

POLITECNICO DI TORINO

Master's Degree in Cybersecurity



Master's Degree Thesis

Software Reverse Engineering: Statistical Defense and Attack Modeling

Supervisors:

Prof. Cataldo Basile

Prof. Bjorn De Sutter

Candidate:

Michele Seira

Academic Year 2025/2026
Torino

Abstract

Software is often distributed in environments that are not controlled by its developers. Once a program runs on a machine controlled by a potential attacker, it can be inspected, modified, and executed under different conditions. For this reason, software protections against reverse engineering are not usually expected to prevent attacks completely. Their goal is rather to increase the amount of work needed to understand or bypass the protected program.

Evaluating whether a protection achieves this goal is not straightforward. Software metrics can describe some properties of a protected program, but they do not directly measure the effort required by a concrete attack. Experiments with human reverse engineers can provide useful evidence, but they are expensive, difficult to repeat, and hard to scale. As a result, the choice of which protection to use often remains based on partial evidence and expert judgement.

This thesis studies a more explicit way to evaluate the impact of software protections on reverse-engineering attacks. The work builds on a recent proposal for a statistical modelling approach in which attacks are described as repeatable procedures and then executed against different protected versions of the same program. In this way, the effect of a protection can be observed through the changes it introduces in the attack process and through the additional work it imposes on the attacker. The case study considered in this thesis is based on code guards, a tamper-proofing technique in which a program checks at runtime whether parts of its code have been modified. Several protected versions of a sample application are produced by using different code-guard configurations. These configurations range from simple designs, where the relation between the hash computation and the hash verification can be recovered with limited analysis, to designs that make this relation harder to reconstruct. Three attack strategies are implemented and evaluated on these versions. The first strategy analyzes a single tampered execution, while the other two compare clean and tampered executions to focus the analysis on the effects of the modification. This makes it possible to study how each protection changes the attack effort and which strategies remain effective as the protection becomes more complex.

The thesis shows that this form of modelling can be applied to concrete protections and can produce measurable evidence of their effect. The results show how different configurations change the estimated effort required by an attack and which attack strategies remain effective once a particular protection is in place. The approach does not provide an absolute measure of security and does not replace the expertise of a human reverse engineer.

Instead, it offers a repeatable and comparable basis for studying protected software variants, with a method that can be extended to other protection techniques and application domains.

Table of Contents

List of Figures	IX
1 Introduction	1
1.1 Objectives	2
1.2 Major Outcomes	2
1.3 Structure of the Thesis	3
2 Background	4
2.1 Software Protection and Tamper-Proofing	4
2.2 Tamper-Proofing and Obfuscation	5
2.3 Obfuscation as a Complementary Protection	5
2.4 The MATE Threat Model	6
2.4.1 Integrity Assets in a MATE Setting	7
2.4.2 Attacker Capabilities and Analysis Techniques	8
2.5 Practical Limits of Client-Side Protection	8
2.6 Tamper-Proofing as Attack-Cost Increase	9
2.7 Desired Properties of a Tamper-Proofing Mechanism	9
2.8 Local and Remote Attestation	11
2.9 Code Guards as an Active Runtime Defense	12
2.9.1 Security Requirements for Code Integrity	12
2.9.2 Functional Structure of Code Guards	13
2.9.3 Detector Placement, Timing, and Precision	14
2.9.4 Delayed Detection and Reaction	16
2.10 ELF Binaries and Runtime Addressing	16
2.11 Pin and Pintools	18
2.12 Data Dependencies	19
2.13 Dynamic Slicing	19
2.14 Differential Dynamic Analysis	20
2.15 Statistical Attack Effort Estimation	21

3	Related Work	24
3.1	Evaluation of MATE Software Protections	24
3.2	Integrity Checking, Code Guards, Self-Checksumming, and Attestation . .	25
3.3	Empirical and Reproducible Studies of Reverse-Engineering Attacks	26
3.4	Program-Analysis Techniques for Attack Modeling	27
3.5	Ranking-Based Inspection and Effort Estimation	28
4	Problem Formulation and Design	30
4.1	Problem Statement	30
4.2	Protected Binary Model	31
4.3	Code Guard Structure	31
4.4	Golden Values and Guard Metadata	32
4.5	Interleaving of Guard-Phase Instructions	33
4.6	Defense Variant Families	34
4.7	Chained Code Guards	36
4.8	Attack Workflow Design	37
4.8.1	Inspection Target	39
4.8.2	Code-Reading Instructions as Analysis Seeds	39
4.8.3	Dynamic Dependency Model	40
4.8.4	Attack 1: Single-Execution Localization	41
4.8.5	Attack 2: Differential Localization	41
4.8.6	External-Channel Limitation of the Standard Dependency Model .	42
4.8.7	Attack 3: External-Channel-Aware Differential Localization	42
4.8.8	Candidate Generation	43
4.8.9	Ranking Interface	43
5	Heuristic Metrics for Attacker-Effort Estimation	45
5.1	Candidate Ranking Model	46
5.2	Opcode Score	46
5.3	Divergence Score	47
5.4	Depth Score	48
5.5	Outgoing Dependency Score	49
5.6	Hit-Count Score	50
5.7	Path Centrality	51
5.8	Tuple Diversity Score	51
5.9	Failure-Observed Score	52
5.10	Code-Reader Multiplier	53
5.11	From Candidate Ranking to Attacker Effort	55
5.12	Design Assumptions and Limitations	56

6	Implementation	58
6.1	Build Environment and Compilation Settings	59
6.2	Implemented Components	59
6.3	Protected Sample Generation	60
6.3.1	Intermediate Executable and Placeholder Metadata	60
6.3.2	Offline Patching	60
6.3.3	Final Protected Samples	61
6.4	Guard Metadata and Golden Values	62
6.4.1	Blob Layout	62
6.4.2	Checksum Seeds and Slot Population	62
6.4.3	Runtime Blob Location	63
6.5	Runtime Code Guard Implementation	63
6.5.1	Attestation Checksum Routines	64
6.6	Implemented Defense Variants	65
6.7	Chained and Syscall-Mediated Guards	65
6.7.1	Normal-to-Master Seed Propagation	65
6.7.2	Syscall-Mediated Dependency Breaking	66
6.8	Instruction-Level Evidence	68
6.8.1	The Code-Reading Seed Instruction	68
6.8.2	Verification-Critical Instructions	68
6.9	Data-Dependency Collection and Attack Workflow Implementation	71
6.9.1	Code-Reading Instruction Collection	71
6.9.2	Baseline Data-Dependency Collection	71
6.9.3	Shortcut Dependencies and Seed Preservation	72
6.9.4	Syscall-Aware Over-Approximation	72
6.9.5	Attack Pipelines	73
6.9.6	Ranking Report Generation	73
6.9.7	Summary of the Attacks	74
6.10	Generated Analysis Artefacts	75
6.10.1	CodeReader and Seed Artefacts	75
6.10.2	Dependency and Differential Artefacts	75
6.10.3	Attack-Specific and Ranking Artefacts	75
6.11	Evaluation Targets and Workflow Scope	76
6.11.1	Controlled Tampering Points	77
6.11.2	Ground-Truth Representation	77
6.11.3	Implementation Limitations	78
7	Evaluation and Validation	79
7.1	Evaluation Objectives and Assumptions	79

7.2	Evaluation Coverage	80
7.3	Ranking Measures and Evaluation Results	80
7.3.1	Baseline Direct-Verification	81
7.3.2	Comparison-Obfuscated	82
7.3.3	Function-Diversified	84
7.3.4	Chained	84
7.3.5	Syscall-Mediated Chained	86
7.3.6	State-Breaking Syscall-Mediated	87
7.4	Normal-to-Master Seed Dependency	88
7.5	Representative Ranking-Based Target Validation	91
7.5.1	Representative ranked candidates	92
7.5.2	Static and runtime inspection of the ranked candidates	92
7.6	Validation of Syscall-Aware Reachability	95
8	Conclusion and Future Works	97
8.1	Conclusions	97
8.2	Future Work	98
	Bibliography	99

List of Figures

2.1	Example of expression-equivalence obfuscation	5
2.2	A taxonomy of obfuscation techniques.	6
2.3	General MATE setting	7
2.4	Attestation, verification, and reaction flow.	14
2.5	Detector precision timeline	14
2.6	ELF sections, segments, and runtime mapping	17
2.7	Differential dynamic analysis.	21
4.1	Conceptual phases of a code guard.	31
4.2	Conceptual relation between a normal guard and its paired master guard. .	37
4.3	Attack workflow schemes.	38
6.1	Two-phase generation process for a protected sample.	61
6.2	Comparison of the source-level and compiled data paths used by the two syscall-mediated variants.	67
7.1	Data propagation from the normal guard to the master guard in the imple- mented chained form.	90
7.2	Alternative Data propagation from the normal guard to the master guard in the implemented chained form.	91

Chapter 1

Introduction

Many software protections are deployed in a setting where the protected program is executed on a machine outside the control of its developer. This is common for desktop applications, games, license managers, and other client-side programs. In this setting, the user of the machine can also be the attacker. The attacker can execute the program, inspect its binary code, attach debugging tools, record executions, and modify instructions or data. This setting is commonly referred to as the Man-At-The-End (MATE) threat model.

In the MATE setting, software protections are not expected to provide absolute security. An attacker with enough time, tools, and knowledge can usually make progress. The practical goal of a protection is therefore to increase the work needed to understand or modify the program. This is the case for obfuscation, tamper detection, code guards, software diversity, and related techniques. These mechanisms aim to delay the attack, make the analysis less convenient, or reduce the possibility of reusing the same attack on many protected instances [1, 2].

Measuring how well a protection achieves this is not straightforward. A protected binary can be described through structural measures, such as its size, the complexity of its control flow, or the number of dependencies it contains. These measures capture how the program changed, but not how much harder the attack became, since a transformation can inflate them without touching the part that the attacker needs to read, while a small change can block the step on which the attacker depends. A protection is therefore better judged by looking at the attack itself.

Recent work has proposed to model reverse-engineering attacks as repeatable procedures and to estimate from them the effort that different attack strategies require [3, 4]. Running the same procedure many times gives a way to compare protected versions of a program under assumptions that are stated explicitly.

So far, this approach has been applied to only a few cases. A model built for one kind of attack does not transfer to another on its own, because the information available to the attacker and the meaning of progress change from one case to the next. The approach therefore still has to be tried on other protections, other attacker goals, and other protected assets, such as the code and data found in games, DRM systems, or license managers.

This thesis works on one of these open cases. It applies attack-effort modelling to code guards, a protection in which the protected asset is executable code. A code guard is a runtime check against tampering. It reads a region of the program's own code, computes an integrity value over it, and compares the result against a reference value that the developer stored in the binary. When the two values do not match, the program can stop, change its behaviour, or react in another way [5, 6, 7]. The attacks model an attacker whose intermediate goal is to locate the instructions involved in computing and verifying the integrity check, as a step toward understanding or disabling the protection logic. Code guards are a good case study because this task is concrete, and because its difficulty depends on how the guard is built.

1.1 Objectives

The goal of this thesis is to study whether attack-effort modelling can be applied to binaries protected with code guards, and what such a model can show about different guard designs. To do this, the work reproduces the attacker's task in a controlled setting and measures the amount of inspection required to reach the guard logic. The goal is divided into three objectives.

The first objective is to produce several protected versions of the same program, each one using a different code-guard configuration. These configurations range from designs where the guard logic is direct and easy to follow to designs where it is spread out and harder to relate to its effect. Because every version protects the same program, the attacks can be compared under controlled conditions.

The second objective is to implement repeatable attack workflows that analyse these binaries and return, for each one, a set of candidate instructions that may belong to the guard. The workflows represent different attacker strategies, from a simple one based on a single execution to more informed ones.

The third objective is to derive a measure of attacker effort from these workflows. Each workflow orders its candidate instructions in the sequence an attacker would inspect them, and the position of the verification-relevant instruction in that order is used as the proxy for effort.

1.2 Major Outcomes

This thesis reports three main outcomes.

The first outcome is that attack-effort modelling can be instantiated for a code-guard scenario. The work defines a repeatable setting in which several protected versions of the same program are analysed by scripted attacks under the same evaluation assumptions.

The second outcome is that this setting captures the effect of the guard design on the modelled attack effort. It shows how different ways of building the guard change the amount of candidate inspection that an attack requires before reaching the verification logic.

The third outcome is a comparison of the three attack strategies on the same protected variants. The comparison shows which strategies remain effective as the guard design becomes more complex, and where a more informed analysis is needed to locate the verification logic.

These outcomes do not provide an absolute measure of security. They give a repeatable way to compare guard designs and attack strategies under the same assumptions.

1.3 Structure of the Thesis

The rest of the thesis is organised as follows. Chapter 2 gives the background needed for the rest of the work, including the MATE model, code guards, and the analysis of binaries and of their execution. Chapter 3 reviews related work on the evaluation of software protections, on code guards, and on effort modelling for reverse-engineering attacks. Chapter 4 presents the design of the protected program, the code-guard variants, and the attack workflows used to build candidate instruction sets. Chapter 5 defines the heuristic metrics used to rank these candidates and shows how the resulting rank is interpreted as an effort proxy. Chapter 6 describes how the design was implemented. Chapter 7 reports the experiments and also validates the interpretation of the rankings they produce. Chapter 8 concludes the thesis and outlines possible future work.

Chapter 2

Background

This chapter introduces the background needed to understand the protection setting, the attacker model, and the analysis workflow used in this thesis. It starts from software protection, tamper-proofing, obfuscation, and the Man-At-The-End threat model. These concepts define the setting in which the protected binary is deployed and inspected.

The chapter then presents code guards as runtime mechanisms for code integrity. It discusses what they protect, how they are structured, and which constraints affect their placement and response. The last part introduces the technical concepts used by the attack workflows, including ELF binaries, dynamic binary instrumentation with Pin, data dependencies, dynamic slicing, differential dynamic analysis, and statistical attack-effort estimation.

2.1 Software Protection and Tamper-Proofing

Software Protection refers to a range of techniques used to make unauthorized use, analysis, or modification of software more difficult [8, 2]. This area includes different goals, such as protecting intellectual property, enforcing licensing policies, limiting reverse engineering, and preserving program integrity [9]. This thesis focuses on the last aspect, namely software tamper-proofing.

Software tamper-proofing is a research area in computer science that integrates techniques from cryptography, reverse engineering, and algorithm design. The goal is to preserve a program’s integrity against adversarial manipulation when an adversary can interact with the executable and attempt to modify it. Specifically, tamper-proofing mechanisms aim to prevent changes that would alter the program’s intended semantics, or to detect such changes and react with a predefined response [9, 10]. Formally, given a program P and its intended behaviour, a tamper-proofing mechanism aims to let P run according to that behaviour under normal conditions, while producing a detection outcome and a defined countermeasure when tampering occurs. This view is important for the rest of the thesis, because the evaluated protections are not passive transformations of the

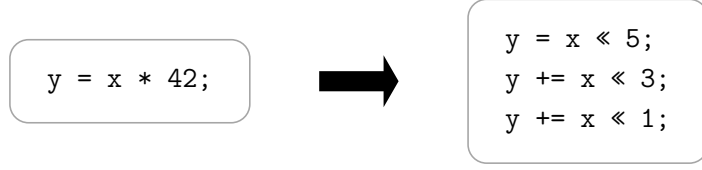


Figure 2.1: Example of expression-equivalence obfuscation.

binary, but runtime mechanisms that check whether selected parts of the program have been altered.

2.2 Tamper-Proofing and Obfuscation

It is essential to distinguish tamper-proofing from obfuscation, since the two techniques address different protection goals [9, 1, 10].

- **Tamper-proofing** focuses on *integrity*: it aims to keep the program’s execution correct and to provide mechanisms that can detect unauthorized changes or make them harder to apply.
- **Obfuscation** focuses on *confidentiality*: it aims to limit what an attacker can learn about a program by making reverse engineering and analysis more difficult, for instance by changing the representation of code and data.

Although the two approaches are distinct, they are often combined in practice. Obfuscation increases the difficulty of locating sensitive logic, while tamper-proofing aims to ensure that modifications to that logic are either prevented, detected, or made costly to exploit [9, 11].

2.3 Obfuscation as a Complementary Protection

Obfuscation is a software protection technique that transforms a program into a different representation while preserving its behaviour. Informally, a program P is transformed into an obfuscated program P' that can still be executed, but from which it is harder to extract useful information. The information to protect depends on the attacker’s goal. It may be the structure of the program, the location of a sensitive check, the value of embedded data, or the logic used to compute a relevant result [9, 1]. Figure 2.1 shows a simple example of this idea, where one expression is replaced by an equivalent sequence of operations that is less direct to recognize.

In this thesis, obfuscation is relevant mainly as a complement to tamper-proofing. Its role is to obscure the structure of the protection logic, for example by mixing guard code with transformed control flow, unrelated computations, or non-obvious data dependencies. This matters because a code guard is easier to analyse when its reading, checking, and response logic appear as a compact and recognizable block.

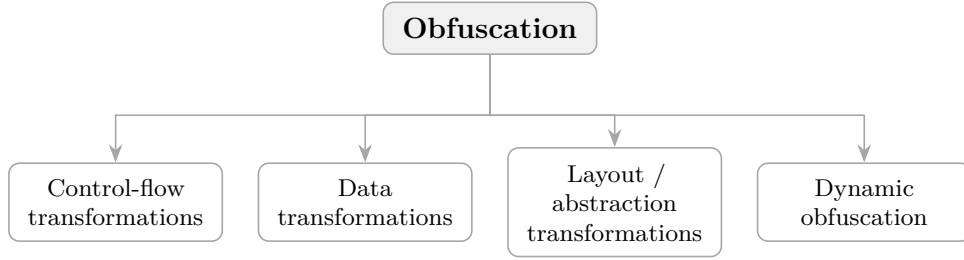


Figure 2.2: A taxonomy of obfuscation techniques.

Obfuscation techniques are commonly classified according to the part of the program representation they transform. Classical taxonomies distinguish, among others, control-flow transformations, data transformations, and layout or abstraction transformations [12, 9]. Later surveys also discuss more advanced and dynamic forms of obfuscation, where the protected representation or behaviour may change during execution [1]. This classification is summarized in Figure 2.2.

Control-flow obfuscation changes the structure of branches, loops, calls, and jumps to make the execution logic harder to follow. Data obfuscation changes how values are represented, stored, or computed, so that the role of relevant variables or constants is less clear. Layout and abstraction transformations modify the organization of the program, for example by splitting, merging, reordering, or renaming program elements. Dynamic obfuscation instead refers to designs in which parts of the program representation or execution behaviour are not fixed once and for all at deployment time, but may change during execution.

These transformations have different effects and costs. Obfuscation is commonly evaluated in terms of *potency*, *resilience*, *stealth*, and *cost* [12, 13, 9, 1]. These criteria provide a more precise vocabulary for discussing the effect of an obfuscating transformation than generic statements about whether a protection is easy to identify or attack.

In this thesis, obfuscation is not analysed as a separate protection mechanism detached from code guards. It is considered through the way the protected samples are built. The samples differ in the structure of their code guards, and some configurations make the protection logic more distributed, less localized, or harder to relate to a single failure point. For this reason, obfuscation remains relevant to the evaluation, because it affects how the attacker observes, follows, and ranks the code involved in the protection.

2.4 The MATE Threat Model

Software protection is often studied under the MATE threat model [8, 2]. In this model, the adversary is the end user and controls the client-side execution environment. Figure 2.3 summarizes this assumption. This includes access to the executable file, the operating system, the runtime state of the process, and the tools needed to inspect or modify the program.

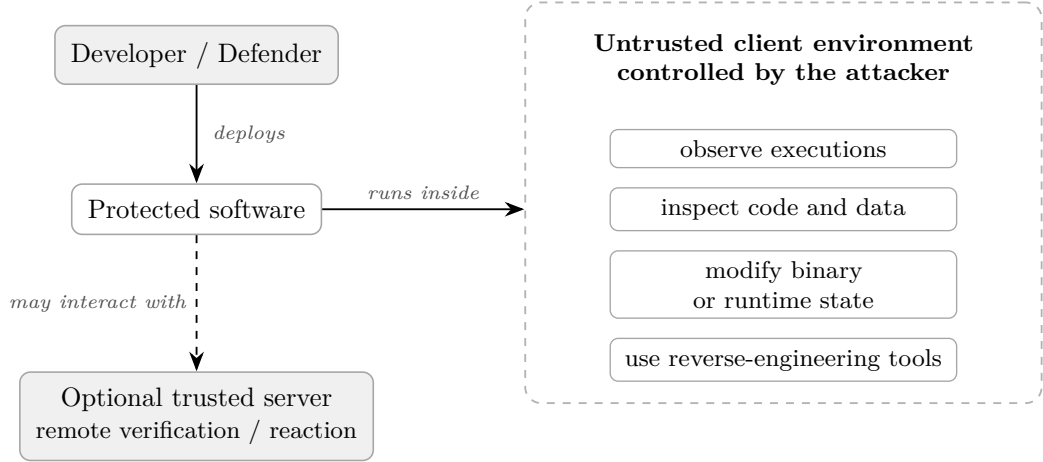


Figure 2.3: General MATE setting. The client-side environment is controlled by the attacker, while optional remote services remain under defender control.

This assumption is relevant for client-side software because the defender cannot rely on physical or administrative control over the target platform. The attacker may therefore combine static analysis of the binary with dynamic observation of its execution. This motivates the analysis model used later in the thesis, where executions are inspected, recorded, and compared to reason about the protection logic.

2.4.1 Integrity Assets in a MATE Setting

In a MATE setting, the protected assets can include code, data, control flow, configuration values, and program behaviour [10, 2]. At a general level, these assets can include both data and program behaviour. Data assets are values or resources produced, stored, or processed by the application, such as inputs, configuration files, internal state, or database content. Behavioural assets are the effects produced by the execution of the program, such as enforcing a license check, refusing an invalid input, or producing only authorized outputs.

Code integrity is central to tamper protection because local code changes can alter the intended behaviour of the program, for example by removing checks, redirecting branches, or modifying protection logic [9, 10]. In these cases, the immediate target is the code, but the final effect is behavioural. The program performs an action that should not be allowed, or fails to perform an action that should enforce the protection.

This connection between code integrity and behavioural integrity motivates the use of code guards in the following sections. By checking whether selected code regions have been modified, a code guard aims to detect changes that could alter the protected behaviour of the program.

2.4.2 Attacker Capabilities and Analysis Techniques

In a MATE setting, the attacker can use both static and dynamic analysis techniques to study the protected program [8, 2]. Static analysis allows the attacker to inspect the binary without executing it, while dynamic analysis observes concrete executions. In practice, the two views are often combined: static analysis gives a structural view of the binary, and dynamic analysis shows which parts of that structure are reached at runtime.

The main analysis views relevant to this setting are static analysis and dynamic analysis.

- **Static analysis.** Static analysis inspects the program without executing it. Disassemblers and decompilers translate the executable into assembly code or a higher-level representation. They allow the attacker to inspect instructions, functions, constants, control-flow structures, and references to data. Tools such as Ghidra and IDA Pro provide disassembly, decompilation, graph-based views, and scripting support for binary inspection [14, 15].
- **Dynamic analysis.** Dynamic analysis observes concrete executions of the program. Debuggers, emulators, tracers, and dynamic binary instrumentation frameworks can be used to inspect registers and memory, follow executed instructions, monitor system calls, and compare executions under different inputs or modifications. Debuggers such as GDB support interactive inspection of runtime state, while frameworks such as Pin, Valgrind, and DynamoRIO support custom runtime analyses [16, 17, 18, 19].

Several program-analysis techniques can be instantiated on top of these views. Dynamic taint analysis tracks how selected values propagate during execution, symbolic or concolic execution reasons about paths and input conditions, and program slicing reduces the inspected code to statements or instructions related to a selected criterion [20, 21, 22, 23, 24]. These are techniques that can support static, dynamic, or hybrid reverse-engineering workflows.

These techniques do not guarantee that the attacker immediately understands the protected program. They define a realistic analysis environment in which the attacker can inspect code, observe runtime behaviour, and test hypotheses about the protection logic. This is the setting assumed later in the thesis, where dynamic instrumentation, execution traces, data dependencies, slicing, and differential analysis are used to build and rank candidate instructions.

2.5 Practical Limits of Client-Side Protection

Client-side software protection operates in an environment where the defender does not control the target platform. Once the program is deployed, the attacker can access the executable, observe its execution, and modify parts of the software stack [8, 2, 25]. This differs from server-side settings, where the defender can usually enforce stronger administrative and physical control over the system.

For this reason, client-side protections are commonly interpreted as mechanisms that increase attacker effort rather than as mechanisms that provide absolute or permanent

security [9, 26, 3]. A determined attacker may spend time analysing the binary, repeat executions, use reverse-engineering tools, and build custom scripts or analysis components. Given enough time and resources, the protection logic may eventually be understood and modified.

This practical limitation is also reflected in formal work on tamper protection. Dedić et al. propose a graph-game model in which the attacker tries to learn the transformation inserted into the program, and the cost of the attack is estimated through the steps required in that model [26]. The result is model-dependent, but it captures a central constraint of client-side protection: the defender cannot assume that the protected program remains opaque to the attacker indefinitely.

2.6 Tamper-Proofing as Attack-Cost Increase

The limits discussed in Section 2.5 change how client-side protections should be evaluated. Since the attacker can eventually inspect the protected program, the relevant question is not whether the protection is impossible to bypass in principle. The relevant question is how much work the attacker must perform before reaching the information needed for a useful bypass [26, 25, 3].

For this reason, a tamper-proofing mechanism should be evaluated not only by looking at the transformation applied to the program, but also by considering the analysis work required from the attacker. This view is consistent with evaluation approaches that model software protection in terms of attacker effort, attack steps, or risk reduction rather than as an absolute security property [26, 27, 25, 3]. A protection may still be useful when it forces the attacker to inspect more code, recover more relations, or test more hypotheses before applying a useful modification.

This view is important for the rest of the thesis. The proposed evaluation does not treat a code guard as secure only because it cannot be bypassed. Instead, it studies how much inspection effort is needed before the relevant protection logic is reached.

2.7 Desired Properties of a Tamper-Proofing Mechanism

Chang and Atallah discuss several properties that a tamper-proofing mechanism should provide to remain useful under adversarial analysis [5]. These properties are useful as a reference model for the design of software protections. At the same time, they cannot be treated as requirements that are always fully achievable in practice. Real implementations must deal with runtime overhead, code size, architectural constraints, and the way the protection is integrated into the target program.

The most relevant properties for this thesis are the following.

- **Resilience.** Resilience describes the ability of a protection mechanism to remain effective even when the attacker finds or modifies part of it [5, 9, 28]. A resilient

design should avoid single points of failure. If removing one check is enough to disable the whole protection, the mechanism is easy to attack once that check has been identified.

A common way to improve resilience is to distribute integrity checks across different parts of the program and to introduce *redundancy*. Multiple checks can protect different code regions, or different checks can contribute to the same protection goal. This forces the attacker to locate and modify several elements instead of applying one local patch. Resilience can also be supported by *stealth*, especially when recognizable or isolated checks help the attacker locate and target the protection logic. However, stealth is not equally important in every protection scenario. Its relevance depends on the attack model, on the role of the protection, and on whether the attacker must identify the protection logic before bypassing it. *Obfuscation* can support resilience by making the protection logic less explicit, although obfuscation alone does not provide tamper-proofing.

In practice, resilience also has an implementation cost. More checks and more redundancy can increase code size and runtime overhead. The design therefore has to balance the number of checks, their position, and the acceptable performance cost.

- **White-box security.** In a MATE setting, the attacker may inspect the binary and observe the execution of the program [8, 2]. The protection should therefore not rely only on the secrecy of its implementation. This follows the same intuition behind Kerckhoffs' principle: the general design of a protection mechanism may become known, so the mechanism should not fail only because the attacker understands how it is structured.

In software protection, this requirement is difficult to satisfy completely because the protected program, the protection logic, and any local reference values are all deployed on a machine controlled by the attacker. For this reason, white-box security should be interpreted as a design goal rather than as a guarantee. The protection should avoid relying on a single obvious secret or on a single check whose removal disables the whole mechanism.

- **Self-defense.** Tamper-proofing is not only a static transformation applied to the binary. It is an active protection strategy based on checks and reactions executed by the program itself. A self-defending mechanism should be able to inspect relevant parts of the program during execution and react when an integrity violation is detected [6, 9, 7].

In the context of code integrity, this usually means computing or verifying integrity information over selected code regions. For example, a protection may compute a value over executable bytes and compare the result with an expected one. If the comparison fails, the program can terminate, alter its behaviour, or trigger another response. This property is important because passive techniques only make analysis harder, while tamper-proofing also tries to detect and respond to modifications.

- **Configurability.** Configurability refers to the possibility of adapting the protection to the program being protected. A configurable mechanism should allow the defender

to choose how many checks are inserted, which regions are protected, where the checks are placed, and how detection is connected to response.

This property is closely related to *resilience*. If checks are always inserted in predictable positions, or if they always follow the same structure, the attacker can learn the pattern and remove them more easily. A configurable design allows checks to be spread across the program and combined in different ways. This can make the protection less regular and reduce the risk that one attack strategy works against all protected instances [25, 28].

Configurability also makes it possible to tune the trade-off between protection strength and overhead. A small program or a performance-critical component may only tolerate a limited number of checks. A more sensitive component may justify stronger protection, more redundancy, or a more complex connection between checking and response.

These properties clarify why tamper-proofing should not be treated as a static feature that is added once and then considered complete. A tamper-proofing mechanism is part of the program execution. It checks selected conditions, propagates the result of those checks, and reacts when tampering is detected.

This observation also explains why protection strategies based only on hiding code are not sufficient in a MATE setting. The attacker can observe the program while it runs, and even hidden logic must eventually execute. For this reason, effective tamper-proofing requires active checks placed inside the execution flow. Code guards are a concrete mechanism in this direction. They check the integrity of selected code regions and connect the result of those checks to a response.

2.8 Local and Remote Attestation

The previous sections described tamper-proofing as a runtime mechanism that detects integrity violations and triggers a reaction. This mechanism can be organized locally or remotely. In a local scheme, both the component that collects integrity evidence and the component that verifies it are deployed inside the protected program. Code guards and self-checking routines are typical examples of this organization [5, 9, 29]. This makes the protection usable without relying on a network service, but it also means that the verifier, the reaction logic, and any local reference values are exposed to an attacker who controls the client.

Remote attestation changes this organization. The protected program still contains an attestation component that computes or collects integrity evidence, but the evidence is sent to a verifier running on a trusted remote server. The server can compare the received evidence against reference information that is not stored in the client, such as expected measurements or hash values. This allows useful verification data to remain outside the attacker-controlled environment [29, 30].

Remote attestation is usually based on a challenge-response structure. A remote party requests an integrity measurement, often using a nonce to prevent replay, and the protected

program computes evidence that is sent back for verification. The verifier then decides whether the client should be considered valid [29, 30]. Remote attestation can also be combined with server-side reaction: if the client fails attestation, the server can refuse to provide a required service or a required part of the computation [29].

This thesis focuses on local code guards. This focus should be understood as a scope restriction within a broader design space, not as a general limitation of code-guard-based protection. In software protection, local code guards, self-checking schemes, remote attestation, and server-side reactions represent different ways of organizing integrity checking, verification, and reaction.

2.9 Code Guards as an Active Runtime Defense

Code guards are a class of runtime tamper-proofing mechanisms used to protect code integrity. Their purpose is to check whether selected executable regions have been modified and to trigger a response when tampering is detected. Unlike passive transformations, code guards execute checks at runtime and connect the result of those checks to a program response [5, 6, 9, 7]. For this reason, they are a concrete example of the self-defense and resilience properties discussed in the previous section.

2.9.1 Security Requirements for Code Integrity

The primary security requirement addressed by a code guard is code integrity. In this context, the goal is to protect executable code against both static and dynamic tampering.

Static tampering refers to modifications applied to the program binary or to linked libraries before execution. Typical examples include changes performed with binary editing tools, patching utilities, or manual rewriting of instructions in the executable image. Dynamic tampering, instead, takes place while the program is running. In this case, the attacker may use debugging facilities or runtime instrumentation to alter the executed code, redirect control flow, or interfere with the intended behaviour of the program.

The distinction between static and dynamic tampering is standard in software tamper-resistance, because an attacker may modify either the stored executable or the running process state [9, 10]. For example, some debugging techniques introduce temporary instruction changes, such as software breakpoints, by replacing original instructions with trap instructions [16]. From the perspective of code integrity, these modifications are still relevant, since they alter the code being executed and may reveal or bypass the protection mechanism itself.

Code guards are related to self-checking and self-checksumming protections, but they are not the same concept. Self-checking is the broader notion. The program contains runtime checks on its own code, data, or execution environment [9, 29]. Self-checksumming is one form of self-checking. In this case, the program computes a checksum or hash over selected code regions and uses the result as evidence that the code has not been modified [6, 7].

Code guards describe one way to organize these checks inside the program. In the work of Chang and Atallah, the protection is not based on a single checker, but on smaller guards distributed across the program. A guard can check a program region, and guards can also protect each other through mutual protection [5]. For this reason, a code guard may use self-checksumming as its checking mechanism, but the notion of a guard also depends on where the check is placed, how it is connected to other guards, and which reaction follows a failed check.

2.9.2 Functional Structure of Code Guards

This section uses a four-stage decomposition to describe code guards at an abstract level. The decomposition is not meant as a new protection model, but as a convenient way to organize checking and reaction mechanisms described in work on guards, self-checking, self-checksumming, and reactive attestation [5, 6, 9, 29, 7]. The four stages are initialization, code evaluation or attestation, verification, and reaction.

- **Initialization.** This stage prepares the information needed by the guard. It defines which code region has to be checked, which checking routine is used, how the result of the check is represented, and where the reference information needed for verification is kept. In a purely local self-checking design, this reference information is available inside the protected program. In a remote-attestation design, part of this reference information, such as expected measurements or hash values, can instead be kept by a trusted verifier outside the attacker-controlled client [29, 30].
- **Code evaluation / Attestation.** This stage evaluates the integrity of a selected code region. In code-integrity protections, this often means computing a value over executable bytes. Depending on the design, the value may be obtained through a checksum, a cryptographic hash, or another verification procedure. Other designs may also consider runtime indicators, such as timing anomalies, when the goal is to detect debugging or instrumentation [6, 9, 7].
- **Verification.** The value produced during attestation is checked against reference information. In local guards, this comparison is performed inside the protected program. In remote attestation, the protected program sends integrity evidence to a remote verifier, which evaluates the evidence outside the attacker-controlled environment [29]. If the evidence is not accepted, the guard has detected a violation of the expected integrity state.
- **Reaction.** This stage depends on the outcome of verification. If verification succeeds, the program continues its normal execution. If verification fails, the protection triggers a reaction. Local reactions may terminate the program, degrade functionality, corrupt later results, or repair modified code. Remote reactions may refuse to provide a required service or stop executing the server-side part of a split computation [5, 29, 9].

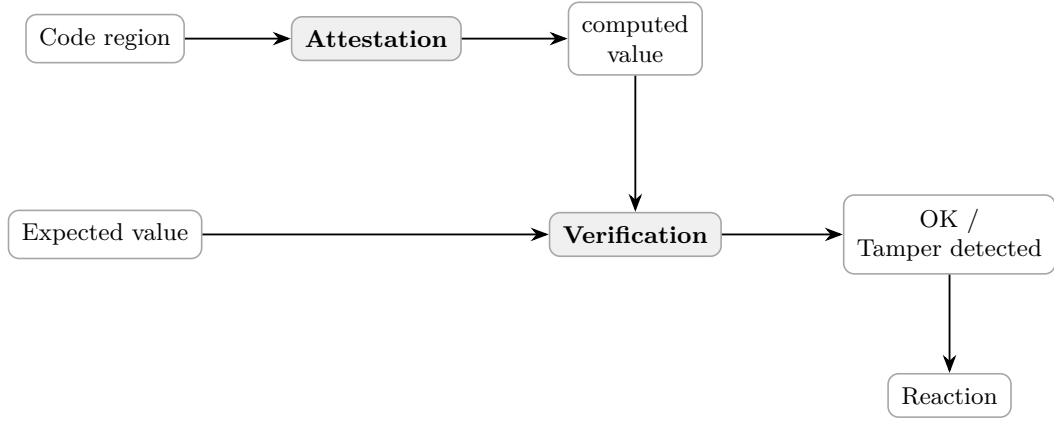


Figure 2.4: Attestation, verification, and reaction flow.

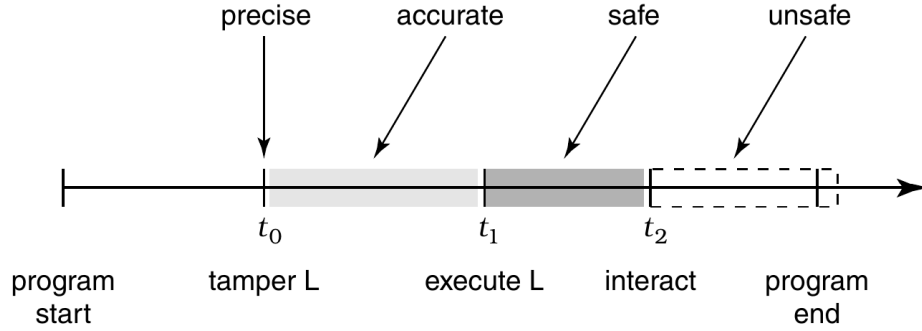


Figure 2.5: Detector precision timeline, taken from the detector-precision model described by Collberg and Nagra [9].

This structure in Figure 2.4 clarifies that a code guard is not only a detector. It is a complete runtime mechanism in which the detection of tampering is connected to a response.

2.9.3 Detector Placement, Timing, and Precision

The effectiveness of a code guard depends not only on what it checks, but also on where and when the check is executed. Checks that are placed in predictable locations, or that follow a recognizable structure, can become easier for an attacker to identify and disable. For this reason, tamper-proofing mechanisms often try to distribute checking and reaction logic across the program and to integrate it with the normal execution flow [9, 5].

The timing terminology used in this section follows the detector-precision model described by Collberg and Nagra [9]. In that model, the usefulness of a detector is related to the point at which tampering is detected with respect to the execution of the modified code and to the first externally visible effect of the tampered execution.

Figure 2.5 summarizes this timing model. At time (t_0), the attacker modifies a location (L) in a program (P). This may happen before execution, for example by editing the executable file, during program start-up, for example through loader-level manipulation, or during execution, for example through a debugger or another runtime manipulation tool. At time (t_1), the program reaches (L), and the modified code is executed for the first time. At time (t_2), the program performs the first interaction event after (t_0). An interaction event is an operation through which the tampered execution can produce an externally visible or persistent effect, such as writing to a file, communicating over the network, or modifying persistent state.

Within this model, detector precision can be described as follows.

Precise detection. Tampering is detected close to t_0 , within a short time after the modification.

Accurate detection. Tampering is detected after t_0 , but before t_1 . In this case, the modified code can still be prevented from executing.

Safe detection. Tampering is detected after t_1 , but before t_2 . The modified code has already executed, and the program may already be in an unintended state. However, the detector can still prevent the first interaction event from taking place.

Unsafe detection. Tampering is detected at or after t_2 , or after program termination. At that point, the tampered execution may already have produced an externally visible or persistent effect, so the remaining reaction may be limited to reporting, logging, invalidating a result, or applying a later penalty.

The term *unsafe* is therefore relative to this detector-precision model. It does not mean that the detector is useless or that the protection is invalid. It means that detection occurs after the first interaction event, so the tampered execution may already have produced an externally visible or persistent effect. Whether this is acceptable depends on the protection goal and on the application domain. If the goal is to prevent the first protected output from being produced, unsafe detection is too late. If the goal is to identify abuse, revoke access, invalidate a result, punish the attacker, or prevent continued use, detection after the first interaction event can still be effective. For example, in an online game, detecting a cheater a few seconds after the cheat was first used may still be acceptable if the server can invalidate the match result, disconnect the client, or apply an account-level penalty.

Another design choice concerns how often the check is executed. A static detector runs once, for example at load time. This can detect modifications already present in the executable image, but it cannot detect changes introduced later during execution. A dynamic detector runs one or more checks while the program executes. This is needed when the attacker may modify code after the program has started.

The frequency of these checks introduces a trade-off. Running checks more often can reduce the time window available to the attacker, but it can also increase runtime overhead. Frequent or regular checks may also become easier to observe and target during analysis. For this reason, placement, timing, and frequency have to be considered together.

2.9.4 Delayed Detection and Reaction

A code guard does not always react immediately at the same point where verification fails. An immediate reaction, such as terminating the process or refusing to continue execution, can reduce the attacker’s window of action. However, it can also give the attacker a clear signal that a specific check has been triggered, which may help in locating the protection logic.

For this reason, tamper-proofing designs may separate checking, verification, and reaction across different points of the execution [9, 29, 11]. The result of a failed verification can be propagated through intermediate state, such as flags, variables, or auxiliary data structures. The reaction is then triggered later, when the connection with the original check is less visible.

Delayed reactions can take different forms. The program may terminate later, return corrupted outputs, degrade functionality, report the violation, or fail only after further computation. The purpose is not only to react to tampering, but also to make the relation between detection and reaction harder to reconstruct.

A delayed reaction is therefore not automatically weaker than an immediate one. It is weaker only with respect to protection goals that require preventing the first externally visible effect of tampering. In other scenarios, delayed reaction can be a deliberate design choice. It may reduce the information given to the attacker, make the causal relation between detection and reaction harder to reconstruct, or allow the defender to react at a more effective point. For example, a local reaction may corrupt later results instead of terminating immediately, and a remote reaction may disconnect a client, refuse a future service request, revoke access, or invalidate a result after the server has received sufficient evidence of tampering [9, 29].

The relevant criterion is therefore not whether the reaction is immediate, but whether the delay is compatible with the security objective. If the objective is to prevent any protected output, the reaction must occur before that output is released. If the objective is deterrence, abuse detection, result invalidation, or continued-service prevention, a later reaction can still be valid.

2.10 ELF Binaries and Runtime Addressing

The Executable and Linkable Format, commonly referred to as ELF, is the standard object file format used by many Unix-like systems for executables, relocatable object files, shared libraries, and core dumps [31, 32]. Since this thesis works directly on compiled binaries, it is useful to introduce the main elements of the ELF format before discussing the implementation of the proposed protection mechanism and the analysis performed on protected binaries.

An ELF file starts with an ELF header. This header identifies the file as an ELF object and contains, among other fields, the offsets of the program header table and the section header table. These two tables describe the same binary from two different perspectives. The section header table describes the organization of the file as seen by the linker and

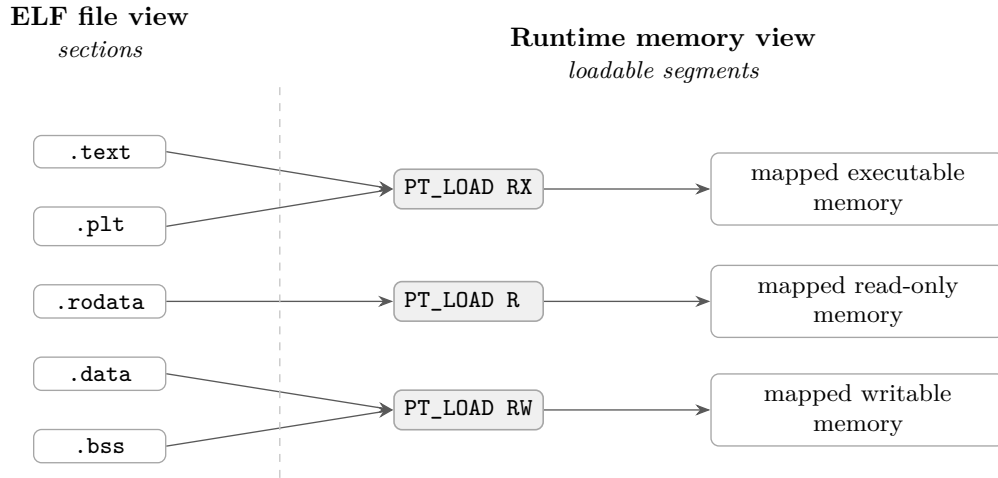


Figure 2.6: Simplified relationship between ELF sections, loadable segments, and runtime memory mappings.

by binary-analysis tools. Sections such as `.text`, `.rodata`, `.data`, `.bss`, and `.plt` are examples of this view. They are used to organize executable code, read-only data, writable data, uninitialized data, dynamic-linking stubs, symbols, relocation information, and other information produced during compilation and linking [33]. This view is also useful for static inspection, since tools such as `readelf` and `objdump` can be used to inspect ELF headers, sections, symbols, relocations, and disassembled code [34].

Figure 2.6 summarizes the relation between ELF sections, loadable segments, and runtime memory mappings.

The program header table has a different role. It describes how the operating system loader, or the dynamic linker, has to create the process image in memory. A program header entry describes a segment, and a segment can contain one or more sections. For executable files and shared objects, this view is the one used at load time, because it specifies which parts of the file have to be mapped into memory and which permissions have to be assigned to them [35]. Among the program header entries, entries of type `PT_LOAD` are particularly important for understanding how an executable is mapped into memory. A `PT_LOAD` segment represents a loadable region of the file. Each loadable segment is associated with permission flags, such as `PF_R` for readable memory, `PF_W` for writable memory, and `PF_X` for executable memory [32].

The distinction between sections and segments is important. Sections are mainly a link-time and analysis-time abstraction, while segments are a load-time abstraction. Section names are useful for static analysis and for tools that inspect the binary. At runtime, however, what matters is whether an address belongs to a memory range mapped from a loadable segment with the required permissions.

It should also be noted that an executable loadable segment does not contain only the main body of the `.text` section. Other sections containing executable code may be mapped in the same segment as well. A common example is the procedure linkage table,

`.plt`, which contains stubs used for dynamic linking. From the loader’s point of view, these bytes are part of an executable memory range because they belong to a loadable segment mapped with execute permission. Therefore, the boundary between application code and other executable support code is not always represented by a single section name. The runtime view is given by the program headers and by the permissions assigned to the corresponding loadable segments.

This distinction becomes particularly relevant in the presence of position-independent executables and address space layout randomization. With these mechanisms, an executable or a shared object is not necessarily loaded at the same virtual address in every execution. As a consequence, an address stored or computed as an offset inside the binary cannot always be interpreted as an absolute runtime address. It has to be resolved relative to the base address at which the corresponding object has been loaded. Once the base address is known, an offset referring to a location inside the binary can be translated into a runtime address by adding it to the base address of the loaded module.

2.11 Pin and Pintools

Pin is used in this work to collect information from the execution of the protected program. It is a dynamic binary instrumentation framework, which means that it can insert analysis code into a binary while the binary is running. This does not require access to the original source code. The tools built with Pin are called Pintools [17]. They are written in C or C++ and use the Pin API to decide which parts of the program have to be instrumented.

Pin performs the instrumentation at run time. It uses a just-in-time compiler to translate the executable code and to add the instrumentation requested by the Pintool before the translated code is executed [17]. This allows a Pintool to observe the program at different levels. The instrumentation can be attached to single instructions, traces, routines, or loaded images. When one of these program points is reached, Pin calls an analysis routine defined by the Pintool. This mechanism can be used for simple tasks, such as counting executed instructions, but also for more specific analyses, such as tracing memory accesses or recording values observed during execution.

A Pintool usually separates the code that selects the instrumentation points from the code that performs the analysis. Hill and Feiok describe a Pintool as being composed of a tool component, one or more instrument components, and one or more callback components [36]. The instrument component decides where Pin has to insert the instrumentation. The callback component contains the analysis code that is executed when the instrumented point is reached. This separation is useful in this thesis because the Pintools are not the final goal of the analysis. They are used to collect runtime information from protected and tampered binaries. This information is then used by the later phases of the workflow, for example to reason about executed instructions, memory accesses, and data dependencies.

2.12 Data Dependencies

Data dependencies play an important role in this work because the analysis needs to reason about how a value produced by one instruction can influence later instructions and later program behaviour. A program execution is not only a sequence of executed instructions. It is also a sequence of values being created, stored, moved, and used again. Following these value relations gives a more useful view of the computation than looking only at the control-flow order in which instructions are executed, because program dependencies make explicit how computations are related through data and control relations [37].

Program dependencies are commonly represented through dependence graphs, in which data dependencies capture definition-use relations between program points and control dependencies capture relations imposed by the control structure of the program [37]. In this thesis, the relevant relation is the data-dependence relation where one instruction defines a value, and another instruction later uses that value. At the machine-code level, this relation can be carried through registers, memory locations, or processor state such as flags. For example, an arithmetic instruction may write a result into a register, and a later instruction may read that register. Similarly, a store instruction may write a value to memory, and a later load instruction may read the same value. In both cases, the later instruction depends on the earlier one because part of its computation uses a value produced before.

These relations can be studied statically or dynamically. Static dependencies describe relations that may occur across the possible executions of the program. This view is useful, but it tends to include relations that are possible in the program even if they do not occur in the execution being analysed. Dynamic dependencies are instead defined with respect to one concrete run. They keep only the producer-consumer relations that actually occurred in that execution. This makes the result more precise for the observed run, but also means that the result is specific to that run [22, 23, 24, 21].

The analysis in this thesis uses this dynamic view. The values observed at runtime indicate which instruction produced a value and which later instruction used it during the recorded execution. This makes it possible to build a data dependency graph for that execution. In this graph, each node corresponds to a single instruction of the binary, and the same node is used for that instruction across all of its executions. An edge connects a producer instruction to a consumer instruction when a value flows from the former to the latter during the recorded execution. Because the same instruction can run more than once, for example inside a loop, two instructions can be connected by several edges, one for each observed value flow. This graph is the object that later analyses traverse, and notions such as the distance between two instructions and value propagation are defined over it rather than over the static structure of the binary.

2.13 Dynamic Slicing

Dynamic slicing is relevant because an analysis rarely needs the whole dependency graph. Often the useful part is the part related to a specific value, instruction, or behaviour.

Program slicing was originally introduced by Weiser as a way to decompose a program with respect to a slicing criterion [22]. A static slice contains the statements that may affect the values of selected variables at a selected program point. Since it is computed without fixing one specific execution, it has to account for all behaviours that may occur.

Dynamic slicing restricts this idea to one concrete execution. Korel and Laski define a dynamic slice on the basis of a computation rather than on the set of all possible executions [23]. Agrawal and Horgan describe the dynamic slice as the set of statements that actually affected the value of a variable occurrence for a given program input [24]. This usually makes the slice smaller than the corresponding static slice, because statements that could have affected the value in another execution are not kept unless they affected it in the observed one.

Slicing can be used in two directions. A backward slice starts from a criterion and collects the program points that influenced it. It answers the question of where a value or behaviour came from. A forward slice starts from a criterion and collects the program points influenced by it. It answers the question of where a value can propagate after the criterion is reached.

This work uses a forward slice. The slicing criterion is a seed instruction selected by the analysis, and the slice contains the instructions influenced by the value produced at that point. The slice is therefore a reduced part of the dynamic dependency graph. It is not meant to prove which instruction is the targeted instruction. Its purpose is to obtain a smaller set of candidate instructions that can be inspected during the later analysis.

2.14 Differential Dynamic Analysis

Dynamic analysis observes the behaviour of a program while it is running. In some cases, one execution only gives a partial view of this behaviour. It may be more useful to run the program under different conditions and compare the observations obtained in each run. This is the basic idea behind differential dynamic analysis. The comparison can involve inputs, program versions, or modified executables. If some values or instructions change only in one of these cases, then they may be related to the difference that was introduced.

This idea is used in different areas of security analysis. In white-box cryptography, differential computation analysis collects execution traces from multiple runs and compares them to recover information about computations that depend on secret data [38]. Differential analysis has also been used to study obfuscated mobile applications. In that setting, the application is executed with different inputs, and the observed behaviour is compared to detect privacy leaks that are hard to find by looking at the code alone [39, 40].

The same principle is useful when analysing protected binaries. A protection may not be easy to identify by inspecting a single execution, especially when the binary contains many instructions that are unrelated to the protection itself. Comparing a clean execution with a tampered execution gives a simple criterion to remove part of this noise. Instructions whose observed values remain the same in both executions are less useful for this localization task.

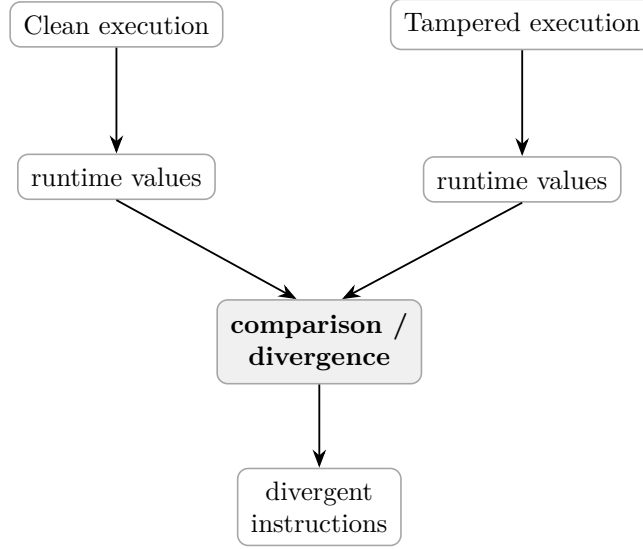


Figure 2.7: Differential dynamic analysis.

Instructions whose values change after tampering are instead more likely to be connected to the protected computation.

Figure 2.7 summarizes the comparison between clean and tampered executions.

In this thesis, this comparison is used to guide the attacker towards a smaller part of the binary. The clean binary and the tampered binary are executed, and runtime information is collected from both executions. The two executions are then compared, and the instructions that show different observed values are kept as divergent points. These points are used as evidence that the tampering affected that part of the computation, and therefore as a criterion to prioritize the following analysis.

2.15 Statistical Attack Effort Estimation

The previous sections motivate an evaluation based on the attack process rather than on program-level complexity alone. Statistical attack-effort estimation provides one way to formalize this idea. Instead of assigning strength directly to a protected binary, it models a reverse-engineering strategy as a sequence of steps and estimates the effort required to execute that strategy [3].

This process-based view also explains why program-level metrics are limited when they are used alone. The effort required by an attack is not a property of the program only. Two attackers working on the same binary may inspect different artefacts, use different analyses, and reach the same target after a different amount of work. A single number computed from the program cannot capture this variation. The alternative used in this thesis is to treat the attack as a process and to estimate the effort from the modelled execution of

that process. Faingnaert et al. proposed a framework for modelling reverse-engineering strategies and estimating the effort needed to execute them [3].

In this framework, an attack is represented as a sequence of steps. During the attack, the model keeps track of the knowledge available to the attacker and of the effort already spent. At each step, the attacker strategy selects the next activity to perform and the program artefacts on which that activity is applied. The activity is then executed in the model, the attacker knowledge is updated, and the effort required for the step is added to the total effort. The general structure can be written as follows:

```
while attack not finished:
    activity, artefacts = strategy(knowledge, totalEffort)
    knowledge, stepEffort = execute(activity, artefacts, knowledge)
    totalEffort += stepEffort
```

The function `strategy` models the decisions taken by the attacker. Given the current knowledge and the effort spent so far, it chooses what should be done next, for example which analysis to run or which candidate artefact to inspect. The function `execute` models the result of the selected activity. It returns the knowledge produced by that activity and an estimate of the effort required to perform it. The attack stops when the modelled goal is reached or when the attacker is assumed to stop because the cost has become too high.

The model does not have to be deterministic. Some decisions may depend on uncertainty. For example, if several artefacts have the same priority, the model may choose between them at random. The same applies to the effort assigned to an activity, which can be sampled from a probability distribution rather than fixed to a single value. One execution of the model therefore gives one effort value. By repeatedly executing the same attack model, the researcher obtains a set of effort values. These values can then be used to describe a distribution of the effort required by the attack.

The simulation is not the same as an attack performed by a human reverse engineer. The loop is run by the researcher who builds the model, and this researcher may have access to ground-truth information about the protected program. This information must be handled carefully. It should not guide the decisions of the `strategy` function, because a real attacker would not know the location of the target in advance. Using ground truth in that way would make the model optimistic and would turn the estimate into a circular result. Instead, ground truth is used to verify whether the modelled attacker has reached the intended target and to assess the effort required for the modelled steps.

A practical aspect of this method is the separation between collecting runtime information and simulating the attack. Repeating the simulation many times should not require a new execution of the protected program for each run. This would be slow and, in some cases, difficult to reproduce in a controlled way. The program can instead be executed a limited number of times to collect the runtime information needed by the model. The attack model is repeatedly executed, then uses the collected data.

Faingnaert et al. demonstrate the method on different reverse-engineering scenarios, including cryptographic key extraction, license-check localization, and the search for cheat opportunities in games [3]. These examples show that the framework is not tied to a

single kind of protected asset. Each attack is described through the intermediate and final artefacts that guide the attacker towards the target. They also show that effort estimation depends on the concrete analysis steps performed by the attacker and on the artefacts selected during the attack.

This thesis adopts the same general view of attack effort, but specializes it to code guards. The attacks considered here produce sets of candidate instructions that may belong to the verification phase of a guard. The real verification instruction is known only for evaluation purposes. It is not used to construct the candidate set or to guide the ranking. The effort of an attack is expressed through the position of this instruction in the produced ranking. If the target appears near the beginning of the ranking, the modelled attacker needs to inspect only a small number of candidates before reaching it. If it appears later, the inspection effort is higher.

This makes it possible to compare different code-guard configurations and different attack strategies through measurable outcomes. The comparison is not based on the claim that a ranking proves which instruction is the verification instruction. The ranking is used as an inspection order for the modelled attacker. The target rank then acts as a proxy for the amount of manual inspection required before the relevant instruction is found. The design of the candidate sets, the ranking model, and the interpretation of the target rank are introduced in Chapter 4.

Chapter 3

Related Work

This chapter positions the thesis with respect to existing work. The problem studied here is narrower than a general evaluation of software protections. It concerns the inspection effort required to locate the verification instruction of a code guard in a protected binary, using a ranked candidate set produced by modelled attack workflows.

The chapter is organized around five aspects of the problem. Section 3.1 reviews work on the evaluation of MATE software protections, with attention to attacker effort, protection strength, and repeatable evaluation methods. Section 3.2 discusses integrity protection, code guards, self-checksumming, and attestation, and defines the protection space in which the evaluated guards are placed. Section 3.3 considers empirical and reproducible studies of reverse-engineering attacks. Section 3.4 reviews the program-analysis techniques used by the modelled attacks, including dynamic dependencies, dynamic slicing, taint analysis, and differential analysis. Section 3.5 discusses ranking-based inspection and effort estimation, which motivates the use of the target rank as an effort proxy.

3.1 Evaluation of MATE Software Protections

Dedić et al. propose one of the earliest formal treatments of attacker effort, with a graph-game model in which the attacker tries to identify and disable the checks inserted by a tamper-protection scheme [26]. The attack cost is expressed as the number of steps required to win the game. The model also motivates distributed protection designs, where the attacker has to reason about several checks instead of removing one isolated point. The idea that locating a check has a cost of its own, worth including in the evaluation, carries over directly to the ranking-based approach used later in this thesis.

Attack-surface metrics offer a different way to measure exposure. Howard et al. and Manadhata and Wing count the methods and resources through which a system can be attacked [41, 42]. This gives a structural picture of a protected program, but it says nothing about the order in which a concrete attacker would examine that surface. The view taken in this thesis is closer to a process, where effort follows from the sequence of steps a modelled attacker actually takes, not from a static count of exposed elements.

The ASPIRE project developed a Goal-Question-Metric methodology for evaluating software protection configurations [27]. Its metrics combine static and dynamic properties of the protected code into approximations of protection strength, treating attack effort as something that can be read off the protected program itself. ASPIRE remains a general evaluation framework, however, and does not address the narrower problem of locating a specific instruction of one a specific defense.

Basile et al. bring a risk-management perspective to the same problem [2, 25]. Their meta-model links software assets to the protections that cover them and to the attacks used against those protections. Later work treats the choice of a protection as a decision aimed at reducing risk or raising the cost of a successful attack. This is compatible with the view adopted here where the attack is not an abstract possibility to defend against, but a process with defined steps, assumptions, and costs.

Faingnaert et al. develop tools and models for studying reverse-engineering strategies in the MATE setting [3]. Their framework represents what the attacker knows and how much effort has been spent, and updates both as the attack proceeds. This makes it possible to compare several protected configurations under the same workflow. Their work is the closest methodological reference for the present thesis. The scope is nonetheless different.

3.2 Integrity Checking, Code Guards, Self-Checksumming, and Attestation

Chang and Atallah organize integrity checks as a distributed set of small guards rather than as a single checker [5]. A guard computes an integrity value over a code region, and guards can also check one another, which avoids a single point of failure. Horne et al. study static and dynamic self-checking more broadly, without committing to one particular checking mechanism [6]. Qiu et al. instead work specifically with self-checksumming, and do so from the attacker’s side. Their dynamic information-flow analysis identifies checksum computation, verification, and the code that reacts to a failed check [7]. Read together, these three works already point to the distinction used in this thesis. Self-checksumming is one possible checking mechanism among others; what makes something a guard, in Chang and Atallah’s sense, has more to do with how a check is placed and connected to other checks than with the specific computation used to verify it. Their own guards happen to use a checksum, but the actual contribution is the checks’ distribution and mutual dependence, not the checksum computation itself.

The choice of where the verifier runs marks a further split in this literature. Seshadri et al. present Pioneer, in which a remote party challenges a client to compute a checksum over its own code, so that the verification decision is taken outside the machine being checked [30]. Viticchié et al. connect this idea to server-side reactions, refusing a service after a failed remote check [29]. Chang and Atallah’s guards, and the self-checking techniques discussed above, sit on the other side of this split. The verifier, the reaction logic, and any reference values are all part of the protected program itself.

Ahmadvand et al. provide a taxonomy of software integrity protection techniques from the system, attack, and defense perspectives [10]. This taxonomy is useful for

positioning the thesis on the attack side, since the thesis does not propose a new integrity protection mechanism, but studies how an attacker can locate protection-related code. The same line of work appears in VirtSC, where virtualization obfuscation is combined with self-checksumming to make integrity checks harder to understand and remove [43].

Van den Broeck et al. observe that a modular separation between application code and protection code can help an attacker locate protection logic faster [11]. They propose hiding the boundary between the protected application and the inserted protection code. This observation supports the effort proxy used in this thesis, where if part of a protection’s strength depends on how difficult its code is to locate, the position of the target logic then the position of the target logic in the ranking affects the amount of effort required from the attacker.

3.3 Empirical and Reproducible Studies of Reverse-Engineering Attacks

Some work evaluates software protections by observing how people perform reverse-engineering attacks. These studies give direct evidence about attacker behavior, but they are expensive to run and difficult to repeat at scale.

Ceccato et al. study professional penetration testers and participants in a public challenge attacking software protected with obfuscation and tamper-detection mechanisms [44]. Their results show that attackers do not follow one fixed procedure. Reading the code, testing hypotheses, and trying to bypass the protection happen together, in whatever order the attack calls for. Static analysis is used alongside debugging and direct manipulation of the running process. The study also reports a difference between professional attackers and less experienced participants, where professionals tend to prefer reproducible and automatable strategies, while less experienced participants rely more often on manual trial and error. This distinction matters here because the attacks modelled in this thesis are repeatable workflows, closer to an automated strategy than to opportunistic manual inspection.

Votipka et al. study the reverse-engineering process itself, through a sixteen-participant observational interview study that is not tied to any specific protection technique [45]. Their goal is different from that of Ceccato et al. Rather than testing a protection, they try to understand how reverse engineers reason as they work, aiming to inform the design of analysis tools. What they observe is close to what Ceccato et al. report in the protection-specific setting. Reverse engineering proceeds through repeated cycles of hypothesis and verification, not through a fixed sequence of steps.

Controlled experiments have also measured how obfuscation affects human analysts. Ceccato et al. run a family of experiments in which participants answer comprehension questions about obfuscated Java source code, and compare their accuracy and response time against unobfuscated baselines [46]. Their results give quantitative evidence that some transformations increase analysis time and reduce comprehension accuracy. The setting is different from this thesis, since the experiment works on source-level Java obfuscation rather than binary-level code guards. The underlying assumption, however, is the same,

because the effect of a protection can be measured through the work it costs an analyst to see past it.

Yakdan et al. run a user study in which participants analyze decompiled malware code produced by different decompilers under timed conditions [47]. Their findings show that the quality of the decompiler output affects how quickly and how accurately participants understand binary code. This matters here because it shows that the representation given to an analyst affects inspection effort. The ranked candidate list used in this thesis follows the same idea of structuring the analyst’s search space, although the ranking itself comes from attack-specific dynamic-analysis signals rather than from decompiler output.

Human-subject studies of reverse engineering are difficult to scale. Recruiting professional reverse engineers is hard, and binary-level tasks are difficult to design well; results also vary a great deal across participants. Zhang et al. discuss these methodological challenges for adversarial reverse-engineering experiments run with students [48]. Students are easier to recruit, but their experience does not match that of professional reverse engineers, which limits how far the results can be generalized. Schrittwieser et al. survey obfuscation and code analysis more broadly, and show that protection effectiveness depends on the protection and on the analysis technique used against it [1]. This supports the need for evaluations that state both the protection and the attack assumptions explicitly.

Scripted and modelled attacks address the scalability problem of human-subject studies directly. Instead of recruiting participants, the researcher defines an attack workflow, applies it to each protected configuration, and compares the results. This makes it possible to evaluate several protection variants and attack strategies under the same assumptions. Modelled attacks do not replace human studies, in fact they only capture the strategies the researcher chooses to model, and cannot reproduce the adaptive reasoning of a human attacker. In contrast, their value is repeatability. The same workflow is applied to every configuration, so differences in the result can be related to differences in the protection or in the attack model, rather than to differences between individual attackers.

Faingnaert et al. provide the main methodological reference for this approach [3]. Their work introduces tools and models for representing reverse-engineering strategies and for estimating the effort needed to execute them on concrete protected targets. The same framework has also been used for game resource localization [4].

3.4 Program-Analysis Techniques for Attack Modeling

The attack workflows modelled in this thesis rely on dynamic data dependencies and forward slicing, introduced in Chapter 2 through the work of Korel and Laski and of Agrawal and Horgan [23, 24], to reduce the set of candidate instructions an attacker has to inspect. Neither technique is a contribution of this thesis. Both are treated here as existing components, used to build the candidate instruction sets from which the verification instruction is later selected and ranked.

Dynamic taint analysis is close to the dependency-following approach used by the attacks. Tools such as Dytan track how tainted data propagates through an execution [20],

and Schwartz et al. survey taint analysis together with forward symbolic execution in security contexts [21]. Qiu et al. apply this idea to self-checksumming protections, following information flows from bytes read from the program’s own code to the conditions that decide the tamper response [7]. Their goal is to identify self-checksumming behavior.

The modelled attacks also use differential analysis, which compares executions of a program and treats the differences between them as evidence about the underlying computation [39]. Here, the comparison is between a clean execution and a tampered execution of the same protected binary, and instructions whose observed values diverge between the two runs are kept as stronger candidates.

The same principle appears with different assets, and sometimes in the opposite direction. Bos et al. recover keys from white-box cryptographic implementations by comparing execution traces, without any knowledge of the underlying design [38]. Faingnaert et al. also compare across executions in K-Hunt++, but they look for what stays the same rather than for what changes, since a candidate key is expected to remain constant when the program runs several times with different plaintexts or passwords, while the plaintext and ciphertext values change from run to run [49]. Continella et al. instead vary the input to a mobile application and compare the resulting outputs to detect privacy leaks that static inspection alone would miss [40]. These three works target different assets and even interpret divergence in opposite ways, but they share the idea that running a program more than once and comparing the results can expose behaviour that would remain hidden in a single trace.

In this thesis, differential analysis is used as part of the attack model. It helps guide the modelled attacker towards a smaller part of the binary.

3.5 Ranking-Based Inspection and Effort Estimation

Ranking is a common way to guide manual inspection when the target cannot be identified with certainty. Fault-localization techniques use this idea in debugging. They assign a suspiciousness score to program entities and place the most likely faulty elements first. Wong et al. describe a family of coverage-based fault-localization heuristics built around this principle [50].

Tarantula, proposed and evaluated by Jones and Harrold, is one of the best-known examples of spectrum-based fault localization [51]. It ranks program statements by comparing their coverage in passing and failing test executions, so that a statement executed more often in failing runs than in passing runs receives a higher suspiciousness score. Its purpose is to get a programmer to the faulty statement with less manual searching.

This thesis uses the same inspection principle in a different setting. The ranked elements are binary instructions, not source-level statements, and the goal is not debugging. The modelled attacker inspects the ranked list to find the verification logic of a code guard, not to locate a software fault, and the ranking itself comes from dynamic analysis of a protected binary rather than from source-level test coverage.

The analogy with fault localization is useful, but limited. Source-level information is not available to the modelled attacker, and the inspection is adversarial, since the attacker wants to understand or bypass the protection rather than fix it. The rank is also only an effort proxy. A top position in the ranking reflects the strength of the analysis signals associated to the piece of program studied. Parnin and Orso show that the practical value of automated debugging techniques depends on how programmers actually use the produced ranking, not only on the numerical score assigned by the tool [52].

Chapter 4

Problem Formulation and Design

This chapter defines the problem addressed by the thesis and presents the design used to study it. It first establishes the problem scope and the protected binary model, and then describes the code-guard structure and the defense variant families.

The second part presents the attack workflow used to estimate attacker inspection effort. It defines the inspection target, the analysis seeds, the dependency model, the three attack models, candidate generation, and ranking.

The chapter focuses on the design-level constructs and models. The concrete implementation choices, scripts, execution traces, and command-line procedures are described in the following chapter.

4.1 Problem Statement

Software protections for the MATE setting are designed to delay reverse-engineering attacks rather than to prevent them completely. The main difficulty is evaluating this delay. Generic software metrics can describe how a protected program changes, but they do not necessarily measure how much harder a concrete attack becomes.

Recent work has proposed a more attack-oriented approach, where reverse-engineering attacks are modelled as repeatable procedures and evaluated on different protected versions of the same program. However, this approach still needs to be tested on additional protection mechanisms, attack objectives, and protected assets. This thesis studies this gap in the domain of code guards, where the protected asset is executable code and the attacker has to reason about runtime integrity checks.

The goal of this thesis is therefore to study whether attack-effort modelling can be applied to code-guarded binaries, how different code-guard configurations change the inspection effort required to identify the logic that protects them, and which attack strategies remain effective as the protection design becomes more complex.

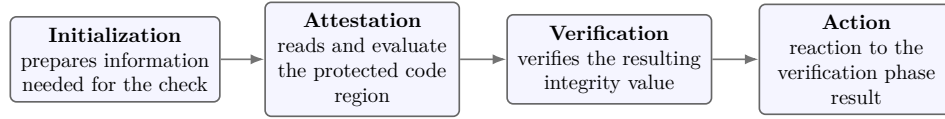


Figure 4.1: Conceptual phases of a code guard.

4.2 Protected Binary Model

The attack workflow considered in this thesis targets an existing program extended with runtime integrity-checking code. The protected binary is derived from a working program, so that the workflow is evaluated on executions with real control flow, multiple code paths, and traces produced by normal program behavior.

The host program is exercised through normal-use scenarios that activate different parts of the binary. These scenarios are used to study guards placed in different code regions without relying on inputs created only for the analysis.

The protection is added without changing the intended logic of the host program. The guard code runs together with the original code and checks selected regions of the binary during execution. When these regions are not modified, the protected binary behaves like the original program. When a monitored region is tampered with, the guard can trigger its response.

The protected binary is the object analyzed by the attack workflow. The workflow observes its executions and reasons about the guard instructions reached during those executions.

4.3 Code Guard Structure

Each code guard injected into the protected binary follows the same conceptual structure. The guard first prepares the information needed for the check, then reads the protected code region, verifies the resulting integrity value, and executes a response if the check fails. This structure is consistent with code-guard and self-checking protections, where runtime checks are used to detect modifications of protected code regions [5, 6, 43].

Figure 4.1 summarizes this four-phase structure at the design level.

Initialization. During initialization, the guard prepares the runtime context required by the integrity check. The protected region, its size, and the expected integrity value are stored in the protected binary as metadata produced during the protection process. At runtime, the guard retrieves the relevant metadata entry and converts it into the concrete values needed by the following phases.

More specifically, the guard resolves the stored code-region reference with respect to the loaded module base, obtains the runtime address of the protected region, loads the corresponding region size and expected integrity value, and stores this information in

the guard state. This initialized state is then used by the attestation phase to read the protected bytes and compute the runtime integrity value.

Attestation. During attestation, the guard reads the bytes of the protected region and computes an integrity value over them. The design uses checksum-like or hash-like algorithms for this operation. Their role is to provide a deterministic runtime check, not cryptographic authentication. For the same input bytes, the algorithm must produce the same value and must be efficient enough to run inside the host program.

The protection argument considered in this thesis does not rely on collision resistance as its main property. The attacker is not modeled as trying to craft a different protected-code byte sequence that produces the same integrity value. Other attack options are possible in practice. For example, an attacker could try to modify the attestation computation, alter the metadata used by the guard, or force the computed value to match the expected one. This thesis deliberately narrows the studied attack target to the verification phase, because the objective is to model the effort required to localize the point where the computed integrity result is consumed by the guard’s decision logic.

Verification. During verification, the computed integrity value is compared with the expected one. This phase determines whether the protected region is accepted as intact or treated as modified.

The instruction, or short instruction sequence, where this decision affects the execution is called the *verification-critical instruction*. This is the main target of the attack workflow studied in this thesis. Finding this instruction means finding the point where the guard’s integrity result is consumed by the control logic of the program.

Action. If verification detects a mismatch, the guard executes a response. The response can differ both in timing and in the type of effect it produces. In terms of timing, the effect can occur immediately after the failed check or be delayed until a later point in the execution. In terms of effect, the response can be explicit, such as aborting the program, or implicit, such as entering a loop, corrupting an intermediate value, changing the output, or following an existing error path.

At the design level, the action is treated as the consequence of the verification decision. The attack workflow therefore focuses on the verification-critical instruction. If that decision point is neutralized, the guard can no longer enforce its response in the intended way. In chained designs, this may not be sufficient to bypass the whole protection, because guards can depend on one another. This case is discussed in Section 4.7.

4.4 Golden Values and Guard Metadata

In the local self-checking setting considered in this thesis, a code guard needs an expected integrity value for each protected region. This value, also called the golden value, is the reference against which the integrity value computed at runtime is checked. Since the

verification is performed inside the protected binary itself, the reference value must be available to the local guard during execution. This differs from remote-attestation settings, where the reference measurement can be held by an external verifier instead of being stored in the program under inspection.

At the design level, the construction assumes a preparation phase that computes the golden values for the protected regions and embeds the information required by the guards into the protected binary. The resulting binary contains both the guard code and the guard metadata needed at runtime. The concrete metadata layout and the implementation of the preparation step are discussed in Section 6.4.

The guard metadata is modeled as a set of data records used by the guards during execution. For each protected region, a metadata record contains at least a reference to the region location, the region size, and the corresponding golden value. The guard code uses these records to initialize the check, compute the runtime integrity value, and compare it with the expected one.

In this design, metadata and executable guard logic are distinct components of the protected binary. Concretely, the golden values and region descriptors are represented as data consumed by the guard, rather than as the executable instructions that implement the guard control flow. The guard instructions may load, transform, or compare values obtained from metadata, but the metadata records themselves are not the verification instructions.

This distinction is structural, not a security boundary. In the MATE threat model, the attacker has offline access to the binary and can inspect or modify both code and data before execution. Placing metadata in a read-only region can prevent ordinary runtime writes, but it does not prevent file-level patching. The distinction is therefore used to define separate analysis targets, not to claim that metadata is protected from the attacker.

This matters for the attack workflow because recovering one component does not automatically recover the other. Locating a metadata record reveals where a stored reference value is kept, but it does not directly identify the instruction where the computed integrity value is checked. Conversely, locating or patching a verification-critical instruction does not by itself describe the metadata record from which the expected value was obtained. The relation between metadata records and guard instructions must be recovered through analysis of the runtime data flow and control flow.

Golden values are fixed after the preparation step. For this reason, the protected code regions must remain unchanged between preparation and execution. Any byte-level change in a monitored region changes the value computed at runtime. This includes both deliberate tampering and legitimate patches applied after the golden values have been embedded. In both cases, the guard observes a mismatch and can trigger its response.

4.5 Interleaving of Guard-Phase Instructions

The guard phases introduced in Section 4.3 are conceptual operations. They do not imply that all instructions implementing a guard must appear in one contiguous routine or execute without intervening host program code. In the protected binaries considered in

this thesis, the instructions that implement initialization, attestation, verification, and action are placed at different program locations and are executed at different points of the host program’s normal control flow.

This distribution has both a static and a dynamic aspect. Statically, the instructions belonging to one guard can be located in different functions or basic blocks instead of forming a compact code block. Dynamically, instructions from the host program, and possibly from other guards, can execute between two operations of the same guard. The guard state is therefore not necessarily confined to a short local instruction sequence. It can be stored, loaded, transformed, or passed across phase boundaries before it is eventually consumed by the verification logic.

This design choice makes the relation between the guard phases less direct for an analyst. Finding an instruction that looks like a verification point does not by itself reveal where the protected bytes were read or where the expected value was loaded from. Conversely, finding code that reads a protected region does not immediately identify the later instruction where the computed integrity value is checked. The analyst has to recover the relation between these operations by following data flow and control flow across the executed program.

The effect on dependency-based analysis must be stated precisely. Executing unrelated host-program instructions between two guard operations does not, by itself, extend the data-dependency path between them. Such instructions only increase the surrounding dynamic trace. They affect the dependency graph when guard values are actually carried through them, for example through memory stores and loads, register transfers, function calls, arithmetic operations, or other instructions that transform or propagate the guard state. In those cases, the path from the protected-code reads to the verification-critical instruction contains additional intermediate nodes, and the workflow has to explore a larger set of instructions before ranking candidate verification points.

In this thesis, this distribution of guard-phase instructions is treated as a property of the designed protection variants, not as an independent experimental variable. Its relevance is that it influences how directly the attack workflow can connect the attestation computation to the verification-critical instruction.

4.6 Defense Variant Families

The evaluation requires a set of protected binaries that differ in controlled ways. A single protection design would only show how the attack workflow behaves against one specific guard structure. For this reason, the thesis uses a family of defense variants. Each variant changes one main aspect of the guard structure while keeping the general protection goal unchanged.

Before introducing the variants, two terms are used to distinguish between a guard as a concrete check and the code that implements it. A *guard instance* is one concrete integrity check associated with a protected region and with the metadata record used for that region. A *guard routine* is a piece of code that implements one or more guard phases,

such as initialization, attestation, verification, or action. Several guard instances can share the same routine, or each instance can be implemented by a separate specialized routine.

The variants are used as comparison points for the attack workflows defined later in this chapter. The purpose is not to prove that one protection is secure in an absolute sense. The purpose is to study how different guard designs influence the attacker-effort estimation model.

Table 4.1 gives an overview of the six variant families. The rest of this section describes them in more detail.

Variant family	Main design change
Baseline direct-verification	Direct comparison between computed and expected integrity value.
Comparison-obfuscated	Verification is expressed through arithmetic or logical operations.
Function-diversified	Guard routines are instantiated separately for different guards.
Chained	Guards are connected through normal/master dependencies.
Syscall-mediated chained	Guard state crosses a file-mediated channel.
State-breaking syscall-mediated chained	The channel is used to break the standard dependency path more strongly.

Table 4.1: Defense variant families and their main design change.

The first family is the *baseline direct-verification variant*. This variant is the reference case. It contains multiple independent guard instances distributed across the execution scenarios of the host program. Each guard instance follows the four-phase structure introduced in Section 4.3. In this variant, the verification phase uses a direct comparison between the computed integrity value and the expected value. The verification-critical instruction is therefore exposed through a recognizable comparison pattern.

The second family is the *comparison-obfuscated variant*. This variant keeps the independent-guard structure, but changes the form of the verification phase. Instead of checking equality through a direct comparison, the evaluated design expresses the same condition through arithmetic or logical operations. In particular, the verification computes a derived value from the computed and expected integrity values and tests that derived value to decide whether the check succeeds. The semantic meaning of the check is unchanged, but the verification-critical instruction is no longer exposed as a simple equality comparison between the two original values.

The third family is the *function-diversified variant*. This variant keeps the same verification style as the comparison-obfuscated family, but changes how the guard routines are organized. Instead of using the same routines for all guard instances, the initialization, attestation, and verification logic are instantiated separately for different guards. As a result, identifying or patching the verification routine of one guard instance does not automatically give the attacker a single shared routine that disables all guards. This variant therefore reduces code reuse across guard instances.

The fourth family is the *chained variant*. In this design, guard instances are no longer fully independent. The state produced or checked by one guard can influence another guard. As a result, the verification of a guard can depend not only on the attestation of its own protected region, but also on values related to other guards. This creates dependencies between guards and makes the protection structure less local than in the independent variants.

The chained family also uses an indirect lookup for guard metadata. The metadata still contains the protected-region descriptors and the expected integrity values, but the guard code does not use a plain direct reference to the metadata record. Instead, it uses an encoded reference that is decoded at runtime to obtain the metadata location. This means that a static reference from the guard code to the metadata record is not immediately visible as a direct address or plain pointer. The purpose is to make the relation between guard code, metadata records, and protected regions less direct to recover.

The fifth family is the *syscall-mediated chained variant*. This variant starts from the chained design and adds a file-mediated transfer of guard state. A value produced during the guard computation is written to a temporary file through a system call and later read back before verification. The purpose is to make part of the propagation between attestation and verification cross a system call boundary, instead of keeping the whole transfer inside ordinary register and memory operations.

The sixth family is the *state-breaking syscall-mediated variant*. This variant also transfers guard state through temporary files, but changes how the value is written and read back in order to break the dependency relation more strongly in the analysis model. The write side stores the value through the file channel, while the read side reintroduces the value through a separate memory location before it is consumed by verification. As a result, the value used by the verification logic is loaded from memory that is not directly connected, in the tracked dependency graph, to the original computation that produced the value.

4.7 Chained Code Guards

The chained variant changes the relation between guard instances. In the independent variants, each guard instance verifies only the integrity value computed for its own protected region. In the chained variant, guard instances are organized in pairs composed of a *normal guard* and a *master guard*. The normal guard computes the integrity value for its protected region and uses it during its own verification phase. The same value is also propagated to the paired master guard, so that the normal guard contributes to another guard's computation instead of remaining only a local check.

Figure 4.2 summarizes this normal/master relation. The horizontal arrow represents the value produced by the normal guard and propagated to the master guard. This value becomes part of the master guard computation, together with the bytes read from the master's own protected region.

A master guard also attests a protected region, but its computation is not based only on the bytes of that region. It also incorporates the value received from the paired normal

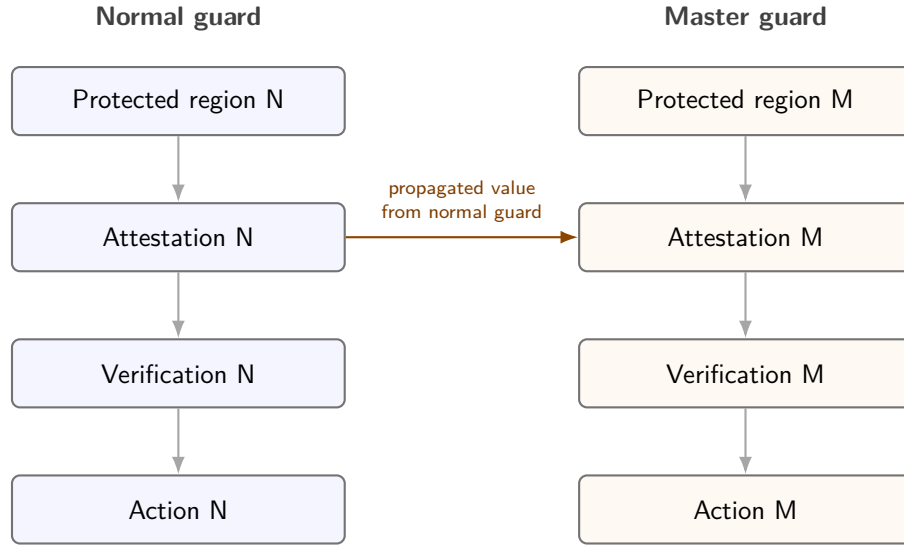


Figure 4.2: Conceptual relation between a normal guard and its paired master guard.

guard. As a result, the master guard’s verification depends on the integrity value computed over its own monitored region and the value propagated from the normal guard. If the protected region checked by the normal guard is modified, the propagated value can change and influence the master guard even when the master’s own protected region is unchanged.

This structure makes the protection less local than in the independent variants. In an independent design, the attacker can reason about each guard instance mainly through the relation between its attestation computation and its own verification point. In the chained design, this relation is extended across guard instances. The verification point of the master guard may depend on state originating from the normal guard. Therefore, neutralizing only the local verification of a normal guard does not necessarily remove all effects of that guard, because the value it produces may still be consumed by the paired master guard. For the attack workflow, the important consequence is that the relevant dependency path can cross the boundary between two guard instances as described in Section 4.5. The workflow has to connect a value produced during the normal guard’s attestation to a later computation or verification point in the master guard.

The design uses different checksum-like algorithms for normal guards and master guards. This is an algorithmic diversification choice. Its purpose is to avoid making all attestation computations recognizable through the same pattern.

4.8 Attack Workflow Design

This section describes the attack workflow used to estimate the inspection effort required to locate the verification point of a code guard. The workflow does not disable the protection automatically and does not prove security of the protected binary. Its role is to reduce the

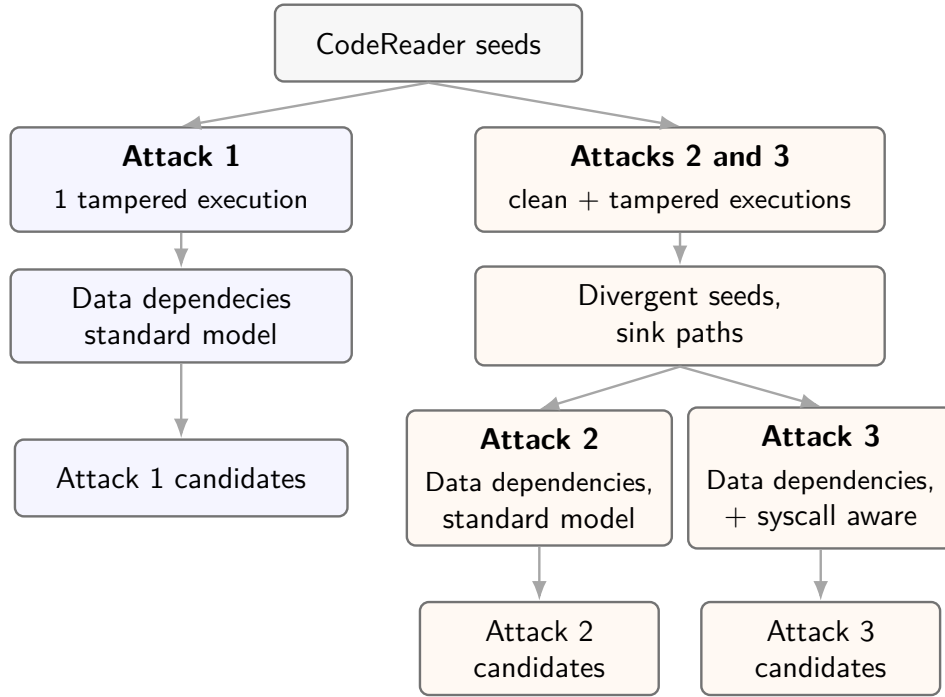


Figure 4.3: Attack workflow schemes.

set of instructions that an attacker has to inspect and to order those instructions according to their relevance.

Figure 4.3 summarizes the common structure of the attack workflow, which will be discussed in detail in the following subsections. The output of the workflow is a ranked list of candidate instructions. A modeled Man-At-The-End attacker inspects this list from the highest-ranked candidate to the lowest-ranked one. The rank of the verification-critical instruction is then used as an effort measure: a lower rank means that the relevant instruction is reached earlier during inspection, while a higher rank means that more candidates must be inspected first.

The workflow has three main steps. First, it selects code-reading instructions as starting points for the analysis, because a self-checking guard must read protected code before computing an integrity value. Then, it follows dynamic data dependencies from these starting points to identify instructions that may depend on values produced during attestation. Finally, it ranks the selected candidate instructions, so the inspection process has a defined order rather than an unordered set of possible targets. The following sections define the inspection target, the seed selection rule, the dependency model, the three attack models, candidate generation, and the ranking interface.

4.8.1 Inspection Target

The modeled attacker inspects the ranked candidates to locate the verification-critical instruction introduced in Section 4.3. This is the instruction selected as the ground-truth target for the evaluation. It corresponds to the point where the value derived from attestation is consumed by the guard logic to decide whether the protected region is accepted or rejected.

This definition does not assume that the whole verification logic always consists of a single machine instruction. In comparison-obfuscated designs, for example, the verification predicate can be implemented by a short sequence of arithmetic, logical, and test-like instructions. However, the evaluation needs one concrete target position against which candidate rankings can be measured. For this reason, each evaluated guard is associated with one verification-critical instruction. This instruction is the point in the sequence closest to the actual guard decision, and neutralizing it would affect whether the guard response is triggered.

The target is more stable than the action executed after a failed check. The action phase can be implemented in different ways, for example by terminating the program, altering program state, or changing the normal control flow. These effects show that the guard has reacted, but they do not directly identify where the decision is made. For this reason, the attack workflow focuses on the verification decision rather than on the final response.

Locating the verification-critical instruction gives the attacker a concrete point to inspect. For an isolated guard, changing this decision point may be enough to disable that guard's response. In chained designs, however, guards can depend on one another. In that case, disabling one verification point may not be sufficient to bypass the whole protection, since other guards may still influence the execution.

The inspection process is modeled as a sequential activity. The attacker receives the ranked candidate list produced by the workflow and inspects the candidates in that order. A candidate is accepted only when there is enough evidence that it participates in the guard decision logic. The effort associated with the attack is therefore related to the number of candidates that must be inspected before reaching the selected verification-relevant instruction.

4.8.2 Code-Reading Instructions as Analysis Seeds

The dependency analysis needs a set of starting points in the execution trace. In this workflow, these starting points are called seeds. Their selection is based on a simple property of self-checking guards: before computing an integrity value, the guard has to read the bytes of the protected code region. This operation is part of the attestation phase and does not depend on the particular checksum algorithm or on the way the guard is interleaved with the rest of the program. Dynamic information-flow analyses for self-checking protections use the same general intuition by following values derived from code locations toward later verification or response effects [7].

For this reason, instructions that read bytes from the executable code region are used as possible entry points for the analysis. They are not assumed to be verification instructions, and they are not assumed to be guards by themselves. Their role is to identify places in the execution where values derived from protected code may start to flow toward later computations.

The seed set is intentionally approximate. Some instructions may read executable bytes for reasons that are not related to code guards. At the same time, an obfuscated guard may split its code-reading activity across several smaller accesses, so that no single instruction represents the whole attestation step. Keeping all these instructions as seeds avoids making an early decision that could remove relevant flows from the analysis.

A seed is therefore only a hypothesis about where attestation-related data may enter the dependency model. The following stages of the workflow use dynamic dependencies and, in the differential attacks, runtime differences between executions to decide which seed paths are more relevant. The way seed quality affects the final order of candidates is defined in the metrics section.

4.8.3 Dynamic Dependency Model

The three attack models start from the same baseline dependency model, which Attack 3 later extends to account for external-channel transfers. The baseline model represents dynamic read-after-write relations observed during one execution of the protected binary.

A dependency exists when an executed instruction uses a value that was produced by an earlier instruction. In this way, the model describes the data flow that actually appeared in the analyzed execution.

This execution-specific view is consistent with dynamic slicing and dynamic dependence graphs, where dependencies are reconstructed from a concrete run rather than from all possible executions of the program [23, 24]. Similar dynamic data-flow representations are also used in reverse-engineering settings to follow values from selected sources through runtime dependencies [53].

The workflow uses this model starting from the selected seeds. From each code-reading instruction, the analysis follows dependencies forward and identifies the instructions that may depend on values derived from the attestation phase. The instructions reached from a seed form the dependency region explored for that seed.

Reachability alone is not enough to identify the verification instruction. An instruction can be reached from a seed and still be unrelated to the guard decision. For this reason, each attack model applies an additional selection rule to the reachable instructions. The single-execution attack focuses on compare-like reachable instructions, while the differential attacks also use differences observed between clean and tampered executions.

The baseline dependency model follows value propagation through user-space registers and user-space memory. This is sufficient for values that remain inside the ordinary execution domain of the program. It is not sufficient when a defense deliberately routes an integrity-relevant value through an external channel. That case is isolated in Section 4.8.6 and is handled by the extended model used in Attack 3.

4.8.4 Attack 1: Single-Execution Localization

Attack 1 represents the simplest attack model considered in this thesis. The attacker observes one execution of the tampered binary and does not use a clean execution as a reference. The goal is to obtain a first candidate set starting only from the behavior observed in that single run.

The attack starts from the code-reading seeds introduced in the previous section. From each seed, the workflow follows dynamic dependencies in the forward direction. Among the reachable instructions, it selects those with a decision-like form, such as comparison-oriented or test-like instructions, and uses them to build a set of possible verification instructions. This choice reflects the role of the verification phase, where the value computed by the guard is eventually used to decide whether the checked code region should be accepted.

The main limitation of this attack is the absence of a baseline execution. Since only the tampered run is available, the workflow cannot directly check whether a reachable instruction changes its behavior because of the modification applied to the protected code.

4.8.5 Attack 2: Differential Localization

Attack 2 models a differential localization strategy. The attacker compares two executions under the same input: one from the unmodified binary and one from the tampered binary. The purpose is to use the difference between the two executions to focus the dependency analysis on the parts affected by the modification of the protected code. This follows the general idea of differential dynamic analysis, where runtime differences between executions are used to reduce the amount of code that must be inspected [39].

The comparison is first applied to the code-reading seeds. A seed whose observed value changes between the two executions is useful for the specific tamper condition, because it is directly affected by the modification. A seed whose value remains the same does not provide the same evidence for that run. This does not mean that non-divergent seeds are unrelated to guards in general. It only means that they are less useful for localizing the effect of the specific modification being analyzed.

The candidate set is then built from dependency paths that connect relevant seeds to instructions whose behavior differs between the clean and tampered executions. These later instructions act as tamper-sensitive sinks. The instructions along the path do not all have to diverge. An instruction may still be included if it connects a relevant seed to a later divergent point, even when its own observed value does not change directly.

Attack 2 is more selective than Attack 1 because it does not rely on reachability alone. An instruction is not considered only because it is reachable from a code-reading seed. It must also belong to a path that links the seed to a tamper-sensitive point. In this way, the clean execution helps separate ordinary reachable code from code that is related to the modification of the protected region.

Attack 2 still relies on the standard dependency model described in Section 4.8.3. If a defense routes integrity-relevant values through an external channel before verification, the path from attestation to verification may be broken for the model. The value still affects

the program behavior, but the standard model cannot follow it across that boundary. This limitation motivates Attack 3.

4.8.6 External-Channel Limitation of the Standard Dependency Model

The standard dependency model follows value propagation inside the ordinary execution domain of the program. Some defense designs, such as the syscall-mediated variants introduced in Section 4.6, route integrity-relevant values through an external channel before those values are used by the verification phase.

At the program level, the relation between attestation and verification is still present. The verification decision still depends on a value derived from the protected code region. However, this relation is not visible to the standard dependency model once the value leaves the domain represented by the model and later re-enters it.

This affects the attack workflow directly. A traversal that starts from a code-reading seed can follow the dependency path only up to the point where the value leaves the modeled domain. The verification-critical instruction may then remain outside the reachable region, even if it still depends on the attestation result at the program level. This limitation motivates the external-channel-aware attack model described in the next section.

4.8.7 Attack 3: External-Channel-Aware Differential Localization

Attack 3 uses the same differential setting as Attack 2, but changes the dependency model used during candidate generation. The clean and tampered executions are still compared to identify tamper-sensitive behavior. The difference is that Attack 3 adds explicit bridge edges for syscall-mediated transfers of guard state.

A bridge edge connects a write-side operation that sends a guard value into the external channel with a read-side operation that later reintroduces a value from the same channel. This edge is not meant to model the full behavior of the external resource. It only represents, for the purpose of the attack workflow, that the value read back by the program can be related to the value previously written by the guard.

With these bridge edges, dependency traversal can continue across the write-read boundary. Candidate generation can therefore include instructions that would remain disconnected under the standard dependency model. This is the main difference from Attack 2.

Attack 3 is therefore intended for defense variants where the relevant guard state is propagated through an external channel before verification. It does not try to reconstruct the complete semantics of that channel. It only adds the minimal connections needed to recover candidate paths between the attestation side and the later verification logic.

4.8.8 Candidate Generation

Candidate generation is the step that turns seeds and dependency information into instructions to inspect. For each candidate, the workflow keeps the seed, or the seeds, from which that instruction was reached. This association is useful because the same instruction may be reached from different code-reading points, and the relevance of those points can affect the later ranking.

The selection rule depends on the attack model. In Attack 1, an instruction is selected if it is reachable from a code-reading seed and has a decision-like form, such as a comparison or a test. In Attacks 2 and 3, the workflow also uses the difference between clean and tampered executions. Divergence is used to identify seed paths that are affected by the tamper condition and to select sink instructions that expose this difference. The candidate set can then include both divergent instructions and non-divergent instructions that lie on a dependency path from a relevant seed to a divergent sink.

For this reason, candidates in Attacks 2 and 3 are not limited to comparison-like instructions, nor are they limited to instructions whose observed values change directly. An instruction can be relevant because it connects the code-reading seed to a later tamper-sensitive point. This is important for guards where the verification logic is spread over several instructions rather than expressed as a single direct comparison.

A candidate is not a confirmed verification instruction. It is only an instruction that the workflow considers worth inspecting. The confirmation of its role in the guard decision is left to the validation procedure described in Section 7.5.2.

4.8.9 Ranking Interface

The attack models described in the previous sections produce candidate instructions. These candidates are not useful to the attacker as an unordered set, because the attacker still has to decide where to start the inspection. The ranking step addresses this problem by assigning an order to the candidates.

At the design level, the ranking step is treated as an interface between candidate generation and manual inspection. It takes the candidate instructions produced by an attack model and orders them according to a priority function. The design does not require the same priority function for all attacks. Different attacker models can use different ranking heuristics, depending on the evidence available to them and on the assumptions they make about the protected binary.

The ranking is not intended to identify the verification-critical instruction with certainty. It models a plausible inspection strategy. Candidates that are more compatible with the expected structure of a code guard are placed earlier in the list, while candidates with weaker evidence are inspected later. In this sense, the ranking model represents the priority rules followed by the modeled attacker.

The information available to the priority function depends on the attack model. Attack 1 uses only one tampered execution, so its ranking can rely only on evidence observed in that execution. Attacks 2 and 3 compare clean and tampered executions, and can therefore use

additional information about how values and dependency paths change after tampering. This difference is reflected in the metric design presented in the following section.

When the ground-truth verification-critical instruction is known, its position in the ranked list can be used as an effort proxy. A lower position means that fewer candidates must be inspected before reaching the target. If the target is not included in the candidate set, the corresponding attack model has failed to recover a path to the verification point under its candidate-generation rules.

The following chapter defines the heuristic metrics used to build this ranking. It explains which signals are used for Attack 1, which signals are used for Attacks 2 and 3, how seed-level information is included, and how the final rank is interpreted as an estimate of attacker effort.

Chapter 5

Heuristic Metrics for Attacker-Effort Estimation

This chapter defines the heuristic metrics used to rank candidate instructions during the analysis of code guards. The goal of these metrics is not to prove that a candidate instruction is the verification instruction. Instead, the metrics model plausible priority rules that a Man-At-The-End attacker could use when inspecting the output of a dynamic and static dependency analysis.

The realism of the metrics is assessed by relating each one to behavior that is reasonable for an attacker performing instruction-level analysis. In particular, the metrics are grounded in the literature on dynamic slicing, dynamic information-flow analysis, self-checksumming, and code guards [23, 24, 53, 7, 5, 6]. The objective is therefore not to claim that the weights are optimal, but to show that each feature formalizes a possible real-world attacker priority.

The metrics introduced in this chapter should be read as assumptions about the attacker model considered in this thesis. Each metric captures one possible criterion for prioritizing instructions, such as proximity to a code-reading seed, frequency of observation, diversity of dependency contexts, rarity of failure-compatible behavior, opcode plausibility, downstream propagation, path centrality, or runtime divergence. None of these criteria is sufficient in isolation. A candidate instruction may score highly for one reason and still belong to ordinary program logic.

For this reason, the metrics are combined into priority functions which only define the order in which the modeled attacker would inspect the candidates produced by the attacks.

Since the three attacks expose different evidence, this thesis does not use a single ranking function for all scenarios. Each ranking only uses the information available to the corresponding attack.

Metric	Attack 1	Attacks 2 and 3	Role
DivergenceScore	–	×	Instruction-level feature
OpcodeScore	–	×	Instruction-level feature
DepthScore	×	×	Instruction-level feature
OutgoingDependencyScore	–	×	Instruction-level feature
HitCountScore	×	–	Instruction-level feature
PathCentrality	–	×	Instruction-level feature
TupleDiversityScore	×	–	Instruction-level feature
FailureObservedScore	×	–	Instruction-level feature
CodeReaderMultiplier	×	×	Seed-level adjustment

Table 5.1: Metrics and secondary adjustments used by the ranking model for the different attack scenarios. Instruction-level features are combined inside the corresponding priority function. The `CodeReaderMultiplier` is not part of the instruction-level priority function; it is applied afterwards as a seed-level adjustment when multiple code-reading seeds are available.

5.1 Candidate Ranking Model

Let i be a candidate instruction produced by one of the attacks implemented in this work. The ranking step assigns a score to each candidate instruction in order to estimate how early an attacker would inspect it.

The final ranking is computed as a weighted combination of the selected heuristic features:

$$\text{PriorityScore}(i) = \sum_k w_k \cdot H_k(i).$$

where $H_k(i)$ is the value of the k -th heuristic feature for instruction i , and w_k is the weight assigned to that feature. Unless stated otherwise, the weights are treated as configurable parameters. In the baseline configuration, the same weight is assigned to all features. This choice avoids assuming a priori that one signal is more important than the others.

The score is not a probability. It should be interpreted as a priority value where higher-scoring candidates are the instructions that the modeled attacker would inspect earlier.

5.2 Opcode Score

The *OpcodeScore* assigns a static priority to a candidate instruction based on its mnemonic. The assumption behind this feature is that an informed attacker, once the protection scheme has been recognized as code-guard-based, can use known patterns of self-checking

protections to bias the search toward instruction classes that are more likely to participate in verification logic.

Prior work on self-checksumming identifies verification points by tracing values derived from code regions toward comparisons, control transfers, writes to executed code, or other effects related to the tamper response [7].

For this reason, comparison instructions such as `cmp` and `test` receive the highest priority. Arithmetic and logical instructions such as `xor`, `sub`, `or`, and `and` receive an intermediate score because they may be used to compute, obfuscate, or encode the verification result. Conditional jumps receive a lower score than direct comparisons, since they usually consume the result of a previous comparison rather than computing the verification value themselves. Data movement instructions receive a low score because they are more often related to propagation than to the final decision.

The score used in this work is:

$$\text{OpcodeScore}(i) = \begin{cases} 1.00 & \text{if } i \in \{\text{cmp}, \text{test}\} \\ 0.85 & \text{if } i \in \{\text{xor}, \text{sub}\} \\ 0.70 & \text{if } i \in \{\text{or}, \text{and}\} \\ 0.55 & \text{if } i \in \{\text{set*}, \text{cmov*}\} \\ 0.40 & \text{if } i \text{ is a conditional jump} \\ 0.30 & \text{if } i = \text{call} \\ 0.20 & \text{if } i \in \{\text{mov*}, \text{push}, \text{pop}\} \\ 0.10 & \text{otherwise.} \end{cases}$$

This score is deliberately heuristic. It does not claim that all verification instructions must be comparisons, nor that other instructions cannot implement verification in an obfuscated design. It models a realistic attacker priority by inspecting earlier instructions whose semantics resemble comparison, difference computation, or decision logic than generic propagation or unrelated instructions.

5.3 Divergence Score

The *DivergenceScore* assigns higher priority to candidate instructions whose observed runtime values differ between a clean execution and a tampered execution. Runtime divergence is used as a heuristic signal for Attacks 2 and 3. It indicates that an instruction is affected by the tampering, but it does not prove that the instruction belongs to the verification logic.

This follows the general idea of differential dynamic analysis, where runtime differences between executions are used to focus the inspection on code whose behavior changes across runs [39]. In this thesis, the same principle is used as one component of the ranking function.

The score uses two binary flags. Each flag has value 1 when the corresponding condition holds and value 0 otherwise. The first flag is the raw divergence flag:

$\text{DivergenceReportFlag}(i)$.

This flag is set to 1 when instruction i appears in the divergence report, meaning that its observed values differ between the clean and tampered executions.

The second flag is the seed-reachable divergence flag:

$\text{DivergenceFlag}(i)$.

This flag is set to 1 when instruction i is divergent and is also reachable from the selected code reader instruction through the dependency structure used during candidate generation. It therefore adds structural evidence: the instruction is not only divergent, but also connected to the code-reading seed from which the attack workflow starts.

The final score is:

$$\begin{aligned} \text{DivergenceScore}(i) = & 0.50 \\ & + 0.325 \cdot \text{DivergenceReportFlag}(i) \\ & + 0.175 \cdot \text{DivergenceFlag}(i). \end{aligned}$$

The constant term 0.50 is assigned to every candidate. By itself, it does not distinguish candidates within this metric. Its role is to limit this metric relative to the other heuristic scores. Without this baseline, the difference between divergent and non-divergent instructions would be too large, and divergence would dominate the ranking more than intended. With the chosen formula, the score is 0.50 for candidates without observed divergence, 0.825 for candidates reported as divergent, and 1.00 for divergent candidates that are also reachable from the selected code reader instruction.

The score remains heuristic. A divergent value may belong to the attestation computation, to an intermediate propagation step, or to ordinary program logic influenced by the tampering. The score only expresses that, all else being equal, instructions with observed tamper-dependent behavior deserve more attention.

5.4 Depth Score

The *DepthScore* measures the distance of a candidate instruction from the selected code-reading seed in the dependency graph. The assumption is that an attacker who starts from a suspicious instruction reading bytes from the `.text` section will follow runtime dependencies forward in search of the verification logic. Candidates reached through fewer dependency steps are given higher priority than candidates reached through longer dependency chains.

This assumption is consistent with dynamic program slicing. Dynamic slicing restricts the analysis to statements and dependencies that actually affected a value during a specific execution, instead of considering all statements that could affect it statically [23]. Dynamic dependence graphs provide a similar execution-specific representation of dependencies between statement occurrences [24]. In this work, the attacker is modeled as using such

an execution-specific dependency structure to navigate from the code-reading seed toward later computations.

The score is defined as:

$$\text{DepthScore}(i) = \frac{\log(2)}{\log(2 + \text{depth}(i))}$$

where $\text{depth}(i)$ is the minimum BFS distance from the selected code reader instruction to instruction i in the induced dependency graph.

The logarithmic decay avoids treating depth as a strict penalty. This is important because the verification instruction is not necessarily close to the code-reading seed. A code guard may accumulate a checksum, transform the value, materialize flags, use indirect control flow, or delay the response. The DepthScore therefore models traversal cost, not verification likelihood. A shallower instruction is cheaper to reach, but it is not necessarily the final verification instruction.

5.5 Outgoing Dependency Score

The *OutgoingDependencyScore* measures whether the value or condition produced by a candidate instruction is observed to influence later instructions in the dependency graph used by the attack workflow. For a candidate instruction i , $\text{outgoing_count}(i)$ denotes the number of dependency edges leaving i toward later instructions in that graph.

The assumption behind this feature is limited. A verification-related instruction is often followed by some later use of its result. For example, a comparison may set flags that are consumed by a conditional branch, or an intermediate predicate may be propagated before it affects the guard response. For this reason, a candidate with observed outgoing dependencies is treated as slightly more informative than a candidate whose result appears to stop at that point.

The absence of outgoing dependencies must be interpreted carefully. In this thesis, dependency edges are not a complete semantic model of the program. They represent only the runtime dependencies observed by the analysis. Therefore, $\text{outgoing_count}(i)$ does not always mean that the instruction has no later effect. For some instruction classes, the missing outgoing edge may be caused by the abstraction used by the dependency tracker.

The score distinguishes three cases:

$$\text{OutgoingDependencyScore}(i) = \begin{cases} 0.40 & \text{if } \text{outgoing_count}(i) \geq 1, \\ 0.30 & \text{if } \text{outgoing_count}(i) = 0 \text{ and } \text{ML}(i), \\ 0.20 & \text{if } \text{outgoing_count}(i) = 0 \text{ and } \neg \text{ML}(i). \end{cases}$$

Here, $\text{ML}(i)$ indicates that instruction i is *model-limited*. A candidate is model-limited when the absence of outgoing dependencies is not reliable evidence that the instruction is semantically terminal. The main example is a `call` instruction. A call may invoke a helper routine, a verification routine, or a response routine, but the effects of the invoked

code are not necessarily represented as outgoing data-dependency edges from the call site itself. In such cases, a zero outgoing count may reflect a limitation of the dependency abstraction rather than the absence of a later effect.

The baseline value 0.20 is assigned to candidates with no observed outgoing dependencies and no model-limitation exception. This value does not reward these instructions. Its role is to prevent this metric from penalizing terminal-looking candidates too strongly relative to the other heuristic scores. The absence of outgoing dependencies is useful evidence, but it is not definitive evidence that the instruction is irrelevant to verification. For this reason, such candidates receive the lowest value in this metric, but they are not assigned a zero contribution.

The intermediate value 0.30 is not a reward for model-limited instructions. It represents uncertainty. These candidates have no observed outgoing dependency, so they are ranked below candidates with explicit downstream propagation. However, they are not penalized as strongly as ordinary instructions whose produced value or condition appears terminal in the dependency graph.

The positive case is intentionally coarse-grained. The score rewards the presence of at least one outgoing dependency, but it does not increase with the number of outgoing edges. A high fan-out may be caused by ordinary value propagation, register copying, memory stores, or imprecision in the dependency graph. The metric therefore uses outgoing dependencies only as a weak priority signal. In this way, candidates whose result is observed to propagate further are ranked slightly above candidates without such evidence.

5.6 Hit-Count Score

The *Hit-count score* measures how often a candidate instruction is executed during the analyzed tampered run. It is defined as:

$$\text{HitCountScore}(i) = \frac{\log(2)}{\log(2 + \text{hit_count}(i))}$$

where $\text{hit_count}(i)$ is the number of dynamic executions of instruction i in the trace collected from the tampered binary. The score gives higher priority to instructions executed only a few times and decreases with a logarithmic decay for instructions executing more often.

The feature is based on a simple attacker-priority assumption. In a dynamic trace, instructions with very high execution counts can be assumed as part of repeatedly executed program behavior, such as loops, parsing, or library calls. An attacker is therefore likely to treat very frequent instructions as ordinary program logic, unless other signals suggest that they are protection-related.

The use of execution frequency as a dynamic feature is supported by the profiling-related literature, where runtime frequencies are used to characterize program behavior [54, 55, 56]. The security-related meaning of the feature exploited here is connected to minimizing the attack surface of the program. Attack surface measurement treats exposed resources, methods, channels, and data as potential attack opportunities [41, 42]. In the current study,

hit count is interpreted as a proxy for dynamic exposure, since frequent materialization, propagation, or access of verification-related values increases the opportunities for the attacker to observe them.

This does not mean that all protection-related instructions have low hit counts. A guard may be placed inside a loop, executed periodically, or triggered several times for redundancy. The metric only captures a plausible filtering strategy according to which, when candidates are otherwise comparable, sparse verification-like instructions are inspected earlier than instructions executed thousands of times as part of ordinary computation.

5.7 Path Centrality

The *PathCentrality* feature measures how often a candidate instruction appears on forward dependency paths from a selected code reader instruction to divergent instructions. Let P be the set of all paths from the code reader instruction s to reachable divergent instructions in the data dependencies graph. Let $P(i)$ be the subset of those paths that contain instruction i . The score is defined as:

$$\text{PathCentrality}(i) = \frac{|P(i)|}{|P|}.$$

This feature models the assumption that an attacker will pay attention to instructions that occur repeatedly along dependency paths from code-reading behavior to divergent behavior. Such instructions may act as propagation points, convergence points, or intermediate transformations through which the attestation-derived value passes.

The metric is motivated by the same execution-specific view used in dynamic slicing and dynamic taint analysis. Dynamic slicing follows dependencies that actually occurred in one execution [23, 24]. Offline taint-analysis frameworks also reconstruct runtime data-flow paths and use them to support reverse engineering tasks [53]. Dynamic information-flow analyses for self-checksumming similarly follow values derived from code locations toward verification or response effects [7]. The present metric uses the same general idea in which is supposed that an instruction occurring on many forward dependency paths is treated as more central to the observed propagation.

This feature has a clear limitation. Instructions close to the code reader instruction may appear in many paths simply because all paths start from the same source. The score may therefore over-rank early attestation or propagation instructions.

5.8 Tuple Diversity Score

The *Tuple diversity score* measures the stability of operand values used by a candidate comparison instruction. Let $\text{unique_tuple_count}(i)$ be the number of distinct operand-value pairs observed for instruction i during the tampered execution. The score is defined as:

$$\text{TupleDiversityScore}(i) = \frac{\log(2)}{\log(2 + \text{unique_tuple_count}(i))}.$$

A high score is assigned to comparisons whose operands remain stable across dynamic executions. When the same comparison is observed with many different operand pairs, the score decreases.

The assumption behind this feature is that a fixed golden-value verification may produce stable operand tuples under a fixed binary and fixed input. Several self-testing and self-defending mechanisms utilize a checksum computation for a protected code region and verify it against an expected value [6, 5]. If the protected code bytes do not change across tampered runs, the value involved in the verification may also remain stable.

This is an attacker-oriented heuristic and not a detection pattern. Normal applications will have static comparisons to constant values, error numbers, configuration variables, and other values. However, better methods of protecting the program can make use of masking, salting, checksum rolling, dispatching, and even key generation based on checksums. In these instances, the actual comparison function does not have low tuple diversity.

Among comparison candidates that are otherwise similar, a low-diversity comparison is inspected earlier because it is compatible with a fixed golden-value check. This signal is not sufficient on its own. It becomes useful when combined with the other heuristic metrics.

5.9 Failure-Observed Score

The *Failure-observed score* captures whether a candidate comparison instruction produced, during the tampered execution, a result that would be consistent with the failure of an integrity check. The metric is defined separately for `cmp` and `test` because the two instructions encode failure-compatible outcomes in different ways.

For a `cmp` instruction c , the flag is:

$$\text{FailureObservedFlag}(c) = \begin{cases} 1 & \text{if } \exists \text{ execution of } c \text{ such that } v_0 \neq v_1 \\ 0 & \text{otherwise.} \end{cases}$$

For a `test` instruction c , the flag is:

$$\text{FailureObservedFlag}(c) = \begin{cases} 1 & \text{if } \exists \text{ execution of } c \text{ such that } (v_0 \& v_1) \neq 0 \\ 0 & \text{otherwise.} \end{cases}$$

The assumption behind the metric is that, if the attacker modifies a protected code region before execution, the computed integrity value should no longer match the expected reference when the affected guard is reached. This assumption is consistent with self-checking and guard-based protections, where tampering with the protected region is expected to affect the value used during verification [6, 5].

The score models a practical attacker behavior. After tampering with the binary, the attacker observes the execution and searches for the point where the modification is detected. A comparison that shows a failure-compatible outcome during the tampered run is therefore more interesting than a comparison that never shows such behavior. This also matches the logic of dynamic analyses that trace values derived from code regions toward verification points [7].

The limitation is immediate because ordinary program logic contains many comparisons that evaluate unequal operands or non-zero tests. Loop bounds, parser checks, string comparisons, error checks, and input-dependent branches may all satisfy the same condition. The feature is therefore intentionally weak and binary.

5.10 Code-Reader Multiplier

In some executions, the CodeReader phase identifies more than one instruction that reads bytes from the `.text` section. Since each of these instructions can be used as a seed for dependency analysis, the framework uses a secondary seed-level criterion to prioritize the dependency graphs generated from different code-reading instructions.

The motivation for this criterion is tied to the differential attack setting. When the attacker creates a tampered binary, the location of the modification is known to the attacker. The goal of the analysis is then to identify the part of the dependency graph whose behavior depends on that modification. A CodeReader seed is therefore more suspicious when it reads a code region that overlaps the tampered region, because such a seed is more directly connected to the modified bytes whose effect the attacker is trying to follow.

This does not assume that the attacker already knows the verification instruction. The tampered region is only the starting point of the analysis. The verification-critical instruction remains the unknown target. The multiplier uses the known tamper location only to prioritize CodeReader seeds whose read zones are compatible with the modified region.

The model distinguishes three cases. First, if no modified region is provided, the selected CodeReader seeds are ranked according to the number of bytes read from the code section. The intuition is that a seed reading a larger portion of `.text` is more compatible with attestation-like behaviour than a seed reading only a small number of bytes. Let n be the number of distinct read-size levels among the selected seeds. Seeds with the same number of bytes read belong to the same ranking level. Rank 1 corresponds to the largest read-size level. The multiplier is then defined as:

$$\text{CodeReaderMultiplier}(s) = \frac{n - \text{rank}(s) + 1}{n}.$$

Second, if the modified region is provided and at least one selected CodeReader seed reads a code region overlapping it, the ranking becomes tamper-aware. In this case, all seeds whose read zones overlap the modified region are placed in the first ranking level and receive multiplier 1. The remaining non-overlapping seeds are then ranked by decreasing

number of bytes read from `.text`. The same rank-based formula is applied, but the first rank is reserved for seeds that actually read the region modified by the attacker.

Formally, let the modified region be represented as the half-open interval

$$T = [t, t + \ell),$$

where t is the image-relative offset of the first modified byte and ℓ is the size of the modified region. In the simplest case, where only one byte or one instruction location is considered, $\ell = 1$. Let a CodeReader seed s read a code zone

$$R_s = [a_s, b_s).$$

The seed is considered to overlap the modified region when

$$a_s < t + \ell \quad \wedge \quad t < b_s.$$

If at least one selected seed satisfies this condition, all overlapping seeds receive rank 1. The non-overlapping seeds receive ranks $2, \dots, n$ according to their distinct read-size levels, sorted in decreasing order. The multiplier remains:

$$\text{CodeReaderMultiplier}(s) = \frac{n - \text{rank}(s) + 1}{n}.$$

For example, assume that three CodeReader seeds are selected. The first reads 600 bytes, the second reads 700 bytes, and the third reads 300 bytes but its read zone overlaps the modified region. Although the third seed reads fewer bytes, it receives rank 1 because it directly reads the region affected by the attacker's patch. The remaining seeds are ranked by read size:

Seed	Bytes read	Reason	Multiplier
s_3	300	overlaps modified region	1.00
s_2	700	largest non-overlapping read	0.66
s_1	600	second non-overlapping read	0.33

Third, if a modified region is provided but none of the selected CodeReader seeds reads a zone overlapping that region, the framework falls back to the size-based ranking. This prevents the absence of an overlap from disabling the multiplier entirely, while avoiding the false assumption that a specific seed is directly related to the modified region.

The CodeReader Multiplier does not replace the instruction-level ranking function. It is a seed-level adjustment applied after the priority score of each candidate instruction has been computed. For a candidate instruction i reached from a seed s , the final score is computed as:

$$\text{FinalScore}(i) = \text{PriorityScore}(i) \cdot \text{CodeReaderMultiplier}(s).$$

In this way, the instruction-level metrics still determine how plausible a candidate instruction is as a verification point, while the CodeReader Multiplier adjusts this score according to the plausibility of the seed from which the dependency graph was generated.

The criterion remains intentionally heuristic. A large code read can be caused by unpacking, copying, instrumentation, or unrelated code-processing logic. Conversely, an obfuscated guard may read the protected region through several small accesses. Therefore, the multiplier should not be interpreted as direct evidence that a seed belongs to the protection logic. It is only a seed-prioritization criterion used to rank dependency graphs before evaluating their candidate verification instructions.

5.11 From Candidate Ranking to Attacker Effort

The heuristic metrics defined above do not directly identify the verification instruction. Instead, they instantiate a priority function that a modeled Man-At-The-End attacker can use to order candidate instructions for manual analysis. Candidates with higher priority are inspected earlier because they are more compatible with the expected structure of code-guard verification logic.

This interpretation is aligned with ranking-based fault localization. In that setting, suspiciousness measures are used to prioritize the inspection of program entities, so that more suspicious statements are inspected before less suspicious ones [50, 51]. Tarantula, for example, ranks program statements by suspiciousness, and the rank of the faulty statement indicates how early a developer would encounter it when following the ranking [51]. This thesis applies the same ranking-based principle to reverse engineering of code-guard protections.

The distinction between ranking and identification is important. A high-ranked candidate is not automatically accepted as the verification instruction. The attacker must still validate whether the candidate explains the observed tamper response. In this context, validation means inspecting the surrounding code, following relevant data and control dependencies, observing runtime state, and possibly rerunning the program after applying local patches. This view is consistent with work on automatic debugging, where ranked program locations support failure investigation but do not replace the reasoning needed to explain the failure [52].

For each candidate, the modeled attacker performs a validation step to decide whether the instruction actually participates in the guard decision logic. At the design level, this validation is not modeled as an automatic classification problem. It represents manual reverse-engineering work, where the attacker inspects the local code context, follows relevant data and control dependencies, observes runtime values when needed, and checks whether modifying the candidate would explain the observed tamper response.

This validation step explains why the attacker does not necessarily stop at the first high-ranked instruction. A high-ranked candidate may still be rejected if it only belongs to attestation, if it propagates an intermediate value without driving the guard decision, or if modifying it does not suppress the tamper response in a valid execution. The selected verification-critical instruction is accepted when it is the point where the attestation-derived value is consumed to produce the guard decision, as defined in Section 4.8.1. The concrete validation procedure used in the case study is described in the validation chapter.

Under this interpretation, the rank of the ground-truth verification-critical instruction approximates the number of candidates that the modeled attacker must inspect before reaching a candidate that can be validated as the verification point. If the average cost of inspecting and validating one candidate is denoted by C , the effort for a run r can be approximated as:

$$Effort(r) \approx C \cdot ExpectedRankInTie(t_r),$$

where t_r is the ground-truth verification instruction for run r . Since C is treated as an unknown constant factor, this work reports the rank itself as the primary observed effort measure:

$$ObservedEffort(r) = ExpectedRankInTie(t_r).$$

The use of **ExpectedRankInTie** accounts for tied scores. If multiple instructions receive the same final score, the attacker has no ranking-based reason to prefer one over the others inside the tie. Therefore, the expected position within the tie provides a more appropriate effort estimate than either the best-case or worst-case rank alone.

If the ground-truth verification instruction is not present in the candidate set, the run is reported as unresolved for that attack variant. No artificial rank is assigned. These unresolved cases are important because they indicate that the attacker model failed to recover a dependency path leading to the verification point. For this reason, the experimental evaluation reports both the target reachability rate and the empirical distribution of observed effort for the runs where the target is reached.

Repeating this procedure across different guards, samples, and attack models produces an empirical distribution of inspection effort. Each repetition gives us one data point, which is how many candidates the attacker would check before hitting the verification statement.

5.12 Design Assumptions and Limitations

The defense designs and attack workflows described in the previous sections are evaluated under a bounded model. The goal is not to show that the protections prevent reverse engineering. The goal is to study how different guard designs affect the effort estimated by the modeled attack workflows.

The protection goal is bounded. The guards are designed to increase the effort required to locate the verification-critical instruction within the attack workflows considered in this thesis. They do not claim to provide absolute security. The evaluation characterizes effort within the selected attack models, not against every possible reverse-engineering strategy.

Guard coverage is selective. Each guard monitors a specific code region. Code outside the monitored regions is not protected by that guard mechanism. The choice of monitored

regions is part of the design of each variant. If an attacker modifies only unmonitored code, the guards described here do not trigger a response.

Code stability is assumed. The stored golden values are computed over the monitored code regions before the protected binary is used. These regions are assumed to remain unchanged between construction and execution. Any later byte-level modification to a monitored region changes the value computed at runtime. This includes both deliberate tampering and legitimate software updates applied after the golden values have been embedded.

Behavioral observability is required for differential workflows. The differential workflows assume that a tampered binary produces behavior that can be distinguished from the behavior of the clean binary. If a modification does not change the observed behavior, the differential analysis has no useful signal to follow. A checksum collision is one possible case, because the computed integrity value may still match the stored golden value. The thesis does not evaluate resistance to adversarial collision construction.

External-channel modeling is conservative. Attack 3 extends the dependency model with conservative connections across an external channel. These connections are useful for recovering paths that would otherwise be broken by the standard dependency model, but they do not describe the complete semantics of the channel. The extension may therefore increase reachability without proving that every recovered path is semantically relevant to verification.

The variant families are finite. The six variant families used in this thesis cover selected design dimensions. They do not represent all possible code-guard designs. The results should therefore be interpreted as evidence about the variants studied here, under the selected host program and attack workflows. They should not be generalized directly to every self-checking protection or every application.

Chapter 6

Implementation

This chapter describes the implementation used to instantiate the design from Chapter 4 and to generate the protected samples evaluated in this thesis. The implementation is source-level. The host application is modified by manually inserting calls to guard components at selected source-code locations. The guard code is then compiled together with the application and becomes part of each generated protected executable.

The implementation should be understood as a sample-generation process rather than as the construction of one single protected binary. Each defense family is built separately and produces its own protected sample. The generated samples share the same host application, the same general guard runtime, and comparable guard placement, but they differ in the implementation of verification, guard organization, chaining, and syscall-mediated state transfer.

The host application used in the experiments is `bzip2`. It serves as the case study on which the protection variants and attack workflows are evaluated. The implementation discussion therefore focuses on the mechanisms that are needed to generate and analyze protected samples, namely the build configuration, the implemented components, the protected-sample generation process, the metadata layout, the runtime guard logic, and the variant-specific implementation choices.

At a high level, each protected sample is produced in two phases. First, the modified application and the guard runtime are compiled and linked into an intermediate executable that contains placeholder metadata. Second, the offline patcher, `bzguard_offline`, computes the expected integrity values for the monitored code regions and writes the patched metadata into the final protected executable. This patching step is executed externally during sample generation.

The runtime part of the protection follows the four-phase guard structure introduced in Section 4.3. At the inserted call sites, the guard code initializes the guard state, computes an integrity value over the monitored code region, verifies this value against the stored reference value, and returns the result to the surrounding application code. In the chained variants, values produced by normal guards are also consumed by paired master guards. In the syscall-mediated variants, part of this state is transported through a file-based channel before it is used again.

Item	Configuration
Host system	EndeavourOS, Arch-like distribution; Linux kernel 6.12.73-1-lts
Architecture	x86-64
Compiler	GCC 15.2.1 20260209
Optimization flags	-O2 -g -Wall -Winline -D_FILE_OFFSET_BITS=64 -I. -ffunction-sections -fdata-sections -fno-lto -fno-merge-constants
Position-independent executable	enabled; ELF type DYN, reported as PIE by file/readelf
Debug symbols in analysis binary	stripped before attack evaluation
Linking mode	libbz2 linked through the static archive libbz2.a; final executable dynamically linked against libc and the system dynamic loader
Pin version	Intel SDE 9.38.0-2024-04-18

Table 6.1: Build environment and compilation settings used to generate the protected samples.

The attack toolchain is separate from the protected samples. The Pin-based tools and Python orchestration scripts execute the generated binaries under instrumentation, collect code-reading instructions and dynamic dependencies, compare clean and tampered executions when required, and produce the ranked candidate instructions used in the evaluation.

6.1 Build Environment and Compilation Settings

All protected samples were built under the same compilation environment to ensure comparability and reproducibility across the experiments. This is important because the attack workflow operates at instruction level. Consequently, the compiler, optimization level, position-independent code generation, linking options, and stripping can affect the generated instruction layout and the module-relative offsets observed by the analysis.

Table 6.1 summarizes the build and analysis environment used in the experiments.

6.2 Implemented Components

The implementation consists of runtime components that are compiled into the protected samples and external components used during sample generation or attack analysis. Table 6.2 summarizes these components.

Component	Language	Role
Guard control logic	C	Implements initialization, attestation, verification, and response integration at the inserted call sites.
Checksum routines	C	Compute the integrity values used by the guards during attestation.
<code>bzguard_offline</code>	C	Computes golden values and patches metadata after linking.
Pin tools	C++	Collect code-reading instructions, instruction values, comparison traces, and dynamic dependencies.
Orchestration scripts	Python	Run the experiments, organize generated artefacts, and invoke the ranking pipeline.

Table 6.2: Main implemented components of the protection and analysis toolchains.

6.3 Protected Sample Generation

This section describes the common generation workflow, from the construction of the intermediate executable to offline metadata patching and preparation of the final protected samples.

6.3.1 Intermediate Executable and Placeholder Metadata

During the first phase, the host-application source code is modified by manually inserting the guard call sites and integrating the runtime guard components, as described in Section 6.5. The modified source code is then compiled and linked to produce an intermediate executable containing the application code, the guard routines, and the metadata slots. At this stage, however, the expected integrity values stored in these slots are still placeholders. They cannot be finalized before linking because the protected regions must first be resolved in the linked code layout and their byte sequences must be available for checksum computation. The intermediate executable, therefore, provides the code layout and metadata records required by the subsequent offline-patching phase.

6.3.2 Offline Patching

The second phase runs `bzguard_offline`, a standalone post-link patcher invoked by the Makefile with the intermediate executable, the output file name, and an ordered list of function symbols identifying the regions to protect.

The patcher reads the ELF symbol table and resolves the virtual address and size of each selected symbol. It then maps each symbol range to the corresponding bytes in the executable and uses these bytes to construct the runtime metadata. The checksum and seeding rules used to populate the metadata slots are described in Section 6.4.2.

The resulting metadata is written over the placeholder blob already present in the intermediate executable. The patcher also derives and writes the encoded module-relative

offsets used by the normal and master guard paths to locate the blob at runtime. Finally, it writes the modified ELF image as the protected executable.

`bzguard_offline` is used only during sample generation and is not linked into the protected executable. The final binary therefore contains the runtime guard code and the patched metadata, but not the post-link patching tool.

6.3.3 Final Protected Samples

The output of the generation process is one protected executable for each defense variant and evaluation case. These final executables contain the modified application code, the runtime guard code, the inserted guard call sites, and the patched guard metadata.

Before attack evaluation, the final executables are stripped. This removes development symbols that would otherwise give semantic information to disassemblers and decompilers.

Unless stated otherwise, concrete assembly listings and offsets shown in this chapter are representative examples taken from one generated sample and should not be read as offsets shared by all protected samples.

Figure 6.1 summarizes the generation process used for each protected sample.

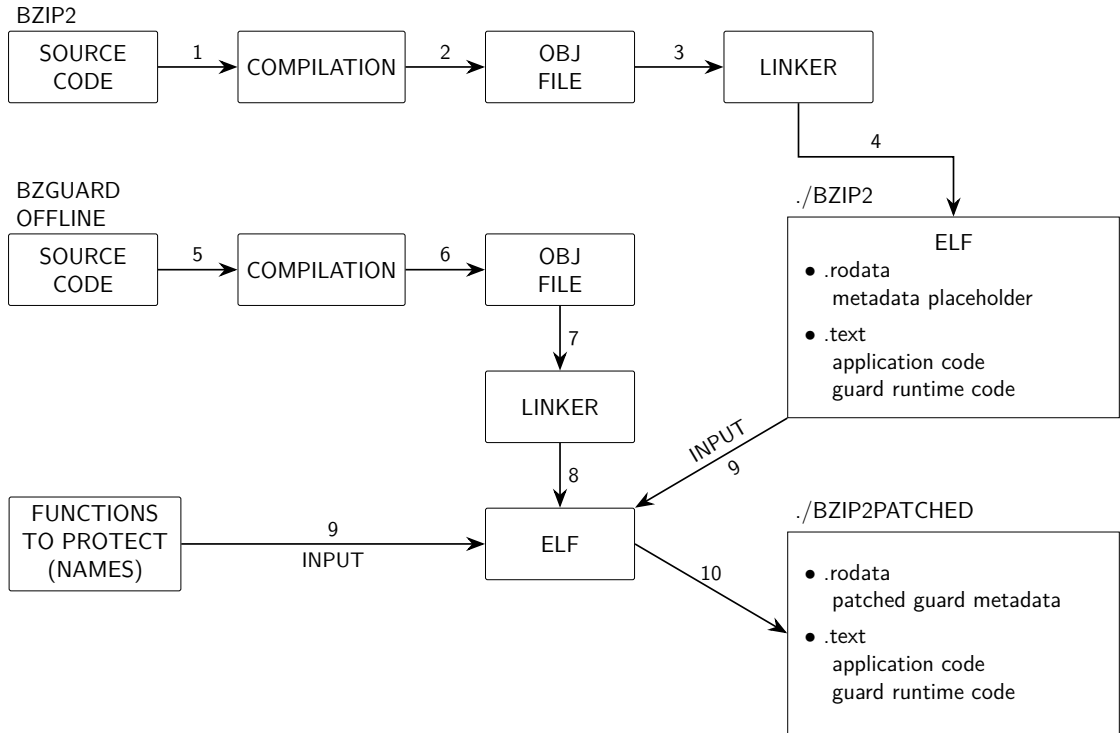


Figure 6.1: Two-phase generation process for a protected sample.

6.4 Guard Metadata and Golden Values

The reference checksums computed by `bzguard_offline` are stored in a binary blob inside the `.rodata` segment of the executable. In the unstripped development build, a dedicated pointer identifies the start of this blob. This pointer is not available in the stripped binary used in the experiments. At runtime, the guard code locates the blob through the encoded offset described in Section 6.4.3.

6.4.1 Blob Layout

The blob starts with a packed 8-byte header. The header is followed by a sequence of packed 16-byte slot records. Table 6.3 shows the fields used by the two structures.

Structure	Offset	Size	Field
Header	0	2	version, with value 1
Header	2	2	zero padding
Header	4	4	slot count
Slot record	0	8	module-relative start address
Slot record	8	4	byte count of the protected region
Slot record	12	4	expected checksum of the protected region

Table 6.3: Packed layout of the guard metadata blob.

Single-guard variant families use six slot records, one for each protected region. Chained variants add six master-guard records.

6.4.2 Checksum Seeds and Slot Population

The order of the function symbols supplied to the offline patcher determines the order and role of the records written into the metadata blob. Each slot stores the module-relative start address and byte count of one monitored region, together with the expected integrity value computed over its bytes.

In the single-guard variants, the blob contains one slot for each normal guard. The expected value for each slot is computed using the initial seed assigned to normal-guard attestation.

In the chained variants, the normal-guard slots are followed by the corresponding master-guard slots. The normal slots are populated first. The expected value computed for each normal slot is then used as the seed when computing the expected value of the paired master slot. The ordering of the entries therefore establishes the correspondence between each normal guard and its master guard. This is the offline counterpart of the runtime value propagation described in Section 6.7. The checksum routines and seed interpretation used for these computations are described in Section 6.5.1.

Most monitored regions correspond to host-application code. Selected master-guard slots can instead refer to guard-related code, allowing the chained variants to protect parts of the normal guard implementation as well as the original application.

Metadata entries are not deduplicated when they refer to the same byte range. The same region can therefore appear in multiple slots. These entries represent the same monitored code region, but remain distinct metadata records and can participate in different normal-to-master relations.

6.4.3 Runtime Blob Location

The offline patcher writes the encoded location of the metadata blob into the final executable. This location is represented as a masked module-relative offset. The stored value identifies the blob relative to the loaded executable image, which makes it suitable for position-independent execution.

The mask is applied with an XOR operation. The decoding key is derived from the module-relative address of a local guard routine. Since the guard routine and the metadata blob are part of the same executable image, the runtime code can derive the corresponding key again during execution.

At runtime, the guard recovers the loaded module base, derives the decoding key, and applies the XOR operation to the encoded value written by the patcher. This recovers the module-relative offset of the metadata blob. The guard then adds this offset to the loaded module base and obtains the runtime address used to read the metadata header and slot records.

Chained variant families use a second encoded value for the master-guard metadata lookup. The offline patcher writes this value during the same patching step. At runtime, the master-guard path decodes the second value with the same mechanism and obtains the metadata blob address from the recovered module-relative offset.

6.5 Runtime Code Guard Implementation

The runtime guard implementation is compiled into each protected sample. It contains the guard control logic, the checksum routines used during attestation, and the call sites manually inserted into the host application. No guard code is loaded dynamically or injected after linking.

The manually inserted guard call sites are kept fixed across defense families for the same guard identifier. For example, if code guard 7 is placed inside the compression routine, the corresponding guard invocation is placed in that same source-level location in each defense family that implements code guard 7. This keeps the evaluated variants comparable because differences between families come from the guard implementation, including the verification pattern, chaining, or syscall-mediated state transfer, and not from moving the guard to a different host-code location.

At each call site, the inserted code executes the guard phases required by the selected family. The initialization phase starts with a call to a slot-specific function that reads the

metadata needed to identify the protected region. It fills the attestation structure with the protected region's module-relative address and byte count. The metadata format is described in Section 6.4.

At run time, the guard reads the patched metadata slot for the selected guard. It combines the stored module-relative address with the loaded module base to reconstruct the runtime address of the protected region, and then computes a checksum over the live bytes in that range. In the prototype, the module base and executable segment bounds are obtained through dynamic-loader metadata, including runtime calls to `dladdr` and `dl_iterate_phdr`. This is a robustness-oriented implementation choice rather than a requirement of the guard model, and a stealthier implementation could avoid these calls by relying on PC-relative references and statically patched offsets.

During verification, the computed checksum is checked against the expected value stored in the metadata slot. The concrete instruction pattern depends on the selected defense variant. Some variants use a direct comparison, while others encode the same decision through arithmetic or logical instructions. These patterns are shown in Section 6.8.2.

The verification function returns a one-or-zero integer flag to the inserted call site. The surrounding application code then uses this flag to decide whether to continue normal execution or trigger the local response.

The guard uses two checksum routines, described in Section 6.5.1.

6.5.1 Attestation Checksum Routines

The attestation phase computes an integrity value over the bytes of the protected region. The implementation uses two checksum routines: normal guards use CRC32 in all six defense variants, while master guards use Adler-32 in the three chained variants.

The CRC32 routine receives the address of the protected byte range, its size, and a seed value. The checksum state is initialized by XORing the seed with `0xffffffff`. The protected region is then processed byte by byte. For each byte, the current state is first XORed with the loaded byte. The following bit-level update applies the reversed CRC32 polynomial `0xedb88320`. The conditional application of the polynomial is expressed with a mask derived from the least significant bit of the current state.

The CRC32 routine is written as an explicit state machine. The states separate byte loading, the eight bit-level update steps, the move to the next byte, and the final return. This structure is equivalent to a standard CRC32 update, but it also produces a concrete instruction pattern for the attestation computation. The returned value is the final checksum state XORed again with a specific mask set as `0xffffffff`.

The Adler-32 routine uses the state format $(b \ll 16) \mid a$. The low 16 bits of the seed initialize `a`, and the high 16 bits initialize `b`. A seed value of 1 therefore corresponds to the standard Adler-32 initial state, with `a = 1` and `b = 0`. The implementation also treats a zero seed as this same initial state. This avoids starting the computation from `a = 0`, which would not match the usual Adler-32 convention. The input is processed in blocks of at most 5552 bytes, following the usual bound used to avoid overflow before applying the Adler modulus.

For each processed byte, the Adler-32 routine adds the byte to `a` and then adds `a` to `b`. At the end of each block, both accumulators are reduced modulo 65521. The returned value packs the two accumulators back into the same 32-bit state format.

Both checksum routines are implemented directly in the guard code rather than called through library functions. Their state updates therefore appear explicitly in the generated instruction trace and can be observed by the dependency analysis.

6.6 Implemented Defense Variants

The implementation instantiates the six variant families defined in Table 4.1. Each family is generated as a separate protected sample, using the same host application and the same general guard runtime. The generated samples differ only in the implementation choices needed to realize the corresponding design change.

The baseline sample uses a direct verification check. The comparison-obfuscated sample replaces this direct check with arithmetic and logical operations. The function-diversified sample keeps the same verification style but instantiates separate verification routines for different guards. The chained samples add the normal/master relation between guards. The syscall-mediated samples keep this relation but move part of the guard state through a file-mediated channel. The state-breaking syscall-mediated samples use the same channel to create a stronger break in the dependency path tracked by the baseline analysis.

All variant families are compiled independently, so the same implementation concept can appear at different instruction offsets in different samples.

6.7 Chained and Syscall-Mediated Guards

This section focuses on the runtime details that affect the dependency analysis. First, it explains how the value produced by a normal guard is passed to its paired master guard. Then, it describes the file-based channel used by the syscall-mediated variants.

6.7.1 Normal-to-Master Seed Propagation

In the chained variants, the relevant runtime relation is the value flow from the normal attestation to the paired master attestation. The host code initialises a local variable, `crcvalue`, to zero and passes it to the normal guard attestation function. This call computes the CRC32 value of the protected bytes with seed 0.

The returned CRC32 value is then passed as the `seed` argument to master guard attestation. The master attestation uses this value as the initial state of its Adler-32 computation. This is the runtime counterpart of the slot population described in Section 6.4.2.

The implementation does not require all normal-guard phases to appear as one contiguous block before all master-guard phases. The phases can be placed at different call sites

in the host program. What matters for the chained design is that the value produced by the normal attestation is later consumed as the seed of the paired master attestation.

This value flow is relevant for the dependency analysis. If the bytes protected by the normal guard are modified, the CRC32 value changes. The changed value is then propagated to the master guard as its seed, so the master-side computation can also differ from the expected value stored in the metadata.

6.7.2 Syscall-Mediated Dependency Breaking

Without the syscall-mediated channel, guard values are propagated through ordinary user-space registers and memory. The dependency collector can therefore connect the instructions that produce these values to the instructions that later consume them. The syscall-mediated variants alter this path by writing the values to temporary files and reading them back before they are used again.

The file-mediated channel preserves the logical value flow in the program, but changes how this flow appears to the dependency collector. A value produced before `write(2)` is stored in a file and later copied into a user-space buffer by `read(2)`. The baseline dependency model does not treat the file system as a provenance carrier (Section 6.9.2).

Moreover, the modification of the destination buffer performed by `read(2)` is carried out by the kernel rather than by an instrumented user-space store instruction. Consequently, the value consumed after the read is not naturally connected to the computation that produced it before the write. The following implementations use this property to introduce different degrees of separation in the dependency path.

Chain syscall-mediated variant

In the chain syscall-mediated variant, both sides of each channel use anonymous memory mappings. On the write side, the guard value is copied into a mapped page, the page is passed as the buffer of `write(2)`, and the mapping is subsequently released. On the read side, another anonymous page is allocated, passed as the destination buffer of `read(2)`, and then read by the guard code.

Allocating a new page does not guarantee that it will have a different virtual address. The kernel can reuse the address of the mapping released by the write side. Since the dependency collector associates memory provenance with virtual addresses, a last-writer entry associated with the previous mapping can remain relevant when the address is reused. The load performed after `read(2)` can then appear to depend on the earlier store into the write-side page, even though the value was logically transported through the file.

This variant uses three independent file channels, carrying the computed checksum, the slot index, and the expected checksum.

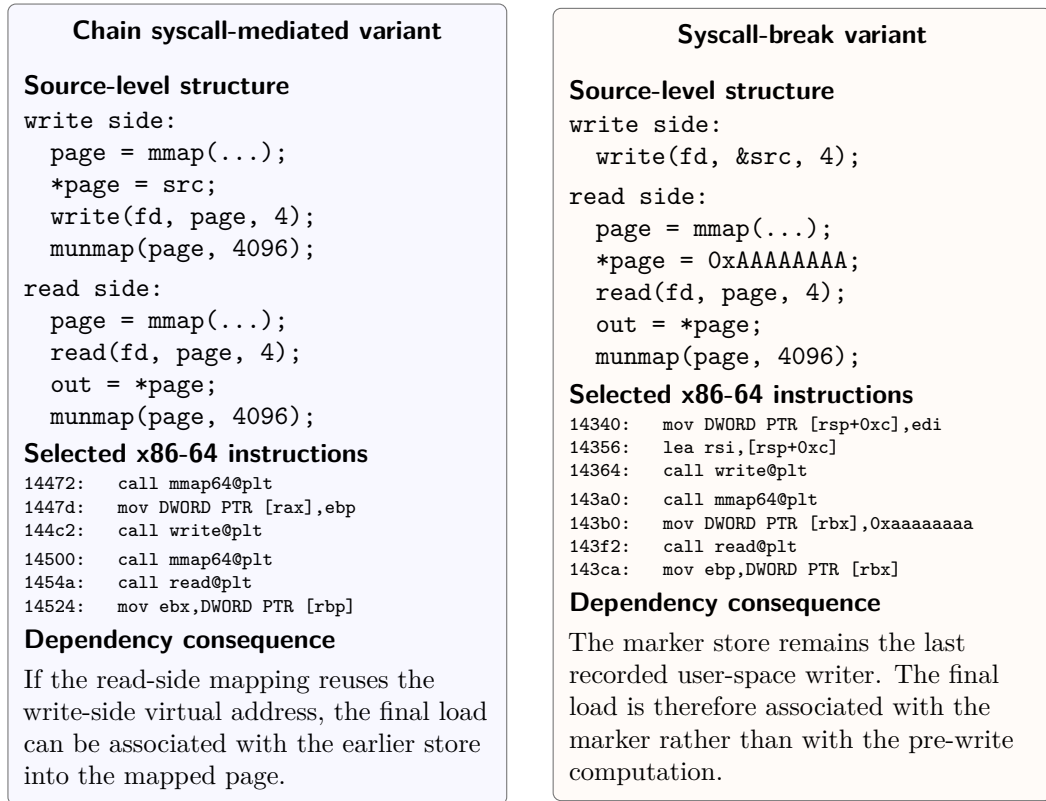


Figure 6.2: Comparison of the source-level and compiled data paths used by the two syscall-mediated variants.

Syscall-break variant

The syscall-break variant removes the mapped page from the write side. The value is instead passed to `write(2)` from a local stack location. The read side still uses an anonymous mapping, but writes a marker into the destination page before invoking `read(2)`.

The marker store becomes the last user-space writer recorded for the mapped address. As noted above, the `read(2)` write is performed by the kernel and is not recorded as an instrumented store. Consequently, the load that retrieves the value appears to depend on the marker store rather than on the pre-write computation.

The syscall-break variant uses six file channels. Three carry normal-guard values and three carry master-guard values. Within each guard class, the channels transport the computed checksum, the slot index, and the expected checksum. Each channel uses a separate file and a distinct marker byte, ranging from `0xAA` to `0xFF`, to prevent provenance left by one channel from being reused by another.

Figure 6.2 compares the source-level structure of the two implementations with the corresponding instructions generated in representative unstripped builds. Only the instructions relevant to the dependency path are shown.

6.8 Instruction-Level Evidence

This section provides representative instruction-level evidence for the runtime patterns described above. The listings use symbolic function names for readability, although these names are not available in the stripped binaries used during attack evaluation. They were obtained from the corresponding unstripped builds, whose instruction layouts match those of the evaluated binaries. The attacks identify instructions through module-relative offsets.

6.8.1 The Code-Reading Seed Instruction

The inner loop of the CRC32 routine reads the protected region one byte at a time. In the representative chained binary, the instruction at `0x14300` performs this code read by loading one byte from the address formed by the protected-region base and the current loop index. The `CodeReader` Pintool records this instruction as a seed for dependency traversal.

```
eg_crc32_algo_normal:
142f0: 89 d0      mov    %edx,%eax
142f2: f7 d0      not    %eax
142f4: 66 90      xchg   %ax,%ax
142f6: 31 d2      xor    %edx,%edx
142f8: 48 39 f2    cmp    %rsi,%rdx
142fb: 73 43      jae    14340
142fd: 0f 1f 00    nopl   (%rax)
14300: 0f b6 0c 17 movzbl (%rdi,%rdx,1),%ecx    ; code-reading seed
```

The attack toolchain identifies this instruction through its module-relative offset. The symbolic function name shown in the listing is used only for readability and is not available in the stripped binary evaluated by the attacks.

6.8.2 Verification-Critical Instructions

Each implemented variant contains a verification step. This step compares the checksum computed during attestation with the expected value stored in the metadata slot. The implementation does not use the same instruction pattern in all sample families. Table 6.4 summarises the patterns that appear in the generated binaries.

The function-diversified sample family uses the same branchless verification pattern as the comparison-obfuscated family. The difference is structural. Instead of using one shared verification routine, the implementation defines separate functions. Each routine implements the same kind of arithmetic check for a different guard slot.

The syscall-mediated chain variants keep the chained verification structure. Their main difference is not the comparison itself, but how values are passed between guard stages. This mechanism is described later in Section 6.7 and Section 6.9.

The following listings show one representative fragment for each distinct verification pattern. The offsets are module-relative offsets from the inspected binaries. The instructions

Sample family	Verification structure	Low-level pattern
Baseline direct-verification	Direct check	CMP; SETE
Comparison-obfuscated	Branchless check	XOR; NEG; OR; SAR
Function-diversified	Branchless check in per-guard functions	Same verification patterns as the comparison obfuscated family
Chained	Chained verification	XOR-based normal check; AND/SUB/SUB/OR master check
Syscall-mediated chained	Chained verification with syscall-mediated value transport	Same verification patterns as the chained family
State-breaking syscall-mediated	Chained verification with dependency-breaking channel	Same verification patterns as the chained family

Table 6.4: Verification instruction patterns across the implemented sample families.

marked as target in the listings are the ground-truth verification-critical instructions used when evaluating the rank produced by the attack workflow.

Direct comparison pattern

The baseline sample family uses a direct comparison between the computed checksum and the expected checksum. The following fragment is taken from direct comparison variant.

```
cg_verify (direct comparison):
...
1311f: 8b 54 18 14    mov    0x14(%rax,%rbx,1),%edx
13123: 31 c0          xor    %eax,%eax
13125: 39 d5          cmp    %edx,%ebp                ;   target
13127: 0f 94 c0       sete   %al
1312a: 48 83 c4 08    add    $0x8,%rsp
1312e: 5b            pop    %rbx
1312f: 5d            pop    %rbp
13130: c3            ret
```

The expected value is loaded into `%edx`. The computed checksum is in `%ebp`. The `cmp` instruction sets the zero flag, and `sete` writes the result to the return register. This is the most direct verification pattern used in the samples.

Branchless XOR pattern

The comparison-obfuscated samples avoid the direct comparison pattern. The following fragment is taken from the second and third families.

cg_verify (Comparison-obfuscated):

```
...
13119: 48 c1 e5 04    shl     $0x4,%rbp
1311d: 33 5c 28 14    xor     0x14(%rax,%rbp,1),%ebx    ;   target
13121: 89 d8          mov     %ebx,%eax
13123: 48 83 c4 08    add     $0x8,%rsp
13127: f7 d8          neg     %eax
13129: 09 d8          or      %ebx,%eax
1312b: 5b            pop     %rbx
1312c: 5d            pop     %rbp
1312d: c1 f8 1f      sar     $0x1f,%eax
13130: 83 c0 01      add     $0x1,%eax
13133: c3            ret
```

The `xor` at `0x1311d` forms the difference between the computed and expected values. If the two values are equal, the result is zero. The following `neg`, `or`, `sar`, and `add` instructions turn this difference into a one-or-zero return value. The same pattern is used in the per-guard verification functions of `no_cmp_diff_func_sc_enable` and in the normal verification routine of the chained samples, at different offsets.

AND-based master pattern

The chained samples also include master guards. Their verification routine uses a different pattern. The following fragment is taken from chained variant.

cg_verify_master (chained variant):

```
...
1420d: 89 ea          mov     %ebp,%edx
1420f: 8b 4c 18 14    mov     0x14(%rax,%rbx,1),%ecx
14213: 21 ca          and     %ecx,%edx    ;   target
14215: 89 d0          mov     %edx,%eax
14217: 29 c8          sub     %ecx,%eax
14219: 29 ea          sub     %ebp,%edx
1421b: 48 83 c4 08    add     $0x8,%rsp
1421f: 09 c2          or      %eax,%edx
14221: 5b            pop     %rbx
14222: 5d            pop     %rbp
14223: 89 d0          mov     %edx,%eax
14225: f7 d8          neg     %eax
14227: 09 d0          or      %edx,%eax
14229: c1 e8 1f      shr     $0x1f,%eax
1422c: 83 f0 01      xor     $0x1,%eax
1422f: c3            ret
```

The `and` at `0x14213` computes an intermediate value from the computed and expected checksums. The two following subtractions compare this intermediate value with both inputs. If the inputs are equal, both subtraction results are zero. Their OR combines the two results into a single difference value. The final instructions convert this value to one

for a match and zero for a mismatch. This pattern is used by master guards in the chained sample families.

6.9 Data-Dependency Collection and Attack Workflow Implementation

The attacks treat `bzip2Patched` as an observed binary. The binary is executed under Pin. The Pintool passes and the orchestration scripts are not linked into the protected program.

This section focuses on the parts of the attack toolchain that were implemented or adapted for the experiments. The two main parts are the `CodeReader` Pintool and the extensions to the `DataDependencies` Pintool. The files generated by the workflows are described later in Section 6.10.

6.9.1 Code-Reading Instruction Collection

`CodeReader` was implemented to find instructions that read executable bytes at runtime. The Pintool instruments memory-read operands and keeps the reads that fall inside executable regions of the main binary. The aim is not to record every memory read. The aim is to identify instructions that inspect code bytes and can be used as seeds for dependency analysis.

The tool uses decoded instruction boundaries from the main executable. A read is classified as a code read only when the read address falls inside decoded instruction bytes. This avoids treating every access to an executable page as a code read. The tool also records whether a read starts at an instruction boundary or in the middle of an instruction. Misaligned code reads are not used as seeds in the attack workflows.

Reader instructions are identified by image name and module-relative offset. These identifiers are stable across executions of a PIE binary. Absolute instruction addresses are not used as the main join key because they can change between runs. The read zones are represented in the same way, using module-relative start and end offsets.

The output of this pass is used to build the seed set. A seed is a code-reading instruction in the main binary. Later stages pass these seeds to `DataDependencies`, which computes the instructions that are dynamically reachable from them through data flow.

6.9.2 Baseline Data-Dependency Collection

`DataDependencies` reconstructs dynamic RAW dependencies from one execution. When an instruction writes to a register or to a user-space memory location, the tool records that instruction as the last writer of that location. When a later instruction reads the same register or memory location, the tool records a dependency from the reader to the last writer.

The output is split into register dependencies and memory dependencies. Together, these dependencies form the dynamic graph used by the attack scripts to compute forward slices from the code-reading seeds. Its trace-based scope is discussed in Section 6.11.

The baseline collector works for values that remain inside user-space registers and memory. It does not model the file system as a provenance carrier. If a value is written to a file with `write(2)` and later read back with `read(2)`, the baseline graph does not naturally connect the pre-write producer to the post-read consumer. This limitation is relevant for the syscall-mediated guard variants described in Section 6.7.

6.9.3 Shortcut Dependencies and Seed Preservation

The dependency collector is executed with shortcut handling enabled. In this mode, MOV-like transfer instructions are treated as transparent. A later consumer is related to the instruction that produced the copied value, rather than to the copy instruction itself. This reduces the number of copy nodes in the dependency graph.

This behaviour is useful for ranking, but it creates a problem for this thesis. Some code-reading instructions reported by `CodeReader` are themselves MOV-like instructions. If shortcut handling bypasses those instructions, the analysis loses the seed from which the forward slice should start.

To avoid this, `DataDependencies` was extended with a `seed_file` input. The file contains the image name and module-relative offset of the seed instructions selected from the `CodeReader` output. When shortcut handling is enabled, the dependency collector checks whether the current instruction is one of these seeds. Non-seed MOV-like instructions still use shortcut semantics. Seed instructions keep normal write semantics and remain explicit nodes in the dependency graph.

This extension is used in all three attacks. It does not change the general shortcut behaviour of the dependency collector. It only prevents selected code-reading instructions from being bypassed. This also explains why seed instructions are treated separately from other transparent transfer instructions in the ranking metrics.

6.9.4 Syscall-Aware Over-Approximation

The second extension concerns file-mediated value flow. The standard dependency collector can observe the user-space instructions before a `write(2)` call and the user-space instructions after a `read(2)` call. It does not model the file as a storage location that carries provenance between the two calls.

Attack 3 adds a conservative over-approximation to reconnect this path. The extension has three parts. The first part links the instruction that produced the value stored in the write buffer to the `write(2)` call site. The second part links observed `write(2)` sites to observed `read(2)` sites. The third part treats the `read(2)` call as the producer of the bytes that are later consumed from the read buffer.

Together, these links reconstruct the path needed by the syscall-mediated samples:

$$\text{producer} \rightarrow \text{write}(2) \rightarrow \text{read}(2) \rightarrow \text{consumer}$$

This is not an exact file-system model. The implementation does not distinguish write-read pairs by file descriptor, pathname, thread identifier, or file offset. A read can therefore be connected to writes that are not part of the same logical channel. This can introduce additional candidates. The purpose of the extension is to restore reachability across the file-mediated channel for the attack model, not to provide precise file provenance.

The three over-approximation parts are exposed as independent options in the dependency module. In the Attack 3 implementation, they are enabled together through a convenience option.

6.9.5 Attack Pipelines

Attack 1 uses one execution of the tampered binary. **CodeReader** identifies code-reading seeds. **DataDependencies** computes forward dependencies from those seeds. The candidate sinks are comparison and test instructions from the main binary. **CmpTrace** records the runtime behaviour of the reached comparison and test sites.

Attack 2 uses a good execution and a tampered execution. **CodeReader** is run on both executions. The same is done for **InstructionValues**, which records the runtime values observed at executed instructions. The two value logs are compared to select seeds whose behaviour differs between the two runs.

DataDependencies is then run with the selected seeds on both executions. The later comparison is driven by the tampered execution. The implementation compares the values reached from the tampered slice with the corresponding values from the good execution and keeps the part of the slice that leads to divergent behaviour.

Attack 3 follows the same differential structure as Attack 2, but enables the syscall-aware over-approximation in **DataDependencies**. This is the only difference between the two differential attack implementations at the dependency collection level.

6.9.6 Ranking Report Generation

Once the Pintool outputs, forward slices, divergence reports, and metric features have been written to the workspace, the ranking module reads **metrics_report.csv** and produces **ranking_report.csv**. This module is part of the attack toolchain; it does not collect new runtime data.

The ranking step is parameterised by the attack identifier. For Attack 1, the candidate rows in **metrics_report.csv** correspond to comparison and test sink instructions reachable from code-reading seeds. For Attacks 2 and 3, they correspond to instructions present in the divergent dependency slices. The score components and their combination follow the formulation described in Section 5.

Seed multiplier. The module computes a **CodeReaderMultiplier** for each seed. In automatic mode, the multiplier is derived from the **CodeReader** artefacts. When **-tampered-offset** and **-tampered-size** are supplied, the module checks whether the code region

read by each seed overlaps the modified byte range. Seeds that read bytes from this range receive a higher multiplier than seeds that do not.

This prioritization assumes that the attacker knows the location and size of the modification they introduced. It does not assume knowledge of the guard responsible for detecting the modification or of its verification-critical instruction. The attacker can only prioritize code-reading instructions that observe the modified bytes.

If no seed overlaps the tampered range, the module falls back to the `num_bytes_read` field in the `CodeReader` output and assigns higher priority to seeds that read a larger amount of executable code.

Manual mode. When `-manual-max-seed-offset` is provided, the seed at the specified offset receives the maximum multiplier and all other seeds are placed in a lower priority class. This mode supports controlled experiments in which the seed choice is fixed externally, for instance to evaluate what rank is achieved when a specific seed is known in advance. It is not the default mode, and results produced under this option are reported separately as `ranking_report_MANUAL.csv`.

Ground-truth annotation. The option `-target-offset` is used to annotate the output, not to influence scoring. After the ranking scores have been computed and the rows ordered, each row is checked against the expected verification-critical instruction offset for that run. The row whose candidate instruction offset matches is marked as the target. This check occurs after scoring and does not affect seed selection, dependency traversal, or any score component.

The ranking report also records tie information for each row: the best rank, the worst rank, and the expected rank within any group of rows that share the same final score. This information supports a fair evaluation when the target instruction is not uniquely ranked.

6.9.7 Summary of the Attacks

Table 6.5 compares the three attack implementations.

Attack	Executions	Main collected data	Extra feature
Attack 1	1, tampered	Code-reading seeds, forward dependencies, comparison behaviour	CmpTrace integration
Attack 2	2, good and tampered	Divergent values, selected seeds, forward dependency slices	Differential value comparison
Attack 3	2, good and tampered	As Attack 2, with syscall-bridged dependencies	Syscall over-approximation

Table 6.5: Comparison of the three attack implementations.

6.10 Generated Analysis Artefacts

Each attack run writes its outputs into a dedicated workspace directory. The workspace is the persistent record of the run. It contains the raw Pintool outputs, the derived slice and divergence artefacts, the metric inputs, and the final ranking files. These artefacts are produced by the external analysis workflow that observes the protected binary; they are not part of the protected binary itself.

The CSV files are the primary artefacts of the analysis. They can be inspected directly, processed automatically, and imported into **Neo4j** for interactive graph inspection. The graph view was used during development to check whether the observed dependency structure matched the expected attack behaviour. The ranking results, however, are derived from the CSV artefacts, not from the interactive graph database.

Table 6.6 summarizes the main artefact groups produced by the attack workflows.

6.10.1 CodeReader and Seed Artefacts

The **CodeReader** artefacts identify the instructions that read bytes from executable code regions and the code zones accessed by those instructions. The attack scripts derive seed files from these outputs. Each seed entry contains an image name, a module-relative offset, and a `seed_id`, which links the seed to the per-seed slice files generated later.

The seed file is also passed to **DataDependencies**, as required by the seed-preservation rule of Section 6.9.3.

6.10.2 Dependency and Differential Artefacts

The **DataDependencies** artefacts are the raw dependency graph used by the attack scripts. Register and memory dependency files encode dynamic read-after-write relations observed during the analyzed execution. The graph is then traversed from the selected seeds to build the forward slices used by the attack workflows.

The differential attacks also produce divergence artefacts. These files record which instruction values differ between the clean and tampered executions and which parts of the forward slices remain relevant after applying that information. Attack 3 produces the same classes of artefacts as Attack 2, but its dependency graph can contain additional syscall-aware edges that bridge the `write(2)` and `read(2)` boundary described in Section 6.9.4.

6.10.3 Attack-Specific and Ranking Artefacts

Attack 1 adds artefacts for comparison and test instructions. The candidate sink list is obtained from the disassembly of the main binary, while the influence edges record which candidate sinks are reachable from which code-reading seeds. The comparison trace then provides the runtime operand observations used by the Attack 1 metrics.

The ranking artefacts are the final outputs used in the evaluation. `metrics_report.csv` stores the per-candidate feature table, while `ranking_report.csv` stores the ranked

Artefact group	Example files	Role
CodeReader outputs	<code>codereader.instructions.csv</code> <code>codereader.read_zones.csv</code> <code>codereader.target_instructions.csv</code>	Identify code-reading instructions and the code zones read at runtime
Seed files	<code>codereader.seed_instructions.csv</code> <code>forward_slices/seeds.csv</code>	Define the entry points for dependency traversal and preserve selected seeds under shortcut semantics
DataDependencies outputs	<code>data_dependencies.register_dependencies.csv</code> <code>data_dependencies.memory_dependencies.csv</code>	Store dynamic register and memory read-after-write dependencies
Syscall trace	<code>data_dependencies.syscall_instructions.csv</code>	Record the tracked system-call sites used by the syscall-aware extension
Attack 1 reachability	<code>all_cmp_test_sinks.csv</code> <code>influences.edges.csv</code> <code>cmptrace.cmp-trace.csv</code>	Connect code-reading seeds to comparison and test candidates
Per-seed forward slices	<code>slice_seed_NNNNN_MMMMM.csv</code> <code>slice_seed_NNNNN_MMMMM_divergent.csv</code>	Store dependency chains from each seed and their divergent subset
Divergence artefacts	<code>divergence_report.csv</code> <code>forward_slices_divergent_nodes.csv</code> <code>forward_slices_divergent_edges.csv</code>	Record value differences between clean and tampered executions
Ranking artefacts	<code>metrics_report.csv</code> <code>ranking_report.csv</code> <code>ranking_report_MANUAL.csv</code>	Store the feature table and the final ranked candidate list

Table 6.6: Main artefact groups produced by the attack workflows.

candidate list derived from those features. When present, `ranking_report_MANUAL.csv` is an auxiliary report produced by the manual seed mode described in Section 6.9.6. It is not treated as the primary ranking result unless the evaluation explicitly states that this mode is used.

6.11 Evaluation Targets and Workflow Scope

This section defines the implementation-level elements needed to interpret the evaluation. It describes where controlled tampering points are placed, how the verification-critical target instruction is represented, and which limits follow from the trace-based attack workflow.

6.11.1 Controlled Tampering Points

The evaluated tampering locations are selected inside code regions monitored by the generated guards. In the single-guard variants, these regions correspond to host-application routines reached by the compression or decompression scenarios. In the chained variants, master-guard slots can also monitor guard-related code. In the implemented sample set, two master-guard slots protect the normal-guard checksum routine, allowing the evaluation to include modifications to guard code as well as to the original application.

A two-byte `xchg %ax,%ax` instruction was inserted into each monitored region as a stable patch point. This instruction acts as a NOP and therefore does not affect the normal behaviour of the protected routine. Tampering was applied to these predefined locations rather than to existing application instructions, so that any resulting response could be attributed to the integrity check rather than to an unrelated malfunction caused by modifying the original program logic.

Each evaluation case associates one controlled patch point with a specific metadata slot. The selected byte modification changes the contents of the monitored region while preserving execution long enough for the corresponding guard to perform attestation and verification. For example, in the representative chained binary, the patch point inside the normal CRC32 routine begins at module-relative offset `0x142f4`. The code-reading instruction at `0x14300`, described in Section 6.8.1, reads this location as part of the attestation computation.

The byte modifications were applied using `hexedit`. Each patch point was also confirmed to be reached during the compression or decompression scenario used for the corresponding experiment. This is required because the trace-based workflow can observe code-reading seeds, dependency paths, and verification behaviour only for guards executed during the recorded run.

6.11.2 Ground-Truth Representation

Each experiment is associated with a ground-truth target instruction. This is the verification-critical instruction that the attacker is expected to locate for the selected guard and defense variant. The target is represented as a module-relative instruction offset, rather than as an absolute runtime address. This representation is needed because `bzip2Patched` is compiled as a position-independent executable, so absolute addresses can change between executions.

The offsets are variant-specific. Each defense variant is compiled and linked independently, so the same verification pattern can appear at different offsets in different binaries. For this reason, the implementation records the expected target offset for each guarded case.

The ground-truth offset is not used to guide the attack. As described in Section 6.9.6, it is applied only after the candidate ranking has been produced by the researcher to annotate the row that corresponds to the expected verification instruction.

6.11.3 Implementation Limitations

The workflow is trace-based. The dependency graph contains only dependencies observed during the recorded execution. Code paths that are not taken by the selected input do not appear in the collected graph, and guard phases that are not executed cannot contribute candidates to the ranking.

The candidate set is limited to instructions in the main protected binary as a design choice. This matches the goal of the thesis, which is to estimate the effort needed to locate verification instructions inside `bzip2Patched`, not inside external libraries.

The dependency graph is an approximation of runtime data flow. Shortcut handling reduces copy noise by treating MOV-like transfers as transparent, while seed preservation keeps selected code-reading instructions visible in the graph. This makes the graph useful for the implemented attacks, but it does not make it a complete semantic model of the program.

The syscall-mediated variants expose a limitation of the original dependency collector by preserving logical value flow through temporary files while breaking the user-space provenance tracked by the baseline analysis. Attack 3 restores reachability through the syscall-aware over-approximation described in Section 6.9.4, but does not model file provenance precisely.

Values routed through temporary files could also be observed by an attacker who monitors the file system during execution. This case is outside the evaluation.

These limits define the interpretation of the results. A reported rank is the position of the ground-truth verification instruction within the candidate set produced by the implemented attack workflow for one concrete execution, input, and defense configuration.

Chapter 7

Evaluation and Validation

The previous chapters defined the protected variants, the attack workflows, and the ranking-based measure used to estimate inspection effort. This chapter has two related goals. First, it evaluates the implemented workflows by applying them to the protected variants and by comparing the resulting candidate rankings. This evaluation measures target presence, target rank, and candidate-set size across guard families and attack strategies.

Second, the chapter validates the interpretation of these rankings. The ranking workflow does not automatically prove that a candidate instruction is verification-critical. Validation is therefore used to check, through ground-truth inspection and runtime experiments, that the accepted target indeed controls the guard outcome and that the syscall-aware extension used by Attack 3 restores the intended reachability across the syscall-mediated channel.

7.1 Evaluation Objectives and Assumptions

The evaluation is based on the protected variants produced for this work, on the three attack workflows described in Chapters 4 and 6, and on the ranking measures defined in Chapter 5. Each workflow is executed on the evaluated configurations, and the resulting rankings are analyzed in terms of target presence, target rank, candidate-set size, and the reliability of observed target presence in cases where the dependency model introduces imprecise propagation. These observations are used to determine whether the workflows and ranking measures capture meaningful differences between the evaluated guard-variant families.

The interpretation of these results relies on a controlled experimental setting. The evaluated binaries are different protected versions of the same base program, so the comparison focuses on the effect of the guard design. Ground truth is used only after the rankings have been computed, in order to interpret the output of the workflows without affecting candidate selection or scoring.

Guard family	# of Code guards	Cases	Runs
Baseline direct-verification	6	6	18
Comparison-obfuscated	6	6	18
Function-diversified	6	6	18
Chained	12	18	54
Syscall-mediated chained	12	18	54
State-breaking syscall-mediated	12	18	54
Total		72	216

Table 7.1: Overview of the evaluation coverage. The *Guards* column reports the number of code-guard instances considered for each guard family. Each evaluation case is executed with Attack 1, Attack 2, and Attack 3.

7.2 Evaluation Coverage

Table 7.1 summarizes the evaluation cases covered in this chapter. The evaluation covers six guard families: baseline direct-verification, comparison-obfuscated, function-diversified, chained, syscall-mediated chained, and state-breaking syscall-mediated.

The baseline direct-verification, comparison-obfuscated, and function-diversified guard families are evaluated on six code-guard instances, from code guard 1 to code guard 6. The chained, syscall-mediated chained, and state-breaking syscall-mediated guard families are evaluated on the same six instances and on an extended set from code guard 7 to code guard 12.

The extended set corresponds to master guard instances. For these instances, two modification cases are considered. In the code-region case, the modification is applied directly to the code region protected by the master guard under evaluation. In the seed-propagation case, the modification is instead applied to the code region protected by the paired normal guard. The attacker is assumed to patch the normal guard by updating its expected value according to the modified protected region, so the normal guard itself still succeeds. However, this patch also changes the value propagated as seed to the master guard. Since the expected value of the master guard still corresponds to the original seed, the master guard detects the mismatch.

This gives 6 evaluation cases for each of the first three guard families and 18 evaluation cases for each of the three chained or syscall-related guard families. In total, the evaluation covers 72 family-case pairs. Since each pair is evaluated with Attack 1, Attack 2, and Attack 3, the complete coverage consists of 216 automatic ranking runs.

7.3 Ranking Measures and Evaluation Results

This section reports the rankings produced for the evaluation cases defined in Section 7.2. Each table reports two values for every evaluation case and attack workflow: the rank of

Code guard	Attack 1	Attack 2	Attack 3
CG 1	10/72	5/364	5/364
CG 2	55/118	5/586	5/586
CG 3	61/207	5/1836	5/1804
CG 4	7/75	2/379	2/379
CG 5	26/84	2/422	2/422
CG 6	55/247	2/1651	2/1452

Table 7.2: Target-rank results for the baseline direct-verification guard family.

the verification-relevant target and the total number of candidates. A cell has the form *rank/total*. If the target is absent from the candidate set, the rank is replaced by NN.

The interpretation of the results must take into account that the same program is evaluated through two input-dependent execution traces. Code guards 1, 2, 3 and the corresponding extended cases for code guards 7, 8, 9 belong to the compression trace, while code guards 4, 5, 6 and the corresponding extended cases for code guards 10, 11, 12 belong to the decompression trace. This distinction makes the comparison more precise, because the results are interpreted both within each trace and across the two traces.

A further trace-level effect comes from the way dynamic dependencies are represented in the ranking workflow. The dependency data are collected during the complete execution trace and then used to build an instruction-level dependency graph. In this graph, nodes correspond to static instruction offsets rather than to individual dynamic executions of those instructions. As a result, when the same guard logic is executed multiple times in one trace, dependencies observed in different dynamic invocations can contribute to the same instruction-level node.

This matters for later code guards within the same trace. A later guard is not assigned more candidates merely because more instructions have executed before it. Rather, the additional executed guard invocations can add dependency edges to the instruction-level graph when they reuse the same seed, checksum, or verification instructions. The forward traversal from the selected seed can therefore reach a larger set of candidate instructions. This aggregation effect explains why later guards in the same trace can have larger candidate sets and, in some cases, different target ranks.

7.3.1 Baseline Direct-Verification

Table 7.2 reports the target-rank results for the baseline direct-verification guard family. The rows correspond to the six evaluated code guards, and the columns correspond to the three attack workflows. The table therefore compares, for this guard family, whether each attack reaches the verification-relevant target and how early the target appears in the corresponding ranking.

All three attacks include the target for all six code guards. This is expected for the baseline direct-verification guard family, where the verification decision is implemented through an explicit comparison instruction.

Code guard	Rank	Depth	Hit count	Tuple count	Failure	Final score
CG 1	10	5	2	1	1	2.48713694
CG 3	61	5	4	3	1	2.17373655

Table 7.3: Attack 1 target-row metric values for CG 1 and CG 3 in the baseline direct-verification guard family.

The candidate counts are affected by the instruction-level aggregation effect described in Section 7.3. The increase is not caused simply by the fact that more instructions execute before a later guard is reached. Rather, the ranking workflow builds an instruction-level dependency graph from dynamic dependency observations. When the same guard logic is executed multiple times in the same trace, dependencies observed in different dynamic invocations can be associated with the same static instruction offsets. As a result, the forward traversal from the selected seed can reach more non-target instructions in later code-guard cases.

For Attack 1, the change in rank is also affected by the scoring of the target instruction itself. Table 7.3 gives a diagnostic comparison of the target row for CG 1 and CG 3. Unlike the previous table, this table does not summarize a complete ranking. Instead, it reports the metric inputs observed for the target instruction and the final score assigned to that target row. The *Depth*, *Hit count*, *Tuple count*, and *Failure* columns are the values used by the Attack 1 scoring function. The *Final score* column is the resulting score after applying the metric combination defined in Chapter 5.

The target has the same dependency depth in both code guards, but it is observed with a larger hit count and a larger number of operand tuples in CG 3. Since the Attack 1 scoring function penalizes larger hit counts and larger tuple diversity values, the target receives a lower final score in CG 3 and appears later in the ranking.

Attacks 2 and 3 produce more stable target ranks for this guard family. The target is present for all six code guards and appears at rank 2 or rank 5. Their candidate sets are larger than those of Attack 1, but the differential signal gives the verification-relevant comparison a stronger position in the ranking.

These results define the reference point for the rest of the evaluation. All three attacks reach the target without difficulty, and the rank differences only reflect how the scoring functions behave, not any actual resistance from the guard. The following sections can be read against this case, which corresponds to a guard offering essentially no protection.

7.3.2 Comparison-Obfuscated

Table 7.4 reports the results for the comparison-obfuscated family. In this guard family, the verification step is no longer exposed as the direct comparison pattern used in the baseline family.

In this guard family, Attack 1 does not report the target in any of the six cases. This follows from the way its candidate set is built. Attack 1 ranks only comparison-like sink

Table 7.4: Target-rank results for the comparison-obfuscated family.

Evaluation case	Attack 1	Attack 2	Attack 3
CG 1	NN/71	7/368	7/368
CG 2	NN/117	31/594	31/594
CG 3	NN/206	63/1860	63/1812
CG 4	NN/74	3/383	3/383
CG 5	NN/83	31/426	30/426
CG 6	NN/246	49/1645	49/1440

Case	Rank	Paths through target	Total paths	Final score
CG 1	7	15	520	2.63505334
CG 2	31	19	973	2.62573442
CG 3	63	17	2785	2.61231132

Table 7.5: Attack 2 target-row path-centrality values for the comparison-obfuscated family.

instructions, whereas the verification decision is encoded through an `xor`-based pattern. The target instruction therefore does not enter the Attack 1 candidate set.

Attacks 2 and 3 include the target in all six cases. Their results are near-identical because this guard family does not use a syscall-mediated channel. The target instruction is the same `xor` in all six cases. It has depth 5, satisfies the divergence condition, and has outgoing dependencies in the induced graph. The main scoring difference from the baseline family is the opcode contribution: `xor` receives *OpcodeScore* 0.85 instead of the value 1.0 assigned to `cmp`. This already places the target at a disadvantage with respect to comparison and test instructions that receive comparable scores on the other metrics.

The variation between the six cases is mainly explained by path centrality. Table 7.5 shows the target row under Attack 2 during the compression trace.

The instructions ranked before the target should not be interpreted as instructions that are necessarily closer to the final verification decision. Some of them belong to earlier checksum or propagation code. Others are comparison-like instructions that receive a higher *OpcodeScore* than the `xor` target. Representative cases are examined in more detail in the following section.

This guard family therefore shows that the differential attacks recover the target after the direct comparison has been removed, but the final position still depends on how central the target is in the observed dependency graph.

The gap between Attack 1 and the differential attacks in this guard family points to a weakness in how Attack 1 selects candidates. Because it looks for comparison-like instructions, replacing `cmp` with `xor` is enough to remove the target from its results. The differential attacks are not affected by this change, since they rely on observed dependencies rather than on the opcode used.

Evaluation case	Attack 1	Attack 2	Attack 3
CG 1	NN/71	3/187	3/187
CG 2	NN/142	3/270	3/270
CG 3	NN/275	3/376	3/376
CG 4	NN/81	3/446	3/446
CG 5	NN/104	3/549	3/549
CG 6	NN/762	3/1188	3/1135

Table 7.6: Target-rank results for the function-diversified family.

7.3.3 Function-Diversified

Table 7.6 reports the results for the function-diversified family. This guard family keeps the comparison-obfuscated verification style but changes the placement of the verification logic.

The pattern is close to the comparison-obfuscated family. Attack 1 never includes the target, while Attacks 2 and 3 include it in all six cases. The ranks under Attacks 2 and 3 are low in every case.

Compared with the previous guard family, function diversification does not prevent the differential attacks from reaching the target. The main effect is that the ranking remains stable across attacks.

In this guard family, the smaller candidate sets and earlier target ranks depend on the implementation structure of the variant. As described in Section 4.6, in the function-diversified family, each guard is implemented through a separate code path. As a result, the data-dependency graph built during the attack is more restricted. The seed is connected to fewer unrelated instructions, thus reducing the number of non-target candidates that can compete with the target and making the dependency path to the divergent instruction clearer.

Splitting the verification logic across separate functions does not make the target harder to find. If anything, the results suggest the opposite, in fact the dependency graph built by the attacker becomes smaller and more focused, and the target is reached at a lower rank than in the comparison-obfuscated family.

7.3.4 Chained

Table 7.7 reports the results for the chained family. This guard family extends the evaluation to the master-guard cases, so the table contains both the first six code-guard instances and the extended cases for CG 7–CG 12.

A consistent difference appears between the code-region and seed-propagation cases for CG 7–CG 12. In each pair, the seed-propagation case has a larger numeric rank than the corresponding code-region case. This difference is not only due to the number of candidates, but also on how the modified region is associated with the CodeReader instruction and on the dependency path that reaches the target.

Evaluation case	Attack 1	Attack 2	Attack 3
CG 1	NN/196	46/1145	46/1145
CG 2	NN/324	64/1778	64/1778
CG 3	NN/484	83/3904	83/3868
CG 4	NN/154	34/933	34/933
CG 5	NN/178	40/1047	40/1047
CG 6	NN/506	53/1755	53/1562
CG 7 code-region	NN/210	51/1183	51/1183
CG 7 seed-propagation	NN/210	75/1223	75/1227
CG 8 code-region	NN/348	65/862	65/862
CG 8 seed-propagation	NN/324	247/1780	245/1780
CG 9 code-region	NN/484	87/1931	87/1913
CG 9 seed-propagation	NN/484	117/3892	117/3852
CG 10 code-region	NN/200	41/1141	41/1137
CG 10 seed-propagation	NN/200	66/1160	66/1156
CG 11 code-region	NN/215	45/1241	45/1235
CG 11 seed-propagation	NN/215	69/1256	69/1250
CG 12 code-region	NN/506	53/1745	53/1550
CG 12 seed-propagation	NN/506	78/1781	78/1587

Table 7.7: Target-rank results for the chained family.

Table 7.8 shows the target rows for CG 7. The same target instruction appears with two code reader offsets. Offset 82592 is the code reader instruction used by the master guard, while offset 82688 is the code reader instruction used by the paired normal guard. This happens because the chained implementation contains both values in the dependency structure. The dependencies of these two instruction offsets is discussed in more detail in Section 7.4.

In the code-region case, the modification is applied to the region protected by the master guard. The ranking therefore promotes the row associated with the master code reader instruction, which receives multiplier 1.0 and appears at rank 51. The row associated with the normal-guard code reader instruction receives multiplier 0.5 and is placed much lower.

In the seed-propagation case, the modification is applied to the region protected by the paired normal guard. Under the assumptions of the ranking workflow, this gives the stronger CodeReader association to normal code reader instruction rather than to the direct master code reader instruction. The first target row reached by the attacker is therefore associated with the normal-guard code reader instruction. The verification instruction is still the master guard verification target, but it is reached through a different code reader association.

This is a consequence of the ranking implementation and of the seed-propagation attack setting. The CodeReader multiplier promotes the code reader instruction that is more directly associated with the modified code region. In the seed-propagation case, the promoted row reaches the target at depth 10 instead of depth 4. Its priority score is therefore lower, and the target appears at rank 75 instead of rank 51.

Case	Seed	Rank	Depth	Multiplier	Priority	Final score
Code-region	82592	51	4	1.0	2.490719	2.490719
Code-region	82688	650	10	0.5	2.382804	1.191402
Seed-propagation	82688	75	10	1.0	2.382656	2.382656
Seed-propagation	82592	649	4	0.5	2.490428	1.245214

Table 7.8: Attack 2 target rows for CG 7 in the chained family.

This example shows that the seed-propagation case does not simply add more candidates. It changes which code reader association is promoted by the ranking and also changes the depth of the dependency path used for the target row. The target is still recovered by the differential attacks, but it is placed lower in the ranking. The same distinction between code-region and seed-propagation cases is also relevant for the syscall-mediated chained and state-breaking syscall-mediated families.

The code-region and seed-propagation cases show that chaining guards together does not protect the target uniformly. What changes the attack cost is which part of the chain is modified, since this determines which code reader instruction association the ranking picks up first.

7.3.5 Syscall-Mediated Chained

Table 7.9 reports the results for the syscall-mediated chained family. The structure is similar to the chained family, but the guard state is routed through a syscall-mediated channel.

Attack 1 again does not include the target in any case. Attacks 2 and 3 include the target in all 18 cases. The ranks under Attack 2 and Attack 3 are close to the chained family for several cases.

The Attack 2 results for this guard family require caution. The standard dependency model does not explicitly reconstruct the syscall-mediated channel. Target presence under Attack 2 is affected by imprecise propagation, such as stale last-writer state retained across system calls or memory-management effects. These cases are treated as apparent target presence under dependency-model imprecision rather than as precise value-flow recovery. Attack 3 provides the more appropriate interpretation for this guard family because it includes syscall-aware propagation.

Compared with the chained family, the syscall-mediated chained family produces slightly larger candidate sets in all corresponding Attack 2 rows. This increase is expected because the syscall-mediated implementation adds instructions around the state-transfer mechanism. However, the target rank is not affected in the same proportion. In several cases, the Attack 2 rank remains close to the chained result.

The presence of the target under Attack 2 in this guard family should be read with some caution, since its dependency model does not reconstruct the syscall channel and can report the target for the wrong reason. Attack 3 gives a more reliable picture here, which is the main justification for including a syscall-aware attack in this evaluation.

Evaluation case	Attack 1	Attack 2	Attack 3
CG 1	NN/196	47/1195	48/1195
CG 2	NN/324	65/1846	67/1846
CG 3	NN/484	84/3974	86/3978
CG 4	NN/154	32/986	55/1357
CG 5	NN/178	37/1108	61/1467
CG 6	NN/506	51/1789	76/1947
CG 7 code-region	NN/210	51/1233	67/1233
CG 7 seed-propagation	NN/210	75/1283	78/1291
CG 8 code-region	NN/348	65/883	73/896
CG 8 seed-propagation	NN/348	88/1818	91/1826
CG 9 code-region	NN/484	87/1951	97/1966
CG 9 seed-propagation	NN/484	117/3964	120/3972
CG 10 code-region	NN/200	41/1205	84/1541
CG 10 seed-propagation	NN/200	65/1226	95/1595
CG 11 code-region	NN/215	45/1305	87/1641
CG 11 seed-propagation	NN/215	68/1323	98/1689
CG 12 code-region	NN/506	53/1769	94/1930
CG 12 seed-propagation	NN/506	77/1815	106/1978

Table 7.9: Target-rank results for the syscall-mediated chained family.

7.3.6 State-Breaking Syscall-Mediated

Table 7.10 reports the results for the state-breaking syscall-mediated family. This guard family is designed to separate the code reader and verification instructions more strongly than the previous syscall-mediated family.

Attack 1 does not include the target in any case. Attack 2 also fails to include the target in all 18 cases. Attack 3 includes the target in every case, with ranks between 47 and 121.

Compared with the syscall-mediated chained family, the state-breaking variant shows why the syscall-aware extension is needed. In the previous guard family, Attack 2 still reports target presence, although that presence must be interpreted with caution. Attack 3 changes the candidate-generation outcome. The path-level evidence for this behaviour is discussed in Section 7.6.

This is the only guard family where Attack 1 and Attack 2 both fail to recover the target in every case, which suggests that separating the code-reading instruction from the verification instruction is an effective countermeasure against attacks that do not account for syscalls. The target is still recovered by Attack 3, so the protection only holds against a limited attacker model, not in general.

Evaluation case	Attack 1	Attack 2	Attack 3
CG 1	NN/194	NN/1141	47/1189
CG 2	NN/318	NN/1728	65/1822
CG 3	NN/476	NN/3832	85/3998
CG 4	NN/153	NN/919	55/1361
CG 5	NN/174	NN/1044	61/1461
CG 6	NN/498	NN/1735	76/1953
CG 7 code-region	NN/206	NN/1145	69/1227
CG 7 seed-propagation	NN/206	NN/1199	80/1303
CG 8 code-region	NN/340	NN/841	81/894
CG 8 seed-propagation	NN/340	NN/1712	92/1846
CG 9 code-region	NN/476	NN/1907	110/1967
CG 9 seed-propagation	NN/498	NN/3854	121/3996
CG 10 code-region	NN/192	NN/1122	86/1543
CG 10 seed-propagation	NN/192	NN/1141	97/1607
CG 11 code-region	NN/206	NN/1228	89/1643
CG 11 seed-propagation	NN/206	NN/1244	100/1701
CG 12 code-region	NN/498	NN/1737	96/1935
CG 12 seed-propagation	NN/498	NN/1766	108/1984

Table 7.10: Target-rank results for the state-breaking syscall-mediated family.

Image	Normal CodeReader offset	Instruction offset	Opcode	Depth	Multiplier
bzip2patched.exe	82688	82592	MOVZX	25	0.5

Table 7.11: Example row connecting the normal-guard CodeReader to the master CodeReader instruction.

7.4 Normal-to-Master Seed Dependency

Section 4.6 already introduced the normal-to-master seed dependency used by the chained variants. Here, the same dependency is considered only to explain why two different code reader instruction offsets are associated with the same master verification target in the ranking data.

Table 7.11 reports one dependency-row extracted from the ranking data for CG 7. The row is not the verification target itself. It is used only to show that, in the instruction-level dependency graph, the CodeReader instruction of the paired normal guard can reach the CodeReader instruction of the master guard. The *Normal CodeReader offset* column identifies the seed instruction from which the forward traversal starts. The *Instruction offset* column identifies the reached instruction. The *Depth* column gives the dependency distance between these two instructions in the graph, and the *Multiplier* column reports the CodeReader multiplier assigned to that row by the ranking workflow.

Table 7.11 shows that the traversal from the normal-guard CodeReader at offset 82688 reaches the master-guard CodeReader at offset 82592.

The assembly fragments in Figure 7.1 are not meant to show the complete generated functions. They only show the instructions that are relevant to the data-flow relation discussed here. Loop-control instructions and surrounding instructions are omitted. The left column shows the corresponding C constructs, while the right column shows the relevant x86-64 instructions generated at -O0. At this optimization level, the two C state variables are kept in distinct stack slots. The variable `seed1` is represented by `[rbp-4]`, while `seed2` is represented by `[rbp-8]`.

The first and third blocks are simplified fragments of the attestation computation. The first block belongs to the normal guard and represents the loop that updates `seed1` with the first bytes read from `a[i]`. In the generated assembly, the instruction `movzx eax, BYTE PTR a[rax]` is the CodeReader instruction for this normal-guard loop, because it reads one protected byte from the array. The loaded byte is then combined with the current value of `seed1` through the stack slot `[rbp-4]`.

The second block is the connector between the normal-guard computation and the master-guard computation. It corresponds to the source-level assignment `seed2 = seed1`. In the generated assembly, the value stored in the stack slot used for `seed1` is loaded and then stored into the stack slot used for `seed2`. This copy initializes the master-guard state with the value computed by the normal guard.

The third block belongs to the master guard and represents the later attestation loop that updates `seed2` with additional bytes read from the same array. Its code reader instruction is again a `movzx eax, BYTE PTR a[rax]` instruction, but this time the loaded byte is combined with the master-guard state stored in `[rbp-8]`.

This structure creates the dependency between the normal-guard CodeReader instruction and the master-guard code reader instruction. The term “seed” in the ranking data refers to the seed instruction of the data-dependency traversal, not to the C variables `seed1` and `seed2`. Starting from the normal-guard code reader instruction, the dependency path propagates through the update of `seed1`, crosses the `seed2 = seed1` connector, and then reaches the master-guard computation. For this reason, the ranking data can contain rows in which the code reader instruction belongs to the normal-guard, while the reached instruction belongs to the master-guard assembly fragment.

Figure 7.2 shows an alternative split form that was considered to separate the normal-to-master propagated state more strongly from the later protected-byte read performed by the master guard. Unlike the form shown in Figure 7.1, the master-guard loop is split into two parts. The byte read from `a[i]` is first stored in a temporary variable, and is only combined with `seed2` in a later merge step.

From the data-dependency point of view, this changes the role of the master-guard code reader instruction. The instruction that reads the later protected byte, `movzx eax, BYTE PTR a[rax]`, belongs to the independent temporary-value path. The normal-to-master propagated state instead reaches the later merge instruction that updates `seed2`. In other words, the dependency path starting from the normal-guard code reader crosses the `seed2 = seed1` bridge and reaches the master-guard merge, but it does not continue backward into the independent read of the temporary value.

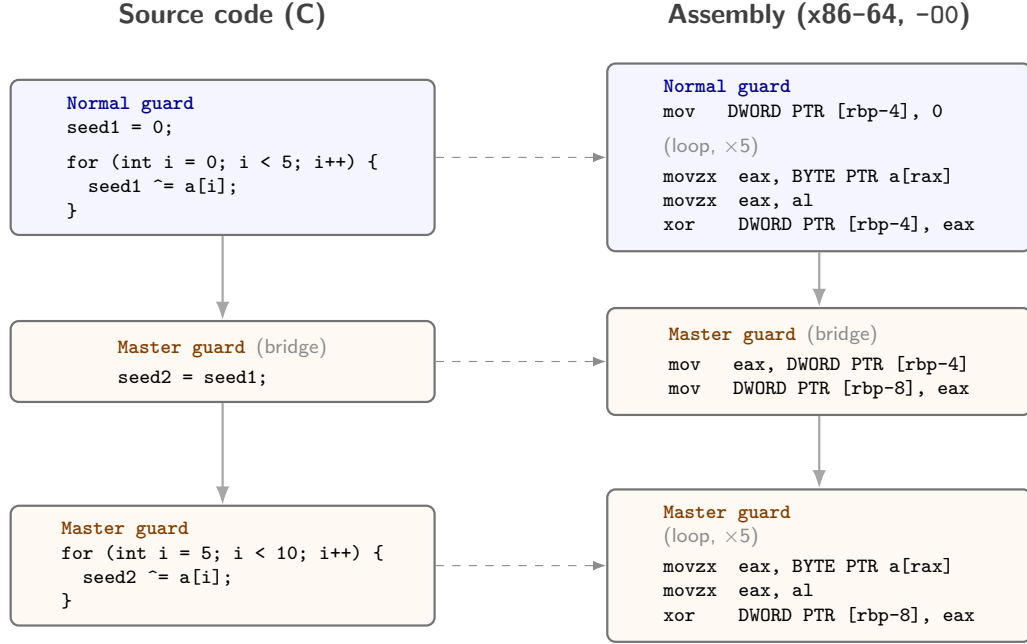


Figure 7.1: Data propagation from the normal guard to the master guard in the implemented chained form.

As a result, the ranking would not contain a row pairing the normal-guard code reader instruction with the master-guard code reader instruction. Instead, the relevant reached instruction on that path would be the master-guard merge instruction. This alternative implementation was not included in the final evaluation. In the evaluated implementation, the propagated value is used directly in the normal-to-master computation, which is why the chained and syscall-mediated chained families expose both the direct master-guard code reader instruction and the paired normal-guard code reader instruction in the ranking data.

The state-breaking syscall-mediated family must be interpreted separately. Under Attack 2, the standard dependency model does not follow the syscall-mediated transfer, so the normal-to-master relation is fully broken in the produced ranking. The target instruction is absent, and the master code reader instruction is not reached from the paired normal-guard code reader instruction. Under Attack 3, the syscall-aware propagation step reconnects the value flow across the syscall-mediated channel. In that case, the ranking expose the relation between the normal guard code reader instruction and the master guard code reader instruction.

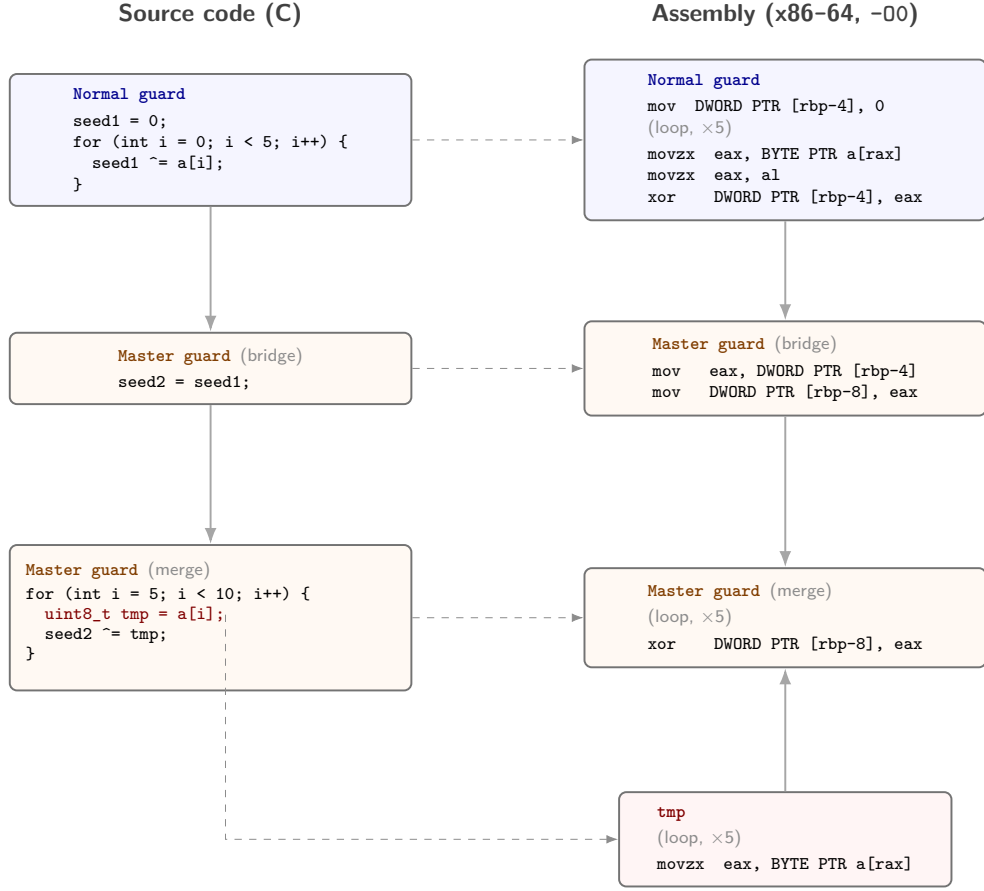


Figure 7.2: Alternative Data propagation from the normal guard to the master guard in the implemented chained form.

7.5 Representative Ranking-Based Target Validation

This section validates the operational meaning of one reported target rank. The attack workflow produces a ranked list of candidate instructions, but the scripts do not automatically prove that a candidate is the instruction an attacker would patch. The rank is meaningful only with respect to the inspection process defined in Section 4.8.1. In this way, the modeled attacker inspects the candidates in ranking order and tests them until reaching an instruction that controls the guard accept/fail outcome.

In the representative case considered here, the researcher replays that modeled inspection process manually. Each candidate ranked before the selected target is treated as a negative control. The candidate is first inspected statically to understand its role in the guard computation, and then tested dynamically by modifying its effect during execution. If the modification only corrupts intermediate checksum or propagation state, and the tampered execution still fails, aborts, or does not complete as a valid accepted execution, the

Rank	Offset	Instruction	Function
1	0x130b3	xor eax, edx	attestation
2	0x130ce	xor eax, ecx	attestation
3	0x130c7	shr ecx, 1	attestation
4	0x130de	not eax	attestation
5	0x13125	cmp ebp, edx	verify

Table 7.12: Inspected ranked candidates for the representative Baseline direct-verification / Code Guard 1 / compression case.

candidate is rejected as the verification-critical target. The inspection then continues with the next candidate in the ranked list.

The selected target is the first candidate whose modification changes the guard outcome and allows the tampered execution to proceed successfully. Therefore, the candidates ranked before it are part of the measured inspection effort. They must be inspected and rejected before the modeled attacker reaches the verification-critical instruction.

The representative case used here is the **Baseline direct-verification / CG_1 / compression** run. This case is suitable because the target rank is low, so every candidate ranked before the selected target can be inspected manually in full. The purpose of the section is to show that the reported rank is not only a numerical position in the output table, but corresponds to a concrete ranking-based target-identification process.

7.5.1 Representative ranked candidates

Table 7.12 summarizes the five inspected ranked candidates. The four earlier candidates are all part of the checksum computation used during attestation. The accepted target is the first candidate that consumes the computed checksum together with the expected value loaded from guard metadata and directly feeds the boolean verification result.

The symbolic function names in Table 7.12 are not available to the attacker during analysis of the stripped binary. They are shown only to explain the semantic role of each offset. The attack workflow itself reports only module-relative offsets.

7.5.2 Static and runtime inspection of the ranked candidates

The candidates ranked before the accepted target are inspected by combining runtime observations with static context. GDB is used first to understand how each instruction is reached during execution and which registers it uses. Ghidra is then used to place the same instruction in the surrounding function. This combination is useful because some instructions may look plausible when considered in isolation, especially `xor` instructions, but their role becomes clearer once their dynamic behaviour and static context are considered together.

Table 7.12 summarizes the five ranked candidates inspected with this combined dynamic and static procedure. The first four candidates belong to `cg_attestation_slot`. They

are part of the checksum computation over the protected byte range. The accepted target belongs instead to `cg_verify`, where the computed checksum is compared with the expected checksum loaded from guard metadata.

At rank 1, offset `0x130b3` executes `xor eax, edx`. Considered alone, this instruction is a plausible candidate for a verification-related operation. An attacker could initially suspect that the operands are the computed value and the expected value, because an XOR between equal values would produce zero. The instruction could also be suspected of producing an intermediate flag or a value later used to derive a guard decision.

Runtime inspection gives the attacker a different interpretation. The instruction is not reached as a single decision point. It is reached repeatedly while the protected byte range is processed. The register state is consistent with a checksum update rather than a guard decision. `EDX` contains the byte loaded immediately before the instruction, while `EAX` contains the current checksum state. Static context supports the same conclusion, because the instruction follows the code-reading seed at `0x130b0`:

```
001130b0  0f b6 16      MOVZX      EDX, byte ptr [RSI]
001130b3  31 d0         XOR        EAX, EDX
```

The modeled attacker therefore rejects `0x130b3` as the verification-critical target. The instruction folds the loaded byte into the checksum state. It does not consume the expected checksum and does not produce the verification result. Modifying it changes the attestation computation, but it does not make the tampered execution a valid accepted execution.

At rank 2, offset `0x130ce` executes another XOR instruction, `xor eax, ecx`. The same initial suspicion applies. An attacker could consider it a possible XOR-based check, or a step that produces a value later interpreted as a success or failure result. Runtime inspection again contradicts this interpretation. The instruction is reached repeatedly inside the same checksum computation, and its operands are temporary checksum-state values rather than a computed checksum and an expected checksum.

The static context places the instruction inside the CRC-like inner loop:

```
001130c0  89 c1         MOV        ECX, EAX
001130c2  83 e0 01      AND        EAX, 0x1
001130c5  f7 d8        NEG        EAX
001130c7  d1 e9        SHR        ECX, 0x1
001130c9  25 20 83 b8 ed AND        EAX, 0xedb88320
001130ce  31 c8        XOR        EAX, ECX
001130d0  83 ea 01      SUB        EDX, 0x1
001130d3  75 eb        JNZ        LAB_001130c0
```

The modeled attacker therefore rejects `0x130ce`. The instruction combines the shifted temporary state with the polynomial-masked value. It is part of checksum propagation, not the guard decision.

At rank 3, offset `0x130c7` executes `shr ecx, 1`. This instruction is less plausible as a direct verification point because it is a shift rather than a comparison or a flag-producing operation. The relevant question for the attacker is whether it behaves like a single decision

instruction or like a repeated update inside a computation. Runtime inspection shows that it is reached repeatedly. Static inspection places it inside the eight-iteration CRC update loop, between the copy of `EAX` into `ECX` and the later combination at `0x130ce`.

The modeled attacker therefore rejects `0x130c7`. The instruction shifts the temporary CRC state. It helps compute the value that will later be passed to the verification function, but it does not check that value against the expected checksum.

At rank 4, offset `0x130de` executes `not eax`. This candidate is closer to the end of attestation than the previous ones, so the attacker has a stronger reason to inspect it carefully. Runtime inspection shows that the instruction is reached after the byte-processing loop. Static context shows that, after all protected bytes have been processed, the checksum is finalized by inverting `EAX`.

001130d5	48 83 c6 01	ADD	RSI, 0x1
001130d9	48 39 fe	CMP	RSI, RDI
001130dc	75 d2	JNZ	LAB_001130b0
001130de	f7 d0	NOT	EAX

The modeled attacker therefore rejects `0x130de` as well. The instruction produces the final checksum value, but it still does not consume the expected checksum. At this point, the expected metadata value has not been loaded locally, and no equality result is produced. Patching this instruction changes the computed checksum output without directly forcing it to match the stored expected value. The runtime test confirms that modifying this finalized value does not produce a valid accepted execution.

The candidate at rank 5 is different. Offset `0x13125` belongs to a function responsible for verification, rather than to the attestation routine that computes the checksum. Static inspection shows that the function retrieves the guard metadata, checks that the selected slot index is valid, loads the expected checksum into `EDX`, and compares it with the attestation-derived value held in `EBP`. Table 7.12, introduced above, summarizes this data flow at the target.

0011311f	8b 54 18 14	MOV	EDX, dword ptr [RAX + RBX*0x1 + 0x14]
00113123	31 c0	XOR	EAX, EAX
00113125	39 d5	CMP	EBP, EDX
00113127	0f 94 c0	SETZ	AL

The last candidate gives the modeled attacker a direct validation test. In the natural tampered execution, `EBP` and `EDX` contain different values, so the comparison fails and `SETZ AL` produces zero. When the attacker forces `EBP = EDX` immediately before the comparison, the same instruction sequence produces one. Continuing execution with this forced successful comparison allows the tampered run to complete successfully.

The modeled attacker therefore accepts `0x13125` as the verification-critical target. This distinguishes it from the four higher-ranked candidates. The earlier instructions are relevant to the guard because they compute the attestation value, but they do not consume the expected checksum and do not produce the guard decision. They may also be ambiguous during the first inspection of the stripped binary. Without symbolic information,

Role	Offset	Opcode	Operands
Code-reading instruction	0x14a00	movzx	ecx, byte ptr [rdi+rdx*1]
Verification instruction	0x147d0	xor	edi, esi

Table 7.13: Endpoints used for the syscall-aware reachability validation.

a byte-processing loop that reads memory and updates a CRC-like state could initially be mistaken for ordinary application logic, for example code related to compression or decompression.

7.6 Validation of Syscall-Aware Reachability

This section validates the reachability claim behind Attack 3, described in Section 4.8.7. In the state-breaking syscall-mediated variants, the attestation value is produced on the write side of a syscall-mediated channel and later used by verification logic on the read side. Attack 2 loses this dependency path at the channel boundary. Attack 3 addresses this break by inserting a synthetic dependency edge between the write-side and read-side syscall events, allowing the data dependencies graph to reach the read-side verification instruction.

The representative case is taken from the state-breaking syscall-mediated family. The starting seed is offset 84480 (0x14a00). The verification-side instruction is offset 83920 (0x147d0). Table 7.13 summarizes the two endpoints used in this validation. The seed is reported by CodeReader as an instruction that reads code bytes. The second endpoint is the verification-side instruction that must become reachable after the syscall-mediated channel.

In the unstripped validation binary, the seed belongs to the attestation function that computes the CRC value on the write side. The instruction at 0x147d0 belongs to the verification function on the read side. The two slices compared below are produced from the same divergent seed and the same code guard.

The dependency path starts at the code-reading instruction. Offset 0x14a00 reads a byte from the protected range, and the following instructions update the CRC state. The path then reaches offset 0x14a40, where the computed CRC value is finalized with `not eax` before the producer returns. Up to this point, the path is ordinary user-space propagation inside `bzip2patched.exe`.

After the CRC value is produced, the value enters the syscall-mediated channel. This is the point where the baseline dependency model loses the path: the value is written through a syscall and later reappears on the read side, but the ordinary memory-provenance model does not connect these two events. Attack 3 adds the conservative connection needed to continue the traversal. The path then crosses from the write-side `libc` node to the read-side `libc` node and returns to `bzip2patched.exe` at 0x147d0, the verification-side consumer.

From	To	Kind	Depth	Phase	Presence
bzip2:0x14a00	bzip2:0x14a04	reg	1	Attestation	Attack 2 and Attack 3.
bzip2:0x14a2f	bzip2:0x14a40	reg	6	Attestation	Attack 2 and Attack 3.
bzip2:0x14a40	libc:0x11c5a2	mem	7	Channel	Attack 2 and Attack 3.
libc:0x11c5a2	libc:0x11ba8f	mem	8	Channel	Attack 3 only.
libc:0x11ba8f	bzip2:0x147d0	reg	9	Verification	Attack 3 only.

Table 7.14: Selected dependency rows showing the syscall-aware path in Attack 3.

Table 7.14 summarizes this path. In the table, **bzip2** denotes **bzip2patched.exe**. The rows before the syscall break are present in both Attack 2 and Attack 3. These rows show ordinary propagation from the code reader instruction through the write-side attestation computation. The rows after the syscall break are present only in Attack 3. These rows are the evidence that the syscall-aware bridge reconnects the write-side channel state to the read-side verification instruction.

The recovered Attack 3 path is shown below in the same form as the forward-slice output:

```

bzip2patched.exe:0x14a00
-> bzip2patched.exe:0x14a04
-> bzip2patched.exe:0x14a0d
-> bzip2patched.exe:0x14a15
-> bzip2patched.exe:0x14a27
-> bzip2patched.exe:0x14a2f
-> bzip2patched.exe:0x14a40
-> libc.so.6:0x11c5a2
-> libc.so.6:0x11ba8f
-> bzip2patched.exe:0x147d0

```

This result validates the intended role of the syscall-aware extension in this representative case. Attack 2 follows the dependency path from the code reader instruction through the write-side attestation computation, but it does not recover the read-side verification instruction after the syscall-mediated break. Attack 3 adds an analysis-level bridge across that break and therefore extends the data dependencies graph to 0x147d0, the read-side verification instruction.

The result should not be read as exact file-system provenance: the bridge does not prove that one specific write is matched with one specific read. It restores the reachability needed by the ranking workflow.

Chapter 8

Conclusion and Future Works

This thesis investigated whether attack-effort modelling can be applied to reverse-engineering attacks against code-guarded binaries. By applying different software protection implementations, the amount of work required by the attacker increases before the relevant protection logic can be understood and modified. For this reason, the thesis focused on modelling the defenses and related attacks, and evaluating them based on the heuristic metrics defined.

8.1 Conclusions

The results of this thesis show that attack-effort modelling can be applied to code-guarded binaries. The proposed approach represents the attack as an inspection process over a ranked set of candidate instructions. For each evaluated protection variant and attack strategy, the analysis measures whether the target verification instruction is reached and where it appears in the inspection order. The target rank is therefore used as a concrete proxy for the amount of effort required by the modelled attacker.

This ranking-based view provides a structured way to compare protection designs and attack strategies. The evaluation captures two relevant aspects of the attack: target reachability and inspection effort. These aspects make it possible to reason about protection strength in terms of the work imposed on the attacker and the attack model required to recover the relevant verification logic.

The evaluation confirms that protection design has a measurable impact on attack effort. Direct-verification guards lead to the lowest effort because the verification logic remains easy to identify. Comparison obfuscation and function diversification increase the effort required by simple localization strategies. Chained and syscall-mediated guards impose a stronger effect because they make the attacker follow more complex relationships between guard components.

The results also show that stronger protections require stronger attack strategies called syscall-aware differential attack. This attack provides the broadest coverage among the evaluated strategies, because it extends the analysis across system-level interactions.

The main conclusion is that protection strength depends on the attacker model used to evaluate it. A protection that is effective against a simple strategy may still be vulnerable to a more adaptive one. At the same time, a protection remains meaningful when it forces the attacker to use a stronger model or to inspect substantially more candidates before reaching the target. In this thesis, protection strength is therefore assessed through the combination of target reachability, target rank, and the attack strategy required to obtain that result.

8.2 Future Work

These conclusions also indicate the main directions for future work. A first direction is to refine the effort model by studying the constant factor C that relates the target rank to the cost of inspecting and validating one candidate instruction. In this thesis, the rank is used as the primary effort measure because C is treated as an unknown multiplier. Future work could replace this constant with a validation-cost model based on the operations required to assess a candidate. Such a model could assign different costs to actions such as following data and control dependencies, setting breakpoints, applying local patches, and rerunning the program to check whether the tamper response is explained. This would make the effort estimate more precise, because candidates with the same rank may still require different amounts of validation work.

A second direction is to extend the attack objective beyond the verification phase. The attacks considered in this thesis focus on locating the instruction where the guard consumes the attestation-derived value, but an attacker may also target the attestation phase or the action phase. Studying these alternatives would give a broader view of how code guards can be attacked and where their strongest and weakest points are. Finally, the same methodology could be applied to more protection techniques, more attack strategies, and other protected assets. This includes game resources, as well as data and checks used in Digital Rights Management systems, software licence managers, and other applications where software protections are deployed to delay reverse engineering.

Bibliography

- [1] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. «Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis?» In: *ACM Computing Surveys* 49.1 (2016), Article 4. DOI: <http://dx.doi.org/10.1145/2886012>.
- [2] Cataldo Basile, Daniele Canavese, Leonardo Regano, Paolo Falcarin, and Bjorn De Sutter. «A Meta-Model for Software Protections and Reverse Engineering Attacks». In: *The Journal of Systems and Software* 150 (2019), pp. 3–21.
- [3] Thomas Faingnaert, Tab Zhang, Willem Van Iseghem, Gertjan Everaert, Bart Coppens, Christian Collberg, and Bjorn De Sutter. «Tools and Models for Software Reverse Engineering Research». In: *Proceedings of the 2024 Workshop on Research on Offensive and Defensive Techniques in the Context of Man At The End Attacks. CheckMATE '24*. New York, NY, USA: Association for Computing Machinery, 2024, pp. 44–58. DOI: 10.1145/3689934.3690817.
- [4] Alessandro Sanna, Waldo Verstraete, Leonardo Regano, Davide Maiorca, and Bjorn De Sutter. *Statistical Effort Modelling of Game Resource Localisation Attacks*. 2026. arXiv: 2603.04261 [cs.CR].
- [5] Hoi Chang and Mikhail J. Atallah. «Protecting Software Code By Guards». In: *Revised Papers from the ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management*. Springer, 2002, pp. 160–175.
- [6] Bill Horne, Lesley Matheson, Casey Sheehan, and Robert E. Tarjan. «Dynamic Self-Checking Techniques for Improved Tamper Resistance». In: *Security and Privacy in Digital Rights Management*. Springer, 2002, pp. 141–159.
- [7] Jing Qiu, Babak Yadegari, Brian Johannismeyer, Saumya Debray, and Xiaohong Su. «Identifying and Understanding Self-Checksumming Defenses in Software». In: *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. ACM, 2015, pp. 207–218. DOI: 10.1145/2699026.2699109.
- [8] Paolo Falcarin, Christian Collberg, Mikhail Atallah, and Mariusz Jakubowski. «Guest Editors' Introduction: Software Protection». In: *IEEE Software* 28.2 (2011), pp. 24–27. DOI: 10.1109/MS.2011.34.
- [9] Christian Collberg and Jasvir Nagra. «Software Tamperproofing». In: *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 2009. Chap. 7, pp. 401–466.

- [10] Mohsen Ahmadvand, Alexander Pretschner, and Florian Kelbert. «A Taxonomy of Software Integrity Protection Techniques». In: *Advances in Computers*. Vol. 112. Elsevier, 2019. DOI: 10.1016/bs.adcom.2017.12.007.
- [11] Jens Van den Broeck, Bart Coppens, and Bjorn De Sutter. «Obfuscated Integration of Software Protections». In: *International Journal of Information Security* 20 (2021), pp. 73–101. DOI: 10.1007/s10207-020-00494-8.
- [12] Christian Collberg, Clark Thomborson, and Douglas Low. *A Taxonomy of Obfuscating Transformations*. Tech. rep. 148. Department of Computer Science, University of Auckland, 1997.
- [13] Christian Collberg, Clark Thomborson, and Douglas Low. «Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs». In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1998, pp. 184–196. DOI: 10.1145/268946.268962.
- [14] National Security Agency. *Ghidra Software Reverse Engineering Framework*. <https://github.com/NationalSecurityAgency/ghidra>. Accessed 2026.
- [15] Hex-Rays. *IDA Pro: Disassembler, Decompiler and Debugger*. <https://hex-rays.com/ida-pro>. Accessed 2026.
- [16] *Debugging with GDB*. Accessed 2026. GNU Project.
- [17] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. «Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation». In: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*. PLDI '05. ACM, 2005, pp. 190–200. DOI: 10.1145/1065010.1065034.
- [18] Nicholas Nethercote and Julian Seward. «Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation». In: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*. ACM, 2007, pp. 89–100. DOI: 10.1145/1250734.1250746.
- [19] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. «An Infrastructure for Adaptive Dynamic Optimization». In: *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2003, pp. 265–275.
- [20] James Clause, Wanchun Li, and Alessandro Orso. «Dytan: A Generic Dynamic Taint Analysis Framework». In: *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. ACM, 2007, pp. 196–206. DOI: 10.1145/1273463.1273490.
- [21] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. «All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution but Might Have Been Afraid to Ask». In: *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 317–331. DOI: 10.1109/SP.2010.26.
- [22] Mark Weiser. «Program Slicing». In: *IEEE Transactions on Software Engineering* SE-10.4 (1984), pp. 352–357. DOI: 10.1109/TSE.1984.5010248.

- [23] Bogdan Korel and Janusz Laski. «Dynamic Program Slicing». In: *Information Processing Letters* 29.3 (1988), pp. 155–163. DOI: 10.1016/0020-0190(88)90054-3.
- [24] Hiralal Agrawal and Joseph R. Horgan. «Dynamic Program Slicing». In: *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*. ACM, 1990, pp. 246–256. DOI: 10.1145/93542.93576.
- [25] Cataldo Basile, Bjorn De Sutter, Daniele Canavese, Leonardo Regano, and Bart Coppens. «Design, Implementation, and Automation of a Risk Management Approach for Man-at-the-End Software Protection». In: *Computers & Security* 132 (2023), p. 103321. DOI: <https://doi.org/10.1016/j.cose.2023.103321>.
- [26] Nenad Dedić, Mariusz H. Jakubowski, and Ramarathnam Venkatesan. «A Graph Game Model for Software Tamper Protection». In: *Information Hiding*. Vol. 4567. Lecture Notes in Computer Science. Springer, 2007, pp. 80–95. DOI: 10.1007/978-3-540-77370-2_6.
- [27] *ASPIRE Security Evaluation Methodology*. Deliverable D4.06 ICT-609734 / D4.06. Advanced Software Protection: Integration, Research and Exploitation (ASPIRE), FP7 project no. 609734, Revision v1.0. ASPIRE Consortium, Oct. 2016.
- [28] Jens Van den Broeck, Bart Coppens, and Bjorn De Sutter. «Flexible Software Protection». In: *Computers & Security* 116 (2022), p. 102636.
- [29] Alessio Viticchié, Cataldo Basile, Andrea Avancini, Mariano Ceccato, Bert Abrath, and Bart Coppens. «Reactive Attestation: Automatic Detection and Reaction to Software Tampering Attacks». In: *Proceedings of SPRO’16*. 2016. DOI: <http://dx.doi.org/10.1145/2995306.2995315>.
- [30] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. «Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems». In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. ACM, 2005, pp. 1–16. DOI: 10.1145/1095810.1095812.
- [31] *elf(5) Linux Manual Page*. <https://man7.org/linux/man-pages/man5/elf.5.html>. Linux man-pages project.
- [32] *System V Application Binary Interface*. <https://refspecs.linuxfoundation.org/elf/gabi4+/contents.html>. The Santa Cruz Operation.
- [33] *Linker and Libraries Guide: Sections*. https://docs.oracle.com/cd/E26505_01/html/E26506/chapter6-94076.html. Oracle.
- [34] *readelf – GNU Binary Utilities Documentation*. <https://sourceware.org/binutils/docs/binutils/readelf.html>. GNU Binutils.
- [35] *Linker and Libraries Guide: Program Header*. <https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-83432/index.html>. Oracle.
- [36] James H. Hill and Dennis C. Feiock. «Pin++: An Object-oriented Framework for Writing Pintools». In: *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*. GPCE ’14. ACM, 2014, pp. 133–141. DOI: 10.1145/2658761.2658777.

- [37] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. «The Program Dependence Graph and Its Use in Optimization». In: *ACM Transactions on Programming Languages and Systems* 9.3 (1987), pp. 319–349. DOI: 10.1145/24039.24041.
- [38] Joppe W. Bos, Charles Hubain, Wil Michiels, and Philippe Teuwen. «Differential Computation Analysis: Hiding Your White-Box Designs is Not Enough». In: *Cryptographic Hardware and Embedded Systems – CHES 2016*. Vol. 9813. Lecture Notes in Computer Science. Springer, 2016, pp. 215–236. DOI: 10.1007/978-3-662-53140-2_11.
- [39] R. Reno Robert. «Differential Execution Analysis for Obfuscation Reduction». In: *Recent Trends in Computer Networks and Distributed Systems Security*. Ed. by Gregorio Martínez Pérez, Sabu M. Thampi, Ryan Ko, and Lei Shu. Vol. 420. Communications in Computer and Information Science. Springer, 2014, pp. 358–369. DOI: 10.1007/978-3-642-54525-2_32.
- [40] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. «Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis». In: *Proceedings of the Network and Distributed System Security Symposium*. NDSS. 2017. DOI: 10.14722/ndss.2017.23465.
- [41] Michael Howard, Jon Pincus, and Jeannette M. Wing. «Measuring Relative Attack Surfaces». In: *Computer Security in the 21st Century*. Springer, 2005, pp. 109–137. DOI: 10.1007/0-387-24006-3_8.
- [42] Pratyusa K. Manadhata and Jeannette M. Wing. «An Attack Surface Metric». In: *IEEE Transactions on Software Engineering* 37.3 (2011), pp. 371–386. DOI: 10.1109/TSE.2010.60.
- [43] Mohsen Ahmadvand, Daniel Below, Sebastian Banescu, and Alexander Pretschner. «VirtSC: Combining Virtualization Obfuscation with Self-Checksumming». In: *Proceedings of the 3rd ACM Workshop on Software Protection*. ACM, 2019, pp. 53–63. DOI: 10.1145/3338503.3357723.
- [44] Mariano Ceccato, Paolo Tonella, Cataldo Basile, Paolo Falcarin, Marco Torchiano, Bart Coppens, and Bjorn De Sutter. *Understanding the Behaviour of Hackers while Performing Attack Tasks in a Professional Setting and in a Public Challenge*. EMSE manuscript version in uploaded PDF.
- [45] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S. Foster, and Michelle L. Mazurek. «An Observational Investigation of Reverse Engineers’ Processes». In: *Proceedings of the 29th USENIX Security Symposium*. USENIX Security ’20. USENIX Association, 2020, pp. 1875–1892. URL: <https://www.usenix.org/conference/usenixsecurity20/presentation/votipka-observational>.
- [46] Mariano Ceccato, Massimiliano Di Penta, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. «A Family of Experiments to Assess the Effectiveness and Efficiency of Source Code Obfuscation Techniques». In: *Empirical Software Engineering* 19.4 (2014), pp. 1040–1074. DOI: 10.1007/s10664-013-9248-x.

- [47] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. «Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study». In: *Proceedings of the 2016 IEEE Symposium on Security and Privacy*. SP '16. IEEE, 2016, pp. 158–177. DOI: 10.1109/SP.2016.18.
- [48] Tab Zhang, Bjorn De Sutter, Christian Collberg, Bart Coppens, and Waleed Mebane. *Empirical Studies on Adversarial Reverse Engineering with Students*. 2026. arXiv: 2603.03875 [cs.SE].
- [49] Thomas Faingnaert, Willem Van Iseghem, and Bjorn De Sutter. «K-Hunt++: Improved Dynamic Cryptographic Key Extraction». In: *Proceedings of the 2024 Workshop on Research on Offensive and Defensive Techniques in the Context of Man At The End Attacks*. CheckMATE '24. Association for Computing Machinery, 2024. DOI: 10.1145/3689934.3690818.
- [50] W. Eric Wong, Vidroha Debroy, and Byoungju Choi. «A Family of Code Coverage-Based Heuristics for Effective Fault Localization». In: *Journal of Systems and Software* 83.2 (2010), pp. 188–208. DOI: 10.1016/j.jss.2009.09.037.
- [51] James A. Jones and Mary Jean Harrold. «Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique». In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2005, pp. 273–282. DOI: 10.1145/1101908.1101949.
- [52] Chris Parnin and Alessandro Orso. «Are Automated Debugging Techniques Actually Helping Programmers?» In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 199–209. DOI: 10.1145/2001420.2001445.
- [53] Baojiang Cui, Fuwei Wang, Tao Guo, and Guowei Dong. «A Practical Off-Line Taint Analysis Framework and Its Application in Reverse Engineering of File Format». In: *Computers & Security* 51 (2015), pp. 1–15. DOI: 10.1016/j.cose.2015.02.006.
- [54] Thomas Ball and James R. Larus. «Efficient Path Profiling». In: *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 1996, pp. 46–57. DOI: 10.1109/MICRO.1996.566449.
- [55] Daniele Cono D’Elia, Camil Demetrescu, and Irene Finocchi. «Ball-Larus Path Profiling Across Multiple Loop Iterations». In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2013, pp. 373–390. DOI: 10.1145/2509136.2509521.
- [56] Bixin Li, Lulu Wang, Hareton Leung, and Fei Liu. «Profiling All Paths: A New Profiling Technique for Both Cyclic and Acyclic Paths». In: *Journal of Systems and Software* 85.7 (2012), pp. 1558–1576. DOI: 10.1016/j.jss.2012.01.046.