aldeid.com/wiki/Exploits/proftpd-1.3.3c-backdoor, 2011.

[97] NASA. NASA-HTTP – two months of HTTP logs from the KSC-NASA WWW server. `http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html`, 1995.

[98] Dohyeong Kim, Yonghwi Kwon, William N. Sumner, Xiangyu Zhang, and Dongyan Xu. Dual execution for on the fly fine grained execution comparison. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 325–338, New York, NY, USA, 2015. ACM.

[99] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium, NDSS '05, San Diego, California, USA, February 3-4, 2017*. The Internet Society.

[100] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 109–124, Washington, DC, USA, 2010. IEEE Computer Society.

[101] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, SP '01, pages 144–159, Washington, DC, USA, 2001. IEEE Computer Society.

[102] Andrew P. Kosoresow and Steven A. Hofmeyr. Intrusion detection via system call traces. *IEEE Softw.*, 14(5), September.

[103] Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly detection using call stack information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, SP '03, pages 62–, Washington, DC, USA, 2003. IEEE Computer Society.

[104] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, SP '01, pages 156–, Washington, DC, USA, 2001. IEEE Computer Society.

[105] Xiaokui Shu, Danfeng (Daphne) Yao, Naren Ramakrishnan, and Trent Jaeger. Long-span program behavior modeling and attack detection. *ACM Transactions on Privacy and Security*, 20(4):12:1–12:28, September 2017.

[106] Xiaokui Shu, Danfeng Yao, and Naren Ramakrishnan. Unearthing stealthy program attacks buried in extremely long execution paths. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 401–413, New York, NY, USA, 2015. ACM.

[107] Z. Li and A. Oprea. Operational security log analytics for enterprise breach detection. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 15–22, Nov 2016.

[108] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM SIGSAC Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[109] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information System Security*, 15(1):2:1–2:34, March 2012.

[110] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, New York, NY, USA, 2011. ACM.

[111] Skylined. `http://www.edup.tudelft.nl/~bjwever/advisory_iframe.html.php`.

[112] Yu Ding, Tao Wei, TieLei Wang, Zhenkai Liang, and Wei Zou. Heap taichi: Exploiting memory allocation granularity in heap-spraying attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 211–221, New York, NY, USA, 2010. ACM.

[113] K2. ADMmutate documentation. `http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz`, 2003.

[114] Theo Detristan, Tyll Ulenspiegel, Yann Malcom, and Mynheer Superbus Von Underduk. Polymorphic shellcode engine using spectrum analysis. `http://phrack.org/issues/61/9.html`, 2003.

[115] Joshua Mason, Sam Small, Fabian Monrose, and Greg MacManus. English shellcode. In *Proceedings of the 16th ACM SIGSAC Conference on Computer and Communications Security*, CCS '09, pages 524–533, New York, NY, USA, 2009. ACM.

[116] Metasploit development team. Metasploit project. `http://metasploit.com`, 2006.

[117] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, USENIX SS '14, pages 941–955, Berkeley, CA, USA, 2014. USENIX Association.

[118] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. SafeDispatch: Securing C++ virtual calls from memory corruption attacks. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium, NDSS '14, San Diego, California, USA, February 23-26, 2014*. The Internet Society.

[119] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 559–573, Washington, DC, USA, 2013. IEEE Computer Society.

[120] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Conference on Security*, USENIX SS '13, pages 447–462, Berkeley, CA, USA, 2013. USENIX Association.

[121] Sandeep Bhatkar and R. Sekar. Data space randomization. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 1–22, Berlin, Heidelberg, 2008. Springer-Verlag.

[122] Antonis Papadogiannakis, Laertis Loutsis, Vassilis Papaefstathiou, and Sotiris Ioannidis. Asist: Architectural support for instruction set randomization. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 981–992, New York, NY, USA, 2013. ACM.

[123] Chris Anley. Creating arbitrary shellcode in unicode expanded strings, the "venetian" exploit. `https://www.helpnetsecurity.com/dl/articles/unicodebo.pdf`, 2002.

[124] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. TACAS'08/ETAPS'08, Berlin, Heidelberg. Springer-Verlag.

[125] Exploits database by offensive security. `https://www.exploit-db.com/`.

[126] Penetration testing software. metasploit. `https://www.metasploit.com/`.

[127] Jonathan Salwan. Shellcodes database for study cases. `http://shell-storm.org/shellcode/`.

[128] Sascha Schirra. Ropper – rop gadget finder and binary information tool. `https://scoding.de/ropper/`.

[129] Sascha Schirra. ROPgadget – gadgets finder and auto-roper. `http://shell-storm.org/project/ROPgadget/`.

[130] Masaki Suenaga. Evolving shell code. Whitepaper, Symantec Security Response, Japan, 2006.

[131] William N. Sumner, Yunhui Zheng, Dasarath Weeratunge, and Xiangyu Zhang. Precise calling context encoding. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 525–534, New York, NY, USA, 2010. ACM.

[132] The llvm compiler infrastructure. `http://llvm.org/`.

[133] Thomas Toth and Christopher Kruegel. Accurate buffer overflow detection via abstract payload execution. In *Proceedings of the 5th International Conference on Recent Advances in Intrusion Detection*, RAID '02, pages 274–291, Berlin, Heidelberg, 2002. Springer-Verlag.

[134] Zhenkai Liang and R. Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proceedings of the 12th ACM SIGSAC Conference on Computer and Communications Security*, CCS '05, pages 213–222, New York, NY, USA, 2005. ACM.

[135] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Comprehensive shellcode detection using runtime heuristics. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 287–296, New York, NY, USA, 2010. ACM.

[136] Ramkumar Chinchani and Eric van den Berg. A fast static analysis approach to detect exploit code inside network flows. In *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, RAID '05, pages 284–308, Berlin, Heidelberg, 2006. Springer-Verlag.

[137] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM SIGSAC Conference on Computer and Communications Security*, CCS '03, pages 290–299, New York, NY, USA, 2003. ACM.

[138] Monirul I. Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. Impeding malware analysis using conditional code obfuscation. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium, NDSS '08, San Diego, California, USA, February 3-4, 2008*. The Internet Society.

[139] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference*, ATC '12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.

[140] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th Conference on USENIX Security Symposium*, USENIX-SS '05, pages 12–12, Berkeley, CA, USA, 2005. USENIX Association.

[141] Pieter Philippaerts, Yves Younan, Stijn Muylle, Frank Piessens, Sven Lachmund, and Thomas Walter. Code pointer masking: Hardening applications against code injection attacks. In *Proceedings of the 8th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'11, pages 194–213, Berlin, Heidelberg, 2011. Springer-Verlag.

[142] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 353–362, New York, NY, USA, 2011. ACM.

[143] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Conference on Security Symposium*, USENIX SS '13, pages 337–352, Berkeley, CA, USA, 2013. USENIX Association.

[144] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[145] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 927–940, New York, NY, USA, 2015. ACM.

[146] Ben Niu and Gang Tan. Per-input Control-Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 914–926, New York, NY, USA, 2015. ACM.

[147] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, USENIX SS '14, pages 401–416, Berkeley, CA, USA, 2014. USENIX Association.

[148] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, USENIX SS '14, pages 417–432, Berkeley, CA, USA, 2014. USENIX Association.

[149] Nicholas Carlini and David Wagner. ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, USENIX SS '14, pages 385–399, Berkeley, CA, USA, 2014. USENIX Association.

[150] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. Evaluating the effectiveness of current anti-ROP defenses. In *Proceedings of the 17th International Conference on Recent Advances in Intrusion Detection*, RAID '14, pages 88–108, Cham, Switzerland, 2014. Springer International Publishing.

[151] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 745–762, Washington, DC, USA, 2015. IEEE Computer Society.

[152] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 952–963, New York, NY, USA, 2015. ACM.

[153] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Conference on Security Symposium*, USENIX SS '15, pages 161–176, Berkeley, CA, USA, 2015. USENIX Association.

[154] Prahlad Fogla, Monirul Sharif, Roberto Perdisci, Oleg Kolesnikov, and Wenke Lee. Polymorphic blending attacks. In *Proceedings of the 15th Conference on USENIX Security Symposium*, USENIX-SS '06, Berkeley, CA, USA, 2006. USENIX Association.

[155] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Emulation-based detection of non-self-contained polymorphic shellcode. In *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*, RAID '07, pages 87–106, Berlin, Heidelberg, 2007. Springer-Verlag.

[156] Kevin Z. Snow, Srinivas Krishnan, Fabian Monrose, and Niels Provos. Shellos: Enabling fast detection and forensic analysis of code injection attacks. In *Proceedings of the 20th USENIX Conference on Security Symposium*, USENIX SS '11, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.

[157] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of the 18th Conference on USENIX Security Symposium*, USENIX-SS '09, pages 169–186, Berkeley, CA, USA, 2009. USENIX Association.

[158] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, ACSAC '06, pages 339–348, Washington, DC, USA, 2006. IEEE Computer Society.

[159] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM SIGSAC Conference on Computer and Communications Security*, CCS '12, pages 157–168, New York, NY, USA, 2012. ACM.

[160] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 601–615, Washington, DC, USA, 2012. IEEE Computer Society.

[161] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 268–279, New York, NY, USA, 2015. ACM.

[162] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical code randomization resilient to memory disclosure. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 763–780, Washington, DC, USA, 2015. IEEE Computer Society.

[163] Yue Chen, Zhi Wang, David Whalley, and Long Lu. Remix: On-demand live randomization. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 50–61, New York, NY, USA, 2016. ACM.

[164] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. Ilr: Where'd my gadgets go? In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 571–585, Washington, DC, USA, 2012. IEEE Computer Society.

[165] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM SIGSAC Conference on Computer and Communications Security*, CCS '03, pages 272–280, New York, NY, USA, 2003. ACM.

[166] Georgios Portokalidis and Angelos D. Keromytis. Fast and practical instruction-set randomization for commodity systems. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 41–48, New York, NY, USA, 2010. ACM.

[167] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 227–242, Washington, DC, USA, 2014. IEEE Computer Society.

[168] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM SIGSAC Conference on Computer and Communications Security*, CCS '08, pages 27–38, New York, NY, USA, 2008. ACM.

[169] F.J. Serna. CVE-2012-0769, the case of the perfect info leak. `http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf`.

[170] Byoungyoung Lee, Long Lu, Tielei Wang, Taesoo Kim, and Wenke Lee. From zygote to morula: Fortifying weakened aslr on android. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 424–439, Washington, DC, USA, 2014. IEEE Computer Society.

[171] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 574–588, Washington, DC, USA, 2013. IEEE Computer Society.

[172] Stephen J. Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. It's a trap: Table randomization and protection against function-reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 243–255, New York, NY, USA, 2015. ACM.

[173] Cristian Cadar, Periklis Akritidis, Manuel Costa, Jean-Philippe Martin, and Miguel Castro. Data randomization. Technical Report MSR-TR-2008-120.

[174] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointguardTM: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th Conference on USENIX Security Symposium*, USENIX-SS '03, pages 7–7, Berkeley, CA, USA, 2003. USENIX Association.

[175] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium*, USENIX-SS '98, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[176] Gerardo Richarte. Four different tricks to bypass StackShield and StackGuard protection. 05 2002.

[177] Bulba and Kil3r. Bypassing Stackguard and Stackshield. `http://phrack.org/issues/56/5.html`, 2000.

[178] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the 7th Annual Network and Distributed System Security Symposium, NDSS '00, San Diego, California, USA, February 3-4, 2000*. The Internet Society.

[179] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th Conference on USENIX Security Symposium*, USENIX-SS '01, Berkeley, CA, USA, 2001. USENIX Association.

[180] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with wit. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 263–277, Washington, DC, USA, 2008. IEEE Computer Society.

[181] Eric Wimberley. Bypassing AddressSanitizer. `https://packetstormsecurity.com/files/123977/Bypassing-AddressSanitizer.html`.

VITA

VITA

Yonghwi Kwon received his B.E. degree in computer engineering from Kunkuk University in 2011. He attended Purdue University from 2012-2018 pursuing his Ph.D. under the guidance of Prof. Xiangyu Zhang and Prof. Dongyan Xu. He received his M.S. and Ph.D. degrees in computer science from Purdue University in 2017 and 2018 respectively. He is broadly interested in solving system security problems via program analysis with a special focus on attack investigation, software exploit prevention, cross-platform binary analysis and reverse-engineering. He has been honored with the ASE Best Paper Award in 2013, ACM SIGSOFT Distinguished Paper Award in 2013, and Maurice H. Halstead Memorial Award in 2017. In the fall of 2018, he joined the faculty of the University of Virginia as an Assistant Professor of Computer Science.