Secrecy Included: Confidentiality Enforcement for Machine Code

vorgelegt von
Tobias Ferdinand Pfeffer, M.Sc.

an der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
- Dr.-Ing. -
Doktor der Ingenieurwissenschaften

genehmigte Dissertation

Promotionsausschuss:
Vorsitzender: Prof. Dr. Markus Brill
Gutachterin: Prof. Dr. Sabine Glesner
Gutachter: Prof. Dr. Florian Tschorsch
Gutachter: Prof. Dr. Christian Hammer

Tag der wissenschaftlichen Aussprache: 19. Juli 2021

Berlin 2022

Abstract

Confidential software is an important requirement for the ongoing digitalization. Yet, since confidentiality is a program-wide property, this requirement is at odds with modern software development. In order to produce software cheaply and quickly, existing software components from third-party manufacturers are regularly included in new products. Consequently, a method is needed to ensure the confidentiality across third-party components, which are usually shipped in compiled form.

In this thesis we present a new method that enforces confidentiality for compiled programs. Our approach protects against leaks through data flow, control flow, termination behavior, and timing of outputs. At the same time, it achieves per-channel transparency, thereby preserving the functionality of secure components. Our solution is the first practical application of the concept of Secure Multi-Execution to machine code.

Secure Multi-Execution is an enforcement mechanism that achieves confidentiality by design. The target program is executed multiple times, where each copy has restricted access to input and output channels. Unfortunately, this multi-execution comes with a significant performance overhead. Thus, we propose two optimizations that significantly increase the efficiency of the protection mechanism. Our principle is to avoid redundant calculations as much as possible. On the one hand, we show how creation of multiplied executions can be delayed through our dynamic instancing method. On the other hand, we show how multiplied executions can be terminated early through our bounding method.

In order not to reduce the security guarantees of Secure Multi-Execution through our optimizations, we additionally present a new scheduling strategy for the multiple executions. This strategy allows us to ensure independent progress of executions, while maintaining external dependencies in their output behavior. We also present our contributions to static analysis of binary code, which serves as a basis for further research into the automatic determination of suitable termination points for our optimizations.

We have successfully applied our method to the Linux operating system and the widely used x86_x64 architecture. In our evaluation of benchmark programs and real-world targets, we show that our system protects against the above mentioned information leaks while changing the functionality of the target program as little as possible. With our work we enable for the first time the protected use of existing components and thus a fast and cost-effective development of confidential programs.

Zusammenfassung

Vertrauliche Software ist eine wichtige Voraussetzung für die fortschreitende Digitalisierung. Da Vertraulichkeit jedoch eine programmweite Eigenschaft ist, steht diese Anforderung im Widerspruch zur modernen Softwareentwicklung. Bestehende Softwarekomponenten von Drittherstellern werden regelmäßig in neue Produkte integriert, um Software kostengünstig und schnell entwickeln zu können. Folglich ist eine Methode erforderlich, um die Vertraulichkeit von Komponenten von Drittherstellern, die in der Regel in kompilierter Form ausgeliefert werden, zu gewährleisten.

In dieser Arbeit stellen wir eine neue Methode vor, die die Vertraulichkeit für kompilierte Programme garantiert. Unser Ansatz schützt vor Informationsverlust durch Datenfluss, Kontrollfluss, Terminierungsverhalten und den Zeitpunkten der Ausgaben. Gleichzeitig wird eine Transparenz für pro Kanal erreicht, wodurch die Funktionalität der sicheren Komponenten erhalten bleibt. Unsere Lösung ist die erste praktische Anwendung des Konzepts der Secure Multi-Execution auf Maschinencode.

Secure Multi-Execution ist ein Enforcement Mechanismus, der Vertraulichkeit per Konstruktion erreicht. Das Zielprogramm wird mehrfach ausgeführt, wobei jede Kopie einen eingeschränkten Zugang zu den Ein- und Ausgabekanälen hat. Allerdings ist diese Mehrfachausführung mit einem erheblichen Performance-Overhead verbunden. Daher präsentieren wir zwei Optimierungen, die die Effizienz des Schutzmechanismus erheblich steigern. Unser Prinzip ist es, redundante Berechnungen so weit wie möglich zu vermeiden. Einerseits zeigen wir, wie die Erstellung von Mehrfachausführungen durch unser dynamisches Instantiieren verzögert werden kann. Zum anderen zeigen wir, wie multiplizierte Ausführungen durch unser Bounding vorzeitig beendet werden können.

Um die Sicherheitsgarantien von Secure Multi-Execution durch unsere Optimierungen nicht zu verringern, stellen wir zusätzlich eine neue Ablaufplanung für die Mehrfachausführungen vor. Diese Strategie ermöglicht es uns, einen unabhängigen Fortschritt der Ausführungen zu gewährleisten und gleichzeitig externe Abhängigkeiten in ihrem Ausgabeverhalten beizubehalten. Wir stellen auch unsere Beiträge zur statischen Analyse des Binärcodes vor, die als Grundlage für weitere Forschungen zur automatischen Bestimmung geeigneter Endpunkte für unsere Optimierungen dient.

Wir haben unsere Methode erfolgreich auf das Linux-Betriebssystem und die weit verbreitete x86_x64-Architektur angewandt. Bei der Auswertung von Benchmarkprogrammen und echten Zielen zeigen wir, dass unser System gegen die oben genannten Informationslecks schützt und gleichzeitig die Funktionalität des Zielprogramms so wenig wie möglich verändert. Mit unserer Arbeit ermöglichen wir erstmals die geschützte Nutzung bestehender Komponenten und damit eine schnelle und kostengünstige Entwicklung von vertraulichen Programmen.

Danksagung

Diese Arbeit ist zu einem großen Teil während der COVID-19 Pandemie im Home Office entstanden. Für diese Zeit gilt mein größter Dank meinen Kollegen Joachim Fellmuth und Dr. Verena Klös, die mir aus der Ferne mit Rat und Tat zur Seite standen. Joachim Fellmuth gilt zusätzlich mein Dank für die gemeinsame Arbeit in der Lehre zu einer Zeit, in der sich die Studierendenzahlen verdoppelten. Es war nicht immer einfach, aber mit vereinter Kreativität und Disziplin haben wir ein wie ich finde stolzes Ergebnis erreicht. Auch möchte ich mich bei den Kollegen des Fachgebiets für die wunderbare Atmosphäre, das eine oder andere Kickerspiel, die turbulenten Filmdrehs und das konstruktive Feedback bedanken. Gerne wäre ich noch ein letztes Mal mit euch zum Retreat gefahren. Schließlich geht mein Dank an die ehemaligen Kollegen, allen voran Prof. Dr. Paula Herber und Dr. Thomas Göthel, die als Postdoktoranden meine ersten Schritte begleitet und meine Arbeit wesentlich beeinflusst haben.

Ein großer Dank gilt Prof. Dr. Sabine Glesner für die Chance zu promovieren und für die vielzählige Unterstützung in diesen Jahren. Ihr großes Vertrauen gab mir die Möglichkeit, meiner Kreativität zu folgen. Sie hat mich auch zu vielen Erfahrungen während der Promotion ermuntert, darunter das Beantragen und Leiten eines eigenen Projekts, die Teilnahme an Berufungskommissionen, die Betreuung von Abschlussarbeiten und Seminaren und das Besuchen von Konferenzen und Summer Schools. Diesbezüglich gilt mein Dank auch den Veranstaltern, Dozenten und Teilnehmern der Markt Oberdorf Summer School 2017, insbesondere den Professoren Magnus Myreen, Daniel Kroening und Nikolaj Bjørner für Ihre entscheidenden Hinweise. Ebenso dankbar bin ich den Veranstaltern, Teilnehmern und beteiligten Unternehmen des Software Campus, sowie dem DLR und dem BMBF für die Förderung meines Projekts. Die Erfahrungen aus den Weiterbildungen und der Projektleitung haben bereits jetzt mehrfach Anwendung gefunden.

Zum Schluss gilt mein Dank natürlich auch meiner Familie und meinen Freunden. Ein konzentriertes Arbeiten an der Promotion wäre ohne sie in den vergangenen Jahren nicht möglich gewesen. Gelegentliche Zerstreuung im berliner Nachtleben allerdings auch nicht. Für beides - und noch viel mehr - bin ich ihnen sehr dankbar.

Contents

List of Symbols **v**

List of Semantics **vi**

List of Figures **vii**

List of Tables **ix**

1 Introduction **1**

2 Background **9**
2.1 Confidentiality 9
2.1.1 Security Model 10
2.1.2 Program Behavior 11
2.1.3 Noninterference 14
2.2 Secure Multi-Execution 19
2.2.1 Local Semantics 21
2.2.2 Global Semantics 22
2.2.3 Transparency 23
2.3 Machine Code 24
2.3.1 Semantics 25
2.3.2 Data Flow 27
2.3.3 Control-Flow Integrity 29
2.4 Summary 30

3 Related Work **33**
3.1 Multi-Execution Enforcement 33
3.2 Language-Based Enforcement 36
3.3 Binary Analysis 37
3.4 Summary 38

I Confidentiality Enforcement for Machine Code 41

4 Threat Model 45
4.1 Decryption Service . . . 45
4.2 Compiled Form . . . 48
4.3 Leaks . . . 52
4.4 Attacks . . . 53
4.5 Summary . . . 55

5 Secure Multi-Execution for Machine Code 57
5.1 Core Elements . . . 58
5.2 Semantics . . . 61
5.3 Example . . . 62
5.4 Summary . . . 67

6 Dynamic Instancing Optimization 69
6.1 Reasoning . . . 70
6.2 Semantics . . . 72
6.2.1 Local Semantics . . . 72
6.2.2 Global Semantics . . . 74
6.3 Implementation . . . 76
6.3.1 Fork Injection . . . 76
6.3.2 Shared Input . . . 78
6.4 Example . . . 78
6.5 Summary . . . 82

7 Bounding Optimization 83
7.1 Synchronization . . . 84
7.2 Semantics . . . 88
7.3 Implementation . . . 90
7.3.1 Setting Boundaries . . . 90
7.3.2 Identifying Input . . . 91
7.4 Example . . . 92
7.5 Summary . . . 96

8 Timing-Sensitive Scheduling 97
8.1 Reordering . . . 98
8.2 Construction . . . 99
8.3 Example . . . 102
8.4 Summary . . . 106

9 Boundary Analysis 109

9.1 Boundary Extraction . . . 109
9.2 Indirect Call Resolution . . . 113
9.2.1 Resolution . . . 113
9.2.2 Summary . . . 116

II Evaluation 119

10 Benchmark 125
10.1 Setup . . . 127
10.2 Results . . . 128
10.3 Summary . . . 131

11 Coreutils 133
11.1 Setup . . . 134
11.2 Results for Word Count . . . 136
11.3 Results for Cat . . . 138
11.4 Summary . . . 143

12 Conclusion 145

Bibliography 149

A Benchmark Programs 159

B Additional Results 169
B.1 Sort . . . 169
B.2 SHA Sum . . . 171

C Stream Input 179
C.1 Interaction . . . 179
C.2 Virtual Filesystem . . . 181
C.3 Example . . . 182

List of Symbols

s	State
$\mathbb{C}$	Abstract channels
$\mathbb{B}$	Boolean values
$\mathbb{N}$	Natural Numbers
ℓ	Security level
$\mathcal{L}$	Security lattice
L	Security level subset
$\sqcup X$	Least upper bound
π	Labeling function
$\xrightarrow{a}$	Event transition
$c?v$	Input event
$c!v$	Output event
$\bullet$	Silent event
e	Environment
$\restriction$	Projection function
δ	Dummy input
σ	Scheduler
Σ	Program text
ρ	Register store
μ	Value store
pc	Program counter
ι	Current instruction
sp	Stack pointer
$\Downarrow$	Expression evaluation
$\diamond$	Input terminator
$\mathcal{B}$	Boundary function
B	Boundary stack
β	Boundary condition
$\times$	Termination event
$\star$	Blocking input
b	Buffer
r	Global input pointer
p	Local input pointer

List of Semantics

2.1 Local Secure Multi-Execution semantics 21
2.2 Global Secure Multi-Execution semantics 23
2.3 Machine Code State 25
2.4 Machine Code Semantics 26

4.1 System call semantics for input and output 50

5.1 Local Secure Multi-Execution semantics for machine code 62

6.1 Local Secure Multi-Execution semantics with Dynamic Instancing 73
6.2 Dynamic Instantiation Semantics 74

7.1 Barrier-Based Bounding Semantics 89

C.1Semantics for volatile/visible input 180

List of Figures

1.1 Schematic comparison of SME and our optimizations 5

2.1 Confidentiality in our program model 11
2.2 Projection function definition 13
2.3 Examples of weak and strong timing-sensitive programs [42] 18
2.4 Secure Multi-Execution with two levels 20
2.5 Explicit flow through pointers 28
2.6 Control-Flow Integrity [1] 30

3.1 Schematic comparison of our SME optimizations 44

4.1 Information flow schematic of the example program 46
4.2 Security lattice for the decryption service 47
4.3 Exemplary timing-attacks 55

5.1 Schematic comparison of unprotected and SME-protected execution 58
5.2 Secure Multi-Execution via Syscall Monitoring 59
5.3 Secure Multi-Execution based enforcement system 60
5.4 Protection against timing-attacks 64
5.5 SME Efficiency 65
5.6 Visualization of Secure Multi-Execution applied to the example . . 66

6.1 Comparison of SME protection without and with Dynamic Instancing 70
6.2 Illustration of Dynamic Instantiation 71
6.3 Fork injection 77
6.4 Efficiency improvement of Dynamic Instancing 79
6.5 Optimized run of the example 80
6.6 Protection against timing-attacks with Dynamic Instancing 80

7.1 Illustration of our Bounding Optimization 84
7.2 Bounds of the information flows in the example program 85
7.3 Illustration of our Bounding Optimization 86
7.4 Bounded run of the example 93
7.5 Efficiency improvement of our Bounding Optimization 94

7.6 Timing-Attacks on SME with bounding 94

8.1 Effect of our timing-sensitive scheduling 98
8.2 Forking new executions with virtual executions 101
8.3 Timing-sensitive optimization of the example 102
8.4 Timing-attack on optimized SME with timing-sensitive scheduling . 105
8.5 Efficiency impact of timing-sensitive scheduling 106

9.1 Control- and data-flow graph of the running example 111
9.2 Instrumented calls example 114
9.3 (Optimized) Secure Multi-Execution toolchain for machine code . . 122

10.1 Information flows in the benchmark programs 127

11.1 Lattices used during evaluation 134
11.2 Schematic CFGs with data-flow and boundaries for the two categories 135
11.3 Effect of confidentiality enforcement on `wc` 136
11.4 Evaluation results for `wc` 137
11.5 Run-time sampling of `cat` 139
11.6 Level sampling of `cat` 140
11.7 Time sampling of the first output event 141
11.8 File split sampling of `cat` 141

B.1 Overhead sampling for different input sizes for `sort` 169
B.2 Overhead sampling for different number of input levels for `sort` . . 170
B.3 Run-time sampling of `sha1sum` 171
B.4 Level sampling of `sha1sum` 172
B.5 Time sampling of the first output event 173
B.6 File split sampling of `sha1sum` 173
B.7 Samplings of `sha224sum` 174
B.8 Samplings of `sha256sum` 175
B.9 Samplings of `sha384sum` 176
B.10 Samplings of `sha512sum` 177

List of Tables

4.1 Exemplary attacks and timing-insensitive results 54

5.1 Noninterference through Secure Multi-Execution for the example from Chapter 4. 63

6.1 Progress-sensitive noninterference of the example in Chapter 4 through SME with Dynamic Instancing. 81

7.1 Timing-insensitive results with Bounding Optimization 95

8.1 Queue states for the execution in Figure 8.3 103
8.2 PSNI-noninterference results of the example Chapter 4 for our timing-sensitive scheduler 104

10.1 Overview of benchmark examples 126
10.2 Evaluation results 129

C.1 Evaluation results 183

Chapter 1

Introduction

Confidential software, meaning software that does not leak sensitive inputs, is required in many areas of modern society. It allows sharing of private information between private citizens as well as company secrets between business partners. Unfortunately, development of confidential software is not an easy task. Confidentiality is a program-wide property, meaning that a program can only be confidential if all of its components are confidential. This is at odds with modern software development. Developers routinely include third-party components to reduce the time to market and development cost of new software. These components may be insecure, especially when not designed with confidentiality in mind. Additionally they are usually shipped closed-source to protect intellectual property. This raises the question how to enforce confidentiality across all components of a program, including third-party code that is shipped in compiled form.

As an example, many nations recently initiated programs to produce applications that track the spread of the COVID-19 pandemic. Given that such an application must collect sensitive data, its use is often voluntary. Yet, the effectiveness of the application depends on its wide-spread use. Thus, the measure will only be successful if the general public can be convinced of the confidential treatment of the collected data. To inspire confidence in the solution, some nations released the source code of the application. Yet, this approach relies on volunteering experts to scrutinize the code, including all reused components. Wherever compiled third-party code is included, this would require thorough analysis of machine code, unlikely to be feasible in practice. Restricted access to the sensitive information, on the other hand, would break the functionality of the application.

The problem is that no practical solution exists to *enforce* strong confidentiality across all components of a system, including third-party components or legacy code. Existing solutions mostly focus on validation of confidentiality for a program.

This limits them in three ways. First, static analysis of binaries is still a challenging task, leading to a high error rate [4, 10, 54, 75]. Second, leaks through external timing behavior are either not covered or lead to a very restrictive computation model [42]. Third, successful identification of a leak is only the first step to a functioning and secure component, which would then require a binary rewriting step to make the target secure. Yet, like binary analysis, binary rewriting is sill a challenging task [85].

Our goal is, thus, to define and develop a new method to enforce confidentiality for machine code. With this new method, it should be possible to prevent potentially insecure components from leaking sensitive information. At the same time, we aim to retain as much of the original functionality of the components as possible. Concretely, outputs produced by secure components should remain the same in content and order. At the same time, leaking outputs from insecure components should be either suppressed or sanitized to not contain sensitive information. Such a solution would allow developers to reuse the functionalities provided by existing software components without breaking the confidentiality requirements of their products.

Naturally, the solution also needs to be practical. First and foremost, this means that it should apply to compiled code. Since most third-party components are shipped in compiled form, focus on a specific high-level language would be insufficient. Yet, the security and transparency guarantees should not require knowledge of the code. Comprehensive analysis of compiled programs is a complex task that still requires oversight by expert personnel, which might not always be available. However, additional information may be used to increase the enforcement *efficiency*. In some recent enforcement approaches, dealing with many users at different levels of security clearance leads to exponential overhead. Since this impedes the wide-spread use of the technique, we aim to find a more efficient solution. In the following, we describe the objectives of our work in more detail.

Secure. Our primary objective is to enforce confidentiality. This means that observation of public outputs from a program execution should not allow an unauthorized observer to infer sensitive inputs to the program. In other words, sensitive inputs should not interfere with public outputs, usually described as *noninterference*. A program is noninterferent when the content of public outputs does not depend on private inputs. Additionally, information may leak due the timing of public outputs. As long as they are not delayed in correlation with sensitive input content or size, we consider the enforcement *timing-sensitive*. Additional information side-channels like power consumption, heat radiation or noise are not the focus of this thesis.

Transparent. The enforcement should not break the functionality of the target program. This means that for *any program*, outputs to authorized users should not change. Additionally, for all originally *secure programs*, unauthorized users should also observe the same outputs in the same order as the original execution. Due to the security requirements, public output from insecure components may have to be altered or removed. For the same reason, it cannot generally be guaranteed that messages appear at the same time under enforcement nor that the order of outputs between channels is maintained.

Efficient. Where enforcement requires additional computation at run time, it usually induces an overhead. Naturally, this overhead should be as small as possible. This can be achieved in two ways, either through optimization of the implementation, or through optimization of the enforcement approach. In this thesis, we focus on the latter. Thus, we aim to reduce the conceptual overhead of the enforcement method. Ideally, the overhead would remain constant, even if the number of users or complexity of the target was increased. More users, with more unique security clearances, usually implies more complex information flows that must be addressed by the enforcement method. A more complex target, roughly estimated by code size, also increases the complexity of the enforcement. An exponential growth due to either attribute would adversely limit the applicability of the approach.

Practical. To be practical, the enforcement system must be applicable to compiled code. This means that it must be able to deal with prevalent peculiarities of compiled code, such as interaction with the operating system, blending of data- and control-flow, lack of high-level information and more. The resulting technical complexity should not result in insecurity. Thus, even where precise analysis is not available or infeasible, security must be guaranteed. Consequently, a practical solution uses expert knowledge only to increase the efficiency. Support of more complex features such as multi-threading, inter-process communication, or socket communication are not the focus of this thesis and probably best be integrated into a kernel-level implementation.

Our proposed solution is based primarily on the concept of Secure Multi-Execution, as described in the seminal paper by Devriese and Piessens [32]. Secure Multi-Execution has been formally proven to provide timing-sensitive noninterference guarantees, while also ensuring transparency for authorized and unauthorized users [16, 65]. This makes it a promising solution, as it satisfies both our security and transparency requirements. Furthermore, it can be formulated independent

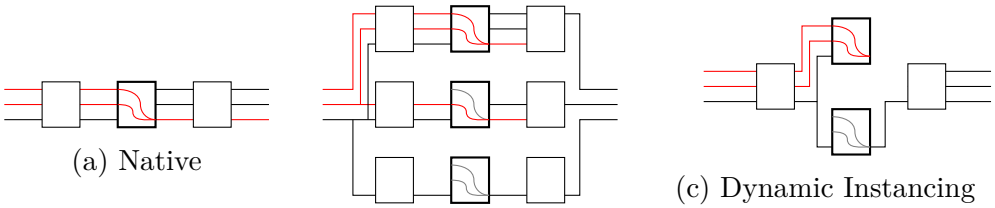
of the operational semantics of the target language. This *theoretically* shows that it can be applied to enforce confidentiality for machine code.

Notwithstanding the benefits of Secure Multi-Execution, its technical complexity has so far discouraged an application to machine code. Thus, we provide the first *practical* application of Secure Multi-Execution to machine code. To achieve this, we reformulate Secure Multi-Execution based on the operational semantics of machine code and implement it as a monitoring system for Linux running on the wide-spread x86_x64 architecture. As the name suggests, Secure Multi-Execution executes the same program multiple times, once for each security clearance that users can hold. When users can hold multiple clearances, the possible combinations require additional executions. This results in an exponential overhead that quickly becomes impractical. Thus, we design and develop new optimizations that allow to reduce the run-time overhead and thus to increase the efficiency of our enforcement method, without sacrificing security or transparency.

First, we introduce a method to instantiate new executions *dynamically*. This allows us to create new executions only when information at an hitherto unserviced clearance is obtained. Notably, it allows us to create as few executions as possible and as late as possible, potentially leading to far less executions than the number of unique security clearances would indicate. Additionally, we introduce a method to terminate multiplied executions when they become redundant. Where two executions perform computations based on the same information, one computation would suffice. Thus, our *bounding* optimization ensures that we terminate executions as soon as possible. In the best case, our optimizations ensure that the enforcement overhead stays constant both with increasing program size and increasing number of security levels, making multi-execution based confidentiality enforcement practical.

Naturally, our optimizations must be realized in accordance with the other criteria. Thus, we additionally introduce a timing-sensitive scheduling for our Bounding Optimization. The key idea is to allow executions for lower levels of security clearance to progress and terminate independently of executions with higher security classification. At the same time, we reorder outputs to higher channels, to guarantee the ideal mixture of timing-sensitive noninterference and order-preserving transparency. Finally, our Bounding Optimization requires additional information that must be provided as input. It can be provided manually, based on developer knowledge, or through static analysis. To aid automatic extraction of termination conditions from binaries, we contribute heuristics to resolve indirect call targets. This allows us to automatically extract termination conditions, which can be refined manually or with further analysis.

The difference between unprotected execution, protected execution, and our



(a) Native

(b) Secure Multi-Execution

(c) Dynamic Instancing and Bounding

Figure 1.1: Schematic comparison of SME and our optimizations

optimized protection is visualized in Figure 1.1. Here, we consider a program that consists of three components, two of which are secure and one that is insecure. Information from the first secure component is passed through the insecure component on to the second secure component. The insecure component connects information with higher clearance to outputs with lowest clearance, which violates confidentiality. This models a situation where an insecure library is included into an otherwise secure system. Using Secure Multi-Execution enforcement, the three components are executed three times, once per security level. This needlessly multiplies execution of the secure parts of the system. Additionally, assuming that input is only obtained on two out of the three levels, it also needlessly creates additional executions for the third level. On the other hand, using our optimized Secure Multi-Execution enforcement, only the insecure component is executed multiple times. It is also only multiplied twice, when input from only two levels is needed. Thus, assuming equivalent cost for each component, the optimized Secure Multi-Execution overhead is only four thirds (133 %), whereas the unoptimized Secure Multi-Execution overhead is nine thirds (300 %). Consequently, in this idealized setting, the efficiency of our optimized enforcement is more than twice as high.

Our work reflects recent findings that show that transparent enforcement of confidentiality requires multi-execution based approaches [2]. It also reflects the finding that these approaches are generally not efficient when applied in a black-box setting, and that increased efficiency requires knowledge of the program code. Our solution demonstrates how a demand-driven multi-execution enforcement can cross the bridge between security, transparency and efficiency. Thus, it offers a promising direction for future research, beyond its practical consequences. In detail, we make the following contributions.

1. We are the first to apply Secure Multi-Execution to machine code and the first to provide a working prototype for Linux on x86_x64 to demonstrate the practicality of the approach. This provides a method to achieve practical security and transparency.

2. We provide a *Dynamic Instancing* optimization for Secure Multi-Execution to increase the enforcement efficiency. Through Dynamic Instancing, we create new executions only when needed. This increases the enforcement efficiency in many cases.

3. We provide a *Bounding Optimization* to further increase the efficiency through termination of multiplied executions. This increases the efficiency of our enforcement method in even more cases and can severely reduce the overhead resulting from a high number of security levels.

4. We provide a *timing-sensitive scheduling* method for our Bounding Optimization of Secure Multi-Execution. Our scheduler enables high enforcement efficiency in many cases, without compromising the high security and transparency guarantees of Secure Multi-Execution.

5. We provide heuristics that aid the automatic extraction of termination conditions for our Bounding Optimization from machine code programs. This shows a promising direction to make our Bounding Optimization applicable to any target, without the need for manual analysis.

Note that we also provide formalized semantics of our enforcement solutions throughout the thesis. These are intended to illustrate their construction as an abstraction from the real code. We do not provide rigorous proofs for these semantics. Yet, to ensure their correctness, we emulated and validated them in a prototypical implementation. We then implemented an enforcement toolchain for real compiled code based on our formalized semantics. We use this toolchain to evaluate both a benchmark set, extracted from various papers and highlighting various challenges for transparent information flow control, and real-world, compiled examples to show the practical effect of our solution with test cases, measurements, and visualized traces. Throughout the thesis, we use an additional, synthetic example to highlight the usefulness and capabilities of our system. Our evaluation results show that our ideas carry over into our implementation and that our solution works on actual binaries. The contributions in this thesis extend our work as published in the following papers.

- Tobias Pfeffer, Thomas Göthel, and Sabine Glesner. *Efficient and Precise Information Flow Control for Machine Code through Demand-Driven Secure Multi-Execution*, page 197–208. Association for Computing Machinery, New York, NY, USA, 2019 (Outstanding Paper Award)

- Tobias Pfeffer and Sabine Glesner. Timing-sensitive synchronization for efficient secure multi-execution. In *Proceedings of the 2019 ACM SIGSAC Con-*

ference on Cloud Computing Security Workshop, CCSW'19, page 153–164, New York, NY, USA, 2019. Association for Computing Machinery

- Tobias Pfeffer, Thomas Göthel, and Sabine Glesner. Automatic analysis of critical sections for efficient secure multi-execution. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, pages 318–325, 2019

- Tobias Pfeffer, Paula Herber, Lucas Druschke, and Sabine Glesner. Efficient and safe control flow recovery using a restricted intermediate language. In *2018 IEEE 27th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 235–240, 2018

For the implementation, we use additional insights from the following paper.

- Konstantin Scherer, Tobias Pfeffer, and Sabine Glesner. I/o interaction analysis of binary code. In *2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 225–230, 2019

The thesis is structured as follows. First, in Chapter 2, we provide background information on confidentiality and on Secure Multi-Execution. We also define the semantics of our machine code model. We use this model, after discussing related works in Chapter 3, when we illustrate the problem addressed in this thesis in Chapter 4. We then demonstrate our solution in four steps. In Chapter 5, we show how we apply Secure Multi-Execution to machine code and demonstrate the effectiveness of our solution with the running example. We then introduce our Dynamic Instancing Optimization in 6 and our Bounding Optimization in Chapter 7. In each section, we also show the effect of our optimizations on the enforcement guarantees and efficiency. In Chapter 8, we introduce our timing-sensitive scheduling to improve the security guarantees of our Bounding Optimization. We close the main part of this thesis in Chapter 9, were we outline our contributions to static control-flow reconstruction of binary code. In the second part of this thesis, we present our evaluation results for benchmark examples in Chapter 10 and for real-world binaries in Chapter 11. Finally, we conclude the thesis and give pointers for future research in Chapter 12. The appendix contains more details on our evaluation and an extension for stream input.

Chapter 2

Background

Our goal in this thesis is to develop a practical method to enforce confidentiality across all components of a system, including machine code. Thus, in this section, we discuss the relevant aspects of confidentiality, enforcement, and machine code. For confidentiality, we provide a definition of *noninterference* [37] and discuss different notions of termination- and timing-sensitivity. We then introduce Secure Multi-Execution [32] as the enforcement method of our choice in depth. Finally, we introduce a model of machine code, including operational semantics, which we use to describe the threat model in the next chapter.

2.1 Confidentiality

Our aim is to ensure that a program cannot leak sensitive information through its behavior. To achieve this, we first need to express confidentiality as a property of programs. Thus, we first define a model of *deterministic interactive programs*, using the definitions from Clark and Hunt [24]. Based on this model, we then define *noninterference* as our confidentiality property, based on definitions from Rafnsson and Sabelfeld [65] and Devriese and Piessens [32]. We elaborate on the various levels of sensitivity with respect to the *termination* and *timing* behavior of the system, as well as the behavior of the *environment*. In the next section, we show how a termination- and timing-sensitive notion of noninterference can be enforced using Secure Multi-Execution. This chapter concludes with our introduction of a machine code model that will be used as reference in the rest of the thesis.

2.1.1 Security Model

In our definition of confidentiality, programs are treated as black boxes. This allows our definitions to define confidentiality independent of the program language and system definition. We assume that only the input and output behavior of the program is observable. Input to, and output from them is realized through abstract channels, collectively denoted $\mathbb{C}$. For most of this thesis, a channel $c \in \mathbb{C}$ represents a file on the system. Yet, it may also represent a network connection or another type of data stream.

To describe confidential communication, we need to equip each channel with a security level. The security level $\ell \in \mathcal{L}$ of a channel c describes the classification of the information that is obtained through channel c. We introduce a labeling function $\pi : \mathbb{C} \mapsto \mathcal{L}$, to assign security levels to channels. It also describes the classification of outputs produced on channel c. To support incomparable levels of security, they are usually partially ordered.

Definition 1 (Partial Order) *A partial order for set $\mathcal{L}$ is a relation $\sqsubseteq: \mathcal{L} \times \mathcal{L}$ that is reflexive (i.e. $\forall l \in \mathcal{L} : l \sqsubseteq l$), transitive (i.e. $\forall l_1, l_2, l_3 \in \mathcal{L} : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3$), and anti-symmetric (i.e. $\forall l_1, l_2 \in \mathcal{L} : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2$). A partial order is* total *if $\forall l_1, l_2 \in \mathcal{L} : (l_1 \sqsubseteq l_2) \vee (l_2 \sqsubseteq l_1)$.*

To describe combinations of incomparable levels, the partially ordered levels must additionally form a lattice [30, 69].

Definition 2 (Lattice) *A lattice $(\mathcal{L}, \sqsubseteq) = (\mathcal{L}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is a partially ordered set $(\mathcal{L}, \sqsubseteq)$ such that every pair of elements $l_1, l_2 \in \mathcal{L}$ have a least upper bound $\sqcup\{l_1, l_2\} \in \mathcal{L}$ and greatest lower bound $\sqcap\{l_1, l_2\} \in \mathcal{L}$. $\sqcup X$ is a least upper bound of a set $X \subseteq \mathcal{L}$ if*

1. *$\forall \ell \in X \,.\, \ell \sqsubseteq \sqcup X$, and*
2. *$\forall ub \in X \,.\, (\forall \ell \in X \,.\, \ell \sqsubseteq ub) \implies \sqcup X \sqsubseteq ub$.*

The definition of the greatest lower bound $\sqcap X$ is analogous.

This setup allows a simple definition of confidential programs. When information does not flow from channels with higher level to channels with lower level, then the program is secure [31]. As an example, we illustrate this definition in Figure 2.1. Here, we assume a simple two-level security lattice $(\mathcal{L}, \sqsubseteq)$ such that $\mathcal{L} = \{L, H\}$ and $\sqsubseteq = \{(L, L), (L, H), (H, H)\}$. Further we assume three channels,

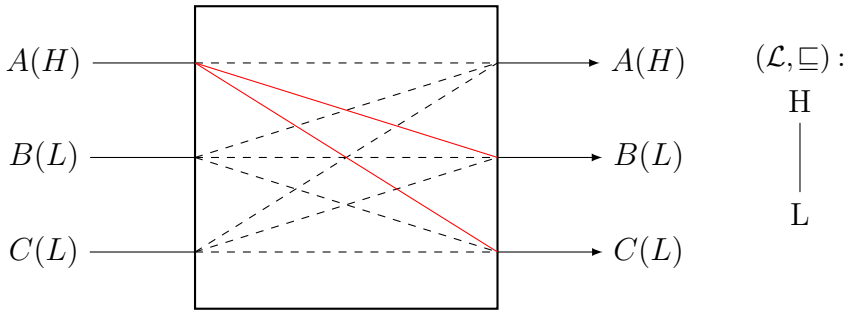


Figure 2.1: Confidentiality in our program model

$\mathbb{C} = \{A, B, C\}$, and security labeling $\pi = \{(A, H), (B, L), (C, L)\}$. Then, flows from channel A to channels B or C violate confidentiality. On the right, we illustrate the security lattice using a Hasse diagram. In a Hasse diagram, each security level is represented by a vertex in a two-dimensional graph. An edge is added between two levels ℓ_1 and ℓ_2 such that ℓ_1 is placed *lower* than ℓ_2, whenever $\ell_1 \leq \ell_2$ and there is no other ℓ_3 where $\ell_1 \leq \ell_3 \leq \ell_2$ would hold.

An important take away from our security model is that at least three parts are needed to violate confidentiality:

1. The program communicates with at least two channels.
2. The channels are equipped with at least two different security levels.
3. Information flows from a channel to another channel with lower security level.

Consequently, programs that communicate with only one channel are assumed to be trivially secure and are thus of no further concern to us. Additionally, we generally assume that at least two different levels of security are used. While the security lattice and mapping depends on the context, whether or not a violating flow exists depends on the program behavior. Thus, we describe a model of the behavior next.

2.1.2 Program Behavior

Since we treat programs as black boxes, we denote the internal state by the abstract state s, hiding the memory and register contents. We write $s_1 \neq s_2$ to describe two different states of a program, without qualifying the difference. To model progress in a black box program, we define its behavior as a transition system (TS).

Definition 3 ((Labeled) Transition System) *A transition system (TS) is a tuple $(S, \rightarrow)$ where S is a set of states and $\rightarrow \subseteq S \times S$ is a transition relation.*

We write $s \to s'$ to denote that the system transitions from state s to state s' in one step. A labeled transition system (LTS) is a TS with a set of labels L such that each transition is labeled. Thus, the labeled transition relation is defined as $\to \subseteq S \times L \times S$. We write $s \xrightarrow{\ell} s'$ do denote that the system transition from state s to state s' with label ℓ.

To express the observable input and output behvaior of deterministic interactive programs, we use the model introduced by Clark and Hunt [24]. The behavior of the program is described as an input-output labeled transition system (IOLTS).

Definition 4 (Input-Output labeled Transition System) *An input-output labeled transition system (IOLTS) is a labeled transition system (LTS) $(S, L, \to)$ where*

$$L := \bullet \mid c!v \mid c?v$$

Input of value v from channel c is described by the input event $c?v$, while output of v to c is denoted $c!v$. Unobservable internal transitions are denoted with the $\bullet$ event.

We then express input and output by annotating the transition between two states with the appropriate event. Thus, $s \xrightarrow{c!v} s'$ means that the system progresses from state s to state s' while emitting value v as output on channel c. $s \xrightarrow{c?v} s'$ means that the system progresses from state s to state s' while obtaining value v as input from channel c. Silent transitions are notated by the $\bullet$-action, e.g. $s \xrightarrow{\bullet} s'$. We omit the target state to denote that a transition is possible. For example, $s \xrightarrow{c?v}$ means that the system could progress from state s to some state while obtaining v from c.

We generally assume that the target programs are *input-neutral* and *deterministic*. Input neutral means that a program that accepts one input value from a channel also accepts any other value from that same channel. In other words, an input-neutral program cannot reject specific values as inputs on channels.

Definition 5 (Input-neutral IOLTS) *An IOLTS $(S, L, \to)$ is* input-neutral *if for all states $s \in S$ and all channels $c \in \mathbb{C}$*

$$\exists v \,.\, s \xrightarrow{c?v} \implies \forall v' \,.\, s \xrightarrow{c?v'}$$

$$
\begin{array}{ll}
(o:\bar{a})\restriction_? = \bar{a}\restriction_?, \text{ if } o \neq c?v & (\bullet:\bar{a})\restriction_\bullet = \bar{a}\restriction_\bullet \\
(c?v:\bar{a})\restriction_? = c?v:(\bar{a}\restriction_?) & (c!v:\bar{a})\restriction_\bullet = c!v:(\bar{a}\restriction_\bullet) \\
(o:\bar{a})\restriction_! = o:(\bar{a}\restriction_!), \text{ if } o \neq c?v & (\bullet:\bar{a})\restriction_\ell = \bullet:(\bar{a}\restriction_\ell) \\
(c?v:\bar{a})\restriction_! = \bullet:(\bar{a}\restriction_!) & (c\$v:\bar{a})\restriction_\ell = \bullet:(\bar{a}\restriction_\ell), \text{ if } \pi(c) \not\sqsubseteq \ell \\
(\bullet:\bar{a})\restriction_c = \bullet:(\bar{a}\restriction_c) & (c\$v:\bar{a})\restriction_\ell = c\$v:(\bar{a}\restriction_\ell), \text{ if } \pi(c) \sqsubseteq \ell \\
(c'\$v:\bar{a})\restriction_c = \bullet:(\bar{a}\restriction_c), \text{ if } c' \neq c & (\bar{a})\restriction_{\overrightarrow{\bullet}} = \varepsilon, \text{ if } \forall \bar{a}' : a:\bar{a}'' \leq \bar{a} \colon a = \bullet \\
(c'\$v:\bar{a})\restriction_c = c'\$v:(\bar{a}\restriction_c), \text{ if } c' = c & (\bar{a}':c\$v:\bar{a}'')\restriction_{\overrightarrow{\bullet}} = \bar{a}':c\$v:(\bar{a}\restriction_{\overrightarrow{\bullet}})
\end{array}
$$

Figure 2.2: Projection function definition

Determinism is often defined as allowing only one successor state for each state of the program. In an interactive systems, determinism allows multiple successors of a state, as long as the difference is due to different input. Note that the difference can only be in the input value, not in the input channel.

Definition 6 (Deterministic IOLTS) *An IOLTS is* $(S, L, \rightarrow)$ deterministic *if for all states* $s \in S$ *and all channels* $c \in \mathbb{C}$

1. $s \xrightarrow{a} s_1 \wedge s \xrightarrow{a} s_2 \implies s_1 = s_2$

2. $s \xrightarrow{a_1} \wedge s \xrightarrow{a_2} \wedge a_1 \neq a_2 \implies a_1 = c?v_1 \wedge a_2 = c?v_2 \wedge v_1 \neq v_2$

We write $s \xrightarrow{\bar{a}} s'$ to mean that the system progresses from state s to state s' in multiple steps, producing the trace (list) of actions $\bar{a}$. To compare different traces, we define a filter function $\restriction$ similar to the definition by Rafnsson and Sabelfeld [65]. The exact definitions are given in Figure 2.2. For example, we consider two traces $\bar{a}_1 = A?3:\bullet:\bullet:B!2:C!1:\bullet$, and $\bar{a}_2 = A?4:\bullet:B!2:C!2:\bullet:\bullet$. We additionally assume the security labeling introduced above, $\pi = \{(A,H),(B,L),(C,L)\}$. Then,

$$\bar{a}_1 \restriction_\bullet = A?3 : B!2 : C!1 \qquad \bar{a}_1 \restriction_A = A?3 : \bullet : \bullet : \bullet : \bullet$$
$$\bar{a}_2 \restriction_\bullet = A?4 : B!2 : C!2 \qquad \bar{a}_2 \restriction_A = A?4 : \bullet : \bullet : \bullet : \bullet$$
$$\bar{a}_1 \restriction_{\overrightarrow{\bullet}} = A?3 : \bullet : \bullet : B!2 : C!1 \qquad \bar{a}_1 \restriction_L = \bullet : \bullet : \bullet : B!2 : C!1 : \bullet$$
$$\bar{a}_2 \restriction_{\overrightarrow{\bullet}} = A?4 : \bullet : B!2 : C!2 \qquad \bar{a}_2 \restriction_L = \bullet : \bullet : B!2 : C!2 : \bullet : \bullet$$
$$\bar{a}_1 \restriction_? = A?3 \qquad \bar{a}_1 \restriction_H = \bar{a}_1$$
$$\bar{a}_1 \restriction_! = \bullet : \bullet : \bullet : B!2 : C!1 : \bullet \qquad \bar{a}_2 \restriction_H = \bar{a}_2$$

For brevity, we write $\bar{a} \restriction_{\ell,\bullet,c}$ to mean $((\bar{a} \restriction_\ell) \restriction_\bullet) \restriction_c$. We also write $\bar{a}_1 =_{\ell,\bullet,c} \bar{a}_2$ to mean $\bar{a}_1 \restriction_{\ell,\bullet,c} = \bar{a}_2 \restriction_{\ell,\bullet,c}$. Thus, for example

$$\bar{a}_1 \neq_B \bar{a}_2 \qquad \bar{a}_1 =_{?,L,\bullet} \bar{a}_2$$
$$\bar{a}_1 =_{B,\bullet} \bar{a}_2 \qquad \bar{a}_1 \neq_{?,H,\bullet} \bar{a}_2$$
$$\bar{a}_1 =_{\bullet,B} \bar{a}_2 \qquad \bar{a}_1 \neq_{!,L,\bullet} \bar{a}_2$$

Based on these definitions, we can give a trace-based characterization of confidentiality next.

2.1.3 Noninterference

The example above hints at a trace-based definition of confidentiality. Both example traces $\bar{a}_1$ and $\bar{a}_2$ obtain the same public inputs. Thus $\bar{a}_1 =_{?,L,\bullet} \bar{a}_2$ holds. Yet, they produce different public outputs. Where $\bar{a}_1$ emits a 1 on public channel C, $\bar{a}_2$ emits a 2. Since the programs are deterministic, this difference must be due to different private inputs. In other words, this means that private inputs *interfered* with public outputs. Consequently, *noninterference*, meaning that private inputs do *not interfere* with public outputs, can be used to define confidentiality [37].

Unfortunately, the definition of noninterference is not as simple as "same public inputs, same public outputs", as described by $\bar{a}_1 =_{?,L,\bullet} \bar{a}_2 \implies \bar{a}_1 =_{!,L,\bullet} \bar{a}_2$. Given this definition, an attacker could vary the number of public inputs, depending on the private inputs. Then, the premise of the implication would be falsified and thus the program would be considered secure under all circumstances. Hence, we must consider the *possible* public inputs instead of the *actual* public inputs in our definition.

We describe the possible inputs to a program collectively as the *environment* [24]. An environment $e : \mathbb{C} \mapsto i^*$ is a mapping from channels to lists of input events. Taking an input from channel c in environment e is denoted $e(c)$. The list of input events of channel c in environment e is denoted e^c. Given an environment e and a program in state s, we say that s performs trace $\bar{a}$ under environment e, denoted $e \models s \xrightarrow{\bar{a}}$, if for each channel the *actual* inputs in $\bar{a}$ form a prefix of the *possible* inputs in e^c: $\bar{a} \leq_{?,c,\bullet} e^c$. Two environments e_1 and e_2 are equivalent regarding level ℓ, written $e_1 =_\ell e_2$, if

$$\forall c \in \mathbb{C}\centerdot\ \pi(c) \sqsubseteq \ell \implies e_1^c = e_2^c$$

In the literature, it is generally assumed that these lists of input events represent streams or queues, such that drawing input from the environment removes said input from the list [24,32,65]. We also assume that access to input is non-blocking and that environments are total. Rafnsson and Sabelfeld extended the formal model with blocking access and nontotal environments to model attacks based on delay of private input [65]. These attacks do not play a prominent role throughout this thesis and are thus discussed in Appendix C.

Definition 7 (Progress-Sensitive Noninterference) *A program starting in state s is* progress-sensitively noninterferent *(PSNI) regarding level ℓ if*

$$e_1 =_\ell e_2 \wedge e_1 \models s \xrightarrow{\bar{a}_1} \wedge e_2 \models s \xrightarrow{\bar{a}_2} \implies \bar{a}_1 =_{!,\ell,\bullet} \bar{a}_2$$

This notion of noninterference is called progress-sensitive noninterference (PSNI), as it takes the order of outputs into account. With all silent events as well as outputs on channels with level not lower than ℓ removed, all that remains is a list of outputs to channels lower or equal ℓ. For these to be equivalent, the two programs must have produced the same ℓ-observable outputs in the same order, independent of potentially differing input from higher channels in the environments e_1 and e_2. Next, we discuss how information can be leaked through the termination behavior and what kind of termination behavior is covered by our definition of noninterference. We then discuss the even more elusive timing-attacks and extend our definition to express a form of weak timing-sensitivity.

Termination

Askarov et al. show that (non-)termination can be used to leak more than one bit [6]. They also provide an example to illustrate the problem, which we repeat

Listing 2.1: Leak through counting [6]

```
1 in(h, H);
2 for (l := 0) to maxNat {
3   out(l, L);
4   if (l == h) then { while true do skip };
5 };
```

in Listing 2.1. The program counts upwards in a public variable and emits the intermediate values on a channel that is publicly observable. By itself, this is not a security violation, as the counting is independent of the secret value stored in variable `h`. However, when the public counter reaches the secret value, the program diverges. Thus, no more values will be emitted on the public channel henceforth. This can be observed by unauthorized users, who can then infer the *exact* value of the secret.

Fortunately, this leak is covered by our definition of noninterference. Because different values of `h` would produce a different amount of public outputs, this program is not progress-sensitive noninterferent. Yet, it highlights another interesting property. Assuming that the secret value in `h` must be positive will *always* diverge. Programs that either always terminate or always diverge are considered *termination-sensitive*, as the termination behavior does not depend on inputs. Unfortunately, PSNI cannot ensure that programs are termination-sensitive, as it does not consider *actual* termination.

This is in accordance with the findings from Ngo et al., who show that termination-sensitivity is impossible to enforce [59]. Thus, they consider the security guarantees achieved by a system that enforces PSNI as *indirect* termination-sensitivity. An indirectly termination-sensitive program may or may not actually terminate after producing the last observable output, as longs as it will never produce another observable action. The reasoning being that an attacker cannot observe termination directly, but only indirectly through the observations of actions. Just as it is impossible to enforce direct termination-sensitivity due to the halting problem, it is also impossible for an attacker to judge whether a silent program has terminated or not.

The difference of direct and indirect termination-sensitivity is shown in Listing 2.2. In contrast to the termination-sensitive program from Listing 2.1, this program is not termination-sensitive. Only when the secret value in `h` coincides with the public value in `l` does it diverge. Otherwise it terminates. However, because no output is produced, it is trivially indirectly termination-sensitive.

Listing 2.2: Leak through (non-)termination

```
1 in(l, L);
2 in(h, H);
3 if (h == l) then {
4   while true do skip;
5 };
```

The program also highlights another interesting point. While for the program shown in Listing 2.1 it is possible to define one value for `h` to ensure termination, this is not the case for the program in Listing 2.2. Here, termination depends on the choice of both `l` and `h` and therefore no static value for `h` can be provided such that the termination behavior is independent of `l`. This is an important distinction for enforcement systems that replace sensitive information with dummy values, such as Secure Multi-Execution. Algehed and Flanagan introduced the term *monotonically* termination-sensitive to describe this difference [2]. In a monotonically termination-sensitive program, higher inputs can be statically replaced by a fixed value without altering the termination behavior. Thus, Listing 2.1 is monotonically termination-sensitive while Listing 2.2 is not. Algehed and Flanagan show that Secure Multi-Execution can enforce monotonically termination-sensitivity, thus such programs are the focus of this thesis.

Timing

Progress-sensitive noninterference considers the order of outputs, but does not describe leaks due to timing behavior. Due to the $\bullet$-filter, all internal events are ignored. Thus, delays due to intermediate computation are also ignored. However, when these delays depend on sensitive information, then information can leak through the timing of public outputs. As an example, we consider the programs shown in Figure 2.3.

Both programs in Figure 2.3 are timing-insensitively noninterferent. The private input from channel H, stored in variable `h`, does not flow to variable `l`, which is eventually emitted as output on the public channel L. Yet, the timing of the output differs. In Figure 2.3a, the output is delayed by the duration of the execution of one `skip` statement, depending on the sensitive information in `h`. As an example, we assume two environments e_1 and e_2. We further assume that $e_1^H = H?1$ and $e_2^H = H?2$. Then,

```
1 l := 0;
2 in(h, H);
3 if (h % 2 == 0) {
4   skip; skip;
5 } else {
6   skip;
7 }
8 out(l, L);
```

(a)

```
1 l := 0;
2 in(h, H);
3 if (h % 2 == 0) {
4   a := b;
5 } else {
6   a := c;
7 }
8 out(l, L);
```

(b)

Figure 2.3: Examples of weak and strong timing-sensitive programs [42]

$$e_1 \models s \xrightarrow{\bar{a}_1 = H?1:\bullet:L!0}$$
$$e_2 \models s \xrightarrow{\bar{a}_2 = H?2:\bullet:\bullet:L!0}$$

Here, $e_1 =_\ell e_2 \wedge e_1 \models s \xrightarrow{\bar{a}_1} \wedge e_2 \models s \xrightarrow{\bar{a}_2} \implies \bar{a}_1 =_{!,\ell,\bullet} \bar{a}_2$ holds, as the differing number of $\bullet$ events are filtered out. However, an unauthorized observer of the public channel can infer whether the private input is even or odd, based on the timing of the public output.

This can be described using *weak timing-sensitive noninterference* (TSNI), where execution of any instruction takes a fixed amount of time. The weak timing model can be expressed by counting the number of silent events in a trace, up to the last output.

Definition 8 *Weak Timing-Sensitive Noninterference A program starting in state s is* weakly timing-sensitive noninterferent *(TSNI) regarding level ℓ if*

$$e_1 =_\ell e_2 \wedge e_1 \models s \xrightarrow{\bar{a}_1} \wedge e_2 \models s \xrightarrow{\bar{a}_2} \implies \bar{a}_1 =_{!,\ell,\overrightarrow{\bullet}} \bar{a}_2$$

The exemplary traces above are not TSNI, as

$$H?1:\bullet:L!0 =_{!,L,\overrightarrow{\bullet}} \bullet:\bullet:L!0 \neq \bullet:\bullet:\bullet:L!0 =_{!,L,\overrightarrow{\bullet}} H?2:\bullet:\bullet:L!0$$

This model protects against leaks on the timing-channel in an idealized setting. It does not take delays based on the architecture, pipeline, caches, process switching and more into account. This is exemplified in Figure 2.3b. Because all instructions are assumed to take the same amount of time in the weak timing-model,

this program is weak timing-sensitively noninterferent. Yet, the instructions `a := b` and `a := c` may actually take a different amount of time. For example when it can be assured that the value of `b` is cached while the value of `c` is not.

Throughout the thesis, we use the weak timing-sensitive model in our formalization, but measure real values in our evaluations. Thus, on the one hand, our measured timing-sensitivity is stronger than the weak model presented here. On the other hand, timing measurements on actual systems are always jittery due to contention for resources. Thus, we aim to show non-correlation with sensitive inputs in our experiments.

2.2 Secure Multi-Execution

Secure Multi-Execution (SME) was first introduced by Devriese and Piessens in their seminal paper on noninterference enforcement [32]. What makes Secure Multi-Execution a promising confidentiality enforcement method is that it combines some of the best characteristics at once. As a multi-execution enforcement mechanism, it offers the best achievable termination-sensitivity [2, 16]. As a scheduling enforcement mechanism, it is timing-sensitive [42, 65]. As a black-box enforcement mechanism, it can be applied without analysis [2, 65]. These results show that an enforcement mechanism based on Secure Multi-Execution can satisfy our criteria of security, transparency.

Secure Multi-Execution enforces noninterference *by design* [32]. A potentially insecure program is transformed such that the resulting program *must be* secure. This is achieved by restricting the access to input from private channels. A program that has no access to private input is trivially secure, as absent information cannot interfere with public outputs. Yet, this restriction also thwarts transparent enforcement. If the private inputs are never obtained, any private output that depends on them cannot be produced by the program. This would violate our requirement that the same secure outputs are produced under enforcement, compared to native execution. Thus, another execution is added under SME, which is restricted in its output behavior. This second execution has access to the private inputs, but is not allow to produce output for public channels. Consequently, the private inputs in this second execution also cannot interfere with public outputs.

We visualize the concept in Figure 2.4. Here, we assume a two-level lattice with $L \sqsubset H$. Through SME, the program is executed twice. One execution is responsible for output on the lower level, one for output on the higher level. Both executions have access to the low input, as flows in the upward direction of the lattice are permitted under confidentiality. Yet, only the higher execution has access to high

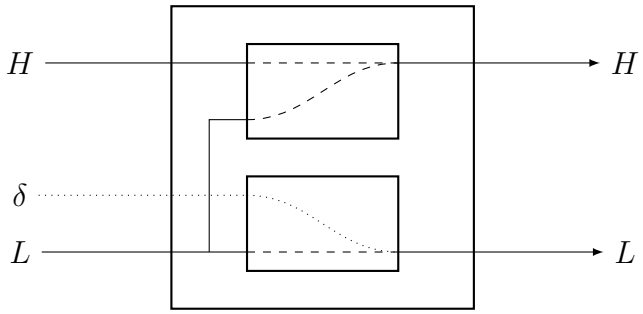


Figure 2.4: Secure Multi-Execution with two levels

inputs. Where the lower execution requires high input, it is fed unclassified input from a dummy source δ instead. Responsibilities for output on channels are split among the two executions. The higher execution produces output only for channels at high security level, the lower execution only for channels at low security level. Thus, all original flows remain unchanged, only insecure flows from higher to lower levels are replaced by secure flows from dummy input.

Naturally, the security lattice might be more complex in a real-world scenario. Yet, the principle remains the same. An execution is created for each level in the security lattice. Each execution is responsible for output to channels with equal security level. Input is restricted to channels with equal or lower level. Thus, all legal flows remain unchanged but insecure flows are removed.

The technical requirements to implement SME enforcement follow from its design. First and foremost, a method is required to intercept input and output operations of the target. Otherwise, input and output operations could not be restricted. Second, a method is required to identify and classify channels targeted by input and output operations. This is necessary to decide when restrictions apply. Third, a method is required to redirect input to a dummy source. And fourth, a method is required to suppress output to channels. We formalize these requirements when we discuss the *local* semantics of SME in Section 2.2.1. Additionally, when input has observable side-effects or is volatile, a method to copy input to higher execution is necessary. We discuss these additional requirements in Appendix C.

As mentioned above, Secure Multi-Execution can also be used to thwart leaks through the timing behavior. This is not a consequence of the transformation of each individual execution, but a consequence of the scheduling of these executions. For example, we could first run the lower executions to termination, before the higher execution is scheduled. This way, it is guaranteed that the termination- or timing-behavior of the higher execution cannot interfere with the termination

$$\textsc{Silent}\ \frac{s \xrightarrow{\bullet} s'}{\ell, \pi, \delta \vdash s \xrightarrow{\bullet} s'}$$

$$\textsc{Input-T}\ \frac{s \xrightarrow{c?v} s' \qquad \pi(c) \sqsubseteq \ell}{\ell, \pi, \delta \vdash s \xrightarrow{c?v} s'}$$

$$\textsc{Input-F}\ \frac{s \xrightarrow{c?v} s' \qquad \pi(c) \not\sqsubseteq \ell \qquad s \xrightarrow{c?\delta} s^{\delta}}{\ell, \pi, \delta \vdash s \xrightarrow{\bullet} s^{\delta}}$$

$$\textsc{Output-T}\ \frac{s \xrightarrow{c!v} s' \qquad \pi(c) = \ell}{\ell, \pi, \delta \vdash s \xrightarrow{c!v} s'}$$

$$\textsc{Output-F}\ \frac{s \xrightarrow{c!v} s' \qquad \pi(c) \neq \ell}{\ell, \pi, \delta \vdash s \xrightarrow{\bullet} s'}$$

Semantics 2.1: Local Secure Multi-Execution semantics

or timing of the lower execution. Unfortunately, this simple strategy may lead to intransparencies and is insufficient when incomparable levels exist in the security lattice. Thus, we discuss more complex scheduling in Section 2.2.2, when we introduce the *global* semantics of SME.

2.2.1 Local Semantics

The local semantics of Secure Multi-Execution are given in Semantics 2.1. An individual execution consists of the abstract state s, just like an unprotected execution. Yet, it is executed in the context of a static security label $\ell \in \mathcal{L}$, a channel classification $\pi : \mathbb{C} \mapsto \mathcal{L}$, and dummy value δ. Since this is a black-box definition, semantics of internal transitions (denoted by $\bullet$ actions) are unaltered. Similarly, input from channels with security classification of equal or lower level as well as output to channels with equal security classification are unchanged. If, however, input from channels with higher security classification is requested, we instead provide the dummy value and progress to the corresponding state s^{δ}. This state must exist, as we assume that programs are *input-neutral*. Similarly, if output to channels at different security classification is requested, it is suppressed, making it appear as a silent action. This alteration is hidden from the program such that we still progress to state s'.

Given an execution at level ℓ in state s that contains no information with classification higher than ℓ, it is clear that this property is retained during ex-

ecution. Due to the interactive nature of the program, the only way to introduce ℓ-information to s would be through input from a channel with classification higher than ℓ, resulting in violating state s'. However, in this case, the execution progresses to s^{δ} which cannot contain classified information as it obtained the unclassified dummy value δ, instead of the classified value v.

Furthermore, since only the ℓ-execution is allowed to produce output on channels with that same classification and only one ℓ-execution exists, output on these channels is produced only by this ℓ-execution. In a secure program, where replacement of inputs from channels with classification higher than ℓ does not alter the outputs on ℓ-channels, it follows that the ℓ-execution produces the exact same output. Thus, the minimum of termination- and timing-insensitive transparency is achieved.

2.2.2 Global Semantics

From the local semantics it follows that each execution guarantees noninterference with respect to its level. This, however, does not mean that any combination of the executions also guarantees noninterference. To illustrate this, consider a scenario with the typical two-level lattice such that $L \sqsubset H$. Following the semantics of Secure Multi-Execution, this gives rise to two executions, one for level L, one for level H. If we now where to combine both executions such that we first execute the H execution and then the L execution, if and when the L would be executed would then be dependent on the H execution. This dependency could then be exploited by an attacker to leak information about the classified inputs in the H execution. Thus, correct scheduling is vital for the timing- and termination-sensitivity of Secure Multi-Execution.

In the original work by Devriese and Piessens, a *low-priority* scheduling strategy is used [32]. In this scheduling strategy, executions are scheduled in ascending order. Thus, one execution starts running when all executions with lower priority have run to completion. Naturally, this is problematic when the lattice is non-linear, such that incomparable levels exist. To be able to define an ascending order, a partially ordered lattice would have to be extended to become a total order. However, any such extension inevitably leads to a situation where scheduling of one execution depends on the timing and termination behavior of another execution at originally incomparable level.

Kashyap et al. discuss this and alternative scheduling strategies in more detail [42]. They show that, to be secure even under very strong assumptions such as no real parallelism, shared finite resources, and perfect timing capabilities of the attacker, executions must run to completion before executions with higher clas-

$$\textsc{Step}\ \frac{s = S(\ell) \qquad \ell, \pi, \delta \vdash s \xrightarrow{a} s'}{\pi, \delta \vdash \ell : \sigma, S \xrightarrow{a} \sigma, S[\ell \leftarrow s']}$$

Semantics 2.2: Global Secure Multi-Execution semantics

sification can be executed. Consequently, none of their strategies allows for such strong guarantees between incomparable levels. They do note, however, that a weaker timing model, where execution times of instructions are uniform and resources are not shared, can be achieved by round-robin scheduling of all executions. Rafnsson and Sabelfeld generalize this further and show that any deterministic and fair scheduler achieves weak timing guarantees [65]. A scheduler σ is an (infinite) list of security levels. It is fair when it schedules all ℓ-runs infinitely often. As an example, Rafnsson and Sabelfeld propose round-robin schedulers.

Secure Multi-Execution is than instantiated for a program starting in state s by cloning the state as often as there are levels. We then use an execution mapping, denoted S, that maps each security level to the corresponding execution state. As shown in the Step rule from our global semantics in Semantics 2.2, the head of the scheduler σ describes the next level to be executed. The state of the corresponding execution is given by $S(\ell)$ and leads to the next state s'. The mapping is than updated accordingly, by setting s' as the value mapped to by ℓ. Any event a performed by the execution is visible from the global view.

2.2.3 Transparency

Apart from the timing-sensitive security guarantees, Secure Multi-Execution (SME) also provides strong transparency [2, 16, 65, 89]. Transparent enforcement means that the behavior of secure components remains the same. Secure Multi-Execution achieves this as the internal computations of the target program are unchanged. Thus, any output produced independent of sensitive input would still be produced under enforcement, where the sensitive input is replaced by dummy values. Naturally, full transparency cannot be assured for components that are leaking information, as that would require the enforced solution to recreate the leak, thwarting the security efforts.

Due to the individual scheduling of individual executions and replacement of sensitive inputs, it may be that secure outputs are produced earlier or later than normal. Thus, it may be that the order of secure outputs from executions on different levels may be changed. Consequently, Secure Multi-Execution does not guarantee transparency across all levels, but rather *per-channel transparency*. This

means that, on each level, the secure outputs are produced with the same contents and the same order.

Additionally, because the outputs produced for the highest security level are always secure, *top-level transparency* can be assured in any case. Thus, the outputs produced on the highest level are always the same in content and order as an unenforced execution of the target would produce. Yet, because of the scheduling necessary to break leaks on the timing-channel, SME cannot assure that outputs are produced at the same time as for unenforced execution. Thus, no timing-sensitive transparency guarantees can be given.

2.3 Machine Code

The targets of our enforcement mechanism are software components in compiled form. More precisely, we target the wide-spread x86_x64 architecture. Yet, this architecture has more than 1000 different instructions and thus far too many to be described fully here. Instead, we provide a minimal machine code language here, similar to the SimpIL language provided by Schwartz et al [68]. The language is also similar to intermediate languages used by binary analysis tools such as the BIL language of BAP [18].

The machine code model highlights two important aspects that make static analysis of binary code very hard [54]. First, instead of abstract variables that are assumed to be uniquely identified by their name, our model uses a memory that maps addresses to values. Second, the control-flow of machine code programs is much less obvious than it is in source-code programs.

Since addresses are a subset of the possible values, this allows multi-level pointers and pointer arithmetic. It also means that any value could be used a pointer, implying that all values must be considered pointers a well. Consequently, it is crucial for static analysis of binaries to combine flavors from numerical analysis and pointer analysis in a single analysis [10]. Note that we also use variable-like registers in our model, but assume that their number is very limited. For example, the x86_x64 architecture has only 16 general purpose registers. Thus, high-level variables are usually mapped to memory, not to registers.

Furthermore, machine code is unstructured, with conditionals and loops implemented through goto-like jump statements. It also lacks abstractions such as functions or scopes, which are hard to recover [4]. Even more problematic, however, is that the targeted code address of jump statements may depend on data flow. Indirect jump statements allow targets to be computed at run-time, with adverse implications for static control-flow analysis. Similar to how static binary

$\mathbb{A} \subseteq \mathbb{N}$	Addresses
$\Sigma : \mathbb{A} \rightharpoonup Inst$	Mapping from address to instruction
$\rho : \mathbb{N} \rightharpoonup \mathbb{V}$	Mapping from registers to values
$\mu : \mathbb{A} \rightharpoonup \mathbb{V}$	Mapping from addresses to values
$pc \in dom(\Sigma)$	Program counter
$\iota \in Inst$	Instruction

Semantics 2.3: Machine Code State

analysis must track pointers and numeric values at the same time, control-flow recovery requires data-flow analysis at the same time. Since data-flow analysis requires a control-flow graph to work on, this has been described as a "chicken and egg"-problem [43].

Next we introduce our machine code semantics in detail. We also use these semantics in Chapter 4, to provide an example of a compiled target. Finally, we introduce control-flow integrity as an increasingly popular mechanism to protected against control-hijacking attacks. We use this information again, when we discuss our contributions to static analysis in Chapter 9.

2.3.1 Semantics

A state in our machine code model is describes as a tuple $\langle \Sigma, \rho, \mu, pc, \iota \rangle$, where Σ is a program-specific mapping from code addresses to statements, which are usually called instructions in this context. The currently executed code address is stored in the program counter *pc*. The currently executed instruction is given by ι. Thus, $\Sigma[pc] = \iota$ describes the instruction at the current location given the program text Σ. We assume that code addresses can be any value, but access to an undefined location in Σ leads to abnormal termination of the program. We use $\xrightarrow{a}$ to denote that the program progresses to a new state with event a, the $\Downarrow$ to denote evaluation of an expression. Static inputs to the evaluation are written on the left side of the $\vdash$. We use two different stores in our model. The *register store* ρ maps register names to values, similar to high-level variable stores. The *memory store* μ maps values to values. Updates to the store are written using the $\leftarrow$ symbol, e.g. $\rho[x \leftarrow v]$ means the value of x in ρ is updated to v. Values are looked up using $\rho(x)$, which gives the value assigned to x in ρ.

The difference between the stores is reflected on instruction level. Registers are updated by assignment statements, where the left operand is a valid register name. The right operand of the assignment is an expression that evaluates to the new value for the updated register. Register names can also be used in any expression,

Expressions:

$$[\textsc{Const}]\frac{c \in \mathbb{V}}{\rho, \mu \vdash c \Downarrow c} \qquad [\textsc{Var}]\frac{v = \rho(x)}{\rho, \mu \vdash x \Downarrow v}$$

$$[\textsc{Load}]\frac{\rho, \mu \vdash e \Downarrow v \qquad v' = \mu(v)}{\rho, \mu \vdash [e] \Downarrow v'}$$

$$[\textsc{Op}]\frac{\rho, \mu \vdash e_1 \Downarrow v_1 \qquad \rho, \mu \vdash e_2 \Downarrow v_2 \qquad v = v_1 \oplus v_2}{\rho, \mu \vdash e_1 \oplus e_2 \Downarrow v}$$

Instructions:

$$[\textsc{Assign}]\frac{\rho, \mu \vdash e \Downarrow v \qquad \rho' = \rho[x \leftarrow v] \qquad \iota = \Sigma[pc+1]}{\Sigma \vdash \rho, \mu, pc, x := e \xrightarrow{\bullet} \rho', \mu, pc+1, \iota}$$

$$[\textsc{Store}]\frac{\rho, \mu \vdash e_1 \Downarrow v_1 \qquad \rho, \mu \vdash e_2 \Downarrow v_2 \qquad \mu' = \mu[v_1 \leftarrow v_2] \qquad \iota = \Sigma[pc+1]}{\Sigma \vdash \rho, \mu, pc, [e_1] := e_2 \xrightarrow{\bullet} \rho, \mu', pc+1, \iota}$$

$$[\textsc{Conditional-T}]\frac{\rho, \mu \vdash b \Downarrow 1 \qquad \rho, \mu \vdash e \Downarrow v \qquad \iota = \Sigma[v]}{\Sigma \vdash \rho, \mu, pc, \texttt{jcc}\ (b)\ e \xrightarrow{\bullet} \rho, \mu, v, \iota}$$

$$[\textsc{Conditional-F}]\frac{\rho, \mu \vdash b \Downarrow 0 \qquad \iota = \Sigma[pc+1]}{\Sigma \vdash \rho, \mu, pc, \texttt{jcc}\ (b)\ e \xrightarrow{\bullet} \rho, \mu, pc+1, \iota}$$

Compound Instructions:

$$\texttt{jmp}\ e ::= \texttt{jcc}\ (0 == 0)\ e$$

$$\texttt{call}\ e ::= \begin{cases} [sp] := pc+1 \\ sp := sp-1 \\ \texttt{jmp}\ e \end{cases} \qquad \texttt{return} ::= \begin{cases} sp := sp+1 \\ \texttt{jmp}\ [sp] \end{cases}$$

Semantics 2.4: Machine Code Semantics

which leads to a lookup of their value in the register store during evaluation. Memory is updated using specific store statements, where the left operand is an expression that evaluates to a memory address. The right operand evaluates to the new value that the memory address subsequently maps to. Memory can be referenced in expressions by providing an expression that evaluates to a memory address. Further expressions allow to provide constants and to compute new values using typical binary and unary operators.

For both the assignment and store statements, the control flow is transferred to the next instruction, meaning the instruction at the next code location. Thus, in these cases, the program counter *pc* is increased by one and the next instruction ι' is loaded from the program text at the next location, $\iota' = \Sigma[pc+1]$. Branching and more complex control-flow transfers are realized with the conditional jump instruction. The conditional jump instruction takes an expression as the first operand that decides whether or not the jump is taken. If the expression evaluates to zero, the jump is *not* taken and the next instruction is loaded instead. If the expression evaluates to 1, the jump is taken. Then the second operand expression is evaluated and control is transferred to the resulting address. Note that we assume that typical comparison operators such as `==` exist and adhere to the logic of the language (i.e., $4 == 5$ evaluates to 0, while $4 == 4$ evaluates to 1). Thus, the conditional jump instruction encapsulates both the mechanic to express structured code in machine code as well as the possibility for computed jumps.

Finally, we add three additional instructions as practical abbreviations for multiple instructions. First, the unconditional jump instruction is a conditional jump instruction with a tautological condition expression such that the jump is always taken. We assume that such an unconditional jump can be trivially detected such that it raises no adverse effects for branch prediction or target feasibility assumptions in control-flow recovery [87]. Second, we add `call` and `return` statements that transfer to a code block and back towards the caller. Here, we assume that a special *stack pointer* register *sp* exists and that it points to a memory region that is unaffected by other memory operations. Thus, when a `call` instruction is follow by a `return` instruction, we assume that the `return` instruction transfers control back the instruction following the `call`.

2.3.2 Data Flow

A fundamental difference between machine code and high-level code is the limited availability of variable-like registers. A register, similar to a high-level variable, is a unique storage element that can only be updated using named assignments. This has the great benefit that for any data flow analysis, it can be safely assumed that

```
p := x                    [p] :=  x
y := q                     y  := [q]
```

Figure 2.5: Explicit flow through pointers

all registers that are not the target of an assignment remain unchanged. Thus, assignments can generally be assumed to be free from side effects, which enables fast and precise data-flow analysis. In contrast to this, updates of the memory store are not side-effect free.

Which part of the memory is updated is specified by an arithmetic expression. Therefore, the result might depend on other variables. Thus, which part of the memory is altered by the update, cannot generally be determined in isolation. As any arithmetic expression can be used, any update may potentially change any part of the memory, not restricted to live references. Even worse, in the case of multi-level pointers, the target of an update may depend on the result of a load instruction, whose target may depend on other variables. Unfortunately, since registers require complex hardware support, processors typically have fewer than 20 general purpose registers. Consequently, abundant high-level variables are usually mapped to memory regions and updated via stores instead of assignments. Practically, this means that all variables are promoted to pointers and all n-level pointers are promoted to $n+1$-level pointers.

To demonstrate the added complexity due to pointers, we consider the code snippet in Figure 2.5. The code on the left uses no pointers. Thus, it is trivial to determine that information from `x` does not flow to `y`, because `p` and `q` are different variables. The assignment of `p` has no effect on the value of `q` and thus the information in `x` has no effect on the information in `y`. This is a result from the assumption that different identifiers describe a different location.

This is not true for the code on the right. Here, the value of `x` is stored at the address described by the value of `p`. Then, the value stored at the address described by the value of `q` is assigned to the variable `y`. Due to the semantics of the load and store instructions, it is possible that the update of the value at address `p` affects the value stored at `q`. Thus, information may flow from `x` to `y`. This can only be ruled out if the *feasible* values of `p` and `q` at this part of the program can be determined. Only if `p` can never have the same value as `q` can we rule out a flow of information.

This illustrates that the wide-spread use of arithmetic, multi-level pointers in machine code severely complicates information flow analysis. We outline existing approaches and their limitations in more detail in Section 3.3. Here, it is sufficient

to state that safe and reasonably precise analyses come with great complexity. As we show next, the complication of data-flow analysis in machine code affects other areas of analysis as well, as both control-flow recovery and system call semantics depend on it.

2.3.3 Control-Flow Integrity

An important consequence from the machine code semantics is the conflation of control and data flow. As shown in Semantics 2.4, most instructions are followed by the succeeding instructions. Yet, the branching instruction allows to transfer the control flow to a computed target, dependent on the current state. This has two closely related consequences. First, precise static analysis of the possible control flow in an binary requires precise data-flow analysis, which, as described above, is infeasible. Second, it means that user data can interfere with the control flow, violating the integrity of the control flow. The latter is of great importance to computer security and is the target of techniques such as control-flow integrity (CFI) which we describe here. As it happens, these mechanism can also be used to solve the first problem, as we show when we introduce our heuristic for efficient control-flow recovery in Section 9.2.

As the name suggests, control-flow integrity is a technique to ensure that user input does not interfere with the control information of an execution. This is possible, wherever control-flow transfer depends on the program state, as is the case in indirect calls and function returns. As outlined above, an indirect call transfers control to the result of a target expression, evaluated in the current machine state. It may thus depend on user information, allowing the user to stage control-hijacking attacks. Conversely, the return instruction transfers control to the value on top of the stack at the current machine state. In a benign program, the top of the stack contains the address of the next instruction after the most recent call. However, since this information is stored in a writeable memory area, it can be overwritten, most notably by buffer-overflow attacks [80].

To counter these attacks, Abadi et al. proposed to enforce control-flow integrity [1]. They achieve the enforcement by instrumenting call and return sites with bit patterns, and a dynamic check that ensures that each control is transferred only between valid patterns. More concretely, the patterns describe equivalence classes of targets. Two destinations are considered equivalent when they can be reached from the same set of sources. Consequently, with these targets enforced at run time, malicious redirection of the control flow is only possible along predefined paths. This reduces the attack surface dramatically.

An example of CFI is given in Figure 2.6. Here, `sort2` sequentially applies the

```
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```

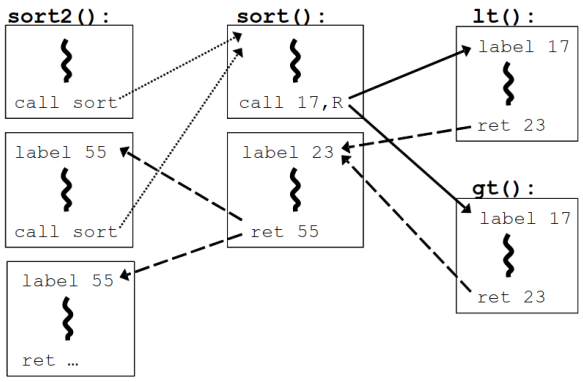


Figure 2.6: Control-Flow Integrity [1]

`sort` function to two lists. The `sort` function is implemented as a higher order function, taking a ordering function as an argument. For list `a`, `lt` is used, for list `b`, `gt`. Consequently, the functions `lt` and `gt` are called from the same source (the `sort`) function and thus labeled equivalently. Conversely, the call site in the `sort` function is labeled to ensure that `lt` and `gt` must return to it. The call to `sort` in the `sort2` function remains unchecked as it can be implemented using secure direct calls. However, the return from `sort` is checked, as it may return to two different locations.

To ensure valid flows, CFI requires a call graph as input that describes legal indirect control transfers. Recent advances in source-level pointer analysis allow fast approximation of call graphs during compilation [49], which in turn has lead to the first practical CFI implementations [83]. Challenges remain, as the approximate nature of the CFI enforcement leaves sufficient room to evade the technique [33].

2.4 Summary

In summary, Secure Multi-Execution is an enforcement method that guarantees progress-sensitive noninterference and, beyond that, also weakly timing-sensitive noninterference. By implication, it also guarantees indirect termination-sensitive noninterference, with direct termination-sensitive noninterference shown to be impossible to enforce. Consequently, it achieves among the strongest security guarantees of current information flow control approaches. Furthermore, it achieves per-channel transparency for secure behavior and top-level transparency in all cases. Thus, it allows to guarantee confidentiality without breaking the functionality of secure components.

Secure Multi-Execution also does not require changes to the target program. Instead, only the input and output behavior is transformed. Hence, it can be used to enforce secure even for targets that are not thoroughly comprehended. This makes it an attractive solution in our scenario, as static analysis of machine code is highly non-trivial. Because of the ubiquitous use of multi-level arithmetic pointers and lack of function scopes, precise information flow analysis quickly becomes infeasible. Unfortunately, due to the existence of computed branching instructions in low-level languages, control flow conflates with data-flow at various points in the code. This raises the related security issue of control-flow integrity, which can be thwarted by instrumenting indirect call sites.

The results discussed here point us in the direction of our work. Because we aim for a transparent enforcement of timing-sensitive noninterference for low-level code, we focus on Secure Multi-Execution. This allows us to side-step the challenges associated with low-level static analysis and achieve a general enforcement method. Yet, because Secure Multi-Execution comes with high performance overhead, we aim to reuse what can be learned about the information flows in the target to apply the enforcement only where necessary.

In the next chapter, we discuss related approaches from information flow control and low-level analysis. Then, in Chapter 4, we use our machine code model to introduce an exemplary target and corresponding attacks. Based on these, we show in Chapter 5 how Secure Multi-Execution can be applied to low-level targets to thwart these attacks. In the following chapters, we focus on optimizations to reduce the overhead of the enforcement. In Chapter 9 we finally discuss out contributions to control-flow recovery, before providing evaluation results for our wok in the second part of this thesis. We conclude with pointers to future work in Chapter 12.

Chapter 3

Related Work

In this section, we discuss works related to ours. For more clarity, we separate them into three groups. First and as the most relevant, we discuss enforcement solutions that use some kind of multi-execution. Then, we discuss language-based enforcement solutions that make up much of the earlier research on confidentiality enforcement. These solutions often focus on higher-level languages, therefore we discuss approaches focusing on binary analysis in a separate, third group. Our research shows that while there are closely related practical approaches for low-level systems such as TightLip or shadow execution, they do not achieve the timing-sensitive noninterference guarantees of Secure Multi-Execution and do not make use of static analysis to increase the enforcement efficiency.

3.1 Multi-Execution Enforcement

Our solution is based on the seminal paper by Devriese and Piessens on noninterference [32]. In this work, the authors propose Secure Multi-Execution (SME) as the technique to enforce timing-sensitive noninterference for interactive and deterministic programs in a transparent way. They also provide an experimental implementation in form of a Javascript engine, and provide benchmark results to demonstrate the effectiveness and practical feasibility of the concept. Furthermore, De Groef et al. later integrated the technique into the confidential web browser FlowFox [28]. With their work, the authors show that Secure Multi-Execution can be used in practice, albeit with some overhead.

The original definition of SME uses a low-priority scheduler, such that executions are scheduled in ascending order of their respective security level. This leads to two problems. First, Devriese and Piessens assume that security levels are

totally ordered. As Kashyap et al. show, their low-priority scheduler is not timing-sensitive between incomparable levels [42]. If neither is lower than the other, than the low-priority scheduler has to choose either, which might lead to a interferent delay in the outputs of the other. Second, in the case of reactive programs where executions do not terminate, lower levels have to be scheduled after higher levels eventually. This might lead to leak on the timing-channel, as Rafnsson and Sabelfeld describe for the FlowFox browser [65]. To remedy the problems, Rafnsson and Sabelfeld propose the use of a fair interleaving scheduler. Kashyap et al. show that this strategy actually achieves weak timing-sensitive noninterference for any security lattice. Yet, they note that no scheduling can achieve strong timing-sensitive noninterference between incomparable levels, as this stricter notion also takes effects such as caching, pipelining, etc. into account.

From the various works that extend the work by Devriese and Piessens, the most related to our work are those from Cormac Flanagan. Inspired by SME, Austin and Flanagan proposed the use of values with *multiple facets* (MF) to simulate computations from multiple executions on a single value [8]. By allowing a single variable to represent different values for different security classifications, they can replace multiple executions with fewer, more powerful executions, which increases the overall enforcement effectiveness. In their overview paper, Bielova and Rezk compare the security and transparency guarantees from different enforcement approaches, including SME and MF [16]. They conclude that SME is more secure and more transparent than MF.

Schmitz et al. later improve on their method, with the introduction of *faceted Secure Multi-Execution* (FSME) [71]. FSME aims to use efficient multiple facets enforcement as long as possible and only falls back on more expensive Secure Multi-Execution when one execution branch seem to have diverged. Thus, their approach improves the termination-sensitivity guarantees compared to the multiple facets approach, while being more efficient than pure SME.

Our optimizations are inspired by the FSME approach, but differ in the realization. While Schmitz et al. target programs written in Haskell, we focus on programs in machine code. Consequently, we cannot make the same adaptions to the language semantics as the authors of FSME, as that would imply changes to the hardware. Additionally, FSME comes with a simple merging mechanism for executions from different branches. On the one hand, Haskell is a structured languages, which simplifies the definition of control-flow merge points in the programs. Furthermore, where the values of two variables differ in two merging states, they can be replaced by a faceted value representing both options. In contrast to this, we require static analysis to define merge points of the control flow, as machine code is unstructured. Even worse, we cannot simply merge two states as

machine code does not support faceted values. Thus, we have to decide on one of the two states to continue execution, as we discuss more thoroughly in Chapter 7. More recently, Algehed et al. proposed more optimizations for the faceted Secure Multi-Execution approach [3]. Yet, as these again require support of faceted values, they do not apply in our scenario. Interestingly, they independently propose a mechanism to remove unnecessary views that uses developer knowledge similar to our bounding optimization. In their most recent work from this year, Algehed and Flanagan prove that transparent information flow control (IFC) is intimately linked to multi execution and, unfortunately, pose the conjecture that transparent, and secure black-box IFC cannot be efficient in the general case [2]. Yet, they note that it is still be applicable in our case and that the situation may be helped by integrating static analysis results.

Another similar approach is that of lightweight dual execution (LDX) by Kwon et al. [47]. Here, the authors propose to use two executions to infer which events causally depend on which preceding event. The two execution are executed in tight lock-step and are provided the same inputs, except for the event in question. The idea bears similarity to the definition of strong dependency by Cohen et al. [26], who propose that an event e_1 depends on another event e_2 when variety in e_2 carries to event e_1. The LDX approach allows to detect information leaks and other security attacks, but does not thwart them.

An early version of multi-execution based confidentiality enforcement for machine code has been proposed by Capizzi et al. [20]. In their shadow execution approach, the target program is executed twice, with each execution in an isolated environment based on virtualization. Consequently, their approach comes with far greater resource requirements than ours and does not allow fine-grained optimizations such as our dynamic instancing or bounding approach. It is limited to two classification levels. Another early approach is proposed by Yumerefendi et al. with their TightLip system [88]. Similar to SME and shadow execution, TightLip uses a sandboxed doppelganger process, which is created dynamically when sensitive information is read. The doppelganger is fed dummy information and both executions continue as usual. When the two processes try to write different outputs, the original execution is replaced by the doppelganger, effectively mitigating the leak. Yet, as the doppelganger lacks the actual input, subsequent messages to authorized users will become intransparent. More false positives may be produced when the doppelganger takes a different execution path due to the dummy input, leading to similar problems as with the termination-insensitive multiple facets approach. In contrast to our work, both systems focus on detecting leaks, but cannot always recover from them.

Many more multi-execution engines exist for machine code with the aim of

guaranteeing control-flow integrity instead of confidentiality. In this case, the same input is fed to slightly altered variants of the target program [27]. When different output is produced, it must be due to interference between the user input and the introduced variations. Zhang et al. provide an overview of the different techniques [91]. In work outside the scope of this thesis, we contributed to the field with a mechanism to detect differences in the control-flow when they occur [64]. The approaches usually only work with a small number of variants and use a simple record and replay approach to multiply inputs [84]. Unfortunately, these techniques do not apply in our scenario where executions are allowed to diverge.

3.2 Language-Based Enforcement

Earlier work in the field follows the certification approach from Denning and Denning [31]. Their groundbreaking work focuses on verifying the confidentiality of a target program based on its operational semantics. The idea is to provide a compile-time checking routine to guarantee confidentiality by design. As such, it is not suited to enforce confidentiality of pre-compiled targets, yet it has inspired field of language-based information-flow security [67]. Consequently, many tools have been produced to develop applications with confidentiality in mind.

For example, Barthe et al. propose to use self-composition to verify confidentiality in programs [12]. The idea is to duplicate the statements in the target program with fresh variables and then use Hoare logics to prove equality of public memory after termination. They also discuss the use of more sophisticated separation logic, which extends Hoare logic to include references to heap-allocated objects [66]. While Barthe et al. state that their approach is not decidable in general, they propose its use to automate or shorten confidentiality proofs. In this regard, Terauchi and Aiken argue that such an optimistic prospect is unrealistic in practice [82]. They used the self-composition approach together with state of the art verification tools but where able to find unsolvable examples.

The idea of self-composition caters to the characterization of noninterference as a hyperproperty [25]. Because it requires at least two traces to identify an interference, Terauchi and Aiken consider it a 2-safety property. Hamlen et al. show that similar properties, such as their secret file property, are not enforceable by either static analysis nor single-trace run-time monitoring [40]. Yet, they show that enforcement can be achieved through program rewriting, hinting at multi-execution based solution. Ngo et al. show that it is impossible to enforce termination-sensitive noninterference, when attackers can observe the actual termination of the target [59]. Instead, they propose to settle for the enforceable

indirect termination-sensitivity, where termination can only be inferred by the absence of outputs.

With the rise of Secure Multi-Execution, Barthe et al. also show how programs can be statically transformed to adhere to the Secure Multi-Execution semantics [13]. Because we focus on low-level languages that are hard to rewrite statically [14, 85], we follow Devriese and Piessens in transforming the target during run-time. In this framework, our optimizations can then be understood as a form of run-time partitioning.

Other approaches make use of extended type systems, which can be checked during compilation. For example, JIF [46, 57] for Java, FlowCaml [76] for Caml, the SPARK Examiner for Ada [22], and most recently F* [52]. However, these do not cover the frequently used languages from the C family and can only guarantee security within the application code. Tools exist to validate confidentiality for Java [78] based on dependence-graphs. This, however, is unlikely to be feasible for compiled code, where dependencies are much harder to analyze [10]. For Java bytecode used in Android devices, information flow analysis can be done using taint tracking [5, 36, 38], which can also be done for machine code [39]. This could be a promising approach to defining boundaries in the future but cannot on its own precisely determine malicious flows [21, 72].

A good overview of dynamic techniques before 2007 can be found in LeGuernics thesis [50]. Bielova and Rezk compare no-sensitive upgrade [90] and permissive upgrade [7] with SME and MF [16]. Their results show that these single-execution monitoring approaches are less secure and less precise than multi-execution based enforcement. Static analysis results can be used to make the monitor more permissive, leading to hybrid monitoring [50]. Yet, such approaches still only allow for mitigation of leaks, not for their repair. As Kashyap et al. note, mitigation does not actually enforce noninterference but merely makes it less harmful [42].

3.3 Binary Analysis

As low-level code does not support types and lacks a range of other supporting features of high-level languages, information flow analysis and certification for binary targets is more difficult to achieve. The interest goes back to decompilation efforts, for example by Cifuentes and Gough in 1995 [23], with the goal to recover high-level code from compiled targets. Such a solution would open legacy code to all kinds of source-code analyses, optimizations, and transformations. Yet, because of the complexity of the problem, much is still done through potentially unsound heuristics. In a recent overview study, Andriesse et al. show that while some

binary analysis tools achieve very precise results, none is reliable for higher levels of optimization and most are poor on function start recognition [4].

The compared approaches include simple, heuristics-based approches [29, 73] that are incapable of precisely resolving indirect jump targets [35, 48]. Resolving indirect jump targets requires precise data-flow and points-to analysis, which is hard to achieve. Yet, precise a CFG is not always necessary, for example when looking for vulnerabilities instead of proving their absence. Thus, in their paper on Bitblaze [79], Song et al. connect unresolvable branches to a special node, that allows Bitblaze to recognize unsound results in later stages. Yet, where sound analysis results are required, more advanced techniques are needed.

In CodeSurfer/x86 [10], Balakrishnan et al. use their value set analysis (VSA) approach first described in [9]. In VSA, the machine code is interpreted in an abstract domain that overapproximates all reachable memory states. Kinder et al. proposed a similar approach with their work on Jakstab [43]. Based on the work by Flexeder et al. on inter-procedural control-flow reconstruction [35], Miahila devised a hierarchical structure of cofibered domains to increase the precision of VSA-based control-flow and data-flow anlyses [55]. Additional binary analysis frameworks exist, such as Binoca [11, 34] or BAP [18, 19].

Lately, approaches based on symbolic execution have gained attention [74, 75]. These are mostly used to prove the presence of vulnerabilities and do not aim for full coverage of the control flow. Milushev et al. used symbolic execution to create abstract models of the possible output bevhavior of compiled targets [56]. These can be used to detect interference by a self-comparison, similar to the self-composition approach on source code. Yet, due to the poor scalability of current symbolic execution engines, the greatest program in their evaluation set contains only 430 instructions. In contrast to this, we successfully enforce confidentiality for the `cat` binary with over 16000 instructions.

Current trends focus on parallelizing the analyses to achieve better scalability [53]. A better scaling and more precise binary analysis solution would help our approach to allow for automatic extraction of boundary conditions, as described in Chapter 9.

3.4 Summary

The related work show the intricate nature of confidentiality enforcement for machine code. What may seem straightforward at first, is actually uncharted territory in many different ways. First, much discussion revolves around which kind of confidentiality can be enforced by which enforcement mechanism, and with which kind

of transparency. Here, multi-execution based approaches such as Secure Multi-Execution are most promising, as they offer strong security and high transparency. Yet, recent works suggest that the inherent inefficiency may be irreducible in the black-box case. This points to solutions that make use of static analysis results to increase the efficiency, such as our optimizations. Unfortunately, while static analysis is well-researched on higher-level languages, low-level languages still pose difficult challenges. Much work has focused on finding violations of the integrity of the control flow, leaving methods to proof the absence of data leaks for future work. Still, the current results are sufficiently precise to aid developers in providing the information needed for our approach.

Part I

Confidentiality Enforcement for Machine Code

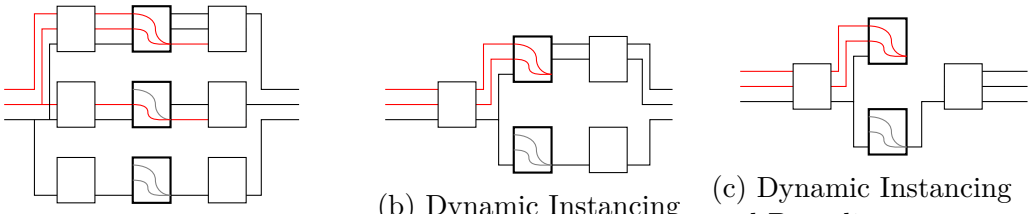
Overview

The problem addressed in this thesis is that software that includes preexisting components may inherit insecure behavior from them. Thus, a protection mechanism is needed to enforce confidentiality across all components in the system. The solution that we propose is based on the concept of Secure Multi-Execution, as introduced by Devriese and Piessens in their seminal paper [32]. Secure Multi-Execution has been formally proven to guarantee timing-sensitive noninterference [65]. It has also been proven to achieve per-channel transparency [65], making it the most secure enforcement mechanism with high transparency [16]. Recent work suggests that multi-execution is a shared feature of all transparent information flow control systems [2].

Yet, two key challenges prevent the wide-spread use of Secure Multi-Execution up to now. First, while there exist implementations for high-level languages such as JavaScript [32] or Haskell [41], no practical solution exists to protect compiled components with Secure Multi-Execution. Second, due to the required multi-execution of the target system, Secure Multi-Execution is very inefficient. With the technique based around executing the target system once per security level, the enforcement overhead grows substantially with the complexity of the target and number of levels.

In this part, we present our solution in six steps. First, in Chapter 4, we introduce a running example program to demonstrate the problem addressed in this thesis. We provide a compiled form of the program in Section 4.2, extending the machine code semantics from Section 2.3 with semantics for interaction with the operating system. Using this example, we show how information leaks can occur and connect the results to a real-world implementation of the example in Section 4.4.

Then, in Chapter 5, we show how we apply the concept of Secure Multi-Execution to machine code. We execute the target multiple times and intercept I/O-requests from the the individual executions. We then alter the input and output semantics according to Secure Multi-Execution semantics. As a result, we



(a) Secure Multi-Execution (b) Dynamic Instancing (c) Dynamic Instancing and Bounding

Figure 3.1: Schematic comparison of our SME optimizations

accomplish our goals of practical, secure and transparent enforcement. We demonstrate this in Section 5.3, where we apply our solution to the running example.

Yet, Secure Multi-Execution is known to be inefficient, as illustrated in Figure 3.1a. The key idea to our optimizations is to reduce redundant computations in two ways. On the one hand, we create as few executions a possible and do so only when needed. We achieve this through our *dynamic instancing* optimization, as described in Chapter 6. Here, executions are created only when information from a hitherto unserviced level is obtained, as illustrated in Figure 3.1b. As we show in Section 6.4, our optimization roughly halves the enforcement overhead for our example.

Furthermore, we terminate as many executions as early as possible, as illustrated in Figure 3.1c. We introduce our *bounding* optimization in Chapter 7. Unfortunately, early termination of executions may break the functionality of the target. This requires the definition of termination conditions, called *boundaries*, that define when it is safe to terminate an execution. Assuming a definition of the boundaries for the running example, our optimization again roughly halves the overhead, as we show in Section 7.4.

Our Bounding Optimization may introduce a dependency between executions at lower levels and executions at higher levels. This could lead to a delay that can be exploited to leak information through the timing side-channel. Thus, we introduce a *timing-sensitive scheduler* in Chapter 8 that guarantees both timing-sensitive security and per-channel transparency, while enabling increased efficiency through our optimizations.

Finally, we outline our contributions to static binary analysis in Chapter 9. A a precursor to automatic extraction of boundary conditions from binary code, we introduce a heuristic to resolve indirect calls during control-flow reconstruction. This allows us to increase the precision of extracted control-flow graphs. These can be used to identify merge points in the unstructured code, from which boundary conditions can be derived.

Chapter 4

Threat Model

In this section we motivate the threat scenario and illustrate leaks due to different kind of flows. We introduce an insecure example program, which we also use throughout the rest of the thesis to illustrate our confidentiality enforcement approach. The example follows the motivation outlined in the introduction of this thesis. It is a combination of high-level code, under control of the developer, and low-level library code, included but not under control of the developer. To be functional, sensitive information must be passed to the low-level component, which may then leak the information with our without malicious intent. We assume that either identifying the insecurities in the included library code is too costly or complex, or that the developer has no alternative to including the insecure library. This motivates the requirement for an automatic, program-wide confidentiality enforcement method that guarantees confidentiality across *all components* of the software system.

4.1 Decryption Service

As the running example of this thesis, we consider a service that has the functionality to decrypt encrypted data from different sources with a given key. Cryptographic algorithms are hard to implement efficiently, prone to side-channel attacks, and crucial to get right. It is therefore advised to use an existing implementation, in this case shipped in the form of a pre-compiled library. The library requires the key as input, as well as an identifier to select the ciphertext input and determine the plaintext output channel. When executed, the library reads the selected ciphertext, performs the decryption using ciphertext and key, and writes the resulting plaintext to the plaintext location. For the sake of the argument, we assume

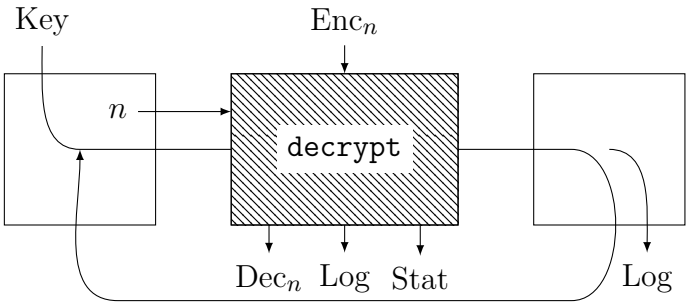


Figure 4.1: Information flow schematic of the example program

that the library additionally produces debug and statistical information, e.g. for performance profiling or code maintenance. Since the library can only decrypt one ciphertext at a time, it is called in a loop until all ciphertexts from different sources have been processed.

Figure 4.1 illustrates the architecture of the example and involved channels as a Hasse diagram. The first component, which is under control of the developer, reads in the key and calls the decryption function, implemented in a black-box library. All that is known to the developer about the information flows in the library is that it requires the key information, that input from the ciphertext channel Enc can be obtained, and that output may be written to the plaintext channel Dec, as well as to the logging channel Log and the statistics channel Stat. Since ciphertext from multiple sources may be processed, the library is executed in a loop, reusing the key information. Thus, there may be n ciphertext and plaintext channels, where n is the number of sources to be processed. When all sources have been processed, the third component, again under control of the developer, writes a final confirmation message to the Log channel and the program terminates. Conclusively, a maximum of $2n+3$ channels are involved in the execution of the program.

The security lattice used to classify the channels in this example is illustrated in Figure 4.2. It is the consequence of various constrains that arise from the scenario. First, it is clear that the ciphertext channel cannot be classified higher than the key channel. Otherwise, users with access to the ciphertext would also be granted access to the key, which consequently grants them access to the plaintext (assuming that the cryptographic routine is not a secret itself). On the other hand, users with access to plaintext implicitly need access to information about the key and the ciphertext, as the plaintext depends on it. Yet, access to one plaintext should not imply access to *all* ciphertexts. Therefore, access to key information should not include access to ciphertext information. Thus, key and ciphertext are classified with incomparable levels, but both are less classified than plaintext information. Additionally, the logging information should be readable by anyone and thus not

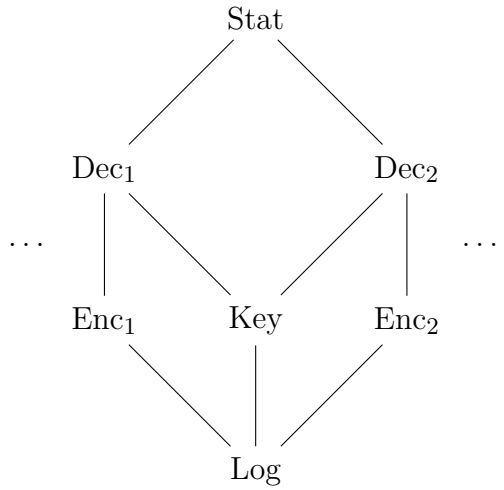


Figure 4.2: Security lattice for the decryption service

contain any information about the key, ciphertexts or plaintexts. Conversely, the statistical information may combine information from all plaintexts and thus must have the highest classification. Note that while the size of the lattice depends on the number of sources in a run of the program, we assume that it is fixed and finite.

Whether or not sensitive information may leak through the library depends on its code. Unlike the developer, who has no knowledge of this code, the attacker does control the library in our scenario. Thus, the library could be arbitrarily complex, include additional malicious features, or be obfuscated against analysis. Yet, as we show here, information may also leak from a simple, straight-forward implementation of the required functionality. Thus, we assume the simple implementation as represented in pseudo-code in Listing 4.1.

We assume that decryption is impossible if no key is provided. Thus, if this is the case, the library logs an error code (i.e. `-1`) and returns immediately. Otherwise, the ciphertext is read in from the Enc channel, specified as an argument. The plaintext is then computed using the ciphertext, the key, and a cryptographic operation, denoted with $\oplus$ here. Subsequently, the plaintext is written to the specified decrpytion channel Dec, and the length of the plaintext is written to the statistics channel. Finally, the length of the key is logged for debugging purposes. As a whole, this code achieves the behavior as required above.

```
1 decrypt(key, Enc, Dec):
2   if (key) then
3     ciphertext := in(Enc);
4     plaintext := key ⊕ ciphertext;
5     out(Dec, plaintext);
6     out(Stat, length(plaintext));
7     out(Log, length(key));
8   else
9     out(Log, -1);
```

Listing 4.1: Pseudo-code of the library

4.2 Compiled Form

When compiled, the pseudo-code implementation from Listing 4.1 is turned into the low-level representation shown in Listing 4.2. We express this implementation in our machine code semantics as outlined in Section 2.3. First, the arguments are stored in memory and the sum of processed bytes is initialized with zero. Then, the length of the key is checked to ensure that a non-empty key has been provided. If this is is not the case, then an error code is written to the Log level and the library returns. Else, the ciphertext is obtained (lines 32-37), the decryption operation is called (line 42), and the plaintext is written to the decryption channel (lines 43-47). Finally, the total number of decrypted bytes is written to the Stat channel (lines 50-53) and the key length is logged (lines 54-57). Then, the execution returns from the library in line 58.

The low-level implementation differs from the high-level pseudo-code in three important ways. First, it lacks functions and structure, instead using the goto-like jump instruction to model complex control flow and specific argument registers to pass arguments to called functions as well as the special `retval` register to return results (similar to the System V AMD64 ABI of modern Linux systems [51]). We also use the argument registers to pass the encryption and decryption channels from the main object to the library. Second, variables holding strings are interpreted as pointers to arrays of fixed size. Thus, the ciphertext is decrypted in chunks of 100 bytes, as shown in line 35. The decryption loop, between lines 32 and 49, is broken when no more input is obtained. This is the case when the `read` system call from line 36 returns 0. Third, input and output is handled via calls to the operating system. Thus, we add a `syscall` instruction to our machine code model and provide the semantics for input and output functionality in Semantics 4.1.

```
22 [enc] := arg1
23 [dec] := arg2
24 [sum] := 0

25 jcc ([key_len] > 0) 32
26 sysno := write
27 arg1 := Log
28 arg2 := -1
29 arg3 := 1
30 call 20
31 jmp 58

32 sysno := read
33 arg1 := enc
34 arg2 := encoded_str
35 arg3 := 100
36 call 20
37 [encoded_len] := retval

38 jcc ([encoded_len] == 0) 50
39 arg1 := key_str
40 arg2 := encoded_str
41 arg3 := decoded_str
42 call ⊕
43 sysno := write
44 arg1 := dec
45 arg2 := decoded_str
46 arg3 := encoded_len
47 call 20
48 [sum] := [sum] + [encoded_len]
49 jmp 32

50 arg1 := Stat
51 arg2 := sum
52 arg3 := 1
53 call 20
54 arg1 := Log
55 arg2 := key_len
56 arg3 := 1
57 call 20

58 return
```

Listing 4.2: Library object code

$$\textsc{Read-Iterate}\ \frac{\begin{array}{c}\rho \vdash (syno, arg_1, arg_2, arg_3) \Downarrow (read, c, b, n) \\ n > 0 \qquad e(c) = v \qquad v \neq \diamond \qquad \mu' = \mu[b \leftarrow v] \\ \rho' = \rho[arg_3 \leftarrow (n-1), arg_2 \leftarrow (b+1), tmp \leftarrow \rho(tmp)+1]\end{array}}{e, \Sigma \vdash \rho, \mu, pc, \texttt{syscall} \xrightarrow{c?v} \rho', \mu', pc, \texttt{syscall}}$$

$$\textsc{Read-Stop}\ \frac{\begin{array}{c}\rho \vdash (syno, arg_1, arg_2, arg_3) \Downarrow (read, c, b, n) \\ n \leq 0 \vee e(c) = \diamond \qquad \rho' = \rho[retval \leftarrow \rho(tmp), tmp \leftarrow 0]\end{array}}{e, \Sigma \vdash \rho, \mu, pc, \texttt{syscall} \xrightarrow{\bullet} \rho', \mu, pc+1, \Sigma[pc+1]}$$

$$\textsc{Write-Iterate}\ \frac{\begin{array}{c}\rho \vdash (syno, arg_1, arg_2, arg_3) \Downarrow (write, c, b, n) \qquad n > 0 \qquad v = \mu(b) \\ \rho' = \rho[arg_3 \leftarrow (n-1), arg_2 \leftarrow (b+1), tmp \leftarrow \rho(tmp)+1]\end{array}}{e, \Sigma \vdash \rho, \mu, pc, \texttt{syscall} \xrightarrow{c!v} \rho', \mu, pc, \texttt{syscall}}$$

$$\textsc{Write-Stop}\ \frac{\begin{array}{c}\rho \vdash (syno, arg_1, arg_2, arg_3) \Downarrow (write, c, b, n) \qquad n \leq 0 \\ \rho' = \rho[retval \leftarrow \rho(tmp), tmp \leftarrow 0]\end{array}}{e, \Sigma \vdash \rho, \mu, pc, \texttt{syscall} \xrightarrow{\bullet} \rho', \mu, pc+1, \Sigma[pc+1]}$$

Semantics 4.1: System call semantics for input and output

The `syscall` instruction handles all kinds of requests to the kernel and thus provides a range of functionalities. Since the instruction takes no argument, the functionality is chosen by assigning platform-specific identifiers to a special register, represented by the *sysno* register. Here, we focus on the `read` and `write` system calls, which represent input and output functionality. Arguments to the system calls are also taken from the special argument registers $arg_{1\ldots3}$. In case of a `read` system call, the arguments specify the channel c, the buffer location b, and the maximum input length n. Input is then obtained by taking values from the environment and storing them at the buffer location b. After each input value, the buffer location is increased by one and the number of requested bytes decreased by one. Additionally, an internal counter register *tmp* is also increased by one, to later return the total number of bytes read. Once the channel is depleted, meaning an EOF-signal (denoted $\diamond$) is returned, or the required input length has reached zero, then the return value register *retval* is set to the counter stored in *tmp* (which is subsequently reset to 0) and the program continues. Following these semantics, the `read` system call loops until the requested amount of input is obtained or the channel is depleted. The `write` system call is analogous to the `read` system call. The system call may also block on input when it is obtained from streams such as network sockets. We show how we handle blocking stream input in Appendix C.

```
 1 sysno := read
 2 arg1 := Key
 3 arg2 := key_str
 4 arg3 := max_key_len
 5 call 20
 6 [key_len] := retval

 7 [i] := 0
 8 jcc ([i] == n_channels) 14
 9 arg1 := [enc_channels + [i]]
10 arg2 := [dec_channels + [i]]
11 call 22
12 [i] := [i] + 1
13 jmp 8

14 sysno := write
15 arg1 := Log
16 arg2 := "."
17 arg3 := 1
18 call 20

19 return

20 syscall
21 return
```

Listing 4.3: Main object code

Because the `syscall` instruction provides different kernel functionality, it is usually wrapped by library functions to provide error handling, improve performance, enrich the functionality, add buffering etc. We reflect this behavior in our example by using a single `syscall` instruction that is called from different points in the code. The `syscall` wrapper is located on line 20 in the main object code, shown in Listing 4.3. The main object code contains the entry point and represents the component under control of the developer. It starts with obtaining the key from the *Key* channel (lines 1-6). We then enter the main loop that advances through the the encryption and decryption channels and calls the library function for each (lines 7-13). This main loop terminates when all $n_{channels}$ have been processed. When this is the case, the main object finally prints a termination symbol to the Log channel (lines 14-18), indicating termination of the service, and exits.

4.3 Leaks

An explicit flow, meaning an information leak through data flow, occurs in the library when the length of the key information is written to the log file. This allows users not authorized to access key information to infer information about the key. To show that this behavior violates progress-sensitive noninterference (PSNI), and thus also timing-sensitive noninterference (TSNI), we consider two executions with no input on the Log channel. We assume that $e_1(Log) = e_2(Log) = Log?\diamond$, and some encryption input, $e_1(Enc_1) = e_2(Enc_1) = Enc_1?a : Enc_1?\diamond$. Yet, the environments provide keys of different lengths, such that $e_1(Key) = Key?x : Key?\diamond \neq e_2(Key) = Key?x : Key?x : Key?\diamond$. We thus get $e_1 \models s \xrightarrow{\bar{a}_1} \Longrightarrow \bar{a}_1 \upharpoonright_{!,Log,\bullet} = Log!1 : Log!\cdot$ and $e_2 \models s \xrightarrow{\bar{a}_2} \Longrightarrow \bar{a}_2 \upharpoonright_{!,Log,\bullet} = Log!2 : Log!\cdot$. These executions violate PSNI because $e_1 =_{Log} e_2 \not\Rightarrow \bar{a}_1 =_{!,Log,\bullet} \bar{a}_2$.

An implicit flow, meaning an information leak through control flow, occurs in the library when no key is provided. In this case, the error value -1 is written to the log file. This violates PSNI, when compared to a trace with key information. Assume $e_3(Log) = e_1(Log)$ and $e_3(Key) = Key?\diamond$. Then $e_3 \models s \xrightarrow{\bar{a}_3} \Longrightarrow \bar{a}_3 \upharpoonright_{!,Log,\bullet} = Log!-1 : Log!\cdot$. And thus $e_1 =_{Log} e_3 \not\Rightarrow \bar{a}_1 =_{!,Log,\bullet} \bar{a}_3$.

Next, we consider the more covert leaks through termination and timing. Information leaks through the termination channel when the execution does not return from invocation of the library. To illustrate this, we assume that the cryptographic operator $\oplus$ diverges when the key is the value 0. Then, we assume $e_4(Log) = e_1(Log)$ and $e_4(Key) = Key?0 : Key?\diamond$. Thus, $e_4 \models s \xrightarrow{\bar{a}_4} \Longrightarrow \bar{a}_4 \upharpoonright_{!,Log,\bullet} = \epsilon$ and therefore $e_1 =_{Log} e_4 \not\Rightarrow \bar{a}_1 =_{!,Log,\bullet} \bar{a}_4$.

Alternatively, when we assume that the cryptographic operator $\oplus$ terminates abnormally when the ciphtertext is 0, then we leak information about the ciphertext through the termination behavior. Assume $e_5(Log) = e_6(Log) = Log?\diamond$, $e_5(Key) = e_6(Key) = Key?x : Key?\diamond$, $e_5(Enc_1) = Enc_1?0 : Enc_1?\diamond$, $e_6(Enc_1) = Enc_1?1 : Enc_1?diamond$, and $e_5(Enc_2) = e_6(Enc_2) = Enc_2?2 : Enc_2?\diamond$. Then $e_5 \models s \xrightarrow{\bar{a}_5} \Longrightarrow \bar{a}_5 \upharpoonright_{!,Dec_2,\bullet} = Key?x$, as the execution terminates abnormally before the second ciphertext is processed. However, $e_6 \models s \xrightarrow{\bar{a}_6} \Longrightarrow \bar{a}_6 \upharpoonright_{!,Dec_2,\bullet} = Key?x : Enc_2?2 : Dec_2!\ldots$, and thus $e_5 =_{Dec_2} e_6 \not\Rightarrow \bar{a}_5 =_{!,Dec_2,\bullet} \bar{a}_6$. Note that the abnormal execution is also visible on the Log channel, which constitutes another leak through the termination channel.

Information leaks due to timing occur when the timestamps of equal outputs are different under the same low inputs. This is even more elusive than termination attacks and depends on the formalization. Yet, assuming that only

the $\oplus$ operator takes time proportional to the input, it is clear that information leaks about the presence and length of ciphertexts. Assume $e_7(Log) = e_6(Log)$, $e_7(Key) = e_6(Key)$, $e_7(Enc_2) = e_6(Enc_2)$, but $e_7(Enc_1) = Enc_1?\diamond \neq e_6(Enc_1)$. Then $e_7 \models s \xrightarrow{\bar{a}_7} \Longrightarrow \bar{a}_7 \restriction_{!,Dec_2,\vec{\bullet}} = Key?x : \bullet : Enc_2?2 : Dec_2!\ldots$. Thus $e_7 =_{Dec_2} e_6 \not\Rightarrow \bar{a}_7 =_{!,Dec_2,\vec{\bullet}} \bar{a}_6$.

To show that these theoretical leaks also occur in practice, we implemented the example in C and compiled it into a binary. We use the binary to test the attacks in practice and discuss the results next. We use the binary again throughout the rest of the thesis, to demonstrate the practicality of our enforcement approach.

4.4 Attacks

In practice, the progress-sensitive attacks can be validated by finding *pairs of inputs* that are the same for all levels below or equal to some ℓ but produce different output for that level. Thus, we execute seven different attacks to show that leaks through explicit, implicit, termination and timing are possible in the example. The results are shown in Table 4.1, where one test case consists of two different inputs.

Case 1 shows that the same inputs lead to the same outputs across all levels, demonstrating that the program is deterministic. This is a prerequisite for noninterference, as it allows to attribute differences in the output to differences in the input. Case 2 shows that a different key leads to different decryption and logging results. This does *not* violate progress-sensitive noninterference (PSNI) for the decryption channels Dec_1 and Dec_2, as the key is classified lower than these levels. Thus, regarding these channels the prerequisite of equal low inputs is not given. It *does* however violate PSNI for the Log level, as here the prerequisite is satisfied but the outputs differ. This attack shows an illegal explicit flow. Case 3 shows an illegal implicit flow. As demonstrated, the presence or absence of the key leads to different logging output. Unlike case 2, this is a consequence of diverting control flow instead of different data.

Cases 4, 5 and 6 show leaking through (non-)termination. To collect the results, we terminate the execution after a sufficient timeout has been reached. In case 4, the zero in the key leads to an immediate divergence of the execution. Thus, no output is produced on any channel. Yet, this only violates PSNI on the Log level, as all other levels have access to the key information and thus this behavior is expected. In case 5, a zero in the first ciphtertext leads to divergence. Consequently, the second ciphertext is not decrypted and no logging output is produced. Both effects are illegal, as users without clearance for the first ciphertext could infer

	Inputs			*Outputs*			
	Key	Enc_1	Enc_2	Dec_1	Dec_2	Log	Stat
1	key	ikoupgnp	epjunvutv	sometext	othertext	3 3 .	8 9
	key	ikoupgnp	epjunvutv	sometext	othertext	3 3 .	8 9
2	abcd	ikoupgnp	epjunvutv	ilqxphps	eqlxnwwwv	4 4 .	8 9
	key	ikoupgnp	epjunvutv	sometext	othertext	3 3 .	8 9
3		ikoupgnp	epjunvutv			-1 -1 .	
	key	ikoupgnp	epjunvutv	sometext	othertext	3 3 .	8 9
4	0	ikoupgnp	epjunvutv				
	key	ikoupgnp	epjunvutv	sometext	othertext	3 3 .	8 9
5	key	0	epjunvutv				
	key	ikoupgnp	epjunvutv	sometext	othertext	3 3 .	8 9
6	key	ikoupgnp	0	sometext			8
	key	ikoupgnp	epjunvutv	sometext	othertext	3 3 .	8 9
7	key	ikou...	epjunvutv	some...	othertext	3 3 .	8000 9
	key	ikoupgnp	epjunvutv	sometext	othertext	3 3 .	8 9

Table 4.1: Exemplary attacks and timing-insensitive results

from these results that the first ciphertext contains a zero. Thus, this is an illegal flow from Enc_1 level to Log and to Dec_2 level. In case 6, the divergence occurs after the first ciphertext has been processed. Thus, only the missing output to Log constitutes an illegal flow here. Note that all three cases look the same when observing the logging output. Thus, the attacker could not differentiate between them. Additionally, since this is a termination attack, the attacker cannot know if the execution diverged indefinitely or will eventually return with a result. However, it is reasonable to assume that an attacker aware of the library code can eventually make that decision with high confidence.

The last test case, case 7, demonstrates a leak through the timing-channel. Here, the first ciphertext is repeated multiple times. As shown, this does not adversely affect any outputs, making this test case PSNI-secure. Yet, the different size of the input does affect the timing behavior. We demonstrate the deviation of the timestamps of the last output to the decryption and logging channel for different sizes of the first ciphertext in Figure 4.3. Although we keep the size of the Enc_2 input constant, a delay proportional to the input size of Enc_1 can be detected in the timestamps of the last output to Dec_2. The same is true for the last output to the Log channel. This shows that the example is not timing-sensitively noninterferent.

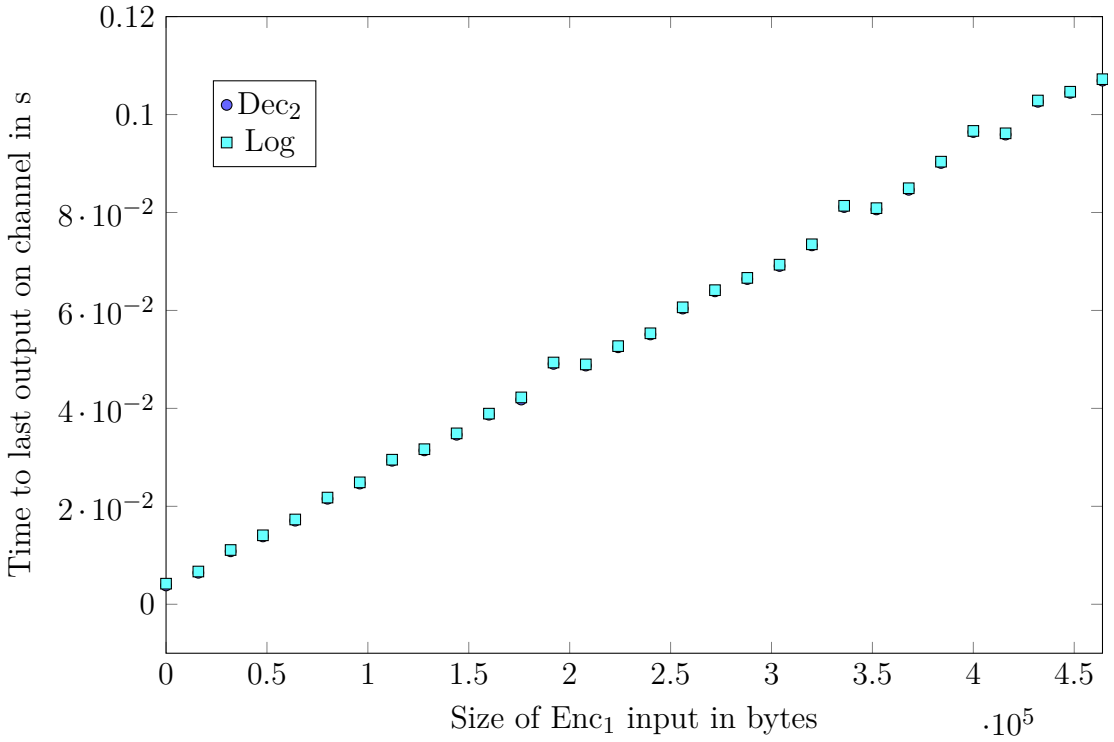


Figure 4.3: Exemplary timing-attacks

4.5 Summary

The example presented in this section demonstrates the problem addressed in this thesis. It represents a software product that is developed using existing components. However, the included code is no confidential. It allows to leak information between levels of a security lattice through various channels. Information may leak explicitly, meaning that data with lower classification flows to an output on a channel with higher classification. Furthermore, information may leak implicitly, meaning through observable changes in the output behavior to lower channels depending on information from higher channels. Internally, this is the result of control flow that depends on sensitive information. Finally, information may also leak through the termination and timing behavior.

Yet, the developer may still wish to include the library, either because the insecurities are unknown or because no better alternative exists. To achieve system-wide confidentiality, this requires an enforcement approach that can be applied to compiled code. In the following, we demonstrate how we enforce confidentiality for machine code using concepts from Secure Multi-Execution. This allows

us to *transform* the semantics of the insecure library at run time, yielding secure behavior while the intended functionality remains unaltered. Because this solution comes with high resource demands, we introduce further optimizations that increase the enforcement efficiency. Together with our timing-sensitive scheduling, this eventually leads to our enforcement method that efficiently guarantees security and transparency for machine code components, including the example demonstrated here.

Throughout the discussion of our enforcement approach, we come back to the example to illustrate the effect on the security, transparency, and efficiency of our contributions. We reuse the attack cases to show how progress-sensitive noninterference for explicit, implicit, and termination-leaks is achieved and we reuse the timing-attack to demonstrate how timing-sensitive noninterference is guaranteed.

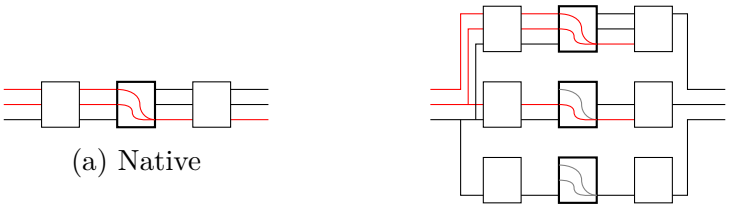
Chapter 5

Secure Multi-Execution for Machine Code

With our work, we aim to provide a timing-sensitive confidentiality enforcement that does not alter the outputs of secure parts of the program. A promising method to achieve these requirements is Secure Multi-Execution (SME), illustrated in Figure 5.1. Under SME enforcement, the target program is executed multiple times, once for each security level. The input semantics for each execution are changed such that executions can only obtain input from channels with equal or lower security level. Additionally, the output semantics are changed such that executions can only produce output on channels with equal security level. Together, this ensures that output on channels at any level has been produced by an execution that never had access to input from channels with higher classification. Thus, noninterference is guaranteed *by design*.

The problem when applying Secure Multi-Execution to machine code is that it is not trivial to change the semantics of the target. Existing solutions focus on Javascript or Haskell, where the interpreter semantics can be altered to secure programs. In machine code, the language semantics are defined by the hardware. Thus, changes to the semantics would require changes to the processing unit. Alternatively, the target program itself could be rewritten. Yet, binary rewriting is a complex task and unlikely to be feasible in a transparent way.

We solve the problem and provide the first implementation of Secure Multi-Execution for machine code with a monitoring approach. With our monitor, we intercept communication between the target and the operating system through the `ptrace` mechanic. When we intercept a system call at run time, we have access to the full state of the execution, including the type of the system call and provided arguments. We then identify channels with file descriptors and classify them based



(a) Native

(b) Secure Multi-Execution

Figure 5.1: Schematic comparison of unprotected and SME-protected execution

on the resource that they provide access to. Depending on the classification of the execution and channel, we manipulate the arguments to replace input or skip output. When dummy values need to be provided, we redirect the input request to a shared file descriptor that we inject into the target program. To support protection against input-delay attacks and input side-effects, we furthermore implement a virtual filesystem that allows to block executions until input is available.

To show that our implementation achieves timing-sensitive noninterference and per-channel transparency for targets in compiled form, we apply our prototype to the running example introduced in Chapter 4. Our results in Section 5.3 show that our implementation thwarts all attacks outlined in Section 4.4, including leaks through the timing-behavior. At the same time, outputs from secure behavior are unaltered. Thus, we fulfill our goals of security, transparency and practicality with our application of Secure Multi-Execution to machine code. Yet, as we show, this solution comes with a high enforcement overhead. Thus, in the following chapters, we introduce two novel optimizations for Secure Multi-Execution to increase the enforcement efficiency.

5.1 Core Elements

The key assumption of our approach is that communication between environment and application must be done via the operating system. This is the case for example in layered operating systems such as Linux. Consequently, all communication between environment and application can be intercepted at operating system level. Given tool support to manipulate the requests before they are executed by the operating system, we can add our run-time transformation rules as a monitoring component. Under Linux, this can be achieved by the native `ptrace` system calls. The resulting conceptual architecture of our Secure Multi-Execution monitoring is illustrated in Figure 5.2.

Legal communication, meaning communication between an execution and chan-

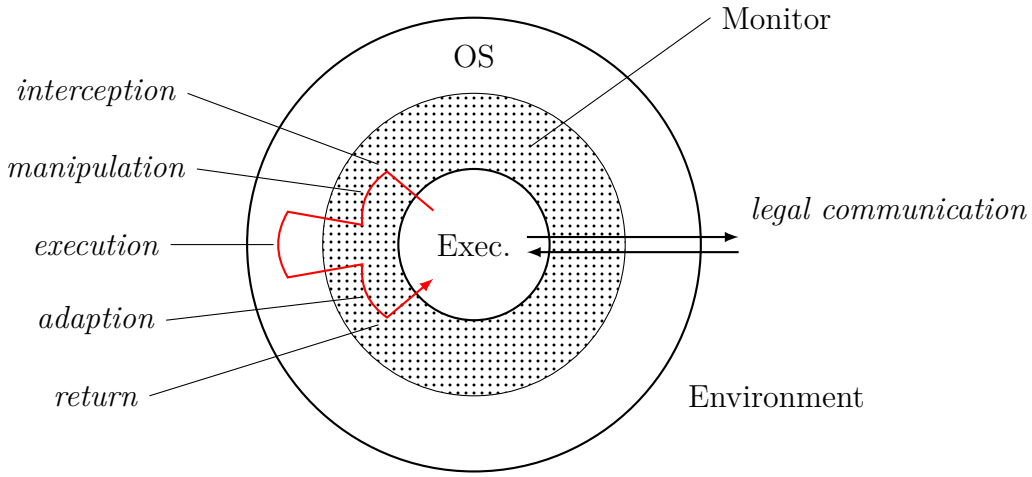


Figure 5.2: Secure Multi-Execution via Syscall Monitoring

nels on corresponding security levels, is merely checked by the monitor and then patched through. This is illustrated on the right. If, however, an execution requests interaction with a channel that it is not allows to access, we intercept the request in the monitor. We then manipulate the request to nullify output, redirect input, and perform additional bookkeeping. Next, we forward the manipulated request to the operating system to perform the manipulated action. The operating system returns the result, which is again intercepted by our monitor. This allows us to adapt the result, for example to hide our manipulations from the execution or fake success. We then return the adapted result to the execution, as illustrated on the left of Figure 5.2. All in all, this allows us to change the intended effect while keeping the changes transparent to the execution.

Naturally, this requires that the operating system as well as our monitor can be trusted. However, trusting the operating system is a given, as otherwise no security can be guaranteed. With Linux as a widely used open-source operating system and our monitor as a rather small additional component, we believe that there is more reason to do so than to trust the plethora of third-party libraries. Consequently, our monitoring component becomes the only target, minimizing the trusted computing base. We give an overview of all parts that make up our Secure Multi-Execution system in Figure 5.3.

Our enforcement system requires four inputs, the target binary, the security lattice, a classification and dummy input. We assume that the security lattice L is finite and does not change during execution. In our application of SME, we furthermore identify files on the system with (abstract) channels. Consequently, our classification π maps file paths to security levels from a given security lattice.

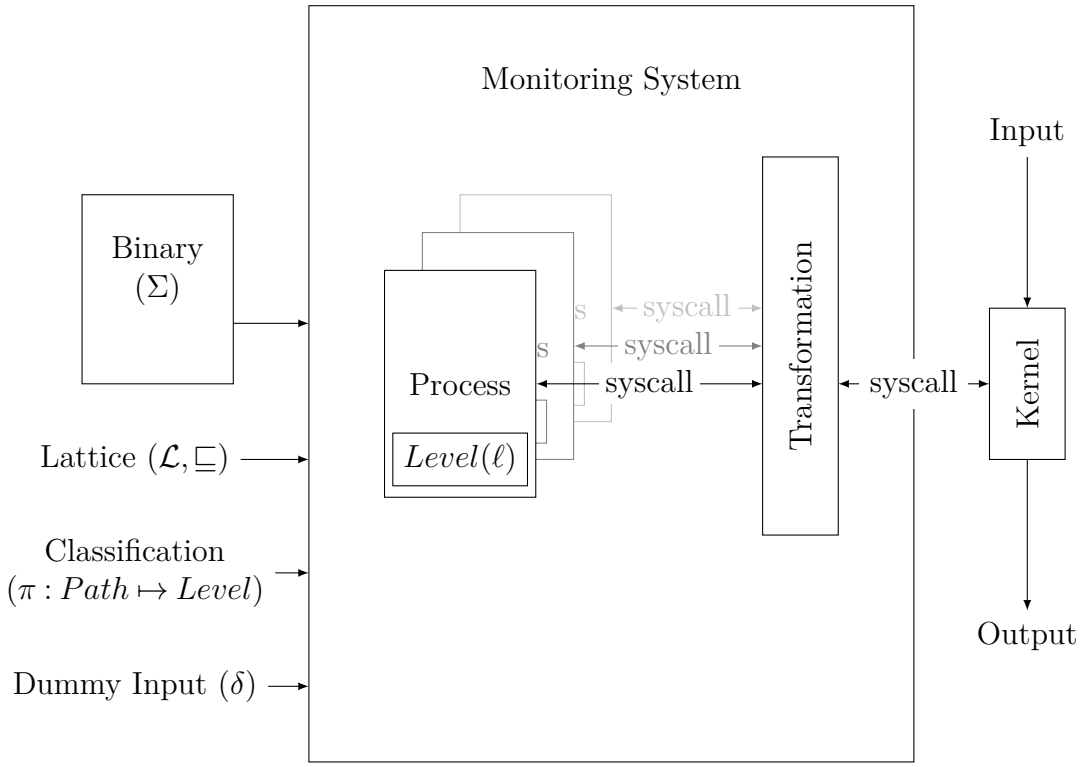


Figure 5.3: Secure Multi-Execution based enforcement system

More complex classification systems are conceivable but not the focus of this thesis. Finally, we define a dummy channel δ, which provides executions with declassified information as a replacement for inaccessible input. The definition of the dummy values is ongoing research that we leave for future work.[1] Thus, we generally use empty input if not otherwise specified.

The Secure Multi-Execution enforcement is then initiated by creating one execution per security level in the lattice L. The individual executions are scheduled in an interleaving and fair manner, for which we use the default process scheduler of the operating system. Thus, we make no changes to the global Secure Multi-Execution semantics. The instantiation of the local Secure Multi-Execution semantics for machine code is more complex. We describe it in more detail next.

[1] Generally speaking, the dummy values should not lead to a crash or other abnormal behavior of the system that breaks its functionality. For example, if some values stored in an XML document should be protected, the dummy input should at least be a well-formatted XML document, as otherwise the target may reject it.

5.2 Semantics

The semantics of our transformation is given in Semantics 5.1. Our goal in the definition of our transformation is to instantiate the local SME semantics with as little alteration to the machine code semantics as possible. As we only want to alter the behavior of input and output operations, we filter for system calls with the appropriate request. Silent internal operations are thus left unchanged, as shown in the Silent rule. This makes our transformation widely applicable, as we do not need specialized rules to handle the vast range of instructions present in low-level languages. Instead, we simply progress from the abstract state s to s', given the instruction ι and next instruction ι' and emit a silent event. Thus, in this instruction, the level of the execution, denoted by ℓ, is ignored.

When an execution is about to execute a `read` system call, we intercept it and compare the classifications. If the requested channel c has lower or equal security level as the execution, then the input operation is permitted. Otherwise, the input is redirected to the dummy channel, denoted by δ. This implement the Input-T and Input-F rules from the local SME semantics as given in Semantics 2.1. Thus, these adaptions ensure that each execution can only access information of equal or lower classification. In our concretization, the channel c is described by the path to the requested resource. For example if the file at `/etc/passwd` is requested, we look up the file path in the user-defined classification denoted by π to derive the security level of the resource. If no classification is given for a path, we assume that it is unclassified (i.e. maps to $\bot$).

Similarly, we intercept calls to the operating system that request `write` functionality. Here, we look up the classification of the target channel c and perform the write only if the execution has the same classification. When this is not the case, we skip the output by setting the requested output amount to zero. In any case, we return a successful write notification to the target binary to ensure transparent enforcement. We achieve this by setting the return value in the *retval* register with the requested n amount of bytes written.

Our adaptions to the machine code semantics are minimal yet sufficient to instantiate the local SME semantics from Semantics 2.1. This allows us to use the same *global* SME semantics, as provided in Semantics 2.2. We thus instantiate one execution per level in the security lattice at the start of the program. The scheduling of the executions is left to the default scheduler of the operating system, from which we only require that it is fair, as otherwise executions may starve and output might be lost. The complete fair scheduler (CFS) of Linux is fair [86].

$$\textsc{Silent}\ \frac{\iota \neq \texttt{syscall} \qquad \rho, \mu, pc, \iota \xrightarrow{\bullet} \rho', \mu', pc', \iota'}{\pi, \delta, \ell \vdash \rho, \mu, pc, \iota \xrightarrow{\bullet} \rho', \mu', pc', \iota'}$$

$$\textsc{Read}\ \frac{\begin{array}{c}\rho \vdash (sysno, arg_1, arg_2, arg_3) \Downarrow (read, c, b, n) \\ \pi(c) \sqsubseteq \ell \qquad \rho, \mu, pc, \texttt{syscall} \xrightarrow{c?v} \rho', \mu', pc', \iota\end{array}}{\pi, \delta, \ell \vdash \rho, \mu, pc, \texttt{syscall} \xrightarrow{c?v} \rho', \mu, pc', \iota}$$

$$\textsc{Read-Dummy}\ \frac{\begin{array}{c}\rho \vdash (sysno, arg_1, arg_2, arg_3) \Downarrow (read, c, b, n) \\ \pi(c) \not\sqsubseteq \ell \qquad \rho^{\delta} = \rho[arg_1 \leftarrow \delta] \\ \rho^{\delta}, \mu, pc, \texttt{syscall} \xrightarrow{\delta?v} \rho', \mu', pc', \iota\end{array}}{\pi, \delta, \ell \vdash \rho, \mu, pc, \texttt{syscall} \xrightarrow{c?v} \rho', \mu', pc', \iota}$$

$$\textsc{Write}\ \frac{\begin{array}{c}\rho \vdash (sysno, arg_1, arg_2, arg_3) \Downarrow (write, c, b, n) \\ \pi(c) = \ell \qquad \rho, \mu, pc, \texttt{syscall} \xrightarrow{c!v} \rho', \mu', pc', \iota\end{array}}{\pi, \delta, \ell \vdash \rho, \mu, pc, \texttt{syscall} \xrightarrow{c!v} \rho', \mu, pc', \iota}$$

$$\textsc{Write-Skip}\ \frac{\begin{array}{c}\rho \vdash (sysno, arg_1, arg_2, arg_3) \Downarrow (write, c, b, n) \\ \pi(c) \neq \ell \qquad \rho' = \rho[arg_3 \leftarrow 0] \\ \rho', \mu, pc, \texttt{syscall} \xrightarrow{\bullet} \rho'', \mu', pc', \iota \qquad \rho''' = \rho''[retval \leftarrow n]\end{array}}{\pi, \delta, \ell \vdash \rho, \mu, pc, \texttt{syscall} \xrightarrow{\bullet} \rho''', \mu', pc', \iota}$$

Semantics 5.1: Local Secure Multi-Execution semantics for machine code

5.3 Example

To demonstrate how our Secure Multi-Execution for machine code transparently enforces confidentiality across a composed system, we rerun the attacks from Section 4.3 on the example. The results are shown in Table 5.1.

As demonstrated, the first case, where all input is the same, is also secure under our Secure Multi-Execution for machine code. The same inputs on all levels lead to the same outputs. This shows that the program is noninterferent in this case and, more specifically, that our enforcement does not introduce any non-determinism to the execution. Also note that the logging output is changed to the error code `"-1 -1"`, representing a missing key. This is a consequence of the enforcement, where the Log-level execution is not allowed to obtain the key. Consequently, to users

	Inputs			*Outputs*			
	Key	Enc_1	Enc_2	Dec_1	Dec_2	Log	Stat
1	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
2	abcd	ikoupgnp	epjunvutv	ilqxphps	eqlxnwwwv	-1 -1 .	8 9
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
3		ikoupgnp	epjunvutv			-1 -1 .	
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
4	0	ikoupgnp	epjunvutv			-1 -1 .	
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
5	key	0	epjunvutv		othertext	-1 -1 .	
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
6	key	ikoupgnp	0	sometext		-1 -1 .	8
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
7	key	ikou...	epjunvutv	some...	othertext	-1 -1 .	8000 9
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9

Table 5.1: Noninterference through Secure Multi-Execution for the example from Chapter 4.

who are only authorized on Log level, the execution behaves as if there is no key at all. At the same time, users with higher authorization do get the correct result on their channels, reflecting that there was a key. This shows that our system repairs the leaking behavior, while keeping the secure behavior unchanged. Note that this is achieved automatically, without requiring any knowledge about the target.

Furthermore, The first case shows that we produce the same results as the native execution for the decryption levels as well as the Stat level. Flows on these levels are secure in the original execution, thus, producing the same outputs is required for per-channel transparency. Output on the Log level is the result of implicit and explicit flows, as described in Section 4.3. Thus, we cannot produce the same outputs as the native execution here, as this would mean reproducing the insecurity. Different outputs on the Log-level are thus a desired consequence of the enforcement.

Naturally, regarding the top level, any program behavior is secure. As users with maximum clearance are allowed to see all information, there is no reason for us to change the output on their level during enforcement. In fact, due to the construction of Secure Multi-Execution, the top-most execution only differs from the native execution in that we only take the top-level outputs from it. Therefore, as it is implied by per-channel transparency, we also achieve top-level transparency

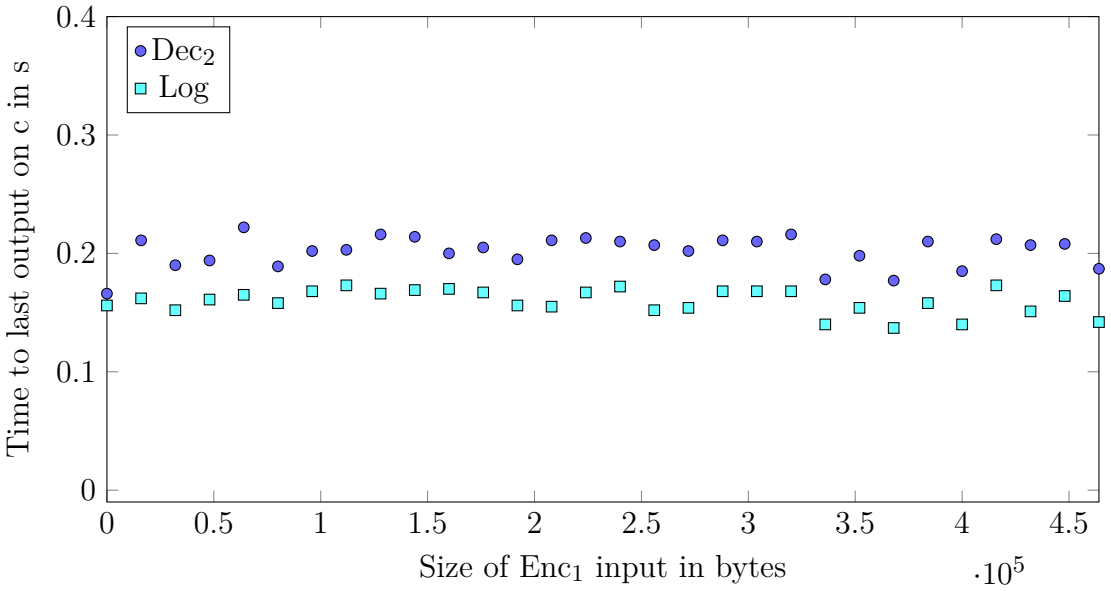


Figure 5.4: Protection against timing-attacks

here.

The changes of the output on Log level are the same for all other cases. As the Log-level execution is not allowed to obtain any information from the key or encryption levels, it always produces the same result. Consequently, this design ensures that no information can leak through implicit or explicit flows on the Log-level. Thus, the originally leaking cases two and three are also noninterferent under our enforcement. Furthermore, because of the independent scheduling, the Log-level execution terminates independently of the decryption progress. Thus, the leaks on the termination channel in cases four to six are also mitigated. For the same reason the termination-channel leak from the first decryption to the second decryption in case five is mitigated. Thus, as demonstrated here, our enforcement ensures practical protection against attacks via implicit, explicit and termination-sensitive leaks. Still, the information on the Log-level provides functional information about the number of processed inputs.

Case seven represents attacks on the timing-channel. In the original execution, the length of the first ciphertext delays the processing of all subsequent output. Thus, as shown in Figure 4.3, there exists a direct correlation between the length of the first ciphertext and the added delay in the timestamps of the outputs on the the Dec_2- and Log-level channels. These represent timing-sensitive leaks. Users authorized to obtain the second ciphertext but not the first could infer the length of the first ciphertext. Additionally, users with access only to the log file could also

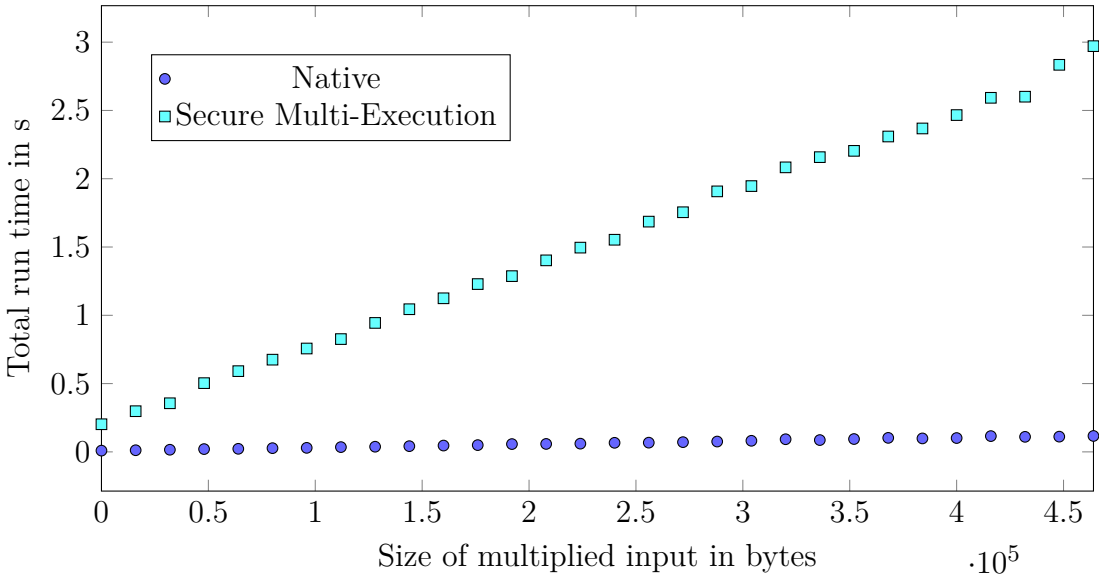


Figure 5.5: SME Efficiency

infer the length of the first ciphertext. We consider these leaks *timing-sensitive* as the effect is strong enough to be measured on a real system, where delays are also introduced due to other mechanics.

As shown in Figure 5.4, this correlation is resolved using our enforcement. Increasing size of the first ciphertext does not lead to a direct correlation with any of the timestamps. This is because the Log, Dec_1, and Dec_2 output are all produced by individual executions, running in parallel, with the Dec_2- and Log-level executions unaware of the Enc_1 input. Thus, observers of the log channel or the second decryption channel cannot infer the length of the first ciphertext under our enforcement. Note that due to the overhead introduced by our enforcement, the noise is also magnified. As the multiplied executions content for resources on the system, the increased load is reflected in the noise. Thus, while attacks cannot leak sensitive information through our enforcement, they potentially can notice that enforcement is happening. We do not consider this an insecurity with respect to confidentiality. Therefore, we consider our practical enforcement secure and transparent, as reflected in the results presented here.

Yet, the increased noise hints at a fundamental problem of Secure Multi-Execution, namely its inefficiency. Due to the multiplied execution, also the resource requirements of the target application are multiplied. Even when the multiple executions are executed concurrently, the additional stress on the system leads to an increasing run time. This is shown in Figure 5.5, where we compare the run times of native execution versus execution under enforcement. As is clearly

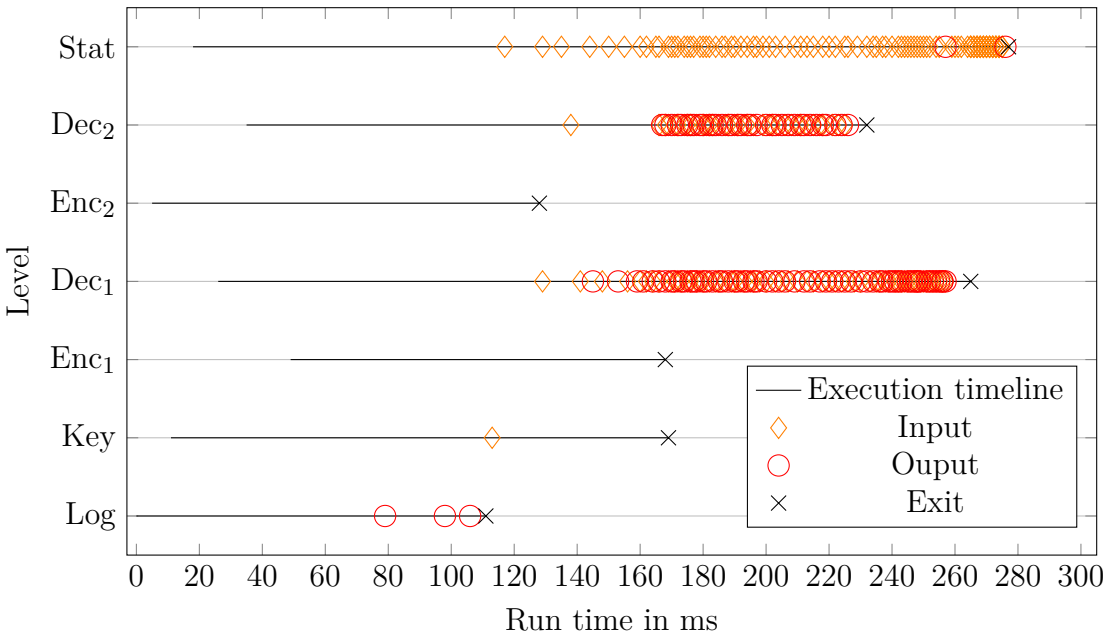


Figure 5.6: Visualization of Secure Multi-Execution applied to the example

shown, the enforcement overhead increases several times with increasing size of the input, compared to the native execution.

The origins of this inefficiency is best seen in the visualized trace of the enforced execution given in Figure 5.6. Here, we show the run-time behavior of the replicated execution for each level. To illustrate the protection against timing-attacks, we used two ciphertexts that differ in length. In our example, the first ciphtertext is twice as long as the second ciphtertext (16.000 versus 8.000 bytes, respectively). As the graph shows, the Log level execution starts running first and attempts to read in the key. Because it does not have access to the Key channel, we replace the input with empty dummy information (not shown). Consequently, reading in encrypted information is skipped in the Log-level execution. Instead, the error codes are emitted in corresponding output events and the execution terminates. Concurrently, the encryption-level executions are started. They also do not get access to the Key information, but additionally, they are also not allowed to write to the Log channel. Thus, they are terminated without emitting output. This is also true in the Key-level execution, which does obtain the key, but is not allowed to access the encrypted information.

As visualized, these four levels terminate very fast. Furthermore, although the decryption level executions perform most of the work, they run concurrently under Secure Multi-Execution, which assures timing-sensitivity. Here, we can also see

the effect of the inner decryption loop. Because the input is processed in chunks, input and output events alternate until all information is processed, A bigger issue is shown in the Stat-level execution. Being the top-level execution, this execution has access to the key and the encryption information. Thus, it recomputes the results of both decryption level executions, but has no access to the decryption channels. The recomputed results are only used to emit the statistical information at the end. To produce these results, the Stat-level execution effectively performs a full run of the original program. Therefore, all other executions become surplus, adding to the enforcement overhead.

5.4 Summary

To achieve timing-sensitive noninterference for machine code, we provide an enforcement system based on Secure Multi-Execution. We implement the necessary changes to the execution semantics using a monitoring approach. Through interception and manipulation of system calls at run time, we can instantiate local SME semantics for executions running compiled code. We demonstrate the effectiveness of our approach with the running example. Here, we show that our enforcement system achieves timing-sensitive noninterference and per-channel transparency. Yet, we also visualize the enforcement overhead, inherent to multi-execution. To tackle this, we introduce our demand-driven optimizations for Secure Multi-Execution next.

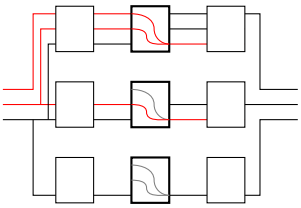
Chapter 6

Dynamic Instancing Optimization

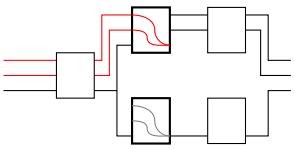
With the application of Secure Multi-Execution (SME) to machine code, we achieve a secure and transparent confidentiality enforcement method. Yet, due to the multiplied executions inherent to SME, this method comes with a high enforcement overhead. To counter this, we introduce a novel optimization to reduce the amount of redundancy and thereby increase the efficiency of our approach.

The key idea to our Dynamic Instancing Optimization is to create as few execution as possible as late as possible. Instead of initially starting one execution per level in the security lattice, we aim to create new executions only when input from an hitherto unserviced level is obtained. In other words, we emulate multiple executions with fewer executions as long as possible, thereby reducing the overhead. The effect is illustrated in Figure 6.1, where the multiplication of the first component can be saved. In this scenario, we further assume that input is only obtained on two out of three levels in security lattice. Our optimization allows to adapt the actual demand, saving additional resources.

The correctness of the emulation follows from the determinism of our target programs. Due to the determinism, two executions starting in the same state progress through the same states as long as they obtain the same input. Under Secure Multi-Execution, all executions with classification higher or equal classification as the input get the same input, while all others also get the same dummy input. Thus, newly arriving input splits the executions into two equivalence classes, those who obtained the input and those who obtained dummy information instead. Out of each of these classes, one execution can be chosen as a representative. Thereby, only two new executions are needed when new input is obtained. In many cases, this leads to a great improvement of the overall enforcement efficiency.



(a) Secure Multi-Execution



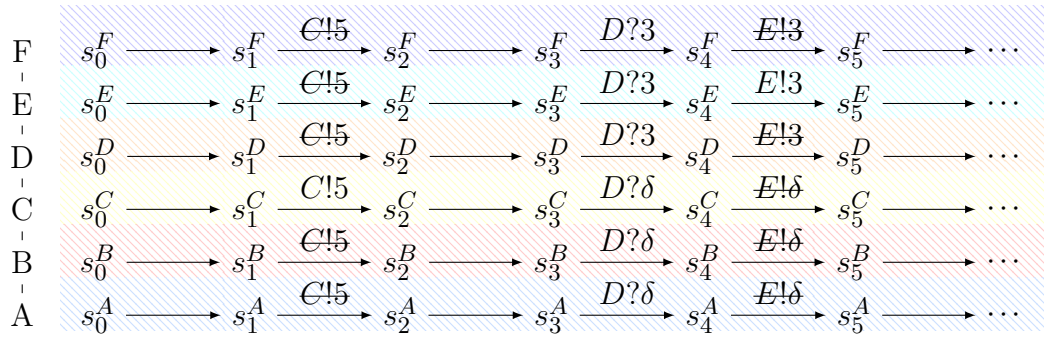
(b) With Dynamic Instancing

Figure 6.1: Comparison of SME protection without and with Dynamic Instancing

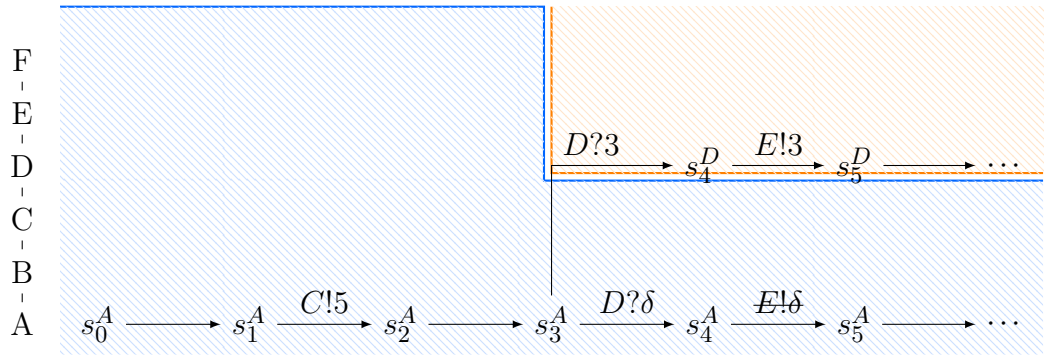
6.1 Reasoning

We illustrate the idea in Figure 6.2. In Figure 6.2a we demonstrate an exemplary run of Secure Multi-Execution with 6 levels. All executions start in the same state s_0. Because the underlying execution is deterministic, all executions progress through the same states s_0 to s_3. This includes output on channel C, which is performed by the respective execution as per design of SME. Note, however, that any of the other executions would have produced the same output, as they are all still in the exact same states. Only when new input from channel D is obtained in state s_3, the executions diverge. However, with all executions on levels lower than D obtaining the same dummy value δ, they form an equivalence class, within which they are still progressing through the same states. The same goes for the executions with classification D and higher.

In contrast to this, Figure 6.2b shows a run of Secure Multi-Execution with our Dynamic Instancing Optimization in the same setting. Here, we initially only create one execution, with the lowest level possible. This execution *emulates* multi-execution throughout the first three states, based on the observation that it contains enough information to produce the same outputs. Thus, in state s_1, it produces the correct output on channel C, emulating the corresponding execution. Only when sensitive information from channel D is obtained in state s_3 is another execution created. To guarantee security, the execution on level A cannot obtain the input from level D. On the other hand, the execution for level D cannot service levels A to C. Thus, both executions are needed to partition the responsibility for outputs on all channels between them. Thus, when output on channel E is created in state s_4, only the D execution is allowed to produce it. The A execution may not have the required information to do so. As can be seen from the illustration, the optimized enforcement requires far less executions. Still, the correct outputs are created in the right order by executions with no access to information higher than the channel that they produce output for.



(a) Unoptimized Secure Multi-Execution



(b) Dynamic Instancing Optimization

Figure 6.2: Illustration of Dynamic Instantiation

Our Dynamic Instancing semantics differ from the original secure multi-execution in three key aspects. First, individual executions are responsible for output on more than one level. Second, initially only one execution is created. And third, further executions are created during execution. Next we discuss the changes to the enforcement semantics in more detail and argue why our optimization preserves the security and transparency guarantees of SME. We then show further evidence of the correctness of our optimization by applying it to the running example. In the same context, we also show that the enforcement efficiency is increased significantly for our example run.

The efficiency can be further increased, when dynamically created executions are terminated. Thus, in the next chapter, we provide another optimization that allows to save redundancy during execution. Together with this Bounding Optimization, our two optimizations allow for very high enforcement efficiency in many cases.

6.2 Semantics

In the original semantics of Secure Multi-Execution, each execution is responsible for output on one level only. This execution has access to information from channels with equal or lower classification, ensuring secure outputs. On the other hand, because *all* levels are *always* serviced by *one* execution each, it is ensured that outputs are neither duplicated, nor missing. Thus, both the security and the transparency guarantees of Secure Multi-Execution are tightly bound to this design principle.

Since this is also the origin of the inefficiency of SME, our optimization changes this principle. In our Dynamic Instancing Optimization, existing executions are responsible for output on a *subset* of the security levels. Thereby, one optimized execution can *emulate* the output behavior of *many* unoptimized SME executions. Consequently, we do not need to always run all executions, but instead run a subset of necessary executions. This gives our optimization the potential to be more efficient than SME in many cases. However, the changes to the semantics must be applied carefully, to retain the original security and transparency guarantees.

6.2.1 Local Semantics

As can be seen in our Dynamic Instancing semantics in Semantics 6.1, we keep the changes to the local semantics as minimal as possible. Specifically, we only change the classification level of the execution from a single level ℓ in the security lattice to a sub-lattice of levels L. This change allows us to use one execution to produce outputs for all levels in the sub-lattice (see the WRITE rule).

Security of our demand-driven optimization is guaranteed by the same design principle as for Secure Multi-Execution. An execution with responsibility for outputs on the level in L can only access information from channels with equal or lower classification than the *lowest level* in L. We describe this level with the infimum of the sub-lattice L, denoted $\sqcap L$. Apart from this alteration, the rules for READ and READ-SKIP are equivalent to those of Secure Multi-Execution. Because of the restricted access, an execution that writes to levels L cannot leak information from levels greater than $\sqcap L$ through explicit or implicit flows. As it is also not allowed to write to levels outside of L, it also cannot leak to levels lower than $\sqcap L$. Leaks through termination- and timing-channels is precluded through the global scheduling of individual executions.

Transparency of SME is a result of the principle that each level is always serviced by exactly one execution with sufficient access to information. Since our

$$\textsc{Silent}\ \frac{\iota \neq \texttt{syscall} \qquad \rho, \mu, pc, \iota \xrightarrow{\bullet} \rho', \mu', pc', \iota'}{\pi, \delta, L \vdash \rho, \mu, pc, \iota \xrightarrow{\bullet} \rho', \mu', pc', \iota'}$$

$$\textsc{Read}\ \frac{\begin{array}{c}\rho \vdash (sysno, arg_1, arg_2, arg_3) \Downarrow (read, c, b, n) \\ \pi(c) \sqsubseteq (\sqcap L) \qquad \rho, \mu, pc, \texttt{syscall} \xrightarrow{c?v} \rho', \mu', pc', \iota\end{array}}{\pi, \delta, L \vdash \rho, \mu, pc, \texttt{syscall} \xrightarrow{c?v} \rho', \mu, pc', \iota}$$

$$\textsc{Read-Dummy}\ \frac{\begin{array}{c}\rho \vdash (sysno, arg_1, arg_2, arg_3) \Downarrow (read, c, b, n) \\ \pi(c) \not\sqsubseteq (\sqcap L) \qquad \rho^{\delta} = \rho[arg_1 \leftarrow \delta] \\ \rho^{\delta}, \mu, pc, \texttt{syscall} \xrightarrow{\delta?v} \rho', \mu', pc', \iota\end{array}}{\pi, \delta, L \vdash \rho, \mu, pc, \texttt{syscall} \xrightarrow{c?v} \rho', \mu', pc', \iota}$$

$$\textsc{Write}\ \frac{\begin{array}{c}\rho \vdash (sysno, arg_1, arg_2, arg_3) \Downarrow (write, c, b, n) \\ \pi(c) \in L \qquad \rho, \mu, pc, \texttt{syscall} \xrightarrow{c!v} \rho', \mu', pc', \iota\end{array}}{\pi, \delta, L \vdash \rho, \mu, pc, \texttt{syscall} \xrightarrow{c!v} \rho', \mu, pc', \iota}$$

$$\textsc{Write-Skip}\ \frac{\begin{array}{c}\rho \vdash (sysno, arg_1, arg_2, arg_3) \Downarrow (write, c, b, n) \\ \pi(c) \notin L \qquad \rho' = \rho[arg_3 \leftarrow 0] \\ \rho', \mu, pc, \texttt{syscall} \xrightarrow{\bullet} \rho'', \mu', pc', \iota \qquad \rho''' = \rho''[retval \leftarrow n]\end{array}}{\pi, \delta, L \vdash \rho, \mu, pc, \texttt{syscall} \xrightarrow{\bullet} \rho''', \mu', pc', \iota}$$

Semantics 6.1: Local Secure Multi-Execution semantics with Dynamic Instancing

Dynamic Instancing Optimization differs from SME in exactly these points, transparency is less obvious here. With some executions responsible for output on multiple levels, we have to ensure that at any time during the execution only one execution is responsible for each level at all times. Additionally, we have to ensure that the execution responsible for output at a level has access to sufficient information. Otherwise, we may either create duplicate output, miss some outputs, or create incorrect output. Together, these requirements drive our global semantics, illustrated in Semantics 6.2.

$$\textsc{Progress}\ \frac{S(\ell) \xrightarrow{a} s'}{\mathcal{L} \vdash \ell : \sigma, S \xrightarrow{a} \sigma, S[\ell \leftarrow s']}$$

$$\textsc{Enter}\ \frac{\begin{array}{c} s = S(\ell) \quad s \xrightarrow{c?v} s' \quad \pi(c) \not\sqsubseteq \sqcap(s.L) \\ \ell_{new} = \pi(c) \sqcup_{\mathcal{L}} (\sqcap(s.L)) \quad \ell_{new} \in s.L \quad (1) \\ s_l = s'[L \leftarrow \{\ell' \mid \ell' \in s.L \wedge \ell' \not\sqsupseteq \ell_{new}\}] \quad (2) \\ s_h = s[L \leftarrow \{\ell' \mid \ell' \in s.L \wedge \ell' \sqsupseteq \ell_{new}\}] \quad (3) \\ S' = S[\ell \leftarrow s_l, \ell_{new} \leftarrow s_h] \end{array}}{\mathcal{L} \vdash \ell : \sigma, S \xrightarrow{c?v} \sigma, S'}$$

Semantics 6.2: Dynamic Instantiation Semantics

6.2.2 Global Semantics

The Enter rule describes our Dynamic Instancing. Its design shapes the transparency and efficiency guarantees of our optimization. It applies whenever the scheduled execution s wants to obtain new input from a channel c with a classification $\pi(c)$ not lower or equal than what this execution has access to ($\pi(c) \not\sqsubseteq \sqcap(s.L)$). In this situation, s will not obtain the input but obtain dummy values instead. When this is the case, we first determine the lowest possible level for new execution that has a) access to all the information of s (and thus information up to level $\sqcap(s.L)$), and b) access to the new input $\pi(c)$. This level ℓ_{new} is given by the join between both accesses, as computed in (1).

Note that we compute ℓ_{new} using the join from the original lattice $\mathcal{L}$. This is because the subset of levels that the current execution s is responsible for (denoted by $s.L$) may itself not form a lattice. Since it only contains some of the levels from the original lattice, it may be that some pair of levels does not have a least upper bound within $s.L$. However, all pairs of levels must have a least upper bound in $\mathcal{L}$ by definition. The greatest lower bound of $s.L$ on the other hand must exist. In the original execution, $s.L$ is the original lattice $\mathcal{L}$ which has a greatest lower bound by definition. In executions that are created dynamically, $s.L$ is formed in a way that a unique greatest lower bound is preserved. For the higher part, it is given by ℓ_{new}, as $s_h.L$ is formed by taking all levels greater than ℓ_{new} and ℓ_{new}. For the lower part, the greatest lower bound is the same as for execution s ($\sqcap(s_l.L) = \sqcap(s.L)$) as we do not interfere with the original greatest lower bound.

Because we use the original join to compute ℓ_{new}, it might not fall within the responsibilities of s. Thus, we check if $\ell_{new} \in s.L$ and if so, we split the responsibilities $s.L$ into two portions. This represents the creation of new equivalence classes

for the executions: those with access to $\pi(c)$, represented by s_h, and those without, represented by s_l. The new execution s_h is responsible for the upper boundary with regard to ℓ_{new} of the original responsibilities $s.L$ (see (3)). To ensure that never two executions are responsible for the same level, the responsibilities of s are changed to all levels strictly not in the upper boundary of ℓ_{new} (see (2)). Finally, both executions are added to the execution mapping S, with ℓ mapping to the lower execution s_l and ℓ_{new} mapping to the higher execution s_h. Also note that s_h is created from s (and not s'), meaning before the input is obtained. As it is guaranteed that s_h has access to this input, it will obtain it next time it is scheduled.

Our global semantics ensure three important points for transparency. First, only one existing execution is responsible for output on a certain level at all times. Second, this execution has access to sufficient information to produce new output. Third, there always is at least one execution serving a level. The first follows from the way we split responsibilities. As the responsibilities of the original process are separated into to disjoint sets, there can never be two processes with joint responsibility for any level. Note that neither sets can be the empty set, for ℓ_{new} is in $s.L$ and at least $\sqcap(s.L)$ is not greater or equal to ℓ_{new}. The second follows from the way we compute the new level ℓ_{new}. Since we join the level of the input $\pi(c)$ with the greatest level of information in s (described by $\sqcap(s.L)$), the new level must have access at least to all information in s and the new input. Thus, it has sufficient information to produce outputs for levels in its responsibility. Once input with higher classification is obtained, another execution is created from this execution with even higher classification. The third point holds when the first execution is initially responsible for the complete lattice. Then, all levels are trivially serviced at the beginning. When a new execution is created from this first execution, the responsibilities are split such that again all levels are serviced by an execution. The same argument holds for each of the new executions and their sub-lattices.

Termination-sensitivity follows from the same arguments as for unoptimized Secure Multi-Execution. As long as the target is *monotonically terminating*, meaning dummy values can be found that don't alter the termination behavior of an execution, then our optimization does not adversely affect the termination behavior of the enforced execution. A major difference is that in our optimized setting, diversion in lower executions can lead to higher executions not being started. However, if the lower execution diverges due to obtained input, then the higher execution would also diverge due to the same input. As it makes no observable difference if output is not created because the responsible execution is diverged or because it has never been instantiated, we expect our optimization to preserve the termination-sensitive guarantees of Secure Multi-Execution.

The instancing of new executions at run-time has two effects on the timing-behavior. First, instancing of a new execution introduces a one-time delay while the operating system is busy cloning the process. Yet, this delay only signifies that a new execution was created because input from a higher level channel was requested. It does not leak whether input was obtained, nor how much, and nothing about its contents. Additionally, the delay only occurs the first time that input is obtained, as afterwards an execution for that level will exist. Thus, we consider the security impact of this delay as negligible. We also expect a kernel-level implementation of our technique to be able to mask the effect.

Second, the number of parallel executions may have a noticeable effect on the performance of individual executions. This is the fact when more executions are created than cores are available on the system. The presence of higher executions may then introduce slight delays in the outputs of lower executions, due to contention for resources. However, we expect this channel to be too noisy to reliably extract leak meaningful information through it. Additionally, a kernel-level implementation with more control over the scheduling should be able to mask the effect.

6.3 Implementation

Before we demonstrate the effectiveness of our Dynamic Instancing Optimization with the running example, we first discuss the necessary changes to the implementation. In particular, we present methods to dynamically clone executions and to unshare input between clones. This is necessary to resolve dependencies that arise from our cloning mechanism.

6.3.1 Fork Injection

Our Dynamic Instancing Optimization requires a mechanism to create new executions as clones from existing executions during run time. The fastest and simplest mechanism to duplicate a running process we discovered, uses the `fork` mechanism intended to spawn new threads in a Unix-like environment. This, essentially, creates a second process that is running on the same code. Unfortunately, the `fork` needs to be executed in the context of the running process. To overcome this problem, we developed a method to inject a fork into a running process.

Due to the architecture of our system, our dynamic enforcement monitor is always signaled when an execution is about to enter a `read` system call. Since the functionality of a system call is defined by an identifier value stored in a register,

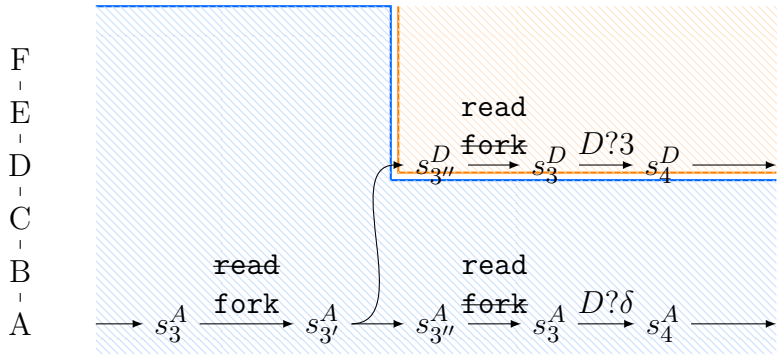


Figure 6.3: Fork injection

it can easily be altered using `ptrace`. For the fork injection, we first backup the identifier and argument values of the original system call. Then, we exchange the identifier for the `fork` ID. The system call is executed and the forked process is connected to our enforcement engine using `ptrace`. Since we replace the original `read` system call, this would effectively suppress an input event. To mask this, we rewind the program counter in the stored process context before executing the system call. When returning from the system call, the altered program counter is loaded and both processes continue from the original system call location, repeating the actual `read`. The copies are identified by unique `PID`s and equipped with a security level ℓ in the monitor.

In the example shown in 6.3, an A-level execution in state s_3 requests to `read` from a channel with higher classification D. The monitor intercepts the `read` requests and replaces it with a request to `fork`, reaching an intermediate state $s_{3'}$. When the execution continues, the operating system creates a clone of execution instead of performing the read, leading to the state $s_{3''}$. Before we pass control to the operating system, we also rewind the program counter in state $s_{3'}$, such that the system call will be requested again. In the newly created states $s_{3''}$ and $s_{3'''}$ we then reset the requested functionality from `fork` back to `read`. Thus, we end up with two executions in state s_3, where one is responsible for levels A to C and the other for levels D to F. We then perform the input operation according to SME semantics, meaning we redirect the A-execution to obtain input from the dummy channel, and allow the D-execution to obtain the regular input. Consequently, state s_4^A and s_4^D differ only in the input value that they obtained, and in the levels that they represent.

A follow-up problem from this mechanic is that cloned executions share the same file description in the Linux kernel. This means that we cannot maintain individual file offsets for cloned executions. Consequently, executions would contest for input from files, leading to a race condition. Thus, we introduce another

mechanic to unshare file offsets in the enforcement monitor.

6.3.2 Shared Input

Because we clone new executions from existing executions, both executions share the same file descriptions under Linux. This means that even when a normal file is accessed, the cursor position is advanced for all executions. Consequently, when a high and a low execution read from the same low file, each execution would only get parts of the input and skip what the other execution has read. This not only prevents transparent execution, it could also be used as a side-channel to leak information from the high execution to the low execution. If reading from the low channel depends on sensitive information, then querying the cursor position in the low execution could be sufficient to infer the sensitive information. Thus, we add a mechanic to truly unshare file accesses between multiple executions.

To achieve this, we keep copies of the cursor positions for each execution and each file in the monitor. Whenever an execution requests input from a file via the `read` system call, we instead start reading from the cursor position provided by the monitor. This can be achieved by replacing `read/readv` system calls with `pread/preadv` variants. These allow specification of a starting offset. We then intercept the amount of bytes read through the system call and advance the corresponding cursor position accordingly. This effectively hides file accesses between executions, even for normal files. As streams are generally buffered in the monitor, no additional handling is necessary. Naturally, writing requests would have to be treated similarly. Yet, in our scenarios, we treat outputs as streams and thus do not support writing at offsets.

Currently, we consider this solution to be the most efficient, as it does not require buffering of inputs in the monitor. However, in the future, kernel-level support for individual file descriptions should make these adaptions obsolete and simplify our enforcement system. Next, we demonstrate the effectiveness of our optimization on the running example. To increase the enforcement efficiency even further, we then introduce another optimization in Chapter 7.

6.4 Example

To demonstrate the effectiveness of our optimization, we compare the run times of unprotected execution, unoptimized SME, and SME with Dynamic Instancing of the example from Chapter 4. The graph in Figure 6.4 shows how the run time increases with the amount of input to be decrypted. It shows that our optimization

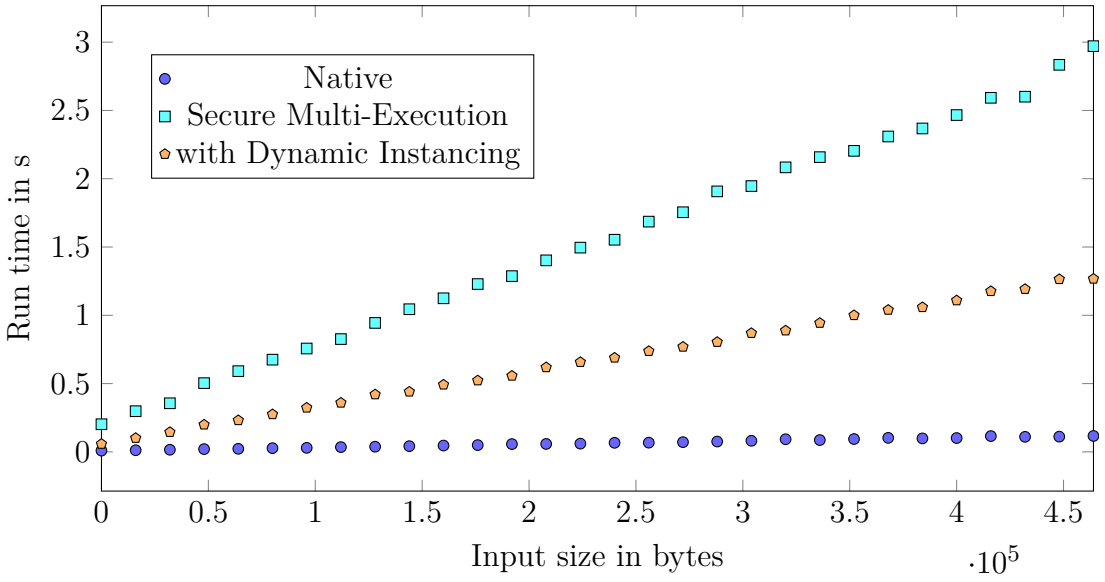


Figure 6.4: Efficiency improvement of Dynamic Instancing

is more than twice as fast as the unoptimized SME execution. The reason for this increase in efficiency becomes apparent when comparing the visualization of the optimized run in Figure 6.5, with that of the unoptimized run in Figure 5.6.

Figure 6.5 shows both aspects of the optimization. First, the initial part of the program, up to obtaining the key information, is not replicated at all. Unlike the unoptimized SME version this part is run just once. Once the key is read in, the execution is duplicated. However, even then only two executions (instead of seven) exist. The responsibilities are split, such that the execution at Key level is also responsible for the decryption and Stat levels, whereas the execution at Log level remains responsible for output on the encryption levels. Yet, as the execution at Log level does not obtain a key, it immediately returns from the decryption library, writing an error code to the log. This change in the semantics is deliberate, and removes the explicit and implicit flow present in the original program.

Meanwhile the execution at Key level requires encryption information. However, as the Enc and Key levels are incomparable, neither the Key execution should obtain the encrypted information, nor can we create an encrypted execution from an execution that holds key information. Thus, as described in our global semantics, we set the level for the new execution to the corresponding decryption level (computed by joining the Key level with the corresponding encryption level). This also means that no execution is ever created for the input level Enc_1, saving additional resources.

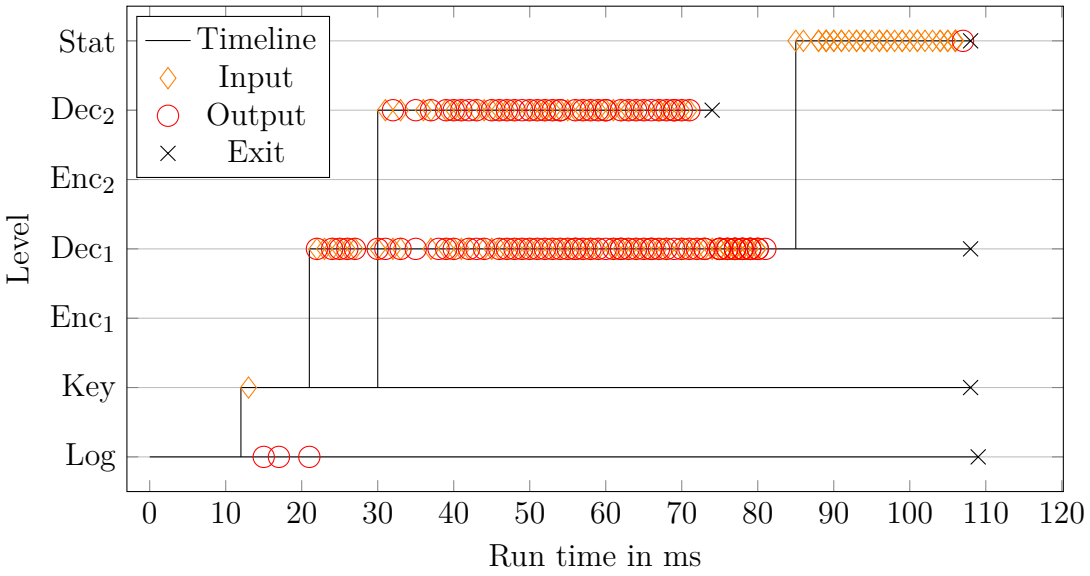


Figure 6.5: Optimized run of the example

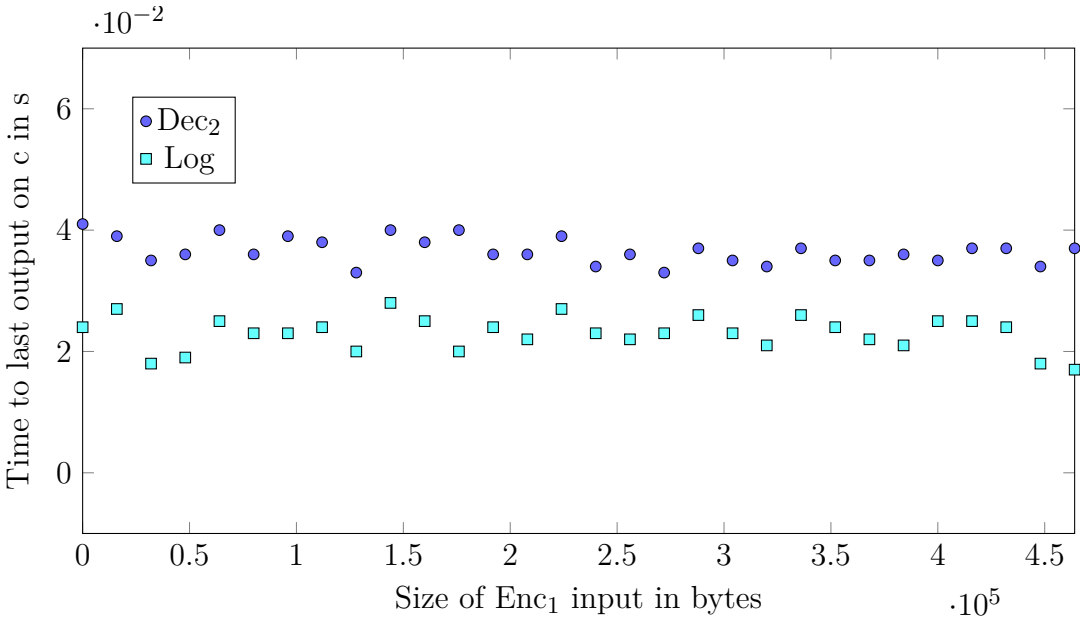


Figure 6.6: Protection against timing-attacks with Dynamic Instancing

	Inputs			*Outputs*			
	Key	Enc_1	Enc_2	Dec_1	Dec_2	Log	Stat
1	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
2	abcd	ikoupgnp	epjunvutv	ilqxphps	eqlxnwwwv	-1 -1 .	8 9
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
3		ikoupgnp	epjunvutv			-1 -1 .	
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
4	0	ikoupgnp	epjunvutv			-1 -1 .	
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
5	key	0	epjunvutv		othertext	-1 -1 .	
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
6	key	ikoupgnp	0	sometext		-1 -1 .	8
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
7	key	ikou...	epjunvutv	some...	othertext	-1 -1 .	8000 9
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9

Table 6.1: Progress-sensitive noninterference of the example in Chapter 4 through SME with Dynamic Instancing.

After creating the Dec_1 execution, the execution on Key level is now only responsible for Key information and the Dec_2 level. Conversely, the Dec_1 execution is responsible for output on the Stat level as well. Consequently, the Dec_2 level is created from the key information, but not responsible for output on the Stat level. Instead, when the Dec_1 level requests Enc_2 input, a new execution on Stat level is created, that recomputes the result for Dec_1 to output the corresponding message on the Stat channel. While this is not optimal in terms of efficiency, it is important that only one execution creates the output on Stat. Thus the Dec_2 execution cannot also be responsible for the Stat level. Because it is not known whether the information emitted to Stat requires knowledge of both Dec_1 and Dec_2 or not, it is unavoidable to eventually create an execution with access to both. Note, however, that in the unoptimized execution, the Stat execution must recompute even the Dec_1 information.

The effect of our optimization on the security and transparency of the enforcement is shown in Table 6.1. As presented in the table, our Dynamic Instancing Optimization does not alter the results of the enforcement, compared to unoptimized SME (cf. Table 5.1). It also achieves progress-sensitive noninterference and per-channel transparency. As Figure 6.6 shows, our optimization furthermore achieves timing-sensitive noninterference. Again, the timestamps of the first outputs on the various channels do not correlate with the length of the first ciphertext

under our enforcement.

6.5 Summary

Our Dynamic Instancing Optimization addresses the efficiency problem of our Secure Multi-Execution based confidentiality enforcement system for machine code. Instead of creating all executions at the beginning, we create new executions dynamically at run time. This allows us to create them only when needed, instead of always creating one execution for each level in the security level. Consequently, we can often use less executions for a large part of the execution and still get the same strong security and transparency results. This is reflected in example results, which optimization dramatically improves the efficiency.

Yet, there is still potential for even more improvement. Dynamically created executions can become obsolete when the information stored therein is no longer live. Then, computations from these executions could also be provided by another execution, which would allow to save additional resources. To make use of this effect, we next introduce another optimization, with the goal to terminate spawned executions as soon as possible. Through our method called *bounding*, we can increase the efficiency even further, by dynamically reducing the number of executions during the run.

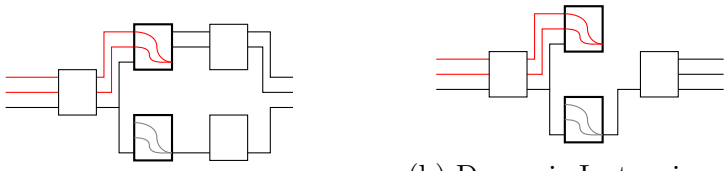
Chapter 7

Bounding Optimization

Our Dynamic Instancing Optimization for Secure Multi-Execution leads to a considerable reduction of the overhead by creating as few executions as possible as late as possible. This saves redundant computations at the *beginning* of a program. Yet, Dynamic Instancing does not help to save redundant computations at the end of the program, leaving potential for increased efficiency unused. Thus, we introduce an additional optimization, called *bounding*, that reduces redundancy at the *end* of the program.

The inspiration for our optimization is illustrated in Figure 7.1. In Figure 7.1a, we show the effect of our Dynamic Instancing Optimization. In this example, redundant computation of the initial, secure component can be saved. However, once sensitive information is obtained in the second, insecure component, a new execution is created and the rest of the program is executed twice. This may be wasteful, if, for example, the last component would be secure again, in the sense that it does not use the sensitive information. In such a scenario, we could potentially save additional resources when the multi-execution can be fully restricted to the insecure component. Then, we could terminate multiplied executions and progress with a single execution, as implied in Figure 7.1b.

Naturally, we cannot terminate executions as longs as the information stored therein is still live. Doing so would harm the transparency of our enforcement. We demonstrate this with the decryption service example from Chapter 4. Figure 7.2 illustrates the information flows through the components in the example. Here, the ciphertext information that is processed in the decryption library is not further needed. Thus, we could terminate multiplied executions responsible for output on Enc-levels and above. The information about the decryption key, however, is reused for the next iteration of the decryption routine. Thus, we cannot terminate the Key-execution early, as we would then lose the key information for subsequent



(a) Dynamic Instancing

(b) Dynamic Instancing and Bounding

Figure 7.1: Illustration of our Bounding Optimization

decryption passes.

Our solution to this problem is to define *boundary conditions* that describe when it is safe to terminate an execution and thereby effectively erase the information mix stored therein. In theory, these conditions could be arbitrarily complex, providing an exact definition of states in which specific information is no longer live. In practice, however, we usually use merge points in the control flow of the program such as after a loop, after a function call, or after a conditional block. For example, safe boundaries in our example are marked with *inner* and *outer* in Figure 7.2. The inner boundary is at the end of the decryption loop body, defining the point where executions with levels not lower or equal to the key level can be terminated. It corresponds with line 12 in Listing 4.3. The outer boundary is after the decryption loop and describes when the execution at key level can be terminated as well. It corresponds with line 14 in Listing 4.3.

Boundaries like these could be defined by the developer. The outer boundary requires no additional information than what can be derived from the application layout about which the developer has full control. The inner boundary requires additional knowledge about whether or not the library is stateful. This information could most likely be obtained from the documentation of the library, simple tests, or other inexpensive ways. However, defining the boundary may not always be trivial. For the rest of this chapter, we assume that these boundaries exist and focus on the termination and merging mechanics needed for this optimization. We return to the question of automatic boundary analysis in Chapter 9.

7.1 Synchronization

Another problem that can impede the early termination of redundant executions is late output to channels with higher classification. According to the Secure Multi-Execution semantics, output on a channel is only allowed for executions with

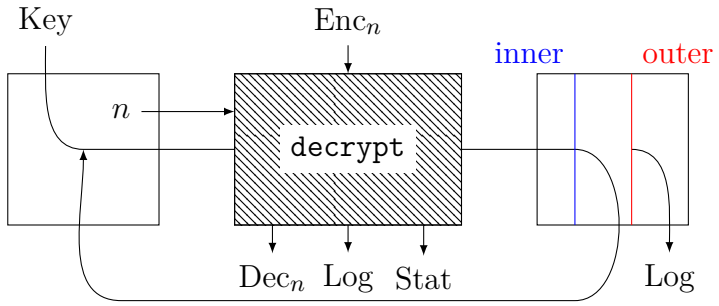


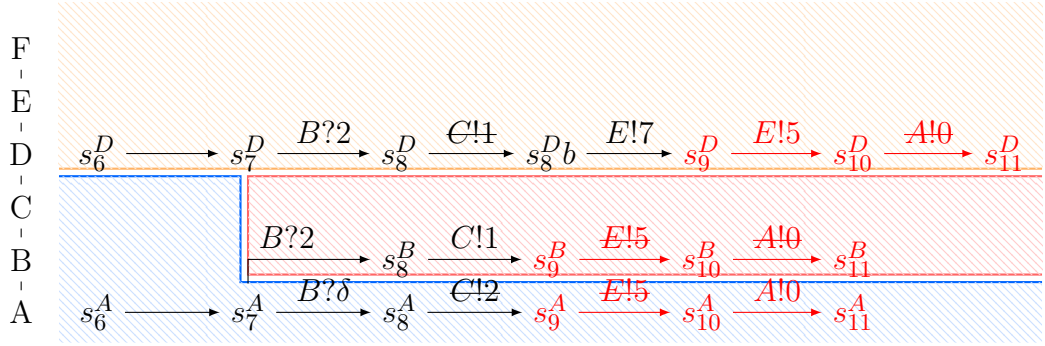
Figure 7.2: Bounds of the information flows in the example program

equal security classification. This implies that after termination of the execution responsible for output on level ℓ, no further output will be produced for that level, thwarting the transparency of the approach. Under our Dynamic Instancing semantics, executions are responsible for output on a subset of the security lattice L. Thus, premature termination of executions with Dynamic Instancing may lead to missing or incorrect output on a range of channels. Yet, since information flows from lower inputs to higher outputs are allowed under confidentiality, we must support the creation of output on higher channels that does not depend on sensitive information.

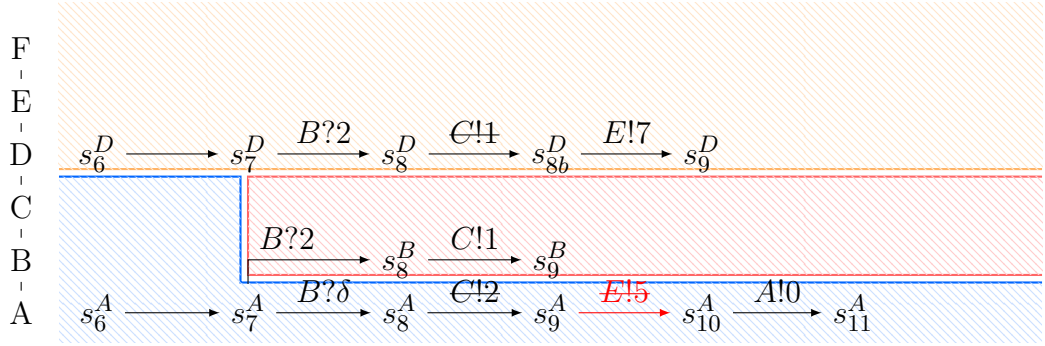
To solve these problems, we introduce two changes to our enforcement semantics. First, we add a synchronization mechanism that ensures that multiplied executions arrive at the boundary together. Then, when both executions have progressed to the barrier, we transfer the output responsibilities from the higher execution back to the lower execution before terminating the higher execution. This ensures that even though the higher execution has been terminated, its responsibilities are still serviced by an execution, namely the corresponding lower execution.

A similar synchronization mechanism is discussed for example by Rafnsson and Sabelfeld for their definition of fully-transparent Secure Multi-Execution [65] or by Schmitz et al. in their definition of faceted Secure Multi-Execution [71]. While it is a relatively simple mechanism that ensures transparent enforcement, it is known to impact the termination- and timing-sensitive security guarantees of Secure Multi-Execution. Thus, for scenarios where termination- and timing-sensitivity is needed, we present a more elaborate scheme in the next chapter.

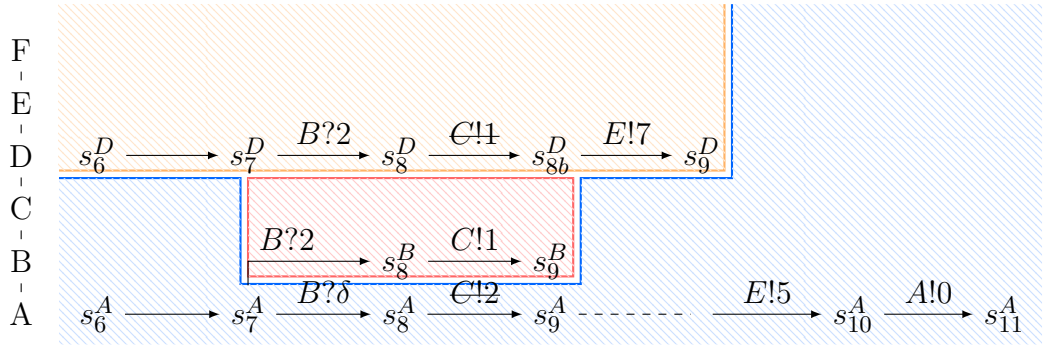
Figure 7.3 illustrates the problem. In Figure 7.3a we show exemplary traces that are the result of Secure Multi-Execution enforcement with our Dynamic Instancing Optimization. In some state s_6, we assume that the execution has already been multiplied once, due to sensitive input with classification D. Then, in state



(a) End traces with Dynamic Instancing



(b) Early termination with boundary in state s_9



(c) Early termination with level reset and barrier synchronization

Figure 7.3: Illustration of our Bounding Optimization

s_7, input with classification B is required, which leads to the creation of another execution. Thus execution at B level takes over the responsibility for output on levels B and C, as described by our Dynamic Instancing semantics (cf. Section 6.2). The executions continue according to Secure Multi-Execution semantics with different information. The D-execution, having obtained the D-input, subsequently produces an additional output on the E-level before state s_9. The other executions, unaware of the D-input, omit this output and progress faster than the D-execution. After state s_9, however, all executions perform the same trace, reflecting that this part of the execution is independent of sensitive information. Consequently, the multiplied execution of states nine to eleven is redundant and could be saved through early termination.

In Figure 7.3b we show the effect if we terminated the redundant executions above level A in state s_9. As illustrated, the redundant computation for the B execution can be saved correctly. Yet, the output on the E channel has become intransparent, as the $E!5$ message is not emitted by the A-execution, following SME semantics. The underlying problem is twofold. First, the A-level execution has insufficient responsibility and is thus not allowed to produce output on the E level. Second, even if it was allowed to emit the correct message, it would be too early, potentially leading to an incorrect order of the messages on the E-channel (thereby violating per-channel transparency).

Figure 7.3c shows the traces resulting from our synchronization mechanism. Here, we assume that the state s_9 forms the *boundary* for all executions. The A and B-executions reach the boundary in states s_9^A and s_9^B, respectively, at the same time. Thus, the B execution can be terminated, transferring the responsibility for output in levels B and C back to the A execution. The A execution must then wait for the D execution to reach the boundary at state s_9^D. When the D execution terminates, the responsibilities for output are transferred back to the A execution. This recreates the initial, very efficient setup, where only one execution without any sensitive information is servicing all levels. Combined, the outputs of the traces with our Bounding Optimization are the same as those without, shown in Figure 7.3a. This shows that our Bounding Optimization does not adversely alter the observable behavior of the target.

Next, we discuss the changes to our Dynamic Instancing semantics in more detail. We then illustrate the effect of our Bounding Optimization on the running example from Chapter 4. As we discuss in the semantics and show in the example, our Bounding Optimization can introduce a timing side-channel. Therefore, we introduce a timing-sensitive scheduling approach in the next chapter that alleviates the problem. Additionally, our Bounding Optimization is parameterized by the definition of the bounding conditions. We thus discuss their automatic extrac-

tion from the target in Chapter 9, highlighting our contributions to static binary analysis.

7.2 Semantics

Our Bounding Optimization is realized in two parts, through local adaptions and global adaptions. We extend the local semantics of individual executions with handling of termination and merging conditions. Termination conditions are our mechanic to specify when to terminate an execution. Thus, we define a boundary β that, when satisfied by the current state s, leads to the termination of the execution, signaled by a termination event $\times$. Yet, once a higher execution responsible for output on certain levels is terminated, these levels would go unserviced. This could lead to intransparency, if later output to once of these level is required. Thus, we secondly require a mechanic to pass the responsibilities for output to the lower execution from which it was forked. To achieve this, we set the same boundary in the lower execution as a merging condition and record a reference to the higher execution. Thus, once the lower execution reaches the boundary, it can retake the responsibilities that it temporarily gave to the higher execution.

The mechanics are formalized in the local rules of our bounding semantics in Semantics 7.1. Boundary conditions are denoted by β, while the reference to the corresponding execution is given by its scheduling level ℓ. Because multiple executions can be created from a single execution, we keep a stack of boundaries for each execution, denoted by B. For created executions, the termination condition is placed on the bottom of the stack, such that $B = (\beta, \varepsilon)$. It can never be removed, as satisfying it leads to termination of the execution, as described by the End rule. If the boundary contains at least one additional element, we assume that this element describes a merging condition. When this condition is satisfied, the execution signals to the monitor that it requests to be joined with the execution at the recorded level. This is described by the Join rule.

Compared to the global rules for Dynamic Instancing semantics from Semantics 6.2, we add creation of the boundaries to the Enter rule and add three more rules for the barrier-styled synchronization. Additionally, we define an abstract function $\mathcal{B} : \mathit{State} \mapsto (\mathit{State} \mapsto \mathbb{B})$ that provides a boundary condition for a given state. Thereby, this function allows to provide different boundary conditions, depending on the current state. Using this function, we generate a boundary β when a new execution is instantiated, as shown by equation four in Enter. For the newly created higher execution, this boundary acts as the termination condition, forming the bottom of its boundary record stack (equation 6). For the continuing

Local rules:

$$\textsc{End}\ \frac{s \models \beta}{\Sigma, \Pi, \delta \vdash (\beta, \varepsilon), L, \pi, s, \iota \xrightarrow{\times} (\beta, \varepsilon), L, \pi, s, \iota}$$

$$\textsc{Join}\ \frac{s \models \beta}{\Sigma, \Pi, \delta \vdash (\beta, \ell) : B, L, \pi, s, \iota \xrightarrow{\ell} B, L, \pi, s, \iota}$$

Global rules:

$$\textsc{Progress}\ \frac{S(\ell) \xrightarrow{a} s' \qquad a \notin (dom(S) \cup \{\times\})}{\mathcal{B}, \mathcal{L} \vdash \ell.\sigma, S \xrightarrow{a} \sigma, S[\ell \mapsto s']}$$

$$\textsc{Enter}\ \frac{\begin{array}{c} s = S(\ell) \qquad s \xrightarrow{c?v} s' \qquad \pi(c) \not\sqsubseteq \sqcap(s.L) \\ \ell_{new} = \pi(c) \sqcup_{\mathcal{L}} (\sqcap(s.L)) \qquad \ell_{new} \in s.L \\ s_l = s'[L \leftarrow \{\ell' \mid \ell' \in s.L \wedge \ell' \not\sqsupseteq \ell_{new}\}] \\ s_h = s[L \leftarrow \{\ell' \mid \ell' \in s.L \wedge \ell' \sqsupseteq \ell_{new}\}] \\ \beta = \mathcal{B}(s) \qquad (4) \\ s'_l = s_l[B \leftarrow (\beta, \ell_{new}) : B] \qquad (5) \\ s'_h = s_h[B \leftarrow (\beta, \varepsilon)] \qquad (6) \\ S' = S[\ell \leftarrow s'_l, \ell_{new} \leftarrow s'_h] \end{array}}{\mathcal{B}, \mathcal{L} \vdash \ell : \sigma, S \xrightarrow{c?v} \sigma, S'}$$

$$\textsc{Ended}\ \frac{S(\ell) \xrightarrow{\times}}{\mathcal{B}, \mathcal{L} \vdash \ell : \sigma, S \xrightarrow{\bullet} \sigma, S}$$

$$\textsc{Wait}\ \frac{S(\ell) \xrightarrow{\ell'} \qquad S(\ell') \xrightarrow{a} \qquad a \neq \times}{\mathcal{B}, \mathcal{L} \vdash \ell : \sigma, S \xrightarrow{\bullet} \sigma, S}$$

$$\textsc{Merge}\ \frac{\begin{array}{c} S(\ell) \xrightarrow{\ell'} s \qquad S(\ell') \xrightarrow{\times} \\ s' = s[L \leftarrow s.L \cup S(\ell').L] \end{array}}{\mathcal{B}, \mathcal{L} \vdash \ell : \sigma, S \xrightarrow{\bullet} \sigma, S[\ell \leftarrow s']}$$

Semantics 7.1: Barrier-Based Bounding Semantics

lower execution, this boundary is added on top of the boundary record stack, indicating that the execution on level ℓ_{new} is the most recent execution created from it (equation 5).

When an execution that satisfies the provided termination condition (and thus emits a termination signal $\times$) is scheduled, we do not update its state. Thus, it remains in the terminating state until replaced by another execution, according to the ENDED rule. Hence, when the corresponding lower execution requests to join with this execution, we check its termination status. If it is not terminated yet, it may still produce output on higher levels. Thus, the lower execution should not progress with reset responsibilities yet. We use the WAIT rule to remain in the current state in this situation. However, if the lower execution requests to join with a terminated execution, we merge the responsibilities into the lower execution and progress as usual (see rule MERGE).

7.3 Implementation

Our monitor handles the setting and checking of boundaries. To do so in an efficient manner, we use breakpoints to track where boundary conditions should be checked. In our case, boundaries usually coincide with specific merging points in the control flow of the target and are thus placed at these points. Additionally, we implement a mechanic to differentiate between different inputs, even when they are requested through the same `syscall` function.

7.3.1 Setting Boundaries

Our semantics suggest that the boundary conditions are evaluated at every step. In practice, evaluating the condition after every instruction of the target would severely worsen the efficiency of the enforcement. Thus, we instead evaluate the condition only at specified code locations. We achieve this by setting breakpoints in the target, and connect the boundary condition as a callback function in our monitor. This allows us to continue executions without monitoring overhead until either a breakpoint or call to the operating system is encountered.

Furthermore, it could happen that multiple boundaries must be evaluated on the same location. As outlined in our semantics, processes hold stacks of boundary conditions to allow nesting of boundaries. For example, a high execution may be created from a low execution with a boundary at symbolic location A. Thus, a breakpoint is set in both executions at location A and connected with the bounding condition. Subsequently, an even higher execution may be created from the

```
1 x := in(A);
2 out(x, B);
3 y := in(C);
4 out(y, D);
5 ...
```

Listing 7.1: Pseudo-code

high execution and also be bounded at location A. Then, in the high execution, two boundary conditions would now hold at the same location. First comes the condition to signal merging with the highest execution, then comes the condition to be merged with the lower execution. Thus, we connect each breakpoint not just with one bounding condition, but rather with a stack of bounding conditions. Upon meeting the breakpoint, the topmost (last added) condition is evaluated first. After handling the topmost condition, the next condition is popped from the stack, simulating a repeated arrival at the breakpoint.

7.3.2 Identifying Input

In the semantics, we also assume a function $\mathcal{B}$ that provides boundary conditions for a given state. The idea is to allow different boundary conditions depending on the conditions under which a new execution is created. A simple example is given by a program that obtains inputs from different channels and services multiple output channels, as shown in Listing 7.1. Here, static analysis would show that information stored in variable x are not live after execution of line 2. Thus, executions created due to the input in line 1 could be terminated upon reaching line 3 to save resources. Executions created due to the input on line 3, on the other hand, should be terminated after execution of line 4. This illustrates a situation where different boundaries should be provided for different states. Here, $\mathcal{B}$ might be defined as

$$\mathcal{B}(s) := \begin{cases} \lambda s' \,.\, (s'.pc) == 3, & \text{if } s.pc == 1 \\ \lambda s' \,.\, (s'.pc) == 5, & \text{if } s.pc == 3 \end{cases}$$

In machine code, however, input is obtained through system calls as described in Section 4.2. As discussed in Section 4.2, often a single `syscall` instruction is used for all input requests to reduce the attack surface. This means that the program counter of the execution obtaining the input will show the location of the `syscall` instruction, instead of the location where the input was originally

requested. Consequently, the program counters of the two calls to `in` shown in Listing 7.1 would be indistinguishable.

We solve this problem by using return values left on the stack instead of the program counter. Assuming that the distance to the top of the stack of the return value is known when the system call is intercepted, we can peek its value and use it to discriminate different input requests. For example, assuming that the return value is the topmost value on the stack (thus at $sp+1$, as sp always points to the next empty cell) and that the calls to `in` return to the next line, we could define $\mathcal{B}$ as

$$\mathcal{B}(s) := \begin{cases} \lambda s' . (s'.pc) = 3, & \text{if } s.\mu[s.sp+1] = 2 \\ \lambda s' . (s'.pc) = 5, & \text{if } s.\mu[s.sp+1] = 4 \end{cases}$$

This shows the strengths of the flexible definition of $\mathcal{B}$. As it takes the complete state as input, bounding conditions can be provided for various different situations. Next, we show how our Bounding Optimization can be used to increase the efficiency of confidentiality enforcement for our example from 4.

7.4 Example

Considering the information flow schematic of the example from Chapter 4, replicated in Figure 7.2, it is clear that the final logging output is independent of any information beyond Log level. Therefore, running any execution with information higher than Log level at this point is unnecessary. Consequently, we consider the execution point between the loop and the output to log level as a *boundary* for any execution higher than Log level. This means that we terminate any execution higher than log level once it progresses to this point.

On the other hand, the outputs created by the decryption library are dependent on the key information as well as information from one encryption channel. Yet, later iterations of the library do not depend on information from previous iterations. This means that executions on levels not lower or equal to Key can be terminated upon return from the library. Thus, we add another, *inner* boundary for executions with information not lower or equal to Key level at this point. Note that this inner boundary is the key to saving the redundant recomputation of the statistical information. This can be seen by comparing Figure 6.5, optimized without bounding, and Figure 7.4, optimized with bounding. Knowing that the statistical information for the second decryption does not depend on information from the first decryption means that we do not need an additional execution on

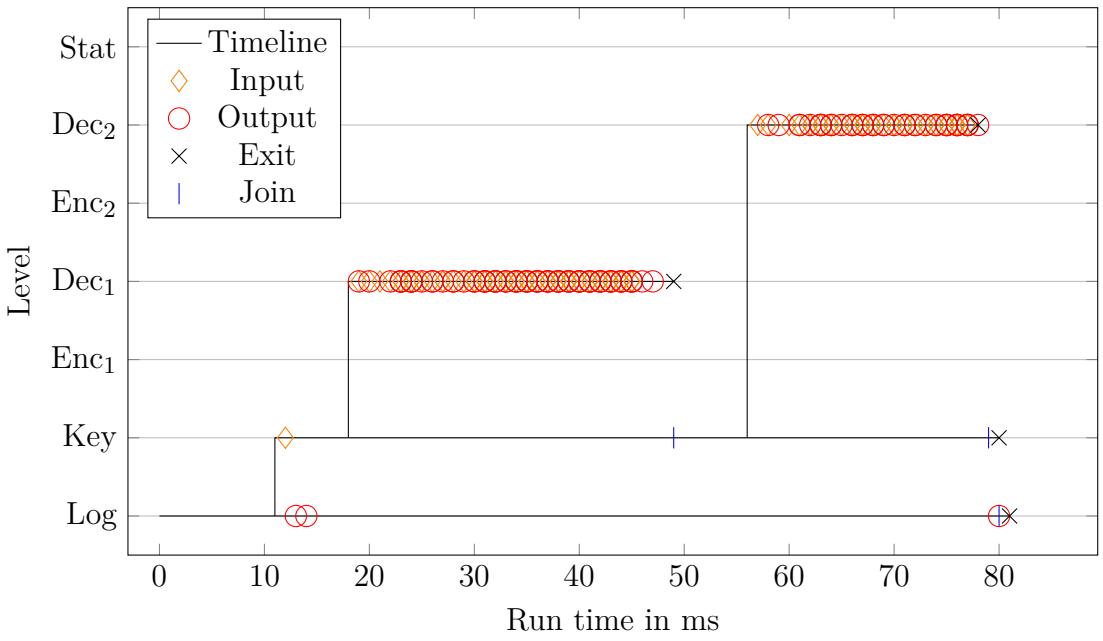


Figure 7.4: Bounded run of the example

Stat level, but can create correct information from the decryption levels. Our barrier-style synchronization further ensures that the Dec_1-execution must be terminated before the Dec_2-execution is started. This ensures that the Stat-level output is emitted from the two executions in correct order. Thus, in our example, we define the boundary as follows.

$$\mathcal{B}(s) := \begin{cases} \lambda s'.(s'.pc) = 14, & \text{if } s.\mu[s.sp+1] = 6 \\ \lambda s'.(s'.pc) = 12, & \text{if } s.\mu[s.sp+1] = 37 \end{cases}$$

As visualized in Figure 7.4, our Bounding Optimization can be used to save even more resources than the Dynamic Instancing Optimization alone. This effect is also shown in Figure 7.5, where the bounded demand-driven optimization saves roughly half of the optimized run time. Compared to unoptimized Secure Multi-Execution, this is a speed up of more than 450%. When compared to Figure 6.5, the speedup becomes apparent. Because the first decryption process reaches the boundary after executing the inner loop body, it is terminated. Additionally, its output responsibilities are merged back into the Key-level execution. Consequently, the Dec_2-level execution that is subsequently forked from the Key-level execution can inherit the responsibility for output on the Stat-channel. As a result, the statistical information does not have to be recomputed, saving the creation of a Stat-level execution.

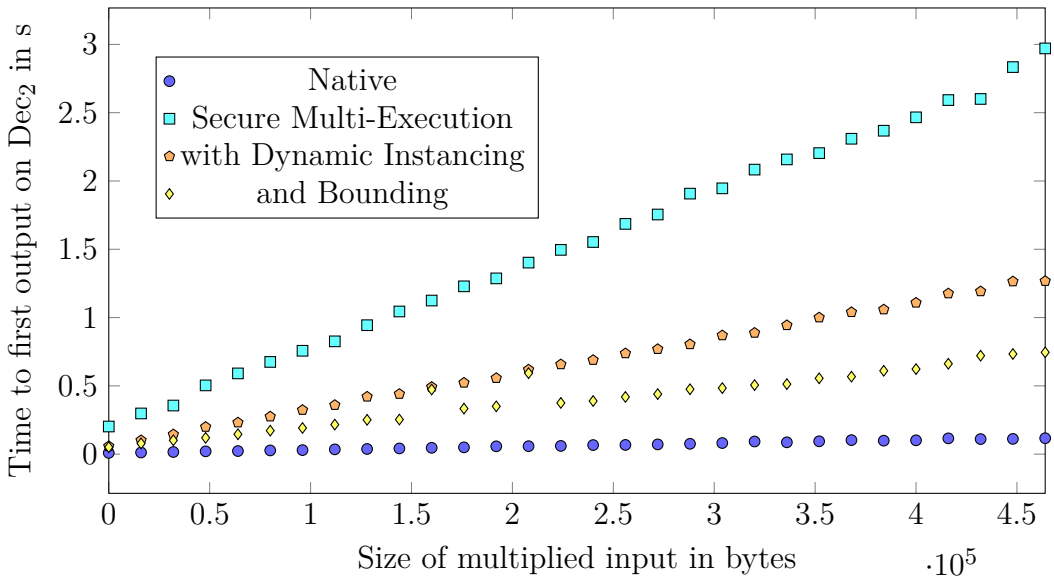


Figure 7.5: Efficiency improvement of our Bounding Optimization

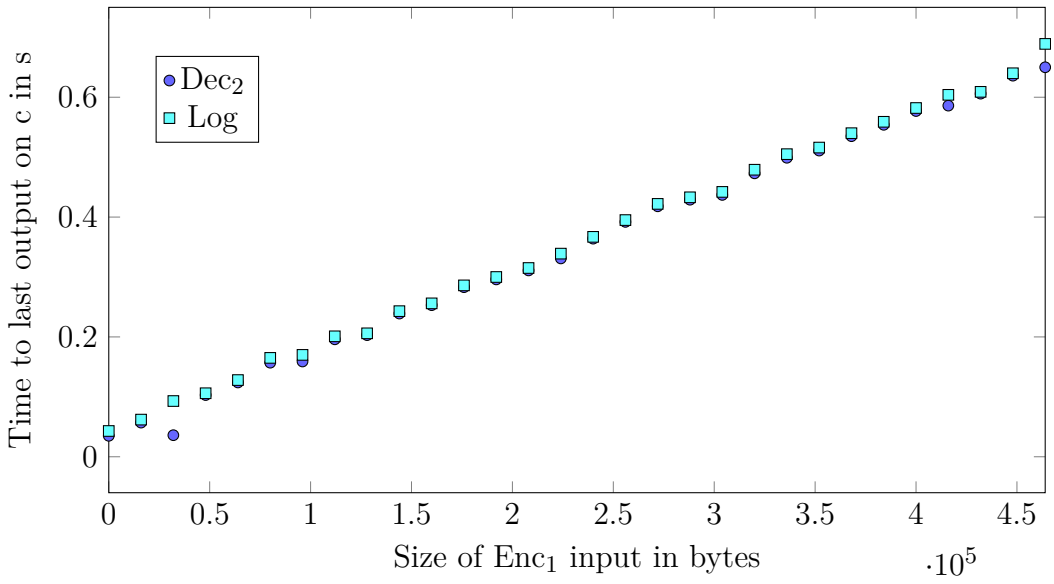


Figure 7.6: Timing-Attacks on SME with bounding

	Inputs			*Outputs*			
	Key	Enc_1	Enc_2	Dec_1	Dec_2	Log	Stat
1	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
2	abcd	ikoupgnp	epjunvutv	ilqxphps	eqlxnwwwv	-1 -1 .	8 9
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
3		ikoupgnp	epjunvutv			-1 -1 .	
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
4	0	ikoupgnp	epjunvutv			-1 -1	
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
5	key	0	epjunvutv			-1 -1	
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
6	key	ikoupgnp	0	sometext		-1 -1	8
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
7	key	ikou...	epjunvutv	some...	othertext	-1 -1 .	8000 9
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9

Table 7.1: Timing-insensitive results with Bounding Optimization

Our timing-insensitive security guarantees are shown in Table 7.1, while our timing-sensitive guarantees are shown in Figure 7.6. The results demonstrate the downside of our barrier-style synchronization. While it is simple to implement and achieves the best efficiency, it cannot guarantee termination- and timing-sensitive noninterference. Because lower executions may have to wait for higher executions to reach the barrier, our synchronization introduces a dependency between these executions. This is shown in Case 4, where the non-termination of the Key-level execution leads to non-termination of the Log-level execution. As the outer barrier is placed before the final Log-level output, but never reached by the Key-level execution, the final output will never be produced. Thus, an observer with Log-level clearance could infer non-termination of the decryption operation.

For the same reason, Cases 5 and 6 show the same problem. In Case 5, the non-termination of the decryption of the first ciphertext hinders progress of the Key-level execution as it waits forever at the inner boundary. Consequently, the second ciphertext is never processed and the final Log-level output never produced. In Case 6, the second ciphertext is processed, but the decryption does not terminate. Consequently, the final Log-level output is not produced. These results show that our barrier-style synchronization for our Bounding Optimization is not termination-sensitive. Comparing the output timestamps in Case 7, as shown in Figure 7.6, additionally shows that it is not timing-sensitive. Here, a clear correlation between size of the first ciphertext and output timestamps for the second

ciphertext becomes apparent. The same correlation shows, when considering the timestamp of the *final* output to the Log-channel.

7.5 Summary

The goal of our Bounding Optimization is to terminate multiplied executions when they become redundant. This state is signified to the executions through boundary conditions. Once the boundary becomes satisfied, the execution is terminated. Its output responsibilities are then merged into the corresponding lower execution. Boundaries are efficiently implemented through breakpoints in the target. To achieve transparency, we also introduce a simple barrier-synchronization mechanic. Our synchronization ensures that exactly one execution is responsible for output on each channel at all times.

As our results show, our Bounding Optimization roughly halves the overhead when compared to Dynamic Instancing alone, and thus roughly quarters the overhead when compared to unoptimized Secure Multi-Execution. Unfortunately, the barrier-synchronization introduces a dependency between executions. Thus, while this mechanism is very simple and most efficient, it reduces the security guarantees to termination- and timing-insensitive noninterference. We introduce a more complex but similarly efficient timing-sensitive scheduling next.

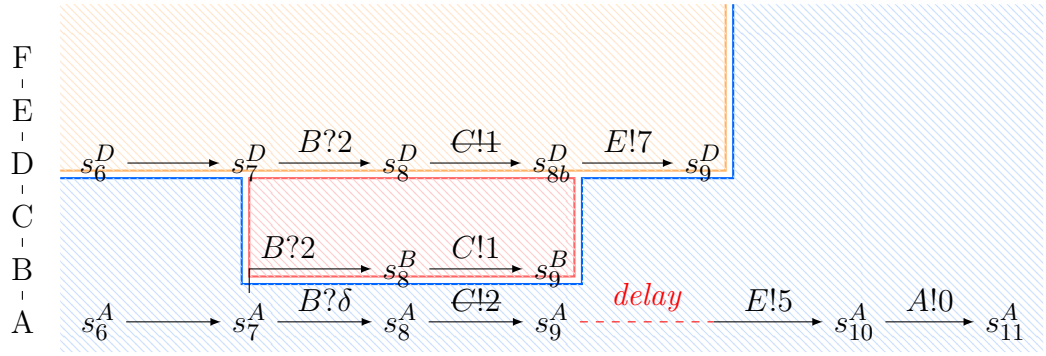
Chapter 8

Timing-Sensitive Scheduling

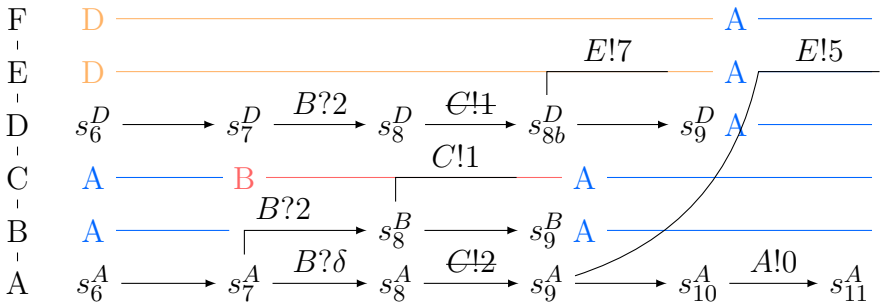
Through the application of Secure Multi-Execution to machine code, we achieve our main goals regarding security and transparency. We further achieve our goal of efficiency through our optimization techniques of Dynamic Instancing and bounding. Unfortunately, the latter optimization is in conflict with the termination- and timing-sensitive security guarantees of our approach. Bounding, when used with barrier-style synchronization, may allow information to leak through termination and timing side-channels. Thus, in this section, we provide a timing-sensitive scheduling scheme for bounded Secure Multi-Execution.

The problem is that barrier-style synchronization introduces a scheduling dependency between executions with different classifications. As a consequence, progress of executions responsible for output on channels with lower classification depends on the progress of executions with access to input from channels with higher classification. Therefore, the timing of output on lower classified channels can enable an observer to infer information about higher classified inputs. Omitting the barriers, on the other hand, can lead to executions overtaking higher executions. This could lead to out-of-order output on higher channels, thwarting the transparency guarantees.

Our solution to the problem is twofold. First, we allow lower executions to progress independently of higher executions. This breaks the termination and timing side-channels, thus enforcing real timing-sensitive noninterference. Second, we introduce reorder buffers for outputs to higher channels. This ensures that the same enforcement mechanism can also guarantee top-level transparency and per-channel transparency. The resulting mechanism allows us to use our Dynamic Instancing and Bounding Optimizations without having to sacrifice security or transparency. Due to the added complexity of the buffering, it is, however, slightly less efficient than our barrier-style scheduling.



(a) Bounding with barrier synchronization



(b) Bounding with timing-sensitive scheduling

Figure 8.1: Effect of our timing-sensitive scheduling

8.1 Reordering

We illustrate the idea behind our timing-sensitive scheduling in Figure 8.1. In Figure 8.1a we show the effect of our barrier-based synchronization, as introduced in Chapter 7. It includes an information leak on the timing-channel through the output on the channel with classification E. Here, an implicit flow in the D-execution induces an extra computation step between states s_8 and s_9. Thus, the D-execution, which is also responsible for output on the E-channel, reaches the boundary state s_9 later than the A-execution. To ensure in-order output on the E-channel, the A execution must therefore wait before producing output on the E-channel. Yet, due to the barrier-style synchronization, this delay is also noticeable in the subsequent output to the A channel, which consequently leaks information about the D-execution. Later output on the A-channel hints at a longer execution time of the D-execution, and thus at different information in these executions.

To prevent this, but still enable our Bounding Optimization, we use our timing-sensitive scheduling as demonstrated in Figure 8.1b. Unlike the barrier-based synchronization, we do not wait for the higher execution before continuing with the lower execution. Instead, we add queues to each channels that ensure correct execution order. Into these queues, we then put buffers that are connected to executions, which allow them to write ahead of the logical order, without having to wait for higher executions. In state s_6, the A-execution services all levels below D, while the D-execution services all other levels. Thus, there are two buffers connected to the A-execution queued on levels B and C, and two buffers connected to the D-execution queued on levels E and F.

When the B execution is created, the responsibility for output on the C level is transferred from the A-execution to the B-execution, according to our Dynamic Instancing semantics from Section 6.2. Consequently, the buffers for channel B and C of the A-execution are terminated and a new buffer on level C is created for execution B. Output to the C-channel from the B-execution is then redirected to this buffer. Since no other execution is servicing the C-level at that time (and therefore the buffer is on top of the queue), the output is emitted immediately. The same is true for the output to the E-channel from the D-execution in state s_{8b}.

However, when the A-execution is ready to produce output on the E-channel, its corresponding buffer is not on top of the queue. Because the D-execution is still busy servicing the E-level at that point, the E-output from the A-execution needs to be delayed. It is thus copied to the buffer of the A-execution. This allows the A-execution to progress without delay, emitting the $A!0$-message as if no sensitive information was present. Once the D-execution reaches the boundary in state s_9 and terminates, the buffers connected to the A-execution are dequeued. Then, the stored $E!5$-message is emitted, preserving the correct order of the outputs on the E-channel and thereby achieving per-channel transparency.

The main difference with the normal scheduling is that we add waiting queues to each level. We use these queues to start executions in correct logical order, as well as to insert outputs at the correct places. Next, we discuss the construction of this scheduling in more detail.

8.2 Construction

We implement the buffers illustrated above as *virtual executions*. Virtual executions are responsible for creating output on their corresponding level. Yet, they contain no code and do not produce output themselves. Instead, they hold a queue

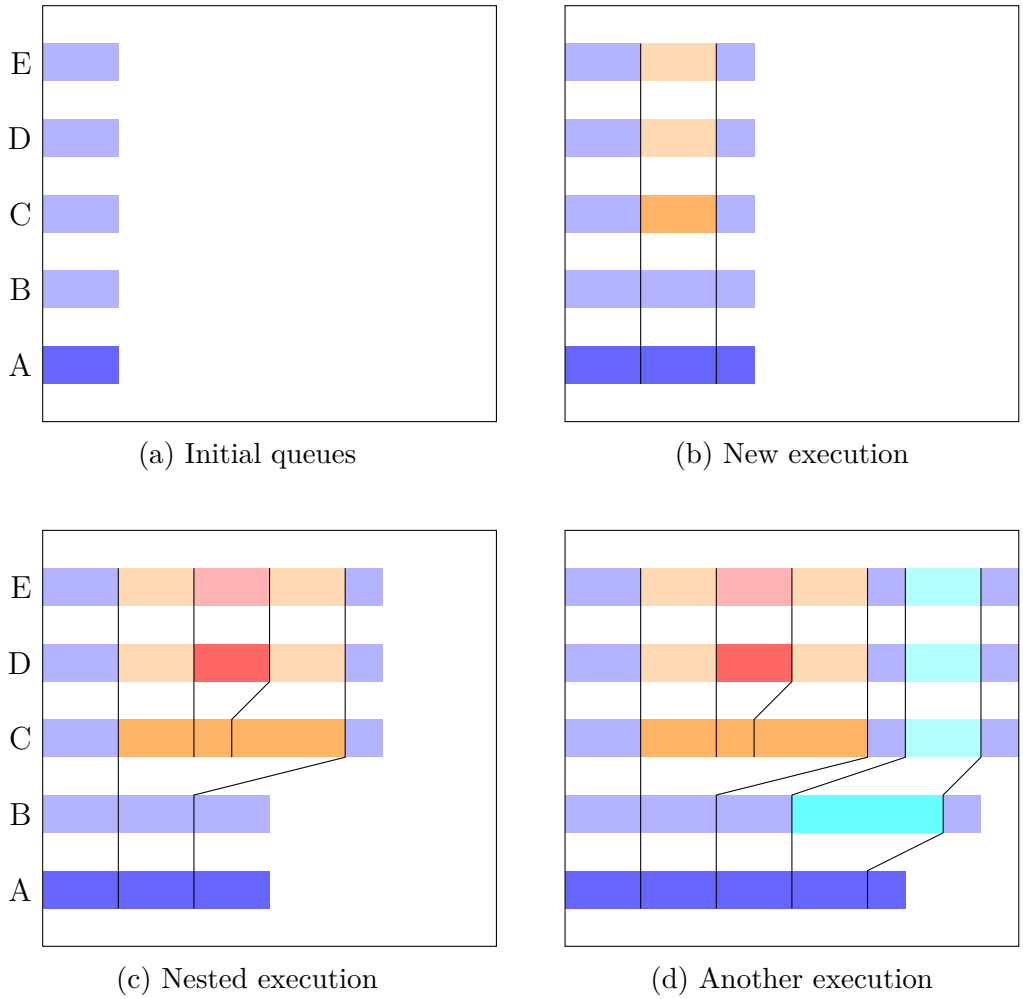
of messages forwarded to them from a corresponding real execution. While scheduled, virtual executions emit the messages in their queues or skip when the queue is empty. As a result, the virtual executions allow us buffer messages ahead of the schedule. Naturally, the introduction of virtual executions leads to adaptions in the other aspects of the scheduling as well.

As before, we initially start only one execution. Under Dynamic Instancing, this execution is responsible for output on all levels. For our timing-sensitive scheduling, we restrict this execution to produce output only on its lowest level, similar to original Secure Multi-Execution semantics. Output to higher levels is instead forwarded to connected virtual executions started on these levels. We thus start with one real execution on the lowest level and virtual executions on higher levels that act as proxies for the real execution. All executions are initially the only entry in the queue at their level and thus are allowed to progress when scheduled.

When input on a new level is obtained, a new execution is forked as described in Section 6.3.1. According to our Dynamic Instancing semantics, the new execution is responsible for output on all levels not smaller than the new level of the levels the old execution was responsible for. Thus, for all these levels, we pass the virtual executions from the old execution to the new execution. Automatically, this passes the right to produce output on these levels from the old execution to the new execution. Yet, the old execution might leave the boundary before the new executions reaches it. In this case we want to allow the old execution, which represents a lower level, to progress directly, without having to wait. However, we also need to reset the output responsibilities at this point. After leaving the boundary, the old, lower execution is again allowed to produce output on the higher levels. But, since in this case the old execution could be ahead of the new execution, its output should be emitted *after* any output produced by the new execution. Thus, we add new virtual executions for the old execution *after* the virtual executions for the new execution. Consequently, the old, lower execution can write to these virtual executions and progress without delay. However, these messages are buffered in the virtual executions until the higher execution has terminated, as the new virtual executions are enqueued behind it.

When an execution terminates, all connected virtual executions are also flagged for termination. Yet, the virtual executions may still contain buffered messages. Thus, we first empty the buffers of the virtual executions when they are on top of the queue. Only when the virtual execution is flagged for termination and its buffers are empty, then we dequeue the execution, moving the next in line to the top of the queue.

We illustrate our construction in Figure 8.2. Initially, only one actual execution is running on the lowest level, here level A. All higher levels are serviced by virtual



(a) Initial queues

(b) New execution

(c) Nested execution

(d) Another execution

Figure 8.2: Forking new executions with virtual executions

executions connected to this execution as shown in lighter color in Figure 8.2a. We then assume that a new execution is created on level C, such that new virtual executions are inserted into the queues as shown in Figure 8.2b. This new execution might then fork another execution on level D and pass its responsibilities to it, as shown in Figure 8.2c. Consequently, the virtual executions for level E is passed to the D execution and new virtual executions for the C-level execution are enqueued behind the D-executions. Due to the difference in information, it may be that the A-execution reaches the boundary faster than the C-execution. It thus regains responsibility on all higher levels and may pass them on to another new execution on B-level, as shown in Figure 8.2d. At any point, only one execution or virtual execution services each level, while lower executions can progress without having to wait.

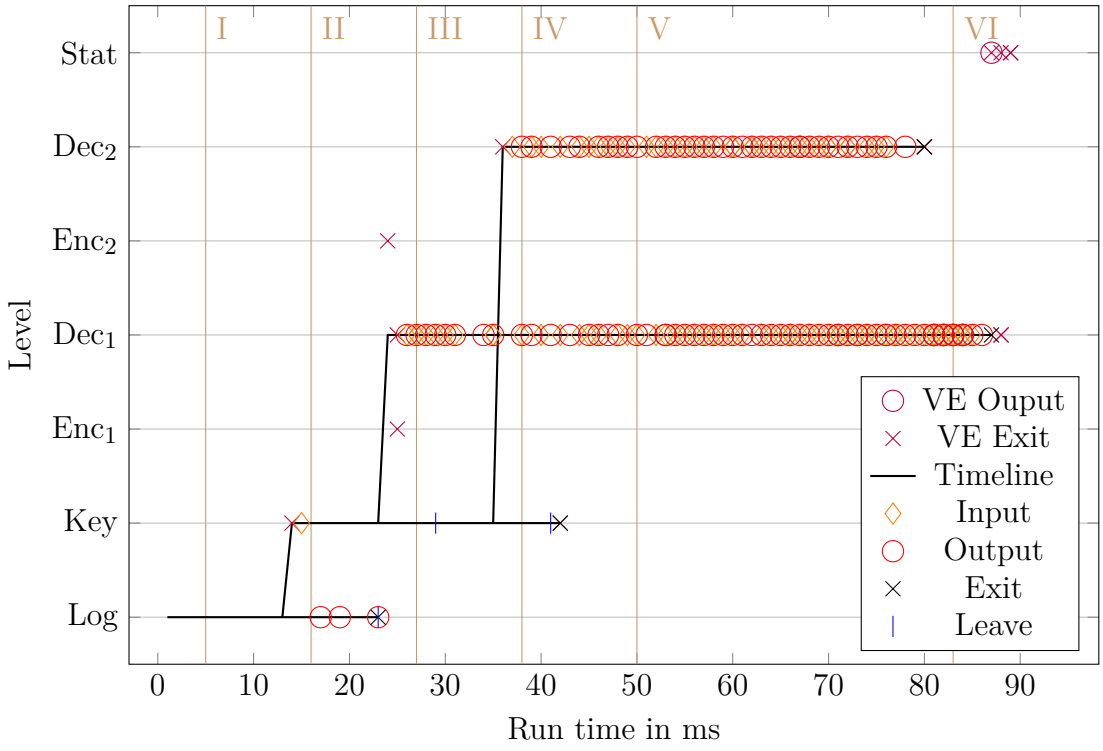


Figure 8.3: Timing-sensitive optimization of the example

In this illustration, we assumed a total lattice. Yet, in our running example from Chapter 4, we use a partially ordered lattice. To show that our method applies to this scenario as well and to demonstrate the security, transparency, and efficiency achieved by our timing-sensitive scheduling, we discuss its effect on our example next.

8.3 Example

Figure 8.3 shows an enforced run of the example in Chapter 4, using our timing-sensitive optimization. Initially, only the execution at the Log level is created. As it is the only execution, it is responsible for output on all levels. Thus, virtual executions for all levels are created and connected with the Log-level execution. This is shown at point I in Table 8.1. Capital letters describe a real executions, small letters represent virtual executions.

Upon requesting the Key information, another execution is started, leading to the situation at point II. The execution on Key level takes over the responsibility

	Queue at point					
Level	I	II	III	IV	V	VI
Stat	l	k l	d1 k l	d1 d2 k l	d1 d2 k l	d1 d2 k l
Dec_2	l	k l	k l	D2 k l	D2 k l	
Enc_2	l	l				
Dec_1	l	k l	D1 k l	D1 k l	D1 k l	D1 k l
Enc_1	l	l				
Key	l	K l	K l	K l		
Log	L	L				

Table 8.1: Queue states for the execution in Figure 8.3

for output on the decryption and Stat levels as well. Thus, the virtual executions for these levels are passed on to the Key execution. However, as now a real execution on Key level exists, the virtual execution on that level is flagged for termination. Since its buffer is empty, it is terminated immediately, scheduling in the Key execution. The Key execution obtains the Key information and proceeds. The boundary for the Key and Log execution is set to the outer boundary, as depicted in 7.2.

The Log-level execution keeps the responsibility for output on the Log and Enc_n levels. Thus, the virtual executions for these levels remain with the Log level. Additionally, since the Log-level execution may produce additional output for the higher levels after leaving the boundary, new virtual executions for the higher levels are created and enqueued after the Key execution and its connected virtual executions. The Log-level execution concurrently produces the error codes on the Log channel, as it has not obtained the Key information. It then surpasses the boundary. This termination triggers the virtual executions for the encryption levels, which remained with the Log-level execution, to be flagged for termination as well. As they are empty, they are removed from the queue. The Log-level execution then produces the final Log output and is terminated shortly before point III.

Meanwhile the Key execution requests the first ciphertext, which triggers another execution creation. The new execution at Dec_1 level is enqueued after the virtual execution form the Key level for the Dec_1 level, which is immediately terminated as it is empty. Furthermore the virtual executions for the Stat level is passed from the Key-level execution to the new Dec_1-level execution, along with the responsibility for output. New virtual executions for the Key-level execution are created and inserted behind the executions connected to the Dec_1 level. Yet, they are inserted *before* the virtual executions from the Log level, as these may

	Inputs			*Outputs*			
	Key	Enc_1	Enc_2	Dec_1	Dec_2	Log	Stat
1	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
2	abcd	ikoupgnp	epjunvutv	ilqxphps	eqlxnwwwv	-1 -1 .	8 9
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
3		ikoupgnp	epjunvutv			-1 -1 .	
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
4	0	ikoupgnp	epjunvutv			-1 -1 .	
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
5	key	0	epjunvutv		othertext	-1 -1 .	
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
6	key	ikoupgnp	0	sometext		-1 -1 .	8
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9
7	key	ikou...	epjunvutv	some...	othertext	-1 -1 .	
	key	ikoupgnp	epjunvutv	sometext	othertext	-1 -1 .	8 9

Table 8.2: PSNI-noninterference results of the example Chapter 4 for our timing-sensitive scheduler

contain output created *after* termination of the Key-level execution. The reasoning here is that the Key- and Dec-executions share the inner boundary, while the Key- and Log-execution share the outer boundary. Thus, the Key-execution will recover responsibility for Dec-level output first, before relinquishing all responsibility back to the Log-level execution.

As the Key-level execution does not obtain the actual ciphertext, it progresses to surpass the boundary while the Dec_1-level execution is running. Since we do not use barriers here, the Key-level execution is allowed to progress independently. It then requests the second ciphertext. This initiates another execution creation for the Dec_2 level. As the Key-level execution has regained responsibility for the Stat level after leaving the boundary, it now passes this responsibility on the the Dec_2 execution. Yet, the Dec_1 execution also still holds responsibility for output on Stat level, represented by a corresponding virtual execution on top of the queue. This results in the queues at point IV, where the virtual process at Stat level from the Key-level execution is split into the Dec_2 virtual execution and a new Key-level virtual execution, enqueued after the Dec_1 virtual execution.

Subsequently, the Key-level execution reaches the inner boundary and then the outer boundary, leading to its termination. Thus, all virtual executions connected to the Key level are flagged for termination and the enqueued virtual execution

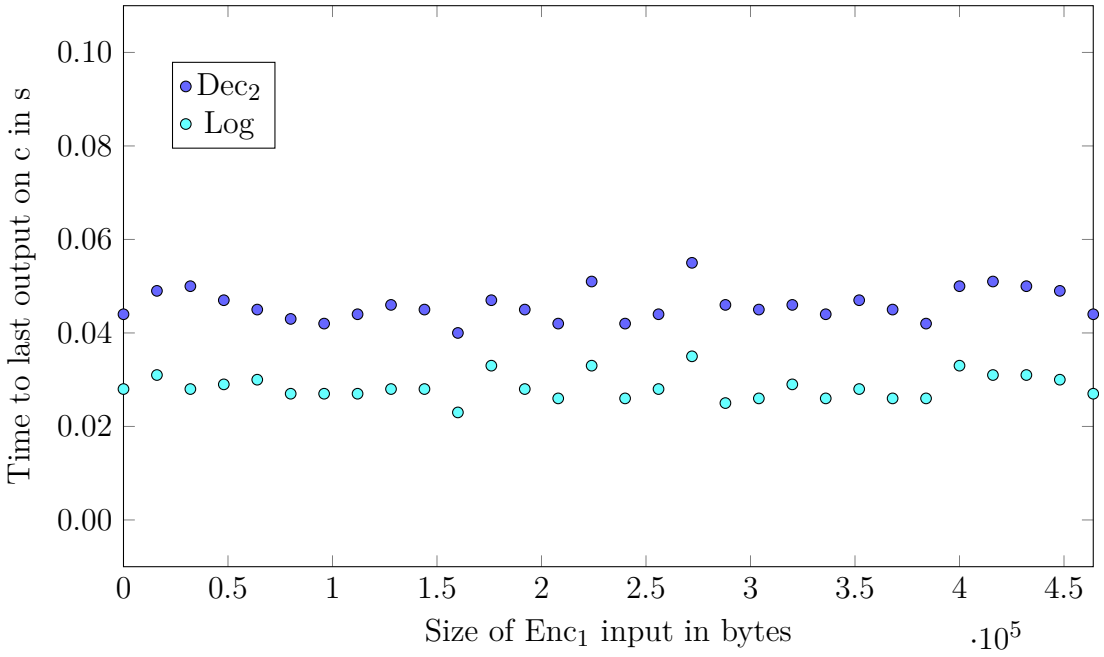


Figure 8.4: Timing-attack on optimized SME with timing-sensitive scheduling

from the Log level for the Key level is dequeued. Since this virtual execution is also empty and the termination flag is already set, it terminates immediately. The result are the queues at point V.

Meanwhile, the decryption level executions progress concurrently. If, as shown here, the decryption of the second ciphertext is completed faster, it also produces output for the Stat level first. However, since the Stat level is higher than the level of the Dec_2 execution, this output is redirected to the corresponding virtual execution. Thus, the output event is buffered after output from the Dec_1 execution, enforcing transparent ordering. When the Dec_2 execution is finished, as shown at point VI, the output remains buffered in the virtual execution on Stat level, which is flagged for termination.

Finally, the Dec_1 execution completes the decryption and produces output on Stat level. As its virtual execution is enqueued before the Dec_2 virtual execution, the redirection of this output ensures the correct order. The Dec_1 execution then reaches the boundary and terminates, dequeueing the Key-level virtual execution and flagging its virtual execution on Stat level for termination. When the Dec_1 virtual execution on Stat level terminates (after emitting the buffered output), the Dec_2 virtual execution is dequeued. The Dec_2 virtual execution then emits the buffered output in correct order and terminates, dequeueing the last virtual execution. When all executions are terminated, the run is complete.

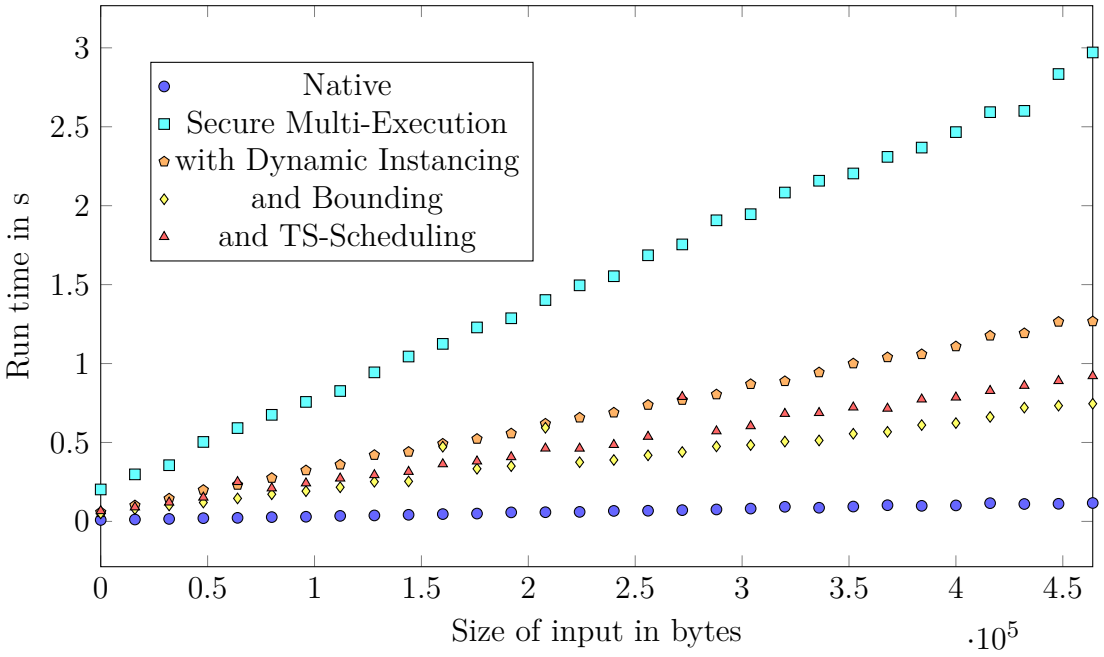


Figure 8.5: Efficiency impact of timing-sensitive scheduling

The resulting timing-sensitivity is further exemplified in Figure 8.4. As shown, no correlation between the size of the Enc_1 input and the Log and Dec_2 output timestamps exists. Thus, an authorized observer could not infer information about the first ciphertext under our enforcement. Successful PSNI-noninterference for the other leaks in demonstrated in Table 8.2. Here, our timing-sensitive scheduler achieves the same results as unoptimized Secure Multi-Execution enforcement. The loss of efficiency compared to barrier-style optimization in this scenario is relatively small, as evidenced in Figure 8.5. This shows that our timing-sensitive scheduling scheme for our demand-driven optimization leads to an confidentiality enforcement method for machine code that is secure, transparent, and efficient.

8.4 Summary

Our timing-sensitive scheduling finally allows us to use our Bounding Optimization without sacrificing the termination- and timing-sensitive security guarantees. At the same time, it also ensures per-channel and top-level transparency. We achieve this by allowing lower executions to progress independently of higher executions, while buffering their outputs. While this more complex strategy proves to be slightly less efficient than simple barrier-style synchronization, it is still a significant

improvement over unoptimized Secure Multi-Execution with all the same benefits.

In the next chapter, we demonstrate our vision towards fully automatic extraction of boundaries from target binaries. While we believe that boundaries can be provided by developers in many cases, advances in static information flow analysis of low-level code may eventually allow to automate this step, further advancing the practicality of our approach.

Chapter 9

Boundary Analysis

The key idea of our bounding optimization is to terminate an execution when the information stored therein will not be used again. Naturally, this requires knowledge about the future of a given state. Previously, we assumed that this information is provided by the developer. Yet, we also envision an automatic approach to extract this information, based on static binary analysis. In the special case of binaries hardened with control-flow integrity, we additionally provide a novel heuristic to resolve indirect call targets during control-flow reconstruction.

The core problem in static boundary analysis is that it requires precise knowledge of the information flows through the target program. Yet, data-flow analysis for machine code is much more complex than for source code targets. As described in Section 2.3.2, the abundance of multi-level arithmetic pointers thwarts efficient analysis in practice. Consequently, we were not able to obtain the necessary results with any existing tools, including the angr analysis platform [75], BAP [17] with Saluki [39] and Primus, and Bindead [55].

Thus, in the following we assume that such a solution exists and present our approach to extract boundaries from information flow analysis results. Based on our results, future work in the area could enable fully automatic boundary extraction, which further strengthens the applicability of our approach. It also highlights the need for a precise information flow analysis solution for binary code.

9.1 Boundary Extraction

Boundaries specify states where an execution has become redundant. Each of the multiplied executions represents a specific mix of information from various

channels. It becomes redundant when the information stored in the execution will not affect future outputs. When this can be guaranteed, then the execution can be terminated safely and the future output can be produced by another execution. In other words, the boundary for a specific mix of inputs is reached, when the obtained information is no longer live.

We propose to automatically extract the boundary from a compiled target in multiple steps. First, a model of the control-flow is needed to determine which outputs are reachable from which inputs. For this, existing solutions for control-flow reconstruction can be used (cf. [4]). In the special case of a target hardened with control-flow integrity, our heuristic to resolve indirect calls, as explained in the next section, can increase the precision of the result. Given the reconstructed control-flow graph (CFG), we then identify input and output requests. For this, we need to partially recover the arguments to system call instructions in the code, to determine the requested functionality. This can be achieved with techniques similar to those from Scherer et al. [70]. Once inputs and outputs are identified in the CFG of the target, a data-flow analysis is required to determine the information flows through the program. Through symbolic execution [75], value-set analysis [10], or future methods, data dependencies between parts of the code can be approximated. Together, these analysis provide a control-flow graph with input and output nodes and data-flow information.

In Figure 9.1, we show a simplified version of the control- and data-flow from our running example, introduced in Chapter 4. First, the key is obtained from the Key channel, creating a key object. This key object is checked when the library is called, assuming that more input is available. When the key object exists, the ciphertext is obtained from the encryption channel. Both the ciphertext and key information are subsequently used to produce the plaintext. The plaintext object is then written to the decryption channel and used again to emit statistical information on the Stat channel. Before returning, the library logs the length of the key object. When all inputs have been processed, the end marker is written to the log and the program exits.

To identify the boundary nodes, we analyze each input individually. For each node, we first find all outputs that use key information or information derived from key information. These nodes represent outputs that may be affected by information stored in the execution started at the input node. Therefore, this execution cannot be terminated as long as one of these nodes is reachable. We thus compute the backwards slice over the control flow, starting from these nodes, until the input node is reached. The resulting subgraph contains all nodes where the information obtained at the input may still be live and thus the corresponding execution should not be terminated. We call this subgraph *critial section*. Con-

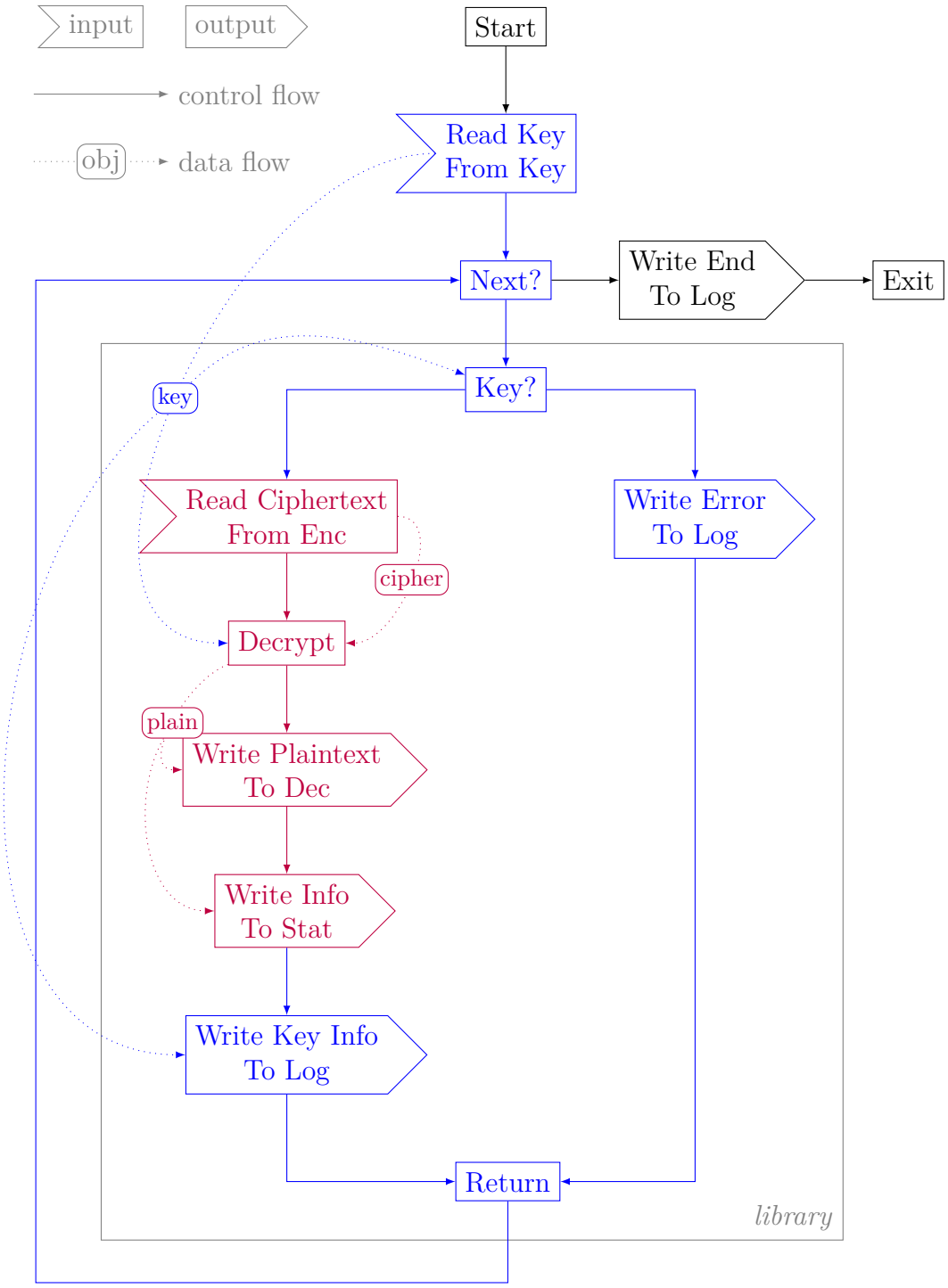


Figure 9.1: Control- and data-flow graph of the running example

versely, the first nodes outside the critical section describes its boundary. These nodes are the destination of an edge where the source node is in the critical section but the destination is not.

In our example, the key information directly affects one output in the library, namely the logging of key the key length. Yet, the outputs that use plaintext information are also affected by key information, since the plaintext is derived from it. These three outputs are reachable from the input through the data flows and thus must be part of the critical section regarding the key information. When computing the backwards slice from these nodes, we get the subgraph highlighted in blue in Figure 9.1. It starts with reading of the key information and includes the check for input and the complete library, encompassing the entire loop. The first node outside this critical section is the final logging node, shortly before exiting the program. Therefore, this node determines the boundary for the execution created when obtaining the key information.

The other input obtains the ciphertext. From the ciphertext, the plaintext is derived, which then affects the outputs in the library, namely the writing of the result to the decryption channel and the writing of statistical information. A backwards slice from these nodes gives the shorter purple section, highlighted in Figure 9.1. It only contains four nodes in the library, bounded by the logging of the key information. This shows that executions created when obtaining the ciphertext can be terminated after writing the statistical information.

In practice, some unsolved problems still persist that require additional work. First and foremost, a scaling, precise data-flow analysis that allows to identify the information flows through compiled code as stipulated here. Such a solution seems to be held back by the complexity of pointer analysis in low-level code. Second, the control-flow graph produced by current reconstruction methods is usually inter-procedural, with unclear function bounds [4]. Consequently, backward slicing on this graph may become imprecise due to missing context-sensitivity. We pursued a graph-based solution to this problem in our work [62]. Third, information created within loops may carry over to the next iteration, which would require to extend the critical section to include the complete loop. This problem could potentially be handled by unrolling the loops in the control-flow graph and add data flows for information from previous iterations. Finally, one problem that arises during control-flow reconstruction is posed by indirect calls. Sound identification of their targets usually requires complex data-flow analysis which often does not scale. Thus, we propose a fast heuristic for targets that are hardened with control-flow integrity protection in the next section.

9.2 Indirect Call Resolution

A prerequisite to any information flow or, in our case, boundary analysis is a sound control-flow graph (CFG) of the target binary. Yet, since machine code is unstructured and higher-level abstractions such as functions are usually not present, control-flow analysis is not as straightforward as it is on source code. As outlined in Chapter 3, algorithms exist to recover much of intended the control flow. However, some special cases require complex data-flow analysis to be resolved precisely. Precise resolution is achieved if the final CFG contains as little infeasible transitions as possible.

One of the major problems are indirect calls. Indirect calls transfer the control to the address specified by a register. For the static analysis, this means that the target of the control-flow transfer cannot be resolved without context. Instead, a precise resolution needs to determine *all feasible values* in the register for each indirect call. Consequently, the control flow in a binary is dependent on its data flow, and thus the control-flow analysis is dependent on data-flow analysis results.

Our solution to the problem is a simple heuristic that efficiently resolves indirect calls protected by control-flow integrity. Through control-flow integrity (CFI), the possible targets of an indirect call can be restricted to a reasonable subset of the functions in the binary. This is achieved by preceding the indirect call-site with a validity test that checks whether the value in the respective register is in a valid target set. Through these checks, the attack surface for code-reuse attacks is drastically reduced. As a side-effect, it allows us to quickly determine a sound and relatively precise approximation of the possible targets of an indirect call, without the need for expensive data-flow analysis.

While the idea to make use of CFI-information in the target is general, the actual resolution algorithm depends on the implementation. Control-flow integrity implementations vary with different compilers. Here, we explain our heuristic based on the CFI-implementation for binaries compiled with Clang and linked with the Gold linker [83]. Similar techniques should apply to Microsoft's Control Flow Guard [58, 81].

9.2.1 Resolution

Clang inserts two different CFI-checks, depending on the number of targets. In Figure 9.2 we show an example of the two types of instrumented calls. Figure 9.2a shows a call with multiple targets, Figure 9.2b shows a call with a single target. Both code blocks end with an indirect call instruction that uses a register to

```
405abe: 48 89 f8          mov    rax,rdi
405ac1: 48 b9 30 1a 41    movabs rcx,0x411a30
        00 00 00 00 00
405acb: 48 29 c8          sub    rax,rcx
405ace: 48 89 c1          mov    rcx,rax
405ad1: 48 c1 e9 03       shr    rcx,0x3
405ad5: 48 c1 e0 3d       shl    rax,0x3d
405ad9: 48 09 c1          or     rcx,rax
405adc: 48 83 f9 02       cmp    rcx,0x2
405ae0: 48 89 7c 24 08    mov    [rsp+0x8],rdi
405ae5: 76 02             jbe    405ae9
405ae7: 0f 0b             ud2
405ae9: 48 8b 44 24 08    mov    rax,[rsp+0x8]
405aee: ff d0             call   rax
```

(a) Instrumented call with multiple targets

```
40591b: 48 89 c1          mov    rcx,rax
40591e: 48 bf d0 19 41    movabs rdi,0x4119d0
        00 00 00 00 00
405928: 48 39 f9          cmp    rcx,rdi
40592b: 48 89 04 24       mov    [rsp],rax
40592f: 74 02             je     405933
405931: 0f 0b             ud2
405933: 48 8b 7c 24 08    mov    rdi,[rsp+0x8]
405938: 48 8b 04 24       mov    rax,[rsp]
40593c: ff d0             call   rax
```

(b) Instrumented call with single target

```
0000000000411a30 <close_stdout>:
 411a30: e9 0b fd fe ff jmp 401740 <close_stdout.cfi>

0000000000411a38 <__vm_wait>:
 411a38: e9 03 6b ff ff jmp 408540 <dummy.cfi>

0000000000411a40 <__stdio_exit>:
 411a40: e9 1b fe ff ff jmp 411860 <__stdio_exit.cfi>

00000000004119d0 <call>:
 4119d0: e9 db 40 ff ff jmp 405ab0 <call.cfi>
```

(c) Indirect call targets

Figure 9.2: Instrumented calls example

determine the target. The instrumented check allows the control flow to reach the indirect call only when a preceding conditional jump is taken. Otherwise, the control falls through to the `ud2` instruction, representing an undefined error state and halting the program. The conditional jump, on the other hand, depends on a preceding compare instruction, which checks the outcome of a preceding computation. This pattern is distinctive enough to be easily identified during control-flow recovery.

The instrumentation code implements a membership validation of the address pointed to by the register and a list of trampoline targets. The trampoline targets, shown in Figure 9.2c, redirect the control flow to the appropriate function start. These trampolines are added to ensure that the possible values for the register follow a known pattern. They are grouped by function signature and aligned to eight bytes distance. For example, the three functions `close_stdout, _ _vm_wait`, and `__stdio_exit` take no arguments and are therefore grouped into the range from `0x411a30` to `0x411a40`. Each indirect call targets addresses with matching function signature and thus from within a subset of the possible targets.

This can be seen at address `0x405ac1` in Figure 9.2a and address `0x40591e` in Figure 9.2b, where the address to the first elements in the group is loaded. We call address the *base* value of the indirect call. This information implies that the call in Figure 9.2a targets functions starting at `0x411a30`, while the single-target call in Figure 9.2a targets the function at `0x4119d0`. For a single-target call, resolution of the *base* value is sufficient to precisely determine the possible target. In the case of multiple targets, the *index* for the *target* in the group is determined by subtracting the *base* value and dividing it by the eight byte alignment. The call is valid when the *index* is smaller or equal to the number of elements in the group. Additional computation ensures that a *target* lower than the *base* leads to an infeasibly large *index*. The number of elements, denoted n, is defined by a constant argument to the comparison, for example shown at address `0x405adc` in Figure 9.2a. Thus, it can be easily recovered from the code pattern. Conclusively, the valid targets can be computed from these values in the following way.

$$targets = \{target \mid target = base + 8 * i, \text{where } i \in \mathbb{N} : 0 \leq i \leq n\}$$

Our resolution algorithm, implemented as a plug-in to the angr binary analysis framework in our `typhoon` toolchain, is shown in Algorithm 1. We use an additional filter function to identify the code pattern and extract the necessary values. As the *base* and number of elements are hard-coded, we can extract them without further data-flow analysis.

Algorithm 1 Indirect Call Resolver resolution

Require: Σ, CFG, check_node
Ensure: targets

```
targets := [ ]
insns := instructions(Σ, check_node)
compare := insns[-3]

if operand_types(compare)[1] = REGISTER then
  n := 0
  movabs := insns[-4]
else
  n := operands(compare)[1]
  movabs := insns[-9]
end if

base := operands(movabs)[1]
for all i ∈ [0, n] do
  targets = targets + [base + i * 8]
end for

return target
```

9.2.2 Summary

With our contributions to static analysis, we show how boundary descriptions could be extracted automatically from control- and data-flow information. From the control-flow graph, all paths between an input node and an output node can be established. With additional data-flow information, it can be validated whether information can flow between the input and an output node along one of these paths. All nodes visited by paths for which this is the case together form the critical section. The boundary is then determined by the first nodes outside this critical section.

Unfortunately, precise static control-flow and data-flow analysis of low-level code is still not feasible. Due to the abundance of multi-level arithmetic pointers and lack of scopes, points-to analysis quickly becomes an intractable, program-wide problem. Because low-level languages support dynamic branching instructions, whose control-flow target depends on the memory state, the problem carries over to control-flow analysis. For the latter, we contribute a heuristic to resolve indirect

calls protected by control-flow integrity techniques. Still, further investigation is needed to find a scalable and automatic solution, on which we could then extend our boundary analysis.

Part II

Evaluation

Overview

To show that our enforcement approach can be used in practice, we implemented a toolchain for the Linux operating system, running on the wide-spread x86_x64 architecture. Our `typhoon` toolchain, shown in Figure 9.3, consists of two parts. The `ddsme` package contains our monitoring system. It takes the target program, the security lattice, a mapping from paths to security levels, and a dummy input source as arguments. The monitoring system handles the invocation, duplication, and progress of executions, contained in individual processes. It also manages the connection to the operating system and enforces the transformations of the execution semantics. In case of an enforcement optimized with barrier- or queue-based bounding, the monitor additionally manages the barriers or queues. In this case, the monitor also takes the boundary information as input.

The boundary information can be provided manually or through our experimental static binary analysis, contained in the `critter` package. Our static analysis is based on the `angr` analysis framework for binary code [75]. We extend the framework with our heuristic to resolve indirect calls based on control-flow integrity information. Additionally, we implemented a fast identification of system call types, based on the results from our joint work with Scherer et al. [70]. From the recovered control-flow graph and identified system calls, we can then derive safe boundaries automatically.

We implemented the toolchain in the PYTHON language, to allow for rapid development of new features. Interaction with the operating system and access to process registers is realized through C-bindings with the `ptrace` library. Access to the process memory, setting of breakpoints, and mapping of file descriptors to paths is realized though the `proc` file system. Processes are traced using the `PTRACE_SYSCALL` command, which allows to step to the next system call or breakpoint. This reduces the number of interruptions of the execution as well as the number expensive context switches. Interception and manipulation of system call functionality is realized through hooks, which are invoked when the execution requests a corresponding system call.

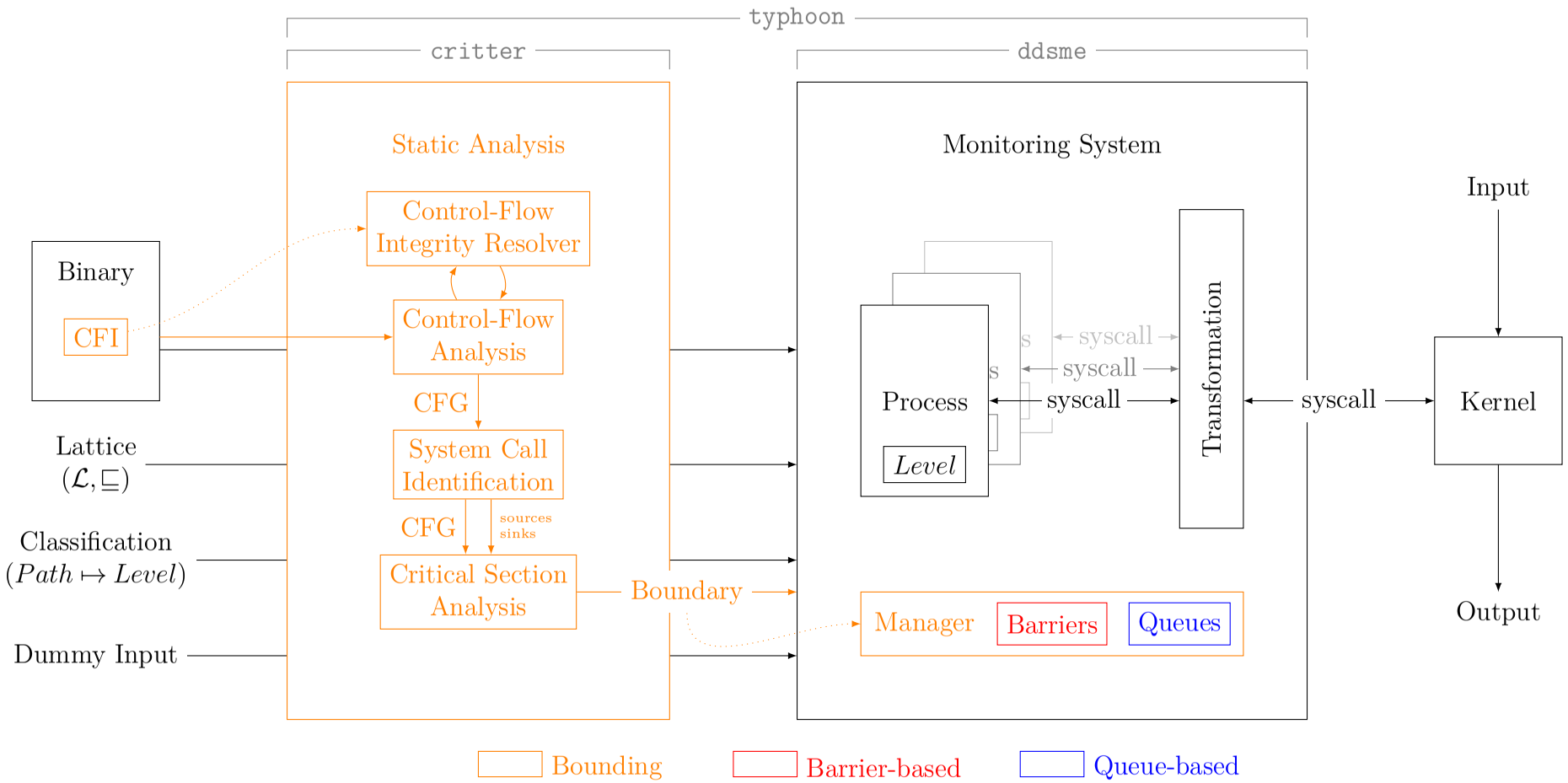


Figure 9.3: (Optimized) Secure Multi-Execution toolchain for machine code

We evaluate our implementation in three different settings. The first setting is the decryption service described in full detail in Chapter 4. For this example, we provided measurements and test results at the end of the previous chapters. The decryption service combines different information leaks in a single example, making it a very challenging scenario. To further demonstrate the versatility and generality of our approach, we provide results for a set of benchmark programs from related work and for assorted programs from the widely-used `coreutils` package next.

The benchmark programs represent a collection of different attacks that are often discussed in related work. Each program highlights a challenging aspect in terms of security or transparency. Our results, shown in Chapter 10, demonstrate that our toolchain achieves both for all examples. This highlights the effectiveness of our approach, guaranteeing timing-sensitive noninterference for all discussed attacks. We further show that we produce the same classified results that an unprotected execution of the target would produce. This shows that we achieve top-level transparency for all programs. Additionally, we show that we produce the same public results as an unprotected execution, as longs as the unprotected execution is timing-insensitively noninterferent. Thus, we also achieve per-channel transparency in all these cases.

We then demonstrate the practicality and optimized efficiency of our enforcement approach by applying it to real-world binaries. The programs `cat`, `sha*sum`, `wc`, and `sort` are widely-used, highly-optimized, and well-tested tools from the `coreutils` package that is part of many Linux distributions. We demonstrate that our toolchain can be applied to them and transparently enforce security in Chapter 11. On the one hand this shows that with our solution, we can handle real binary code, despite the technical challenges that this entails. Furthermore, it shows that our approach provides a general solution and does not have to be adjusted for different targets, with the exception of providing individual boundaries. Finally, we compare our Secure Multi-Execution enforcement solution with and without our optimizations in place. Here we can show that our optimizations can reduce the enforcement overhead significantly in many cases.

We summarize our contributions and results when we conclude this thesis in Chapter 12. There, we also provide pointers for future work, for example how to make our method more accessible to developers. We provide details on the benchmark programs in Appendix A, as well as more evaluation results for the `sha*sum` family of programs and the `sort` tool in Appendix B. In Appendix C, we demonstrate how our toolchain can be used to thwart input-delay attacks.

Chapter 10

Benchmark

With the benchmark tests, we show that our approach achieves security and transparency for a range of examples from related work. We used all 14 examples introduced by Bielova et al. to discuss transparency [16], five examples from the discussion about termination-sensitvity by Askarov et al. [6], four examples further addressing tremination-sensitivity from Ngo et al. [59], and one additional, unpublished example to highlight attacks via the timing side-channel[1]. Together, these 24 examples cover explicit and implicit flows, and leaks through the termination and timing-channels. They also contain secure programs that have been falsely flagged as insecure by other approaches in the past [16]. In contrast to this, we can show that our approach achieves per-channel transparency for these examples.

To apply our toolchain to the examples, we translated the examples to C code and compiled them into binaries. The examples provided in the related work are usually represented in a simple, interactive, WHILE-like language. While they do not always provide semantics for the language, most constructs are well-known and can be directly translated to C code. An exception are input and output commands that are not always clearly defined. Still, due to the simplicity of the examples, we assume the semantic gap between the original specification and our implementation is minimal.

Next, we provide details on the examples, our implementation, and configuration for the tests. We then provide evaluation results regarding progress-sensitive noninterference and finally demonstrate timing-sensitive noninterference enforcement with our toolchain. In the next chapter, we focus on practicality and effi-

[1] We were unable to find a suitable example for timing-attacks in the related work. The effects discussed in the timing-sensitive example from Kashyap et al. [42] are too small to be measured against the noise of a system under stress.

Source	ID	*Leak* E	I	T	Description
[6]	a01	✓			Counting loop with public output of the index until `h` is reached.
	a02		✓	✓	a01, but diverging after the loop.
	a03			✓	a02, but no implicit flow in loop limit.
	a04			✓	Divergence on specific `h`.
	a05	✓		✓	Direct leak of `h` then divergence.
[16]	b01		✓		Implicit flow on specific private value.
	b02			✓	Divergence on specific `h`.
	b03				b01, but `l` is sanitized.
	b04			✓	Divergence on combination of `l` and `h`.
	b05				Sanitized implicit flow.
	b06		✓		`l` implicitly changed.
	b07			✓	Divergence through implicit flow.
	b08				Always divergence.
	b09				Sanitized implicit flow.
	b10		✓		Implicit flow of `h` in one branch.
	b11		✓		Transitive implicit flow through `l2`.
	b12		✓		Implicit flow of `h`.
	b13		✓		Implicit flow on combination of `l` and `h`.
	b14		✓	✓	Implicit divergence or flow.
[59]	n01		✓		Public input dependent on `h`.
	n02			✓	Divergence on specific `h`.
	n03		✓		Implicit output, then divergence.
	n04				Sanitized implicit output.
-	u01	Timing-Attack			Output delayed through computation.

Leak types: E = explicit, I = implicit, T = termination

Table 10.1: Overview of benchmark examples

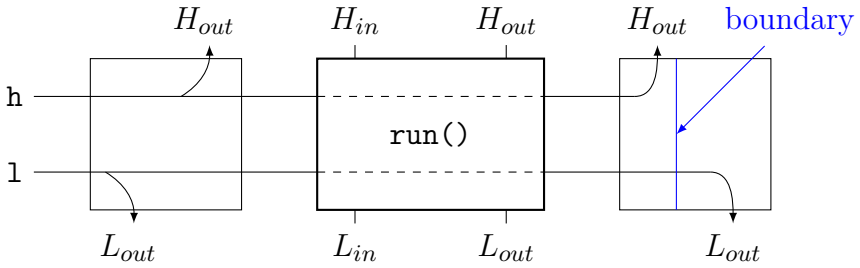


Figure 10.1: Information flows in the benchmark programs

ciency when applying our toolchain to real-world binaries.

10.1 Setup

An overview of the benchmark set and represented attacks is given in Table 10.1. Examples from Askarov et al. [6] and Ngo et al. [59] focus on leaks through termination (or divergence, resp.), while Bielova et al. [16] focus more on implicit leaks and transparency. Out timing attack is inspired by examples from Kashyap et al. [42]. We discuss is in more depth after discussing progress-sensitive noninterference.

To implement the examples, we translated them to C code. Details on each example are given in Appendix A. The examples implement a `run` function that is called from a common `main` function. A schematic of the composition and information flows is given in Figure 10.1. The main function declares two variables `l` and `h`. We use a two-level lattice and classify them as low and high, respectively. Their initial value is set to some unused value in the examples (e.g. 42). The values of both variables are also emitted on their respective channels before and after the example is run to detect interferences in the examples or intransparency. Shortly before the `l` value is emitted for the last time, the `h` variable becomes dead as the information stored therein is not used again. Thus, we place the boundary between the last outputs of the `h` and `l` variables. When the execution reaches this program location, the high execution will be signaled to merge or terminate, depending on the synchronization mechanism.

For the evaluation, we ran each test case with altering private inputs (i.e. $\{0, 1, 2, 7, 17, 400\}$) but constant public input. To test different execution branches, we reran the test cases with different public values (i.e. $\{0, 1, 2\}$). We provided an unrelated dummy value to not adversely affect the termination criteria (i.e. 7). Diverging executions were interrupted after a generous timeout. To assess the

timing-sensitivity of our enforcement, we additionally timed the outputs on the low channel. Table 10.2 shows the results of for all our test cases, noting different security and transparency criteria.

To measure progress-sensitive noninterference (PSNI), we compared the public outputs of test cases with equal low inputs. The criteria is violated when at least one test case produces differing public output. Timing-sensitive noninterference (TSNI) is less straight forward to asses, as there are natural differences in the timing of outputs. We attributed values between 0.000s and 0.010s to stress on the evaluation system. Thus, we consider a maximum difference in the timestamp of public outputs below 0.010s as secure. Note that TSNI requires PSNI, as otherwise public outputs are incomparable. Note also that while PSNI implies indirect termination-sensitivity, direct termination-sensitivity (TS) cannot generally be enforced, as proven by Ngo et al. [59]. We mark an example as TS when either all tests terminated or all tests diverged.

In terms of transparency of the enforcement, we validated two criteria, namely top-level transparency (TLT) and per-channel transparency (PCT). TLT is given when the enforcement produces the same high output content as the unprotected run. We aim to generally guarantee TLT both for secure and insecure programs, such that it should always be fulfilled. Additionally, we aim to guarantee PCT for secure programs, such that the low channel output contents are the same as for an unprotected run. In an insecure program, we may have to alter the low-channel outputs to achieve noninterference, thus PCT cannot always be established. Thus, we only evaluate PCT for programs that are originally PSNI-secure.

We explain the results of our timing-insensitive test cases next, and then focus on the timing-sensitive test case.

10.2 Results

Considering the results from Table 10.2, we can see that they reflect our expectations from Table 10.1. Except for programs `b03, b05, b08, b09, n04` and `u01`, all programs are violating progress-sensitive noninterference (PSNI). By implication, these also do not satisfy TSNI. For those that do, no measurable delays in public outputs could be measured. Some cases (`a04, b02, b04, b07, b14, n02`) are also not termination-sensitive. In these cases, high inputs interfered with the termination behavior.

In terms of security, our results show that our Secure Multi-Execution implementation (S) does indeed achieve PSNI for all examples. Furthermore, it also satisfies our timing-sensitive noninterference criteria (TSNI) and both the top-level

				Security									*Transparency*					
	Native			PSNI			TSNI			TS			TLT			PCT		
ID	PSNI	TSNI	TS	S	B	Q	S	B	Q	S	B	Q	S	B	Q	S	B	Q
a01	×	×	✓	✓	✓	✓	0.002	0.120	0.005	✓	✓	✓	✓	✓	✓	-	-	-
a02	×	×	✓	✓	✓	✓	0.003	0.002	0.003	✓	✓	✓	✓	✓	✓	-	-	-
a03	×	×	✓	✓	✓	✓	0.002	0.007	0.005	✓	✓	✓	✓	✓	✓	-	-	-
a04	×	×	×	✓	×	✓	0.005	×	0.003	×	×	×	✓	✓	✓	-	-	-
a05	×	×	✓	✓	✓	✓	0.001	0.005	0.002	✓	✓	✓	✓	✓	✓	-	-	-
b01	×	×	✓	✓	✓	✓	0.002	0.003	0.003	✓	✓	✓	✓	✓	✓	-	-	-
b02	×	×	×	✓	×	✓	0.001	×	0.005	×	×	×	✓	✓	✓	-	-	-
b03	✓	0.000	✓	✓	✓	✓	0.002	0.003	0.004	✓	✓	✓	✓	✓	✓	✓	✓	✓
b04	×	×	×	✓	×	✓	0.002	×	0.003	×	×	×	✓	✓	✓	-	-	-
b05	✓	0.000	✓	✓	✓	✓	0.003	0.003	0.006	✓	✓	✓	✓	✓	✓	✓	✓	✓
b06	×	×	✓	✓	✓	✓	0.001	0.003	0.005	✓	✓	✓	✓	✓	✓	-	-	-
b07	×	×	×	✓	×	✓	0.002	×	0.006	×	×	×	✓	✓	✓	-	-	-
b08	✓	0.000	✓	✓	✓	✓	0.001	0.001	0.001	✓	✓	✓	✓	✓	✓	✓	✓	✓
b09	✓	0.000	✓	✓	✓	✓	0.002	0.003	0.003	✓	✓	✓	✓	✓	✓	✓	✓	✓
b10	×	×	✓	✓	✓	✓	0.002	0.005	0.005	✓	✓	✓	✓	✓	✓	-	-	-
b11	×	×	✓	✓	✓	✓	0.001	0.004	0.006	✓	✓	✓	✓	✓	✓	-	-	-
b12	×	×	✓	✓	✓	✓	0.002	0.005	0.006	✓	✓	✓	✓	✓	✓	-	-	-
b13	×	×	✓	✓	✓	✓	0.002	0.002	0.003	✓	✓	✓	✓	✓	✓	-	-	-
b14	×	×	×	✓	×	✓	0.002	×	0.004	×	×	×	✓	✓	✓	-	-	-
n01	×	×	✓	✓	✓	✓	0.002	0.004	0.005	✓	✓	✓	✓	✓	✓	-	-	-
n02	×	×	×	✓	✓	✓	0.001	0.001	0.001	✓	✓	✓	✓	✓	✓	-	-	-
n03	×	×	✓	✓	✓	✓	0.001	0.004	0.005	✓	✓	✓	✓	✓	✓	-	-	-
n04	✓	0.000	✓	✓	✓	✓	0.001	0.004	0.006	✓	✓	✓	✓	✓	✓	✓	✓	✓
u01	✓	0.011	✓	✓	✓	✓	0.001	0.015	0.002	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 10.2: Evaluation results

```
1  #include "../common.h"

2  void run() {
3      // for i = 0 to secret {keep busy}; output 1
4      h = input(Hin);
5      for (int i = 0; i < h; i ++) {
6          for (int j = 0; j < 3000; j++) {
7              int a = (i * h) / (i ^ h);
8          }
9      }
10     output(Lout, 1);
11 }
```

Listing 10.1: Benchmark program u01.c

transparency (TLT) and per-channel transparency (PCT). This shows that our implementation translates the theoretical guarantees to practice. As expected, our implementation cannot generally guarantee direct termination-sensitivity (TS). Here, `n02` poses an exception, as our dummy value always leads to divergence in this case.

The results for our optimized Secure Multi-Execution with barrier-synchronization (B) show the expected outcome. Because the boundary is set before the final public output, divergence of the high execution led to a suppression of this value in the termination-insensitive cases. Consequently, our barrier-synchronization fails to guarantee PSNI (and therefore TSNI) for some cases. Additionally, in case `a01`, the output to the public channel is delayed while the high execution is executing the loop. Thus, in this case, it fails TSNI while providing PSNI guarantees. Top-level and per-channel transparency, on the other hand, is guaranteed as expected.

In contrast to this, our timing-sensitive scheduler based on queues (Q) does achieve the same guarantees as unoptimized Secure Multi-Execution. Because our scheduler breaks dependencies between lower and higher executions, divergences of higher executions do not affect lower executions. Consequently, PSNI is achieved even for the termination-insensitive cases. For the same reason, it furthermore achieves TSNI in all cases. Due to the reordering of output, is also achieves top-level and per-channel transparency. As a result, our optimized Secure Multi-Execution with timing-sensitive scheduling achieves the same strong security and transparency guarantees as the unoptimized implementation.

Finally, we consider the timing-attack from `u01` in more detail. In Listing 10.1, we show the code of the `run` function for this example. The attack is inspired by the

examples from Kashyap et al. [42]. They use an implicit flow to introduce a delay into the public output that is dependent on private information. We replicate this scenario here, by performing an time-consuming (and nonsensical) computation based on private information. Thus, the greater the value stored in `h`, the longer the delay.

This is reflected in our results, which show that the program is PSNI, but not TSNI without enforcement. Considering the enforcement results, our unoptimized Secure Multi-Execution once again does achieve TSNI for this example. Because the computation is performed based on consistent dummy input in the public execution, different private information has no effect on the timestamp of the public output. Thus, the maximum difference in timing is 0.001, representing noise from the system. In contrast to this, the barrier-synchronized optimization again fails to guarantee TSNI, as it waits for the high execution to finish the computation before progressing to the final public output. Yet, our queue-based termination-sensitive scheduler alleviates the problem and successfully enforces timing-sensitive noninterference for this test case as well.

10.3 Summary

Our benchmark test cases show that the results provided in previous chapters are not coincidental. Even for 24 different examples from related work, highlighting different kinds of leaks and challenges for transparency, our enforcement could consistently guarantee security and transparency. While our barrier-synchronized optimization fails to enforce termination- and timing-sensitivity as expected, our queue-based timing-sensitive scheduler does indeed provide the same guarantees as our unoptimized Secure Multi-Execution implementation.

Consequently, progress- and timing-sensitive noninterference could be guaranteed for all examples, including termination-insensitive cases and an attack on the timing-channel. At the same time, top-level transparency could be guaranteed for all examples as well, while per-channel transparency could be guaranteed for all originally secure examples. This shows that our approach provides strong security without unintentionally breaking the functionality of the target.

Next, we show that our approach can also be applied to more complex real-world examples from the `coreutils` collection. Here, we also show the impact of our optimizations on the enforcement efficiency in two different settings. The results demonstrate the practicality of our approach, as well as the improved efficiency. For a discussion of input-delay attacks and our countermeasure, we refer the reader to Appendix C.

Chapter 11

Coreutils

Besides the benchmark programs that demonstrate the effectiveness of our approach against various attacks, we also validated our approach on several programs from the popular `coreutils` collection. This demonstrates that existing code that was neither programmed with confidentiality in mind, nor crafted to fit our solution, still benefits from our optimization. Additionally, although the use-cases of `coreutils` programs are usually quite simple, their code is highly optimized, which leads to complex binaries. Yet, our approach can be applied to all supported examples with no adjustments except for binary-specific boundaries. Finally, we use the examples to also highlight the efficiency improvements that we achieve through our optimizations for Secure Multi-Execution. Note that we are only interested in the relative improvement of efficiency when compare to our unoptimized Secure Multi-Execution implementation. Construction of a more efficient implementation, perhaps directly integrated in the kernel, is left for future work.

We evaluate our approach on four different targets from the `coreutils` suite, namely `wc` (word count), `sort`, `cat`, and the `sha*sum` family. We chose these targets as they take multiple files as input, which allows us to perform tests with multiple input channels. They also use a limited set of file-handling system calls that we support in our prototype. To support more examples, handling additional system calls would have to be implemented. From our experience, we expect this to range from relatively simple, for which `lseek` is an example, to very intricate, for example for the multi-purpose system call `ioctl`. Ultimately, support of all system calls in our prototype would come close to mirroring the kernel in our monitor running in user space. Therefore, we suggest for the future to integrate our approach with the kernel instead.

Considering our bounding optimization, the targets differ in two important as-

(a) LABHT-Lattice (b) Total-order lattice

Figure 11.1: Lattices used during evaluation

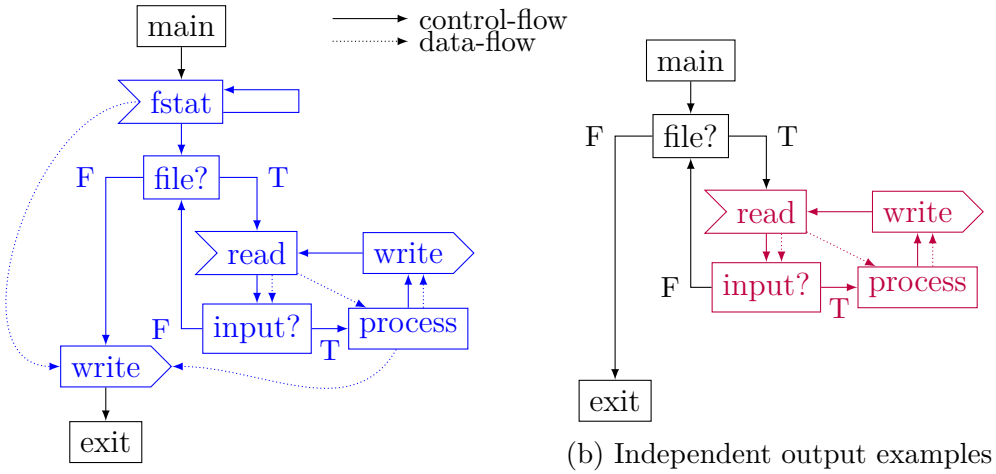
pects. In `wc` and `sort`, input is obtained very early and remains live throughout most of the execution. Thus, it represents the worst case for our optimizations. Still, we show that we can transparently enforce security and benefit from dynamic instancing in cases where input is obtained on a subset of the possible channels. Conversely, `cat` and the `sha*sum` programs obtain and process new input sequentially. Here, information from one file is dead after the file has been processed. New information is also only accessed when the previous processing is completed. Consequently, these targets lend themselves well to our bounding optimization, leading to a significant reduction of the enforcement overhead.

We discuss this difference in more detail next. Then, we give an overview of the test setup, with a focus on the used lattices. Subsequently, we first show how our dynamic instancing can lead to a reduction of the enforcement overhead in worst-case programs such as `wc`. Then, we demonstrate the effect of our optimizations including bounding on better suited targets such as `cat`. To keep this section brief, and because they show the same effects, we moved the addtional results for the `sort` and `sha*sum` programs to Appendix B.

11.1 Setup

For the evaluation of the `coreutils` targets, we feed the targets test files with random content of controlled length. In each run, we first produce the expected output using an unprotected execution. This gives us a baseline for subsequent tests to ensure that our results are secure and transparent. Each mark in the following plots represents a successful test case. We additionally measure the time before and after execution of a test case to determine the real run time.

During our evaluation, we use two different lattices. To demonstrate that we



(a) Dependent output examples

(b) Independent output examples

Figure 11.2: Schematic CFGs with data-flow and boundaries for the two categories

can handle incomparable levels, we use the LABHT-lattice shown in Figure 11.1a. During our tests, we compare the outputs on the **H**-level, as correct output on this level demonstrates both security (omission of $\top$-level information) and per-channel transparency for the complex case with incomparable levels. Thus, to create the baseline, we execute the program normally but replace the $\top$ information with the dummy input. Then, we run the protected execution with the actual $\top$ information, using the same dummy source. Consequently, when the two outputs are exactly the same, we show that our protection a) removes the sensitive $\top$ information from the output, while b) retaining the original order and content at H-level. Therefore, each successful test shows both security and transparency of our enforcement. To eliminate differences due to resource contention between incomparable levels during tests of scalability with the lattice size, timing, and file splitting, we use the total-order lattice shown in Figure 11.1b in these settings.

In our tests, we also use different boundaries, as illustrated in Figure 11.2. In the case of `wc` and `sort`, shown in Figure 11.2a, most of the program is part of the critical section. Input for all channels is obtained right after starting the program when information about the input files is checked using `fstat`. This information is used in the final output and thus remains live right until the program exits. While we can provide a safe boundary here, it does not lead to increased performance as it merely replaces the normal exit with our termination procedure in the executions. Thus, we use only our dynamic instancing optimization in these programs.

In contrast to this, `cat` and programs in the `sha*sum` family process each file sequentially. Thus, information is obtained very late and becomes dead when

```
118  520 3811 /home/pfeffer/.bashrc
 45   76 2643 /etc/passwd
163  596 6454 total
```

(a) Unprotected output

```
118  520 3811 /home/pfeffer/.bashrc
  0    0    0 /etc/passwd
118  520 3811 total
```

(b) Protected output

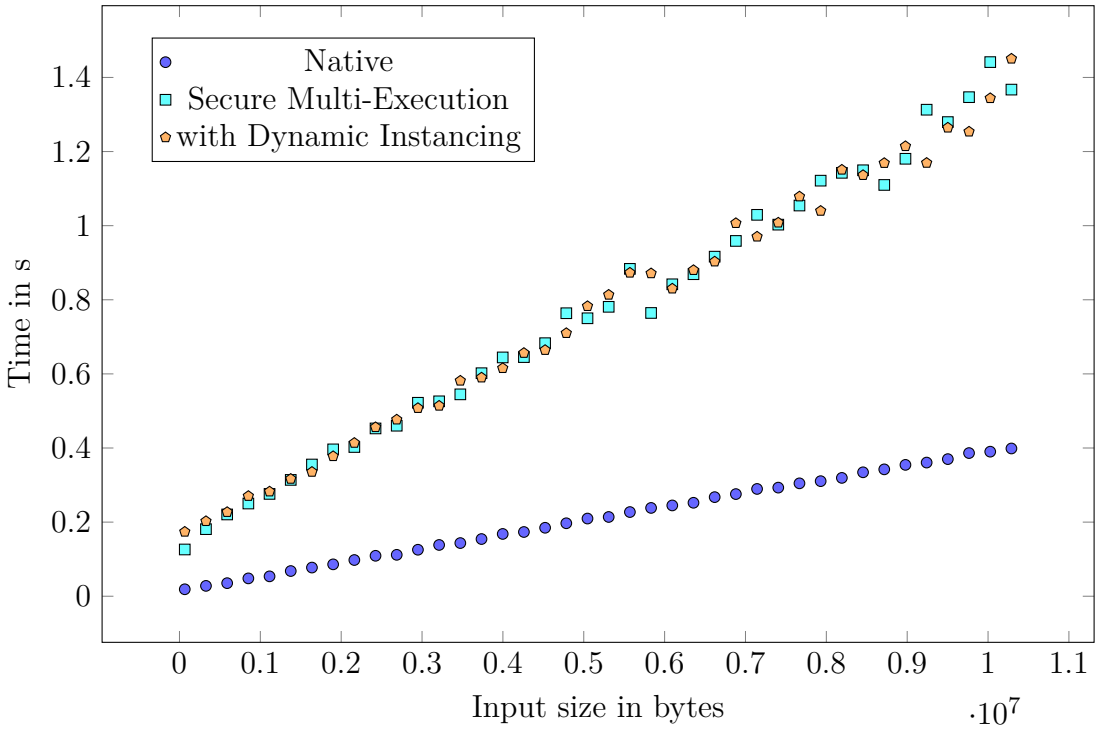
Figure 11.3: Effect of confidentiality enforcement on `wc`

the inner loop is left. Consequently, we can use our bounding optimization to terminate newly created executions after they handled each individual file. As a result, a maximum of two executions are running at any time, proposing a great increase in enforcement efficiency.

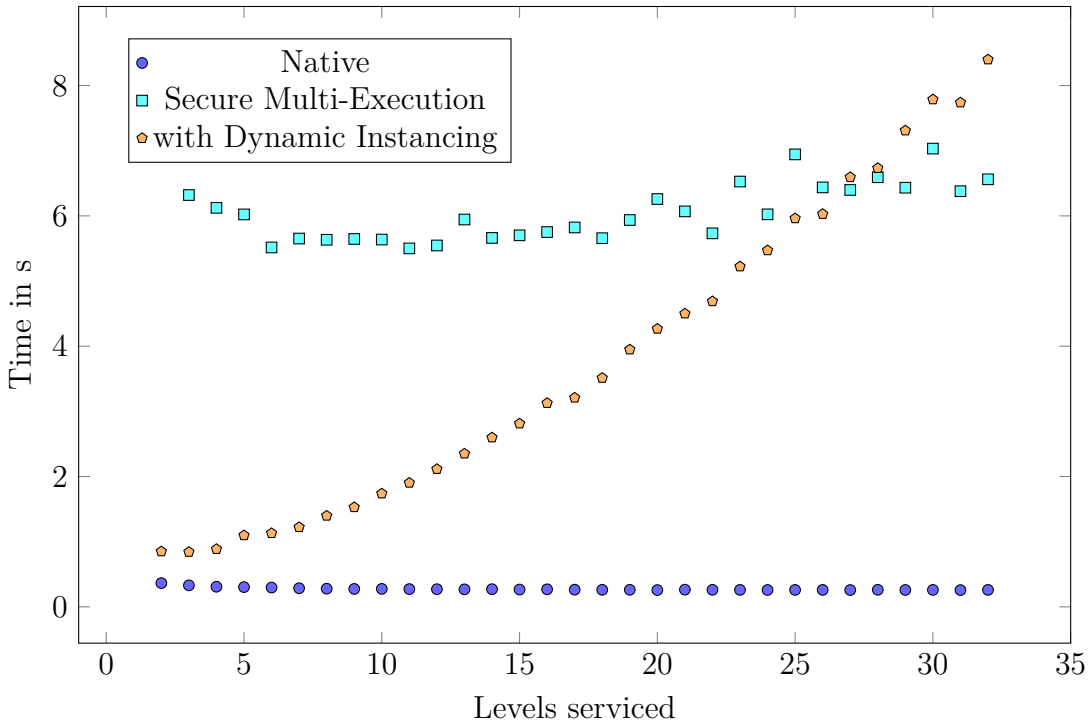
11.2 Results for Word Count

As the name suggests, word count (`wc`) counts the words in all provided files and prints the result. The results are listed per file as well as overall. In Figure 11.3 we illustrate the effect of our confidentiality enforcement on the word count program. Above, we show the output produced by an unprotected run using `/home/pfeffer/.bashrc` and `/etc/passwd` as input. Assuming that no information about `/etc/passwd` should be leaked, we show the output produced by a protected run below. As can be seen, the revealing counters for `/etc/passwd` have been replaced with benign counters for empty dummy input. The total counters reflect this change correctly. Except for the definition of the boundaries, no static analysis, binary rewriting, nor knowledge of the target was necessary to achieve this effect.

We do the same kind of experiment in Figure 11.4a, yet with five input files and the more complex LABHT-lattice. Each mark in the plot demonstrates a case where the secret top-level information is removed, while all other counters are unaltered. The plot shows how the run-time increases with larger input sizes for both unprotected execution (native) and protected execution of two kinds (unoptimized and optimized). As expected, the Secure Multi-Execution of the target induces a significant overhead. Unfortunately, our dynamic instancing also does not help in



(a) Overhead sampling for different input sizes



(b) Overhead sampling for different number of input levels

Figure 11.4: Evaluation results for `wc`

this case, as input information is obtained very early in the execution. Thus, we very quickly must create executions for all levels, leading to the same overhead as unoptimized Secure Multi-Execution.

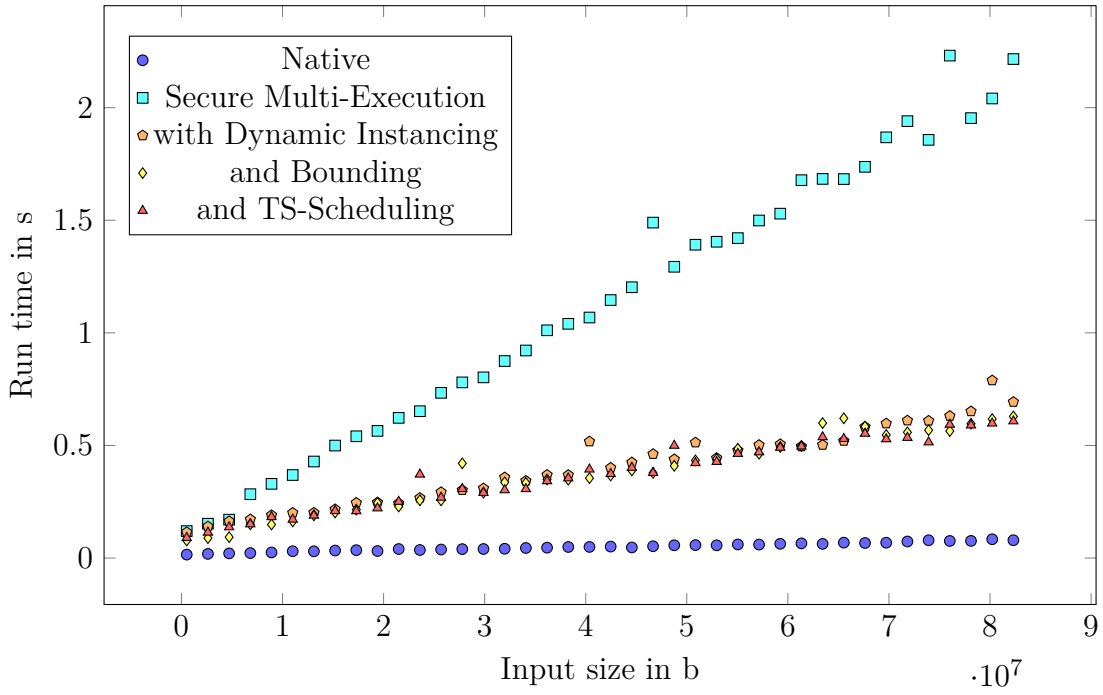
Yet, our optimization can still improve the performance in some cases. Figure 11.4b shows the run times when the amount of input remains the same but is divided between increasingly many security levels. Here, we use a total lattice with 32 levels. Because unoptimized Secure Multi-Execution cannot adapt dynamically to the actually used levels, it must always pessimistically create all 32 executions to service all levels in case they are needed. Naturally, this leads to a high overhead in all cases. The overhead remains relatively stable here, as the total amount of input is the same. In contrast to this, since our dynamic instancing does allow to adapt to the actually used levels at run time Consequently, we achieve a significant efficiency improvement in may cases. At half the levels actually in use, we only need roughly half the time to process all inputs. When nearly all levels are actually serviced, our optimization effect diminishes. This shows that even in worst-case targets such as `wc`, our optimization can lead to a significant increase in performance, depending on how the program is used.

11.3 Results for Cat

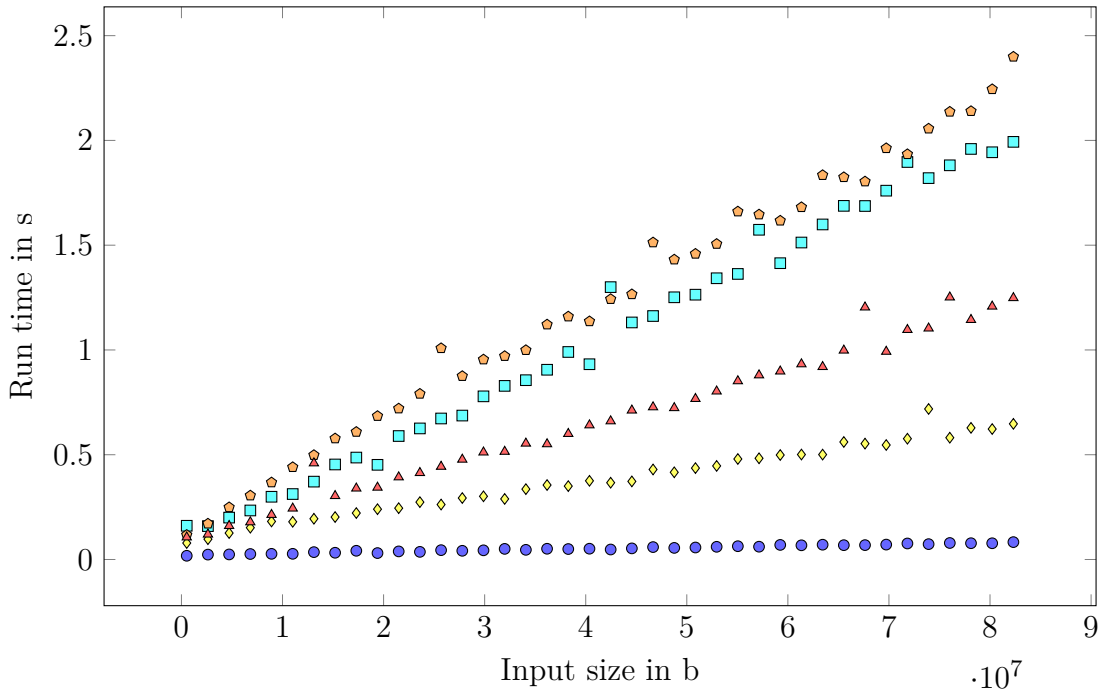
The `cat` tool follows the schema shown in 11.2b. The purpose of the program is to concatenate multiple input files into the output file. Thus, the contents of each file are emitted without change. Because inputs are processed independently, we can terminate the corresponding execution after each file. This leads to a great reduction of the performance overhead. As shown in Figure 11.5, our optimizations reduce the overhead to up to one third compared to unoptimized protection.

Interestingly, in Figure 11.5a, our bounding optimization does not increase the performance beyond our dynamic instancing optimization. This effect is a result of the order in which input is provided to the program. Here, we provide the input in ascending total order of the lattice, meaning L, A, B, H and then T-level. Each new input thus leads to the creation of a new execution, while the existing, *lower* executions obtain empty dummy information instead. Therefore, only the new execution handles the new input, while the other executions skip the processing due to empty input. The effect is so strong that dynamic instancing alone leads to the full improvement.

When the input is provided in reverse order, the situation is different. As shown in Figure 11.5b, here dynamic instancing does not lead to an increase in efficiency. Because higher executions are created first and then have access to

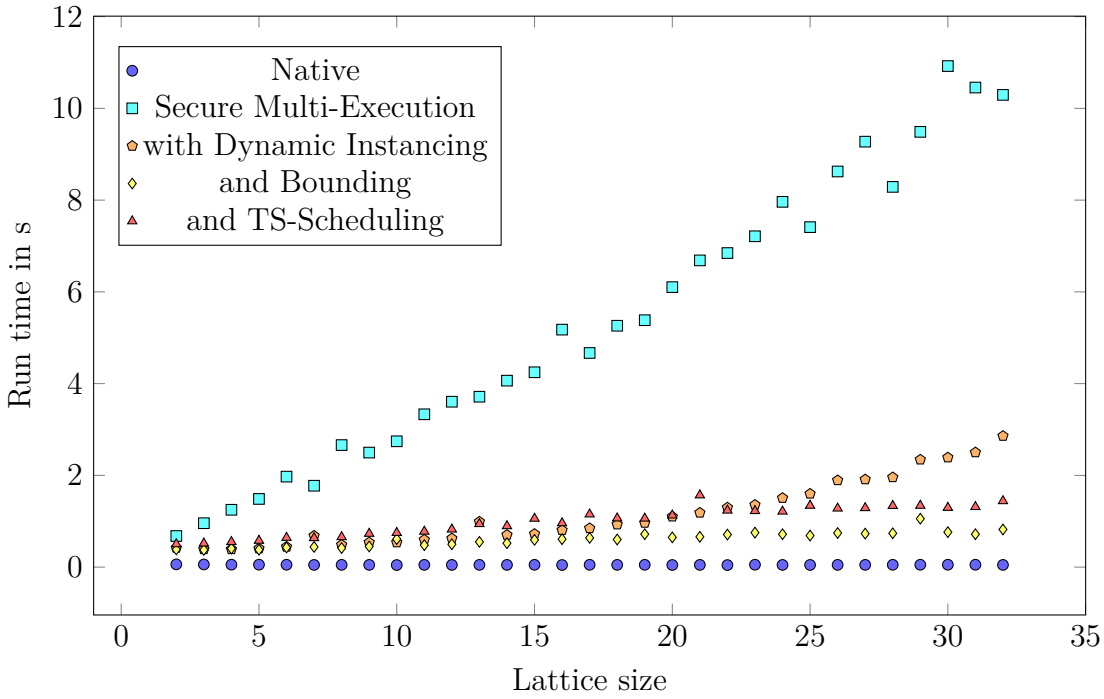


(a) LABHT-order

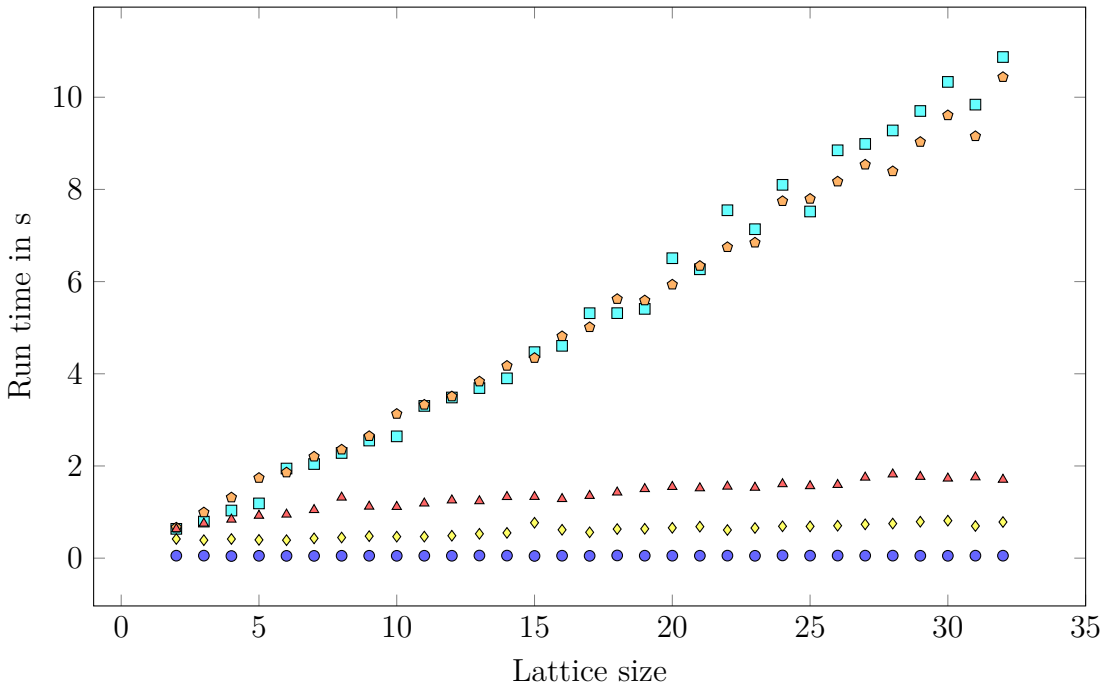


(b) THBAL-order

Figure 11.5: Run-time sampling of `cat`



(a) Ascending total order



(b) Descending total order

Figure 11.6: Level sampling of `cat`

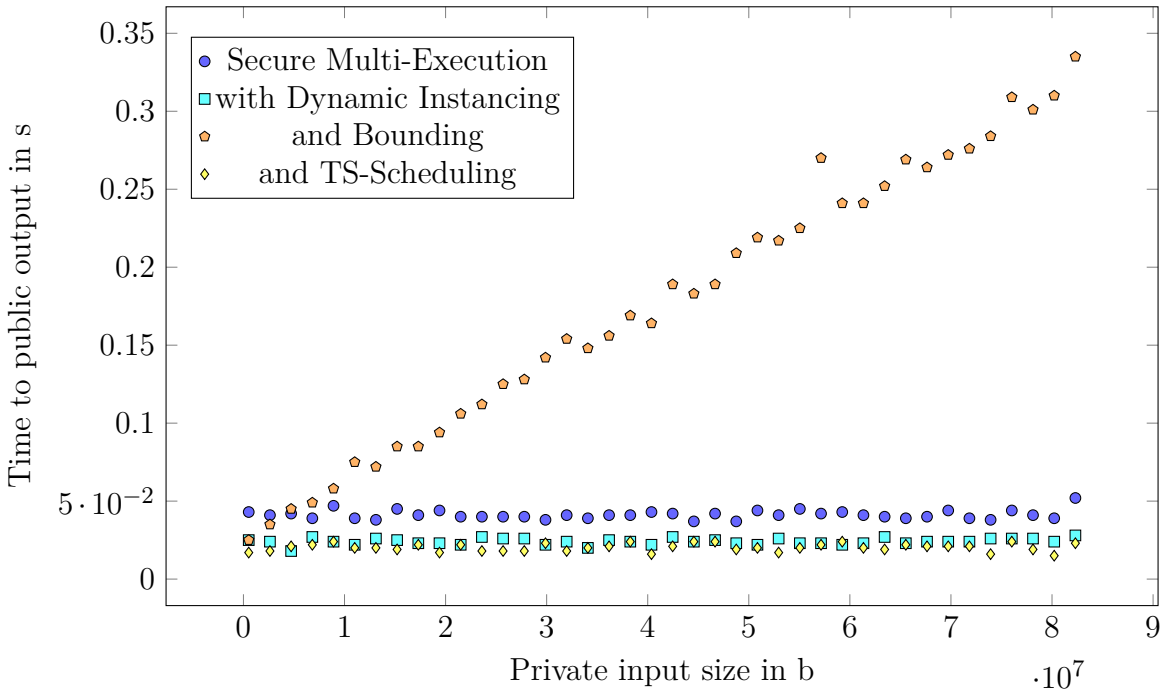


Figure 11.7: Time sampling of the first output event

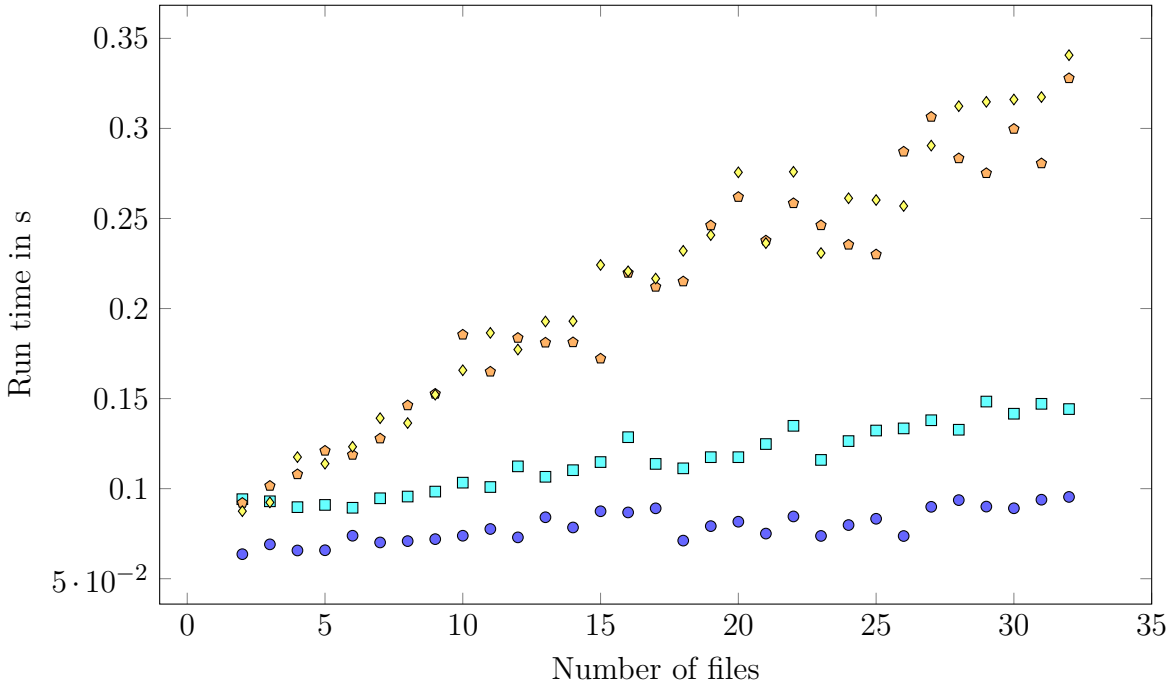


Figure 11.8: File split sampling of `cat`

subsequent lower input, all executions obtain all input. Consequently, redundant computations occur when only dynamic instancing is used. Yet, as the graph shows, through our bounding optimization, we can achieve the same speedups as for the best-case order of inputs. With executions terminated early, we again achieve a situation where only one execution obtains the actual input, saving all redundant computations. Finally, because higher channels are served first under this order, and thus are more busy, our timing-sensitive scheduling becomes more complex as well. Thus, as expected, it performs slightly worse than our simpler barrier-style synchronization.

The significant performance increase persists when the complexity of the lattice is increased. In Figure 11.6, we measured the run times of different protection configurations for a total-order lattice with increasingly many levels. Unlike the measurement performed for word count, where we pessimistically assumed that the number of actually needed levels is not know before execution, here we optimistically assume that it is. Consequently, unoptimized Secure Multi-Execution performs better for smaller as they require less executions. Still, our optimizations significantly improve the performance and allow the enforcement to scale with the lattice complexity. Only when the input order is detrimental to our dynamic instancing and no bounding is used, will our optimizations achieve the same overhead as unoptimized Secure Multi-Execution. In all other cases, our optimizations achieve the desired effect.

Our tests show that bounding greatly decreases the enforcement overhead. Yet, they also show that a simple barrier-style synchronization performs slightly better than our timing-sensitive, queue-based scheduling. However, as the name suggests, our timing-sensitive scheduling also protected against leaks through the timing-channel. This is shown in Figure 11.7, where we measure the elapsed time until the public output is finally emitted. As can be seen, all protection configurations correctly remove any correlation between sensitive input size and the public output timestamp, except for bounding with barrier-style synchronization.

Finally, an interesting effect shows in Figure 11.6a for a larger number of levels in the lattice. Because the same amount of input is split up between an increasing number of files, the size of each file decreases with each additional level. Consequently, the time spent in the critical section reduces for each execution, which amplifies the overhead for the creation of a new execution. Thus, for a complex lattice with small file sizes, the overhead of dynamic instancing starts to rise. To isolate this effect, we performed another test with a small amount of input that is split across an increasing number of files. The results are shown in Figure 11.8. Here we can see that for increasingly many files, and thus increasingly smaller critical sections, the overhead of our more complex optimizations starts to rise. This

points to interesting question regarding the optimal size of the critical section. It may be beneficial to move the boundaries out of small critical loops, to prevent a repeated creation and termination of executions.

All in all, our results for `cat` show the promising effects of our optimizations. Not only do they reduce the enforcement overhead for executions that spent an increasing amount of time inside the critical section. They also allow the enforcement to scale with the complexity of the lattice. Our timing-sensitive scheduling further ensures that these benefits do not come at the cost of the strong security guarantees of Secure Multi-Execution.

11.4 Summary

Beyond `wc` and `cat`, we also performed the same tests again for `sort` and program from the `sha*sum` family. The results are shown in Appendix B and generally support our findings for `wc` and `cat`. Like `wc`, `sort` adheres to the unfortunate flows shown in Figure 11.2a and thus lends itself badly to our optimizations. Still, our dynamic instancing can again be used to increase the performance compared to unoptimized Secure Multi-Execution when only a subset of levels is serviced. The programs in the `sha*sum` family, on the other hand, follow the construction of `cat`. Unlike `cat`, the `sha*sum` programs do process the input to compute the SHA hash of the files, leading to longer run times overall. Yet, again our optimizations can significantly reduce the enforcement overhead, allow the enforcement to scale with larger lattices, and ensure timing-sensitivity. Consequently, our finding show that Secure Multi-Execution can be made efficient for real-world programs in complied form.

Chapter 12

Conclusion

In this thesis, we addressed the problem of information leaks from library code included in new software. Including third-party library code is often necessary to reduce the development costs and time to market of the software product. Yet, library code is usually shipped in compiled form, which is notoriously hard to analyze. Thus, a method is needed to enforce confidentiality across all components of a software product, including compiled third-party library code.

Our goal in this thesis is to design and develop a mechanism to enforce confidentiality of compiled software components. We propose a solution that is secure, transparent, efficient, and practical. Security in this context means that no information about sensitive inputs can be inferred from observing public outputs of the enforced program. This means that no information leaks neither through data flow, control flow, termination behavior, nor timing-behavior. Conversely, transparency means that all secure behavior remains intact. This ensures that the enforcement mechanism does not break the functionality of the target, which would make the solution impractical. Lastly, the enforcement mechanism should be practically applicable to machine code. This implies that the enforcement is efficient, meaning any imposed performance overhead scales with the complexities of the target and application scenario.

Our solution is based on the concept of Secure Multi-Execution as proposed by Devriese and Piessens in 2010 [32]. Under Secure Multi-Execution, the target is executed multiple times. Each execution is responsible for outputs on a certain security level and only has access to information with equal or lower classification. This way, security is ensured by design. The individual executions can be scheduled independently, achieving protection against termination- and timing-attacks as well. Additionally, Secure Multi-Execution has been shown to achieve high transparency [16], making it a good fit for our requirements.

Yet, until now, no practical application of Secure Multi-Execution to machine code existed. Thus, it was unclear whether the technique can be applied, how it can be applied, and how well it performs. Furthermore, the practicality of Secure Multi-Execution is severely limited by its inefficiency. As a consequence of the multi-execution, the resource requirements of the target application are also multiplied. In many cases, this leads to an unacceptable overhead, which grows exponentially with the granularity of the security levels.

We solve these problems with five contributions. First, we are the first to apply Secure Multi-Execution to machine code, demonstrating the effectiveness of the approach in our scenario. To achieve this, we provide concrete implementations for the various abstract concepts that are used to formally define Secure Multi-Execution. The resulting system successfully enforces a termination- and timing-sensitive notion of confidentiality across all components of a pre-compiled target. At the same time, it produces correct outputs for secure behaviors, demonstrating high transparency of the approach.

To improve the enforcement efficiency, we propose two novel optimizations. The key idea is to remove as much as possible of the redundant computation that was introduced through the multiplication of the execution. In our *Dyanmic Instancing* optimization, we aim to create as few executions as possible and do so as late as possible. This means that we effectively reduce redundant computation from executions in equivalent states. The key idea is that in a deterministic system all executions behave the same, as longs as no secret information has been obtained. Our second optimization, called *bounding*, aims to terminate as many executions as early as possible. This is enabled by reusing static analysis results that provide information about the future behavior of executions. Whenever an execution can be replaced by another execution without loss of observable outputs, we terminate one executions and emulate its behavior with another execution. This way, we can save redundant computations after multiple executions have converged in their behavior without impairing the transparency of our enforcement.

While our Bounding Optimization provides great reduction of the enforcement overhead, it impacts the termination- and timing-sensitive security guarantees of Secure Multi-Execution, and requires a-priori knowledge about the target. Regarding the first problems, we introduce a timing-sensitive scheduling that breaks revealing dependencies between executions, while reordering outputs to preserve transparency. The necessary a-priori knowledge could be provided by the application developer, yet we also worked towards an automatic solution. Here, we show how it could be extracted from information flow analysis results and provide a heuristic to aid in the control-flow recovery process.

Finally, we integrated our contributions into a single toolchain for the enforce-

ment of confidentiality of machine code. We applied this toolchain to a range of benchmark examples, collected from related work, as well as some assorted binaries from the `coreutils` suite. Our benchmark validation shows that our enforcement mechanism protects against leaks through data flow, control flow, termination, and timing. At the same time, it provides per-channel and top-level transparency. Furthermore, our validation of programs from the `coreutils` package shows that our enforcement extends to real-world binaries. Our Dyanmic Instancing optimization significantly increases the efficiency in many cases, effectively halving it in our running example. Where applicable, our Bounding Optimization further significantly increased the efficiency in the remaining cases and allows our approach to scale with increasing lattice sizes.

While our work provides a practical solution to confidentiality enforcement for low-level components, it also raises new, promising questions. Most pressing is the need for a precise and scalable information flow analysis for binary code. The problem seems to be that dynamic memory addresses require knowledge of the possible execution contexts, which thwarts compositional approaches. Yet, Meng et al. started to work in that direction with efforts to parallelize their DynInst analysis framework [53]. Separation logic is another approach to deal with compositional analysis in the presence of heaps. Reynolds worked towards a low-level application, but the work remains unfinished [66].

A different path could be taken altogether, when focusing on developer-provided knowledge. Descriptions of critical sections could be included into higher-level languages, allowing developers to bind multi-execution enforcement to object lifetimes. Then, executions created together with objects could be terminated by garbage collection, also leasing to an automatic description of boundaries. Such implementation work could go hand in hand with a kernel-level implementation of our approach, which promises to greatly reduce the monitoring overhead. Recent work by Koning et al. on multi-variant execution shows that a switch from `ptrace` to `dune` [15] as the interfacing library could greatly increase the performance [45]. More targets could also be supported with the integration of network communication predicates, on which we are currently working.

Finally, an area closely connected and mutually beneficial is that of testing. On the one hand, our work requires the definition of dummy inputs that do not crash the target. Here, fuzzing could be of use. Yet, since in our case dummy values have to be provided at run time, a more complex system may be necessary. Promising recent work by Singh et al. show how advances in artificial intelligence could lead to systems that can provide well-formatted dummy inputs [77]. On the other hand, our system provides the ability to compare outputs produced by legacy code when differing input is applied. This could be useful for testing of

hyperproperties [25, 44]. Besides, it also allows to automatically group inputs into equivalence classes based on their effect on outputs, which could aid change impact analysis and legacy code forensics. Finally, it could be used to apply game-based reenforcement learning to bug hunting, with one player trying to produce safe output and the other player trying to crash the target or leak information.

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, page 340–353, New York, NY, USA, 2005. Association for Computing Machinery.

[2] Maximilian Algehed and Cormac Flanagan. Transparent ifc enforcement: Possibility and (in)efficiency results. In *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*, pages 65–78, 2020.

[3] Maximilian Algehed, Alejandro Russo, and Cormac Flanagan. Optimising faceted secure multi-execution. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 1–115, 2019.

[4] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. An In-Depth analysis of disassembly on Full-Scale x86/x64 binaries. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 583–600, Austin, TX, August 2016. USENIX Association.

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 259–269, New York, NY, USA, 2014. Association for Computing Machinery.

[6] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In Sushil Jajodia and Javier Lopez, editors, *Computer Security - ESORICS 2008*, pages 333–348, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[7] Thomas H. Austin and Cormac Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Pro-*

gramming Languages and Analysis for Security, PLAS '10, New York, NY, USA, 2010. Association for Computing Machinery.

[8] 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, page 165–178, New York, NY, USA, 2012. Association for Computing Machinery.

[9] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In Evelyn Duesterwald, editor, *Compiler Construction*, pages 5–23, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[10] Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6), aug 2010.

[11] Sébastien Bardin, Philippe Herrmann, Jérôme Leroux, Olivier Ly, Renaud Tabary, and Aymeric Vincent. The bincoa framework for binary code analysis. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 165–170, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[12] G. Barthe, P.R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.*, pages 100–114, 2004.

[13] Gilles Barthe, Juan Manuel Crespo, Dominique Devriese, Frank Piessens, and Exequiel Rivas. Secure multi-execution through static program transformation. In Holger Giese and Grigore Rosu, editors, *Formal Techniques for Distributed Systems*, pages 186–202, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[14] Erick Bauman, Zhiqiang Lin, and Kevin W Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[15] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 335–348, USA, 2012. USENIX Association.

[16] Nataliia Bielova and Tamara Rezk. A taxonomy of information flow monitors. In Frank Piessens and Luca Viganò, editors, *Principles of Security and Trust*, pages 46–67, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[17] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. Bap: A binary analysis platform. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 463–469, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[18] David Brumley, Ivan Jager, Edward J Schwartz, and Spencer Whitman. The bap handbook, 2013.

[19] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 353–368, Washington, D.C., August 2013. USENIX Association.

[20] Roberto Capizzi, Antonio Longo, V.N. Venkatakrishnan, and A. Prasad Sistla. Preventing information leaks through shadow executions. In *2008 Annual Computer Security Applications Conference (ACSAC)*, pages 322–331, 2008.

[21] Lorenzo Cavallaro, Prateek Saxena, and R Sekar. Anti-taint-analysis: Practical evasion techniques against information flow based malware defense. *Secure Systems Lab at Stony Brook University, Tech. Rep*, pages 1–18, 2007.

[22] Roderick Chapman and Adrian Hilton. Enforcing security and safety models with an information flow analysis tool. In *Proceedings of the 2004 Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-Time & Distributed Systems Using Ada and Related Technologies*, SIGAda '04, page 39–46, New York, NY, USA, 2004. Association for Computing Machinery.

[23] Cristina Cifuentes and K. John Gough. Decompilation of binary programs. *Software: Practice and Experience*, 25(7):811–829, 1995.

[24] David Clark and Sebastian Hunt. Non-interference for deterministic interactive programs. In Pierpaolo Degano, Joshua Guttman, and Fabio Martinelli, editors, *Formal Aspects in Security and Trust*, pages 50–66, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[25] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *2008 21st IEEE Computer Security Foundations Symposium*, pages 51–65, 2008.

[26] Ellis Cohen. Information transmission in computational systems. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, SOSP '77, page 133–139, New York, NY, USA, 1977. Association for Computing Machinery.

[27] Benjamin Cox and David Evans. N-variant systems: A secretless framework for security through diversity. In *15th USENIX Security Symposium (USENIX Security 06)*, Vancouver, B.C. Canada, July 2006. USENIX Association.

[28] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. Flowfox: A web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 748–759, New York, NY, USA, 2012. Association for Computing Machinery.

[29] Bjorn De Sutter, K De Bosschere, Peter Keyngnaert, and Bart Demoen. On the static analysis of indirect control transfers in binaries. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, volume 2, pages 1013–1019, 2000.

[30] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, may 1976.

[31] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, jul 1977.

[32] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *2010 IEEE Symposium on Security and Privacy*, pages 109–124, 2010.

[33] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 901–913, New York, NY, USA, 2015. Association for Computing Machinery.

[34] Emmanuel Fleury, Olivier Ly, Gérald Point, and Aymeric Vincent. Insight: An open binary analysis framework. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 218–224. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[35] Andrea Flexeder, Bogdan Mihaila, Michael Petter, and Helmut Seidl. Interprocedural control flow reconstruction. In Kazunori Ueda, editor, *Programming Languages and Systems*, pages 188–203. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[36] Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Highly precise taint analysis for android applications. Technical report, TU Darmstadt, Tech. Rep, 2013.

[37] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11, 1982.

[38] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information flow analysis of android applications in droidsafe. In *2015 Network and Distributed System Security (NDSS)*, 2015.

[39] Ivan Gotovchits, Rijnard van Tonder, and David Brumley. Saluki: finding taint-style vulnerabilities with static property checking. In *Proceedings of the NDSS Workshop on Binary Analysis Research*, volume 2018, 2018.

[40] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1):175–205, jan 2006.

[41] Mauro Jaskelioff and Alejandro Russo. Secure multi-execution in haskell. In Edmund Clarke, Irina Virbitskaite, and Andrei Voronkov, editors, *Perspectives of Systems Informatics*, pages 170–178, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[42] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *2011 IEEE Symposium on Security and Privacy*, pages 413–428, 2011.

[43] Johannes Kinder. *Static Analysis of x86 Executables*. PhD thesis, Technische Universität Darmstadt, 2010.

[44] Johannes Kinder. Hypertesting: The case for automated testing of hyperproperties. In *Workshop on Hot Issues in Security Principles and Trust (HotSpot)*, 2015.

[45] Koen Koning, Herbert Bos, and Cristiano Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In

2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pages 431–442, 2016.

[46] Elisavet Kozyri, Owen Arden, Andrew C. Myers, and Fred B. Schneider. Jrif: Reactive information flow control for java. In Joshua D. Guttman, Carl E. Landwehr, José Meseguer, and Dusko Pavlovic, editors, *Foundations of Security, Protocols, and Equational Reasoning: Essays Dedicated to Catherine A. Meadows*, pages 70–88. Springer International Publishing, Cham, 2019.

[47] 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 Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 503–515, New York, NY, USA, 2016. Association for Computing Machinery.

[48] Arun Lakhotia and Prabhat K Singh. Challenges in getting formal with viruses. *Virus Bulletin*, 9(1):14–18, 2003.

[49] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 278–289, New York, NY, USA, 2007. Association for Computing Machinery.

[50] Gurvan Le Guernic. *Confidentiality enforcement using dynamic information flow analyses*. PhD thesis, Kansas State University, 2007.

[51] HJ Lu, Michael Matz, J Hubicka, A Jaeger, and M Mitchell. System v application binary interface. *AMD64 Architecture Processor Supplement*, 2018.

[52] Jean-Joseph Marty, Lucas Franceschino, Jean-Pierre Talpin, and Niki Vazou. Lio*: Low level information flow control in f*. *hal-03137132*, 2020.

[53] Xiaozhu Meng, Jonathon M. Anderson, John Mellor-Crummey, Mark W. Krentel, Barton P. Miller, and Srđan Milaković. Parallelizing binary code analysis, 2020.

[54] Xiaozhu Meng and B Miller. Binary code is not easy. Technical report, Tech. rep., Computer Sciences Department, University of Wisconsin, Madison, 2015.

[55] Bogdan Mihaila. *Adaptable Static Analysis of Executables for proving the Absence of Vulnerabilities*. PhD thesis, Technische Universität München, 2015.

[56] Dimiter Milushev, Wim Beck, and Dave Clarke. Noninterference via symbolic execution. In Holger Giese and Grigore Rosu, editors, *Formal Techniques for Distributed Systems*, pages 152–168, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[57] Andrew C Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow. *Software release. Located at http://www. cs. cornell. edu/jif*, 2005, 2001.

[58] Microsoft Developer Network. Control flow guard. `https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx`, 2015. Last checked 2017-07-13.

[59] Minh Ngo, Frank Piessens, and Tamara Rezk. Impossibility of precise and sound termination-sensitive security enforcements. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 496–513, 2018.

[60] Tobias Pfeffer and Sabine Glesner. Timing-sensitive synchronization for efficient secure multi-execution. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, CCSW'19, page 153–164, New York, NY, USA, 2019. Association for Computing Machinery.

[61] Tobias Pfeffer, Thomas Göthel, and Sabine Glesner. *Efficient and Precise Information Flow Control for Machine Code through Demand-Driven Secure Multi-Execution*, page 197–208. Association for Computing Machinery, New York, NY, USA, 2019.

[62] Tobias Pfeffer, Thomas Göthel, and Sabine Glesner. Automatic analysis of critical sections for efficient secure multi-execution. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, pages 318–325, 2019.

[63] Tobias Pfeffer, Paula Herber, Lucas Druschke, and Sabine Glesner. Efficient and safe control flow recovery using a restricted intermediate language. In *2018 IEEE 27th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 235–240, 2018.

[64] Tobias F. Pfeffer, Stefan Sydow, Joachim Fellmuth, and Paula Herber. Protecting legacy code against control hijacking via execution location equivalence checking. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 230–241, 2016.

[65] Willard Rafnsson and Andrei Sabelfeld. Secure multi-execution: Fine-grained, declassification-aware, and transparent. *Journal of Computer Security*, 24(1):39–90, 2016.

[66] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[67] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[68] K. Sagonas, M. Pettersson, R. Carlsson, P. Gustafsson, and T. Lindahl. All you wanted to know about the hipe compiler: (but might have been afraid to ask). In *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang*, ERLANG '03, page 36–42, New York, NY, USA, 2003. Association for Computing Machinery.

[69] Ravi S. Sandhu. Lattice-based access control models. *Computer*, 26(11):9–19, 1993.

[70] Konstantin Scherer, Tobias Pfeffer, and Sabine Glesner. I/o interaction analysis of binary code. In *2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 225–230, 2019.

[71] Thomas Schmitz, Maximilian Algehed, Cormac Flanagan, and Alejandro Russo. Faceted secure multi execution. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1617–1634, New York, NY, USA, 2018. Association for Computing Machinery.

[72] Daniel Schoepe, Musard Balliu, Benjamin C. Pierce, and Andrei Sabelfeld. Explicit secrecy: A policy for taint tracking. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 15–30, 2016.

[73] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 45–54, 2002.

[74] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. *2015 Network and Distributed System Security (NDSS)*, 2015.

[75] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016.

[76] Vincent Simonet. The flow caml system. *Software release. Located at http://cristal.inria.fr/˜simonet/soft/flowcaml*, 116:119–156, 2003.

[77] Rishabh Singh, William Blum, and Mohit Rajpal. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596*, November 2017.

[78] Gregor Snelting, Dennis Giffhorn, Jürgen Graf, Christian Hammer, Martin Hecker, Martin Mohr, and Daniel Wasserrab. Checking probabilistic noninterference using joana. *it-Information Technology*, 56(6):280–287, 2014.

[79] 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 R. Sekar and Arun K. Pujari, editors, *Information Systems Security*, pages 1–25, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[80] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, page 48–62, USA, 2013. IEEE Computer Society.

[81] Jack Tang and Trend Micro Threat Solution Team. Exploring control flow guard in windows 10. *Available at ht tp://blog. trendmicro. c om/trendlabs-security-intelligence/exploring-control-flow-guard-in-windows-10*, 2015.

[82] Tachio Terauchi and Alex Aiken. Secure information flow as a safety problem. In Chris Hankin and Igor Siveroni, editors, *Static Analysis*, pages 352–367, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[83] 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 *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, San Diego, CA, August 2014. USENIX Association.

[84] Stijn Volckaert, Bart Coppens, and Bjorn De Sutter. Cloning your gadgets: Complete rop attack immunity with multi-variant execution. *IEEE Transactions on Dependable and Secure Computing*, 13(4):437–450, 2016.

[85] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. From hack to elaborate technique—a survey on binary rewriting. *ACM Comput. Surv.*, 52(3), jun 2019.

[86] C.S. Wong, I.K.T. Tan, R.D. Kumari, J.W. Lam, and W. Fun. Fairness and interactive performance of o(1) and cfs linux kernel schedulers. In *2008 International Symposium on Information Technology*, volume 4, pages 1–8, 2008.

[87] Liang Xu, Fangqi Sun, and Zhendong Su. Constructing precise control flow graphs from binaries. Technical report, University of California, Davis, 2009.

[88] Aydan R. Yumerefendi, Benjamin Mickle, and Landon P. Cox. Tightlip: Keeping applications from spilling the beans. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*, Cambridge, MA, April 2007. USENIX Association.

[89] Dante Zanarini, Mauro Jaskelioff, and Alejandro Russo. Precise enforcement of confidentiality for reactive systems. In *2013 IEEE 26th Computer Security Foundations Symposium*, pages 18–32, 2013.

[90] Stephan Arthur Zdancewic and Andrew Myers. *Programming languages for information security.* Cornell University, 2002.

[91] Gaofei Zhang, Zheng Zhang, Bolin Ma, and Jiang Wang. Multi-variant execution: State-of-the-art and research challenges. In *2020 12th International Conference on Communication Software and Networks (ICCSN)*, pages 196–201, 2020.

Appendix A

Benchmark Programs

```
1 #include "../common.h"

2 void run() {
3     // for i = 0 to secret {output i on
        public_channel};
4     h = input(Hin);
5     for (int i = 0; i < h; i ++)
6         output(Lout, i);
7 }
```

Listing A.1: Benchmark program a01.c

```
1 #include "../common.h"

2 void run() {
3     // for i = 0 to secret {output i on
        public_channel}; while true do skip;
4     h = input(Hin);
5     for (int i = 0; i < h; i ++)
6         output(Lout, i);
7     while (1);
8 }
```

Listing A.2: Benchmark program a02.c

```
1  #include "../common.h"

2  void run() {
3      // for i = 0 to maxNat {output i on
          public_channel; if (i = secret) then (while
          true do skip)};
4      h = input(Hin);
5      for (int i = 0; i < 1000; i ++) {
6          output(Lout, i);
7          if (i == h) {
8              while (1);
9          }
10     }
11 }
```

Listing A.3: Benchmark program a03.c

```
1  #include "../common.h"

2  void run() {
3      // if (secret = 0) then (while true do skip);
          public = 0
4      h = input(Hin);
5      if (h == 0) {
6          while (1);
7      }
8      l = 0;
9  }
```

Listing A.4: Benchmark program a04.c

```
1  #include "../common.h"

2  void run() {
3      // output secret on public_channel; while true do
          skip
4      h = input(Hin);
5      output(Lout, h);
6      while (1);
7  }
```

Listing A.5: Benchmark program a05.c

```
1 #include "../common.h"

2 void run() {
3     // if h = 0 then l = 1; output l
4     h = input(Hin);
5     l = 0;
6     if (h == 0) l = 1;
7     output(Lout, l);
8 }
```

Listing A.6: Benchmark program b01.c

```
1 #include "../common.h"

2 void run() {
3     // l = 1; (while (h=1) do skip); output l
4     h = input(Hin);
5     l = 1;
6     while (h == 1);
7     output(Lout, l);
8 }
```

Listing A.7: Benchmark program b02.c

```
1 #include "../common.h"

2 void run() {
3     // if h = 0 then l = 1; l = 0; output l
4     h = input(Hin);
5     if (h == 0) l = 1;
6     l = 0;
7     output(Lout, l);
8 }
```

Listing A.8: Benchmark program b03.c

```
1  #include "../common.h"

2  void run() {
3      // if l = 0 then {while h = 0 do skip;} else {
          while h = 1 do skip}; output l
4      h = input(Hin);
5      l = input(Lin);
6      if (l == 0) {
7          while (h == 0);
8      } else {
9          while (h == 1);
10     }
11     output(Lout, l);
12 }
```

Listing A.9: Benchmark program b04.c

```
1  #include "../common.h"

2  void run() {
3      // if h = 0 then l = 0 else l = 0; output l
4      h = input(Hin);
5      if (h == 0) {
6          l = 0;
7      } else {
8          l = 0;
9      }
10     output(Lout, l);
11 }
```

Listing A.10: Benchmark program b05.c

```
1  #include "../common.h"

2  void run() {
3      // if h = 0 then l = 1; if l = 1 then l = 0;
          output l'
4      int l2  = 0;
5      h = input(Hin);
6      if (h == 0) l = 1;
7      if (l == 1) l = 0;
8      output(Lout, l2);
9  }
```

Listing A.11: Benchmark program b06.c

```
1  #include "../common.h"

2  void run() {
3      // if h = 0 then l = 0 else l = 1; if l = 0 then
          {while true do skip;} else l = 0; output l
4      h = input(Hin);
5      if (h == 0) {
6          l = 0;
7      } else {
8          l = 1;
9      }
10     if (l == 0) {
11         while (1);
12     } else {
13         l = 0;
14     }
15     output(Lout, l);
16 }
```

Listing A.12: Benchmark program b07.c

```
1 #include "../common.h"

2 void run() {
3     // l = 0; if h = 0 then skip else {while true do
          l = 1}; output l
4     h = input(Hin);
5     l = 0;
6     while (1) {l = 1;}
7     output(Lout, l);
8 }
```

Listing A.13: Benchmark program b08.c

```
1 #include "../common.h"

2 void run() {
3     // l = 0; if h = 0 then l = 0 else skip; output l
4     h = input(Hin);
5     l = 0;
6     if (h == 0) l = 0;
7     output(Lout, l);
8 }
```

Listing A.14: Benchmark program b09.c

```
1 #include "../common.h"

2 void run() {
3     // if h = 0 then l' = 1 else l = 1; output l
4     int l2 = 0;
5     h = input(Hin);
6     if (h == 0) {
7         l2 = 1;
8     } else {
9         l = 1;
10    }
11    output(Lout, l);
12 }
```

Listing A.15: Benchmark program b10.c

```
1  #include "../common.h"

2  void run() {
3      // if h = 0 then l = 0 else l = 1; if l = 0 then
          l' =  0 else l' = 1; output l'
4      int l2 = 0;
5      h = input(Hin);
6      if (h == 0) {
7          l = 0;
8      } else {
9          l = 1;
10     }
11     if (l == 0) {
12         l2 = 0;
13     } else {
14         l2 = 1;
15     }
16     output(Lout, l2);
17 }
```

Listing A.16: Benchmark program b11.c

```
1  #include "../common.h"

2  void run() {
3      // if h = 0 then l = 0 else l = 1; output l
4      h = input(Hin);
5      if (h == 0) {
6          l = 0;
7      } else {
8          l = 1;
9      }
10     output(Lout, l);
11 }
```

Listing A.17: Benchmark program b12.c

```
1  #include "../common.h"

2  void run() {
3      // if l = 0 then {if h = 1 then l = 1 else skip;}
           else {if h = 0 then l = 0 else skip;}; output
           l
4      h = input(Hin);
5      if (l == 0) {
6          if (h == 1) {
7              l = 1;
8          }
9      } else {
10         if (h == 1) {
11             l = 0;
12         }
13     }
14     output(Lout, l);
15 }
```

Listing A.18: Benchmark program b13.c

```
1  #include "../common.h"

2  void run() {
3      // if h = 0 then {while true do skip;} else {if h
           = 1 then l = 1 else l = 2;}; output l
4      h = input(Hin);
5      if (h == 0) {
6          while (1);
7      } else {
8          if (h == 1) {
9              l = 1;
10         } else {
11             l = 2;
12         }
13     }
14     output(Lout, l);
15 }
```

Listing A.19: Benchmark program b14.c

```
1  #include "../common.h"

2  void run() {
3      h = input(Hin);
4      if (h > 1) {
5          l = input(Lin);
6      }
7  }
```

Listing A.20: Benchmark program n01.c

```
1  #include "../common.h"

2  void run() {
3      output(Lout, 1);
4      h = input(Hin);
5      if (h > 1) {
6          while (1);
7      }
8  }
```

Listing A.21: Benchmark program n02.c

```
1  #include "../common.h"

2  void run() {
3      h = input(Hin);
4      if (h > 1) {
5          output(Lout, 0);
6      } else {
7          output(Lout, 1);
8      }
9      while (1);
10 }
```

Listing A.22: Benchmark program n03.c

```
1 #include "../common.h"

2 void run() {
3     h = input(Hin);
4     if (h > 1) {
5         output(Lout, 0);
6     } else {
7         output(Lout, 0);
8     }
9 }
```

Listing A.23: Benchmark program n04.c

Appendix B

Additional Results

B.1 Sort

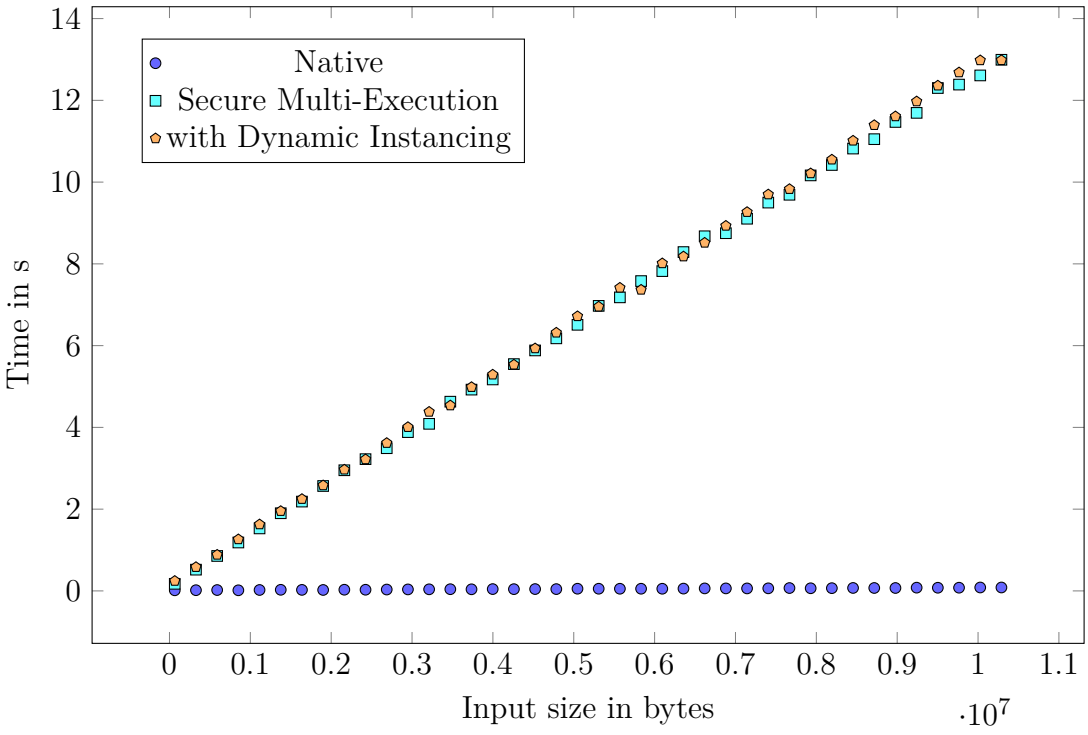


Figure B.1: Overhead sampling for different input sizes for `sort`

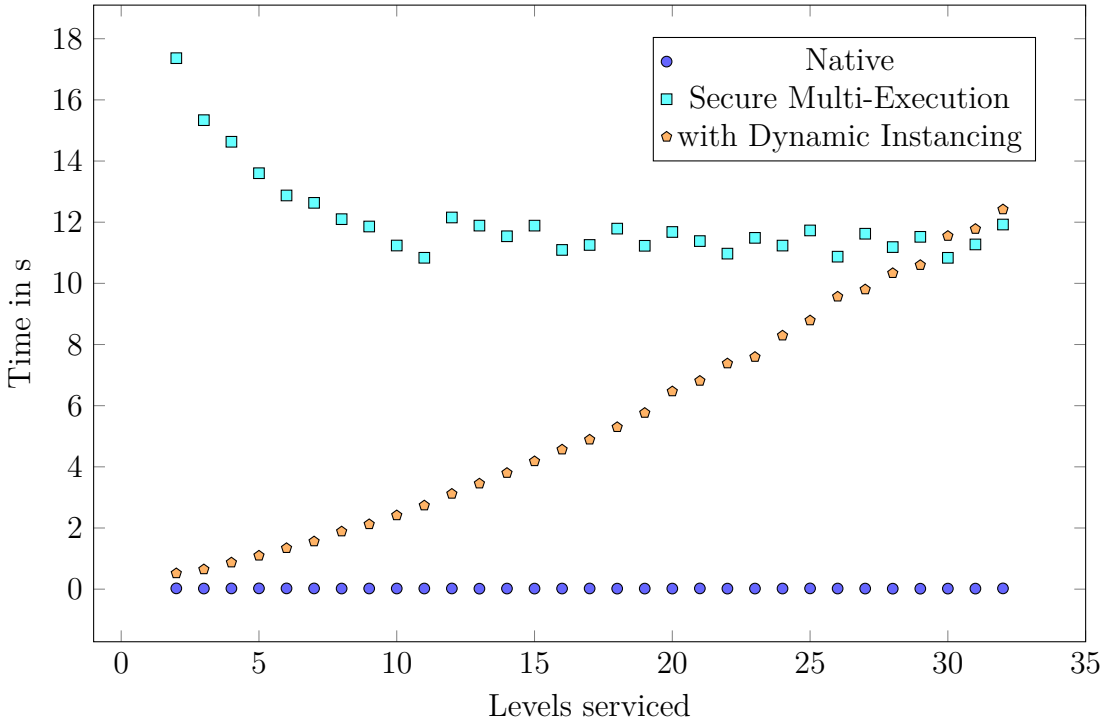
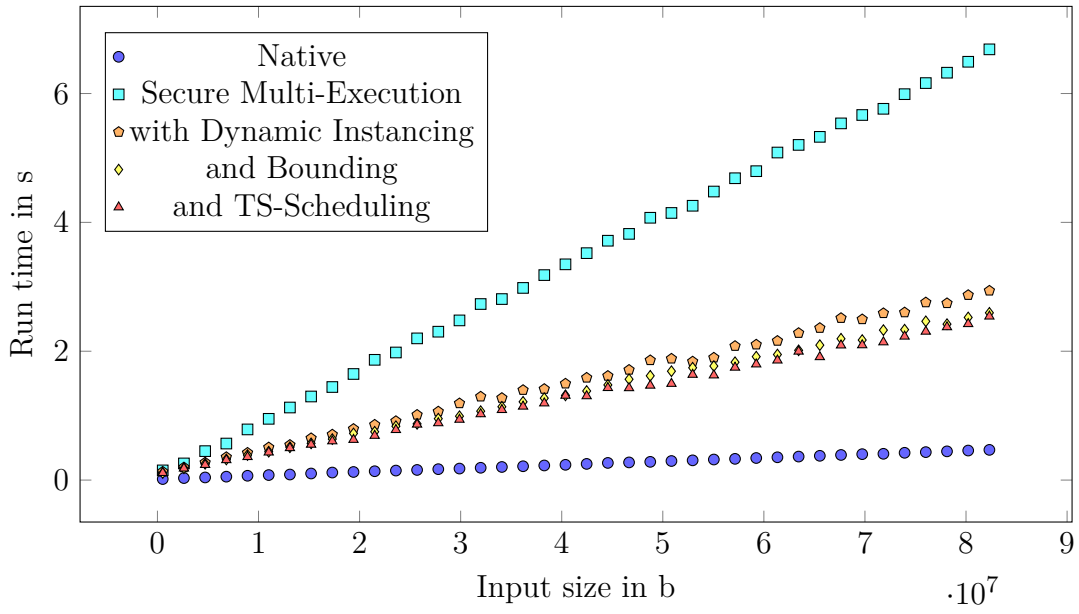
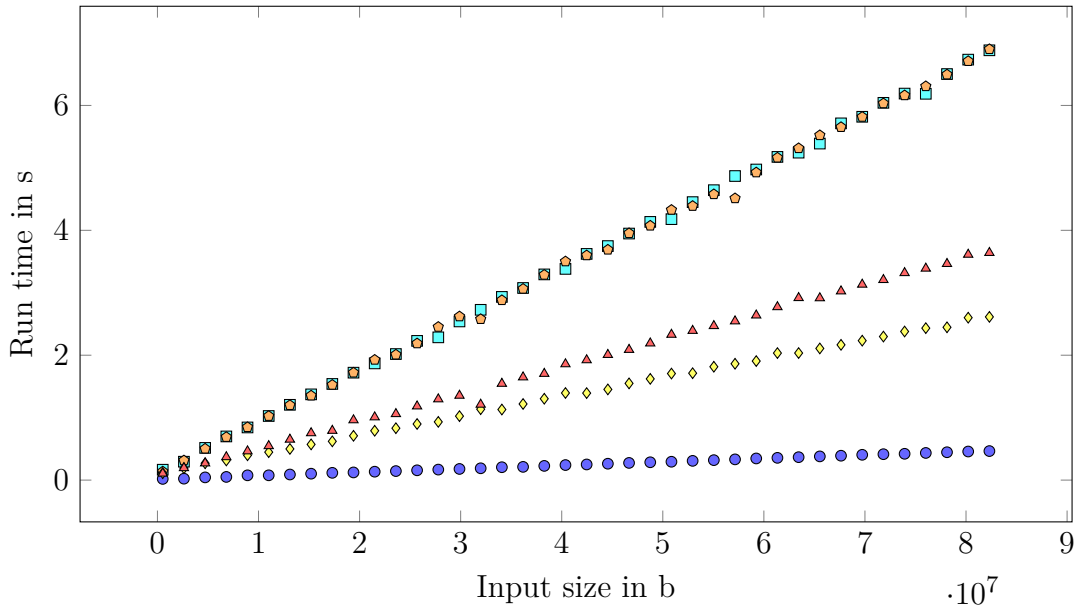


Figure B.2: Overhead sampling for different number of input levels for `sort`

B.2 SHA Sum

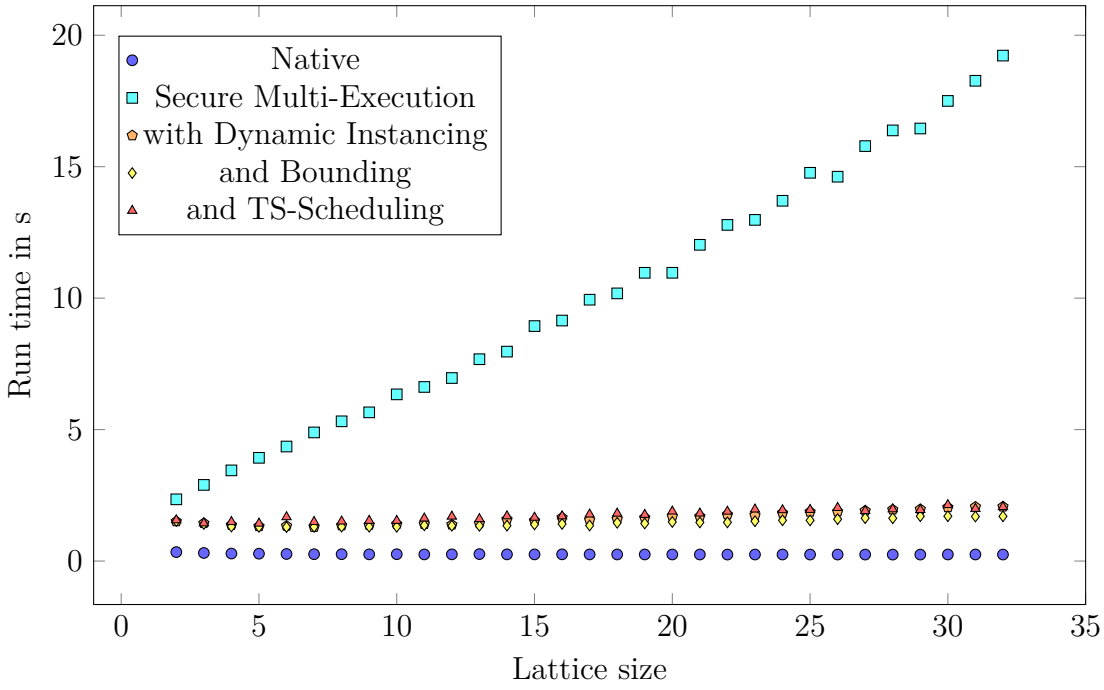


(a) LABHT-order

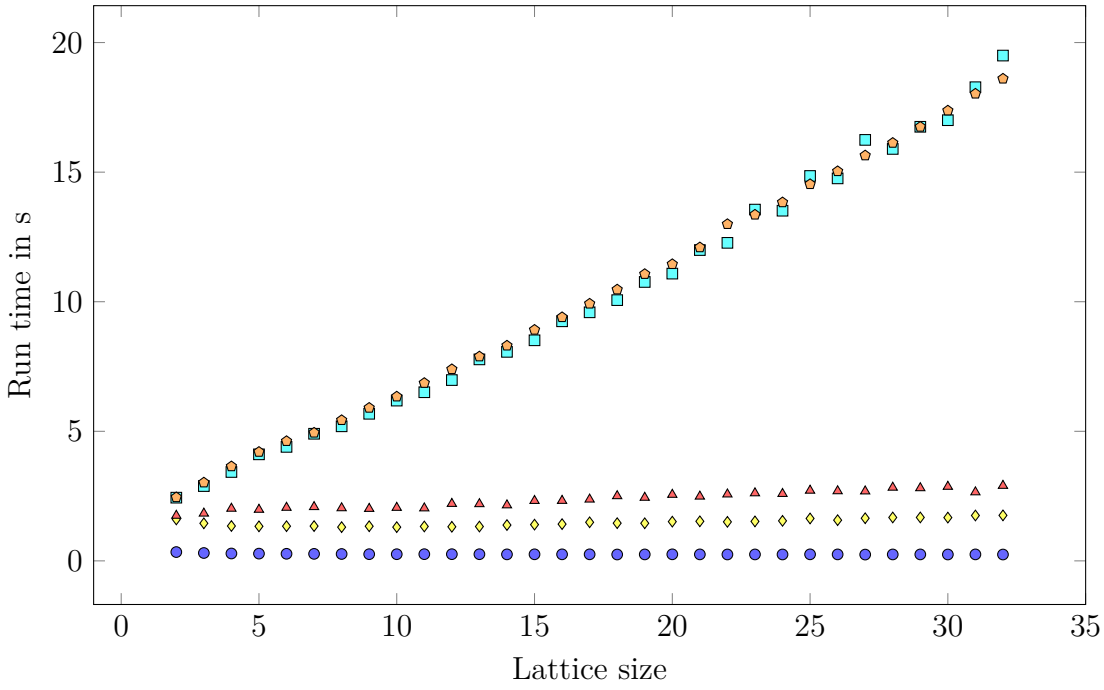


(b) THBAL-order

Figure B.3: Run-time sampling of `sha1sum`



(a) Ascending total order



(b) Descending total order

Figure B.4: Level sampling of `sha1sum`

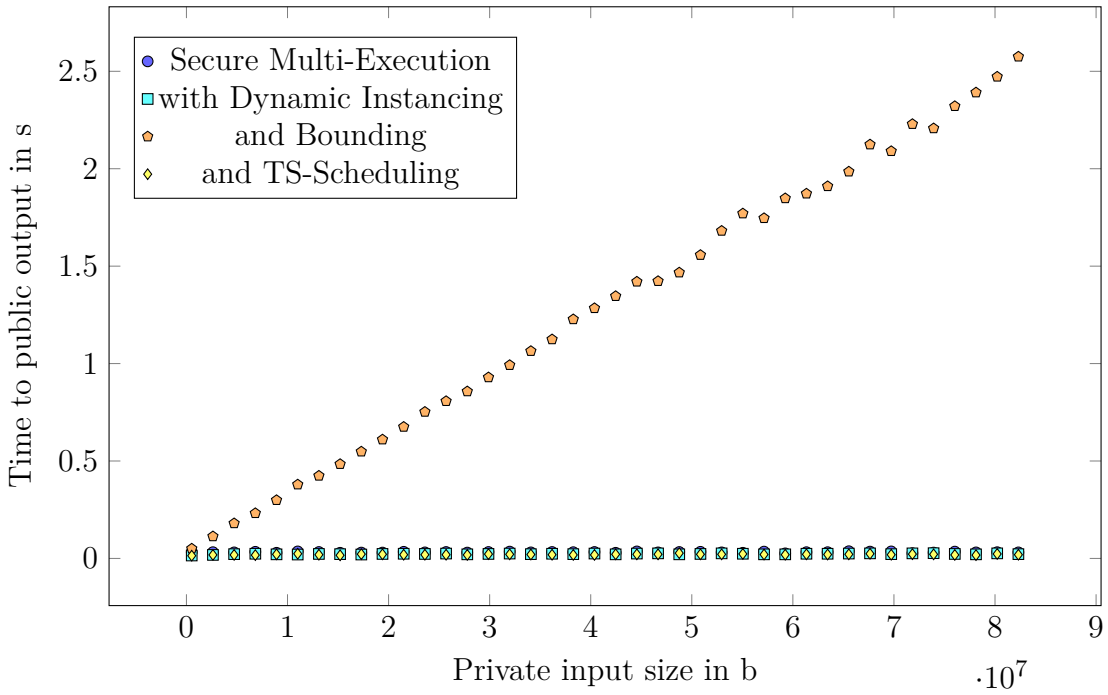


Figure B.5: Time sampling of the first output event

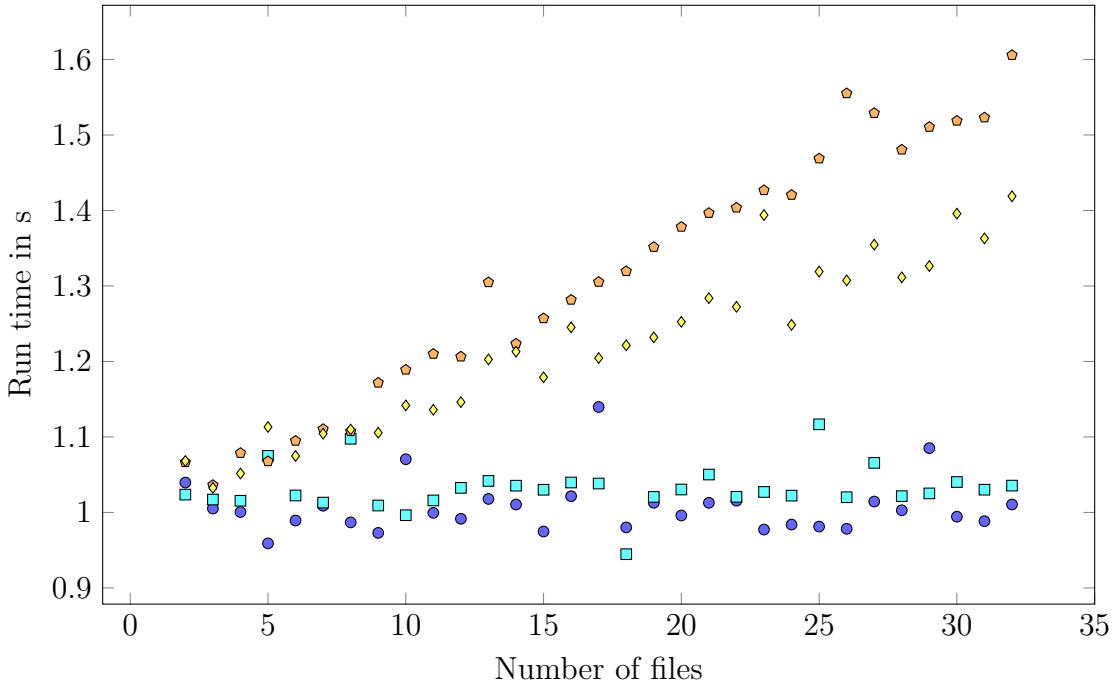
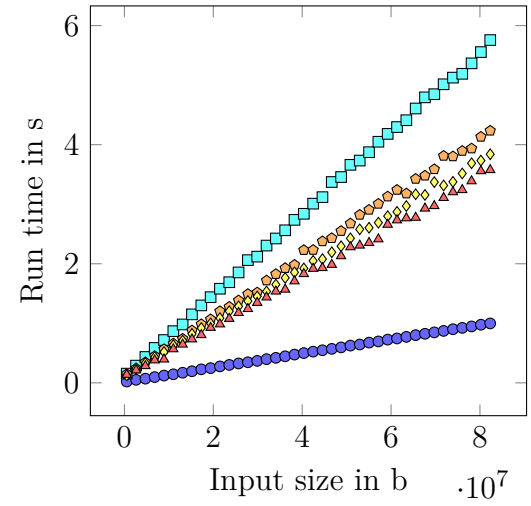
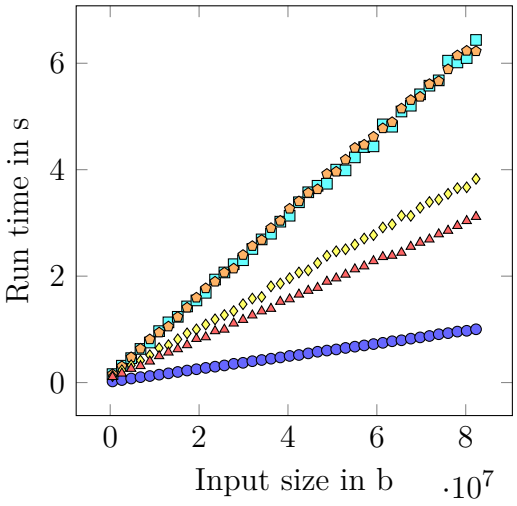


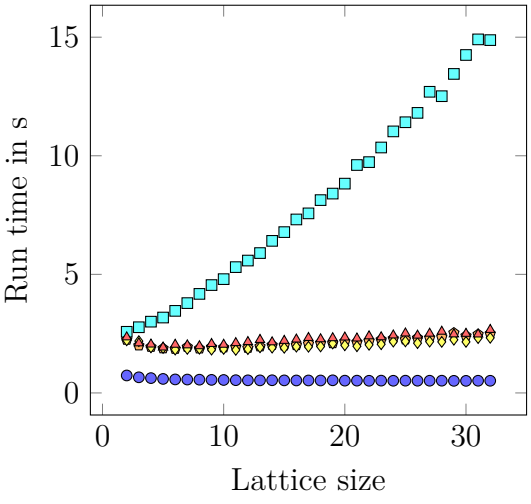
Figure B.6: File split sampling of `sha1sum`



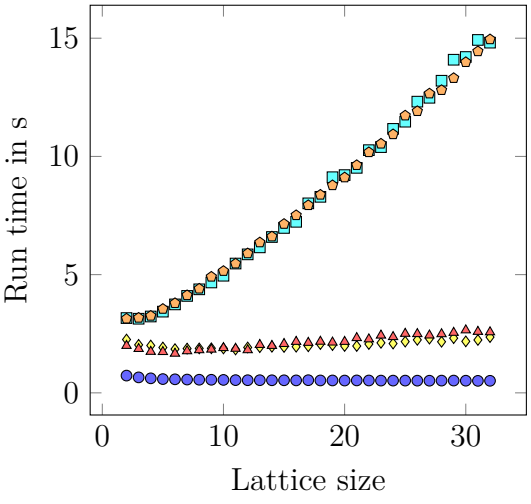
(a) LABHT-order



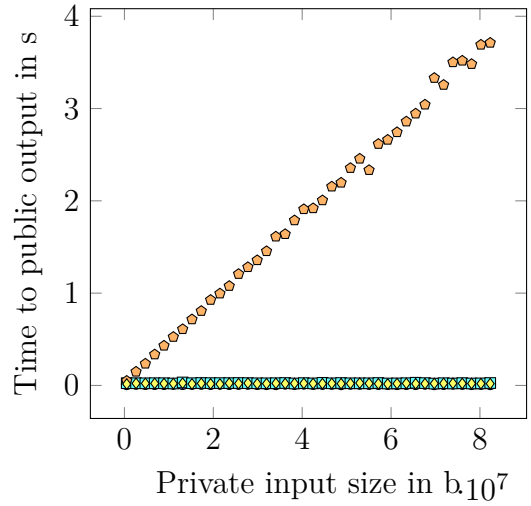
(b) THBAL-order



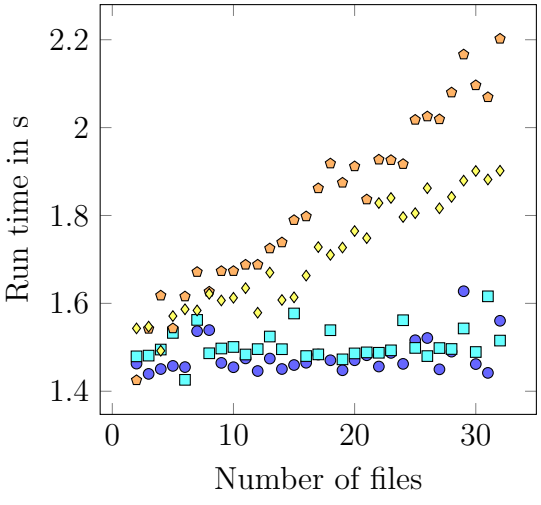
(c) Ascending total order



(d) Descending total order

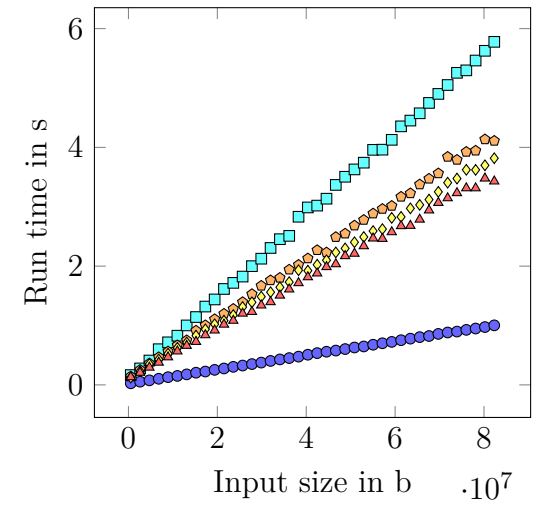


(e) Time sampling of the first output event

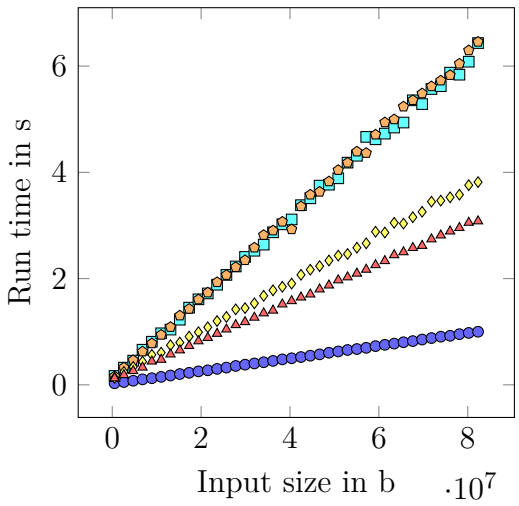


(f) File split sampling of `sha224sum`

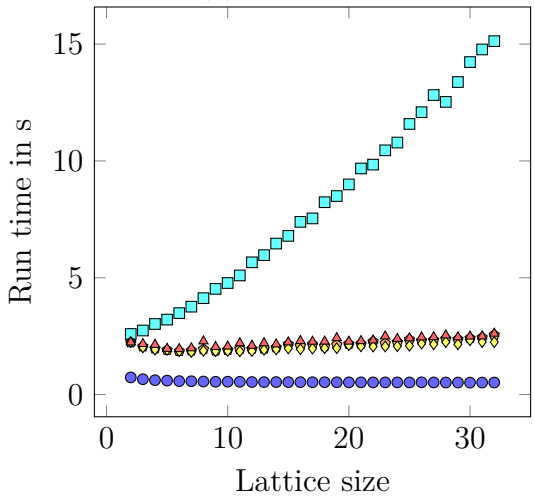
Figure B.7: Samplings of `sha224sum`



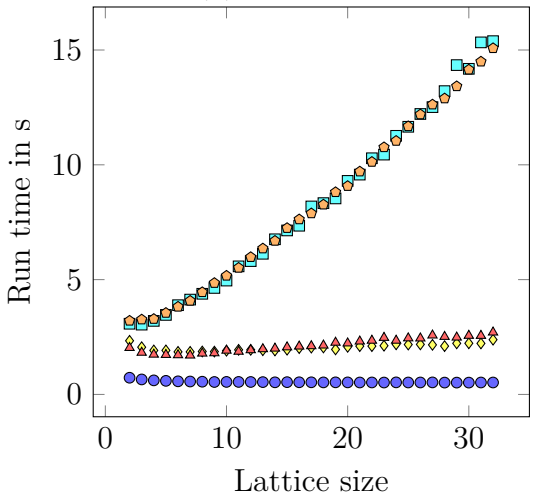
(a) LABHT-order



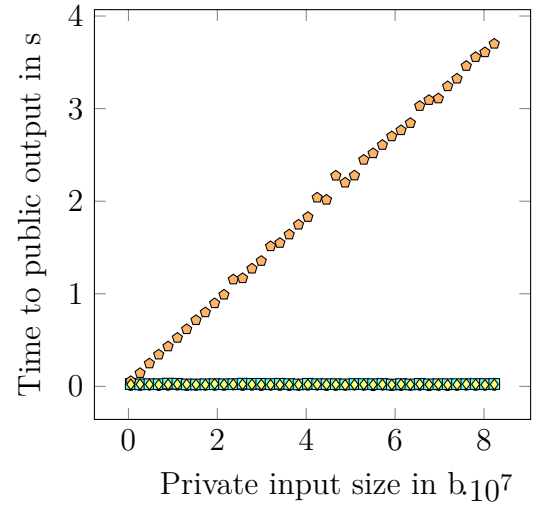
(b) THBAL-order



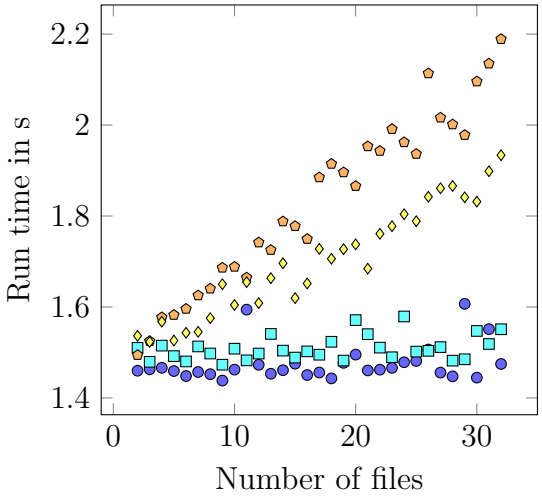
(c) Ascending total order



(d) Descending total order

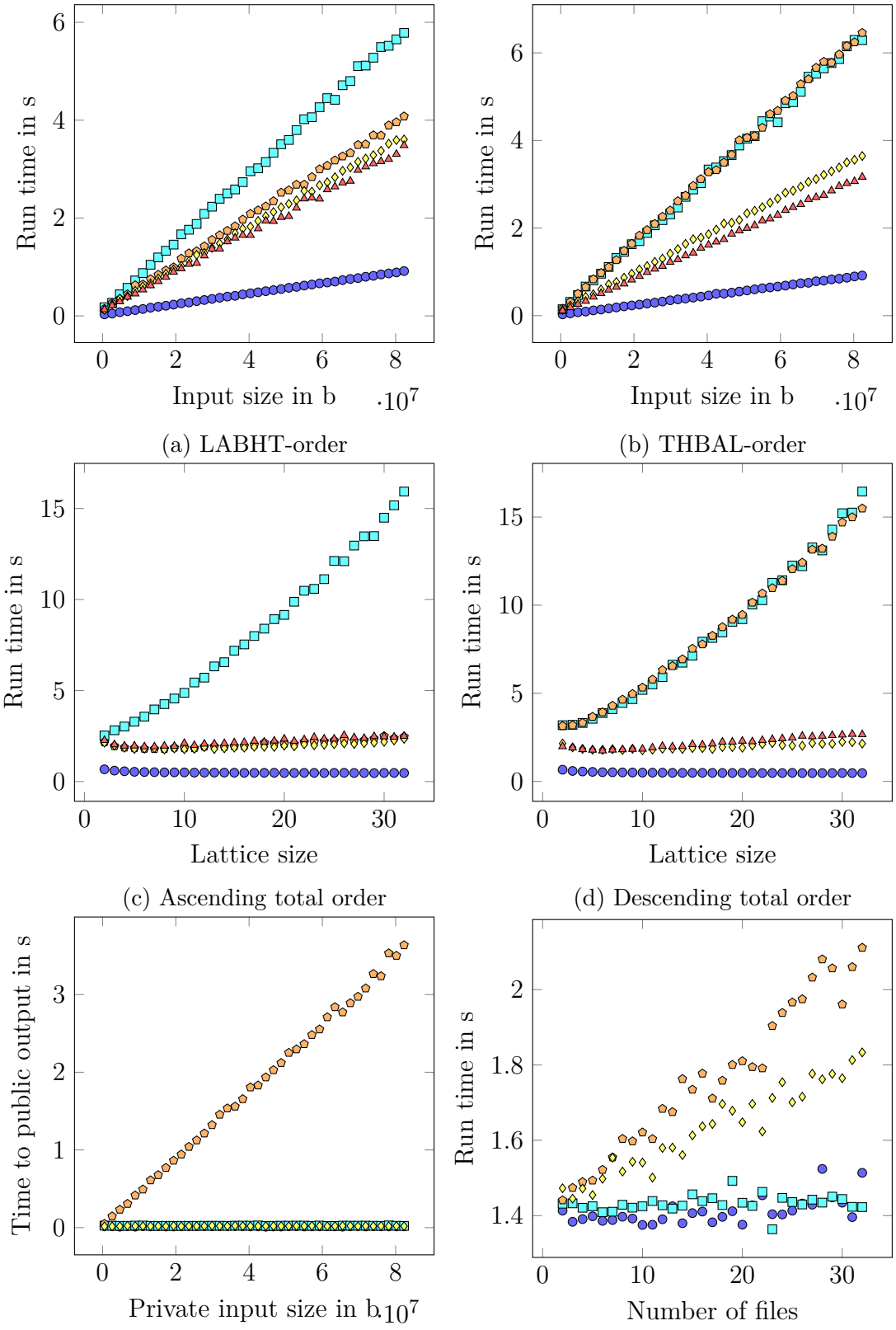


(e) Time sampling of the first output event



(f) File split sampling of `sha256sum`

Figure B.8: Samplings of `sha256sum`



(a) LABHT-order

(b) THBAL-order

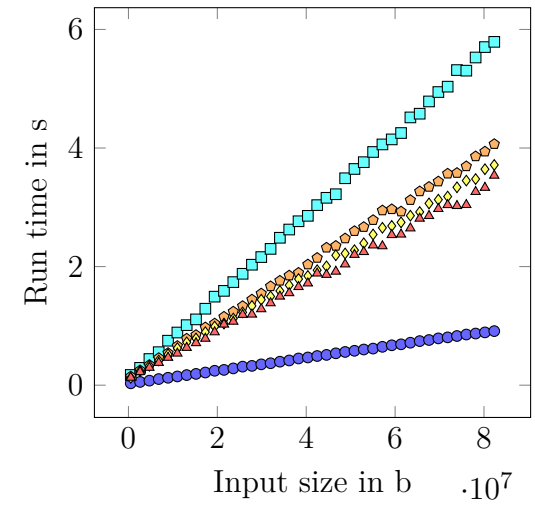
(c) Ascending total order

(d) Descending total order

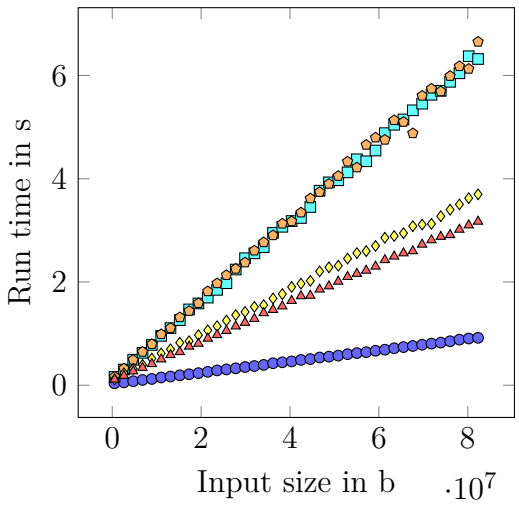
(e) Time sampling of the first output event

(f) File split sampling of `sha384sum`

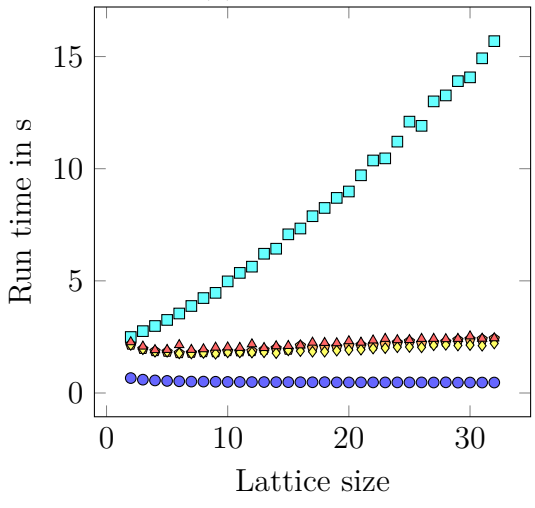
Figure B.9: Samplings of `sha384sum`



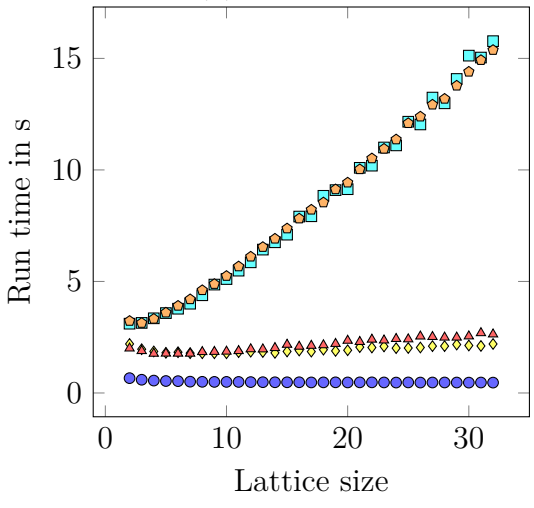
(a) LABHT-order



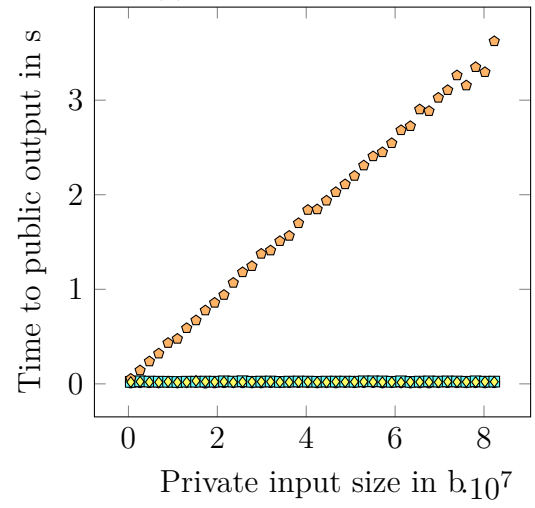
(b) THBAL-order



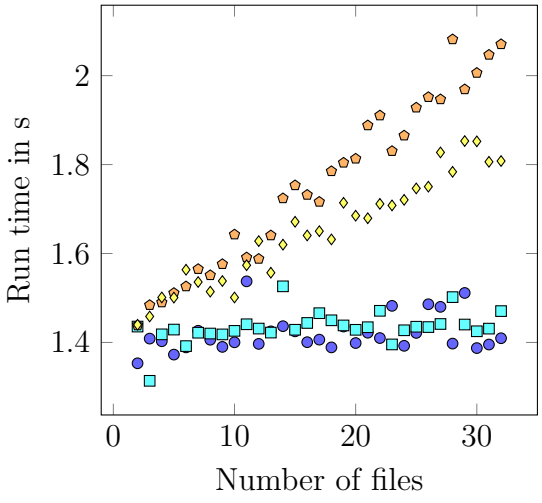
(c) Ascending total order



(d) Descending total order



(e) Time sampling of the first output event



(f) File split sampling of `sha512sum`

Figure B.10: Samplings of `sha512sum`

Appendix C

Stream Input

In the previous descriptions of our contributions, we generally assumed that the same input can be consumed by multiple executions and that input events are side-effect free. These assumptions are valid for all examples in this thesis, where we consider inputs from files. Yet, Secure Multi-Execution was originally designed to also cope with volatile, observable input, for example from (network) streams. To show that our methods are compatible with these settings, we discuss how our work can be extended to handle stream input here. First, we introduce the necessary changes to the semantics, then we discuss how these can be implemented for low-level code. Finally, we evaluate our approach with a short benchmark example from Rafnsson and Sabelfeld [65].

C.1 Interaction

A noteworthy extension to this fundamental definition of Secure Multi-Execution (SME) is the treatment of input that induces side-effects [32] and blocking input [65]. Input with side effects occurs, for example, when a user is asked to provide a value during execution. Following our SME definition, each execution equal or higher than the user's classification may request that value. Thus, the user may be asked multiple times, which harms the transparency of the approach. Also, the timing of the request may leak information, for example if the input is reached faster when the classified information conforms to some checks.

Additionally, requested input may not always be available. For example in a network communication, input may be consumed faster than it is produced. This may lead to producer-controlled delay in the consumer program, which could allow to leak information. Clark and Hunt investigated attacks based on the user's

Local:

$$\textsc{Input-Block}\frac{\pi(c) = \ell \qquad s \xrightarrow{c?\star} s'}{\ell, \pi, \delta, b, r \vdash s, p \xrightarrow{c?v} s, p}$$

$$\textsc{Input-New}\frac{\pi(c) = \ell \qquad s \xrightarrow{c?v} s' \qquad v \neq \star}{\ell, \pi, \delta, b, r \vdash s, p \xrightarrow{c?v} s', p}$$

$$\textsc{Input-Wait}\frac{s \xrightarrow{c?v} s' \qquad \pi(c) \sqsubset \ell \qquad p(c) > r(c)}{\ell, \pi, \delta, b, r \vdash s, p \xrightarrow{\bullet} s, p}$$

$$\textsc{Input-Old}\frac{\begin{array}{c} s \xrightarrow{c?v} s' \qquad \pi(c) \sqsubset \ell \\ p(c) \leq r(c) \qquad v = b(c, p(c)) \\ s \xrightarrow{c?v'} s'' \qquad p'(c) = p(c) + 1 \end{array}}{\ell, \pi, \delta, b, r \vdash s, p \xrightarrow{\bullet} s'', p'}$$

$$\textsc{Input-Dummy}\frac{s \xrightarrow{c?v} s' \qquad \pi(c) \not\sqsubseteq \ell \qquad s \xrightarrow{c?\delta} s^{\delta}}{\ell, \pi, \delta, b, r \vdash s, p \xrightarrow{\bullet} s^{\delta}, p}$$

Global:

$$\textsc{Step}\frac{b \vdash S(\ell) \xrightarrow{a} s' \qquad a = c?v \implies v = \star}{\ell : \sigma, b, r, S \xrightarrow{a} \sigma, b, r, S[\ell \leftarrow s']}$$

$$\textsc{Buffer}\frac{b \vdash S(\ell) \xrightarrow{c?v} s' \qquad b' = b[c \leftarrow b(c) : v] \qquad r' = r[c \leftarrow r(c) + 1]}{\ell : \sigma, b, r, S \xrightarrow{c?v} \sigma, b', r', S[\ell \leftarrow s']}$$

Semantics C.1: Semantics for volatile/visible input

behavior [24]. They show that for *deterministic* programs, user behavior can be abstracted as input streams. Thus, Rafnsson and Sabelfeld extend the notion of environments and introduce the special input value $\star$ to the environments [65]. Unlike other input values, the $\star$ is not buffered, such that $e(c)$ returns the next value even if not all $\star$-values have been consumed yet.

The Secure Multi-Execution semantics can then be extended to thwart leaks through side-effects or user-induced delays. The key idea is to only allow the execution at the same level to obtain new input from a channel. The (potentially volatile) input value is then buffered inside the monitor and forwarded to higher executions when needed. Consequently, if an execution requires this input before the respective execution has obtained it, the execution is blocked. Thus, no

information about sensitive inputs can leak through the input side-effect.

The necessary changes to the SME semantics are shown in Semantics C.1. Here, we introduce new local and global rules. The major difference to our previous definition of the local SME semantics is the introduction of local input pointers for each channel, denoted $p : \mathbb{C} \mapsto \mathbb{N}$. This pointer keeps track of how much input has been obtained from a channel, representing the current cursor position in the global buffer. The global buffer b is a list of inputs for each channel that allows random access through a cursor position. Finally, the global input pointer $r : \mathbb{C} \mapsto \mathbb{N}$ denotes the amount of input obtained for a channel. It allows to check whether a higher execution is requesting lower input faster than the corresponding execution.

In contrast to the previews definition of SME, we use four different local input rules here. The Input-Block rule shows how a non-total environment is handled by idling on the same state. Conversely, when the environment is ready to provide new input on channel c, the Input-New rule shows how it can be obtained by the fitting execution. When input from lower channels is requested, both the Input-Wait and Input-Old rules apply. We check whether input is available in the global buffer by comparing the local input pointer p with the global input pointer r for that channel. When the execution is ahead of the buffer, we wait. Otherwise, we obtain the value v from the buffer for channel c, denoted $b(c)$, at the location pointed to by $p(c)$. We then feed this buffered input to the execution and advance the local input pointer. Finally, in case an execution requires input from a higher channel, we perform the usual Input-Dummy rule.

In the global rules, we Step the executions as usual, unless they obtain new actual, input. When this is the case, be Buffer the new input value by appending it to the corresponding buffer $b(c)$, and increase the global input pointer for that channel. As a consequence of these semantics, new input is obtained only once by the according execution and input delays from higher channels cannot effect lower executions.

C.2 Virtual Filesystem

We implement the semantics from Semantics C.1 through a virtual file system. Since our implementation runs in user-space, we cannot extend the kernel to buffer streaming input. Instead, we buffer all input in the monitoring process. Executions interacting with blocking streams do not return from the corresponding system call until either input is provided or the endpoint is terminated. Thus, the Input-Block and Input-New rules follow from the behavior of the operating system.

```
1 #include "../common.h"

2 void run() {
3     // in H h ; out L 0
4     h = input(Hin);
5     output(Lout, l);
6 }
```

Listing C.1: Insecure input-delaying program from [65]

```
1 #include "../common.h"

2 void run() {
3     // in H h ; out L 0
4     l = input(Lin);
5     output(Lout, l);
6 }
```

Listing C.2: Secure input-delaying program from [65]

When input from a lower channel is obtained, we compare our buffer size with the local input pointer for the corresponding execution. When the execution is ahead of the buffer, we implement the Input-Wait rule by performing a `wait` on the buffer in the corresponding monitoring thread. The execution is thus descheduled until awoken by a corresponding call to `notify`. This call is issued when the responsible execution execution writes new input to the buffer. Then, all waiting executions progress, obtain the new values from the buffer, and advance their respective local input pointers.

C.3 Example

To demonstrate the security of our implementation, we use two example programs from Rafnsson and Sabelfeld [65]. Listing C.1 shows an insecure program, where a delay in the sensitive H-input normally leads to a delay in the public L-output. Conversely, Listing C.2 shows a secure program, where the timestamp of the L-output depends only on L-input. This is reflected in the evaluation results shown in Table C.1.

Here, we fed input to both programs with increasing delays. At the maximum,

		Security						*Transparency*					
	Native	PSNI			TSNI			TLT			PCT		
ID	TSNI	S	B	Q	S	B	Q	S	B	Q	S	B	Q
Listing C.1	0.401	✓	✓	✓	0.001	0.405	0.003	✓	✓	✓	✓	✓	✓
Listing C.2	0.001	✓	✓	✓	0.002	0.002	0.001	✓	✓	✓	✓	✓	✓

Table C.1: Evaluation results

we delayed the private input by 0.4 seconds. This shows in the measurement of the low outputs from the insecure example. Here, the average delay across multiple executions was 0.401 seconds, directly correlating with the private input delay. In contrast, the average output delay in the secure example was 0.001 seconds, which can be attributed to noise on the system and thus does not constitute an information leak.

Note that both programs are PSNI-secure, which our enforcement could preserve. Also, the programs produced the same outputs under enforcement, reflected in the transparency results of our evaluation. With regarding to thwarting of the input-delay attacks, both our Secure Multi-Execution and queue-based optimization achieved timing-sensitive noninterference for the insecure example. Only our barrier-based enforcement did not, as expected. Here, the delay in the private input leads to a delay in the private execution which subsequently reaches the barrier later and thus also affects progress in the public execution.

fin.