Binding Code and Data to Intrinsic Hardware Properties for Software Protection



Daniel Dorfmeister

Complete copy of the thesis submitted to the Department of Computer Science at the University of the Bundeswehr Munich to obtain the academic degree of

Doktor-Ingenieur (Dr.-Ing.)

Examiners:
1. Prof. Dr. techn. Stefan Brunthaler
2. Prof. Dr. ir. Stijn Volckaert

The doctoral thesis was submitted to the University of the Bundeswehr Munich on February 18, 2026 and accepted by the Department of Computer Science on May 20, 2026.
The oral examination took place on June 16, 2026.

Acknowledgements

I am deeply grateful to Prof. Stefan Brunthaler for his supervision and continuous guidance throughout this dissertation. He consistently challenged my ideas, set high expectations, and provided critical feedback that strengthened both the technical depth and clarity of this work. I am thankful to Prof. Stijn Volckaert for his role in shaping the ideas and direction of this research through valuable discussions, as well as for serving as my second examiner.

I owe special thanks to Bernhard and Flavio for their constant support, close collaboration, and many discussions, and for keeping me motivated and making this time enjoyable. I thank Ruben, Mathéo, Hannes, and Martin for their significant contributions and valuable collaboration. I am grateful to Thomas and Juliana for their support and encouragement. I also thank my colleagues at the university and the project partners, especially my coauthors, for insightful exchanges and productive collaboration.

I am grateful to Prof. Heinz Dobler for his early encouragement to pursue a doctoral degree and for supporting me in finding my supervisor. Finally, I thank my parents for their continuous support and encouragement throughout my academic journey.

The research reported in this dissertation has been funded by the Federal Ministry for Innovation, Mobility and Infrastructure (BMIMI), the Federal Ministry for Economy, Energy and Tourism (BMWET), and the State of Upper Austria in the frame of the COMET Module Dependable Production Environments with Software Security (DEPS) (FFG grant no. 888338) and the SCCH competence center INTEGRATE (FFG grant no. 892418) within the COMET – Competence Centers for Excellent Technologies Programme managed by Austrian Research Promotion Agency FFG.

Kurzfassung

Industrielles Reverse Engineering gefährdet das in Software enthaltene Fachwissen, da ihre Replikation meist kostengünstiger ist als die eigenständige Neuentwicklung. Diese Dissertation stellt ein konzeptionelles Modell zur Hardware–Software-Bindung vor, das die beabsichtigte Softwareausführung an intrinsische, einzigartige und nicht reproduzierbare Hardwareeigenschaften koppelt, um das Kopieren und Reverse Engineering der Software zu erschweren. Die Software wird bei der Entwicklung und Kompilierung so transformiert, dass ihre beabsichtigte Form erst zur Laufzeit mithilfe von Hardwareeigenschaften wiederhergestellt wird, die als Hardware-Fingerabdruck dienen. Infolgedessen zeigt die geschützte Software ausschließlich auf dem Originalgerät ihr beabsichtigtes Verhalten, auf nachgebauter oder nicht autorisierter Hardware hingegen ein unbeabsichtigtes Verhalten.

Den Schwerpunkt der vorliegenden Arbeit bilden zwei auf Rowhammer basierende Schutzmechanismen – Rowhammer ist ein Hardwarefehler in DRAM, der die Extraktion von Hardware-Fingerabdrücken ermöglicht. Der erste Ansatz, GlueZilla, bindet Programmbefehle an spezifische DRAM-Chips und erfordert eine geräteabhängige Wiederherstellung der beabsichtigten Ausführung zur Laufzeit. Angluerus ergänzt diesen Mechanismus durch die Bindung von Programmdaten. Beide Ansätze nutzen zuverlässige und gerätespezifische, durch Rowhammer induzierte Bitflips, um das beabsichtigte Programmverhalten während der Ausführung dynamisch wiederherzustellen.

Ergänzend dazu stellt die Dissertation drei weitere Ansätze vor, die industriespezifische Anwendungsszenarien adressieren und andere Hardware-Fingerabdrücke verwenden. Minotaurox leitet Kontrollzustandsübergänge aus einem Hardware-Fingerabdruck ab und bindet damit den Programmkontrollfluss, wobei betriebssichere Übergänge auch bei einem falschen Fingerabdruck gewährleistet bleiben. Sphinxor schützt sensible Daten, indem es diese nur unter Verwendung des korrekten Hardware-Fingerabdrucks dekodiert und andernfalls betriebssichere Alternativen liefert. Locki überträgt das Konzept auf neuronale Netze, indem deren Gewichte zur Laufzeit mithilfe eines aus der Hardware abgeleiteten kryptographischen Schlüssels wiederhergestellt werden.

Umfassende Sicherheitsanalysen und empirische Evaluierungen auf mehreren Plattformen demonstrieren die praxistaugliche Extraktion zuverlässiger Hardware-Fingerabdrücke ohne zusätzliche Hardware. Die Ergebnisse zeigen zudem, dass die vorgeschlagenen Schutzmechanismen Reverse Engineering bei akzeptablem Overhead unwirtschaftlich machen. Insgesamt etabliert diese Dissertation die Hardware–Software-Bindung als praxistaugliche und vielseitige Grundlage zum Schutz industrieller Software und erhöht die Hürde für unbefugtes Kopieren und Reverse Engineering erheblich.

Abstract

Industrial-scale reverse engineering compromises valuable knowledge embodied in industrial software, as cloning is often cheaper than independent development. This dissertation introduces a conceptual model for hardware–software binding that ties intended software execution to intrinsic, unique, and unclonable hardware properties, aiming to hinder the copying and reverse engineering of software. Software is transformed at design and compile time such that its intended form is restored only at run time using hardware properties that act as a hardware fingerprint. As a result, protected software exhibits its intended behavior only on the original device and unintended behavior on cloned or unauthorized hardware.

The core contribution comprises two protection mechanisms based on Rowhammer, a hardware fault in DRAM that enables hardware fingerprinting. The first approach, GLUEZILLA, binds program instructions to specific DRAM chips and requires device-dependent restoration of intended execution at run time. ANGLUERUS complements this mechanism by binding program data. Both approaches employ reliable and device-unique Rowhammer-induced bit flips to dynamically restore intended program behavior during execution.

Complementing these mechanisms, the dissertation presents three additional approaches that address restricted yet common industrial scenarios using other hardware fingerprints. MINOTAUROX derives control-state transitions from a hardware fingerprint to bind program control flow, ensuring safe transitions under an incorrect fingerprint. SPHINXOR protects sensitive data by decoding it only when evaluated with the correct hardware fingerprint and otherwise yields safe alternatives. LOCKI extends the concept to neural network models by decrypting their weights at run time using a hardware-derived key.

Comprehensive security analyses and empirical evaluations across several platforms demonstrate the practical extraction of reliable hardware fingerprints without requiring additional hardware and show that the proposed protection mechanisms render reverse engineering uneconomical with acceptable overhead. Overall, this dissertation establishes hardware–software binding as a practical and versatile protection primitive for industrial software, substantially raising the barrier for unauthorized copying and reverse engineering.

Contents

List of Excursuses **xiii**

List of Definitions **xv**

List of Figures **xvii**

List of Listings **xix**

List of Tables **xxi**

List of Abbreviations **xxiv**

Publications **xxv**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Reverse Engineering . . . 4
1.3 Software Protection . . . 6
1.3.1 Copy Protection . . . 7
1.3.2 Reverse Engineering Protection . . . 7
1.3.3 Tamper Protection . . . 8
1.3.4 Software Diversity . . . 9
1.3.5 Hardware–Software Binding . . . 10
1.4 Contributions . . . 11

2 Background **13**
2.1 Dynamic Random-Access Memory . . . 13
2.1.1 DRAM Organization . . . 14
2.1.2 DRAM Addressing . . . 15
2.1.3 Memory Allocation . . . 15
2.2 Rowhammer . . . 15
2.2.1 Memory Massaging . . . 16
2.2.2 Rowhammer Exploits . . . 17
2.2.3 Rowhammer Mitigations . . . 17
2.2.4 Advanced Rowhammer Techniques . . . 18
2.3 Hardware Fingerprinting . . . 18
2.3.1 Dongles . . . 20
2.3.2 Trusted Platform Modules . . . 20
2.3.3 Trusted Execution Environments . . . 20
2.3.4 Physically Unclonable Functions . . . 21
2.3.5 Rowhammer-based Fingerprinting . . . 21

3 Threat Model and Protection Concept **23**
3.1 Scenarios . . . 24
3.2 Threat Model . . . 25
3.3 Conceptual Protection Model . . . 27
3.4 Building Blocks . . . 28
3.4.1 Hardware Fingerprinting . . . 28
3.4.2 Transformations . . . 29
3.4.3 Constructing Scenario-Specific Approaches . . . 30

4 Rowhammer-Based Hardware–Software Binding **31**
4.1 Rowhammer Analysis . . . 33
4.1.1 Rowhammer Properties . . . 33
4.1.2 Memory Templater . . . 34
4.1.3 Bit Flip Reliability . . . 35
4.2 Design . . . 40
4.3 GlueZilla . . . 44
4.3.1 Junction Instruction Selection . . . 45
4.3.2 Compilation . . . 46
4.3.3 Loading . . . 47
4.3.4 Execution . . . 48
4.4 Angluerus . . . 48
4.4.1 Junction Symbol Selection . . . 49
4.4.2 Compilation . . . 49
4.4.3 Execution . . . 51

5 PUF-Based Hardware–Software Binding **53**
5.1 SRAM-Based Hardware Fingerprinting . . . 54
5.2 Minotaurox . . . 55
5.2.1 Design . . . 56
5.2.2 Implementation . . . 62
5.3 Sphinxor . . . 62
5.3.1 Design . . . 64
5.3.2 Implementation . . . 67

6 Hardware–Software Binding for NN Models **69**
6.1 Locki . . . 70
6.1.1 Design . . . 72
6.1.2 Implementation . . . 72

7 Evaluation **75**
7.1 Evaluation of Rowhammer-Based Approaches . . . 75
7.1.1 Security Analysis . . . 76
7.1.2 Empirical Evaluation . . . 82
7.1.3 Discussion . . . 88
7.2 Evaluation of PUF-Based Approaches . . . 91
7.2.1 Security Analysis . . . 91

7.2.2 Empirical Evaluation 94
7.2.3 Discussion 95
7.3 Evaluation of Approach for AI Models 97
7.3.1 Security Analysis 97
7.3.2 Empirical Evaluation 99
7.3.3 Discussion 101
7.4 Comparative Evaluation 102
7.4.1 Security Comparison 103
7.4.2 Implementation Comparison 104

8 Conclusion **107**
8.1 Summary 107
8.2 Future Work 108
8.3 Lessons Learned 109

References **111**

List of Excursuses

1 Linking 50
2 Tracing 51
3 Abstract State Machines 56
4 Symbolic Execution 57
5 Neural Networks 69
6 Symmetric Encryption 71

List of Definitions

1 Target and Cloned Device 2
2 Intended and Unintended Behavior 3
3 Intrinsic and Extrinsic Hardware Fingerprinting 19
4 Static and Dynamic Hardware Fingerprinting 19

List of Figures

1.1 Overview of different mechanisms to protect software. 2
1.2 Intended and unintended program execution. 3
1.3 Comparison of static and dynamic analysis. 4
1.4 Overview of different types of software protection techniques. 6

2.1 Schematic overview of DRAM organization. 14
2.2 Comparison of hammer patterns. 18
2.3 Comparison of techniques for hardware fingerprinting. 19

3.1 Attack–defense tree. 25
3.2 Overview of conceptual protection model. 27

4.1 Design of GlueZilla and Angluerus. 32
4.2 Architecture of the memory templater. 35
4.3 Bit flip reliability. 38
4.4 Distribution of bit flips. 39
4.5 Reliability of bit flips out of 1 000 Rowhammer tests per flip direction. 39
4.6 Comparison of the design principles underlying GlueZilla and Angluerus. 41
4.7 Fields of x86 instructions. 42
4.8 Overview of GlueZilla's phases. 45
4.9 Run-time transformations in GlueZilla. 46
4.10 Junction bits inside control transfer destinations. 46
4.11 Run-time transformations in Angluerus. 49

5.1 Design of Minotaurox. 55
5.2 Traffic-light states and corresponding control-state values. 58
5.3 Control-state transitions of Protected1WayStopGoLight. 59
5.4 Design of Sphinxor. 63
5.5 PID controller responses using different parameters. 63

6.1 Design of Locki. 70

7.1 Most common mnemonics of PLC programs compared to `coreutils` programs. 78
7.2 Performance impact of GlueZilla on SPEC CPU 2017. 85
7.3 Performance impact of GlueZilla and Angluerus on SPEC CPU 2017. 87
7.4 Performance evaluation of Sphinxor's prototype for different PUF sizes. . . 95
7.5 Comparing precision and recall of original and protected models. 100
7.6 Results of evaluating Locki's proof of concept. 101

List of Listings

2.1 Basic steps of a Rowhammer attack. 16

4.1 Simplified templating code for double-sided hammering. 34
4.2 Simple C program to demonstrate ANGLUERUS. 43

5.1 General form of control-state ASM rules. 56
5.2 Correct ASM specification of a one-way traffic light. 58
5.3 MINOTAUROX's hardware–software binding procedure. 59
5.4 MINOTAUROX uses a PUF to choose the next control state. 61
5.5 Generic PID controller. 64
5.6 SPHINXOR's secure recovery of protected *Kp*, *Ki* and *Kd* values. 66

List of Tables

4.1 Predictive probabilities and 95% credible intervals. 36
4.2 Examples of x86 instruction pairs differing by a single bit. 42

7.1 Number and size of data symbols in analyzed programs. 81
7.2 Overview of NN models used in our experiments. 99
7.3 Overview of approaches. 102
7.4 Comparison of approach security. 103
7.5 Overview of approach implementations. 104
7.6 Comparison of deployment effort and overheads. 105

List of Abbreviations

AI	artificial intelligence
API	application programming interface
ASLR	address space layout randomization
ASM	abstract state machine
BROL	Basically Runs On L1
BTB	branch target buffer
CATT	Can't Touch This
CFF	control-flow flattening
CFI	control-flow integrity
CNN	convolutional neural network
CPU	central processing unit
CRA	code-reuse attack
CRP	challenge–response pair
DBA	dynamic binary analysis
DBT	dynamic binary translation
DC	direct current
DDR	double data rate
DEPS	Dependable Production Environments with Software Security
DIMM	dual inline memory module
DoS	denial of service
DRAM	dynamic random-access memory
DRK	device root key
ECC	error correction code
ECDSA	Elliptic Curve Digital Signature Algorithm
EK	endorsement key
ELF	executable and linkable format
FHE	fully homomorphic encryption
FNN	feedforward neural network
FPGA	field-programmable gate array
FSM	finite-state machine
GDDR	graphics double data rate
GPU	graphics processing unit
HCIC	hardware-assisted control-flow integrity checking
HESP	Hardware-Entangled Software Protection
I/O	input/output
iff	if and only if
IoT	internet of things
ITLB	instruction translation lookaside buffer
JIT	just-in-time
JOP	jump-oriented programming
JTAG	Joint Test Action Group (IEEE Standard 1149.1)
LED	light-emitting diode
LTO	link-time optimization
MAD	memory allocation diversity

MCU microcontroller unit
ML machine learning
NN neural network
OS operating system
PC personal computer
PID proportional–integral–derivative
PLC programmable logic controller
PUF physically unclonable function
RISC reduced instruction set computer
RNG random number generator
RNN recurrent neural network
ROP return-oriented programming
RoT root of trust
SEPP secure execution PUF-based processor
SGX software guard extensions
SIB scale–index–base
SMT satisfiability modulo theories
SMT simultaneous multithreading
SPEC Standard Performance Evaluation Corporation
SRAM static random-access memory
SVM support vector machine
SWD Serial Wire Debug
TEE trusted execution environment
TPM Trusted Platform Module
TRR target row refresh
UART universal asynchronous receiver-transmitter
USB Universal Serial Bus
VM virtual machine
vtable virtual function table
XOR exclusive or

Publications

This dissertation is based on the following peer-reviewed publications, ordered by recency:

Fischer, B., **Dorfmeister, D.**, Ferrarotti, F., Rass, S., Mayrhofer, R., "ASM-based behavioral attestation: Bridging formal specification and runtime conformance checking of security properties," to appear in *Proceedings of the 24th ACM/IEEE International Symposium on Formal Methods and Models for System Design (MEMOCODE 2026)*.

Sochor, H., Schwandtner, M., **Dorfmeister, D.**, Fischer, B., Ferrarotti, F., Fuß, J., "Locki: Neural networks that only behave when unlocked via runtime decryption," to appear in *Proceedings of 37th International Conference on Database and Expert Systems Applications (DEXA 2026)*.

Fischer, B., **Dorfmeister, D.**, Ferrarotti, F., Penz, M., Kargl, M., Zeinzinger, M., Eibensteiner, F., "Software-hardware binding for protection of sensitive data in embedded software," in *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing*. ACM, May 2025, pp. 570–577. DOI: 10.1145/3672608.3707855.

Fischer, B., **Dorfmeister, D.**, Lampesberger, H., Hermann, E., "Leveraging Rowhammer for physically unique and non-tamperable device identification," *Procedia Computer Science*, vol. 253, pp. 1154–1164, Feb. 2025, 6th International Conference on Industry 4.0 and Smart Manufacturing. DOI: 10.1016/J.PROCS.2025.01.177.

Mechelinck, R., **Dorfmeister, D.**, Fischer, B., Volckaert, S., Brunthaler, S., "GlueZilla: Efficient and scalable software to hardware binding using rowhammer," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, F. Maggi, M. Egele, M. Payer, and M. Carminati, Eds., Springer, Jul. 2024, pp. 416–438. DOI: 10.1007/978-3-031-64171-8_22.

Dorfmeister, D., Ferrarotti, F., Fischer, B., Schwandtner, M., Sochor, H., "A PUF-based approach for copy protection of intellectual property in neural network models," in *Software Quality as a Foundation for Security*, P. Bludau, R. Ramler, D. Winkler, and J. Bergsmann, Eds., Springer, Apr. 2024, pp. 153–169. DOI: 10.1007/978-3-031-56281-5_9.

Zeinzinger, M., Langer, J., Eibensteiner, F., Petz, P., Drack, L., **Dorfmeister, D.**, Ramler, R., "Comparative analysis of SRAM PUF temperature susceptibility on embedded systems," in *2023 International Conference on Electrical, Computer and Energy Technologies (ICECET)*, IEEE, Nov. 2023, pp. 1–8. DOI: 10.1109/ICECET58911.2023.10389242.

Dorfmeister, D., Ferrarotti, F., Fischer, B., Haslinger, E., Ramler, R., Zimmermann, M., "An approach for safe and secure software protection supported by symbolic execution," in *Database and Expert Systems Applications - DEXA 2023 Workshops*, G. Kotsis, A. M. Tjoa, I. Khalil, B. Moser, A. Mashkoor, J. Sametinger, and M. Khan, Eds., Springer, Aug. 2023, pp. 67–78. DOI: 10.1007/978-3-031-39689-2_7.

Wiesinger, M., **Dorfmeister, D.**, Brunthaler, S., "MAD: Memory allocation meets software diversity," in *First Workshop on DRAM Security (DRAMSec)*, Jun. 2021. [Online]. Available: https://dramsec.ethz.ch/papers/mad.pdf.

The dissertation is further based on the following unpublished manuscript:

Sochor, H., Fischer, B., **Dorfmeister, D.**, Ferrarotti, F., "Deriving cryptographic keys from memory-intrinsic entropy."

Für augenblicklichen Gewinn verkaufe ich die Zukunft nicht!

I will not sell the future for instant profit!

— Werner von Siemens

1 Introduction

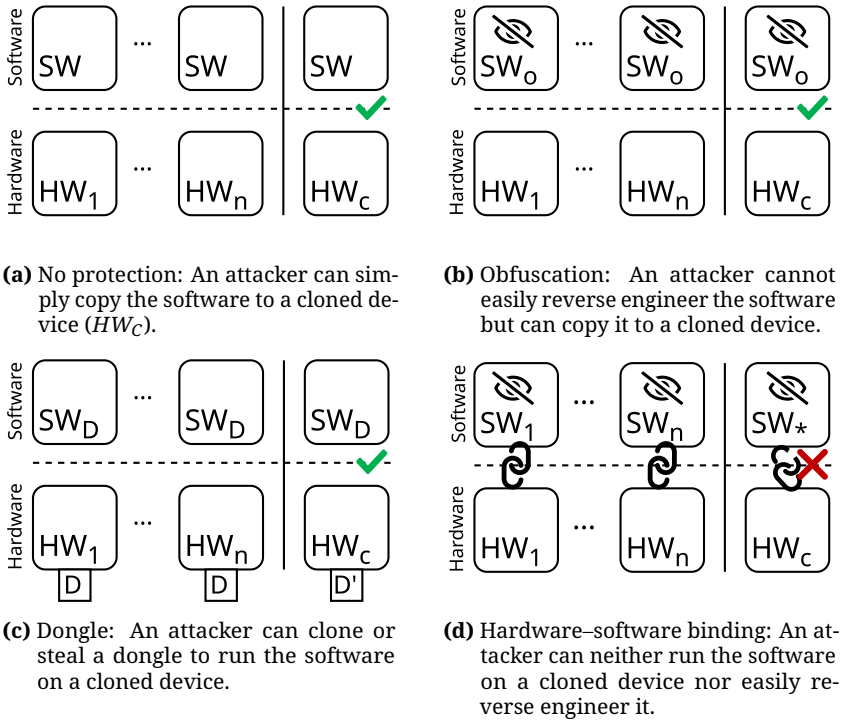
Contents

1.1 Motivation 1
1.2 Reverse Engineering 4
1.3 Software Protection 6
1.3.1 Copy Protection 7
1.3.2 Reverse Engineering Protection 7
1.3.3 Tamper Protection 8
1.3.4 Software Diversity 9
1.3.5 Hardware–Software Binding 10
1.4 Contributions 11

1.1 Motivation

Estimated at 4.1 billion euros in 2024, product piracy in the German mechanical and plant engineering industry imposes substantial economic costs and poses a significant threat to prosperity [1]. Following a gradual decline from the peak of 7.9 billion euros in 2014 to 6.4 billion euros in 2022, the sudden drop in 2024 may be the result of significantly increasing anti-piracy efforts observed in 2022 [2]. While competitors remain the main plagiarizers, professional large-scale counterfeiters and state-owned enterprises have gained significant prominence since 2018 [1]. This shift toward better-resourced adversaries able to circumvent increasingly effective protection measures underscores that investments into safeguarding technical knowledge pay off.

Mechanical engineering companies traditionally invest substantial amounts into research and development, resulting in high-tech products with correspondingly high premiums. To offset these premiums and avoid incurring prior research and development costs, industrial-scale reverse engineering enabling piracy has become a billion-euro business. Industry findings suggest that this is financially sound: the viability of reverse engineering rests crucially on whether it is cheaper to steal technical knowledge than to develop it



(a) No protection: An attacker can simply copy the software to a cloned device (HW_C).

(b) Obfuscation: An attacker cannot easily reverse engineer the software but can copy it to a cloned device.

(c) Dongle: An attacker can clone or steal a dongle to run the software on a cloned device.

(d) Hardware–software binding: An attacker can neither run the software on a cloned device nor easily reverse engineer it.

Figure 1.1: Overview of different mechanisms to protect software.

in-house. If the cost of reverse engineering is lower than actual development, reverse engineering becomes the rational choice.

Consider companies that use domain-specific knowledge accumulated over many years to fine-tune control algorithms or train machine learning (ML) models to operate advanced machinery. This software results from significant investments in research and development and constitutes a key competitive advantage. While reverse engineering sophisticated hardware is challenging, copying the software to a cloned machine typically poses little or no difficulty, as it runs unmodified on both original and cloned machinery. Consequently, attackers can avoid incurring substantial innovation costs by duplicating and reusing this knowledge; see Figure 1.1a. Moreover, software without strong protection remains vulnerable to extraction and analysis using reverse engineering techniques, removing the need for adversaries to independently develop equivalent functionality, as discussed in Section 1.2.

The effectiveness of traditional protection techniques—such as obfuscation or hardware dongles—remains limited, as discussed in Section 1.3. Software obfuscation impedes reverse engineering but does not prevent copying; see Figure 1.1b. Dongles, on the other hand, restrict software execution across machines, but an attacker can clone or steal them or remove the checking mechanism, thereby allowing program copies to run on a cloned machine; see Figure 1.1c. In contrast, we propose a conceptual model to protect against reverse engineering *and* unauthorized copying of software. We bind a software instance to one specific hardware instance, the *target device*; see Figure 1.1d. The key idea is to create software that operates correctly *only* on the target device it was compiled for and exhibits degraded, incorrect, or unintended behavior on all other devices, including exact hardware clones.

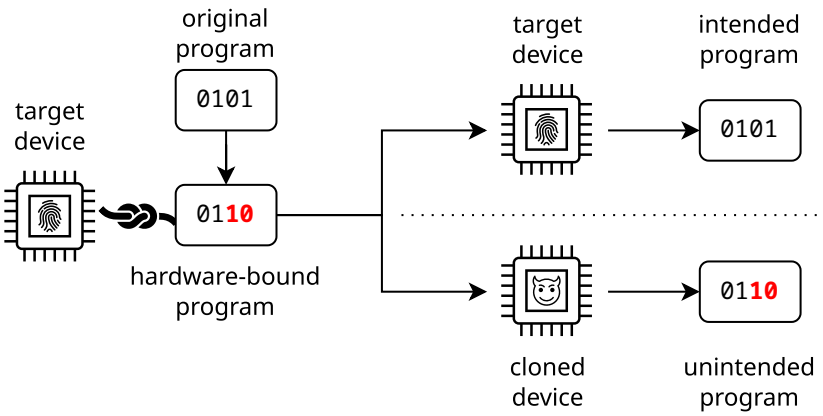


Figure 1.2: A program is bound to the target device: executing the program on the target device results in intended behavior, while executing it on a cloned device yields unintended behavior.

Definition 1: Target and Cloned Device

The *target* device is the original device that we bind the *target* program to, i.e., the only device that should be able to execute the program correctly. A *cloned* device is any other compatible device that replicates the specifications of the target device.

Our concept, as outlined in Figure 1.2, binds software to intrinsic hardware properties. We use physical properties of hardware originating in manufacturing variations, making them effectively unclonable. We modify a program's behavior within the binary and use the device-specific properties to restore the original execution at run time. As a result, the program becomes inherently bound to the target device, and only this device restores the *intended behavior*. On all other devices, the program fails to reconstruct the intended state and instead exhibits *unintended behavior*.

Definition 2: Intended and Unintended Behavior

Intended behavior refers to a program being executed as implemented by the programmer, before any protections are applied that change the program's behavior. *Unintended* behavior is any deviation from the intended behavior. We want a program to show unintended behavior iff the program is executed on a cloned device.

We demonstrate our concept through five complementary approaches:

1. GlueZilla binds program behavior to Rowhammer-induced bit flips in main memory by focusing on instruction-level transformations that dynamically switch between different control flows.
2. Angluerus follows the same principle as GlueZilla but modifies data rather than instructions, such as constants used in branch conditions.
3. Minotaurox uses data extracted from hardware fingerprints to bind software to hardware by determining the next control state a protected program should assume.
4. Sphinxor also relies on hardware-extracted data but uses it to decode sensitive data.
5. Locki uses hardware-extracted keys to decrypt parameters of ML models during execution.

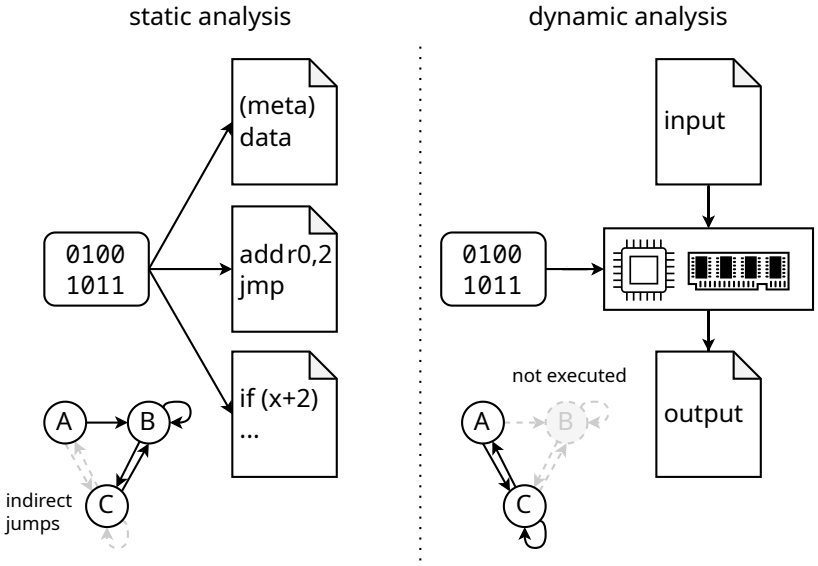


Figure 1.3: Static analysis inspects binaries and derives disassembled or decompiled code, which may miss indirect control-flow transfers. Dynamic analysis executes the binary on concrete inputs and observes only executed paths and outputs.

Before we talk about the contributions of this dissertation in more detail in Section 1.4, we first want to better understand the challenges we have to overcome and how we can improve upon state-of-the-art software protection techniques.

1.2 Reverse Engineering

Reverse engineering of software aims to recover high-level design and implementation knowledge from existing software systems [3]. Reverse engineering encompasses a broad spectrum of techniques that analyze source code, binaries, or observed behavior to support program comprehension, maintenance, and security analysis [4–12].

Beyond benign use cases, reverse engineering can also be employed to extract knowledge from software, remove protection mechanisms, or create software clones, i.e., new programs that replicate the functionality of an existing program without directly copying it. An adversarial reverse engineer might first need to find a way to obtain a binary copy of the program or a process image, which can require physical access to the target system. Such acquisition attempts may be hindered by electrical or physical countermeasures, e.g., by actively erasing memory upon intrusion detection.

Once an adversary gains access to a copy of the software to reverse engineer, generally speaking, there are two options: static and dynamic analysis, as shown in Figure 1.3. Static analysis extracts information from program code or binaries without execution, whereas dynamic analysis derives knowledge from observing program execution [9]. Since both methods pose limitations, they can be combined in hybrid approaches.

In general, a reverse engineer may leverage the following classes of techniques, which we discuss in detail below:

- Physical techniques
  - Invasive access
  - Non-invasive access
    - Debug and test interfaces
    - External or removable storage
    - Memory remanence
  - Run-time memory extraction
  - Fault injection
  - Side-channel methods
- Program analysis techniques
  - Static analysis
    - Disassembly, decompilation
    - Control- and data-flow graphs
  - Dynamic analysis
    - Debugging, instrumentation
    - Emulation and sandboxing
    - I/O observation
  - Hybrid approaches

Physical techniques. Physical techniques [13, 14] aim to extract knowledge by exploiting direct access to the hardware on which a program is stored or executed. They target non-volatile storage or volatile memory contents and may operate either offline or while the system is running. *Invasive techniques*, such as decapping or chip-off, physically modify or remove the memory package to access the silicon directly. *Non-invasive techniques* leverage exposed debug or test interfaces, as well as external or removable storage, to obtain firmware images or memory contents without modifying the device. Memory remanence attacks, such as cold-boot attacks, exploit the persistence of data in volatile memory after power loss. *Run-time memory extraction* captures memory contents during execution and can reveal decrypted code, dynamic data structures, or intermediate computation results. *Fault injection and side-channel methods* manipulate or observe physical properties such as voltage, timing, power consumption, or electromagnetic emissions to induce faults or infer internal state. As these techniques operate below the software abstraction layer, they can bypass or weaken purely software-based protection mechanisms.

Program analysis techniques. Program analysis techniques [4] operate at the software level and derive information either from code or its run-time behavior.

Static analysis [9–11, 15] is inherently incomplete because it operates without executing the program, relying solely on the inspection of code or binaries. This limitation means it cannot observe run-time behavior, such as dynamically generated code, indirect control flows, or input-dependent logic. Moreover, static techniques often struggle with obfuscated or packed binaries and suffer from challenges like aliasing, pointer analysis, and resolving dynamic dispatch in object-oriented programs. Despite these limitations, static analysis remains powerful due to its scalability and ability to provide a broad overview of a program's structure. In addition to traditional methods like pattern matching, disassembly, and decompilation, analysts may also compare binaries across versions or devices to identify changes and differences. A reverse engineer may extract control and data flow graphs, call graphs, used dependencies and thus ultimately gain knowledge about potential vulnerabilities.

Dynamic analysis [7–9, 11, 16, 17] is incomplete in the sense that it only reveals behavior exhibited during observed executions. If certain code paths are not taken—due to missing inputs, environmental conditions, or timing constraints—they cannot be analyzed. This makes dynamic analysis highly dependent on the quality and coverage of

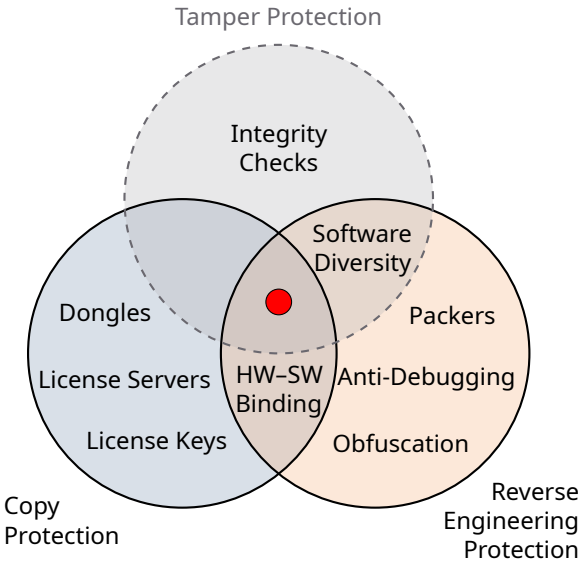


Figure 1.4: Overview of different types of software protection techniques and the ● aim of this dissertation, their intersection.

test cases or input data and can incur significant time overhead. Nevertheless, dynamic techniques are powerful for uncovering run-time behavior such as self-modifying code, unpacking routines, or evasive behavior. Dynamic analysis relies on debugging, tracing, or instrumentation to capture run-time information such as executed paths, variable states, and interactions with external components. Emulation, sandboxing, and fuzzing can further facilitate dynamic analysis—potentially supported by performance counters and breakpoints implemented in hardware.

Hybrid approaches [7, 16] aim to mitigate the limitations of purely static or dynamic techniques by combining their strengths. For example, symbolic execution [9] can explore all paths of a program statically but suffers from path explosion and constraint-solving complexity. Concolic execution [9] addresses both limitations by using concrete inputs to guide symbolic exploration, improving feasibility and scalability. In contrast, static analysis can be used to identify critical code paths that are then executed to observe run-time behavior. Reverse engineering can also benefit from historical data—such as previous execution traces or older program versions—and employ machine learning or clustering to prioritize analysis targets or suggest likely behavior.

1.3 Software Protection

Software protection encompasses techniques and mechanisms designed to safeguard software from unauthorized use, duplication, modification, and analysis. These methods are critical for protecting knowledge and preventing exploitation of software vulnerabilities. Figure 1.4 shows an overview of software protection techniques for several protection targets, i.e., copying, reverse engineering, and tampering. In this section, we discuss these techniques in further detail. Effective protection strategies typically combine complementary mechanisms, such as software diversity and hardware–software binding. We aim to combine these mechanisms to provide comprehensive coverage across all previously mentioned protection targets.

1.3.1 Copy Protection

Traditional methods for controlling software usage include license keys and servers, and dongles. *License keys* [12] are often validated against an internal algorithm or a remote server, which unlock functionality when correctly entered. *License servers* [12] maintain a central record of active installations, enforcing limits on concurrent usage or machine-specific activation. These approaches are vulnerable to key sharing, cracking, and server emulation, and thus offer limited defense against determined attackers.

A *dongle* [5] is a hardware device, typically connected via USB, that must be present for the software to operate. It usually contains a secret key, e.g., a license key, that the software checks at run time to verify authenticity. An attacker can often bypass the check, clone the dongle, remove the verification logic, or simply connect the dongle to another device, allowing copies of the software to execute on cloned or unauthorized hardware [18].

1.3.2 Reverse Engineering Protection

Two research directions aimed at protecting against reverse engineering are software obfuscation and packing, which have traditionally been applied to securing malware and commercial software suites. Additional concepts like weird machines and anti-debugging can be used to further frustrate an adversary.

Obfuscation and packing. Software obfuscation [19–21] obscures programs while preserving their semantics. Effective obfuscation techniques raise the bar for an adversary by ensuring resilience against automated deobfuscation [7, 22] so that reverse engineering becomes too costly while keeping run-time and memory overheads low.

For example, many control-altering transformations to obfuscate control flow rely on *opaque variables and predicates* [23]. A variable is opaque if it has a known property at obfuscation time, and a predicate is opaque if its truth value is known at obfuscation time, yet they are designed to remain computationally difficult to resolve during analysis. When used in conditional statements, they can be combined with *dead code insertion*, where non-executed code is added to disguise the path meant to be executed. *Control-flow flattening (CFF)* [24] flattens control flow of a program into a uniform structure—a switch statement within a loop—hiding the execution order of basic blocks. This removes branching and hides the actual control flow behind dispatcher variables, which can be opaque variables. Instructions can also be substituted by instruction sequences that preserve semantics or by instructions that are not reducible to operators or statements in the source language.

Packing [12, 21, 25] can be regarded as a form of obfuscation, as the program code is typically transformed by compression, encryption, or a combination of both. A prepended auxiliary program, the packer, unpacks the program code into memory during program startup. Since this only hinders static analysis, a packer might resort to unpacking only the code to be executed next and repacking it afterward. This way, the program is never completely loaded into memory, limiting the impact of memory dumps. Furthermore, a program can be *virtualized* by transforming it into bytecode for a custom virtual machine (VM) deployed together with the protected program. Combined with software diversity (see Section 1.3.4), each program instance can be deployed with an individual VM [26].

Ultimately, obfuscated software is shipped to all users in an identical representation. Consequently, once an adversary successfully deobfuscates one program, the protection for all programs becomes ineffective.

Weird machines. *Return-oriented programming (ROP)* is an attack technique that executes arbitrary logic by chaining existing instruction sequences ending in returns, avoiding the need to inject code [27]. Several approaches repurpose ROP for hiding computations [28–31]. These techniques can be viewed as forms of *weird machines*—unconventional computation models built on unintended program behavior [32]. Except for ROPfuscator [28]—a compiler pass—they operate purely at the binary level, limiting their applicability and exposing behavior at the architectural level, where it is observable by, e.g., debuggers.

Evtyushkin et al. formalized the concept of *microarchitectural weird machines* [33], which exploit microarchitectural entities used for speculative execution like caches and branch predictors to encode and process data without leaving architectural traces. They construct *weird registers* from cache states and build *weird gates* that manipulate these based on speculative behavior, chaining them into *weird circuits*. These designs demonstrate Turing-complete computation but face practical limitations: complex operations require large circuits of weird gates, and data propagation between gates must either rely on (i) *weird wires*, limiting options, or (ii) sacrifice stealth by lifting to the architectural level.

ExSpectre [34], by contrast, embeds malicious code in speculative execution paths that are not executed during normal program operation. Nevertheless, this code is still present in the binary and can be identified by analysis techniques that explicitly trace speculative execution paths [35–37].

Anti-debugging. *Anti-debugging* [38] aims to prevent an attacker from performing dynamic analysis. Software can detect whether it is being debugged by side effects of the process. Upon detection of a debugger, the software can, e.g., terminate itself, circumvent execution of an algorithm containing knowledge, or behave abnormally.

Nanomites [39] fork a process of a protected program. One process, the *tracer*, attaches itself to the other process, the *tracee*. Tracer and tracee communicate with each other using signals, and the tracer deobfuscates the tracee's code in memory, which is required for correct execution. Additionally, only one tracer, or debugger, can be attached at a time [8, 40], hence the approach also protects the tracee by debug blocking. Thus, an attacker cannot simply dynamically analyze the protected process.

1.3.3 Tamper Protection

Although not the focus of this dissertation, tamper protection is crucial not only to maintain safe and secure execution of programs but also to keep protections against copying and reverse engineering in place. For example, an attacker might try to remove a license check from a program to break its copy protection. Run-time *integrity checks* [41, 42] can increase the effort required for tampering and dynamic analysis by forcing attackers to identify and neutralize multiple, potentially interdependent self-checks.

Signing [13] binaries and verifying their signature can prevent tampering but only works as long as an adversary cannot remove the verification procedure. Thus, a defender might resort to *self-checking* [43, 44], where a program checks itself during its execution. We can distinguish between static self-checking, where validation is performed at program start, and dynamic self-checking, where integrity is checked throughout the run time of the

program. Various existing methods typically compute checksums for parts of a program at run time and compare them to predetermined checksums embedded in the program. Mutually checking code regions can safeguard against removal of the validation procedure, and restoring illegitimately modified code guarantees intended execution [45]. However, self-checking usually has to be combined with protections against reverse engineering to present an obstacle to an attacker.

Control-flow integrity (CFI) [46, 47] enforces that the control flow of a program follows a precomputed control-flow graph, preventing attackers from redirecting execution to unintended code paths. This is particularly effective against code-reuse attacks such as ROP, which often rely on tampering with control-flow data. CFI's strength lies in constraining the run-time attack surface rather than acting as a tamper-resistant mechanism. It therefore complements integrity checks by limiting the set of viable control-flow manipulations during an attack.

Another complementary approach is *remote attestation* [48], which allows a trusted verifier to attest the integrity of a remote system. Typically, the system generates a cryptographic proof—often based on measurements of its software state—that the verifier can check against expected values. This technique is particularly useful in distributed environments where local integrity checks alone are insufficient. *Run-time attestation* [49, 50] extends this concept by continuously monitoring the execution state of a program, enabling detection of dynamic tampering or control-flow deviations. These methods are often supported by trusted hardware components such as Trusted Platform Modules (TPMs) or trusted execution environments (TEEs), which serve as a root of trust (RoT) by performing cryptographic measurements of the software stack and providing hardware-signed evidence to external verifiers during the attestation process.

1.3.4 Software Diversity

Economies of scale resulting from software monoculture favor attackers over defenders—one exploit can affect billions of systems, facilitating tampering. Software diversity, introduced by Cohen in 1993 [51], addresses this problem by generating unique variants of a program for each deployment. The goal is to reduce exploit reuse by ensuring that no two software instances are identical. Software diversity can be applied pre-distribution—during implementation, compilation, or linking—and post-distribution, i.e., during installation, loading, or execution [52].

Pre-distribution approaches primarily target fault tolerance and mitigate memory corruption, code reuse, tampering, and reverse engineering. Methods include *N-version programming* [53] and *code and data diversification* [54]. Fully automated diversity is inherently limited because it must preserve program semantics, leaving high-level transformations such as algorithmic changes to human designers. Pre-distribution diversity has been criticized as logistically infeasible due to the difficulty of distributing individualized binaries. However, advances in automated software distribution and cloud-based delivery have largely mitigated this limitation [55, 56].

In contrast, post-distribution techniques often involve dynamic transformations to prevent code reuse, code injection, and just-in-time (JIT) attacks. Examples include *instruction set randomization* [57] and *address space layout randomization (ASLR)* [58]. Post-distribution diversity increases the difficulty of dynamic analysis and tracing. However,

since the underlying binary remains identical across deployments, its effectiveness against large-scale reverse engineering is limited compared to pre-distribution diversity.

Despite its theoretical benefits, software diversity faces adoption hurdles beyond the logistics of binary distribution, including unacceptable performance overheads introduced by some approaches. Additional obstacles include incompatibility with code signing due to binary modification and increased complexity in error reporting caused by changes in program layout [52].

1.3.5 Hardware–Software Binding

Hardware–software binding ties the correct execution of software to unique properties of the underlying hardware so that the software behaves correctly only on authorized devices. Early examples encrypt software with keys derived from physically unclonable functions (PUFs) [59, 60]. A PUF yields device-unique responses to challenges due to manufacturing variability; cf. Section 2.3.4. These responses serve as cryptographic keys so that the protected software can only be decrypted on the hardware it is bound to. Building on these foundational concepts, various approaches have been proposed to implement hardware–software binding, which we discuss in the following.

SEPP [61] is a RISC processor that decrypts encrypted program instructions in its instruction pipeline using a key derived from a ring oscillator PUF [62]. SEPP thus averts revealing keys and decrypted code and additionally prevents code injection by only executing code encrypted for the specific processor. However, this approach does not only need a special processor but also dedicated PUF hardware.

HCIC [63] uses two control flow checking mechanisms to prevent code-reuse attacks (CRAs) such as ROP and jump-oriented programming (JOP): (i) checking the encrypted Hamming distance between PUF responses and return addresses, and (ii) linearly encrypting PUF responses and instructions at jump target addresses. This hinders static analysis and ensures that only legitimate paths can be executed, but does not prevent dynamic analysis or protect data.

Zhang et al. [64] propose a lock mechanism to protect field-programmable gate array (FPGA) designs. They use a PUF response to control an added finite-state machine (FSM)—only if this FSM reaches an unlocking state, the original code is executed. Bossuet et al. [65] argue that the proposed mechanism suffers from a security–overhead trade-off, where low-overhead configurations are vulnerable to brute-force attacks while secure configurations cause high performance overheads. To address this, Zhang et al. [66] proposed adding self-looping states and increasing the number of inputs to ensure resilience against brute-force attacks while minimizing the overhead.

HESP [67] binds software to hardware by determining jump addresses at run time through queries to a dynamic random-access memory (DRAM) PUF (cf. Section 2.3.4) whose responses depend on fine-grained timing characteristics of the protected software. The protected binary also incorporates self-checking inspired by prior work [68]. However, an attacker can compare multiple versions of the protected program—differences hint at the self-checking and can thus be eliminated. The DRAM PUF used by HESP relies on a kernel module to access low-level timing and raw memory state, which reduces stealthiness and enables record–replay attacks.

BROL [69] combines PUF-based encryption with cache-only execution to protect software on commodity hardware. At load time, a preloader decrypts the binary and executes it on dummy data to populate the CPU's cache and uses physical memory aliasing to erase the plaintext code from main memory. The program then executes exclusively from the cache, rendering memory dumping and most dynamic analysis techniques ineffective. BROL achieves strong protection without dedicated security hardware, but the size of the protected code is constrained by the cache capacity, which limits applicability in complex workloads.

Frank et al. [70] propose an FPGA-based data-binding mechanism using a PUF response derived from the same memory the data is stored on. The location of the PUF cells is hidden by random selection depending on a user-specific key, hindering modeling attacks. This mechanism can be used both to preserve the confidentiality of data and to restrict execution of software to a specific device. However, their approach requires specialized hardware, as it relies on a custom memory controller to intentionally violate memory timing specifications to trigger bit flips for PUF measurements.

From a security perspective, hardware–software binding raises the attacker's cost but does not make reverse engineering impossible. Attackers may attempt to emulate or model the PUF or replay recorded responses. Therefore, binding schemes are most effective when combined with additional protections, e.g., code diversification, integrity checks, and tamper-resistant hardware RoTs.

1.4 Contributions

Existing software protection techniques each address specific threats but have inherent limitations. Obfuscation and packing can slow down analysis, yet they offer no defense against copying. Anti-debugging mechanisms and run-time self-checks increase the effort required for dynamic analysis but can be bypassed by determined adversaries. Hardware dongles are external components that must be connected at run time, which may be unsuitable for some deployment scenarios, and their transferability allows execution on different systems once the dongle is obtained or cloned. License-key systems enforce authorized execution but typically rely on a single enforcement point, such that bypassing the check or extracting a secret compromises the protection. Software diversity reduces the impact of software monoculture by producing unique binaries, but it does not prevent copying or reverse engineering of individual binaries.

This work thus establishes a conceptual protection model integrating multiple protection strategies to provide comprehensive defense against copying and reverse engineering, as depicted in Figure 1.4: Hardware–software binding raises the cost of copying by tying execution to device-specific properties. Combined with software diversity techniques, we can raise the bar for attackers attempting reverse engineering or tampering.

The central contribution of this dissertation is the demonstration that intended software behavior can be made to depend on device-specific physical properties. This shift enables protection mechanisms in which copying or analyzing the software yields valid but unintended behavior. Building on this insight, we develop a conceptual protection model and instantiate it in five approaches that protect different aspects of software across diverse hardware platforms.

In detail, this dissertation makes the following contributions:

- We analyze risks of technical knowledge exfiltration from software and identify corresponding mitigation strategies (Sections 1.2, 1.3 and 3.2).
- We propose a conceptual protection model combining the strengths of complementary protection mechanisms, i.e., hardware–software binding and software diversity (Chapter 3).
- We systematize hardware fingerprinting techniques and their underlying hardware platforms (Chapter 2), demonstrate the suitability of Rowhammer as a hardware fingerprinting mechanism (Section 4.1), and present two methods to fingerprint SRAM (Section 5.1).
- We introduce five concrete approaches that refine our conceptual model for specific scenarios and binding targets—program code and data, as well as ML models—in Chapters 4 to 6:
  1. GlueZilla binds program behavior to instruction-level transformations using Rowhammer-induced bit flips (Section 4.2).
  2. Angluerus applies Rowhammer-based transformations to program data (Section 4.2).
  3. Minotaurox uses hardware-extracted fingerprints to dynamically determine program control states (Section 5.2.1).
  4. Sphinxor decodes secret data at run time based on hardware-derived fingerprints (Section 5.3.1).
  5. Locki protects the execution of ML models by decrypting model parameters using a hardware fingerprint (Section 6.1.1).
- We provide reference implementations for our approaches (Sections 4.3, 4.4, 5.2.2, 5.3.2 and 6.1.2), see also `https://github.com/COMET-DEPS`.

We evaluate the effectiveness of these approaches and their implementations, demonstrating that our concept significantly increases the attacker's effort while maintaining correct execution on the target device (Chapter 7). Finally, we conclude this dissertation by summarizing the main results, outlining promising directions for future research, and sharing lessons learned (Chapter 8).

Alles Gescheite ist schon gedacht worden,
man muss nur versuchen, es noch einmal zu denken.

All intelligent thoughts have already been thought;
what is necessary is only to try to think them again.

— Johann Wolfgang von Goethe

2 Background

Contents

2.1 Dynamic Random-Access Memory . . . 13
2.1.1 DRAM Organization . . . 14
2.1.2 DRAM Addressing . . . 15
2.1.3 Memory Allocation . . . 15
2.2 Rowhammer . . . 15
2.2.1 Memory Massaging . . . 16
2.2.2 Rowhammer Exploits . . . 17
2.2.3 Rowhammer Mitigations . . . 17
2.2.4 Advanced Rowhammer Techniques . . . 18
2.3 Hardware Fingerprinting . . . 18
2.3.1 Dongles . . . 20
2.3.2 Trusted Platform Modules . . . 20
2.3.3 Trusted Execution Environments . . . 20
2.3.4 Physically Unclonable Functions . . . 21
2.3.5 Rowhammer-based Fingerprinting . . . 21

In this chapter, we introduce the prerequisite for software–hardware binding, hardware fingerprinting, allowing us to identify a specific device. Our approaches described in Chapter 4 use Rowhammer for fingerprinting. Rowhammer is a hardware fault of DRAM, we thus first describe DRAM, continue with Rowhammer and finally discuss how we can use Rowhammer for fingerprinting, besides other methods.

2.1 Dynamic Random-Access Memory

DRAM serves as main memory in many computing platforms, including PCs and embedded systems [71]. JEDEC standardizes DRAM, with the most recent standards being DDR4 [72] and DDR5 [73]. In this section, we describe how DRAM is organized and addressed, as well as how operating systems manage memory.

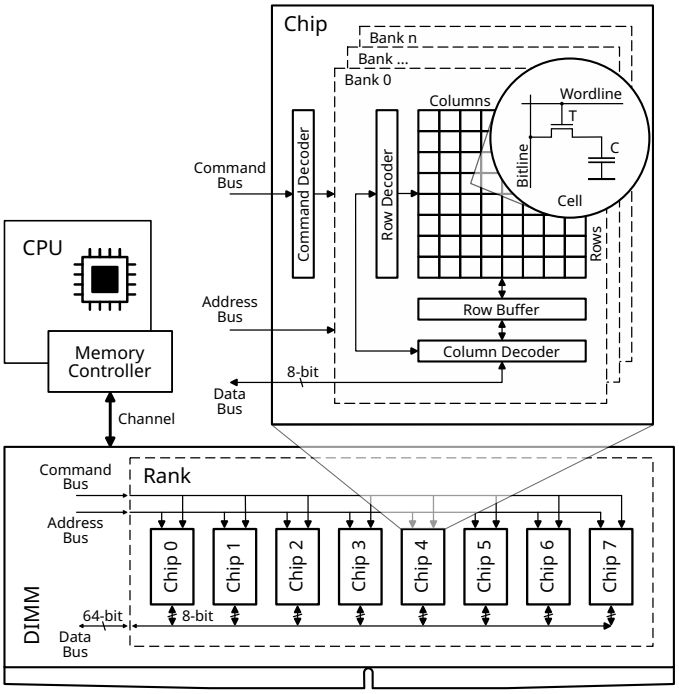


Figure 2.1: Schematic overview of DRAM organization showing an individual memory module (DIMM) and its connection to the CPU, DRAM chips, and a DRAM cell consisting of a storage capacitor (C) and an access transistor (T).

2.1.1 DRAM Organization

Figure 2.1 shows how DRAM is typically organized. The memory controller of the CPU, responsible for communication with DRAM, is connected to dual inline memory modules (DIMMs) via channels consisting of multiple buses. Each memory module consists of one or more ranks, which contain multiple chips each. Ranks are logically organized in banks, which spread across all the rank's chips. A bank consists of a matrix of cells, row and column decoders for address selection, and a row buffer. The cells in a row of the matrix are connected via the wordline, the cells in a column are connected with the bitline.

Each cell stores one bit of data, represented by a charge in a capacitor. A logical 1 can be represented by a charge above a certain threshold value (in true-cells) or below it (in anti-cells) and vice-versa for a logical 0. A capacitor losing its charge thus results in a change of its stored value, a so-called *bit flip*. In true-cells, the value changes to 0 ($1 \rightarrow 0$), in anti-cells to 1 ($0 \rightarrow 1$). To prevent discharging, the charge in all capacitors must be *refreshed* regularly, typically every 64 ms [71].

If the memory controller wants to access data in a row, it must first *activate* it. If the access transistor of a cell is activated via the wordline, the charge stored in the capacitor is transmitted to the row buffer via the bitline. The contents of the row buffer can then be read or overwritten by the memory controller, typically corresponding to a cache line, which is 64 bytes on x86. If another row of the same bank should be activated, the contents of the row buffer are written back to the active row by recharging its capacitors and only then the new row's content is transferred to the row buffer.

2.1.2 DRAM Addressing

The CPU's memory controller translates virtual addresses to physical addresses. Using so-called *mapping functions*, a row mask, and a column mask, the physical addresses are then translated to DRAM addresses consisting of a channel, module, rank, bank, row, and column address. Each mapping function combines bits of the physical address, typically using a bitwise exclusive or (XOR). The results of all mapping functions combined identify one specific bank of memory used in the system. Decoders within the DRAM bank use the row and column addresses to select the corresponding row and column. The DRAM address only identifies individual bytes; bits cannot be addressed directly.

The mapping functions are typically not documented publicly and depend on the memory controller and the DRAM's organization. Nevertheless, several tools that reverse engineer the mapping functions for certain memory controllers are available [74–78].

2.1.3 Memory Allocation

Most operating systems use virtual memory, organized in pages of 4 KiB. These pages fit into physical frames of the same size, thus are not only virtually but also physically contiguous. If a piece of physically contiguous memory larger than a page is required, hugepages [79] can be used, which are typically available in the sizes of 2 MiB and 1 GiB on x86.

Many operating systems use a *buddy allocator* to manage virtual memory. The buddy allocator divides memory into blocks of sizes that are powers of two and splits or merges them when processes (de)allocate memory. This allocation strategy is efficient but also predictable to some extent, enabling memory massaging as described in Section 2.2.1.

2.2 Rowhammer

Rowhammer [80] is a disturbance error in DRAM. Rapidly accessing rows, also referred to as *hammering*, can speed up charge leakage in cells of adjacent rows, leading to flips of their logical bit values *before* the cells are refreshed. This is likely due to voltage fluctuations of the wordline disturbing the charge in adjacent rows. In the literature, the rows being hammered are the so-called *aggressor rows*, while the adjacent rows that potentially show bit flips are called *victim rows*.

Repeatedly accessing the same data typically does not reactivate the row in which the data is stored. Instead, the data is read from the row buffer or the CPU's cache. To apply Rowhammer effectively, we must not only (i) repeatedly access a row but also, between accesses, (ii) close the row and (iii) invalidate the corresponding cache line. Listing 2.1 shows these basic steps of a Rowhammer attack. We can close a row by activating another row of the same bank because they share the same row buffer and only one row can be loaded into the row buffer at a time. We can determine whether two rows are on the same bank, either probabilistically [81] or using the mapping functions described in Section 2.1.2. There are also multiple methods to invalidate cache lines, e.g., using the x86 instruction `clflush` [81] or cache eviction [82].

Rowhammer-induced bit flips tend to be reproducible, enabling their use in exploits [83]. Because Rowhammer operates below the memory protection abstractions defined by the

```
void hammer(Addr[] rows, int rounds) {
  // ensure prerequisites to overwrite row buffer (ii)
  assert(|rows| ≥ 2)
  assert(forall row in rows: row.bank == rows[1].bank)

  for 1 .. rounds {
    for row in rows:
      access(row) // (i, ii) load (new) row into row buffer
    for row in rows:
      flush(row)  // (iii) invalidate cache lines
  }
}
```

Listing 2.1: Basic steps of a Rowhammer attack.

operating system and enforced by the CPU, attackers can manipulate data undetected without explicit write access to the affected memory locations. In other words, Rowhammer enables effective *writes* to a privileged process's memory page while requiring only *read* access to pages of an unprivileged process, provided that the pages of both processes reside in physically adjacent rows. The latter can be achieved through memory massaging, which is a prerequisite for many Rowhammer exploits. We proceed with overviews of memory massaging, Rowhammer exploits, potential mitigations, and techniques to evade them.

2.2.1 Memory Massaging

Memory massaging refers to the manipulation of the physical memory layout such that an attacker-selected page containing critical data resides in a physical frame affected by Rowhammer. Typically, an attacker achieves this by forcing the memory allocator into a predictable state. On systems such as Linux, which use a buddy allocator for physical memory management, this predictable state can be exploited. Buddy allocators manage memory in power-of-two-sized blocks and deterministically split and merge adjacent blocks, allowing attackers to influence physical frame reuse through controlled allocation and deallocation patterns. As a result, attackers can exhaust available memory and then deallocate a specific page residing in a vulnerable physical frame. By subsequently triggering a page (re)allocation in the victim process, the attacker can cause the page to be placed, with high likelihood, into the vulnerable frame.

Prior work describes several memory massaging methods, most notably:

- *Memory spraying* involves filling memory with attacker-controlled pages and reclaiming or exhausting memory to induce reuse of frames. Seaborn et al. [81] filled large parts of memory with attacker-controlled page tables by repeatedly mapping a file into memory, increasing the probability that a Rowhammer-induced bit flip targets a page table entry. Gruss et al. [82] demonstrated this in JavaScript environments.
- *Flip Feng Shui* [84] leverages page deduplication to colocate attacker and victim pages.
- *Phys Feng Shui* [85] exhausts available memory to bring the memory allocator into a predictable state and then forces the allocator to place the victim page in a frame vulnerable to Rowhammer.

- *Memory waylaying* [86] involves repeatedly allocating and deallocating memory to align victim data in a specific physical layout. The attacker uses a side channel to infer the frame the victim page is located in.
- *Memory chasing* [86] exploits `fork` and cache eviction, making it faster than memory waylaying at the cost of not being stealthy, i.e., potentially detectable by the operating system.

2.2.2 Rowhammer Exploits

Most successful Rowhammer exploits take three steps:

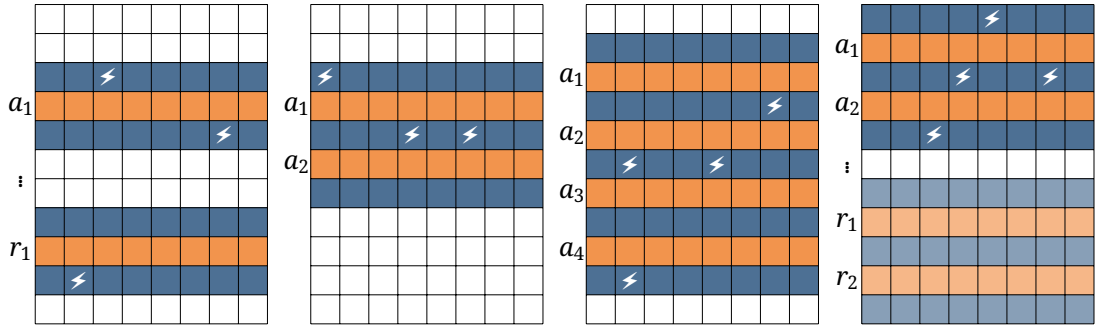
1. The attacker probes the DRAM for Rowhammer-susceptible cells using a process called *memory templating* [85]. The result of this process is a list of vulnerable cells along with their corresponding aggressor rows and flip directions ($0 \rightarrow 1$ or $1 \rightarrow 0$).
2. The attacker loads the page in which they want to induce a bit flip into a vulnerable frame via *memory massaging*.
3. The attacker starts *hammering* the aggressor rows and causes bits to flip in the victim process's newly (re)allocated page.

Several security researchers demonstrated Rowhammer-based exploits, e.g., to gain kernel privileges [81], extract sensitive information from a victim VM using a malicious co-hosted VM [87], gain unauthorized access to co-hosted VMs [84], or to escape from browser sandboxes [82, 87, 88]. Kwong et al. [89] even showed that Rowhammer can be used as a read primitive to leak sensitive information of other processes. GPUHammer [90] demonstrates accuracy degradation attacks on ML models by inducing bit flips in GDDR6 DRAM used in GPUs.

2.2.3 Rowhammer Mitigations

Rowhammer was first shown to be present in DDR3 DRAM [80]. Manufacturers aim to increase the structural density of DRAM chips, which exacerbates their susceptibility to Rowhammer [91–93]. A straightforward mitigation approach is to increase the refresh rate of memory cells, thereby narrowing the time windows for Rowhammer [91]; however, this causes significant performance and energy overheads. Therefore, the DDR4 standard [72] describes target row refresh (TRR), a technique that intends to detect Rowhammer attacks and prevent bit flips by refreshing potential victim rows more often [94]. The DDR5 standard [73] refines this mechanism and additionally recommends implementing ECC.

Due to the limitations of hardware-based mitigations, particularly for already deployed DRAM chips, software-only approaches remain of interest. Several proposed mitigations aim to increase access randomness, enforce spatial separation, or monitor anomalous memory access patterns. For example, CATT [95] extends the memory allocator to prevent bit flips from crossing privilege boundaries. ANVIL [96] detects high-frequency memory accesses leveraging performance counters and refreshes nearby potential victim rows. Finally, MAD [97] introduces entropy to memory allocation to hinder memory massaging and consequently deters Rowhammer exploits.



(a) Single-sided pattern **(b)** Double-sided pattern **(c)** Many-sided pattern **(d)** Non-uniform pattern

Figure 2.2: Comparison of hammer patterns. Aggressor rows (a_n; r_n denotes randomly picked aggressor rows) are highlighted in orange and victim rows in blue, lighter colors signify hammering at lower frequency.

2.2.4 Advanced Rowhammer Techniques

Advanced Rowhammer techniques were able to evade the hardware-based mitigations, e.g., TRRespass [75] and Blacksmith [98] for DDR4 DRAM. Both tools work by circumventing mechanisms to track suspicious row accesses indicating hammering. Figure 2.2 depicts different hammer patterns; TRR invalidates single-sided (a) and double-sided patterns (b). TRRespass simply hammers a larger number of rows at once using many-sided (or n-sided) hammer patterns (c). This approach does not circumvent all TRR implementations. Therefore, Blacksmith uses non-uniform hammer patterns (d). These patterns access additional aggressor rows at various frequencies to undermine TRR, which works on a wider range of memory modules when compared to TRRespass.

ECCploit [99] shows that even memory equipped with ECC remains vulnerable, as bit flips can still be induced under specific conditions. By using a side channel in the ECC memory controller itself, attackers can inject reliable bit flips while evading detection. ZenHammer [78] and Phoenix [100] further demonstrate that DDR5 DRAM, despite refined mitigation mechanisms and increased refresh rates, remains susceptible to Rowhammer. These findings highlight that Rowhammer remains effectively unmitigated even in modern DRAM chips. However, a large-scale study of fully automated Rowhammer attacks reported that bit flips were induced in only 12.5% of tests conducted on 822 unique systems [101]. Most tests failed before executing the Rowhammer tools because the required mapping functions could not be reverse engineered successfully.

2.3 Hardware Fingerprinting

The prevalent paradigm in software diversity (cf. Section 1.3.4) is to be *hardware agnostic* [52]. While each user receives a unique binary, this binary could as well be executed on any other user's hardware. While offering protection against tampering and reverse engineering, software diversity does not hinder copying. By combining software diversity with hardware–software binding, we can overcome this limitation.

In order to bind software to hardware, it is necessary to uniquely identify a specific device. Several *hardware fingerprinting* techniques exist, providing a root of trust (RoT)

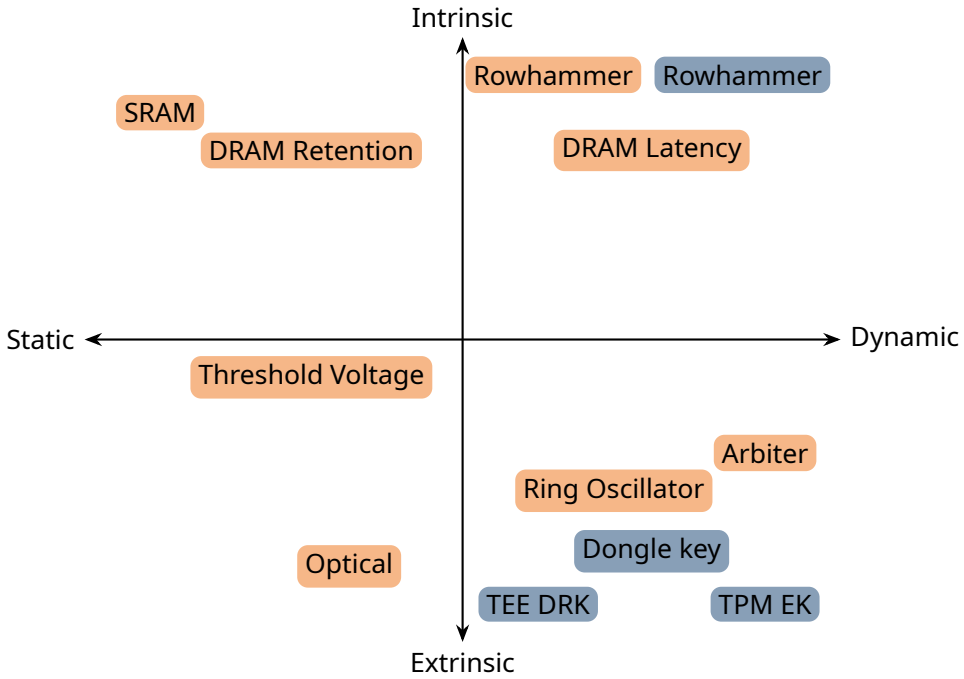


Figure 2.3: Comparison of techniques for hardware fingerprinting: PUFs and others.

for hardware–software binding. Figure 2.3 illustrates the hardware fingerprinting techniques discussed in the following sections. We position these techniques along two axes: *intrinsic–extrinsic* and *static–dynamic*.

Definition 3: Intrinsic and Extrinsic Hardware Fingerprinting

An *intrinsic* hardware fingerprinting technique derives the device fingerprint from physical properties inherent to existing circuits or components, whereas an *extrinsic* technique requires the integration of additional circuitry or components to produce a measurable response.

Definition 4: Static and Dynamic Hardware Fingerprinting

A *static* hardware fingerprinting technique cannot be accessed during program run time, whereas a *dynamic* technique can be queried during execution.

Many techniques lie along a continuum rather than at the extremes of these spectra. For example, the threshold voltage PUF [102] exploits intrinsic device properties but still requires extrinsic circuitry for measurement. Similarly, DRAM retention PUFs [103–105] can be queried during program execution, but doing so may incur substantial overhead or even risk destructive effects on the memory state, making it unsuitable as a dynamic hardware fingerprint. In the following sections, we discuss existing hardware fingerprinting techniques.

2.3.1 Dongles

Hardware dongles [42, 106] can act as external RoTs for hardware–software binding. Software can encrypt critical code fragments, configuration data, or decryption keys using the dongle's secret key. At run time, the protected software queries the dongle and performs the necessary decryption. As the key never leaves the dongle, execution is possible only when the legitimate dongle is connected. However, this does not prevent unauthorized use of the software on another device, since the dongle is device-agnostic and can be connected to any other device, allowing software to be executed there. Furthermore, dongles' external nature facilitates their cloning, e.g., by intercepting their hardware interface [18].

2.3.2 Trusted Platform Modules

TPMs [107] are dedicated cryptoprocessors, although their functionality can also be implemented in firmware. They serve as hardware RoTs by providing device-unique keys, secure storage, and cryptographic operations in a tamper-resistant environment. A TPM features a device-unique endorsement key (EK) and supports *sealing*, allowing software to encrypt its decryption keys or configuration data such that only the same TPM in a specific platform configuration can unseal them. At run time, the protected application requests the TPM to unseal the data and continues execution only if the platform state matches the expected measurements. This mechanism enables hardware–software binding by tying software execution to both the specific device and its verified configuration.

TPMs typically operate as external co-processors, so communication incurs latency [108] and could be intercepted or emulated via bus-level attacks [109–111]. Sealed data can become inaccessible if legitimate firmware or software updates change the platform configuration, complicating maintenance [112]. The cryptographic operations themselves are relatively slow [108], making fine-grained or per-instruction binding impractical. Finally, TPMs are not universally available [108], limiting the applicability of this protection method.

2.3.3 Trusted Execution Environments

TEEs, such as Intel SGX [113] or Arm TrustZone [114], are hardware-based environments that provide isolated execution and hardware-rooted device keys, e.g., a device root key (DRK). Unlike TPMs, which only expose fixed cryptographic primitives, TEEs allow software to execute sensitive code or handle confidential data inside a protected enclave, which cannot be accessed or tampered with by the operating system or other software. A program can place decryption logic or critical functionality inside the enclave so that code or data is decrypted only after successful attestation of the platform identity and verification of the enclave measurement. This enables hardware–software binding at run time by tying the software's behavior to both the genuine device and the specific enclave instance.

Enclaves typically have constrained memory and restricted APIs, which can make them unsuitable for large or time-critical workloads. Transitions between normal and secure execution worlds introduce latency, side-channel or microarchitectural attacks have demonstrated the potential leakage of secrets [115], and interposer devices can be used to break their security [116]. Furthermore, TEEs are not universally available, particularly on legacy or resource-constrained systems, limiting the applicability of this protection approach [117].

2.3.4 Physically Unclonable Functions

PUFs are a hardware security primitive that uses physical properties of a device to generate a unique response [118]. PUFs implement a challenge–response protocol, where the challenge is a random value, and the response is generated by the PUF using the underlying hardware. PUFs are typically used to identify a device, e.g., for authentication. Therefore, the PUF response is compared to a reference value determined beforehand to decide whether the device is authentic.

Many different types of PUFs exist [102, 119]. Extrinsic PUFs rely on dedicated hardware, e.g., the ring oscillator PUF, the arbiter PUF, and the optical PUF. Intrinsic PUFs, e.g., PUFs based on SRAM [60, 119, 120], DRAM [103–105, 121, 122] and flash [123, 124], are based on standard hardware and rely on unique physical properties of circuits due to unintended deviations in the manufacturing process. Because of their physical nature, which can be influenced by environmental changes, PUF responses are often noisy and thus require some sort of error correction. Over the past two decades, researchers have proposed several uses for PUFs, including identification and authentication [125, 126], cryptographic key storage [127, 128], and hardware–software binding [67, 68].

SRAM PUFs. One specific way of implementing a PUF is using the startup behavior of SRAM chips, in particular the preferred startup values of the individual SRAM cells [60, 119, 120]. Essentially, powering any given SRAM chip causes a random pattern to appear among its individual cells, where some cells power up to a 1 and the others to a 0. Except for a small amount of noise, the emerging pattern is the same each time that the chip is powered on [129]. Each SRAM chip shows a different pattern, making individual chips distinct, allowing construction of PUFs.

DRAM PUFs. DRAM stores bits of information in DRAM cells, which must be refreshed regularly to not lose its charge. DRAM retention PUFs [103–105] utilize this behavior, as they are based on whether individual DRAM cells lose their charge within a specific period of time when they are not refreshed, which varies from cell to cell. Another possible implementation is the DRAM Latency PUF [122], which deliberately violates timings for certain DRAM operations, e.g., reading from DRAM, to cause errors that are used for the PUF response. DRAM PUFs can also be based on startup values and Rowhammer [130].

2.3.5 Rowhammer-based Fingerprinting

The Rowhammer PUF [121, 131] is based on DDR2 DRAM and an FPGA, and uses Rowhammer to generate responses. The authors validated their PUF's uniqueness and robustness and showed that its entropy is high enough to derive cryptographic keys from. Their design uses a fixed memory area and requires a kernel module for PUF interaction. They use the Jaccard index to compare PUF responses, which eliminates the need for identical responses and tolerates a certain level of noise in the responses.

FPHammer [132] and FP-Rowhammer [133] both strive to identify devices by applying Rowhammer. Both are intended to be tracking mechanisms for devices, e.g., used for browser fingerprinting. FPHammer and FP-Rowhammer use DDR4 DRAM and Intel CPUs.

Li et al. found the Jaccard index to be too unreliable for device identification and thus modify the Jaccard index for FPHammer to be less susceptible to noise [132]. This makes their approach work better to identify DRAM modules but less suitable to identify whole PC systems as we discovered [134]. Venugopalan et al. use a different approach in FP-Rowhammer, where they hammer a chunk of memory repeatedly and use an empirical probability distribution to overcome differences in the individual results [133]. They then use the Jensen-Shannon divergence to compute the similarity of this distribution to a previously extracted distribution.

Mens et corpus una eademque res sit,
quae iam sub cogitationis,
iam sub extensionis attributo concipitur.

The mind and the body are one and the same thing,
which is conceived now under the attribute of thought,
now under the attribute of extension.

— Baruch Spinoza (Benedictus de Spinoza)

3 Threat Model and Protection Concept

Contents

3.1 Scenarios . . . 24
3.2 Threat Model . . . 25
3.3 Conceptual Protection Model . . . 27
3.4 Building Blocks . . . 28
3.4.1 Hardware Fingerprinting . . . 28
3.4.2 Transformations . . . 29
3.4.3 Constructing Scenario-Specific Approaches . . . 30

Usually, source code written in a compiled programming language is translated into machine code and stored as a binary image. This binary is then deployed to any compatible device it should be executed on and stored in the device's persistent storage. Once launched, the binary image is loaded into memory and remains there throughout its run time. Let us assume an attacker may obtain the binary, e.g., by accessing storage or memory of a device it is deployed on, or by intercepting its transfer, such as during an update. Given the binary, an attacker can copy the program to another compatible device and execute it or attempt to reverse engineer it to extract the contained knowledge, as described in Section 1.2.

Software can be protected using one or more of the protection mechanisms described in Section 1.3. However, industrial control software frequently lacks such protections, partially rooted in limitations of existing protections. For example, the hardware in industrial machines is often resource-constrained, limiting the applicability of traditional software protection techniques. Many legacy machines remain operational today, having been built when hardware lacked dedicated security features.

In this chapter, we first present the main three industrial scenarios considered in this dissertation. We then describe the capabilities of an adversary attempting to copy or reverse engineer software in these scenarios. Next, we introduce our conceptual protection model, which uses hardware–software binding to hinder copying and reverse engineering. Finally, we briefly discuss possible instantiations of this model for each of the three scenarios, which we later elaborate into concrete approaches in Chapters 4 to 6.

3.1 Scenarios

We consider three deployment scenarios in which industrial software requires protection. Each scenario specifies the hardware platform, software stack, and operational constraints. These assumptions determine which attack vectors are realistic (cf. Section 3.2) and which protection mechanisms are applicable (cf. Section 3.4). The scenarios were provided by industrial partners within the Dependable Production Environments with Software Security (DEPS) project [135] and serve as a point of reference throughout the remainder of this dissertation.

Scenario 1: PC with DRAM. We consider PCs equipped with x86-64 CPUs and DDR3 or DDR4 DRAM. We assume the DRAM modules are susceptible to Rowhammer. Prior work has shown that most DDR3 and DDR4 DRAM modules are vulnerable to Rowhammer attacks [92, 93, 98, 99]. Given that no comprehensive or general Rowhammer mitigation is implemented to date [75, 78, 83, 98, 99], we consider this a realistic assumption.

The PCs use Linux as their operating system, allowing execution of arbitrary programs, potentially with elevated privileges. Deployed user-space binary-only programs—implemented, e.g., in Structured Text or C—can control industrial machinery. We focus on this scenario in Chapter 4 and propose two alternative software protection approaches.

Scenario 2: MCU with SRAM. We look at Arm-based MCUs, using flash memory and SRAM. They have essential security features, such as read and write protection and a memory protection unit. Intercepting the buses between the processor and SRAM requires substantial physical intervention, such as chip scraping, since both components are integrated on a single chip. It has been shown that MCUs' firmware protection can be defeated by a sufficiently motivated attacker [136], but this is outside the scope of this dissertation.

The system runs bare-metal without an operating system, i.e., the hardware executes a single monolithic binary with exclusive access to all resources. The program we aim to protect implements a control algorithm for industrial machinery and is typically written in C. Because malfunction of the machine is potentially dangerous, we need to ensure the applied protection does not violate its safety properties. Chapter 5 presents two approaches for this scenario.

Scenario 3: NN models. We consider the deployment of NN models executed via Python, e.g., using TensorFlow, on edge devices or industrial machines. The hardware provides a RoT, such as a TPM or PUF. The RoT can be used for secure key storage, ensuring an adversary cannot extract the key from storage. In Chapter 6, we propose a concrete approach to protect NN models.

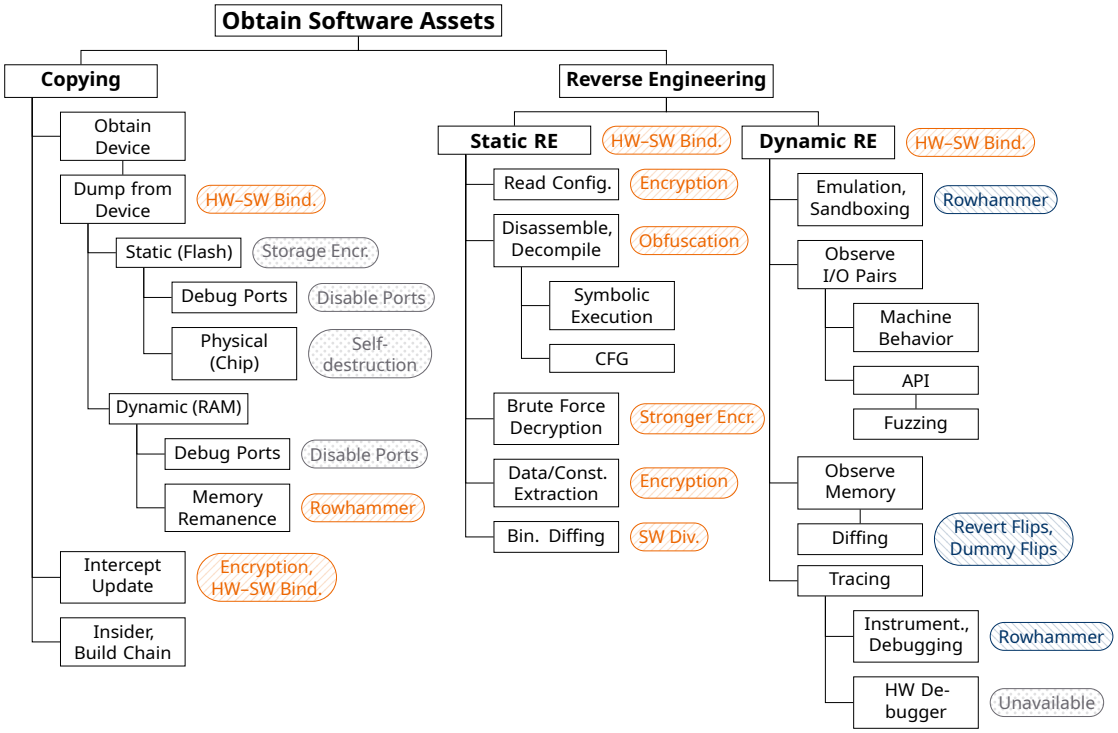


Figure 3.1: Attack–defense tree showing ☐ attacks, ⊘ defenses against copying and reverse engineering on cloned devices, ⊘ defenses against dynamic analysis on the target device, and ⊘ supplementary defenses outside the scope of this dissertation. Note that the attacks are not exclusive and can be combined in any way.

3.2 Threat Model

The adversary aims to obtain software assets, i.e., gain unauthorized control of the software itself or parts of it, as illustrated in Figure 3.1. We distinguish two strategies: the adversary may (i) simply obtain a binary of the software and copy this binary to a cloned device to replicate the original machine's behavior, or (ii) extract knowledge from the software or parts of it by means of reverse engineering, potentially to create a software clone.

Let us first consider copying. If adversaries can obtain a device containing the program, they can try to extract it from the device. They can do so by trying to access the binary stored on the device, e.g., in flash, or access the process image, i.e., the program loaded into memory. Copying the stored binary may be done by using debug ports or accessing the storage itself, e.g., removing it and connecting it to a different device. The process image can potentially also be extracted using debug ports or by gaining access to the device and trying to copy the memory contents, including memory remanence attacks. An attacker can also opt to intercept an update process to obtain the binary or at least parts of it. Insiders or attackers targeting the build chain, including the compilation, linking, and deployment infrastructure, may not only gain access to the binary but potentially also to the source code. While these software supply chain threats are critical, they fall outside the protection domain of the mechanism proposed in this dissertation. Robust industry standards and

defensive frameworks already exist to mitigate these risks [137]. While hardware protections such as storage encryption, disabling debug ports, or employing self-destruction mechanisms can mitigate many straightforward attacks, they are not always feasible or sufficient in practice. Deploying these protections may be costly, require specialized hardware, or interfere with legitimate maintenance and debugging. For these reasons, we focus on complementary hardware–software binding mechanisms, which provide protection even when conventional hardware protections are limited or impractical.

Once adversaries have obtained a binary or parts of it, they may attempt to reverse engineer the software. Furthermore, they can use the knowledge extracted from the software to clone it. As outlined in Section 1.2, we distinguish between static and dynamic techniques. Static techniques encompass reading configuration data, disassembling or decompiling code, analyzing control flow, brute-forcing decryption, extracting constants or embedded data, and finding differences between binaries. Dynamic techniques, in contrast, involve observing the software during execution through emulation, sandboxing, memory inspection, fuzzing, tracing, or monitoring I/O behavior to infer run-time behavior and internal algorithms. Encryption, obfuscation, and software diversity mitigate specific reverse engineering techniques, whereas hardware–software binding addresses reverse engineering more generally. Cloning software from observing a machine's I/O behavior is highly labor-intensive and requires substantial domain-specific knowledge. If an adversary possesses sufficient expertise to achieve this, an independent reimplementation of the software may be equally feasible.

We assume a man-at-the-end adversary engaging in industrial espionage with white-box access to the protected software. The adversary can obtain one or more genuine software instances and deploy them on cloned hardware replicating the specifications of the target device precisely. The adversary does not have access to the original source code. All approaches presented in this dissertation aim to hinder copying, static analysis, and dynamic analysis on cloned devices. By binding the software's intended execution to the physical properties of the target device, removal of the protection mechanism would render the application unusable. The attacker can run the software on the target device but cannot perform on-site dynamic analysis on the target device due to hardware-enforced access protections commonly found in industrial machinery [138]. However, we discuss some assurances our approaches can still provide in this setting in Chapter 7, mostly drawing from properties of Rowhammer.

Regarding scenario 3 (NN models), the adversary can arbitrarily query the NN model, observe its outputs, and use the resulting input–output relations to attempt model extraction. Such black-box attacks rely on retraining a surrogate model and are computationally expensive, while typically yielding only an approximation of the original model [139]. We therefore consider this attack vector out of scope and instead refer to complementary protection mechanisms [140]. Moreover, our approach does not aim to preserve the privacy of the underlying training data.

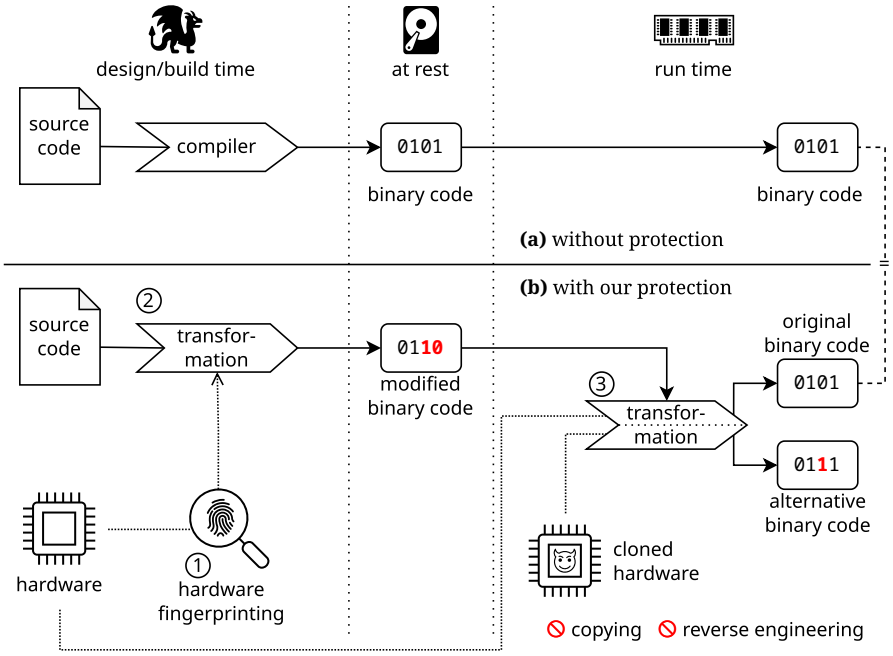


Figure 3.2: Comparing the lifecycle of **(a)** an unprotected program with **(b)** a program protected by hardware–software binding as proposed in this dissertation, hindering both copying and reverse engineering. We use (1) a hardware fingerprint to (2) transform source code pre-deployment, which is (3) reverted at run time iff the program is executed on the original hardware.

3.3 Conceptual Protection Model

We propose a conceptual model (cf. Figure 3.2) to protect software, including ML models, against unauthorized copying and reverse engineering, focusing on industrial software. Our concept integrates intrinsic hardware properties (cf. Definition 3) into the software execution logic to bind the application to its target device (cf. Definition 1). Our protection aims to ensure that attempting to reverse engineer the software offers no advantage over independent redevelopment. Successful copying or reverse engineering is acceptable only if it requires greater effort than redevelopment. Another key property of our model is that the protected software can still be executed on cloned devices, yet it behaves differently. This unintended behavior (cf. Definition 2) on cloned devices significantly limits the adversaries' capabilities to copy and reverse engineer the software.

As we can see in Figure 3.2, compared to classical software distribution **(a)**, our proposed concept **(b)** includes three additional steps, the building blocks of our concept:

(1) *hardware fingerprinting* to uniquely identify the target device,
(2) *design-/build-time transformations* to transform the program using the hardware fingerprint at design or build time to its unintended form, and
(3) *run-time transformations* to retransform the program binary at run time to its intended form iff the program is executed on the target device.

In the remainder of this chapter, we present realizations of our building blocks and their combinations.

3.4 Building Blocks

The building blocks are the core components of our conceptual model and can be realized in practice using different options. This section is structured into three parts: hardware fingerprinting, transformations, and their possible combinations. We first discuss options for hardware fingerprinting—note that specific fingerprinting methods may be incompatible with certain transformation techniques. Second, we present possible design-/build-time and run-time transformations in parallel, since the latter are the inverse of the former. Finally, we show how the building blocks can be combined to create approaches aimed at the considered scenarios.

3.4.1 Hardware Fingerprinting

In order to bind software to hardware, we first must determine identifying properties of the target device. To achieve this, we can use any of the hardware fingerprinting techniques discussed in Section 2.3. Due to our focus on industrial machines, we prefer intrinsic hardware fingerprinting over extrinsic ones. Intrinsic techniques avoid the need for additional components, which is crucial for legacy machines but potentially also benefits newly developed machines. This enables manufacturers to deploy a uniform protection mechanism across their product lines without modifying hardware designs. Hereinafter, we describe four options for hardware fingerprinting, although our concept is not limited to these.

Option 1: Rowhammer. Even if we limit ourselves to intrinsic techniques, we still have to choose among several techniques. To enable transformations that offer the strongest resistance to reverse engineering, we must choose a dynamic technique. This is because dynamic techniques allow us to identify the hardware repeatedly throughout run time, ideally just before a run-time transformation takes place. As we want to keep the run-time overhead low, we use Rowhammer-induced bit flips as fingerprinting primitive: hammering can be performed at any time during run time of a protected program and takes only a few milliseconds. Additionally, we can perform Rowhammer from any user-space process, and analyzing this process is likely to disturb the Rowhammer effect, complicating reverse engineering.

Due to Rowhammer being a hardware vulnerability, its suitability for hardware identification is not obvious. Schaller et al. demonstrated that PUFs based on Rowhammer-induced bit flips are possible [121]. While they used FPGAs with DDR2 memory, we later introduced another Rowhammer-based PUF [134] and demonstrated its feasibility on commodity PCs with DDR4 memory. Using our PUF, we can identify not only a DRAM module but also the PC in which it is installed. Instead of using a set of Rowhammer-induced bit flips as a PUF, we can also use these bit flips individually at run time, as analyzed in Section 4.1. Furthermore, we can potentially also use other primitives such as RowPress [141], where keeping a DRAM row open for a long period of time can cause bit flips in nearby rows.

Option 2: SRAM PUFs. Depending on scenario-specific constraints, alternative techniques may be required. Specifically, while their static nature limits the possible transformations, we have identified SRAM-based PUFs as the most practical technique available in scenario 2 (MCU with SRAM). SRAM PUF implementations typically require helper data used for error correction, which can easily exceed the storage available on an MCU. To address this limitation, we can identify SRAM cells with unpredictable initialization values and exclude them from the PUF response, reducing the required helper data [142, 143]; see also Section 5.1.

Option 3: Deriving keys from memory-intrinsic entropy. We developed an algorithm to derive a key from memory-intrinsic entropy [144], i.e., any bit string extracted from hardware or raw PUF response; cf. Section 5.1. For example, we can use Rowhammer-induced bit flips or SRAM's initialization patterns as input for our algorithm to reconstruct a key through a statistical model. In case the hash of the key has an unexpected value, we query the PUF again and repeat key reconstruction. As querying the PUF may not be viable—in case of SRAM's initialization patterns a reboot is required—we can use more input data to increase the probability of a key being reconstructed successfully.

Option 4: TPM-based sealing. As discussed in Section 2.3.2, TPMs can be used to *seal* data, e.g., cryptographic keys and configuration data. Sealing encrypts the data such that it can only be decrypted by the same TPM and only if the system configuration remains unchanged and the software is uncompromised. While TPMs are not generally available on all devices, the systems used in scenario 3 (NN models) have them available. Unlike the previous options, this approach does not provide an intrinsic hardware fingerprint. Instead, we seal data used for protection such that it can be unsealed only on the target device. From a practical perspective, unsealing the data is equivalent to extracting a hardware fingerprint.

3.4.2 Transformations

The transformation we apply to protect the target program must be reversible at run time iff the program is executed on the target device. Ideally, the retransformation at run time, i.e., the restoration of the intended code, happens just before the affected code is executed and is reverted directly after its execution. This complicates dynamic analyses such as debugging and memory dumps. In the following, we discuss four options for transformation that can be applied before distribution of a program, e.g., during design, implementation, compile or link time. Note that this is not an exhaustive list of options, focusing primarily on transformations suitable for the considered scenarios.

Option 1: Code and data transformations. Code and data transformations can be applied automatically, e.g., by a compiler or linker. We can transform any part of code and data, from instructions over jump targets to initial values of variables. For example, we can substitute a `je` with a `jne` instruction, inverting the branch taken by a program. This way, the unintended program still is a valid program while not executing as intended by the programmer. Rowhammer can be used to flip bits in program code loaded to memory—one bit is sufficient to transform `jne` back to `je`, as shown in Section 4.2. There are other options, such as attaching a tracer modifying the program code based on the hardware fingerprint.

Option 2: Control-state transitions. Industrial control programs can be described as *control-state algorithms*, in which execution is guided by a dedicated control state stored in memory that determines the current mode of operation. Updating this control state governs which actions are executed. Control-state transitions can be derived from hardware fingerprints, e.g., via a PUF by providing the current control state as a challenge and interpreting the response as the next control state, as detailed in Section 5.2.1. Ideally, the PUF provides a high number of challenge–response pairs to ensure that every control-flow transition is uniquely bound to the hardware, thereby maximizing resistance to reverse engineering.

Option 3: Encoding. We can encode code and data such that decoding is possible only when the correct hardware fingerprint is provided. This allows us to control the result of decoding if an unexpected hardware fingerprint is used. For example, if we choose to encode data using Boolean expressions, we can use the hardware fingerprint to evaluate these expressions. By considering alternative data values during construction of the expressions, the decoding process yields predefined values when an incorrect hardware fingerprint is supplied, thereby ensuring safe operation. For details, see Section 5.3.1.

Option 4: Encryption. By treating the hardware fingerprint as a cryptographic key, we can encrypt sensitive code and data. Before executing the code or accessing the data, we decrypt it using the previously determined hardware fingerprint. For example, we can encrypt the weights of NN models, as demonstrated in Section 6.1.1.

3.4.3 Constructing Scenario-Specific Approaches

By combining the building blocks of our conceptual protection model, we can construct specific approaches aimed at different scenarios. The subsequent chapters demonstrate this concept by detailing these approaches and their prototypical implementations. In Chapter 4, we focus on scenario 1 (PC with DRAM) by combining hardware fingerprinting via Rowhammer with code and data transformations. For scenario 2 (MCU with SRAM), Chapter 5 presents two more approaches: We demonstrate them by using an SRAM PUF, although they are more generic with respect to hardware fingerprinting and may leverage any PUF. The approaches protect programs by transforming them such that control-state transitions are bound to the hardware fingerprint or data is encoded. Finally, Chapter 6 addresses scenario 3 (NN models) by presenting an approach to bind NN models to hardware by encrypting their weights with a key extracted from hardware, e.g., from memory-intrinsic entropy or by using TPM-based sealing.

Il n'y a jamais dans la nature deux êtres qui soient parfaitement l'un comme l'autre, et où il ne soit possible de trouver une différence interne, ou fondée sur une dénomination intrinsèque.

For there are never in nature two beings which are exactly alike, and in which it is not possible to find a difference either internal or based on an intrinsic property.

— Gottfried Wilhelm Leibniz

4 Rowhammer-Based Hardware–Software Binding

Contents

4.1 Rowhammer Analysis . . . 33
4.1.1 Rowhammer Properties . . . 33
4.1.2 Memory Templater . . . 34
4.1.3 Bit Flip Reliability . . . 35
4.2 Design . . . 40
4.3 GlueZilla . . . 44
4.3.1 Junction Instruction Selection . . . 45
4.3.2 Compilation . . . 46
4.3.3 Loading . . . 47
4.3.4 Execution . . . 48
4.4 Angluerus . . . 48
4.4.1 Junction Symbol Selection . . . 49
4.4.2 Compilation . . . 49
4.4.3 Execution . . . 51

Our hardware–software binding approaches GlueZilla and Angluerus, shown in Figure 4.1, leverage the Rowhammer effect [80], enabling software to induce controlled bit flips in DRAM by repeatedly accessing—*hammering*—neighboring memory cells. These bit flips are highly dependent on physical characteristics of the DRAM chip caused by manufacturing variations, making them effectively unclonable. We transform the program's binary into an *unintended form* and use the device-specific DRAM behavior to restore its *intended behavior* at run time; cf. Definition 2. As a result, the binary becomes inherently bound to the target device's memory: only on this device, hammering flips the required bits to restore intended behavior. On other devices, the program fails to reconstruct the intended behavior and exhibits unintended behavior. Unlike conventional software protection mechanisms, which can often be bypassed through static or dynamic analysis, our approaches make reverse engineering significantly more costly and uncertain. The protection logic does not rely on explicit checks or visible anti-reverse-engineering structures but instead

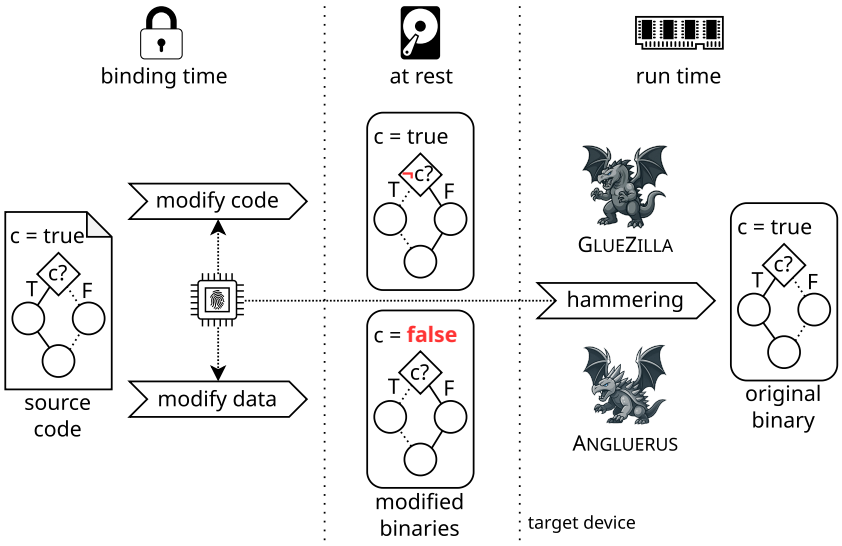


Figure 4.1: Our approaches use a memory template to compile source code into unintended binary code: GLUEZILLA modifies code and ANGLUERUS data. We restore the intended behavior of the program by hammering it on the target device exhibiting the used memory template.

ties software behavior directly to hardware properties. This design frustrates common attack vectors such as binary copying, patching, or emulation, as the intended behavior cannot be recovered without both the binary and the target device it was compiled for.

GLUEZILLA [145–147] exemplifies an approach to bind a program's behavior to hardware characteristics, focusing primarily on instruction-level transformations. GLUEZILLA uses Rowhammer-induced bit flips to dynamically switch between two execution modes: an *intended* mode that behaves correctly on the target device and an *unintended* mode that activates on all others, including hardware clones. By embedding these control-flow variations into the program's binary, GLUEZILLA ensures that only the correct hardware configuration will produce the expected functionality. This is achieved entirely in user space, without requiring hardware modifications or embedding detectable anti-reverse-engineering artifacts.

ANGLUERUS [148, 149] builds on the same foundation but introduces a data-level binding process. Instead of protecting program instructions, ANGLUERUS targets specific data symbols—such as variables used in branch conditions—whose initialization values are compiled into the binary in a deliberately incorrect—*unintended*—form. During deployment, a separate tracer process performs Rowhammer to flip the required bits in memory, restoring the *intended* values at run time. This binding again depends on DRAM behavior unique to the target device; running the same binary elsewhere fails to produce the correct bit flips, resulting in the usage of wrong data and ultimately a malfunctioning program.

Together, GLUEZILLA and ANGLUERUS demonstrate how Rowhammer-based hardware–software binding can effectively increase the cost and complexity of industrial reverse engineering by ensuring that the program exhibits its intended behavior only on the target device. Before we dive deeper into the design and implementation of GLUEZILLA and ANGLUERUS in Sections 4.2 to 4.4, we first analyze Rowhammer to better understand how we can employ it for software protection against the threats described in Section 3.2. This analysis is critical, as our protection mechanisms rely on sufficiently reliable Rowhammer-induced bit flips.

4.1 Rowhammer Analysis

During the manufacturing of DRAM chips, minor unintended deviations occur, which do not interfere with normal operation. However, these can lead to cells being susceptible to Rowhammer; typically, only a small percentage of cells are affected. Because these deviations cannot be controlled in manufacturing, they are unique to a specific DRAM chip and practically impossible to replicate. Thus, also the cells affected by Rowhammer are unique.

4.1.1 Rowhammer Properties

We lay out a number of properties of the Rowhammer effect, which we will later use as building blocks for machine-aware computations that are stealthy against reverse engineers and attackers. In the past, Rowhammer was primarily used for malicious exploits because it allows stealthy cross-process memory modifications. Apart from this property, the Rowhammer effect has many other properties, including:

1. *Unclonable and unique bit flip pattern*: Whether a DRAM cell is susceptible to Rowhammer or not depends on many uncontrollable variations during the manufacturing process of the DRAM chip. The distribution and sensitivity of Rowhammer-susceptible memory cells are, therefore, unclonable and practically unique per memory region.
2. *Bit flips in the full memory range*: Rowhammer-susceptible cells and their corresponding aggressor rows are distributed throughout the whole physical memory. This allows us to use Rowhammer to perform intra- and inter-process alterations of code and data. Inter-process bit flips, in particular, are well documented in the literature on Rowhammer attacks [81, 82, 84].
3. *No static knowledge of victim cell locations*: For many hammer patterns (cf. Figure 2.2), one can deduce the victim row based on knowledge of the location of the aggressor rows. For example, in the single-sided, double-sided, one-location, many-sided, half-double, and non-uniform hammer patterns, the victim rows are located next to the aggressor row(s) [75, 80, 81, 86, 98, 150, 151]. However, the precise locations of Rowhammer-susceptible cells are still unknown because any combination of the 65 536 bits in a victim row can be affected, assuming a DRAM row holds 8 KiB. In more complex hammer patterns, e.g., non-uniform hammering [98], deducing the precise location of the victim row(s) becomes significantly more difficult and is no longer straightforward.
4. *Bit flip success depends on execution speed*: If the aggressor accesses happen too slowly, e.g., because the CPU executes different instructions (of any process) in between, the victim cells will not lose enough charge in between row refreshes, resulting in no bit flips.
5. *Bit flip information isolated from operating system and CPU*: Bit flips in memory happen directly in DRAM without memory write instructions. They are the result of repeated memory reads to different locations in physical memory. The operating system and CPU are, therefore, unaware of any modifications in memory due to bit flips. One important consequence is that the operating system will not trigger any faults, e.g., when flipping bits in pages mapped as read-only or pages that are mapped in another process's address space.

```
rows := allocate_memory()

for row in rows[1 .. |rows| - 4] {
  // split rows into aggressors and victims
  Addr[] victims := [row, row + 2, row + 4]
  Addr[] aggrs   := [row + 1, row + 3]

  // initialize all rows byte-wise
  init(victims,    0x00)
  init(aggressors, 0xff)

  // n-sided hammering, see Listing 2.1
  hammer(aggressors, 100000)

  // scan victims for bit flips
  for victim in victims {
    if victim ≠ 0:
      store(victim, aggressors, "n-sided", "0 to 1")
  }
}
```

Listing 4.1: Simplified templating code for double-sided hammering and bit flips from 0 ⟶ 1.

We introduce the idea of using these Rowhammer properties for purposes other than malicious exploitation. Schaller et al. already explored the application of property 1 to build a Rowhammer PUF [121]. Property 2 and property 5 allow programs to make memory modifications to any process, including their own, without the operating system or the CPU noticing. These properties, furthermore, enable a new form of self-modifying code, even in non-writable code pages. Due to property 3, the static image of the binary does not contain any direct information on which modifications are performed at run time. These properties limit the amount of information one can extract from the static image about the run-time behavior.

4.1.2 Memory Templater

Our approaches require a priori information about the behavior of Rowhammer on the target device. We gather this information by measuring the Rowhammer-susceptibility of the device's physical memory using a number of hammer patterns that are known to induce bit flips [75, 80, 81, 86, 98, 150, 151]. For all discovered vulnerable cells, we collect the parameters required to reproduce the bit flip later on, including

- the physical address of the vulnerable byte and the bit index,
- the corresponding aggressor row addresses,
- the hammer pattern, and
- the flip direction, i.e., whether the bit flipped to 0 or to 1.

Listing 4.1 shows the simplified templating code for only one hammer pattern and one flip direction. We repeat this scan several times to assess the reliability of each bit flip. Our memory templater [145, 152, 153] and additional analysis tools consist of approximately 3 500 lines of C++ code.

Our templater is primarily based on Google's *rowhammer-test* [154], which was also used by Seaborn et al. [81], and on the TRRespass fuzzer [75]. The templater has extensive

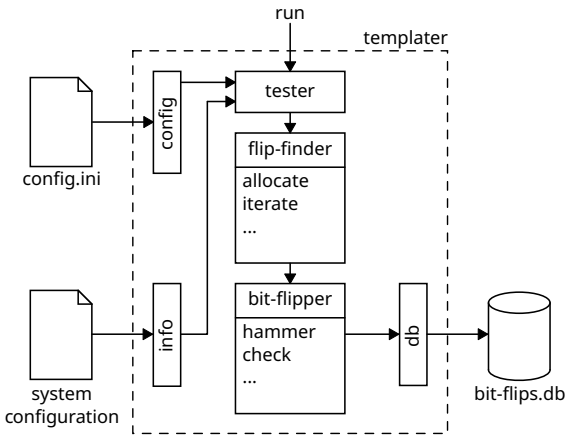


Figure 4.2: Architecture of the memory templater. The tool uses a configuration file and the system configuration as input, allocates and hammers memory, and stores the detected bit flips in a database file.

configuration options to automatically test certain memory areas several times. It is also possible to configure hammer patterns found by the TRRespass and Blacksmith fuzzers. The templater supports both DDR3 and DDR4 memory, given that appropriate mapping functions are passed to accurately target DRAM rows. These mapping functions can be determined using existing tools [74–77]. After hammering, the templater stores the detected bit flips in an SQLite database for further analysis.

Figure 4.2 shows the architecture of our templater. Its two sources of input are the system configuration, including the used DRAM modules, and a configuration file. The configuration file determines, e.g., which memory area should be allocated, which hammer pattern should be used, and how often the performed Rowhammer tests should be repeated. Hugepages [79] can be used to test large contiguous memory areas—we use 1 GiB hugepages in our experiments. In order to map virtual to physical addresses, our tool needs access to the `/proc/pid/pagemap` interface [155], which in turn requires root privileges. The output of the tool is the configuration used for the experiment and the obtained bit flips, defined by their physical addresses and specific bit position within the addressed byte.

The program itself consists of three classes. The *tester* class reads the configuration and initiates the memory allocation process. The memory allocation itself is carried out by the *flip-finder* class. In addition, this class is responsible for iterating over the allocated memory area using the configured hammer pattern. The *bit-flipper* class performs the actual Rowhammer tests, including a check for any bit flips that have occurred, and then saves these in the database file.

4.1.3 Bit Flip Reliability

Our approaches require *reliable bit flips*, i.e., those inducible with high probability via Rowhammer. As Rowhammer is fundamentally a hardware fault, the reliability of Rowhammer-induced bit flips can vary depending on memory characteristics and environmental factors. However, our experiments based on a statistical model show that under controlled conditions, a sufficient number of reliable bit flips can be achieved.

Repetitions (r)	Trials (N)	Predictive Probability	95% Credible Interval
10	10	≈ 0.917	[0.744, 0.993]
100	100	≈ 0.990	[0.970, 0.999 6]
1 000	1 000	≈ 0.999	[0.996 8, 0.999 997]
10 000	10 000	$\approx 0.999\,9$	[0.999 5, 0.999 999 7]

Table 4.1: Predictive probabilities and 95% credible intervals for different numbers of repetitions. Assumes $r = N$.

4.1.3.1 Statistical model

Our goal is to select bit flips that can reliably serve as a hardware–software binding primitive while minimizing the risk of unintended program behavior or failures when a program is executed on the target device. GLUEZILLA relies on reliable bit flips for correctness, i.e., we require very high individual bit flip reliability, e.g., $p \geq 0.99$. ANGLUERUS can tolerate lower individual flip reliability, since it can employ redundancy.

We model each bit flip at a given location as a Bernoulli process with unknown true probability p of flipping in a single hammering attempt. Repeating the hammering experiment N times, we observe r successful flips, modeled as $r \sim \text{Binomial}(N, p)$. To estimate p and its uncertainty, we use Bayesian inference with an uninformative prior $p \sim \text{Beta}(1, 1)$. This leads to a posterior distribution $p \mid r \sim \text{Beta}(r + 1, N - r + 1)$. The predictive probability of success in the next trial is

$$\Pr(\text{flip on next trial}) = \frac{r+1}{N+2}.$$

While creating a Rowhammer-based PUF for device identification [134], we observed that after approximately ten repetitions of Rowhammer tests, almost no new bit flip locations are discovered. This number of repetitions is sufficient for the PUF, as we compare fingerprints using a similarity measure—the Mahalanobis distance—that effectively filters out noise but still allows for device identification. For the GLUEZILLA approach, however, we need high reliability in every run and for each bit flip. To achieve high confidence that a bit flip is truly reliable, we require the lower bound of the 95% credible interval for p to exceed a threshold, such as 0.99 for GLUEZILLA. As shown in Table 4.1, the improvements in confidence from $N = 10$ to $N = 100$ repetitions and from $N = 100$ to $N = 1\,000$ repetitions are substantial, evidenced by the narrowing credible intervals and increasing predictive probabilities. In contrast, the gain from increasing repetitions beyond $N = 1\,000$ to $N = 10\,000$ is negligible in practical terms, providing only a minimal reduction in uncertainty despite the substantially greater effort. Therefore, $N = 1\,000$ repetitions suffice to confidently select bit flips with true reliability $p \geq 0.99$ under the assumption that observed success rates are consistent with this level of reliability.

For ANGLUERUS, individual bit flips can have lower reliability because the approach leverages redundancy: it requires flipping at least m out of n candidate bits to ensure the intended program behavior—see also the example in Section 7.1.2.1, where we use $m = 1$

and $n = 3$. The overall program success probability is, therefore, given by the cumulative probability of achieving at least m successes in n independent trials:

$$\Pr(\text{program works}) = \sum_{\substack{S \subseteq \{1,\ldots,n\} \\ |S| \geq m}} \left(\prod_{i \in S} p_i \prod_{j \notin S} (1 - p_j) \right).$$

This expression accounts for all subsets of flips where at least m succeed and correctly reflects the assumption that any m or more successful flips suffice, regardless of which ones. In the special case where all bit flip probabilities are equal, i.e., $p_i = p$ for all i, this reduces to the binomial survival function:

$$\Pr(\text{program works}) = \sum_{i=m}^{n} \binom{n}{i} p^i (1 - p)^{n-i}.$$

Thanks to this redundancy, even moderately reliable bit flips combine to yield a high overall success probability. For example, with $n = 3$ and $m = 1$ and the bit flip reliability $p = 0.9$, the success probability is $1-(1-0.9)^3 = 0.999$. Similarly, with $n = 8$ and $m = 5$, the probability is

$$\sum_{i=5}^{8} \binom{8}{i} 0.9^i (1 - 0.9)^{8-i} \approx 0.999\,7.$$

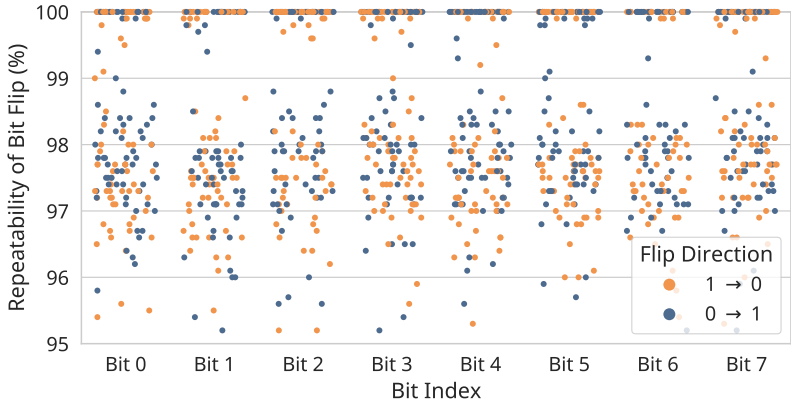
As a result, $N = 100$ repetitions are sufficient to estimate each p_i with adequate accuracy for ANGLUERUS, since the lower bound of the 95% credible interval in this case exceeds 0.9 when the observed success rate is high (e.g., $r = 100$), providing sufficient confidence for the required reliability. Note that this conclusion assumes bit flips are independent and that environmental variability does not substantially alter p_i between tests; see also the discussion in Section 4.1.3.3.

4.1.3.2 Empirical Evaluation

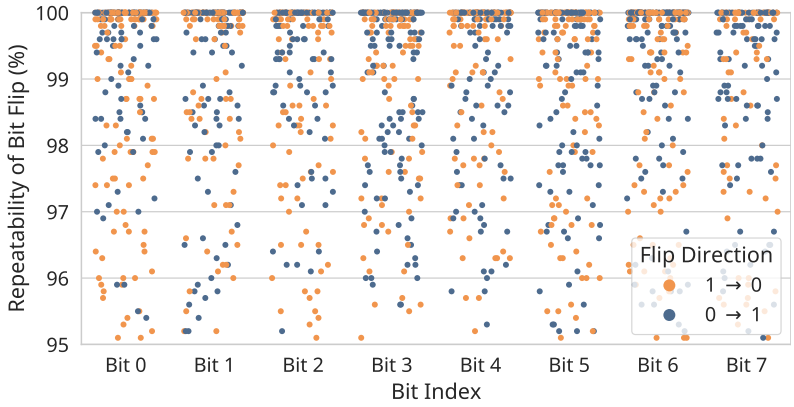
We performed multiple experiments using our memory templater on different DRAM/PC combinations and found that a share of Rowhammer bit flips are highly reliable, i.e., they flip in all tests under specific operating conditions.

For example, Figure 4.3a shows the results of two Rowhammer experiments using 54 MiB of consecutive memory of the DDR3 module S_1 from Samsung. For each experiment, we ran 1 000 Rowhammer tests using a double-sided hammer pattern, with victim and aggressor rows initialized differently to capture both flip directions, i.e., $1 \rightarrow 0$ for true-cells and $0 \rightarrow 1$ for anti-cells. While some flips occurred sporadically, which we consider to be noise, several bits flipped consistently in all iterations, making them reliable bit flips suitable for our approaches. We observed reliable bit flips across all bit indices (bits 0–7 of a byte) in both flip directions.

We conducted the same experiment testing 800 KiB of the DDR4 module H_1 from SK Hynix using a non-uniform hammer pattern generated by Blacksmith [98]. Despite the smaller memory region, DDR4's higher susceptibility to bit flips yielded a comparable result. Figure 4.3b shows that DDR4 also provides a sufficient number of reliable bit flips. Thus, we conclude that both DRAM types can offer reliable bit flips suitable for our approaches.



(a) Double-sided pattern on DDR3 module S_1.



(b) Non-uniform pattern on DDR4 module H_1.

Figure 4.3: Bit flip reliability with 1 000 iterations per flip direction. Each dot represents a bit flip at a specific location in DRAM. They are grouped by their bit index within the respective directly addressable byte.

We also evaluated the distribution of bit flips to rule out repeating and thus predictable patterns; see also [134, 152]. Figure 4.4 shows an excerpt of the bit flips of the DDR4 module S_2, specifically the first 256 bits of 128 contiguous rows in one of its banks. Marking whether the bit flips happen in true- or anti-cells does not indicate a clear pattern, comparable to some memory areas of the DDR3 memory modules examined by Bepary et al. [156]. If one looks at the distribution of true- and anti-cells on all 65 536 bits per row and additional 1 152 rows we tested, there still is no easily recognizable pattern in the distribution. The number of bit flips per row varies but the number of true- and anti-cells is approximately equal per row.

Figure 4.5 shows the bit flips of a 5 MiB contiguous memory area of a bank of module H_1 and illustrates how often individual bit flips occur in 1 000 Rowhammer tests. In order to display the entire spectrum, we divide the data into 20 bins of equal size. Many bit flips fall into the first bin, i.e., they rarely occur, in at most 50 out of our 1 000 tests. There is also a considerable number of bit flips in the last bin, which occurred in 950–1 000 of our tests. Out of these, a substantial percentage occurred in all 1 000 Rowhammer tests and can therefore be considered reliable. Concretely, 4 611 bit flips occurred in all tests, corresponding to 3.4% of all observed bit flips.

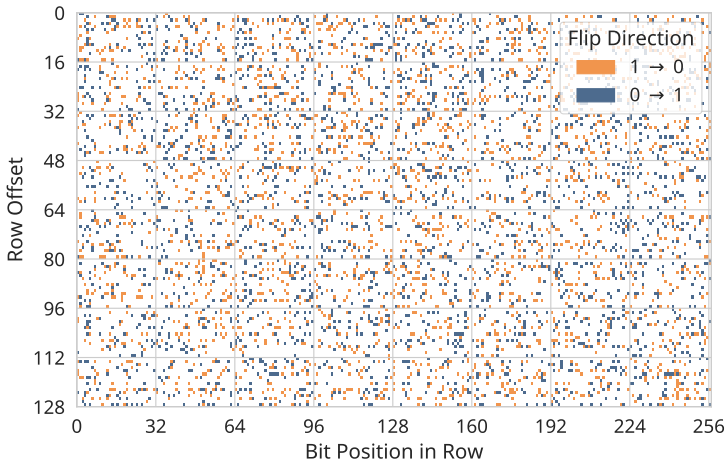


Figure 4.4: Distribution of bit flips on S_2 showing the first 256 cells of 128 contiguous rows. Each dot represents a bit flip. In order to generate as many bit flips as possible, the refresh interval was configured to the highest possible value.

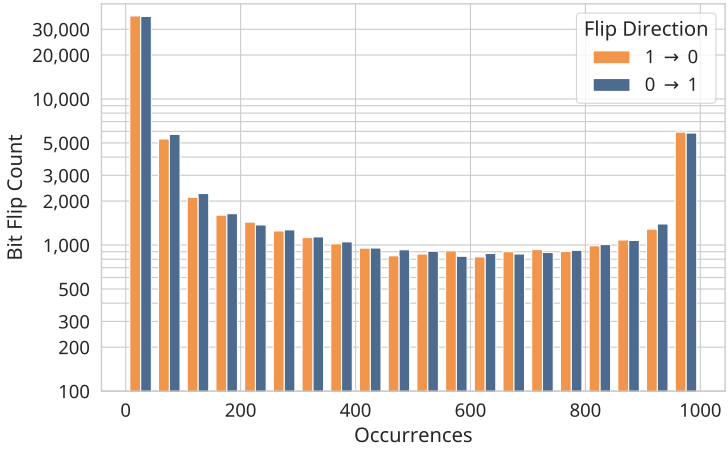


Figure 4.5: Reliability of bit flips out of 1 000 Rowhammer tests per flip direction. We tested a 5 MiB large contiguous memory area on a bank of H_1. Note that the y-axis is scaled logarithmically.

4.1.3.3 Discussion

Our data confirms the existence of reliable Rowhammer-induced bit flips. However, we cannot guarantee 100% reliability because the Rowhammer effect is still a hardware disturbance error and numerous factors might undermine the reliability, e.g., ambient temperature, hardware aging, and interference with other running software. The intended execution in GLUEZILLA and ANGLUERUS might not be fully reconstructed if a bit does not flip as expected.

To maximize reliability, we assume that software and hardware vendors do not include any constructs that effectively counteract our Rowhammer-based program transitions. These include advanced error correction codes for memory transactions, which could undo the bit flips, or anti-tamper protections, which could detect changes in the executing

program code. We deem this assumption reasonable, as existing countermeasures on commodity systems are insufficient to fully prevent Rowhammer-induced bit flips, particularly when advanced hammering techniques are used; cf. Section 2.2.4.

As many industrial systems are considered critical, we can use different measures to increase reliability. Rather than just one bit flip, we can combine multiple distinct bit flips to enable each program to transition to its intended form. Whether the transition is performed at run time therefore depends on the success of one or a majority of the required bit flips.

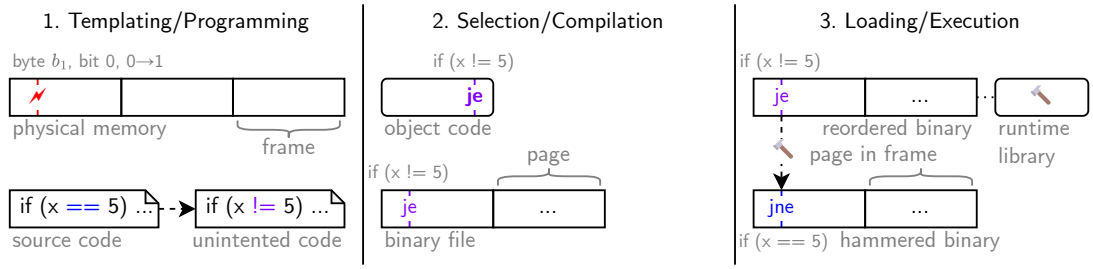
When flipping bits in data, integrity checks can detect data corruption, allowing us to take corrective measures. If discrepancies are detected, we can attempt to recover the original data using redundancy. In cases where the majority of bits flip as intended and only a small portion of the data remains in its unintended form, we can use error correction codes—such as Hamming or Reed-Solomon codes—to automatically reconstruct the intended data. Using error correction does not weaken the protection, as it only compensates for residual noise after the correct hardware-dependent flip pattern has been induced and does not reduce the attacker's uncertainty without those flips. Additionally, the data layout can be optimized to place critical information in memory regions that are more susceptible to reliable bit flips. For example, we can use several unreliable bit flips in image or sensor data. Using masks, thresholds, or checksums, we can detect and filter out potentially corrupted data caused by unreliable bit flips.

In the absence of suitable error correction mechanisms, our method remains applicable to non-critical applications where occasional degraded executions are acceptable. For example, we can keep the intended and unintended executions functionally equivalent while making the unintended form less efficient by using suboptimal algorithms, increased resource consumption, or additional user interaction. In this case, the program always preserves its critical functionality but executes efficiently only on the target machine. For systems requiring absolute reliability with no tolerance for degraded execution, a different, potentially weaker, protection mechanism may be more appropriate.

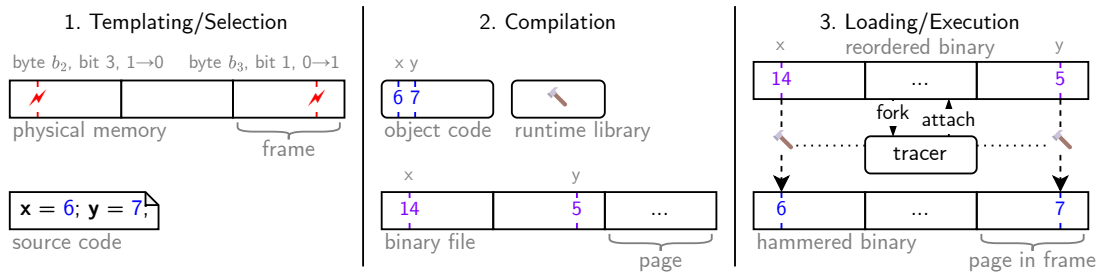
Environmental influences, such as temperature in particular, can have a significant impact on bit flip reliability [93, 157]. To take this into account, we can follow a similar approach to the one employed by Penz et al. [143]. For this purpose, we propose evaluating the bit flips in several tests throughout a defined temperature range and considering only the bit flips that occur across all tests over the defined temperature range for our approach. This also means that certain bit flips could be linked to a specific temperature range [157]. While aging might also influence the bit flips, we consider it to be unlikely that bit flips we rely on suddenly do not flip anymore as the number of bit flips typically increases in older DRAM [158].

4.2 Design

In this section, we dive deeper into the design of GLUEZILLA and ANGLUERUS. Both approaches leverage the Rowhammer effect to bind software to hardware, ensuring that programs exhibit different behaviors depending on the machine they execute on. Protected programs exhibit their intended behavior only on the target device, i.e., the device whose memory template matches the one we used during building the program. On any other device, they



(a) GLUEZILLA requires a memory template, the source code to be changed to its unintended form, and the selection of junction instructions after compilation. The compiler moves the instructions to pages at certain offsets so that the loader can align them with bit flips at run time. The protected process hammers its own memory to flip its instructions to their intended forms.



(b) ANGLUERUS also requires a memory template and the selection of data symbols. The linker moves the symbol values to pages at certain offsets so that the runtime library can align them with bit flips at run time and hammer these values to their unintended form. The protected process forks itself and attaches the forked process as a tracer, which hammers the protected process so that the symbols assume their intended values.

Figure 4.6: Comparison of the design principles underlying GLUEZILLA and ANGLUERUS.

remain in the unintended execution mode. These two execution modes only differ at *junction points*—program locations where the execution depends on a Rowhammer-induced bit flip. For example, a branch condition in the intended execution could be inverted in the unintended execution, or a function call could be redirected to another function in the unintended execution. If chosen specifically to not arouse suspicion, it becomes difficult for an attacker to distinguish unintended from intended behavior. Reliable distinction of both behaviors requires expert understanding of the program's intended behavior. Adversaries typically lack this knowledge since gaining this knowledge is the primary reason for reverse engineering the program in the first place.

Our approaches leverage several of the Rowhammer properties we laid out in Section 4.1.1 to implement the run-time switch from the unintended to intended execution mode. Property 2 allows us to use Rowhammer-induced bit flips to transform the unintended into the intended code. For the transition to intended code to be successful, the protected program requires specific bits to flip. Due to property 1, the desired bit flip pattern is unclonable and only present on the device for which we compiled the software instance. This ensures the intended execution mode is only reconstructed on the target device. The static binary image, furthermore, does not contain any information about the bits that must be flipped to obtain the intended code, as explained by property 3. Finally, property 5 states that the CPU does not explicitly perform the required memory modifications. There is, thus, no need for any explicit branching instruction inside the protected program that conditionally reconstructs the intended code. Instead, reconstruction is triggered by a

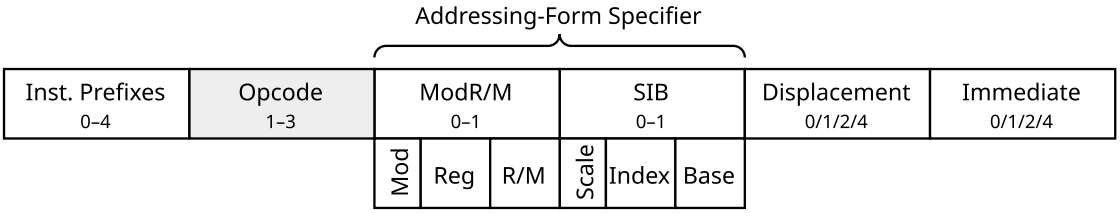


Figure 4.7: Fields of x86 instructions, their byte size ranges, and the subfields of the ModR/M and SIB bytes.

Assembly Code	**Machine Code (Hexadecimal)**
Instruction Prefixes	
`add {eax,ebx ↔ rax,rbx}`	`40 01 d8 ↔ 48 01 d8`
Opcode	
`{and ↔ xor} cl,al`	`20 c1 ↔ 30 c1`
`{je ↔ jne} +0x05`	`74 05 ↔ 75 05`
`{add ↔ adc} eax,ebx`	`01 d8 ↔ 11 d8`
`{sub ↔ sbb} eax,ebx`	`29 d8 ↔ 19 d8`
`{inc ↔ dec} al`	`fe c0 ↔ fe c8`
`{movsx ↔ movzx} eax,byte [rbx]`	`0f be 03 ↔ 0f b6 03`
`add {al,al ↔ eax,eax}`	`00 C0 ↔ 01 C0`
`add eax, {0x7fc0837f ↔ 0x7f; add eax,0x7f}`	`81 c0 7f 83 c0 7f ↔ 83 c0 7f 83 c0 7f`
ModR/M	
`mov al,{bl ↔ cl}`	`8a c3 ↔ 8a c1`
`mov al,[{rax ↔ rcx}]`	`8a 00 ↔ 8a 01`
SIB	
`mov al,[rax+rbx*{1 ↔ 2}]`	`8a 04 18 ↔ 8a 04 58`
`mov al,[rax+{rbx ↔ rcx}*1]`	`8a 04 18 ↔ 8a 04 08`
`mov al,[{rax ↔ rcx}+rbx*1]`	`8a 04 18 ↔ 8a 04 19`
Displacement	
`mov al,[rax+{0x10 ↔ 0x11}]`	`8a 40 10 ↔ 8a 40 11`
Immediate	
`add rax,byte {0x07 ↔ 0x47}`	`48 83 c0 07 ↔ 48 83 c0 47`
`call +{0x1000 ↔ 0x1100}`	`e8 00 10 00 00 ↔ e8 00 11 00 00`

Table 4.2: Examples of x86 instruction pairs differing by a single bit, grouped by encoding field.

Rowhammer procedure that induces device-specific bit flips at run time. As explained by property 5, these memory modifications also work in non-writable code pages.

Both GLUEZILLA and ANGLUERUS transform the program into the unintended form before deployment and use Rowhammer at run time to revert it back to the intended form. However, they differ significantly in how they apply this principle to the program to be protected, each offering unique advantages. GLUEZILLA performs these transformations on *program instructions* while ANGLUERUS transforms *program data*. Thus, we distinguish two types of junction points, *junction instructions* and *junction symbols*.

```
1 volatile int x = 6; // configured to flip bit 3, 14 in binary
2 volatile int y = 7; // configured to flip bit 1,  5 in binary
3
4 int main() {
5   if (x > 6)        // true only if on cloned device
6     return x + y;   // on cloned device: 14 + 5 = 19
7   else
8     return x * y;   // on target device:  6 * 7 = 42
9 }
```

Listing 4.2: A simple C program to demonstrate our approach. ANGLUERUS is used to protect the data symbols x and y. This example is also visualized in Figure 4.6b.

Figure 4.6a illustrates the basic steps of GLUEZILLA. First, we create a *memory template* in order to identify reliable Rowhammer-induced bit flips. The programmer creates the unintended form of the program to be protected by inverting the operator == to !=. The respective x86 opcodes for these operators differ in only one bit. Next, we compile the unintended source code with the GLUEZILLA toolchain and select the machine instructions that should differ between the unintended and intended program form, i.e., `je` in our example. During the second compilation, the GLUEZILLA compiler uses both the memory template and the instruction selection to build the protected binary. In the final step, when loading the binary into memory, the GLUEZILLA loader reorders the pages of the program to align the selected instructions with the Rowhammer-susceptible cells. It then induces the bit flips, recreating the intended form of the program if executed on the target device. That is, bit 0 in `je` (binary opcode `0111 0100`) is flipped from 0 to 1, resulting in `jne` (`0111 0101`).

In addition to the mandatory opcode, which selects the operation to be performed, x86 instructions may contain several optional fields [159], as illustrated in Figure 4.7. Among these optional fields, instruction prefixes modify execution semantics, and immediate data provides constant operands. The addressing-form specifier and the displacement encode memory addressing and register selection. Table 4.2 lists examples of symmetric instruction pairs in which GLUEZILLA can induce bit flips.

We illustrate ANGLUERUS's approach with the example in Listing 4.2 and Figure 4.6b. Just like in GLUEZILLA, the process begins with *memory templating* to identify Rowhammer-susceptible cells in the DRAM of the target device. Next, the programmer specifies which bits of a data symbol value in the target program should differ between the unintended and intended program form. We use symbols `x` and `y` in our example. The program is then compiled and linked using the ANGLUERUS toolchain. The toolchain transforms the selected symbols to their unintended values, i.e., `x = 14`, `y = 5`, and aligns them with the previously identified Rowhammer-susceptible memory cells. This program form would incorrectly evaluate the conditional on line 5 positively, hence executing the unintended operation on line 6 on unintended operands. At run time, ANGLUERUS performs Rowhammer from a separate process to induce the necessary bit flips, i.e., bit 3 in the value of `x` and bit 1 in the value of `y`. This restores the intended values of the symbols, i.e., `x = 6`, `y = 7` and the program now executes the intended operation on line 8. For more advanced examples, see Section 7.1.2.1.

For both approaches, the required bit flips occur only on the target device, and running the binary on any other device results in execution of the unintended form. GLUEZILLA directly protects executable code by inducing bit flips in instructions, which makes it suitable when protecting the behavior of a program is the main objective. However, GLUEZILLA relies on individual bit flips in junction instructions to transform them, restricting the possible positions of junction bits to those that result in valid instructions when flipped (cf. Section 7.1.1.1) and limiting its tolerance to unreliable bit flips. In contrast, ANGLUERUS targets data values and thus can tolerate unreliable bit flips, e.g., by employing redundancy and validation at run time. Since GLUEZILLA and ANGLUERUS target different sections of binaries and exhibit complementary strengths, combining them can yield a stronger overall protection mechanism.

4.3 GLUEZILLA

In this section, we describe the implementation of GLUEZILLA, demonstrating our approach to binding software to hardware using Rowhammer-based instruction transformations. At a high level, GLUEZILLA performs the following phases to protect a program; see also Figure 4.8:

1. *Preparation Phase*:
   (a) *Memory Templating*: First, we discover Rowhammer-susceptible cells in a physical memory region on the target device and store all parameters to reproduce the bit flips later. Next, we analyze the reliability of the bit flips and select the most reliable ones.
   (b) *Junction Instruction Selection*: A junction instruction has two forms that differ in only one bit, the *junction bit*: (i) the original, intended instruction, and (ii) its unintended instruction counterpart. Suitable junction instructions and their junction bits are either selected automatically by the compiler through random selection or manually by the developer.
2. *Compilation Phase*: The compiler matches each junction instruction—or, more precisely, each junction bit—with a suitable Rowhammer-susceptible cell from the memory template and thus with a physical memory frame. It then lays out the code accordingly and emits the junction instructions in their unintended form in the final binary.
3. *Loading Phase*: While loading the program, we massage the memory from user space, i.e., we force the memory allocator via exhaustive allocations to return the required physical frames. Then, we load all code pages into the expected frames.
4. *Execution Phase*: Finally, we hammer the junction bits in memory. On the target device, we transform the junction instructions from their unintended into their intended form; cf. Figure 4.9. When running on any other device, the junction instructions remain in their unintended form.

With memory templating established in Section 4.1.2, the subsequent execution phases are detailed below.

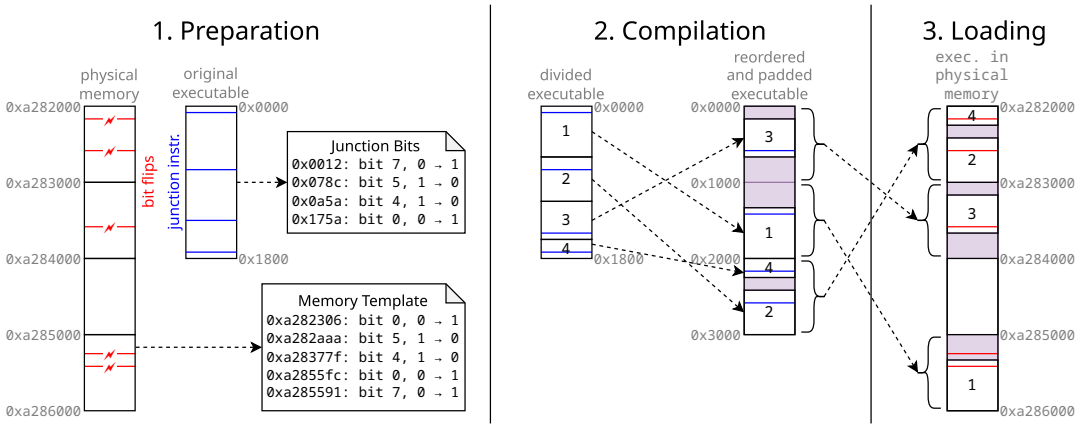


Figure 4.8: GLUEZILLA generates a memory template on the target device and compiles the protected program such that all developer-specified junction instructions can be loaded into Rowhammer-susceptible DRAM cells at run time.

4.3.1 Junction Instruction Selection

Junction instructions define the points where the intended and unintended execution of the program differ. The effort to reverse engineer the protected program increases with the difference between both execution forms and, therefore, with the number of junction instructions; cf. Section 7.1.1.1. The differences should not be obvious to reverse engineers, e.g., the unintended program should not crash. Furthermore, we want to guarantee that the program is not actively harmful, even when it executes its unintended code. The aforementioned reasons make junction instruction selection challenging to automate because it requires reasoning about the program's behavior.

An additional challenge lies in the sparse distribution of Rowhammer-susceptible cells across DRAM. In practice, this distribution limits us to a single bit flip per junction instruction—we refer to this bit as the *junction bit*. Fortunately, if chosen well, flipping one bit in a junction instruction often suffices to make the intended and unintended executions of a program differ substantially. This is especially true on architectures with dense instruction sets, e.g., x86. On such architectures, there are many valid machine instructions that differ in only one bit from several distinct instructions with different semantics; cf. Table 4.2. The resulting valid instruction after the bit flip is even allowed to differ in encoding length as long as the total sequence is still valid. By carefully selecting the junction instruction, we can create a different control flow—e.g., by changing loop predicates or jump/call destinations—and data flow—e.g., by changing arguments or constants—between the intended and unintended executions.

We classify junction instructions with respect to their position-related constraints into the following two classes:

1. *Unmovable junction instructions* only require that their junction bits are aligned with suitable Rowhammer-susceptible DRAM cells. Figure 4.9 illustrates an unintended `je` instruction, which becomes an intended `jne` instruction after the bit flip ②. This instruction must be placed in memory so that we can flip bit 0 of the instruction from 0 to 1.

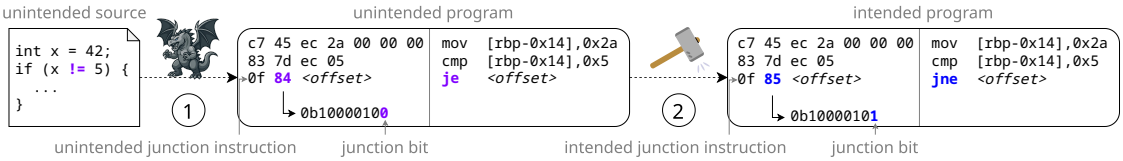


Figure 4.9: At run time, GLUEZILLA uses Rowhammer to transform junction instructions from their unintended into their intended form.

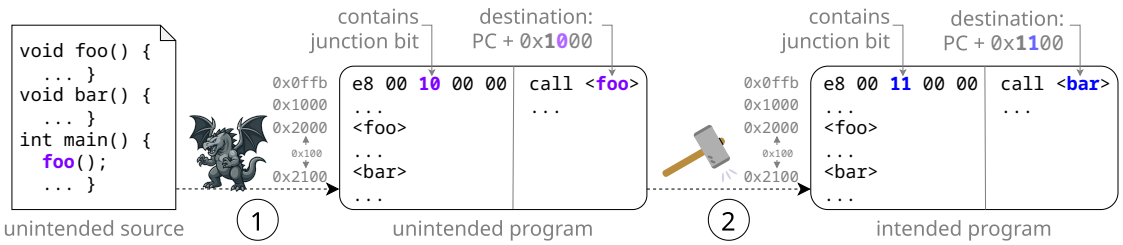


Figure 4.10: Junction bits inside control transfer destinations add constraints to the relative position of the destinations.

2. *Movable junction instructions* create inter-instruction placement constraints. Figure 4.10 illustrates an unintended call to `foo`, which becomes an intended call to `bar` after the bit flip ②. If we choose bit 8 of the second byte of the branch target operand as our junction bit, the target changes from `0x1000` to `0x1100` during the run-time transformation from the unintended to the intended execution mode. This puts a relative placement constraint on the two call targets (`foo` and `bar`) as they must be exactly `0x100` bytes apart.

To select junction instructions, we implemented both an automatic and a manual operation mode. The automatic operation mode selects a specified number of unmovable and movable junction instructions randomly and uniformly across the binary. However, this mode provides no guarantees regarding software behavior on cloned machines, as the applied changes can significantly alter execution semantics. In the worst case, this may cause harm to the machine or its environment. Thus, we also support a manual operation mode to account for safety-critical scenarios.

In manual operation mode, we rely on a developer to modify their source code to unintended form and keep track of the changed instructions, i.e., the junction instructions. To facilitate this task, the developer can construct the unintended behavior by adding subtle alterations to the existing control and data flow graph of the intended behavior. To increase the number of available alterations, the developer can perform semantics-preserving transformations, such as data structure or function parameter permutation.

4.3.2 Compilation

The transformation from intended to unintended execution mode only works if we can position the program's code in physical memory such that every junction bit is stored in a Rowhammer-susceptible DRAM cell. This poses the following challenges:

1. *Bit Alignment*: Machine instructions are aligned to byte boundaries in physical memory. Thus, if the n-th bit within a junction instruction is a junction bit, then it is only possible to store this bit in the m-th cell of a DRAM row if n and m are congruent modulo 8, i.e., $n \equiv m \pmod 8$.
2. *Page Alignment*: For practical reasons, and although not strictly required, code pages within a program binary should be aligned to page boundaries when loaded into virtual and physical memory. This means that if a junction instruction is at page offset n within a code page of a program binary, said instruction can only be stored at byte offset m in a page frame if n and m are congruent modulo the page size.

We developed a compiler toolchain and a binary loader to tackle these challenges. To overcome the first challenge, our compiler consults the target device's memory template, created using the memory templater introduced in Section 4.1.2, to find candidate cells for each junction bit that complies with the required bit alignment. We use this list of candidates to constrain our *layouting passes*. These layouting passes assign each code page to a physical page frame with a suitable candidate DRAM cell, where that page's contents must be loaded. We emit information about these assignments alongside the program binary in the form of a *loader map*. Afterward, the layouting passes apply dead nop insertion and instruction/basic block reordering [52] to move each junction instruction to the page offset of the candidate cell, which solves the page alignment problem. In the end, the linker links the runtime library containing the massaging, loading, and hammer code. We implemented all logic related to the custom binary layout in the machine code sub-project of LLVM 11 in almost 1 200 lines of C++ code.

4.3.3 Loading

GlueZilla only works if it can guarantee that every junction bit is stored in its corresponding physical memory cell. However, user-space programs cannot directly control the physical memory mapping performed by the operating system. We tackle this challenge using *memory massaging* from user space, similar to prior Rowhammer attacks [81, 84–86, 89, 151]. In detail, we massage the operating system's page frame allocator so we can load every code page that appears in our loader map into its corresponding physical frame.

The massaging techniques presented in Section 2.2.1 are built for attack scenarios where the attacker controls the massaging process and tries to load pages of the victim process in a specific physical page frame [81, 84–86, 89, 151]. These techniques typically leave a time window between the freeing of Rowhammer-susceptible page frames by the massaging process and the reallocation of said frames by the victim program. In this time window, the operating system may assign the desired frame to other programs running in the background. Most attacks tolerate this probabilistic behavior by relying on the victim program requiring only a small number of Rowhammer-susceptible physical page frames and can simply be relaunched if an allocation fails.

In contrast, programs protected by GlueZilla might require many different page frames in both victim and aggressor rows and therefore probabilistic massaging is inadequate. To address this challenge, we design a *deterministic* massaging technique by performing the massaging step from within the same process whose page frames we want to hammer. This closes the time window because we do not have to deallocate frames only to trigger

their reallocation in a different process. Our technique allocates pages until we find all required physical page frames in the loader map. We implemented our binary loader and supporting tools in approximately 1 200 lines of Rust code.

4.3.4 Execution

After loading all code pages into their corresponding page frames, we start hammering to transition into the intended execution mode. We flip the junction bits by rapidly hammering each junction bit's corresponding aggressor rows with the hammer pattern determined during templating.

In most cases, this operation works only in one direction: it can transform a junction instruction from its unintended form into its intended form but not vice versa. Thus, after executing a junction instruction once, it would remain in its intended form until the program stops. However, it is also possible to revert the transformation after executing the junction instruction. For example, we could reset the contents of page frames containing junction instructions by either explicitly reloading the code from disk or by forcing the operating system's pager to evict the frame and, subsequently, page it back in. Thus, we restore the junction instructions' unintended forms, either periodically or immediately after executing them.

Other degrees of freedom lie in the timing of the transformation and the placement of the hammering code. Rather than applying the transformation at program startup, we can also perform it immediately before the execution of each junction instruction or at any point in between. Similarly, we can invoke the transformation from within the GLUEZILLA-protected process or from an external process. By separating the transformation code in time and space from the junction instruction it transforms, we can increase stealthiness and resilience against reverse engineering. However, too much separation, especially in time, can facilitate memory snapshotting attacks, as we discuss in Section 7.1.3.

While hammering, the protected program uses approximately all available memory bandwidth. To enable a successful transformation, no other program running on the system should consume a significant portion of the memory bandwidth while the protected program is hammering. Multiple GLUEZILLA protected programs can run on the same system simultaneously as long as they adhere to this constraint, i.e., they must synchronize their Rowhammer-based transformations to avoid memory-bandwidth conflicts. This synchronization does not convey secret information, as the resulting timing is already observable by a determined attacker due to Rowhammer-induced memory-bandwidth saturation.

4.4 ANGLUERUS

This section shows the implementation of ANGLUERUS, showcasing our hardware–software binding approach using Rowhammer-based data modifications. Our approach binds the initialization values of a program's data symbols, e.g., constants and variables, to the target device so that the program runs as intended only on this device. The phases are similar to those described for GLUEZILLA. However, instead of junction instructions, in ANGLUERUS we select *junction symbols*. Each junction symbol has two value forms: (i) the original, intended symbol value, and (ii) its unintended counterpart. Like GLUEZILLA's

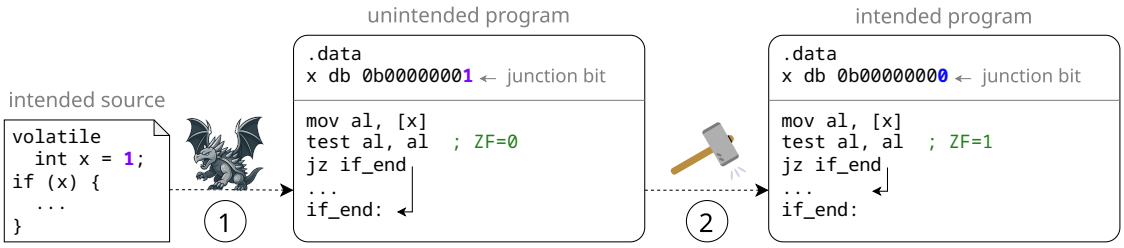


Figure 4.11: At run time, ANGLUERUS uses Rowhammer to transform symbol values from their unintended into their intended form.

junction instructions, the junction symbol's values are identical except in the junction bit. Furthermore, ANGLUERUS attaches a separate process to the target process as *tracer*. This tracer allocates the aggressor rows and hammers them on request of the target process.

Our prototype currently only supports programs implemented in C compiled for 64-bit x86 systems with DDR4 memory running Linux. We implemented ANGLUERUS in approximately 1 400 lines of C++ code based on LLVM 18 and reuse the memory templater described in Section 4.1.2.

4.4.1 Junction Symbol Selection

One must specify the symbol name—or more precisely the corresponding source-level identifier—and the index of the bit to be flipped within its value. Optionally, one of the following selection modes may be provided:

- *intended* (default): The symbol value in the source code is in its intended form. The compiler will later change the symbol value to its unintended form.
- *unintended*: The symbol value in the source code represents its unintended form. This mode is useful if it is more intuitive to program the unintended behavior and resembles GLUEZILLA's approach.
- *dummy*: Places the bit to flip in a memory cell so that the cell is discharged before hammering and thus cannot flip, i.e., places 0 in a true-cell and 1 in an anti-cell. This mode can be used to confuse an attacker.
- *debug*: Sets a bit to 0 or 1 at run time using `ptrace` instead of hammering.

ANGLUERUS works on global variables in the `.data` and `.bss` sections and constants in the `.rodata` section of the binary. In addition, they must be declared with the `volatile` type qualifier to ensure that the compiler does not remove seemingly redundant load operations for the same symbol value. This is required to make sure the value is reloaded from memory after the junction bits in the value were flipped.

4.4.2 Compilation

The ANGLUERUS compiler is based on *Clang*, the LLVM frontend for C. We added a function pass that inserts calls to our runtime library to invoke hammering. Our prototype is currently limited to flipping bits at startup of a process. Ideally, this pass would invoke the runtime before each access to the protected data so that the data could be hammered just before it is accessed. For details on how we could achieve this, see Section 4.4.3.

During linking (see Excursus 1), the compiler ensures that all symbol values are in their unintended form. This means that all symbols selected with the *intended* selection mode are changed to their unintended form ①; cf. Figure 4.11. We also ensure that we (i) create dedicated sections for every object symbol representing data objects, and (ii) zero-initialize all uninitialized global variables and put them in the `.data` section instead of the `.bss` section.

EXCURSUS 1: LINKING

Compilers and assemblers translate source code into machine code and store the machine code in a separate object file per translation unit. A linker then combines these object files into, e.g., a binary in the executable and linkable format (ELF) used on Linux [160]. ELF files consist of various sections, e.g., the `.text` section containing the actual machine code, the `.data` section holding initialized variables, and the `.rodata` section storing constants. Symbol names identify objects—such as functions and variables—and thus must be unique within the object files to be linked. While these names correspond to source-level identifiers, compilers often employ name mangling to encode scope and type information into a unique, linkable string. The linker assigns addresses to the symbols and ensures every reference to the symbol within the machine code uses the appropriate address.

Next, we match all junction bits in the junction symbols with a Rowhammer-susceptible cell from the memory template. In this step, ANGLUERUS encounters the same challenges faced by GLUEZILLA, as discussed in Section 4.3.2. We solve the challenge of bit alignment the same way as in GLUEZILLA by matching each junction bit with a Rowhammer-susceptible cell that meets the required bit alignment. Similarly to GLUEZILLA, we put each symbol value on a separate page and pad it to match the junction bit's page offset with the frame offset of the selected Rowhammer-susceptible cell, solving the page alignment. Each page containing a junction symbol value is now associated with one physical frame. We store this data in the *loader map*, just like in GLUEZILLA. In ANGLUERUS, however, we store this map within an additional metadata section within the final binary. We also add all other required metadata, e.g., the hammer frequency and the aggressor rows along with their initial values, in extra metadata sections. In the end, the compiler includes the static runtime library containing the massaging and loading code.

For an illustrative example, consider the variable `x` from Listing 4.2, which has an intended value of 6. Flipping its third bit causes its unintended value to be 14. First, we read the configuration files specifying that bit 3 in byte 0 of the integer symbol `x` should be flipped in the default *intended* selection mode. Second, the compiler changes the value of `x` from 6 (**0**110 in binary) to 14 (**1**110). Third, the compiler chooses a victim cell on the target machine, i.e., a cell flipping from 1 to 0 so we can flip bit 3 back to 0. The victim cell must also represent bit 3 within a byte due to byte alignment, and its address must be a multiple of 4 because of alignment of 4-byte integers. The compiler puts the object symbols corresponding to `x` on a separate page at an offset so that it aligns with the victim cell within its frame. Finally, the compiler adds three synthetic sections to the binary: (i) a section containing the parameters for hammering, (ii) a mapping of victim pages of the binary to victim frames on the target device, and (iii) the aggressor rows and their

initial values. While this information is visible to an attacker, it does not disclose the position of the bit to be flipped within the victim page.

As presented in Section 4.4.1, ANGLUERUS supports the *debug* selection mode to simulate bit flips for testing purposes. Instead of hammering bits at run time, we use `ptrace` to flip bits. To support this, the compiler inserts another metadata section containing the offset and flip direction of the bit flip, along with the probability of the bit flip occurring.

4.4.3 Execution

The ANGLUERUS runtime library is a static library implemented in approximately 400 lines of C++ code. The entry point of the runtime is called at the beginning of the `main` function. The runtime (i) forks the process into a *tracee* executing the program logic and a *tracer* performing Rowhammer on the tracee and (ii) attaches the tracer to the tracee; see Excursus 2. Next, the runtime massages the memory (cf. Section 4.3.3) to allocate the frames defined in the synthetic section of the binary—frames of the victim rows in the tracee and frames of the aggressor rows in the tracer. To check the frame a page is loaded into, we access the `pagemap`, which requires root privileges—we drop these permissions once the allocation has finished.

EXCURSUS 2: TRACING

On Linux, `ptrace` [40] is a system call that allows the implementation of debuggers. Specifically, a *tracer* can attach to another process, the *tracee*, and, e.g., access its memory. Linux allows at most one tracer to be attached to a tracee at any time. Thus, attaching a tracer by default can be used to hinder reverse engineering by debugging. This method is also referred to as *debug blocking* [39].

The tracee copies the pages containing the junction symbol values to the victim frames and remaps these frames to the virtual addresses the binary was originally loaded to. The aggressor frames are initialized according to the configuration. The tracer then performs Rowhammer by accessing its aggressor frames repeatedly, causing bit flips in the victim frames of the tracee containing the symbol values ②; see Figure 4.11.

In order to hammer the symbol values just before they are accessed, we could use breakpoint instructions in the tracee. Upon reaching such a breakpoint instruction, the tracer would be invoked, could identify the symbol, and hammer its corresponding aggressor rows. To mitigate the run-time overhead of just-in-time hammering, breakpoints can be positioned further upstream in the execution flow or synchronized with the program's idle time, such as when it is awaiting input.

上醫醫未病，中醫醫欲病，下醫醫已病。

The superior doctor prevents sickness;
the mediocre doctor attends to impending sickness;
the inferior doctor treats actual sickness.

— Attributed to Sun Simiao (孫思邈)

5 PUF-Based Hardware–Software Binding

Contents

5.1 SRAM-Based Hardware Fingerprinting 54
5.2 MINOTAUROX 55
5.2.1 Design 56
5.2.2 Implementation 62
5.3 SPHINXOR 62
5.3.1 Design 64
5.3.2 Implementation 67

In this chapter, we focus on fail-safe approaches for hardware–software binding building onto PUFs. We bind a program to be protected to a target device so that the program executes as intended only on that device. On any other device, the program behaves differently but still meets the required safety constraints for industrial applications. This also applies in case the PUF delivers an unexpected response, e.g., due to tampering [161]. In other words, even when showing unintended behavior, the program must still behave safely.

Similar to our Rowhammer-based approaches in Chapter 4, we present two approaches, one focusing on code, the other on data. First, MINOTAUROX makes control-state transitions depend on a PUF responses and thus protects the control flow of control-state algorithms. In case of an incorrect PUF response, we choose the next control state randomly but ensure the chosen transition does not lead to a violation of the safety constraints. Second, SPHINXOR protects sensitive data by encoding it together with alternative data that can affect the performance of the system in a detrimental way but not lead to a violation of a safety constraint. When using the correct PUF response for decoding, we receive the sensitive data; otherwise, we receive any of the safe alternatives.

Our approaches are not strictly limited to PUFs but can employ any hardware fingerprint that provides sufficient entropy and can ideally be queried dynamically, which offers stronger protection. However, in alignment with scenario 2 (MCU with SRAM), we choose SRAM PUFs described in the following section.

5.1 SRAM-Based Hardware Fingerprinting

While we can use the startup values of SRAM cells to construct a PUF (cf. Section 2.3.4), we must account for noise. In particular, not all SRAM cells exhibit consistent startup values, and their stability depends on environmental factors such as temperature [129, 162, 163]. To handle noise, *fuzzy extractors* [164] derive a stable key by using helper data that enables reliable reconstruction. For example, the reusable fuzzy extractor by Canetti et al. [165] relies on randomized sampling during enrollment, which affects the generated helper data.

Deterministic fuzzy extractor. We propose a fuzzy extractor that uses deterministic patterns for sampling [142, 143]. To achieve reliable reconstruction on resource-constrained MCUs, the extractor uses *stability masks* that permanently exclude unstable SRAM cells. Only bits that were observed to be stable across multiple startups and temperatures are retained. This significantly reduces the effective noise and, consequently, the amount of helper data required to suit the memory available on MCUs. The fuzzy extractor is used to reconstruct a predefined secret, which we denote as the *device identifier*. Concretely, during enrollment, this identifier is encrypted using the stable PUF response derived by the fuzzy extractor. At run time, the PUF response obtained at startup and the helper data are used to decrypt the identifier. If the reconstructed PUF response deviates beyond the tolerated noise level, decryption fails, as is the case on cloned devices.

On the original device, the reconstructed response differs only in a few tolerated bit positions, allowing correct decryption. On a different device, the PUF response differs substantially, resulting in an incorrect identifier. Although the device identifier is reconstructed via decryption, it can be interpreted as a bit string equivalent to a PUF response. Moreover, we can partition the identifier into multiple independent segments and treat each as a separate challenge–response pair to be used independently by our protection mechanisms MINOTAUROX and SPHINXOR.

Deriving cryptographic keys from memory-intrinsic entropy. As an alternative to the fuzzy extractor, we can employ our method for deriving cryptographic keys from memory-intrinsic entropy [144]. Our method assigns each physical memory cell an empirical probability of exhibiting a certain value, e.g., the likelihood that the startup value of an SRAM cell is 1. We can also use other forms of memory-intrinsic entropy, such as the reliability of a Rowhammer-induced bit flip in DRAM; cf. Section 4.1.3.

During enrollment, we first measure the memory entropy by repeatedly probing the memory and estimating per-cell probabilities. We then select a uniformly random key of fixed length, similar to the device identifier described above. For each key bit, we construct a vector representing a subset of the probed memory cells. A vector evaluates to a bit value by applying a simple threshold to the number of cells exhibiting the characteristic value. We assign cells to vectors such that the probability that a vector evaluates to the intended key bit is maximized while balancing correctness across all bits. The assignment relies on the measured probabilities and an optimization strategy that avoids concentrating unreliable cells in a single vector. We store only the list of selected physical cell addresses and a cryptographic hash of the key.

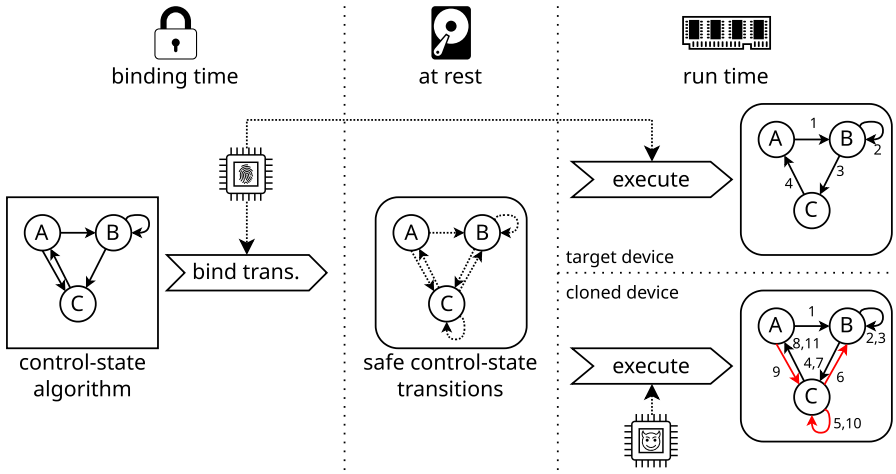


Figure 5.1: MINOTAUROX binds control-state transitions to a hardware fingerprint. Execution on the target device performs the intended control-state transitions, while execution on a cloned device results in unintended but safe control-state transitions.

At run time, we probe the selected memory cells, assign their values to the vectors, and apply the threshold function to reconstruct each key bit. The concatenation of all reconstructed bits yields a candidate key. We verify correctness by comparing the hash of the reconstructed key against the stored reference hash. If verification fails, we can repeat the reconstruction without leaking information about the key. Both approaches rely only on minimal helper data and are therefore suitable for deployment on resource-constrained embedded systems, such as MCUs.

5.2 MINOTAUROX

MINOTAUROX [166] allows us to protect the wide class of control-state algorithms [167]. A control-state algorithm is an algorithm whose behavior can be characterized by a finite set of abstract control states and well-defined transitions between them. A control state denotes a designated location in the algorithm state that determines which actions are executed in a given step.

Our approach protects control-state updates by making them depend on a hardware fingerprint, as we can see in Figure 5.1. We use a function that determines the update to the control state based on the current control state and the hardware fingerprint. On the target device, the function returns the intended next control state, while on a cloned device, it returns a randomly selected control state among those that do not lead to a violation of the defined safety properties. Even if the hardware fingerprint of the target device cannot be determined correctly, execution remains safe, as only safe control-state transitions are performed.

We propose to use symbolic execution to determine the safe control-state transitions. The verification of software safety by means of symbolic execution is not new [168]. However, to the best of our knowledge, it has not been used to support safety in the context of hardware–software binding yet.

```
if ctlState = i then
  rule
  if cond_1 then
    ctlState := j_1
  ...
  if cond_n then
    ctlState := j_n
```

Listing 5.1: General form of control-state ASM rules.

5.2.1 Design

This section introduces the key ideas of our approach. First, we must choose a PUF meeting the following requirements:

- It must be possible to query the PUF at run time, and its response time should not adversely affect the functionality of the target program; cf. Definition 4.
- The response of the PUF must be reasonably reliable so that the software executes correctly on the target machine with high probability. Ultimately, this probability depends on the fault-tolerance requirement of the specific application.
- The number of different challenge–response pairs of the PUF should ideally be greater than or equal to the number of control states of the target program. A smaller number also works, but it makes the protection less secure against reverse engineering.

We use ASMs (see Excursus 3) to formally specify the proposed copy-protection mechanism and to show its applicability to the class of *control-state ASMs* [167]. A control-state ASM [167, Def. 2.2.1] is an ASM whose rules follow the form defined in Listing 5.1 and capture the class of control-state algorithms—industrial control programs belong to this class [169]. Such ASMs have a finite set of control states $ctlState \in \{1, \ldots, n\}$ and terminate when no rule condition $cond_i$ is satisfied for a given control state i. Conceptually, control states act as the internal states of an FSM and represent different system modes. Control-state ASMs form a normal form for control-state algorithms; they support synchronous parallelism and manipulation of data structures.

EXCURSUS 3: ABSTRACT STATE MACHINES

An ASM with states of vocabulary Σ operates by executing its *transition rule* defined by the following basic rule forms:

- *Assignment rule*: $f(t_1, \ldots, t_n) := t_0$
  where $f \in \Sigma$ and $t_0, t_1, \ldots, t_n$ are terms of vocabulary Σ, updates the value of function f at location $(t_1, \ldots, t_n)$ to the value of t_0 in the next state.
- *Conditional rule*: `if` φ `then` r `[else` r'`]`
  Executes rule r if condition φ (a first-order sentence of vocabulary Σ) holds, otherwise executes optional rule r'.
- *Choose rule*: `choose` x `with` φ `do` r
  Executes rule r with an arbitrary value of x satisfying condition φ, expressing nondeterminism.
- *Let rule*: `let` $x = t$ `in` r
  Binds the value of term t to variable x for use within rule r, enabling term sharing and modular specification.

The execution of $f(t_1, \ldots, t_n) := t_0$ in a given state S proceeds as follows: (i) all parameters $t_0, t_1, \ldots, t_n$ are assigned their values, say $a_0, a_1, \ldots, a_n$, (ii) the value of $f(a_1, \ldots, a_n)$ is updated to a_0, which represents the value of $f(a_1, \ldots, a_n)$ in the next state. Pairs of a function name f (fixed by the signature) and optional arguments $(a_1, \ldots, a_n)$ of dynamic parameter values a_i, are called locations. They are the ASM concept of memory units, which are abstract representations of memory addressing. Location value pairs (l, a) are called *updates*, the basic units of state change.
The notion of an ASM *run* or *computation* is an instance of the classical notion of the computation of transition systems. An ASM computation step in a given state simultaneously executes all updates of all transition rules whose guard is true in the state. If and only if these updates are consistent, the result of their execution yields a next state. A set of updates is consistent if it does not contain pairs $(l, a), (l, b)$ of updates to a common location l with $a \neq b$. Simultaneous execution, as obtained in one step through the execution of a set of updates, provides a useful instrument for high-level design to locally describe a global state change.

Industrial control programs usually must satisfy certain safety and security properties. The main challenge for MINOTAUROX is to ensure that these properties are maintained. Our method achieves this using symbolic execution [170, 171] to determine the safe control-state transitions at design time. Symbolic execution (see Excursus 4) applies to ASMs the same way it does to high-level programming languages [172]. This is not surprising, as ASMs are executable abstract programs [173], ASM rules can be read as pseudocode over abstract data types.

EXCURSUS 4: SYMBOLIC EXECUTION

Symbolic execution abstractly executes a program covering multiple possible inputs of the program that share a particular execution path through the code. In concrete terms, instead of considering normal runs in which all locations take on actual values from the base set of the ASM states, we consider runs in which some state locations may take on symbols (depicted by lowercase Greek letters) representing arbitrary values. ASM runs will proceed as usual, except that the interpretation of some expressions will result in symbolic formulas.

We now demonstrate MINOTAUROX through a simple example, illustrating our approach using PUFs to bind the correct execution of a control program to its target machine. Moreover, it illustrates how symbolic execution can be used to ensure that the protected control ASM satisfies all safety constraints of the original control ASM. Following the example, we condense our approach into a precise method that can be automated. Given an appropriate PUF, our method can turn any given control-state ASM specification into an equivalent specification that will run as intended only on the target machine and satisfy the required safety constraints.

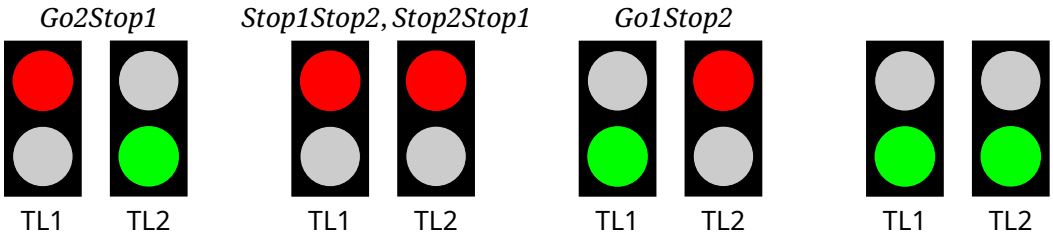


Figure 5.2: Possible physical states of a one-way stop-and-go traffic light and the corresponding control-state values. States in which both lights are green represent undesired behavior and therefore have no corresponding control-state value.

```
1WAYSTOPGOLIGHT =
  if phase in {Stop1Stop2, Go1Stop2} and Passed(phase) then
    StopLight(1) := ¬StopLight(1)
    GoLight(1) := ¬GoLight(1)
    if phase = Stop1Stop2 then
      phase := Go1Stop2
    else
      phase := Stop2Stop1
  if phase in {Stop2Stop1, Go2Stop1} and Passed(phase) then
    StopLight(2) := ¬StopLight(2)
    GoLight(2) := ¬GoLight(2)
    if phase = Stop2Stop1 then
      phase := Go2Stop1
    else
      phase := Stop1Stop2
```

Listing 5.2: Correct ASM specification of a one-way traffic light.

Example. We introduce our approach through a simple example, the ASM specification of a control algorithm for one-way stop-and-go traffic lights [169]. The proper behavior of this algorithm is defined by the ASM rule in Listing 5.2. For this example, there are four possible control states as shown in Figure 5.2. Both lights being green represents unsafe, undesirable behavior and therefore has no corresponding control-state value.

We propose to bind this control-state algorithm to the hardware using the sub-rule CHOOSEPHASE in Listing 5.3, which updates the value of *phase* to the correct next phase shown in Listing 5.2 only if the algorithm is executed on the target device. Otherwise, CHOOSEPHASE updates *phase* by choosing non-deterministically among the set of *safePhases*, i.e., the phases that will not lead to $GoLight(1) = GoLight(2) = true$, which would violate the safety constraint. This procedure ensures that if the software is copied verbatim to a cloned device, it will behave safely but also incorrectly and non-deterministically, making it challenging for an attacker to reverse engineer the protection.

We use symbolic execution to systematically determine the set of *safePhases*, specifically (i) to determine the necessary conditions for a next state to be safe, and (ii) to verify that this is indeed the case. Note that we must ensure that the protection mechanism will not introduce any bug that leads to an unsafe state. First, we assign a symbolic value to all relevant state locations:

$$L_0 = \{phase \mapsto \alpha, Passed(phase) \mapsto \beta, StopLight(1) \mapsto \gamma, GoLight(1) \mapsto \neg\gamma, \\ StopLight(2) \mapsto \delta, GoLight(2) \mapsto \neg\delta\}.$$

```
1 ChoosePhase =
2   let nextPhase = queryPUF(phase) in
3     if nextPhase in safePhases then
4       phase := nextPhase
5     else
6       choose p in safePhases do
7         phase := p
```

Listing 5.3: MINOTAUROX's hardware–software binding procedure.

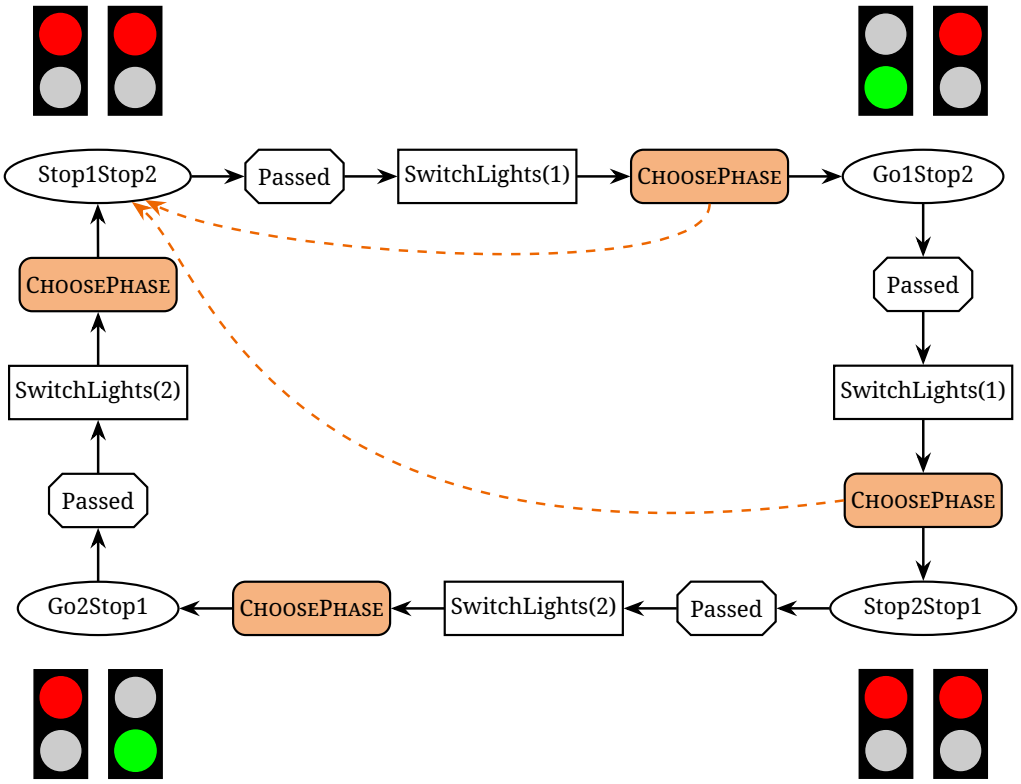


Figure 5.3: Control-state transitions of Protected1WayStopGoLight are determined by ChoosePhase, which selects the next control state based on a PUF responses. On the target device, ChoosePhase selects the intended next control state (→), while on cloned devices it may choose alternative safe control states (e.g., - →).

Note that the relevant locations—or equivalently locations of interest—are all those that correspond to dynamic function names that appear in the conditional statements of 1WayStopGoLight; cf. Listing 5.2. This might be excessive since there could be locations that are not relevant to preserving the required safety constraint, i.e., that $\neg(\mathit{GoLight}(1) \wedge \mathit{GoLight}(2))$. The problem is that if we only consider a subset of locations, there is the possibility that we miss an execution path that leads to a violation of the safety constraint that we want to preserve. In general, this is undecidable, and compromises might be necessary for complex applications.

Next, we look at all value path conditions resulting from executing the *protected* rule 1WayStopGoLight, i.e., Protected1WayStopGoLight, obtained by replacing lines 6, 8, 13 and 15 in 1WayStopGoLight with the sub-rule ChoosePhase; see also Figure 5.3. Without loss of generality, in the symbolic execution we can simply assume that ChoosePhase

assigns a symbolic value α' (possibly different from α) to *phase*, since, at this point, we are only interested in determining the path conditions where the current value α of *phase* can lead to $GoLight(1) = GoLight(2) = true$ in the next state. This gives us 4 path conditions and 4 corresponding symbolic state locations, one for each of the 4 possible symbolic step transitions $a, b, c,$ and d (executions) of PROTECTED1WAYSTOPGOLIGHT.

- Path conditions P_a and set of symbolic state locations L_a of transition a:

$$\begin{aligned} P_a \equiv & (\alpha = Stop1Stop2 \vee \alpha = Go1Stop2) \wedge \beta \wedge \alpha = Stop1Stop2 \\ L_a = & \{phase \mapsto \alpha', Passed(phase) \mapsto \beta, StopLight(1) \mapsto \neg\gamma, GoLight(1) \mapsto \gamma, \\ & StopLight(2) \mapsto \delta, GoLight(2) \mapsto \neg\delta\} \end{aligned}$$

- Path conditions P_b and set of symbolic state locations L_b of transition b:

$$\begin{aligned} P_b \equiv & (\alpha = Stop1Stop2 \vee \alpha = Go1Stop2) \wedge \beta \wedge \alpha \neq Stop1Stop2 \\ L_b = & \{phase \mapsto \alpha', Passed(phase) \mapsto \beta, StopLight(1) \mapsto \neg\gamma, GoLight(1) \mapsto \gamma, \\ & StopLight(2) \mapsto \delta, GoLight(2) \mapsto \neg\delta\} \end{aligned}$$

- Path conditions P_c and set of symbolic state locations L_c of transition c:

$$\begin{aligned} P_c \equiv & (\alpha = Stop2Stop1 \vee \alpha = Go2Stop1) \wedge \beta \wedge \alpha = Stop2Stop1 \\ L_c = & \{phase \mapsto \alpha', Passed(phase) \mapsto \beta, StopLight(1) \mapsto \gamma, GoLight(1) \mapsto \neg\gamma, \\ & StopLight(2) \mapsto \neg\delta, GoLight(2) \mapsto \delta\} \end{aligned}$$

- Path conditions P_d and set of symbolic state locations L_d of transition c:

$$\begin{aligned} P_d \equiv & (\alpha = Stop2Stop1 \vee \alpha = Go2Stop1) \wedge \beta \wedge \alpha \neq Stop2Stop1 \\ L_d = & \{phase \mapsto \alpha', Passed(phase) \mapsto \beta, StopLight(1) \mapsto \gamma, GoLight(1) \mapsto \neg\gamma, \\ & StopLight(2) \mapsto \neg\delta, GoLight(2) \mapsto \delta\} \end{aligned}$$

In the next step, we group the path conditions that lead to the same symbolic state. In our running example, we see that whenever the condition $P_a \vee P_b$ is satisfied, we get the symbolic state $L_a = L_b$. Likewise, whenever the condition $P_c \vee P_d$ is satisfied, we get the symbolic state $L_c = L_d$. Using standard logical equivalences, we can simplify the path conditions as follows:

$$\begin{aligned} P_a \vee P_b &\equiv (\alpha = Stop1Stop2 \vee \alpha = Go1Stop2) \wedge \beta \\ P_c \vee P_d &\equiv (\alpha = Stop2Stop1 \vee \alpha = Go2Stop1) \wedge \beta. \end{aligned}$$

Moreover, if we restrict our attention to the locations of interest and their symbolic values following a step transition, we can clearly see that in the next state the assertion $GoLight(1) = GoLight(2) = true$ holds, i.e., the safety constraint is violated, iff $Passed(phase)$ and

$$\begin{aligned} &\big((phase = Stop1Stop2 \vee phase = Go1Stop2) \wedge \neg GoLight(1) \wedge GoLight(2)\big) \vee \\ &\big((phase = Stop2Stop1 \vee phase = Go2Stop1) \wedge \neg GoLight(2) \wedge GoLight(1)\big). \end{aligned}$$

Finally, we get that *safePhases* is the set

$$\{x \in \{Stop1Stop2, Go1Stop2, Stop2Stop1, Go2Stop1\} \mid \neg \mathrm{Cond}(x)\}$$

where

$$\begin{aligned} \mathrm{Cond}(x) \equiv &\big((x = Stop1Stop2 \vee x = Go1Stop2) \wedge \neg GoLight(1) \wedge GoLight(2)\big) \vee \\ &\big((x = Stop2Stop1 \vee x = Go2Stop1) \wedge \neg GoLight(2) \wedge GoLight(1)\big)\big). \end{aligned}$$

```
1 ChooseCtlState(x, y) =
2   let nextCtlState = puf(g(x, y)) in
3     if nextCtlState in safeCtlStates then
4       ctlState := nextCtlState
5     else
6       choose s in safeCtlStates do
7         ctlState := s
```

Listing 5.4: MINOTAUROX uses a PUF to choose the next control state. Note that our implementation relies directly on the challenge rather than using g; see Listing 5.3.

Method. We now generalize the example to a precise method that can be automated. Let P be a control-state ASM with finitely many control states $ctlState \in \{1, \ldots, m\}$, whose rules $r_1, \ldots, r_m$ are all of the form defined in Listing 5.1. We assume, without loss of generality, that each r_i is guarded by **if** $ctlState = i$ **then** r_i. We define:

$$A = \{(i, j) \mid i, j \in \{1, \ldots, m\} \wedge subRuleOf(ctlState := j, r_i)\}$$

where $subRuleOf(s, r)$ is a Boolean function that evaluates to true only if s is a sub-rule of r. That is, $(i, j) \in A$ iff the rule r_i updates the control state to j.

The mechanical steps to copy-protect P with safety property Ψ using a (suitable) PUF $puf : C \to R$, with the set of challenges C and corresponding responses R, are described in the following seven steps. Note that step 1 is simply a prerequisite to implement the function *queryPUF* used in Listing 5.3 in the one-way traffic lights example. Indeed, the ChoosePhase rule results from applying step 2 of our method. The next two steps in our method simply refine the application of the proposed protection based on updating control states via *puf*, i.e., it corresponds to using ChoosePhase to select the next control state in our previous example. Steps 5–7 are a formalization and generalization of the symbolic execution analysis done in the one-way traffic lights example to determine the safe control-state transitions, i.e., to determine the set of *safePhases* in our example.

1. Fix an injection $f : C' \to R$ such that $C' \subseteq C$, $|C'| = |A|$ and $f(x) = puf(x)$, and a corresponding bijection $g : A \to C'$. Note that this function f is simply an injective restriction of *puf* to a subdomain C' of size $|A|$, as required to encode all possible control-state transitions of P (using the function g). If it is not possible to fix such an injective function f, this means that the function *puf* is not "big enough" to encode all control-state transitions. In this latter case, we simply consider that *puf* is not suitable for our protection strategy. An alternative would be to apply the protection to a subset of control states, but this would unnecessarily complicate the presentation of the general method.
2. For each tuple $(x, y) \in A$, define ChooseCtlState(x, y) as in Listing 5.4.
3. For each control-state rule r_i ($i = 1, \ldots, m$) of P, replace every occurrence of a sub-rule of the form $ctlState := j$ by the ChooseCtlState$(ctlState, j)$ sub-rule defined in the previous step.
4. In each control-state rule r_i ($i = 1, \ldots, m$) of P, replace the condition $ctlState = i$ in its guard by $ctlState \in \{x \mid \exists (y, z) \in A\ (g(y, z) = x \wedge y = i)\}$.

5. Assign a different symbolic value to all locations in the initial state that correspond to terms that appear in some conditional statements of an `if` rule in P, i.e., to all locations of interest. This results in an assignment *symVal*, such that $symVal(h(\bar{t})) = \alpha_i$ iff α_i is the symbolic value assigned to the location $(h, \bar{t})$.
6. Symbolically execute P on the (symbolic) initial state built in the previous step. This results in a set *Sym* of pairs of the form (φ_i, L_i), where φ_i is a path condition (as in standard symbolic execution) and L_i is a set that maps each term (corresponding to a location with a symbolic value) to its symbolic value in the successor state of the initial one.
7. Finally, set $safeCtlStates = \{x \in \{1, \ldots, m\} \mid \neg \mathrm{Cond}(x)\}$, where

$$\mathrm{Cond}(x) \equiv \bigvee_{(\varphi_i, L_i) \in Sym} \left(\varphi_i' \wedge \Psi(L_i)\right)$$

and φ_i' is the expression obtained by replacing the symbolic values in φ_i by the terms corresponding to the locations with these symbolic values in the initial state. In turn, $\Psi(L_i)$ is obtained by first replacing in Ψ (i.e., in the safety constraint) every term t_j by the symbolic expression $L_i(t_j)$ and then replacing the symbolic values in the resulting expression by terms corresponding to the locations with these symbolic values in the initial state.

5.2.2 Implementation

We have implemented a proof of concept [174] for the one-way traffic lights example in approximately 600 lines of C code. Our proof of concept targets a NUCLEO-F767ZI development board with an Arm Cortex-M7 MCU controlling a traffic light circuit. In Section 5.2.1, we require a dynamic PUF with a sufficient number of challenge–response pairs. While not fulfilling these requirements, we still opted to use an SRAM PUF due to hardware constraints. By querying the PUF at startup and applying error correction, we receive a device identifier. We can later access this identifier instead of the PUF for minimal delays—by only using portions of the identifier, we can reach a sufficient number of challenge–response pairs. For the security implications of this implementation, see Section 7.2.1.1.

5.3 SPHINXOR

Some control algorithms are not complex enough to be worth protecting—instead, the knowledge lies in the data used by these algorithms. Thus, we present SPHINXOR [142], our fail-safe approach to bind data to hardware, shown in Figure 5.4. At its core, SPHINXOR creates Boolean expressions based on sensitive data to be protected, alternative data leading to unintended but safe behavior, and a PUF response. At run time, we can restore the sensitive data by evaluating these Boolean expressions using the correct PUF response. When evaluated with any other PUF response, the encoding yields one of the predefined safe alternatives. While we instantiate this idea using Boolean expressions, our approach is not limited to this representation. Instead, we can employ any automated encoding that deterministically maps each PUF response to either the protected data or a safe alternative, rather than failing as classical encryption would.

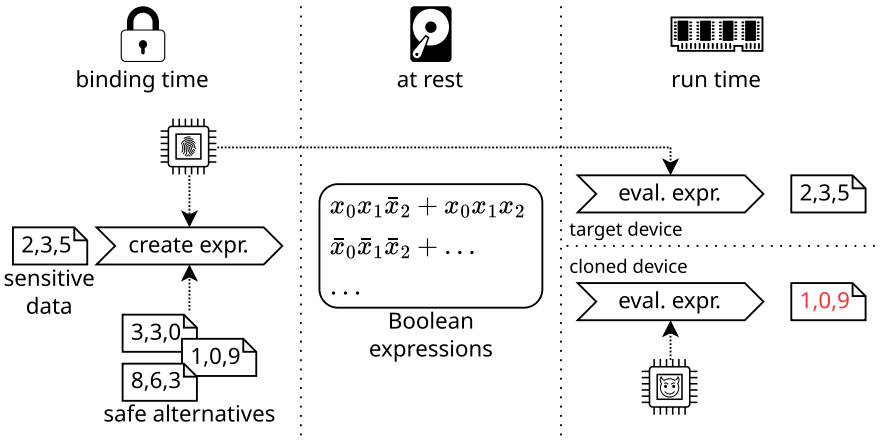


Figure 5.4: SPHINXOR creates Boolean expressions from sensitive data and safe alternatives such that evaluation on the target device yields the sensitive data. Evaluation on a cloned device results in one of the safe alternatives, leading to degraded but safe execution.

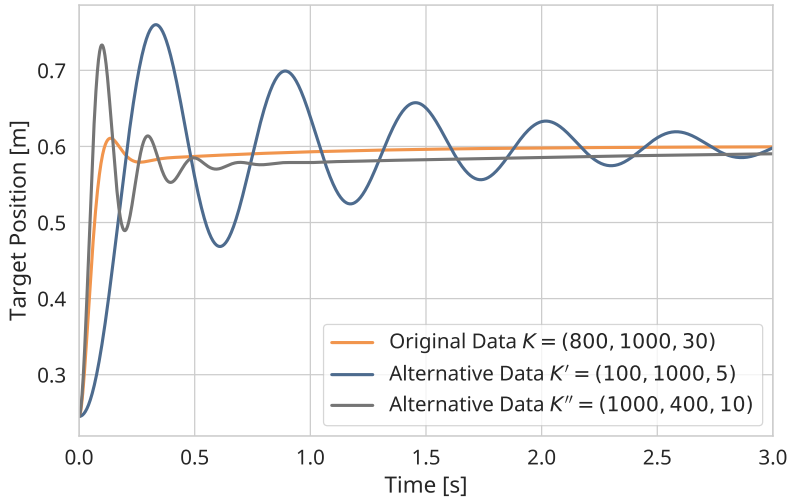


Figure 5.5: PID controller responses using different parameters: the controller reaches the target position at 0.6 m quickly and precisely with the original data K, oscillates around the target with the alternative data K', and fails to reach the target when using K''.

We illustrate our protection mechanism by considering a generic PID controller. They are used for a wide range of industrial applications, e.g., they can be used to control the position of a DC motor shaft. The ASM rule in Listing 5.5 specifies the step-by-step behavior of a generic PID controller algorithm. We assume the initial state of *mode* = `init` and that the time interval between controlling actions is *dt*. Note that *Kp*, *Ki* and *Kd* take constant values. For complex applications, they are typically fine-tuned by domain experts through trial and error, as there is no general analytical or computational method to determine their optimal values [175]. For complex controllers, finding optimal *Kp*, *Ki* and *Kd* parameters is not trivial, thus we aim to protect them. Figure 5.5 shows the effect of using non-optimal alternative values K' and K'': while the controller reaches the target position of the motor shaft efficiently when using the original data K, using the alternative data leads to oscillation around the target position or missing it altogether.

```
PIDCONTROLLER =
  if mode = init then
    previousIntegral := 0
    previousError := 0
    Kp := 800
    Ki := 1000
    Kd := 30
    mode := execute
  if mode = execute then
    if passed(dt) then
      let error = setPoint − measuredValue in
      let p = error in
      let i = previousIntegral + error · dt in
      let d = (error − previousError) / dt in
      let output = Kp · p + Ki · i + Kd · d in
        if safeLowerRange ≤ output ≤ safeUpperRange then
          adjustControlTo(output)
        if safeLowerRange > output then
          adjustControlTo(safeLowerRange)
        if output > safeUpperRange then
          adjustControlTo(safeUpperRange)
          previousIntegral := i
          previousError := error
```

Listing 5.5: Generic PID controller.

5.3.1 Design

We introduce our protection mechanism using a concrete example, namely safeguarding the optimal proportional, integral, and derivative parameters of a PID controller. We stress that the same mechanism can, in general, be applied to protect any valuable data within a program, provided that the target device offers a reliable hardware fingerprint for device identification. From the context, it should be clear that our protection mechanism is widely applicable and not restricted to specific scenarios, as long as fail-safe operation is supported by alternative values that result in degraded yet safe system behavior.

Assume we would like to protect the values Kp_0, Ki_0 and Kd_0 of a PID controller as the one shown in Listing 5.5 and that we have a list of $m > 2$ alternative values $Kp_1, Ki_1, Kd_1, \ldots, Kp_m, Ki_m, Kd_m$ for the constants of the PID controller. Here, m should be chosen as large as possible; see the security analysis in Section 7.2.1.2. The alternative values are suboptimal but stable, i.e., they converge to the desired point, but they do so in more steps than when using the optimal values. The idea is that only these alternative values can be obtained from a copy of the protected PID controller. Unless an attacker can get hold of the correct PUF responses from the target device, the actual optimal values Kp_0, Ki_0 and Kd_0 cannot be inferred from the protected program alone. Moreover, even if an attacker knows the correct PUF responses, they would still need to perform a complex dynamic analysis of the protected PID controller to reveal these values.

Let $T = \{(Kp_i, Ki_i, Kd_i) \mid 0 \leq i \leq m\}$ and let $V = \{x \mid y \in T \text{ and } x \text{ appears in } y\}$. Then $|V|$ is the number of different values that appear in the union of the domains of the relation T. We fix an arbitrary bijective function $toBin : V \cup V' \to \{0,1\}^n$, where V' is an arbitrary set of values of size $2^n - |V|$ such that $V' \cap V = \emptyset$. Of course, n must be greater than or

equal to $\lfloor \log_2 |V| \rfloor + 1$. We abuse notation and write $toBin(Kp_i, Ki_i, Kd_i)$, where $(Kp_i, Ki_i, Kd_i) \in T$, to denote the binary string $toBin(Kp_i) \cdot toBin(Ki_i) \cdot toBin(Kd_i)$ of length $3n$.

We further assume a procedure *encodeExprs* that protects a given tuple $\overline{\varphi}$ of Boolean expressions, so that they can be decoded securely by a function *recoverExprs* executed with the appropriate credentials on the target device. In our prototype, we implement *encodeExprs* using encryption, where the target device provides the decryption key—alternatives are, of course, possible. We let *hashValue* be the token used to verify successful execution of *recoverExprs*, i.e., $hash(recoverExprs(encodeExprs(\overline{\varphi}))) = hashValue$. Finally, we assume a function $queryPUF(k)$ that returns a binary substring of length k of the device identifier, which serves as the hardware fingerprint.

Our copy-protection mechanism is implemented as follows:

1. Define a partition of $\{0,1\}^k$, for some $k \geq \lfloor \log_2(m+1) \rfloor + 1$, into $m+1$ nonempty subsets A_i, where $0 \leq i \leq m$. Recall that $m+1$ is the number of different triples of proportional, integral, and derivative values in T.
2. Choose a c with $1 < c \leq m$ and define $f : \{0,1\}^k \to \{0,1\}^{3n}$ and $f' : \{0,1\}^k \to \{0,1\}^{3n}$ as follows:

$$f(x) = toBin(Kp_i, Ki_i, Kd_i), \qquad \text{if } x \in A_i;$$
$$f'(x) = \begin{cases} f(x), & \text{if } x \in A_i \text{ for } 1 \leq i \leq c, \\ toBin(Kp_c, Ki_c, Kd_c), & \text{if } x \in A_i \text{ for } i = 0 \text{ or } c < i \leq m. \end{cases}$$

3. Build tuples of Boolean expressions $\overline{\varphi} = (\varphi_0, \ldots, \varphi_{3n-1})$ and $\overline{\varphi}' = (\varphi'_0, \ldots, \varphi'_{3n-1})$ on Boolean variables $x_0, \ldots, x_{k-1}$ (using the procedure explained below), such that for every assignment $val : \{x_0, \ldots, x_{k-1}\} \to \{0,1\}$ and binary string $b_0 \ldots b_{3n-1} \in \{0,1\}^{3n}$, the following holds:

$$f(val(x_0), \ldots, val(x_{k-1})) = b_0 \ldots b_{3n-1} \quad \text{iff } val(\varphi_i) = b_i \text{ for all } i = 0, \ldots, 3n-1$$

and

$$f'(val(x_0), \ldots, val(x_{k-1})) = b_0, \ldots, b_{3n-1} \quad \text{iff } val(\varphi'_i) = b_i \text{ for all } i = 0, \ldots, 3n-1.$$

4. Set *encodedExprs* := $encodeExprs(\overline{\varphi})$.
5. Choose $queryPUF(k)$ so that its response belongs to A_0 on the target machine.
6. Modify the code of the PID controller by replacing the assignment of constant values to *Kp*, *Ki* and *Kd* (e.g., lines 5–7 in Listing 5.5) with the code in Listing 5.6. Thus, if $\overline{\varphi}$ is recovered successfully (lines 2–4), then the algorithm evaluates each of these Boolean expressions using the k bits in the response of *queryPUF* as assignment for the Boolean variables $x_0, \ldots, x_{k-1}$ (line 4), where the i-th bit is assigned to x_i for $i = 0, \ldots, k-1$. The function *eval* returns a binary string $b_0, \ldots, b_{3n-1}$ such that b_i is 1 iff φ_1 is true for the given assignment based on the response of *queryPUF*. Then, the inverse of the function *toBin* returns (provided $queryPUF \in A_0$) the optimal values for *Kp*, *Ki*, *Kd*. Otherwise, the algorithm assumes we are not on the target machine and uses the Boolean expressions $\overline{\varphi}'$ instead. In this case (line 6), the values assigned to *Kp*, *Ki*, *Kd* are, by construction of these Boolean expressions, among the suboptimal alternatives.

```
RecoverProtectedValuesPID =
  let φ̄ = recoverExprs(encodedExprs) in
    if hash(φ̄) = hashValue then
      Kp, Ki, Kd := toBin^-1(eval(φ̄, queryPUF(k)))
    else
      Kp, Ki, Kd := toBin^-1(eval(φ̄′, queryPUF(k)))
```

Listing 5.6: SPHINXOR's secure recovery of protected *Kp*, *Ki* and *Kd* values.

The Boolean expressions as required for step 3 above can be built using sum-of-products of literals. Let us define the concrete procedure for obtaining $\overline{\varphi}$. The procedure for $\overline{\varphi}'$ is identical, except that the function f' replaces f in the definition. For every $i = 0, \ldots, 3n-1$, we build the following "care" set:

$$C_i = \{(a_0, \ldots, a_{k-1}) \in \{0,1\}^k \mid f(a_0, \ldots, a_{k-1}) = b_0 \ldots b_{3n-1} \ \wedge \ b_i = 1\}.$$

Then, we can define each φ_i as a sum-of-products Boolean algebra expression of the form:

$$\varphi_i \equiv \sum_{(a_0,\ldots,a_{k-1}) \in C_i} (g(a_0) \cdots g(a_{k-1}))$$

where for $j = 0, \ldots, k-1$,

$$g(a_j) = \begin{cases} x_j, & \text{if } a_j = 1, \\ \bar{x}_j, & \text{if } a_j = 0. \end{cases}$$

If the care set C_i for some i is empty, then $\varphi_i \equiv 0$.

Let us illustrate the process of obtaining the appropriate Boolean expressions through a simple example.

Example. Let $T = \{(2,3,5), (3,3,0), (1,0,9), (8,6,3)\}$, where the optimal values for Kp_0, Ki_0 and Kd_0 are 2, 3 and 5, respectively. That is, the values we want to protect are those corresponding to the tuple $(2,3,5) \in T$. Then $V = \{0,1,2,3,5,6,8,9\}$ by definition. We interpret the function *toBin* as the standard conversion from decimal numbers in $\{0, \ldots, 15\}$ to binary numbers of length $n = 4$. For instance, $toBin(2,3,5) = toBin(2) \cdot toBin(3) \cdot toBin(5) = 0010\,0011\,0101$. We fix $k = 3$. We can now build the Boolean expressions needed to apply our strategy.

First, we take the following arbitrary partition of $\{0,1\}^3$ into $m+1 = |T|$ subsets (see step 1 above):

$$A_0 = \{000, 011\} \qquad A_1 = \{001, 010\}$$
$$A_2 = \{100, 101\} \qquad A_3 = \{110, 111\}$$

By definition in step 2 of our strategy, we obtain

$$f(000) = f(011) = toBin(2,3,5) = 0010\,0011\,0101$$
$$f(001) = f(010) = toBin(3,3,0) = 0011\,0011\,0000$$
$$f(100) = f(101) = toBin(1,0,9) = 0001\,0000\,1001$$
$$f(110) = f(111) = toBin(8,6,3) = 1000\,0110\,0011$$

Choosing $c = 2$, gives us that $f'(x) = f(x)$ whenever $x \notin A_0 \cup A_1$ and $f'(000) = f'(011) = f'(001) = f'(010) = toBin(1, 0, 9) = 0001\,0000\,1001$.

Then, the resulting Boolean expressions for f are:

$$\varphi_0 \equiv \psi_3 \qquad \varphi_1 \equiv 0 \qquad \varphi_2 \equiv \psi_0 + \psi_1$$
$$\varphi_3 \equiv \psi_1 + \psi_2 \qquad \varphi_4 \equiv 0 \qquad \varphi_5 \equiv \psi_3$$
$$\varphi_6 \equiv \psi_0 + \psi_1 + \psi_3 \qquad \varphi_7 \equiv \psi_0 + \psi_1 \qquad \varphi_8 \equiv \psi_2$$
$$\varphi_9 \equiv \psi_0 \qquad \varphi_{10} \equiv \psi_3 \qquad \varphi_{11} \equiv \psi_0 + \psi_2 + \psi_3$$

where

$$\psi_0 \equiv \bar{x}_0 \cdot \bar{x}_1 \cdot \bar{x}_2 + \bar{x}_0 \cdot x_1 \cdot x_2 \qquad \psi_1 \equiv \bar{x}_0 \cdot \bar{x}_1 \cdot x_2 + \bar{x}_0 \cdot x_1 \cdot \bar{x}_2$$
$$\psi_2 \equiv x_0 \cdot \bar{x}_1 \cdot \bar{x}_2 + x_0 \cdot \bar{x}_1 \cdot x_2 \qquad \psi_3 \equiv x_0 \cdot x_1 \cdot \bar{x}_2 + x_0 \cdot x_1 \cdot x_2$$

Regarding f', the Boolean expressions $\varphi'_0, \varphi'_1, \varphi'_4, \varphi'_5$ and φ'_{10} are exactly as their corresponding Boolean expressions φ_i defined above for f. The remaining Boolean expressions corresponding to f' are as follows:

$$\varphi'_2 \equiv \varphi'_7 \equiv \varphi'_9 \equiv 0 \qquad \varphi'_3 \equiv \varphi'_8 \equiv \psi_0 + \psi_1 + \psi_2$$
$$\varphi'_6 \equiv \psi_3 \qquad \varphi'_{11} \equiv \psi_0 + \psi_1 + \psi_2 + \psi_3$$

Note that the Boolean expressions obtained through the described procedure can be simplified in most cases. For instance, ψ_3 is equivalent to $x_0 \cdot x_1$ as its result is independent of the value assigned to x_2. Moreover, one can use a heuristic logic minimizer such as Espresso [176], to eliminate redundancies in an automated way. The result is not guaranteed to be the global minimum but is typically quite close. The advantage of applying logical minimization is that the resulting Boolean expressions tend to be much shorter and thus require less evaluation time.

5.3.2 Implementation

Our prototype is implemented in approximately 900 lines of C code and consists of three parts: (i) a simulated PID controller, (ii) an implementation of an SRAM PUF, and (iii) an implementation of the protection mechanism explained in Section 5.3.1. Like for MINOTAUROX, we developed our prototype for NUCLEO-F767ZI development boards with Arm Cortex-M7 MCUs; cf. Section 5.2.2. Based on this prototype, we built a demonstrator targeting MCUs used in industrial machines [177].

We enroll our SRAM PUF to one of the MCUs, the *target MCU*. We reserve the last 32 bytes of the 512 KiB of SRAM available on the MCU for our PUF. This allows us to extract a viable device identifier with 18 bytes. We implement the function *queryPUF* to return a substring of the identifier. We use up to 2 bytes returned by *queryPUF* as assignment for generating the Boolean expressions $\overline{\varphi}$, which we then minimize using Espresso [178]. We implement *encodeExprs* to encrypt the minimized expressions with AES-128 using the remaining 16 bytes of the identifier and *recoverExprs* so that it decrypts them. For added security, the encryption key should ideally come from a different source than *queryPUF*, e.g., from a different kind of PUF, as discussed in Section 7.2.3.2.

At startup of our prototype on the target MCU, we query the SRAM PUF. We then extract the key to decrypt the Boolean expressions $\overline{\varphi}$. Subsequently, we execute the simulated PID controller with the constants Kp_0, Ki_0 and Kd_0 obtained by evaluating $\overline{\varphi}$. In this case, we observe that the PID controller operates as expected.

Mens sana in corpore sano.

A healthy mind in a healthy body.

— Juvenal (Decimus Iunius Iuvenalis)

6 Hardware–Software Binding for NN Models

Contents

6.1 Locki 70
6.1.1 Design 72
6.1.2 Implementation 72

The rapid adoption of artificial intelligence (AI) has elevated the strategic value of trained ML models in both industrial and commercial settings. Neural networks (NNs; see Excursus 5) in particular have become central to a wide range of applications, yet developing them requires substantial investment in data acquisition, data curation, and large-scale training [179]. As a result, NN models constitute valuable assets that adversaries may attempt to copy or reverse engineer. Successful model extraction enables an attacker to bypass the full development pipeline and erodes the model owner's competitive advantage. This threat is particularly relevant in deployment scenarios where models are distributed to untrusted environments or executed on customer-controlled hardware, such as scenario 3 (NN models).

Excursus 5: Neural Networks

There are many kinds of (artificial) NNs such as feedforward neural networks (FNNs), recurrent neural networks (RNNs) and convolutional neural networks (CNNs). All have in common that their architecture can be defined by some type of directed graph with a set of nodes V, set of edges $E \subseteq V \times V$ and functions $(a_v)_{v \in V}$, where for every node $v \in V$ (usually excluding the input nodes), $a_v : \mathbb{R} \to \mathbb{R}$ is a continuous function known as *activation function* at v. A concrete NN model associates this skeleton architecture with a *weight* $w_e \in \mathbb{R}$ with every edge $e \in E$ and a *bias* b_v with every node $v \in V$. NN models compute functions, but the way in which this is done varies depending on the type of NN used. A constant in this sense is that the computation takes place in the nodes and is propagated through the graph, using the activation functions, weights, and biases. The weights and biases are learned from data.

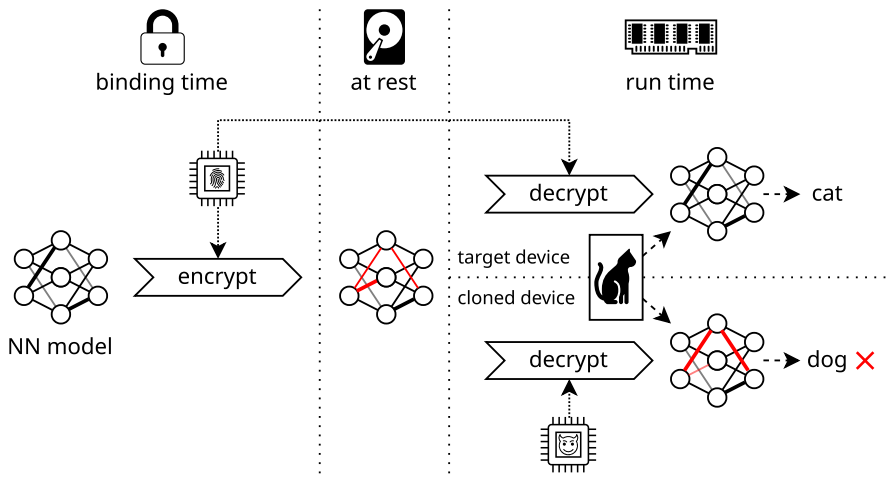


Figure 6.1: LOCKI uses a key derived from a hardware fingerprint to encrypt weights of an NN model. Decryption yields the original model only on the target device, while execution on a cloned device results in a degraded model.

Existing approaches such as watermarking or passports primarily support post-hoc attribution and ownership verification but do not provide strong protection against unauthorized use [180, 181]. Other defenses restrict the execution environment using TEEs [182, 183] or rely on fully homomorphic encryption (FHE) [184, 185], but these mechanisms either assume strong platform security guarantees or introduce considerable run-time overhead. Further approaches either protect only model outputs rather than the model itself [186], or require device-specific retraining, limiting scalability [187]. NN-Lock [188] uses a custom encryption scheme to encrypt all weights of a model and thus causes substantial inference-time overhead in practice. Consequently, there remains a need for protection mechanisms that hinder cloning and reverse engineering while avoiding retraining and impractical run-time overheads.

In this chapter, we introduce LOCKI, which applies hardware–software binding to NN models by encrypting the pre-trained model's weight. According to scenario 3 (NN models), we can use a TPM to seal the decryption key; thus, the model can only be decrypted on the target device. Alternatively, the key can be derived from an intrinsic hardware fingerprint, such as those described in Section 5.1. Since NN models cannot intrinsically ensure safety, we omit providing safe fallbacks and assume a separate safety system to be in place, such as a *safety monitor* [189] that detects safety violations or a *safety controller* [190] that overrides unsafe behavior. Although we developed the protection mechanisms presented in Chapters 4 and 5 for traditional software, they can be adapted to protect ML models—ultimately just another form of software—as outlined in Section 8.2.

6.1 LOCKI

We present LOCKI [191, 192], our protection mechanism that encrypts the weights of selected layers of a given NN model such that the model performs with full precision and recall only when the correct decryption key is provided at run time, as shown in Figure 6.1. Sealing the decryption key with a TPM or extracting it from a PUF binds the model to the

target device, hindering copying and reverse engineering. Unauthorized copies on cloned devices remain executable, preserving the appearance of a functional model, but suffer a substantial degradation in precision and recall due to decryption with an incorrect key.

Crucially, our approach offers inherent resistance to analytical weight-recovery attacks. When protected layers use non-invertible activation functions, encrypted weights cannot be algebraically reconstructed. When invertible activations are required, encrypting at least two consecutive layers induces nonlinear dependencies that render exact analytical extraction computationally infeasible.

Our method further minimizes exposure to reverse engineering by keeping model weights encrypted both when stored and in memory during run time, decrypting them only momentarily when needed inside the inference pipeline. This design narrows the attack surface even under some sophisticated adversarial capabilities. Our evaluation across multiple NN architectures demonstrates that the approach is practical and effective at preserving the model's performance under authorized use while degrading it for unauthorized replicas.

Excursus 6: Symmetric Encryption

In symmetric encryption, *cleartext* data m is encrypted into a *ciphertext* $c = \mathcal{E}_k(m)$ using a randomly chosen *secret key* k and a fixed publicly known algorithm $\mathcal{E}$. Conversely, a ciphertext c can be decrypted using k into the cleartext m. We write this as $m = \mathcal{E}_k^{-1}(c)$, where $\mathcal{E}^{-1}$ is an algorithm that decrypts c using the key k rather than the inverse of a function. The purpose of encryption is to make it infeasible to extract any information about m from c as long as k is not known. In order to achieve such a strong security property, encryption has to ensure that ciphertexts look like random data.

So-called *stream ciphers* are lightweight and fast symmetric encryption algorithms. Basically, they expand the key k into a stream of random-looking bits and then encrypt the cleartext by applying bitwise XOR to that stream and the cleartext. This resembles the one-time pad encryption but with a key that may be much shorter than the cleartext to be encrypted. In addition to the secret key k, they use a *nonce* n so that we have $c = \mathcal{E}_{k,n}(m)$. Such a nonce can be publicly known but must not be reused with the same key. When equal cleartexts, in our case NN weights, are encrypted with the same key, the resulting ciphertexts may be identical, which leaks equality information about the plaintexts. To prevent this, encryption schemes typically incorporate a nonce, a value that is available during decryption and ensures that identical plaintexts encrypt to different ciphertexts.

When an attacker can only read encrypted plaintexts, encryption with a stream cipher suffices to guarantee confidentiality. If ciphertexts can be modified, and the behavior of a system that decrypts the modified ciphertexts and uses the plaintexts can be observed, we have to take additional measures. Typically, a message authentication tag $t = T_k(c)$ is stored together with the ciphertext c that depends on the key k and cannot be forged, i.e., no valid tag can be produced for a modified c. As a result, any modification of ciphertexts can be detected, so that we can prevent active attacks. Modern stream ciphers like ChaCha20-Poly1305 [193, 194] incorporate such tags and automatically perform a verification of the tags during decryption.

6.1.1 Design

When an NN model is executed, each neuron—e.g., belonging to a dense layer—computes an output signal o from n input signals $v_1, ..., v_n$, weighted by corresponding weights $w_1, ..., w_n$, and transformed by an activation function $\mathcal{S}$:

$$o = \mathcal{S}(o'), \text{ where } o' = \sum_{j=1}^{n} w_j v_j.$$

We aim to encrypt the model using a stream cipher (see Excursus 6) with a key k such that the weights remain encrypted in memory at all times. For efficiency, each weight w_j should be encrypted independently of all others using the same key. The same should hold for decryption: each encrypted weight w'_j can be independently decrypted to w_j. This property is satisfied by stream ciphers such as ChaCha20-Poly1305 [193, 194]. Specifically, the stream cipher $\mathcal{E}$ uses the key k and a unique nonce, here the index j, to encrypt each weight:

$$w'_j = \mathcal{E}_{k,j}(w_j).$$

If the model is used without decrypting its encrypted weights with the correct key, its performance deteriorates as the weights remain encrypted, although the impact depends on the affected weights. This degradation can be used to deliberately control the model's behavior. Encrypting only a small subset of weights leads to reduced but still usable accuracy. In contrast, encrypting all weights causes the model to lose predictive capability entirely, effectively reducing it to the behavior of a random classifier.

Given the correct key k, we can decrypt a weight w'_j with the stream cipher used previously for encryption:

$$w_j = \mathcal{E}^{-1}_{k,j}(w'_j).$$

Moreover, we can compute the neuron's outgoing signal using the encrypted weights $w'_1, ..., w'_n$ and the activation function $\mathcal{S}$. This is done by decrypting the weights with the key k during the computation, based on the incoming signals $v_1, ..., v_n$:

$$o = \mathcal{S}\left(\sum_{j=1}^{n} \mathcal{E}^{-1}_{k,j}(w'_j) \cdot v_j\right).$$

6.1.2 Implementation

For our proof of concept, we leverage built-in extension mechanisms of existing deep learning frameworks to implement run-time decryption while preserving access to established optimization and execution pipelines. Without this integration, the loss of graph-level optimizations would significantly degrade run-time performance. To the best of our knowledge, no existing framework supports cryptographic operations at the granularity of individual neurons during execution. However, TensorFlow allows the definition of custom layers, which we use to realize encrypted layers within NN models. We used Python 3.10 and TensorFlow 2.14 for our implementation.

Our implementation is based on TensorFlow's Keras API and uses the `Layer` extension interface. We implemented an encrypted dense layer that replaces a standard dense layer. The encrypted layer stores its weight matrix and bias vector exclusively in encrypted form and overrides the `call` method to perform decryption at run time. For encryption, we use the ChaCha20 stream cipher as implemented in the PyCryptodome library with a 256-bit key and a 96-bit nonce.

To apply the protection, our proof of concept transforms an existing pre-trained NN model by selecting a densely connected layer and replacing it with its encrypted counterpart. The original parameters are extracted, encrypted, and transferred to the new layer. All other layers remain unchanged, preserving the original model architecture and behavior. This transformation does not require retraining.

During inference, the encrypted layer decrypts its parameters, performs the standard dense-layer computation, and immediately discards the decrypted values. Clear-text weights therefore exist only transiently during layer execution and are neither cached nor stored persistently in memory. Correct computation is possible only when the appropriate decryption key, which is available exclusively on the target device, is provided.

जानीयात्प्रेषणे भृत्यान्
बान्धवान्व्यसनागमे।
मित्रं चापत्तिकालेषु
भार्यां च विभवक्षये॥

Test a servant while in the discharge of his duty,
a relative in difficulty,
a friend in adversity,
and a wife in misfortune.

— Cāṇakya (चाणक्य)

7 Evaluation

Contents

7.1 Evaluation of Rowhammer-Based Approaches . . . 75
7.1.1 Security Analysis . . . 76
7.1.2 Empirical Evaluation . . . 82
7.1.3 Discussion . . . 88
7.2 Evaluation of PUF-Based Approaches . . . 91
7.2.1 Security Analysis . . . 91
7.2.2 Empirical Evaluation . . . 94
7.2.3 Discussion . . . 95
7.3 Evaluation of Approach for AI Models . . . 97
7.3.1 Security Analysis . . . 97
7.3.2 Empirical Evaluation . . . 99
7.3.3 Discussion . . . 101
7.4 Comparative Evaluation . . . 102
7.4.1 Security Comparison . . . 103
7.4.2 Implementation Comparison . . . 104

In this chapter, we analyze the security properties of our approaches, focusing on their resistance to copying and reverse engineering. We further evaluate the functionality and performance of their reference implementations. We discuss limitations and potential improvements of both the approaches and their implementations. The chapter follows the structure of Chapters 4 to 6, devoting one section to each chapter, and concludes with a comparative evaluation of security guarantees, deployment effort, and performance overheads.

7.1 Evaluation of Rowhammer-Based Approaches

In this section, we evaluate and discuss our Rowhammer-based protection mechanisms, GlueZilla and Angluerus, with respect to security, functionality, and performance.

7.1.1 Security Analysis

First, we analyze the security of our Rowhammer-based approaches in general given the threat model in Section 3.2, and then focus on specifics of our approaches GLUEZILLA and ANGLUERUS, respectively.

Static analysis. Our use of Rowhammer allows us to reconstruct the intended program without any explicit information at run time about the critical junction bits due to property 2 (bit flips in the full memory range) and property 3 (no static knowledge of victim cell locations). The protected binary therefore does not directly encode this information, so static analysis can only reveal the layout of the unintended program. Even if an attacker performs differential analysis across binaries compiled for different machines, the junction bits remain indistinguishable, since binary-layout differences alone provide no information about which bits are junction bits. Brute-force identification of critical bits becomes infeasible due to the exponential growth of possible bit combinations and the difficulty of reasoning about their behavioral effects. A detailed analysis of how an attacker may narrow the search space is provided in Sections 7.1.1.1 and 7.1.1.2.

By inspecting the loader map, the attacker can learn which page frame is assigned to each code page. However, due to property 1 (unclonable and unique bit flip pattern), the attacker knows neither the location nor the number of Rowhammer-susceptible cells these page frames contain. Due to Rowhammer property 3, even if the attacker learns the location of the victim row based on the aggressor rows, they cannot reliably infer the induced bit flips. When using complex hammer patterns, the victim row is not adjacent to the aggressor row and bit flips can happen anywhere in the program, providing additional protection.

Dynamic analysis. In this scenario, the attacker has a copy of the software instance and a cloned device. The protected program is always loaded at the same physical addresses, dictated by the loader map. Property 1 states that the number and location of Rowhammer-susceptible DRAM cells are different between any two machines. Therefore, on a cloned machine, the junction bits will not reside in the required Rowhammer-susceptible cells, leading to either no bit flips or unexpected bit flips. The attacker, therefore, has no incentive to perform any type of dynamic analysis on this program execution since they cannot learn anything about the intended behavior.

7.1.1.1 GLUEZILLA

In this section, we present an attacker model tailored to the threat model in Section 3.2. Accordingly, we do not model unrestricted dynamic analysis on the target device. The attacker assumes that at most one junction bit is used per instruction but that such junction bits may occur in many different instructions across the program. The following quantitative analysis therefore provides a lower bound on the attacker's effort.

We model the program as a sequence of x86 instructions $I = (i_1, \ldots, i_{|I|})$. Each instruction $i \in I$ is encoded by an opcode and potentially one or more optional fields as shown in Figure 4.7. Optional prefixes and the opcode determine whether an addressing-form specifier is present, consisting of the ModR/M and possibly a scale–index–base (SIB)

byte. Depending on the addressing mode, a displacement field may be present. Some instructions also specify an immediate operand.

Let B_i^x be the set of *junction bit candidates* of field x in instruction i, i.e., the set of bit positions within field x such that flipping the bit at that position yields another valid instruction. Specifically, B_i^{pre} denotes the set of junction bit candidates within the instruction prefix, B_i^{opc} within the opcode, B_i^{modrm} within the ModR/M byte, B_i^{sib} within the SIB byte, B_i^{disp} within the displacement field, and B_i^{imm} within the immediate field. If field x is not present in an instruction, then $B_i^x = \emptyset$.

We define the set of junction bit candidates for instruction i as

$$B_i = B_i^{\text{pre}} \cup B_i^{\text{opc}} \cup B_i^{\text{modrm}} \cup B_i^{\text{sib}} \cup B_i^{\text{disp}} \cup B_i^{\text{imm}}.$$

The number of junction bit candidates for instruction i is given by $k_i = |B_i|$, where $k_i \geq 0$. The total number of candidates across the program is

$$k = \sum_{i \in I} k_i.$$

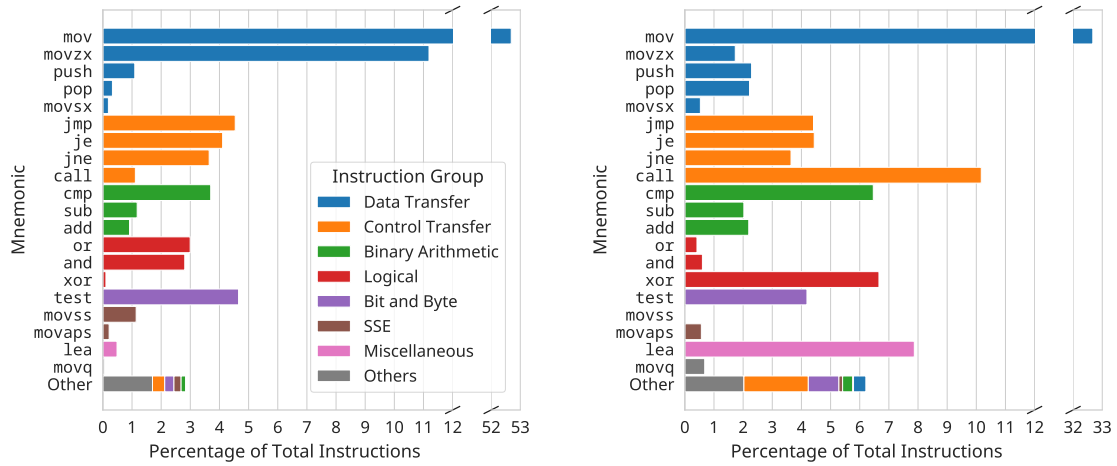
We express uncertainty in terms of bits by taking the base-2 logarithm, which yields the number of binary decisions required to identify the correct candidate under a uniform prior. The corresponding flip-position entropy is therefore

$$H_{\text{pos}} = \log_2 k.$$

Empirical PLC encoding characteristics. We empirically determine characteristics of instruction encoding to ground our model. Due to our focus on protecting industrial control programs, we analyzed a dataset of 640 PLC programs implemented in Structured Text [195]. We used OpenPLCv3, which transpiles Structured Text programs to C++ and subsequently translates to x86 assembly code using `g++`. Note that OpenPLCv3 adds auxiliary C++ code that may influence the observed instruction mix and encoding characteristics.

The PLC programs' assembly code contains on average approximately 10 000 instructions. The average observed opcode length is about 1.2 bytes and the average instruction length is 3.6 bytes. Encoding analysis shows that approximately 35% of instructions carry at least one prefix byte, 87% contain a ModR/M byte, fewer than 1% contain an SIB byte, roughly 55% contain a displacement, and about 23% contain an immediate. The observed SIB frequency is unusually low; in the `coreutils` programs, SIB bytes occur in about 15% of instructions, reflecting more complex addressing modes.

Figure 7.1a shows the most common instruction mnemonics of the PLC programs, classified by instruction groups according to Intel [159]. This illustrates that data and control transfer instructions, binary arithmetic, logical, and bit and byte instructions dominate PLC binaries, all within the group of general-purpose instructions. For comparison, Figure 7.1b shows the most common mnemonics of 113 `coreutils` programs. Akshintala et al. [196] observed a similar distribution for general-purpose C/C++ applications. In Table 4.2, we show how commonly observed instructions can be transformed to others with just one bit flip.



(a) Mnemonics of PLC programs [195].

(b) Mnemonics of `coreutils` programs.

Figure 7.1: Most common mnemonics of PLC programs compared to `coreutils` programs. Note the broken x-axes, required due to the disproportionately high prevalence of `mov` instructions.

Junction bits in different instruction fields. We model the estimated average number of valid junction bit candidates per instruction as

$$\tilde{k} = p^{\text{pre}} \cdot \tilde{k}^{\text{pre}} + p^{\text{opc}} \cdot \tilde{k}^{\text{opc}} + p^{\text{modrm}} \cdot \tilde{k}^{\text{modrm}} + p^{\text{sib}} \cdot \tilde{k}^{\text{sib}} + p^{\text{disp}} \cdot \tilde{k}^{\text{disp}} + p^{\text{imm}} \cdot \tilde{k}^{\text{imm}}.$$

Here, p^x denotes the probability that an instruction contains field x, and $\tilde{k}^x$ denotes the estimated number of format-preserving bit flip candidates contributed by field x when present. We use rounded-down field frequencies from our empirical analysis to estimate p^x:

$$p^{\text{pre}} = 0.3, \quad p^{\text{opc}} = 1, \quad p^{\text{modrm}} = 0.8, \quad p^{\text{sib}} = 0, \quad p^{\text{disp}} = 0.5, \quad p^{\text{imm}} = 0.2.$$

For instruction prefixes, we restrict attention to candidates preserving the prefix type. In the case of the common prefix `rex`, which extends operand width and register addressing in x86-64, only bit flips in the low bits preserve the prefix type, which yields only a few candidates for junction bits. We set $\tilde{k}^{\text{pre}} = 1$ to conservatively count only a single prefix bit as candidate when a prefix is present.

Bit flips in opcodes are constrained because a flipped opcode may require a different instruction format and thus invalidate the decoding of subsequent bytes. We therefore consider only *format-preserving opcode flips*, i.e., bit flips that keep the same requirements with respect to ModR/M, SIB, displacement, and immediate presence and size. We set $\tilde{k}^{\text{opc}} = 0.2$, reflecting that a non-trivial fraction of opcodes permit at least one format-preserving opcode bit flip; cf. Table 4.2.

For the ModR/M byte, we exclude flips in the R/M subfield because it determines the presence of the SIB byte and the displacement size and may therefore alter instruction length. Counting only bits in the Reg subfield guarantees validity and underestimates the true number of usable bit flips; we thus set $\tilde{k}^{\text{modrm}} = 3$. Bit flips in all three subfields of the SIB byte preserve instruction length and only modify effective address computation. Some bit flips may redirect accesses to invalid locations, others remain semantically plausible,

e.g., when alternative registers are used for address calculations within the same memory region. We therefore distinguish syntactic validity from semantic plausibility and model

$$\tilde{k}^{\text{sib}} = \alpha_{\text{sib}} \cdot 8, \quad 0 < \alpha_{\text{sib}} \leq 1.$$

Accounting for semantic filtering of base and index register changes, we set $\alpha_{\text{sib}} = 1/2$, which yields $\tilde{k}^{\text{sib}} = 4$.

A bit flip in a displacement or immediate preserves instruction validity because these fields are payload bits. Although flipping any displacement bit preserves valid decoding, only a subset of displacement values are semantically plausible because they still must reference valid memory addresses. Similar to $\tilde{k}^{\text{sib}}$, for an n-bit displacement we model

$$\tilde{k}^{\text{disp}} = \alpha_{\text{disp}} \cdot n, \quad 0 < \alpha_{\text{disp}} \leq 1.$$

For PLC programs where displacements are typically small and structured, we conservatively set $\alpha_{\text{disp}} = 1/4$. For an average length of 2 bytes, $n = 16$ bits and thus $\tilde{k}^{\text{disp}} = 4$. For an n-bit immediate, we count n junction bit candidates and therefore set $\tilde{k}^{\text{imm}} = 16$ bits as an average immediate size.

Expected number of junction bit candidates. Substituting the values defined above into the model yields an expected number of valid junction bit candidates per instruction of

$$\tilde{k} \approx 8.$$

Using $\tilde{k} \approx 8$, the flip-position entropy grows logarithmically with the instruction count. For programs with $|I| = 1\,000$, 10 000 and 100 000 instructions, we obtain $H_{\text{pos}} \approx 13$, 16, and 19 bits, respectively. An entropy of H_{pos} bits corresponds to approximately $2^{H_{\text{pos}}}$ valid junction bit hypotheses under this attacker model. For the analyzed PLC programs, the average binary size is approximately 10 000 instructions, which places the flip-position entropy at roughly 16 bits, corresponding to about $2^{16} \approx 66\,000$ junction bit hypotheses.

This value captures only positional uncertainty under the assumption of a single junction bit per instruction and therefore underestimates the true attacker work factor. In practice, an attacker must first rule out invalid bit positions, which can require substantial analysis, particularly for bits that influence memory addressing. Bit flips in such fields can reinterpret data in registers or select different registers; cf. Table 4.2. Due to variable instruction lengths, a single bit flip may also shorten an immediate field and cause the remaining bytes to be decoded as subsequent instruction. Furthermore, it is also possible to induce multiple adjacent bit flips within the same byte [197] or even bit flips in the program counter [198]. Accounting for these effects further increases the number of plausible junction bit hypotheses per instruction. Nevertheless, even under conservative assumptions, identifying and validating the correct junction bit hypotheses remains costly. An attacker may therefore resort to differential analysis.

Differential analysis. The attacker can perform a differential attack on multiple binary instances compiled for different machines. The unintended behavior and the set of junction bits are identical across instances. Only the order of code fragments and the amount of padding between them differ. These reorderings alone do not disclose which instructions contain junction bits.

Assume a DRAM row of 8 KiB, i.e., it contains 65 536 bit positions. Let n denote the number of junction bit candidate positions within the row. Let m be the attacker-assumed maximum number of bit flips that can be induced within the same row. Under this model, the attacker must consider all ways of selecting k distinct bit positions within the row, where $0 \leq k \leq m$. The number of sets of possible bit flip positions per row grows combinatorially with n and m:

$$\sum_{k=0}^{m} \binom{n}{k}.$$

We assume that the compiler emits code fragments consisting of 35% code and 65% nop instructions, reflecting the average size overhead for 100 junction instructions in SPEC CPU 2017 benchmarks determined in Section 7.1.2.2. Furthermore, we assume that 20% of code bits yield valid instructions when flipped, based on the determined average instruction length of 3.6 bytes and $\tilde{k} \approx 8$. On a single-channel machine, this results in approximately 4 600, 1.1×10^7 or 1.6×10^{10} sets of possible bit flip positions per row if the attacker assumes at most one, two or three bit flips, respectively. For each junction bit candidate, the attacker must reason about its effect on the program behavior, which is difficult to automate and renders brute-force attacks impractical due to prohibitive time requirements.

In addition, diversification can further distort the attacker's view; cf. Section 1.3.4. This includes introducing additional layout randomization and inserting code fragments that are never executed. Such measures break static analysis and further reduce the effectiveness of differential attacks.

7.1.1.2 ANGLUERUS

While GLUEZILLA protects instructions, ANGLUERUS complements this approach by protecting the initial values of data symbols. As a result, ANGLUERUS is not restricted to pairs of valid instructions. Instead, any bit within a symbol value can be flipped, and multiple bit flips can be employed more easily, as data objects may span a much larger number of bytes than individual instructions.

Analyzing the protected program statically reveals only the unintended behavior of the program. An attacker would need to determine the symbols and the bits within their values to which we applied our protection. They could do so by looking at the synthetic sections that contain data on the victim pages. Stripping the binary can make this task harder, e.g., by removing the section header table and symbols. However, it would still be possible to locate the junction symbols on otherwise empty pages. Thus, filling the space around the junction symbols with random data is crucial to render a brute-force attack impractical. Furthermore, the number of combinations of bit flips an attacker would need to correctly identify increases exponentially with the number of junction symbols.

	Data Symbols Per Program				Data Bytes Per Program			
Dataset	Min	Max	Mean	Median	Min	Max	Mean	Median
PLC-BEAD [195]	52	139	55.7	54	177	216 865	2 380.7	281
`coreutils`	1	2 125	228.4	142	4	47 008	3 777.9	2 182

Table 7.1: Number and size of data symbols in analyzed programs.

If a program contains only one protected 32-bit value on a page identified by an attacker, this requires at most 32 tries to identify the correct bit flip, assuming the attacker can recognize the correct behavior of the program. If we protect 5 symbols with one bit flip each and fill the remainder of the 4 KiB pages they are located on with random data, this causes an attacker to have to try up to $(4 \cdot 1\,024 \cdot 8)^5 = 3.8 \times 10^{22}$ combinations of bit flips. To contextualize this estimate, we analyzed the data symbols present in real-world binaries. Table 7.1 summarizes the results for PLC programs and `coreutils` programs also analyzed in Section 7.1.1.1. Even these comparatively small binaries contain dozens to hundreds of data symbols. Since the attacker's search space grows exponentially in the number of protected symbols, protecting tens of symbols substantially amplifies the infeasibility of static attacks. If an attacker can localize a victim page and identify the actual data stored on it, the brute-force effort is governed by the size of the protected symbol values. Table 7.1 shows that the amount of data per program is on the order of several KiB on average in both datasets. Let us consider the smallest observed PLC binary with 52 symbols and 177 bytes of data. On average, symbols in this program span about 3.4 bytes or roughly 27 bits. Assuming one bit flip per symbol and 27 junction bit candidates per symbol, the attacker must consider $27^5 \approx 1.4 \times 10^7$ possible combinations for 5 protected symbols or $27^{10} \approx 2.1 \times 10^{14}$ possible combinations for 10 protected symbols. Consequently, protecting multiple symbols leads to a combinatorial explosion, even when an attacker can precisely localize victim pages and identify on-page symbol values, and there is only a single junction bit per value.

The above estimates assume that all bit patterns of a protected data symbol are semantically distinct. In practice, some symbols admit only a small set of meaningful values, e.g., integers used to encode Boolean values or enumerations, or are typically initialized to a fixed constant such as 0, e.g., iterators. In contrast, many data symbols have non-trivial initial values and represent thresholds, memory offsets, lookup tables, cryptographic keys, images, ML models, or even machine code. These objects typically span many bytes as arrays or structured data and admit a large space of semantically distinct and valid values. Consequently, protecting only a limited number of symbols requires selecting symbols with complex initial values.

Even if an attacker can identify the protected symbols through static analysis, this knowledge alone is insufficient to restore the intended behavior. Replacing protected data with symbolic variables and performing symbolic execution exposes data dependencies and path conditions but yields a large space of feasible valuations without an oracle for correctness. Moreover, many values lead to valid but unintended executions, which symbolic execution cannot easily prune. As a result, identifying the correct bit flips still requires exhaustive exploration of a large candidate space, rendering such attacks impractical.

Several properties of ANGLUERUS frustrate the endeavor to dynamically analyze the protected program on the target device. Memory snapshotting by attaching a debugger to the protected program and dumping its memory after the program takes its intended form is hindered by the tracer. This technique is known as *debug blocking* [39] (cf. Excursus 2), as the Linux kernel only allows one debugger to be attached at once, and the tracer must be attached to receive the intended form. Allowing more than one debugger to be attached would likely require the attacker to modify the kernel or emulate the system. Modifying the kernel would break the mapping between symbols and bit flips and introducing additional delays would likely cause hammering to fail due to property 4 (bit flip success depends on execution speed).

If the attacker has full access to the target device, they could template the DRAM themselves and patch the binary. To frustrate the attacker in this scenario, we can include dummy bit flips (cf. Section 4.4.1) and unreliable bit flips for irrelevant data.

7.1.1.3 Combining GLUEZILLA and ANGLUERUS

The approaches of GLUEZILLA and ANGLUERUS are complementary and can thus be combined. An attacker then must explore a larger search space of junction bits in both instructions and data to restore intended behavior. In Section 7.1.1.1, we determined a flip-position entropy of about 16 bits, corresponding to roughly 2^{16} instruction-level junction bit hypotheses. If, in addition, ANGLUERUS protects five data symbols, each with 27 junction bit candidates as determined in Section 7.1.1.2, this yields 27^5 data-level hypotheses. Considering both effects jointly, the attacker must therefore explore approximately $2^{16} \cdot 27^5 \approx 9.2 \times 10^{11}$ hypotheses. Many incorrect hypotheses at the instruction or data level still result in valid but unintended executions, which hinders efficient pruning based on behavioral feedback. This illustrates that a combination of GLUEZILLA and ANGLUERUS can substantially increase the protection strength relative to applying only one of these approaches individually. The combined approach increases protection strength with only limited additional deployment effort, as GLUEZILLA and ANGLUERUS use the same memory template.

7.1.2 Empirical Evaluation

In this section, we evaluate the functionality of our prototypes GLUEZILLA and ANGLUERUS and quantify their overhead in terms of file size, startup time, and run time.

7.1.2.1 Functionality Evaluation

To evaluate the practicality of our prototypes, we applied them to the `coreutils` programs `ls` and `cp`. For GLUEZILLA, we constructed an unintended behavior for both programs and used the manual operation mode described in Section 4.3.1 to select junction instructions.

For `cp`, we made the unintended variant of the program delete rather than copy the source file. We implemented the unintended behavior by modifying the following:

1. Two function signatures to add an extra pointer indirection for some arguments.
2. The struct layout of two structs to put common fields at the same offset.
3. A function call that serves as movable junction instruction (cf. Section 4.3.1) with the `remove` function as unintended destination and the `copy` function as intended one.

ANGLUERUS reimplements this protection using a different mechanism:

1. Instead of flipping a bit in the function pointer, we place both the `remove` and `copy` function addresses in an array.
2. We protect the index variable used to select the function, resulting in the same behavioral difference between the target and cloned devices.

For `ls`, we built the unintended behavior by adding disruptive operations to the intended behavior such that it no longer prints the current directory correctly. We implemented this behavior using GLUEZILLA by adding the following:

1. A conditional function call that increments the pointer to the element to print. The condition always evaluates to true, so the unintended form skips an entry in each iteration. This condition acts as an unmovable junction instruction (cf. Section 4.3.1) in which we flip an operand so it evaluates to false in the intended execution.
2. A time dependence that jumps over the print code between 3 p.m. and 6 p.m. This jump is a movable junction instruction with the print code as the intended destination and the code thereafter as the unintended destination.
3. Incrementing the pointer to the current entry's name by two bytes causes each entry to omit its first two letters. The increment instruction is an unmovable junction instruction in which we flip the constant from 2 to 0.

In ANGLUERUS, we reimplemented these three changes using a different technique:

1. For changes 1 and 2, we added protected global constants to the condition expressions rather than flipping bits in the instructions themselves.
2. For change 3, we replaced the hardcoded constant with a global variable and protected its value.

We performed this experiment on both a DDR3 and a DDR4 machine; cf. Section 7.1.2.2. For each machine, we took two different DRAM modules that contain Rowhammer-susceptible cells, one of which is associated with the program by the GLUEZILLA or ANGLUERUS compiler, respectively. At run time, we verified that the intended execution is only unlocked on the associated module, while on the other module the program performs its unintended execution.

ANGLUERUS: Combining multiple bit flips. The Elliptic Curve Digital Signature Algorithm (ECDSA) [199] is a public key signature algorithm using random nonces. Reusing a nonce for encryption can be exploited to recover the private key [200]. The OpenSSL implementation of ECDSA uses three sources of randomness. We patched OpenSSL so that it has access to these sources only if executed as intended on the target device.

The first source of randomness is the `getrandom` system call or reading bytes from random devices, e.g., `/dev/urandom`. We patched out this source by removing the system call and using `/dev/zero` as the primary random device, so all read bytes have the value 0. We then flip a bit in ANGLUERUS's unintended mode to change this path to an invalid one on the target device, forcing OpenSSL to use another device and thus obtain a proper nonce.

Second, OpenSSL uses a digest to mitigate a potential random number generator (RNG) vulnerability. We add a maximum allowed length for the digest, initialized with a large enough power of two. This allows us to configure an unintended value of 0, effectively bypassing the mitigation.

Third, OpenSSL uses the address of an RNG to seed it. Due to ASLR, this address changes at each startup. Thus, we use a similar approach as before and limit the length of the address, setting it to 0 for unintended execution.

We implemented a small program that uses the patched OpenSSL version and uses ECDSA. We built this program using ANGLUERUS and noticed different nonces being used at each run of the program on the target device, as intended. When executed on a different device, the program reused the same supposed nonce for encryption, allowing us to recover the private key. This case study demonstrates that ANGLUERUS remains applicable even when the underlying bit flips are not fully reliable. One of the three bit flips occurring is sufficient to restore a source of randomness, so that OpenSSL works as intended.

7.1.2.2 Performance Evaluation

To evaluate the performance of our prototypes, we examine the file-size, startup-time and run-time overheads. We measure the increase in run time using the SPEC CPU 2017 benchmark suite. In the following, we first look at the peculiarities of GLUEZILLA and ANGLUERUS followed by a comparison of the overheads caused by our prototypes.

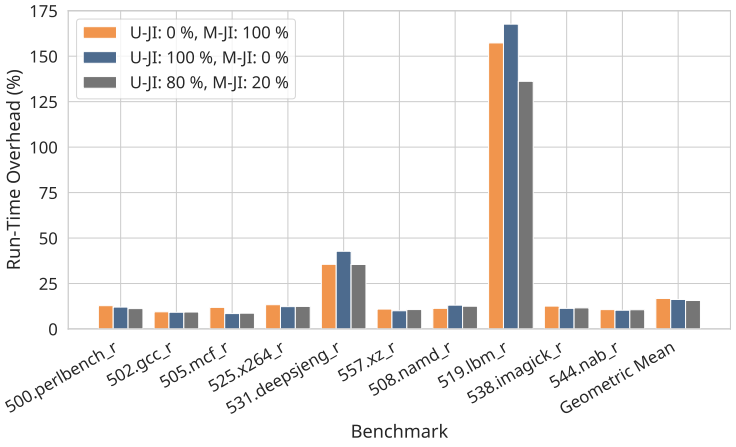
We performed our experiments on PCs equipped with either DDR3 or DDR4 memory. Our DDR3 machine has an Intel Core i7-2600 CPU and runs Fedora 39 with Linux 6.6. We used four DRAM modules with 12 GiB of DDR3 memory in total, operating at 1333 MT/s. Our DDR4 machine uses an Intel Core i7-8700K CPU, running Ubuntu 24.04 with Linux 6.8. We equipped the DDR4 machine with a 32 GiB DDR4 module from Samsung (S_3) or a 16 GiB DDR4 module from SK Hynix (H_2, similar to H_1 we used in Section 4.1.3.2).

For DDR3 memory, we used the double-sided hammer pattern to induce bit flips [81]. Due to the presence of Rowhammer mitigations on DDR4 such as TRR, we used many-sided [75, 151] and non-uniform hammer patterns [98].

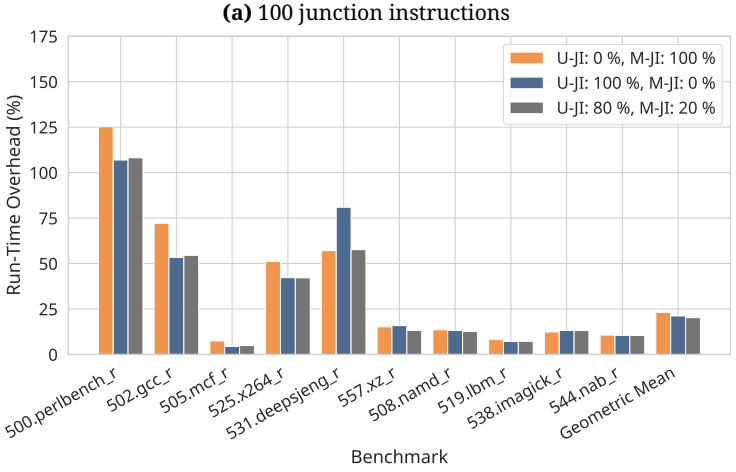
Our current prototypes perform all massaging and hammering at program launch, so these steps only add overhead when the protected program starts. The startup overhead heavily depends on the current state of the memory allocator. For the average case, when most of the required page frames are free, this takes about 2.7 seconds on our DDR3 machine for a program requiring about 1 000 page frames.

GLUEZILLA. At run time, the custom binary layout adds performance overhead because of reduced cache utility due to reduced code locality. We evaluated this overhead on all C benchmarks of the SPEC CPU 2017 benchmark suite on our DDR3 machine. We omit C++ benchmarks as our prototype does not currently support C++ exceptions. To reduce interference with other processes, we disabled SMT and turbo boost, isolated one core to which we pinned the protected program, and set the scaling governor to *performance*.

For each benchmark, we used the automatic operation mode, which selects a provided number of unmovable and movable junction instructions randomly and uniformly across the binary. This mode allows us to measure the effect of the custom layout on large quantities of junction instructions. Furthermore, we defined the unintended form to be equal to the intended form, so we do not require the Rowhammer step in our layout-related overhead measurements.



(a) 100 junction instructions



(b) 1% junction instructions

Figure 7.2: Performance impact of GLUEZILLA on SPEC CPU 2017. U-JI and M-JI denote *unmovable* and *movable* junction instructions, respectively.

Figure 7.2a shows the run-time overhead of binaries containing 100 uniformly distributed junction instructions, each requiring one junction bit. We divide these in unmovable and movable junction instructions in three configurations, represented by the three bars per benchmark. For each configuration, we ran three experiments per benchmark program—each with a different set of junction bits—and report the geometric mean. Figure 7.2b shows the overhead when 1% of all machine instructions in the binary are junction instructions. This setting is the maximum amount of junction instructions we could select with a memory template containing about 200 000 Rowhammer-susceptible DRAM cells.

Our current compiler prototype does not yet take into account any optimization regarding code placement and alignment, cache prefetching and utilization, branch probabilities, etc. It may even deteriorate the optimizations performed by earlier compiler passes, resulting in an almost worst-case overhead. This explains why both figures show a relatively high impact for the modified binary layout. However, in the realistic scenario of 100 junction instructions, the run-time overhead is limited to 16%. The geometric mean of the size overhead of all benchmarks over the three configurations is 185% for 100 junction instructions and 843% when 1% of all instructions are junction instructions.

We investigate these results using the hardware performance counters available on our benchmark machine. In our experiments using 100 junction instructions, all measured performance loss is caused by a variety of effects related to suboptimal code layout, including instruction cache misses, ITLB misses, and CPU frontend stalls. For `519.lbm_r`, however, we found that almost all overhead is caused by branch resteers in the CPU frontend. `519.lbm_r` is the smallest benchmark program and only includes around 1 800 instructions that fit on three 4 KiB pages in the baseline binary. Adding 100 junction instructions to small binaries results in many small code fragments, often containing only one or two instructions separated by large amounts of padding. To fit the spare distribution of Rowhammer-susceptible cells in physical memory, the padding can be multiple KiB long. As suggested by Godbolt [201], this greatly underuses the available entries in the branch target buffer (BTB) leading to many branch mispredictions. We found that the protected program had approximately 12 000 times more branch resteers compared to the baseline version. During a branch resteer, the branch address calculator updates an entry in the BTB that caused a branch misprediction or misidentification and restarts instruction fetching. These additional resteers account for 10% of all reference clock cycles.

In our experiments with 1% of all instructions as junction instructions, more benchmarks were affected by branch resteers. However, the impact was not as pronounced as in `519.lbm_r` with 100 junction instructions. The most affected benchmarks were `500.perlbench_r`, `502.gcc_r`, `525.x264_r` and `531.deepsjeng_r`. In these experiments, `519.lbm_r` includes only about 18 junction instructions (1% of its 1 800 assembly instructions). As a result, it experiences less run-time overhead compared to our earlier evaluation using 100 junction instructions.

ANGLUERUS. We use the automated mode to select 100 global symbols and randomly place a bit flip in each. We must thus rule out the benchmarks with fewer than 100 available symbols. We used our DDR4 machine and ran each experiment on a single CPU core to reduce interferences with other processes.

We used a many-sided pattern with 24 aggressor rows, i.e., with 49 physical frames (1 for the victim, 2 per aggressor) required for each junction bit. We observed a hammering time proportional to the number of junction bits and independent of the amount of memory available on the machine, as we expected. For our many-sided pattern with 24 aggressors hammered one million times each, we measured the hammering to take 1.4 s per victim. For comparison, we also ran GLUEZILLA on the DDR4 machine and measured hammering to take 2.3 s.

During intended execution, we observe a lower performance overhead compared to GLUEZILLA. Unlike GLUEZILLA, ANGLUERUS does not modify the code layout and therefore avoids the associated performance overhead. Since it reorders and separates the symbols, it creates data cache misses, resulting in negligible overhead. This overhead stays relatively small because ANGLUERUS does not split individual symbol values into pieces but moves whole values around, preventing cache misses when processing a value. The downside of this approach is that each symbol can have only one junction bit, whereas GLUEZILLA splits basic blocks to allow any instruction to have a junction bit.

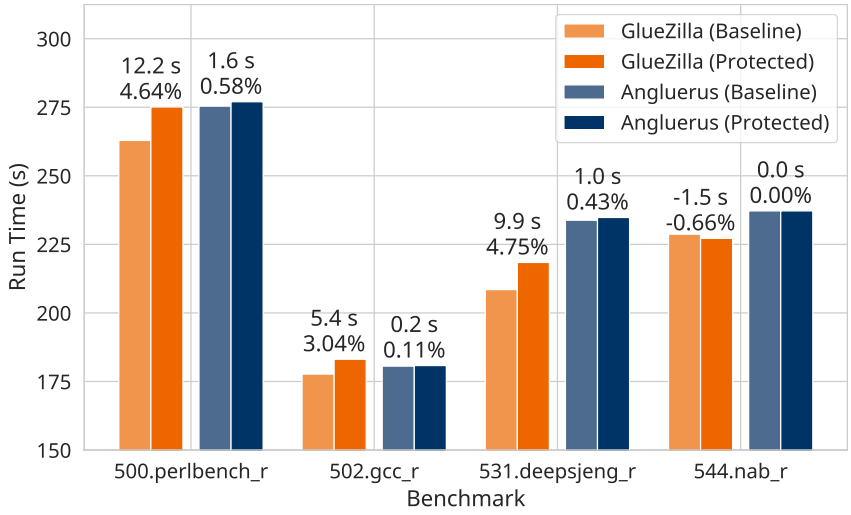


Figure 7.3: Performance impact of GLUEZILLA and ANGLUERUS on SPEC CPU 2017 when using 100 junction bits.

Comparison. We compare our prototypes GLUEZILLA and ANGLUERUS considering file size, startup time and run time.

In general, the file size increases by roughly 4 KiB per junction instruction or symbol because the linker moves them to separate pages. For ANGLUERUS, due to linking to the ANGLUERUS runtime library, the binary size increases by about 200 KiB. For GLUEZILLA, we do not include the runtime in the binary but rather as a shared library of 15 MiB, which must be present on the system and loaded into memory.

At startup, the protected program must allocate specific frames and hammer them. Because both prototypes use the same exhaustive approach for allocation, this step is faster on machines with less memory. We measured an average time overhead of 25 s when the machine is equipped with 16 GiB of memory and 60 s when using 32 GiB for ANGLUERUS, versus 36 s on the 16 GiB module, and 72 s on the 32 GiB module for GLUEZILLA. We observed variations of up to 30 percent around this average for the same program in the same conditions. Our hypothesis to explain these variations is that this time depends on the current state of the memory allocator and that the different runs may influence each other since they are all massaging this allocator.

We used SPEC CPU 2017 to evaluate and compare the run-time overhead of both prototypes. We prevented optimizations of the compiler and the CPU for evaluation purposes. GLUEZILLA builds on link-time optimization (LTO), which we could thus not disable, lowering the baseline. Note that GLUEZILLA and ANGLUERUS also build on different versions of LLVM, further impacting the baselines. For ANGLUERUS, this is necessary, among other things, to make the junction symbols `volatile` and to move them to additional pages. For comparison, we used 100 bit flips per benchmark for GLUEZILLA and therefore only benchmarks including at least 100 global symbols for ANGLUERUS. We ran each benchmark (cf. Figure 7.3) three times and determined the median run-time overhead. The geometric mean when using 100 junctions bits for protection is 4.06% for GLUEZILLA and 0.30% for ANGLUERUS.

7.1.3 Discussion

Rowhammer-based binding is limited to devices containing DRAM and increases memory requirements, making it unsuitable for, e.g., some internet of things (IoT) devices. Protection approaches based on PUFs with relaxed hardware requirements but weaker security guarantees might be a better choice for such systems; see Chapter 5.

Other DRAM fault mechanisms can preserve the core benefits of our approach. For example, RowPress [141] induces bit flips by holding a row open for extended periods, reducing access overhead and bypassing some Rowhammer mitigations. While more timing-sensitive, it may provide comparable unclonability when Rowhammer is ineffective.

We now consider attacks under full access to the target device. Dynamic binary translation (DBT) tools like Valgrind do not preserve binary layout or page-to-frame mappings and thus cannot observe the intended bit flips. Layout-preserving dynamic binary analysis (DBA) tools may leverage hardware tracing to analyze execution, but Rowhammer-based transitions remain difficult to detect due to property 5 (bit flip information isolated from OS and CPU) and the absence of explicit self-modifying instructions. Stealthy hammering primitives further remove distinctive signatures [82, 85, 96, 202–205], forcing periodic memory checks that either miss bit flips or introduce overhead sufficient to disrupt Rowhammer timing due to property 4 (bit flip success depends on execution speed).

Attackers could perform their own templating step using the full set of Rowhammer parameters—e.g., the aggressor rows, the hammer pattern, and access counts and speeds—they extracted from the binary. Using the loader map, they could identify the junction instruction or symbol value that gets loaded into a Rowhammer-susceptible cell. To raise the attack cost, we can introduce additional hammer sequences that flip irrelevant bits or induce no bit flips at all. Distinguishing effective from ineffective hammering then requires repeated dynamic analysis across many candidates, which increases the attack effort.

Rowhammer-based binding is inherently complementary to other protection mechanisms and can be combined with them to increase the overall level of protection. For example, pairing our Rowhammer-based binding with hardware dongles introduces dual-factor security: while the binding ensures that the software executes correctly only on the target device, the dongle can provide cryptographic authentication—potentially even serving as an additional hardware fingerprint. This combination significantly raises the complexity of an attack, as an adversary must replicate not only the physical DRAM behavior of the target device but also bypass or clone the dongle mechanism. Furthermore, integrating Rowhammer-based binding with software obfuscation adds another layer of defense against static and dynamic analysis, making reverse engineering more difficult and less economically viable.

Implementation limitations. Our prototypes presented in Sections 4.3 and 4.4 only focus on the functional requirements and, therefore, have some limitations related to security and performance. Both prototypes use a naive page-allocation strategy, which can fail under memory pressure and incurs startup overhead. Reserving pages at boot time or using hugepages eliminates these issues.

Our prototypes require access to the Linux pagemap, for which we launch the program with `sudo` and later switch to an unprivileged user. Because this is bad security practice,

we could leverage a kernel module that exposes a memory massaging interface limited to the memory of the current process. Such an interface would not give new capabilities to the attacker because it provides the same functionality as massaging from user space with the information in the loader map.

One limitation of our current prototypes is their susceptibility to memory snapshotting. Since our current implementations support only a one-time transition from unintended to intended execution at program start, an attacker could capture a full-memory snapshot after this transition and observe the intended code and data. A possible mitigation is to trigger Rowhammer-induced modifications during run time and to let the program transition back and forth between unintended and intended execution; see Section 4.3.4. Designing such a mechanism requires further research to avoid excessive run-time overhead.

7.1.3.1 GLUEZILLA

While the approach of GLUEZILLA demonstrates the feasibility of instruction-based binding using Rowhammer, it deliberately targets x86 systems with commodity DRAM. This design choice leverages the dense and redundant encoding of x86 instructions, which enables instruction transformations through single bit flips. As a consequence, extending the approach to other architectures may be less effective, as they may not offer sufficient instruction pairs that differ in only one bit.

Similarly, the sparse and irregular distribution of Rowhammer-susceptible bits on some DRAM devices motivates our focus on a limited number of carefully placed junction instructions. This requires alignment of selected instructions to specific physical DRAM cells, which we achieve using nop padding and basic block reordering, potentially increasing binary size and complicating debugging.

Implementation limitations. We choose to construct the required binary layout during the compilation phase because all necessary information is available at this stage. However, this approach results in large binaries of which a large portion is nop padding. Alternatively, we could put each page in a separate ELF segment and directly load it in the assigned page frame. This approach only has intra-page padding but requires more information to transfer between the compiler and the linker.

The padding also identifies the edges of each code fragment, which either contains a single junction instruction or is the destination of a movable junction instruction. This information narrows the adversary's search space for junction instructions, which makes it easier to reconstruct the intended program. For example, large padding regions can reveal likely intended destinations, allowing an adversary to correlate control-flow transfers that differ in a single bit and infer movable junction instruction locations.

We can eliminate this attack by limiting the attacker's capability to identify potential intended targets based on the leftover intra-page padding. One approach is to substitute the padding with dead code or with used code that does not have any positional constraints. In both cases, it should not be obvious where the substituted padding ends and the intended target starts by making the substituted padding as benign-looking as possible, for example:

1. Whenever the padding ends with a return instruction, there should be a function symbol associated with the succeeding code, i.e., the intended target.

2. If the padding ends with an unconditional jump, the padding should also include (dead) jumps to the succeeding code.
3. When the padding consists of dead code, we can include apparent data dependencies between the dead padding code and the intended target's code.
4. If the padding consists of used code, we can include data dependencies delimited by an opaque predicate between the padding and the intended target.

To reduce run-time overhead, we can define the binary layout problem as an optimization problem to increase code locality. The optimization variables include (i) the size and order of the code fragments, (ii) the location of the junction instruction within the fragment, and (iii) the selection of the Rowhammer-susceptible cell with respect to its page frame offset.

To demonstrate feasibility, we rely on manual selection of junction instructions, which can require significant effort depending on the required security level, i.e., the distance between the intended and unintended forms. Automating this process requires combined syntactic and semantic analyses to identify safe and inconspicuous junction instructions.

7.1.3.2 ANGLUERUS

Execution on cloned devices might result in unexpected and potentially unsafe behavior due to unforeseen bit flips. Unsafe behavior could be circumvented by introducing a mechanism that terminates the program on startup when it does not detect the target device, i.e., some expected bit flips do not occur. However, this mechanism would decrease the stealthiness of ANGLUERUS, making safety systems a better choice [189, 190].

Implementation limitations. The ANGLUERUS prototype currently protects global variables and constants. Support can be extended to local constants, string literals, and `static` locals, which reside in `.rodata` or `.data`, whereas most other locals remain ineligible due to stack or heap allocation. Therefore, we have implemented an LLVM pass that promotes locals to `volatile` globals.

Currently, it is rather easy for an attacker to identify the junction symbols once they find a way to dump the memory of the protected program. This is due to the padding the junction symbol values have on their distinct pages resulting from them being put at certain offsets to align them with the positions of memory cells vulnerable to Rowhammer. We could use innocent-looking data as padding, including misleading patterns indicating, e.g., file format headers or cryptographic constants. Furthermore, we can apply software diversity techniques [52, 54, 206–208] to fully randomize the data section and thus prevent differential analysis across binaries.

We can move the functionality of the tracer to a standalone program to increase the stealthiness because the protected binary would not contain any code indicating the use of Rowhammer. An attacker unaware of the external process performing Rowhammer cannot infer the intended form of the program from the protected binary alone. To further increase stealthiness, hammering during run time and reverting the bit flips can hide the intended form of the program due to it never being completely present in memory. Supporting run-time hammering requires reducing hammering latency. Our

initial experiments reduced hammering time by over 90%, at the cost of lower bit-flip reliability and thus increased templating effort.

Another drawback of the current prototype is the manual selection of symbols and the potential adaptation of the program. Automating this process could lead to unwanted run-time behavior and would require additional analyses to rule out harmful changes. We therefore envision a mechanism that assists the programmer by identifying candidate symbols to which the protection can be applied automatically, while leaving the final choice to the programmer. We could also target compiler-introduced data structures that guide control flow or data hidden from the programmer, such as the virtual function table (vtable) index used in C++ for dynamic dispatch. Modifying the vtable index on a cloned device would allow us to redirect execution to a different function higher in the class hierarchy.

7.2 Evaluation of PUF-Based Approaches

We first analyze the security of our PUF-based protection mechanisms MINOTAUROX and SPHINXOR. We then evaluate the functionality and performance of their reference implementations. Finally, we discuss the approaches and their limitations.

7.2.1 Security Analysis

We analyze the security properties of MINOTAUROX and SPHINXOR, focusing on their resistance to static and dynamic attacks under the threat model in Section 3.2.

7.2.1.1 MINOTAUROX

The safety of MINOTAUROX, our proposed method to protect control-state software, is ensured by construction. To eliminate the protection, an attacker needs to perform an analysis that enables them to determine the correct value of *nextCtlState* every time the rule in Listing 5.4 is called, or equivalently, whenever the current *ctlState* is updated. The attacker can only access a binary of the target program, which they can analyze using static and dynamic techniques.

Static analysis. We assume that an attacker can gain access to any number of protected binaries compiled using different PUFs, i.e., for different target devices. Static analysis may reveal structural information about the program. In particular, an attacker can identify where the control state is stored, which values it may take, and where control-state updates are performed. They may also determine the sets of control-state values that lead to a safe next step, i.e., the potential values of *nextCtlState*.

However, this information does not disclose the concrete values that *nextCtlState* should take at different execution stages. In particular, the attacker cannot determine the value of the function *puf* (see line 2 of Listing 5.4) for a given challenge using static analysis. Even if the attacker successfully decompiles the binary and bypasses any applied obfuscation, this still provides no information about the challenge–response pairs of *puf*.

An attacker could attempt to understand the full logic of the control-state program from the decompiled code. This task is time-consuming and complex, even for the simple

example of a one-way traffic light control algorithm in Section 5.2.1. In fact, writing the algorithm from scratch would likely be faster. As the complexity and number of control states increase, this reverse-engineering task quickly becomes impractical. Therefore, pure static analysis alone is not a significant threat to our protection method.

Dynamic analysis. We consider two scenarios for dynamic analysis. In the first one, the attacker has access to a cloned device. This gives the attacker the ability to execute the program repeatedly and observe its run-time behavior, which is an advantage over pure static analysis. However, it still does not allow the attacker to predict *nextCtlState*, since the PUF cannot be cloned. Thus, program executions on this hardware remain non-deterministic with respect to *nextCtlState*.

In the second scenario, the attacker can run the program on the target device, i.e., the unique system where the function *puf* will return the correct responses. Then, the attacker can trace the correct program flow via dynamic analysis. This can still be time-consuming, in particular if the system is complex with many possible execution paths. Moreover, this dynamic analysis can interfere with PUF responses. For example, operations on DRAM are time-dependent, including the Rowhammer PUF [121, 131], for instance, where memory must be accessed at high frequency, which would be hindered by other processes. The attacker could bypass this by querying the PUF separately from the protected program and then integrating the responses into the binary, but this would, once again, require a complex, time-consuming, and detailed analysis. This relative weakness can be dealt with by using complementary obfuscation techniques and/or physical protection mechanisms.

7.2.1.2 SPHINXOR

For illustrative purposes, we consider the attacker's options to break the protection applied to a PID controller, which corresponds to discovering the optimal values Kp_0, Ki_0 and Kd_0 used by the target MCU.

Static analysis. The attacker gets access to the protected binary. We assume they can successfully decompile this binary. In doing so, they can learn that the values of Kp, Ki and Kd are calculated using the subroutine specified in Listing 5.6. Moreover, they can also learn the content of the constants *hashValue*, k, the Boolean expressions $\overline{\varphi}'$, and the static functions *toBin*, $toBin^{-1}$ and *eval*. That is, we assume the attacker can find out all details except the responses of *recoverExprs* and *queryPUF*. This is naturally the case in a static analysis, since the underlying PUF responses can only be obtained by accessing the target device.

With the given information, an attacker can obtain the alternative (suboptimal) values $Kp_1, Ki_1, Kd_1, \ldots, Kp_m, Ki_m, Kd_m$ by calculating $toBin^{-1}(eval(\overline{\varphi}', s_i))$ for all 2^k binary strings s_i of length k and collecting the results into a set S. However, this neither reveals the optimal values Kp_0, Ki_0, Kd_0 used on the target MCU by the PID controller nor provides sufficient information to infer them logically. In the unlikely scenario where the attacker obtains the correct Boolean expressions $\overline{\varphi}$ by other means, static analysis can reveal how these expressions differ from the fallback $\overline{\varphi}'$. To prevent these differences from uniquely identifying the optimal values, we construct $\overline{\varphi}$ to include multiple suboptimal values in addition to the optimal ones. Consequently, without additional dynamic information, these differences do not uniquely identify the optimal values Kp_0, Ki_0, Kd_0. We therefore conclude that static analysis alone is insufficient to break the protection.

Dynamic analysis on cloned MCU. The attacker now also has access to a clone of the MCU bound to the protected binary. Unless the attacker can somehow bypass the protection afforded by *recoverExprs*, the program executed on this cloned device uses the Boolean expressions $\overline{\varphi}'$ to decode the values for Kp, Ki and Kd. Indeed, in our prototype the decryption key is, by construction of the PUF (see Section 5.1), incorrect. Independently of the chosen assignment, these alternative Boolean expressions $\overline{\varphi}'$ never return the protected values Kp_0, Ki_0 and Kd_0.

Therefore, an attacker has no incentive to perform dynamic analysis on the cloned MCU. This only changes if the attacker can somehow gain access to the Boolean expressions $\overline{\varphi}$. We stress that this would be an unlikely event considering our prototype, as it would require getting a copy of the decryption key, e.g., via industrial espionage. Assuming the attacker has access to $\overline{\varphi}$, then they could perform the following dynamic analysis on the cloned device:

1. For all 2^k binary strings s_i of length k, collect the results of $toBin^{-1}(eval(\overline{\varphi}, s_i))$ in set S and of $toBin^{-1}(eval(\overline{\varphi}', s_i))$ in S'.
2. Evaluate the results of running the PID algorithm on the cloned device for all triples (Kp_i, Ki_i, Kd_i) in $S \setminus S'$, choosing the one with optimal results.

In the worst case, step 1 involves evaluating the Boolean expressions with 2^n assignments, where n is the number of variables x. However, doing so is possible with relatively low effort. Logic minimization techniques can reduce the number of assignments that must be considered for the Boolean variables drastically, unless the size of $S \setminus S'$ is of similar order of magnitude as the number of assignments. Step 2 is time-consuming, since for each value in $S \setminus S'$ the attacker needs to run the protected program, record the outputs, and identify Kp_i, Ki_i and Kd_i with the best performance using domain knowledge. We can conclude that the protected values can be identified if the attacker knows the Boolean expressions $\overline{\varphi}$. However, this requires considerable reverse engineering effort if $\overline{\varphi}$ contains not only the values from $\overline{\varphi}'$ and the secret value, but also additional safe alternatives that are not contained in $\overline{\varphi}'$. Whether this effort compensates the gain for the attacker clearly depends on the application at hand. For PID controllers, this is likely not the case.

Dynamic analysis on target MCU. In our threat model (cf. Section 3.2), we assume that there is sufficient protection against reading the memory, registers, and execution flow in place. This implies that debugging and communication interfaces, such as JTAG, SWD and UART are disabled or blocked. In addition, manufacturer-specific functions such as readout protection are activated. Using an MCU also means that all core components, such as the processor and SRAM are housed on a single chip, making it difficult to intercept the communication within; this would require sophisticated physical attacks such as chip scraping. Under these conditions, even a dynamic analysis on the target device cannot directly reveal the correct data values. The only relative advantage would be that the attacker could now observe the optimal outputs on the target MCU for different inputs. However, they would still not be able to read the PUF responses (nor the values of Kp_0, Ki_0 and Kd_0 for that matter) from memory at run time because the debugging interface was disabled. This makes the attack described for dynamic analysis on a cloned MCU ineffective in recovering the protected data, even though we now have access to the target MCU.

Considering a man-at-the-end scenario where an attacker has full access to the target MCU and the protected software instance bound to it, they can simply read the protected values Kp_0, Ki_0 and Kd_0 from memory. This scenario can be mitigated by adding another security layer such as leveraging TEEs for handling sensitive data, which we leave for future work.

7.2.2 Empirical Evaluation

We empirically evaluate the implementations of MINOTAUROX and SPHINXOR in terms of functionality and performance.

7.2.2.1 MINOTAUROX

Functionality evaluation. Our proof of concept, described in Section 5.2.2, illustrates how MINOTAUROX affects the behavior of a simple control-state algorithm when executed on the target device and on a cloned device [174]. We deploy the protected control software to a microcontroller board resembling those used in industrial systems, controlling LEDs to simulate traffic lights. Executing the protected firmware on the target board produces the expected sequence of control states (or phases) and therefore correct traffic light behavior. We then extract the firmware using an off-the-shelf flashing and debugging tool and flash the extracted firmware to a second, equal board. Although the cloned device continues to operate safely—at no point do both lights turn green simultaneously—it no longer follows the intended control sequence. Individual control states are repeated or altered depending on the PUF-dependent control-state selection, leading to behavior that is safe but functionally incorrect. This confirms that the proposed mechanism provides effective protection against cloning and reverse engineering. For reference, the unprotected control software executes identically on both boards.

Performance evaluation. Since we have implemented only a proof of concept for a simple example to demonstrate the feasibility of MINOTAUROX, it does not permit a representative empirical performance evaluation. However, we provide a theoretical analysis of the performance overhead introduced by the proposed protection mechanism. Each control-state transition requires querying the PUF. The response time of PUFs can vary greatly, depending on the underlying hardware. For our approach, we ideally use a dynamic PUF; cf. Definition 4. Considering our traffic light example, control-state transitions occur sparsely, thus response times of several seconds are acceptable. In this case, the PUF can even be queried in the time windows between control-state transitions, eliminating the performance overhead completely. For example, the Rowhammer PUF can be queried in a few seconds [131] and would thus be suitable.

The DRAM Latency PUF can be evaluated in 88 ms [122], making it a viable option in more time-constrained scenarios. However, for industrial control program with real-time constraints, even this might still be too high. In addition, as demonstrated by scenario 2 (MCU with SRAM), we might face hardware constraints that prevent the use of dynamic PUFs. As discussed in Section 5.1 and also implemented in our proof of concept, we can resort to query an SRAM PUF at startup of the device and extract a device identifier for later use. This identifier can then be used instead of a PUF response, reducing the introduced overhead to a minimum. However, this decreases the security our approach provides, since we have to store the device identifier in memory.

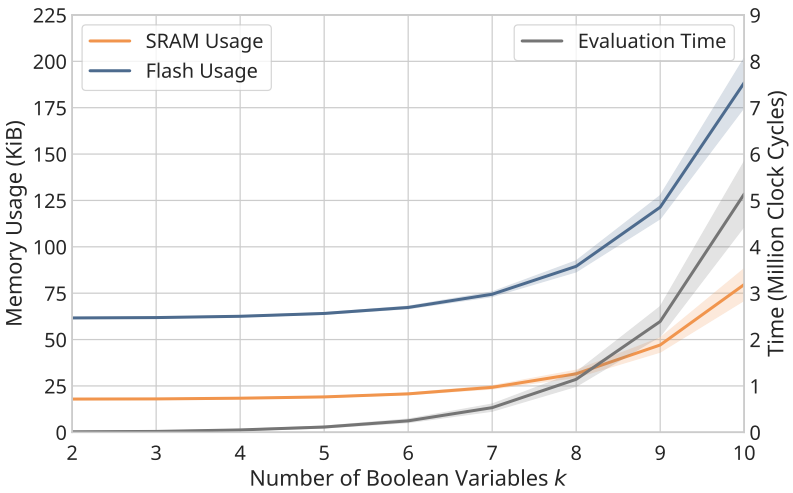


Figure 7.4: Performance evaluation of SPHINXOR's prototype for different PUF sizes. The error bands show the standard deviation for different numbers of alternative data values m.

7.2.2.2 SPHINXOR

Functionality evaluation. We take the view of a potential attacker and copy the SPHINXOR prototype (cf. Section 5.3.2) to an MCU with exactly the same specifications, the *cloned MCU*. Upon execution of the protected program, we observe that the key obtained from the PUF response does not decrypt the Boolean expressions correctly. We then see that the Boolean expressions $\overline{\varphi}'$ are used as fallback option and evaluated with a Boolean assignment obtained from the (incorrect) PUF response. The PID controller thus uses values for Kp, Ki and Kd that are not optimal. Although it still reaches a stable point, it does it inefficiently, performing clearly worse than on the target MCU. Breaking the encryption to obtain $\overline{\varphi}$ does not improve the performance on the cloned MCU, as the PID controller still uses an incorrect assignment for the Boolean expressions (due to an incorrect PUF response), and thus uses non-optimal values for Kp, Ki and Kd.

Performance evaluation. To evaluate the performance of our prototype, we consider the memory footprint of the protection mechanism and the added time penalty. For the latter, we measured the evaluation time of the resulting Boolean expressions for different numbers k of Boolean variables as well as different numbers of alternative data values m. The results are shown in Figure 7.4. We calculated memory usage with *PlatformIO*, which adds the size of the `.text` and `.data` sections of the binary program for the flash usage, and the size of the `.bss` and `.data` sections for the memory usage [209]. With larger values for k, the size of the Boolean expressions $\overline{\varphi}$ increases exponentially (as expected), whereas m has only limited influence. With exponential growth in the size of the Boolean expressions, SRAM and flash memory usage grow exponentially, as does the startup time of the prototype due to longer evaluation times. For $k = 6$, the average evaluation time is 1.14 ms at a CPU frequency of 216 MHz, for $k = 10$ it is 23.79 ms and thus still acceptable.

7.2.3 Discussion

We discuss the strengths and limitations of MINOTAUROX and SPHINXOR.

7.2.3.1 MINOTAUROX

In its current initial form, MINOTAUROX requires us to manually identify the control states of the target program. Additionally, we require a formal specification of the safety constraints that need to be preserved. The safety constraints necessarily need to be translated into a symbolic form so that symbolic execution can be applied to prove their preservation. In future work, we plan to apply this protection method to an industrial case study, developing the necessary tools to automate the required tasks. This would allow us to evaluate aspects such as the scalability of the proposed method and the concrete effort required to reverse engineer it.

Note that step 7 of our method in Section 5.2.1 might result in long formulae whose evaluation could unacceptably affect the run time of a complex control-state algorithm. In principle, we can simplify the resulting expressions using logical equivalences, as in the example presented in Section 5.2.1. Theorem provers in general, and satisfiability modulo theories (SMT) solvers in particular, can assist in this task. The latter have a long history of successful application in conjunction with symbolic execution.

7.2.3.2 SPHINXOR

A main advantage of using Boolean expressions to encode the protected values instead of just encrypting them, is that this enables the implementation of dual-layer protection. In our prototype, we simply use two different PUF responses, one to decrypt the Boolean expressions and another one to obtain the correct assignment for the Boolean variables in those expressions. Instead, we could consider a different source for the key needed to decrypt the Boolean expressions or even replace encryption with GLUEZILLA's approach, provided the device has DRAM. If an attacker can break the encryption, they still cannot access the target values unless they can also access the required Boolean assignment. Nevertheless, the attacker could pinpoint a set of data values that includes the secret ones. However, they cannot distinguish the correct (secret) values from the suboptimal ones unless the behavior of the machine in each case is evaluated by a domain expert. Finally, if an attacker cannot decrypt the Boolean expressions but instead can get the right assignment, the fallback Boolean expressions still do *not* return the secret data values.

On legacy hardware, we could use our mechanism to implement a secure update procedure, which transmits encrypted data to a device, and the best performance can only be achieved based on the correct PUF response. If we consider newer MCUs that have additional security features, such as TEEs, our dual-layer approach could be applied to receiving sensitive data in encrypted form and transmitting it decrypted into the secure enclave, which then evaluates the Boolean expressions, thus representing an additional protection layer.

Essential future work should consider how to better handle the secret data values later at run time so that even a memory dump does not result in these values being revealed. A possible approach could be to keep these data values in registers. Another possibility and a further security layer would be to take advantage of the mentioned TEEs.

A limiting factor discussed in Section 7.2.2.2 is the exponential growth in size of the resulting Boolean expressions in relation to the considered number of Boolean variables. Nevertheless, our protection mechanism increases memory usage only during startups, and the time penalty is within reasonable limits for the target applications. We leave the exploration of alternative, potentially more efficient encodings to future work.

7.3 Evaluation of Approach for AI Models

We evaluate the security of LOCKI, our approach for protecting NN model weights, followed by an empirical evaluation of its implementation and a discussion of its limitations.

7.3.1 Security Analysis

First, we show how an attacker could recover all weights of a single neuron, bypassing any encryption of its weights, provided that a number of inputs and outputs are known and the activation function $\mathcal{S}$ is strictly monotonic. Consider a neuron with inputs $v_1, \ldots, v_n$ and weights $w_1, \ldots, w_n$. Its pre-activation sum and final output are

$$o' = \sum_{i=j}^{n} w_j v_j, \text{ and } o = \mathcal{S}(o'), \text{ respectively.}$$

If an attacker can observe (or choose) a collection of input vectors $\mathbf{v}^{(i)} = (v_1^{(i)}, \ldots, v_n^{(i)})$ together with the corresponding outputs $o^{(i)}$, then invertibility of $\mathcal{S}$ lets them compute

$$\mathcal{S}^{-1}(o^{(i)}) = o'^{(i)} = \sum_{j=1}^{n} w_j v_j^{(i)}.$$

Thus, each pair $(\mathbf{v}^{(i)}, o'^{(i)})$ yields a linear equation in $w_1, \ldots, w_n$. Collecting n such equations gives rise to a linear system

$$V\mathbf{w} = \mathbf{o}',$$

where $V_{ij} = v_j^{(i)}$, $\mathbf{w} = (w_1, \ldots, w_n)^\intercal$, and $\mathbf{o}' = (o'^{(1)}, \ldots, o'^{(n)})^\intercal$. If the input vectors $\mathbf{v}^{(i)}$ are linearly independent, then the matrix V is invertible and the system has a unique solution:

$$\mathbf{w} = V^{-1}\mathbf{o}'.$$

This yields exact recovery of all weights in polynomial time. In summary, any neuron whose activation function is strictly monotonic, and thus invertible, can be fully compromised if the attacker observes enough linearly independent input–output pairs.

Now let us reason about the attacker's ability to perform such a weight-recovery attack, assuming partial access to the model's training data. Specifically, we consider the scenario where the attacker knows several input–output pairs from the training set and aims to compute the incoming and outgoing signals of a single neuron for one such data point.

Let us first assume that all model weights are known, *except* for those assigned to the neuron under attack. In this case, the attacker can compute the neuron's incoming signals by executing the known part of the model and recording the values that reach the neuron's inputs. However, computing the neuron's *output* signal is more challenging. Due to the applied encryption scheme, the neuron's direct output is not visible during execution.

Instead, the attacker must attempt to invert the model. Given a known input–output pair and the full set of weights (except those of the target neuron), the attacker can propagate the known output backwards through the network. By inverting each layer step by step, assuming that the activation functions are invertible and all other weights are known,

they can compute the pre-activation outputs of each preceding neuron until they reach the output of the neuron in question.

As a result, the attacker eventually obtains both the input signals $v_1, \ldots, v_n$ and the output signal o of the neuron for multiple training samples. Since the activation function is invertible, they can compute the pre-activation value. Using this information, they can now simply compute the original weights as described above, solving a system of linear equations using standard linear algebra techniques. Although feasible in principle, this attack still entails considerable reverse-engineering effort in practice.

Next, let us consider the implications of protecting two or more neurons in the model. There are two options for this, each with different consequences for security:

1. *Protecting neurons in the same layer*: In this scenario, each protected neuron receives the same set of incoming signals, and their outputs can be recovered independently. Crucially, the same set of input–output pairs for the entire model can be reused to attack each protected neuron individually. As a result, the attacker needs only as many input–output examples as the maximum number of inputs to any protected neuron in the layer in order to recover the full set of weights for all protected neurons. This makes the layer particularly vulnerable: recovery of all protected weights can be parallelized and requires only a minimal number of queries.
2. *Protecting consecutive neurons*: Here, the output of one protected neuron serves as the input to the next protected neuron. In this configuration, weight recovery becomes substantially more difficult. Since the intermediate values between protected neurons are not observable due to the encryption scheme, the attacker cannot apply the linear inversion strategy directly. Even if the activation functions are invertible, the attacker would need to solve a nested system of nonlinear equations, where each unknown depends on the previous unknowns. This introduces greater computational complexity and increased uncertainty, potentially deterring or delaying the attack.

Consider the following scenario involving two densely connected layers, K and L, each consisting of n neurons. The outgoing signals of neurons in layer K serve as the incoming signals for neurons in layer L. Let $k \in K$ and $l \in L$ denote the first neuron in each respective layer. Assume both k and l are protected, meaning that their associated weights are encrypted and thus unknown to the attacker.

Neuron k receives incoming signals $v_1, \ldots, v_n$ and has weights $w_1, \ldots, w_n$. Its output signal is denoted o_k. Neuron l receives inputs $u_1, \ldots, u_n$ with weights $m_1, \ldots, m_n$ and produces output o_l.

Let the activation function $\mathcal{S}$ be invertible. Then, the pre-activation value o_l' of neuron l can be written as:

$$o_l' = \mathcal{S}^{-1}(o_l) = \sum_{i=1}^{n} m_i u_i.$$

Since u_1 is the output of neuron k, we have:

$$u_1 = o_k = \mathcal{S}\left(\sum_{j=1}^{n} w_j v_j\right).$$

Model	Dataset	Model Type	Classes	Arch.	Layers	Weights	% Encr.
Digits	[210]	Image Classif.	10	FNN	3	305 312	33.3%
Fashion	[211]	Image Classif.	10	FNN	3	305 312	33.3%
Audio	[212]	Audio Recogn.	8	CNN	10	4 876 829	32.9%
Text	[213]	Text Analysis	2	RNN	5	480 053	0.004%

Table 7.2: Overview of NN models used in our experiments, including the dataset used, details about the models, the total number of weights, and the percentage of encrypted weights after applying our protection.

Since the activation function $\mathcal{S}$ is nonlinear in most NNs (e.g., sigmoid, ReLU, tanh), substituting this into the expression for o'_l yields a nonlinear equation:

$$o'_l = m_1 \cdot \mathcal{S}\left(\sum_{j=1}^{n} w_j v_j\right) + \sum_{i=2}^{n} m_i u_i.$$

This equation contains the unknown weights $w_1, \ldots, w_n$ of neuron k and $m_1, \ldots, m_n$ of neuron l. For each training example where the attacker knows the full input vector $v_1, \ldots, v_n$, the secondary inputs $u_2, \ldots, u_n$, and the final output o_l, the attacker obtains one such equation. Collecting enough such equations results in a nonlinear system over the $2n$ unknown weights.

In contrast to the single-neuron case, where recovery of weights reduces to solving a linear system, this setting leads to a nonlinear system of equations. The nested application of $\mathcal{S}$ significantly increases the complexity of weight recovery. Nevertheless, in principle, if $\mathcal{S}$ is known, invertible, and sufficiently well-behaved (e.g., differentiable), and enough training samples are available, numerical methods such as gradient-based solvers or symbolic solvers may still be applied to attempt recovery.

To sum up, we can draw two important conclusions:

1. Attacks based on weight recovery can be effectively prevented by using a *non-invertible activation function*, since the attacker cannot compute the pre-activation values needed to set up the linear (or nonlinear) systems of equations.
2. If the use of an *invertible activation function* is required for the model architecture or performance, then our proposed encryption scheme should be applied to *at least two consecutive layers* to achieve a meaningful level of protection. This change introduces nonlinearity and dependency between protected neurons, making it very hard for an attacker to isolate and recover individual sets of weights.

7.3.2 Empirical Evaluation

Functionality evaluation. We conducted empirical experiments on different NN models to validate our proof of concept; cf. Section 6.1.2. Table 7.2 provides an overview of the evaluated models, which (i) classify handwritten digits and (ii) different categories of clothing, (iii) recognize predefined audio commands, and (iv) perform sentiment analysis on reviews. For each model, we performed a total of 1 500 test runs. First, we executed the original models 500 times, measuring precision, recall, run time, and memory usage on standard hardware without GPU acceleration. Next, we applied our protection

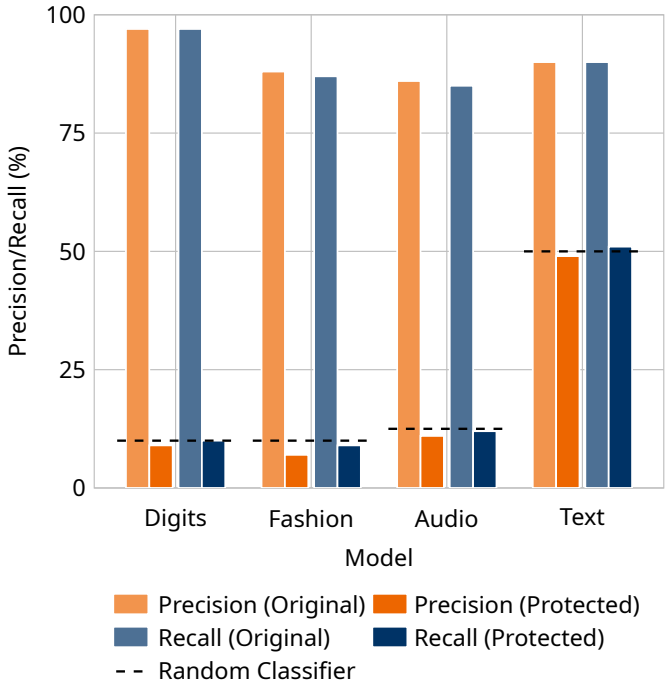


Figure 7.5: Comparing precision and recall of original and protected models. The dashed line indicates the expected performance of a random classifier, which depends on the number of classes; cf. Table 7.2.

scheme to densely connected layers within each model. We then conducted additional test runs with the protection enabled, running the model 500 times with both the correct and an incorrect decryption key.

Figure 7.5 shows the original models' precision and recall as well as their performance when an incorrect key is used to decrypt the protected model, which is the case on a cloned device. Precision and recall of the protected model are identical to those of the original model iff the correct secret key stored on the target device is used. All experiments show that decryption with an incorrect key derived on a cloned device causes a significant drop in both precision and recall to the level of a random classifier, as indicated by the dashed lines in Figure 7.5. Note that for the *Text* model, a random classifier achieves 50% accuracy, as the task is binary classification.

Performance evaluation. Figure 7.6 illustrates the increase in run time and memory usage after applying our protection strategy. In all of our experiments, the run-time increase is on the order of milliseconds, even for larger models with more than a million encrypted parameters as in the case of the audio recognition model. The notably lower run-time increase for the *Text* model is due to fewer than 1% of weights, more specifically 17 weights in total, needed to be encrypted to reduce the model to a random classifier. This underscores that carefully selecting high-impact weights presents significant potential for future optimization. Regarding memory overhead, we observe that it remains below a 10% increase in all cases. In principle, the approach requires only storing which weights are encrypted, resulting in linear scaling. This property makes the method suitable for memory-constrained environments. Additional memory is required for decryption and if a memory-based PUF is used as the key derivation mechanism.

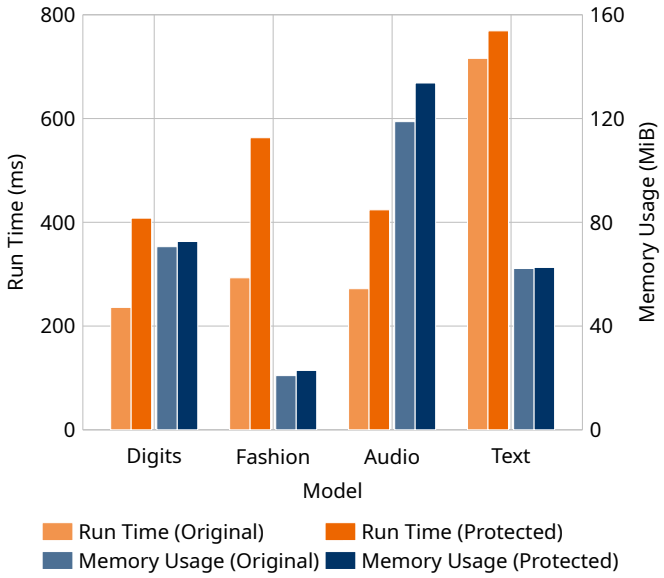


Figure 7.6: Results of evaluating LOCKI's proof of concept comparing original and protected models.

7.3.3 Discussion

Our evaluation confirms the feasibility and effectiveness of the proposed method: encrypting a portion of the model weights significantly degrades model precision and recall for unauthorized users. Only authorized parties with the correct key can restore the model's intended behavior. Decryption is performed just-in-time at run time, which minimizes the exposure window of decrypted weights and substantially reduces the attack surface. Regarding practical applicability, our experiments show that the run-time overhead introduced by the encryption layer stays within milliseconds while leaving room for further optimization. Furthermore, our method imposes memory overhead below 10% in all cases, which is particularly beneficial for deployments on resource-constrained platforms such as embedded industrial systems.

It is important to emphasize that our experiments do not aim to constitute a formal proof of cryptographic security. Instead, we demonstrate that the proposed mechanism operates reliably in practice and significantly reduces model accuracy for adversaries who do not possess the hardware-derived decryption key. Our protection targets adversaries who gain access to the model but cannot extract the correct key. Encrypting the model's weights mitigates a broad class of white-box attacks that rely on access to internal model details. In particular, it prevents model-stealing attacks, understood here as attempts to reconstruct model parameters or faithfully replicate model behavior through direct inspection of the model. It further hinders membership-inference attacks, which aim to determine whether specific data points were included in the training set by exploiting access to gradients, activations, or loss values [214–216]. Similarly, model-inversion attacks seek to reconstruct sensitive input features from detailed internal model responses [217–219]. While purely black-box inversion remains possible in principle, it is significantly less accurate and computationally more demanding [139]. Our approach therefore forces adversaries into weaker black-box access by protecting the internal model state.

Approach	Hardware Fingerprint	Binding Target		Design-/Build-Time Transformations	Safe by Design
		Type	Specific		
GLUEZILLA ANGLUERUS	Bit flips in DRAM	Code Data	Instructions Data symbols	Instr. substitution Data modification	no
MINOTAUROX SPHINXOR	Any PUF/TPM	Code Data	Transitions Constants	Bind transitions Boolean eval.	yes
LOCKI	Any PUF/TPM	Data	NN weights	Data encryption	no

Table 7.3: Overview of the approaches, including their building blocks and binding targets, and whether they are safe by design.

Despite these advantages, limitations remain. Our method does not protect against potential flaws in the underlying stream cipher or its implementation—such vulnerabilities could be discovered in the future and would compromise the entire scheme. Furthermore, certain attack strategies lie entirely outside the scope of our threat model. For instance, model theft through querying—e.g., by observing model outputs and training a surrogate—or retraining based on leaked or stolen training data remains a plausible threat. Attacks targeting the training process itself or threats to system availability, such as data poisoning attacks or adversarial denial of service (DoS), are also clearly not mitigated by our approach and continue to pose open challenges. Nonetheless, our protection mechanism is orthogonal to other defense strategies and can therefore be combined with them to counter these attacks.

7.4 Comparative Evaluation

In this section, we compare our proposed hardware–software binding approaches:

1. our *Rowhammer-based* approaches GLUEZILLA and ANGLUERUS (cf. Chapter 4) targeting scenario 1 (PC with DRAM),
2. the *PUF-based* techniques MINOTAUROX and SPHINXOR (cf. Chapter 5) targeting scenario 2 (MCU with SRAM),
3. and LOCKI, our protection mechanism for *NN models* (cf. Chapter 6) targeting scenario 3 (NN models).

Due to the different scenarios addressed (cf. Section 3.1), these approaches are not always directly comparable, and thus our analysis remains at an abstract level. However, approaches developed for the same scenario allow for direct comparison and are therefore grouped in this section's tables. We compare (i) the selected building blocks (cf. Section 3.4), (ii) security properties of the approaches, and (iii) the deployment effort and overheads of their implementations.

Table 7.3 summarizes the differences in hardware fingerprints, binding targets, and applied transformations. The Rowhammer-based approaches GLUEZILLA and ANGLUERUS leverage physical bit flips in DRAM, whereas the other protection mechanisms may rely on any suitable PUF or even TPM-based sealing. GLUEZILLA and ANGLUERUS target either instructions or data symbols using instruction substitution or data modification, respectively, but neither provides fail-safe guarantees. Thus, we must consider fail-safety when

Approach	Resistance to Copying	Resistance to Reverse Engineering			Stealthiness
		Static	Dynamic	Target Device	
GlueZilla	●●●	●●●	●●●	●●●	●●●
Angluerus	●●●	●●●	●●●	●●●	●●●
MinotaurOX	●●●	●●●	●●○	●●○	●○○
SphinxOR	●●●	●●●	●●●	●○○	●○○
Locki	●●●	●●●	●●○	●○○	●○○

Table 7.4: Comparison of approach security: ●○○ denotes minimum, ●●○ medium, and ●●● maximum resistance/stealthiness. Note that all classifications apply to the approach itself and not necessarily to its concrete implementation.

applying these methods to ensure the safety of the target program's execution. In contrast, MinotaurOX protects control-state transitions by deriving them from a PUF response, and SphinxOR protects constants by assigning the PUF response to Boolean variables and evaluating them to obtain the constant values. Both approaches ensure fail-safe operation by design, providing safe fallbacks in case of an incorrect PUF response. Locki encrypts weights of NN models using a key derived from a hardware fingerprint. Since NN models do not provide safety guarantees, we assume an additional safety system that checks the model output [189, 190]. Such a safety system may also be used in conjunction with GlueZilla and Angluerus to ensure safe deployment.

7.4.1 Security Comparison

Table 7.4 presents an abstract security analysis comparing our approaches. Copying of a program protected by any of our approaches is infeasible, since the program executes correctly only on its target device. All approaches exhibit strong resistance to static analysis, since the target program lacks information found only in the target device's hardware. Dynamic analysis on a cloned device does not reveal any secrets stored on the target device, but it can reduce the attacker's search space by eliminating options that are inconsistent with the observed execution behavior. Specifically, MinotaurOX narrows down the targets of control-state transitions by allowing only safe transitions and Locki provides hints about the encrypted weights, potentially supporting an adversary. In practice, however, the remaining options and thus the resulting search space remain substantial.

We assume that dynamic analysis of the program on the target device is not straightforward due to additional protections hindering direct access to the target device. We deem this a realistic assumption since we focus on industrial machines, where the target device is fully customizable by the machine's manufacturer. However, an adversary might still be able to dynamically analyze the target program on the device it is bound to. GlueZilla and Angluerus propose to apply and revert run-time transformations during execution such that the intended program is never fully present in memory. Furthermore, analyzing the target program likely disturbs the Rowhammer effect due to additional memory accesses and delays in execution. Even if just some run-time transformations fail, the program's intended behavior cannot be analyzed on the target device. MinotaurOX reveals only one intended control-state transition at a time, rendering analysis infeasible for larger

Approach	Status	HW Fingerprint	Platform	Impl. Stack
GLUEZILLA ANGLUERUS	Prototype	Rowhammer	x86, Linux	C++/Rust, LLVM C++, LLVM
MINOTAUROX SPHINXOR	PoC Prototype	SRAM PUF	Arm MCU	C
LOCKI	PoC	—	—	Python, TensorFlow

Table 7.5: Overview of approach implementations, including the chosen hardware fingerprint, target platform, and implementation stack.

programs. SPHINXOR reveals the protected data at run time, and the program potentially keeps them in untrusted memory. However, dynamic analysis on the target device might be infeasible due to its resource-constrained nature; cf. scenario 2 (MCU with SRAM). LOCKI decrypts weights just in time, leaving a short time window for an adversary—ideally, the decryption is performed in a TEE or similar. For all of our approaches, an adversary with full control over the target device can try to reconstruct the hardware fingerprint. For GLUEZILLA, ANGLUERUS and MINOTAUROX, the adversary would still need to simulate the hardware fingerprint on the cloned device. SPHINXOR and LOCKI pose much less of an obstacle to the adversary in this regard, since providing the extracted key-like hardware fingerprint at program startup is sufficient.

Rowhammer-based methods generally maintain higher stealthiness, as their modifications are tied to low-level physical effects difficult to detect without specialized hardware access. Due to a lack of telltale patterns in the protected code, e.g., specific interface accesses, an adversary might have difficulties figuring out that the code is protected. Furthermore, the run-time transformations can even be performed from another process, increasing stealthiness. In contrast, our PUF-based techniques rely on PUF responses, which can reveal the presence of protection mechanisms, potentially enabling an attacker to remove the protection.

7.4.2 Implementation Comparison

We compare the reference implementations of our approaches in Table 7.5. We implemented the Rowhammer-based prototypes based on LLVM for x86 systems with Linux. The C implementations of our PUF-based approaches target Arm MCUs and both use the same SRAM PUF. Our proof of concept for LOCKI is based on TensorFlow and implemented in Python using a simulated PUF, making it platform-agnostic.

Table 7.6 compares the deployment effort and the overheads introduced by our prototypes/proof of concepts during run time. Due to the heterogeneous nature of the target platforms, ranging from x86 PCs to Arm MCUs, a direct quantitative comparison across all five implementations is not feasible. Instead, Table 7.6 provides a qualitative comparison that reports deployment effort and overheads relative to the other approaches.

GLUEZILLA and ANGLUERUS require thorough analysis of the target device's DRAM to select suitable bit flips, which complicates deployments. MINOTAUROX and SPHINXOR use an SRAM PUF, which can be determined more easily but still requires analysis. LOCKI proposes sealing the secret key with a TPM, which would require minimal effort. Furthermore,

Implementation	Deployment Effort		Overhead	
	Fingerprinting	Transform.	Run Time	Size/Memory
GLUEZILLA	●●●	●●●	●●○	●●○
ANGLUERUS		●●○	●○○	●●○
MINOTAUROX	●●○	●●○	●●○	●○○
SPHINXOR		●○○	●○○	●●●
LOCKI	●○○	●○○	●●○	●○○

Table 7.6: Comparison of deployment effort and overheads: ●○○ denotes low, ●●○ medium, and ●●● high effort/overhead. Note that all classifications only apply to the specific implementation of an approach and not the approach itself.

the current prototype of GLUEZILLA requires selection of instructions, requiring disassembly of the target program; ANGLUERUS's prototype requires only a list of symbols to protect. MINOTAUROX requires an ASM specification of the target program and an analysis of safe control-state transitions. Especially SPHINXOR and LOCKI are easy to deploy, since only the data to be protected needs to be provided, the protection can then be applied automatically.

Our prototypes GLUEZILLA and ANGLUERUS showed moderate memory overheads due to padding of junction points on memory pages, while ANGLUERUS causes less run-time overhead compared to GLUEZILLA. The proof of concept for MINOTAUROX incurs moderate run-time overhead but minimal size overhead, while the SPHINXOR prototype achieves lower run time but comes with significantly increased code size due to the encoding of constants as Boolean expressions. The proof of concept for LOCKI shows that its run-time overhead strongly depends on the NN model to be protected, since the number of weights that must be encrypted to impact the model's performance varies, which can cause high relative overheads for smaller models. Note that the prototypes and even more so the proof of concepts are not optimized to reduce run-time and memory-usage overheads.

Overall, our Rowhammer-based approaches excel in stealth and pose a significant obstacle even for a reverse engineer controlling the target device. The PUF-based mechanisms provide fail-safety and are easier to deploy on resource-constrained devices, especially those without DRAM. LOCKI protects NN weights and can leverage a wide range of hardware fingerprints, depending on the target deployment platform. The choice between these strategies ultimately depends on the type of software to be protected, the target platform, and the required strength of the protection.

Imagination is more important than knowledge. For knowledge is limited, whereas imagination embraces the entire world.

— Albert Einstein

8 Conclusion

Contents

8.1 Summary 107
8.2 Future Work 108
8.3 Lessons Learned 109

This dissertation has explored the paradigm of hardware–software binding as a defense against unauthorized copying and reverse engineering of industrial software. By leveraging intrinsic hardware properties, this work shows how we can bind software to physical properties of a target device such that the software executes as intended only on that device, while exhibiting unintended behavior on cloned devices. This final chapter summarizes the findings of the five presented approaches realizing our conceptual model, outlines future directions for research in the field of hardware–software binding, and concludes with lessons learned.

8.1 Summary

This dissertation addresses industrial software piracy by establishing a novel protection paradigm: binding software behavior to intrinsic hardware properties. The core of this work is a conceptual protection model that transforms programs into an unintended form before deployment and restores their intended form at run time only on authorized hardware, identified by a hardware fingerprint. We demonstrated the feasibility of this conceptual model through five specific approaches targeting three distinct industrial scenarios:

- *Scenario 1 (PC with DRAM)*: GLUEZILLA leverages Rowhammer-induced bit flips to protect instructions such that their intended form can be restored only on the target device. ANGLUERUS flips bits in values of data symbols so that the intended values can only be restored on the target device.

- *Scenario 2 (MCU with SRAM)*: Developed for resource-constrained systems, MINOTAUROX uses SRAM PUF responses to determine control-state transitions, and SPHINXOR employs Boolean encoding to ensure that sensitive control parameters retain safe, albeit suboptimal, values if executed on cloned devices.
- *Scenario 3 (NN models)*: LOCKI protects neural network models by encrypting their weights and sealing the decryption key via a TPM or extracting it from a hardware fingerprint.

Our research confirms that it is possible to identify devices using intrinsic properties like Rowhammer-induced bit flips and SRAM startup values without the need for specialized hardware often not available on legacy machines. The security analyses of our approaches show a search space that exceeds practical capabilities of adversaries, likely making reverse engineering more costly than the independent redevelopment of the software. The reference implementations of our approaches show their practical feasibility.

While highly effective from a security standpoint, our evaluation also revealed several practical limitations. The Rowhammer-based approaches are inherently dependent on the presence of DRAM, excluding a range of computing devices. Additionally, selecting optimal junction points currently requires considerable developer effort to ensure that the unintended program behavior remains safe and inconspicuous. Environmental factors, particularly temperature variations, can influence bit flip reliability, requiring robust templating across operational ranges. Using SRAM PUFs or TPMs circumvents the dependence on DRAM but consequently eliminates the advantages of dynamic hardware fingerprinting. Dynamic techniques allow repeated identification of the hardware at run time, ideally immediately before a transformation takes place, and therefore provide stronger resistance to reverse engineering.

Ultimately, this dissertation advances software protection by binding programs and ML models to intrinsic hardware properties. In doing so, it establishes hardware-bound transformation as a general protection principle: programs are deployed in an unintended representation and restored at run time through device-specific physical properties. By combining hardware–software binding with software diversity, our protection model transcends traditional protection mechanisms that aim to deter either copying or reverse engineering. Hardware-dependent software behavior thus creates a substantial barrier for adversaries and provides an additional layer of defense for industrial software.

8.2 Future Work

Building on the foundations laid in this work, several promising directions for future research remain. Beyond extending individual mechanisms, these directions aim to advance our hardware–software binding approaches toward deployable, general-purpose protection systems.

Currently, selecting junction points requires manual effort to ensure safe operation of programs protected by GLUEZILLA and ANGLUERUS. Automatically identifying junction points would lower the deployment effort of our protection, e.g., assisted by AI, fuzzing, or verification. As industrial hardware eventually transitions to newer DRAM standards, Rowhammer-based hardware fingerprinting must be adapted to account for the refined

mitigation mechanisms of DDR5 DRAM, which have been demonstrated insufficient to prevent Rowhammer [78, 100]. A key research problem is to characterize which Rowhammer-induced bit flips remain stable across environmental changes and device aging. A solution deployable in industrial settings must also consider software updates and hardware replacement. Providing formal guarantees that hardware–software binding preserves safety properties would be essential to support certification.

While this dissertation focused on programs executed on CPUs, extending Rowhammer-based binding to GPUs would allow for protecting high-performance computing tasks and local ML inference pipelines. GPUs also use DRAM demonstrated to be vulnerable to Rowhammer [90].

Future iterations of LOCKI could explore the use of fine-grained pruning matrices that prune weights of an NN model at run time. Binding the pruning matrix to a target device has the potential to degrade the inference on a cloned device. For instance, one could prune the most important weights on a cloned device while keeping altered weights that would be pruned on the target device. To protect other types of machine learning models, a possibility is to adapt the Boolean encoding used by SPHINXOR, e.g., to protect the parameters of a support vector machine (SVM).

Integrating MINOTAUROX with run-time attestation would not only mitigate copying and reverse engineering but also detect tampering during execution. Specifically, the dispatcher function of MINOTAUROX could log control-state values and, if required, additional data. A verifier can then check these logs against ASM transition rules to confirm that the observed behavior conforms to the specification [50]. Furthermore, the PUF required by MINOTAUROX can also serve as a RoT for run-time attestation.

Another promising direction is the exploration of hybrid hardware fingerprints. Rather than relying on a single source of entropy, future systems could combine multiple intrinsic hardware properties and additionally leverage complementary platform features, such as TPMs or TEEs. Fusing independent fingerprint sources has the potential to improve robustness, entropy, and thus attack resistance.

8.3 Lessons Learned

Although Rowhammer has received significant attention in academic research and the media, its exploitation in practice remains difficult. To the best of our knowledge, there have been no reports of successful Rowhammer attacks in real-world deployments. Consequently, costly defensive measures against Rowhammer are often uneconomical, as the attack is unlikely to pose a realistic threat in most environments when simpler attack vectors are available. At the same time, our work shows that even subtle hardware effects can be carefully engineered into effective protection mechanisms, motivating the use of Rowhammer defensively rather than offensively.

Control software is frequently entirely unprotected and therefore vulnerable to copying, reverse engineering, and tampering. Even limited protection is thus preferable to none, particularly in industrial environments characterized by long device lifetimes and widespread legacy systems. When Rowhammer-based protection cannot be applied, alternative PUF-based mechanisms still raise the cost for an attacker compared to an unprotected system. In practice, reverse engineering is more prevalent and more systematic

than one might assume, especially in highly specialized industries. This reinforces the need to proactively defend valuable knowledge. However, designing such mechanisms reveals a fundamental tension between safety guarantees and strong hardware–software binding, as measures that enforce fail-safe behavior often weaken resistance against reverse engineering. Conversely, opting for alternative protection mechanisms may fail to present a substantial obstacle to a determined adversary.

Our attempts to reverse engineer AMD's DRAM mapping functions showed us that substantial insight into hardware behavior can be obtained using software-only techniques. Although we did not succeed in fully recovering the mapping functions due to a lack of electronic test instruments, our studies produced hypotheses that were later confirmed by subsequent work [78]. Nevertheless, modeling these hypotheses as Boolean relations between address bits inspired the use of Boolean expressions for the protection of sensitive data. This experience highlights how exploratory research often leads to outcomes that were not anticipated at the outset, uncovering new directions that emerged outside the original research objectives.

Looking forward, AI-assisted reverse engineering significantly lowers the cost of program analysis. Recent systems already enable AI to orchestrate reverse engineering tools such as disassemblers and decompilers, autonomously steering analyses and iterating over results with minimal human involvement [220]. This development reduces the required expertise and time to extract knowledge from software and lowers the barrier for large-scale and systematic reverse engineering. In this context, hardware–software binding provides a particularly strong line of defense by tying correct program behavior to physical properties that cannot be inferred or reconstructed through software analysis alone, including AI-based approaches. At the same time, ML models increasingly embody valuable knowledge. Protecting their key parameters therefore becomes as important as protecting classical software, especially in industrial environments, where direct extraction enables knowledge theft beyond what retraining alone would provide.

> *The best thing about a Boolean is even if you are wrong, you are only off by a bit.*
>
> — Anonymous

References

[1] *VDMA study industrial security and product piracy 2024*, VDMA, 2024. [Online]. Available: https://www.vdma.eu/documents/34570/4802302/VDMA+Study+Industrial+Security+and+Product+Piracy+2024.pdf.

[2] *VDMA study product piracy 2022*, VDMA, 2022. [Online]. Available: https://www.vdma.org/documents/34570/51629660/VDMA+Study+Product+Piracy+2022_final.pdf.

[3] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, 1990. DOI: 10.1109/52.43044.

[4] F. Nielson, H. Riis Nielson, and C. Hankin, *Principles of Program Analysis*. Springer, 1999. DOI: 10.1007/978-3-662-03811-6.

[5] E. Eilam, *Reversing, Secrets of Reverse Engineering*. Wiley, 2005, ISBN: 978-0-7645-7481-8.

[6] G. Canfora, M. Di Penta, and L. Cerulo, "Achievements and challenges in software reverse engineering," *Communications of the ACM*, vol. 54, no. 4, pp. 142–151, Apr. 2011. DOI: 10.1145/1924421.1924451.

[7] B. Dang, A. Gazet, and E. Bachaalany, *Practical Reverse Engineering, x86, x64, ARM, Windows® Kernel, Reversing Tools, and Obfuscation*. Wiley, Feb. 2014, ISBN: 978-1-118-78731-1.

[8] R. O'Neill, *Learning Linux Binary Analysis*. Packt, 2016, ISBN: 978-1-78216-710-5.

[9] D. Andriesse, *Practical Binary Analysis, Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly*. No Starch Press, 2018, ISBN: 978-1-59327-912-7.

[10] C. Eagle and K. Nance, *The Ghidra Book, The Definitive Guide*. No Starch Press, 2020, ISBN: 978-1-7185-0102-7.

[11] M. A. Markstedter, *Arm Assembly, Internals & Reverse Engineering*. Wiley, 2023, ISBN: 978-1-119-74530-3.

[12] S. Domas and C. Domas, *x86 Software Reverse-Engineering, Cracking, and Counter-Measures*. Wiley, 2024, ISBN: 978-1-394-19988-4.

[13] R. Anderson, *Security Engineering, A Guide to Building Dependable Distributed Systems*, 3rd ed. Wiley, 2020, ISBN: 978-1-119-64278-7.

[14] J. van Woudenberg and C. O'Flynn, *The Hardware Hacking Handbook, Breaking Embedded Security with Hardware Attacks*. No Starch Press, 2022, ISBN: 978-1-59327-874-8.

[15] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *25th USENIX Security Symposium (USENIX Security 16)*, USENIX Association, Aug. 2016, pp. 583–600, ISBN: 978-1-931971-32-4. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/andriesse.

[16] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Computing Surveys*, vol. 44, no. 2, Mar. 2008. DOI: 10.1145/2089125.2089126.

[17] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach, "Dynamic malware analysis in the modern era—a state of the art survey," *ACM Computing Surveys*, vol. 52, no. 5, Sep. 2019. DOI: 10.1145/3329786.

[18] U. Piazzalunga, P. Salvaneschi, F. Balducci, P. Jacomuzzi, and C. Moroncelli, "Security strength measurement for dongle-protected software," *IEEE Security & Privacy Magazine*, vol. 5, no. 6, pp. 32–40, Nov. 2007. DOI: 10.1109/MSP.2007.176.

[19] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, Tech. Rep. 148, Jan. 1997.

[20] C. Collberg, J. Nagra, and F.-Y. Wang, "Surreptitious software: Models from biology and history," in *Computer Network Security*, V. Gorodetsky, I. Kotenko, and V. A. Skormin, Eds., Springer, 2007, pp. 1–21. DOI: 10.1007/978-3-540-73986-9_1.

[21] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?" *ACM Computing Surveys*, vol. 49, no. 1, 2016. DOI: 10.1145/2886012.

[22] D. Beste, G. Menguy, H. Hajipour, M. Fritz, A. E. Cinà, S. Bardin, T. Holz, T. Eisenhofer, and L. Schönherr, "Exploring the potential of LLMs for code deobfuscation," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, M. Egele, V. Moonsamy, D. Gruss, and M. Carminati, Eds., Springer, 2025, pp. 267–286. DOI: 10.1007/978-3-031-97620-9_15.

[23] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '98, ACM, 1998, pp. 184–196. DOI: 10.1145/268946.268962.

[24] C. Wang, J. Davidson, J. Hill, and J. Knight, "Protection of software-based survivability mechanisms," in *2001 International Conference on Dependable Systems and Networks*, 2001, pp. 193–202. DOI: 10.1109/DSN.2001.941405.

[25] N. Mavrogiannopoulos, N. Kisserli, and B. Preneel, "A taxonomy of self-modifying code for obfuscation," *Computers & Security*, vol. 30, no. 8, pp. 679–691, Nov. 2011. DOI: 10.1016/J.COSE.2011.08.007.

[26] B. Anckaert, M. Jakubowski, and R. Venkatesan, "Proteus: Virtualization for diversified tamper-resistance," in *Proceedings of the ACM Workshop on Digital Rights Management*, ser. DRM '06, ACM, 2006, pp. 47–58. DOI: 10.1145/1179509.1179521.

[27] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07, ACM, 2007, pp. 552–561. DOI: 10.1145/1315245.1315313.

[28] G. De Pasquale, F. Nakanishi, D. Ferla, and L. Cavallaro, "ROPfuscator: Robust obfuscation with ROP," in *2023 IEEE Security and Privacy Workshops (SPW)*, 2023, pp. 1–10. DOI: 10.1109/SPW59333.2023.00026.

[29] P. Borrello, E. Coppa, D. C. D'Elia, and C. Demetrescu, "The ROP needle: Hiding trigger-based injection vectors via code reuse," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, ser. SAC '19, ACM, Apr. 2019, pp. 1962–1970. DOI: 10.1145/3297280.3297472.

[30] D. Mu, J. Guo, W. Ding, Z. Wang, B. Mao, and L. Shi, "ROPOB: Obfuscating binary code via return oriented programming," in *Security and Privacy in Communication Networks*, X. Lin, A. Ghorbani, K. Ren, S. Zhu, and A. Zhang, Eds., Springer, 2018, pp. 721–737. DOI: 10.1007/978-3-319-78813-5_38.

[31] K. Lu, S. Xiong, and D. Gao, "RopSteg: Program steganography with return oriented programming," in *Proceedings of the 4th ACM conference on Data and application security and privacy*, ser. CODASPY '14, ACM, Mar. 2014, pp. 265–272. DOI: 10.1145/2557547.2557572.

[32] T. Dullien, "Weird machines, exploitability, and provable unexploitability," *IEEE Transactions on Emerging Topics in Computing*, vol. 8, no. 2, pp. 391–403, 2020. DOI: 10.1109/TETC.2017.2785299.

[33] D. Evtyushkin, T. Benjamin, J. Elwell, J. A. Eitel, A. Sapello, and A. Ghosh, "Computing with time: Microarchitectural weird machines," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21, ACM, Apr. 2021, pp. 758–772. DOI: 10.1145/3445814.3446729.

[34] J. Wampler, I. Martiny, and E. Wustrow, "ExSpectre: Hiding malware in speculative execution," in *Network and Distributed System Security (NDSS) Symposium 2019*, Curran Associates, Inc., 2019. DOI: 10.14722/NDSS.2019.23409.

[35] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled detection of speculative information flows," in *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, May 2020, pp. 1–19. DOI: 10.1109/SP40000.2020.00011.

[36] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, "Oo7: Low-overhead defense against Spectre attacks via program analysis," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2504–2519, 2021. DOI: 10.1109/TSE.2019.2953709.

[37] R. Brotzman, D. Zhang, M. T. Kandemir, and G. Tan, "SpecSafe: Detecting cache side channels in a speculative world," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, Oct. 2021. DOI: 10.1145/3485506.

[38] M. N. Gagnon, S. Taylor, and A. K. Ghosh, "Software protection through anti-debugging," *IEEE Security & Privacy Magazine*, vol. 5, no. 3, pp. 82–84, May 2007. DOI: 10.1109/MSP.2007.71.

[39] "Apriorit code protection for Linux: Scheme, main components," Apriorit R&D. (2013), [Online]. Available: https://www.apriorit.com/wp-content/uploads/2023/03/Apriorit_SDK_Linux_Code_Protection.pdf (visited on 01/12/2026).

[40] *ptrace(2): Process trace, Linux man page*, Free Software Foundation. [Online]. Available: https://linux.die.net/man/2/ptrace (visited on 01/12/2026).

[41] M. Ahmadvand, A. Pretschner, and F. Kelbert, "Chapter eight - a taxonomy of software integrity protection techniques," in ser. Advances in Computers, A. M. Memon, Ed., vol. 112, Elsevier, 2019, pp. 413–486. DOI: 10.1016/BS.ADCOM.2017.12.007.

[42] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley, 2009, ISBN: 978-0-321-54522-0.

[43] D. Aucsmith, "Tamper resistant software: An implementation," in *Information Hiding*, R. Anderson, Ed., Springer, 1996, pp. 317–333. DOI: 10.1007/3-540-61996-8_49.

[44] B. Horne, L. Matheson, C. Sheehan, and R. E. Tarjan, "Dynamic self-checking techniques for improved tamper resistance," in *Security and Privacy in Digital Rights Management*, T. Sander, Ed., Springer, 2002, pp. 141–159. DOI: 10.1007/3-540-47870-1_9.

[45] H. Chang and M. J. Atallah, "Protecting software code by guards," in *Security and Privacy in Digital Rights Management*, T. Sander, Ed., Springer, 2002, pp. 160–175. DOI: 10.1007/3-540-47870-1_10.

[46] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05, ACM, 2005, pp. 340–353. DOI: 10.1145/1102120.1102165.

[47] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys*, vol. 50, no. 1, Apr. 2017. DOI: 10.1145/3054924.

[48] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla, "SWATT: Software-based attestation for embedded devices," in *2004 IEEE Symposium on Security and Privacy*, IEEE, 2004, pp. 272–282. DOI: 10.1109/SECPRI.2004.1301329.

[49] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-FLAT: Control-flow attestation for embedded systems software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, ACM, 2016, pp. 743–754. DOI: 10.1145/2976749.2978358.

[50] B. Fischer, D. Dorfmeister, F. Ferrarotti, S. Rass, and R. Mayrhofer, "ASM-based behavioral attestation: Bridging formal specification and runtime conformance checking of security properties," to appear in *Proceedings of the 24th ACM/IEEE International Symposium on Formal Methods and Models for System Design (MEMOCODE 2026)*.

[51] F. B. Cohen, "Operating system protection through program evolution," *Computers & Security*, vol. 12, no. 6, pp. 565–584, Oct. 1993. DOI: 10.1016/0167-4048(93)90054-9.

[52] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *2014 IEEE Symposium on Security and Privacy*, IEEE, May 2014, pp. 276–291. DOI: 10.1109/SP.2014.25.

[53] A. Avizienis and L. Chen, "On the implementation of N-version programming for software fault tolerance during execution," in *Proceedings of the International Computer Software and Applications Conference, COMPSAC '77*, 1977, pp. 149–155.

[54] F. Berlakovich and S. Brunthaler, "R2C: AOCR-resilient diversity with reactive and reflective camouflage," in *Proceedings of the Eighteenth European Conference on Computer Systems*, ser. EuroSys '23, ACM, 2023, pp. 488–504. DOI: 10.1145/3552326.3587439.

[55] M. Franz, "E unibus pluram: Massive-scale software diversity as a defense mechanism," in *Proceedings of the 2010 workshop on New security paradigms - NSPW '10*, ACM Press, 2010, p. 7. DOI: 10.1145/1900546.1900550.

[56] A. Homescu, T. Jackson, S. Crane, S. Brunthaler, P. Larsen, and M. Franz, "Large-scale automated software diversity—program evolution redux," *IEEE Transactions on Dependable and Secure Computing*, vol. 14, no. 2, pp. 158–171, 2017. DOI: 10.1109/TDSC.2015.2433252.

[57] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, ser. CCS '03, ACM, 2003, pp. 272–280. DOI: 10.1145/948109.948146.

[58] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ser. CCS '04, ACM, 2004, pp. 298–307. DOI: 10.1145/1030083.1030124.

[59] E. Simpson and P. Schaumont, "Offline hardware/software authentication for reconfigurable platforms," in *Cryptographic Hardware and Embedded Systems - CHES 2006*, L. Goubin and M. Matsui, Eds., Springer, 2006, pp. 311–323. DOI: 10.1007/11894063_25.

[60] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls, "FPGA intrinsic PUFs and their use for IP protection," in *Cryptographic Hardware and Embedded Systems - CHES 2007*, P. Paillier and I. Verbauwhede, Eds., Springer, 2007, pp. 63–80. DOI: 10.1007/978-3-540-74735-2_5.

[61] S. Kleber, F. Unterstein, M. Hiller, F. Slomka, M. Matousek, F. Kargl, and C. Bösch, "Secure code execution: A generic PUF-driven system architecture," in *Information Security*, L. Chen, M. Manulis, and S. Schneider, Eds., Springer, 2018, pp. 25–46. DOI: 10.1007/978-3-319-99136-8_2.

[62] A. Maiti and P. Schaumont, "Improved ring oscillator PUF: An FPGA-friendly secure primitive," *Journal of Cryptology*, vol. 24, no. 2, pp. 375–397, Apr. 1, 2011. DOI: 10.1007/S00145-010-9088-4.

[63] J. Zhang, B. Qi, Z. Qin, and G. Qu, "HCIC: Hardware-assisted control-flow integrity checking," *IEEE Internet of Things Journal*, vol. 6, no. 1, pp. 458–471, 2019. DOI: 10.1109/JIOT.2018.2866164.

[64] J. Zhang, Y. Lin, Y. Lyu, and G. Qu, "A PUF-FSM binding scheme for FPGA IP protection and pay-per-device licensing," *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 6, pp. 1137–1150, 2015. DOI: 10.1109/TIFS.2015.2400413.

[65] L. Bossuet and B. Colombier, "Comments on 'A PUF-FSM binding scheme for FPGA IP protection and pay-per-device licensing'," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 11, pp. 2624–2625, 2016. DOI: 10.1109/TIFS.2016.2553454.

[66] J. Zhang and G. Qu, "Rebuttal to 'Comments on "A PUF-FSM binding scheme for FPGA IP protection and pay-per-device licensing"'," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 11, pp. 2626–2627, 2016. DOI: 10.1109/TIFS.2016.2553443.

[67] W. Xiong, A. Schaller, S. Katzenbeisser, and J. Szefer, "Software protection using dynamic PUFs," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 2053–2068, 2019. DOI: 10.1109/TIFS.2019.2955788.

[68] F. Kohnhäuser, A. Schaller, and S. Katzenbeisser, "PUF-based software protection for low-end embedded devices," in *Trust and Trustworthy Computing*, M. Conti, M. Schunter, and I. Askoxylakis, Eds., Springer, 2015, pp. 3–21. DOI: 10.1007/978-3-319-22846-4_1.

[69] R. Mechelinck and S. Volckaert, "BROL: Cache-only execution for software protection," in *Proceedings of the 2nd Microarchitecture Security Conference*, Feb. 2026. DOI: 10.46586/UASC.2026.004.

[70] F. Frank, M. Schmid, F. Klement, P. Palani, A. Weber, E. B. Kavun, W. Xiong, T. Arul, and S. Katzenbeisser, "Secure data-binding in FPGA-based hardware architectures utilizing PUFs," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '24, ACM, 2024, pp. 423–435. DOI: 10.1145/3634737.3656996.

[71] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2007, ISBN: 978-0-12-379751-3.

[72] *JEDEC standard DDR4 SDRAM JESD79-4D*, JEDEC Solid State Technology Association, Jul. 2021. [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd79-4a.

[73] *JEDEC standard DDR5 SDRAM JESD79-5C.01*, JEDEC Solid State Technology Association, Jul. 2024. [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd79-5c01.

[74] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-CPU attacks," in *25th USENIX Security Symposium (USENIX Security 16)*, USENIX Association, Aug. 2016, pp. 565–581, ISBN: 978-1-931971-32-4. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl.

[75] P. Frigo, E. Vannacc, H. Hassan, V. v. der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "TRRespass: Exploiting the many sides of target row refresh," in *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, May 2020, pp. 747–762. DOI: 10.1109/SP40000.2020.00090.

[76] M. Wang, Z. Zhang, Y. Cheng, and S. Nepal, "DRAMDig: A knowledge-assisted tool to uncover DRAM address mapping," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218599.

[77] M. Wurzer, "DRAMA in embedded systems - determining the address mapping function," M.S. thesis, University of Applied Sciences Upper Austria, Hagenberg, Sep. 2022. [Online]. Available: https://permalink.obvsg.at/fho/AC16690818.

[78] P. Jattke, M. Wipfli, F. Solt, M. Marazzi, M. Bölcskei, and K. Razavi, "ZenHammer: Rowhammer attacks on AMD Zen-based platforms," in *33rd USENIX Security Symposium (USENIX Security 24)*, USENIX Association, Aug. 2024, pp. 1615–1633, ISBN: 978-1-939133-44-1. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/jattke.

[79] *HugeTLB pages, The Linux kernel documentation*, The kernel development community. [Online]. Available: https://www.kernel.org/doc/html/v6.18/admin-guide/mm/hugetlbpage.html (visited on 01/12/2026).

[80] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014. DOI: 10.1145/2678373.2665726.

[81] M. Seaborn and T. Dullien. "Exploiting the DRAM rowhammer bug to gain kernel privileges." (Mar. 2015), [Online]. Available: https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html (visited on 01/12/2026).

[82] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in JavaScript," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds., Springer, 2016, pp. 300–321. DOI: 10.1007/978-3-319-40667-1_15.

[83] O. Mutlu, A. Olgun, and A. G. Yağlıkcı, "Fundamentally understanding and solving RowHammer," in *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, ser. ASPDAC '23, ACM, Jan. 2023, pp. 461–468. DOI: 10.1145/3566097.3568350.

[84] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip Feng Shui: Hammering a needle in the software stack," in *25th USENIX Security Symposium (USENIX Security 16)*, USENIX Association, Aug. 2016, pp. 1–18, ISBN: 978-1-931971-32-4. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/razavi.

[85] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic rowhammer attacks on mobile platforms," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, ACM, Oct. 2016, pp. 1675–1689. DOI: 10.1145/2976749.2978406.

[86] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another flip in the wall of Rowhammer defenses," in *2018 IEEE Symposium on Security and Privacy (SP)*, IEEE, May 2018, pp. 245–261. DOI: 10.1109/SP.2018.00031.

[87] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup est machina: Memory deduplication as an advanced exploitation vector," in *2016 IEEE Symposium on Security and Privacy (SP)*, IEEE, May 2016, pp. 987–1004. DOI: 10.1109/SP.2016.63.

[88] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand pwning unit: Accelerating microarchitectural attacks with the GPU," in *2018 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2018, pp. 195–210. DOI: 10.1109/SP.2018.00022.

[89] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "RAMBleed: Reading bits in memory without accessing them," in *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, May 2020, pp. 695–711. DOI: 10.1109/SP40000.2020.00020.

[90] C. S. Lin, J. Qu, and G. Saileshwar, "GPUHammer: Rowhammer attacks on GPU memories are practical," in *34th USENIX Security Symposium (USENIX Security 25)*, USENIX Association, Aug. 2025, pp. 5719–5738, ISBN: 978-1-939133-52-6. [Online]. Available: https://www.usenix.org/conference/usenixsecurity25/presentation/lin-shaopeng.

[91] O. Mutlu, "The RowHammer problem and other issues we may face as memory becomes denser," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, IEEE, Mar. 2017, pp. 1116–1121. DOI: 10.23919/DATE.2017.7927156.

[92] J. S. Kim, M. Patel, A. G. Yağlıkçı, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting RowHammer: An experimental analysis of modern DRAM devices and mitigation techniques," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, May 2020, pp. 638–651. DOI: 10.1109/ISCA45697.2020.00059.

[93] L. Orosa, A. G. Yaglikçi, H. Luo, A. Olgun, J. Park, H. Hassan, M. Patel, J. S. Kim, and O. Mutlu, "A deeper look into RowHammer's sensitivities: Experimental analysis of real DRAM chips and implications on future attacks and defenses," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21, ACM, Oct. 2021, pp. 1182–1197. DOI: 10.1145/3466752.3480069.

[94] H. Hassan, Y. C. Tugrul, J. S. Kim, V. van der Veen, K. Razavi, and O. Mutlu, "Uncovering in-DRAM RowHammer protection mechanisms: A new methodology, custom RowHammer patterns, and implications," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21, ACM, 2021, pp. 1198–1213. DOI: 10.1145/3466752.3480110.

[95] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "CAn't Touch This: Software-only mitigation against rowhammer attacks targeting kernel memory," in *26th USENIX Security Symposium (USENIX Security 17)*, USENIX Association, Aug. 2017, pp. 117–130, ISBN: 978-1-931971-40-9. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/brasser.

[96] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "ANVIL: Software-based protection against next-generation rowhammer attacks," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16, ACM, 2016, pp. 743–755. DOI: 10.1145/2872362.2872390.

[97] M. Wiesinger, D. Dorfmeister, and S. Brunthaler, "MAD: Memory allocation meets software diversity," in *First Workshop on DRAM Security (DRAMSec)*, Jun. 2021. [Online]. Available: https://dramsec.ethz.ch/papers/mad.pdf.

[98] P. Jattke, V. Van Der Veen, P. Frigo, S. Gunter, and K. Razavi, "Blacksmith: Scalable rowhammering in the frequency domain," in *2022 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2022, pp. 716–734. DOI: 10.1109/SP46214.2022.9833772.

[99] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting correcting codes: On the effectiveness of ECC memory against Rowhammer attacks," in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, May 2019, pp. 55–71. DOI: 10.1109/SP.2019.00089.

[100] D. Meyer, P. Jattke, M. Marazzi, S. Qazi, D. Moghimi, and K. Razavi, "Phoenix: Rowhammer attacks on DDR5 with self-correcting synchronization," in *2026 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2026. [Online]. Available: https://comsec-files.ethz.ch/papers/phoenix_sp26.pdf.

[101] M. Heckel, N. Sayadi, J. Juffinger, C. Fiedler, D. Gruss, and F. Adamsky, "FlippyRAM: A large-scale study of Rowhammer prevalence," in *Network and Distributed System Security (NDSS) Symposium 2026*, 2026. [Online]. Available: https://flippyr.am/FlippyRAM.pdf.

[102] T. McGrath, I. E. Bagci, Z. M. Wang, U. Roedig, and R. J. Young, "A PUF taxonomy," *Applied Physics Reviews*, vol. 6, no. 1, Feb. 2019. DOI: 10.1063/1.5079407.

[103] C. Keller, F. Gürkaynak, H. Kaeslin, and N. Felber, "Dynamic memory-based physically unclonable function for the generation of unique identifiers and true random numbers," in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, Jun. 2014, pp. 2740–2743. DOI: 10.1109/ISCAS.2014.6865740.

[104] S. Sutar, A. Raha, and V. Raghunathan, "D-PUF: An intrinsically reconfigurable DRAM PUF for device authentication in embedded systems," in *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*, 2016, pp. 1–10. DOI: 10.1145/2968455.2968519.

[105] W. Xiong, A. Schaller, N. A. Anagnostopoulos, M. U. Saleem, S. Gabmeyer, S. Katzenbeisser, and J. Szefer, "Run-time accessible DRAM PUFs in commodity devices," in *Cryptographic Hardware and Embedded Systems – CHES 2016*, B. Gierlichs and A. Y. Poschmann, Eds., Springer, 2016, pp. 432–453. DOI: 10.1007/978-3-662-53140-2_21.

[106] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation - tools for software protection," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 735–746, 2002. DOI: 10.1109/TSE.2002.1027797.

[107] *Trusted platform module (TPM) summary*, Trusted Computing Group, Apr. 2008. [Online]. Available: https://trustedcomputinggroup.org/resource/trusted-platform-module-tpm-summary/ (visited on 01/12/2026).

[108] D. N. Jha, G. Lenton, J. Asker, D. Blundell, and D. Wallom, "Trusted platform module-based privacy in the public cloud: Challenges and future perspective," *IT Professional*, vol. 24, no. 3, pp. 81–87, May 2022. DOI: 10.1109/MITP.2022.3147968.

[109] J. Winter and K. Dietrich, "A hijacker's guide to the LPC bus," in *Public Key Infrastructures, Services and Applications*, S. Petkova-Nikova, A. Pashalidis, and G. Pernul, Eds., Springer, 2012, pp. 176–193. DOI: 10.1007/978-3-642-29804-2_12.

[110] J. Winter and K. Dietrich, "A hijacker's guide to communication interfaces of the trusted platform module," *Computers & Mathematics with Applications*, vol. 65, no. 5, pp. 748–761, 2013, EUROPKI-2011. DOI: 10.1016/J.CAMWA.2012.06.018.

[111] J. Boone, *TPM genie: Interposer attacks against the trusted platform module serial bus*, Mar. 2018. [Online]. Available: https://github.com/nccgroup/TPMGenie/blob/master/docs/NCC_Group_Jeremy_Boone_TPM_Genie_Whitepaper.pdf.

[112] U. Kühn, K. Kursawe, S. Lucks, A.-R. Sadeghi, and C. Stüble, "Secure data management in trusted computing," in *Cryptographic Hardware and Embedded Systems – CHES 2005*, J. R. Rao and B. Sunar, Eds., Springer, 2005, pp. 324–338. DOI: 10.1007/11545262_24.

[113] V. Costan and S. Devadas, *Intel SGX explained*, Cryptology ePrint Archive, Paper 2016/086, 2016. [Online]. Available: https://eprint.iacr.org/2016/086.

[114] *Arm security technology, Building a secure system using TrustZone technology*, ARM Limited, 2009. [Online]. Available: https://documentation-service.arm.com/static/5f212796500e883ab8e74531 (visited on 01/12/2026).

[115] X. Zhang, J. Wang, Y. Cheng, Q. Li, K. Sun, Y. Zheng, N. Zhang, and X. Li, "Interface-based side channel in TEE-assisted networked services," *IEEE/ACM Transactions on Networking*, vol. 32, no. 1, pp. 613–626, 2024. DOI: 10.1109/TNET.2023.3294019.

[116] A. Seto, O. K. Duran, S. Amer, J. Chuang, S. van Schaik, D. Genkin, and C. Garman, "WireTap: Breaking server SGX via DRAM bus interposition," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '25, ACM, 2025, pp. 708–722. DOI: 10.1145/3719027.3765204.

[117] M. H. Rahman, *A comprehensive survey on hardware-software co-protection against invasive, non-invasive and interactive security threats*, Cryptology ePrint Archive, Paper 2024/1892, 2024. [Online]. Available: https://eprint.iacr.org/2024/1892.

[118] B. Gassend, D. Clarke, M. Van Dijk, and S. Devadas, "Silicon physical random functions," in *Proceedings of the 9th ACM conference on Computer and communications security*, ser. CCS02, ACM, Nov. 2002, pp. 148–160. DOI: 10.1145/586110.586132.

[119] C. Herder, M.-D. Yu, F. Koushanfar, and S. Devadas, "Physical unclonable functions and applications: A tutorial," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1126–1141, Aug. 2014. DOI: 10.1109/JPROC.2014.2320516.

[120] D. E. Holcomb, W. P. Burleson, and K. Fu, "Power-up SRAM state as an identifying fingerprint and source of true random numbers," *IEEE Transactions on Computers*, vol. 58, no. 9, pp. 1198–1210, 2009. DOI: 10.1109/TC.2008.212.

[121] A. Schaller, W. Xiong, N. A. Anagnostopoulos, M. U. Saleem, S. Gabmeyer, S. Katzenbeisser, and J. Szefer, "Intrinsic Rowhammer PUFs: Leveraging the Rowhammer effect for improved security," in *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, IEEE, May 2017, pp. 1–7. DOI: 10.1109/HST.2017.7951729.

[122] J. S. Kim, M. Patel, H. Hassan, and O. Mutlu, "The DRAM latency PUF: Quickly evaluating physical unclonable functions by exploiting the latency-reliability tradeoff in modern commodity DRAM devices," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 194–207. DOI: 10.1109/HPCA.2018.00026.

[123] P. Prabhu, A. Akel, L. M. Grupp, W.-K. S. Yu, G. E. Suh, E. Kan, and S. Swanson, "Extracting device fingerprints from flash memory by exploiting physical variations," in *Trust and Trustworthy Computing*, J. M. McCune, B. Balacheff, A. Perrig, A.-R. Sadeghi, A. Sasse, and Y. Beres, Eds., Springer, 2011, pp. 188–201. DOI: 10.1007/978-3-642-21599-5_14.

[124] S. Larimian, M. R. Mahmoodi, and D. B. Strukov, "Lightweight integrated design of PUF and TRNG security primitives based on eFlash memory in 55-nm CMOS," *IEEE Transactions on Electron Devices*, vol. 67, no. 4, pp. 1586–1592, 2020. DOI: 10.1109/TED.2020.2976632.

[125] G. E. Suh and S. Devadas, "Physical unclonable functions for device authentication and secret key generation, The 44th annual design automation conference 2007," in *Proceedings of the 44th annual conference on Design automation - DAC '07*, ser. DAC '07, ACM, 2007, pp. 9–14. DOI: 10.1145/1278480.1278484.

[126] S. S. Kumar, J. Guajardo, R. Maes, G.-J. Schrijen, and P. Tuyls, "Extended abstract: The butterfly PUF protecting IP on every FPGA," in *2008 IEEE International Workshop on Hardware-Oriented Security and Trust*, 2008, pp. 67–70. DOI: 10.1109/HST.2008.4559053.

[127] B. Škorić, P. Tuyls, and W. Ophey, "Robust key extraction from physical uncloneable functions," in *International Conference on Applied Cryptography and Network Security*, Springer, 2005, pp. 407–422. DOI: 10.1007/11496137_28.

[128] D. Lim, J. W. Lee, B. Gassend, G. E. Suh, M. Van Dijk, and S. Devadas, "Extracting secret keys from integrated circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 10, pp. 1200–1205, 2005. DOI: 10.1109/TVLSI.2005.859470.

[129] S. Samanta, B. Ray, and A. Milenkovic, "Analysis of temperature effect on SRAM PUF for low cost applications," in *2025 IEEE Physical Assurance and Inspection of Electronics (PAINE)*, 2025, pp. 1–7. DOI: 10.1109/PAINE66113.2025.11320191.

[130] N. A. Anagnostopoulos, S. Katzenbeisser, J. Chandy, and F. Tehranipoor, "An overview of DRAM-based security primitives," *Cryptography*, vol. 2, no. 2, 2018. DOI: 10.3390/CRYPTOGRAPHY2020007.

[131] N. A. Anagnostopoulos, T. Arul, Y. Fan, C. Hatzfeld, A. Schaller, W. Xiong, M. Jain, M. U. Saleem, J. Lotichius, S. Gabmeyer, J. Szefer, and S. Katzenbeisser, "Intrinsic run-time Row Hammer PUFs: Leveraging the Row Hammer effect for run-time cryptography and improved security," *Cryptography*, vol. 2, no. 3, pp. 1–45, 2018. DOI: 10.3390/CRYPTOGRAPHY2030013.

[132] D. Li, D. Liu, Y. Ren, Z. Wang, Y. Sun, Z. Guan, Q. Wu, and J. Liu, "FPHammer: A device identification framework based on DRAM fingerprinting," in *2023 IEEE 22nd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2023, pp. 1031–1040. DOI: 10.1109/TRUSTCOM60117.2023.00144.

[133] H. Venugopalan, K. Goswami, Z. Abi Din, J. Lowe-Power, S. T. King, and Z. Shafiq, "FP-Rowhammer: DRAM-based device fingerprinting," in *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '25, ACM, 2025, pp. 1141–1157. DOI: 10.1145/3708821.3733880.

[134] B. Fischer, D. Dorfmeister, H. Lampesberger, and E. Hermann, "Leveraging Rowhammer for physically unique and non-tamperable device identification," *Procedia Computer Science*, vol. 253, pp. 1154–1164, Feb. 2025, 6th International Conference on Industry 4.0 and Smart Manufacturing. DOI: 10.1016/J.PROCS.2025.01.177.

[135] T. Ziebermayr, "Protecting intellectual property rights of industrial software," *Procedia Computer Science*, vol. 180, pp. 862–866, 2021, Proceedings of the 2nd International Conference on Industry 4.0 and Smart Manufacturing (ISM 2020). DOI: 10.1016/J.PROCS.2021.01.336.

[136] J. Obermaier and S. Tatschner, "Shedding too much light on a microcontroller's firmware protection," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, USENIX Association, Aug. 2017. [Online]. Available: https://www.usenix.org/conference/woot17/workshop-program/presentation/obermaier.

[137] C. Crossley, *Software Supply Chain Security, Securing the End-to-End Supply Chain for Software, Firmware, and Hardware*. O'Reilly, 2024, ISBN: 978-1-098-13370-2.

[138] P. Ackerman, *Industrial Cybersecurity, Efficiently secure critical infrastructure systems*. Packt, 2017, ISBN: 978-1-78839-515-1.

[139] K. Zhao, L. Li, K. Ding, N. Z. Gong, Y. Zhao, and Y. Dong, *A systematic survey of model extraction attacks and defenses: State-of-the-art and perspectives*, 2025. DOI: 10.48550/ARXIV.2508.15031.

[140] Z. Wang, L. Shen, T. Liu, T. Duan, Y. Zhu, D. Zhan, D. Doermann, and M. Gao, "Defending against data-free model extraction by distributionally robust defensive training," in *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36, Curran Associates, Inc., 2023, pp. 624–637. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/file/0207c9ea9faf66c6e892c3fa3c167b75-Paper-Conference.pdf.

[141] H. Luo, A. Olgun, A. G. Yağlıkçı, Y. C. Tuğrul, S. Rhyner, M. B. Cavlak, J. Lindegger, M. Sadrosadati, and O. Mutlu, "RowPress: Amplifying read disturbance in modern DRAM chips," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23, ACM, 2023. DOI: 10.1145/3579371.3589063.

[142] B. Fischer, D. Dorfmeister, F. Ferrarotti, M. Penz, M. Kargl, M. Zeinzinger, and F. Eibensteiner, "Software-hardware binding for protection of sensitive data in embedded software," in *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing*. ACM, May 2025, pp. 570–577. DOI: 10.1145/3672608.3707855.

[143] M. Penz, M. Zeinzinger, M. Kargl, F. Eibensteiner, P. Petz, and J. Langer, "SRAM PUFs for device authentication on resource-constrained systems," in *2025 9th International Conference on Cryptography, Security and Privacy (CSP)*, 2025, pp. 169–176. DOI: 10.1109/CSP66295.2025.00035.

[144] H. Sochor, B. Fischer, D. Dorfmeister, and F. Ferrarotti, "Deriving cryptographic keys from memory-intrinsic entropy."

[145] R. Mechelinck, D. Dorfmeister, B. Fischer, S. Volckaert, and S. Brunthaler, "GlueZilla: Efficient and scalable software to hardware binding using rowhammer," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, F. Maggi, M. Egele, M. Payer, and M. Carminati, Eds., Springer, Jul. 2024, pp. 416–438. DOI: 10.1007/978-3-031-64171-8_22.

[146] R. Mechelinck, D. Dorfmeister, B. Fischer, S. Volckaert, and S. Brunthaler, "DEPS: Leveraging hardware faults for binding software to hardware," in *22. Kolloquium Programmiersprachen und Grundlagen der Programmierung*, T. Noll and I. Fesefeldt, Eds., 2023, p. 8. DOI: 10.18154/RWTH-2023-10034.

[147] R. Mechelinck, D. Dorfmeister, B. Fischer, S. Brunthaler, and S. Volckaert, "Talk: GlueZilla: Efficient and scalable software to hardware binding using rowhammer," in *Proceedings of the 1st Microarchitecture Security Conference*, Mar. 2025. DOI: 10.46586/UASC.2025.101.

[148] M. Vergnolle, "Defending against reverse engineering using Rowhammer," M.S. thesis, Technical University of Munich, Munich, Oct. 2024.

[149] D. Dorfmeister, M. Vergnolle, and S. Brunthaler, "Angluerus: Data-based hardware-software binding through Rowhammer," in *Tagungsband des 23. Kolloquium Programmiersprachen und Grundlagen der Programmierung*, S. Brunthaler, Ed., 2026, p. 11. DOI: 10.18726/154170.

[150] A. Kogler, J. Juffinger, S. Qazi, Y. Kim, M. Lipp, N. Boichat, E. Shiu, M. Nissler, and D. Gruss, "Half-Double: Hammering from the next row over," in *31st USENIX Security Symposium (USENIX Security 22)*, USENIX Association, Aug. 2022, pp. 3807–3824, ISBN: 978-1-939133-31-1. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/kogler-half-double.

[151] F. de Ridder, P. Frigo, E. Vannacci, H. Bos, C. Giuffrida, and K. Razavi, "SMASH: Synchronized many-sided rowhammer attacks from JavaScript," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021, pp. 1001–1018, ISBN: 978-1-939133-24-3. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/ridder.

[152] B. Fischer, "Design of a rowhammer-based unique hardware identification mechanism," M.S. thesis, University of Applied Sciences Upper Austria, Hagenberg, May 2023. [Online]. Available: https://permalink.obvsg.at/fho/AC16895522.

[153] J. Thiele, "Implementation of a tool to automatically create Rowhammer fingerprints," M.S. thesis, University of the Bundeswehr Munich, Neubiberg, Jul. 2023.

[154] *google/rowhammer-test: Test DRAM for bit flips caused by the rowhammer problem*, Google, 2014. [Online]. Available: https://github.com/google/rowhammer-test.

[155] *Examining process page tables, The Linux kernel documentation*, The kernel development community. [Online]. Available: https://www.kernel.org/doc/html/v6.18/admin-guide/mm/pagemap.html (visited on 01/12/2026).

[156] M. K. Bepary, B. M. S. B. Talukder, and M. T. Rahman, "DRAM retention behavior with accelerated aging in commercial chips," *Applied Sciences*, vol. 12, no. 9, p. 4332, Jan. 2022. DOI: 10.3390/APP12094332.

[157] L. Orosa, U. Rührmair, A. Giray Yağlikçi, H. Luo, A. Olgun, P. Jattke, M. Patel, J. S. Kim, K. Razavi, and O. Mutlu, "SpyHammer: Understanding and exploiting RowHammer under fine-grained temperature variations," *IEEE Access*, vol. 12, pp. 80 986–81 003, 2024. DOI: 10.1109/ACCESS.2024.3409389.

[158] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: A large-scale field study," *Communications of the ACM*, vol. 54, no. 2, pp. 100–107, Feb. 2011. DOI: 10.1145/1897816.1897844.

[159] "Intel® 64 and IA-32 architectures software developer's manual, Combined volumes: 1, 2a, 2b, 2c, 2d, 3a, 3b, 3c, 3d, and 4," Intel Corporation. (Oct. 2025), [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/671200.

[160] J. R. Levine, *Linkers and Loaders*. Morgan Kaufmann, 2000, ISBN: 978-1-55860-496-4.

[161] S. Zeitouni, D. Gens, and A.-R. Sadeghi, "It's hammer time: How to attack (Rowhammer-based) DRAM-PUFs," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6. DOI: 10.1109/DAC.2018.8465890.

[162] L. Drack, "SRAM based physical unclonable functions for low-cost embedded systems," M.S. thesis, University of Applied Sciences Upper Austria, Hagenberg, May 2022. [Online]. Available: https://permalink.obvsg.at/fho/AC16587210.

[163] M. Zeinzinger, J. Langer, F. Eibensteiner, P. Petz, L. Drack, D. Dorfmeister, and R. Ramler, "Comparative analysis of SRAM PUF temperature susceptibility on embedded systems," in *2023 International Conference on Electrical, Computer and Energy Technologies (ICECET)*, IEEE, Nov. 2023, pp. 1–8. DOI: 10.1109/ICECET58911.2023.10389242.

[164] Y. Dodis, L. Reyzin, and A. Smith, "Fuzzy extractors: How to generate strong keys from biometrics and other noisy data," in *Advances in Cryptology - EUROCRYPT 2004*, Springer, 2004. DOI: 10.1007/978-3-540-24676-3_31.

[165] R. Canetti, B. Fuller, O. Paneth, L. Reyzin, and A. Smith, "Reusable fuzzy extractors for low-entropy distributions," *Journal of Cryptology*, vol. 34, Sep. 2020. DOI: 10.1007/S00145-020-09367-8.

[166] D. Dorfmeister, F. Ferrarotti, B. Fischer, E. Haslinger, R. Ramler, and M. Zimmermann, "An approach for safe and secure software protection supported by symbolic execution," in *Database and Expert Systems Applications - DEXA 2023 Workshops*, G. Kotsis, A. M. Tjoa, I. Khalil, B. Moser, A. Mashkoor, J. Sametinger, and M. Khan, Eds., Springer, Aug. 2023, pp. 67–78. DOI: 10.1007/978-3-031-39689-2_7.

[167] E. Börger and R. Stärk, *Abstract State Machines, A Method for High-Level System Design and Analysis*. Springer, 2003, ISBN: 978-3-642-62116-1.

[168] M. Ahmed and M. Safar, "Symbolic execution based verification of compliance with the ISO 26262 functional safety standard," in *2019 14th International Conference on Design & Technology of Integrated Systems In Nanoscale Era (DTIS)*, IEEE, Apr. 2019, pp. 1–6. DOI: 10.1109/DTIS.2019.8735046.

[169] E. Börger and A. Raschke, *Modeling Companion for Software Practitioners*. Springer, 2018, ISBN: 978-3-662-56639-8.

[170] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976. DOI: 10.1145/360248.360252.

[171] C. S. Pasareanu, *Symbolic Execution and Quantitative Reasoning, Applications to Software Safety and Security*. Springer, 2020, ISBN: 978-3-031-01423-9.

[172] V.-A. Paun, B. Monsuez, and P. Baufreton, "Integration of symbolic execution into a formal abstract state machines based language," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 11 251–11 256, 2017, 20th IFAC World Congress. DOI: 10.1016/J.IFACOL.2017.08.1610.

[173] E. Börger, *The role of executable abstract programs in software development and documentation*, 2022. DOI: 10.48550/ARXIV.2209.06546.

[174] "DEPS: One-way-traffic lights demo – IP protection through hardware-software binding," SCCH Software Competence Center Hagenberg. (2025), [Online]. Available: https://youtu.be/J57ZB_64jak (visited on 01/12/2026).

[175] K. Ogata, *Modern Control Engineering*, 5th ed. Prentice Hall, 2010, ISBN: 978-0-13-615673-4.

[176] R. K. Brayton, G. D. Hachtel, L. A. Hemachandra, A. R. Newton, and A. L. M. Sangiovanni-Vincentelli, "A comparison of logic minimization strategies using ESPRESSO: An APL program package for partitioned logic minimization," in *ISCAS'82*, IEEE, 1982, pp. 42–48.

[177] "DEPS, Plasser & Theurer: Schutz geistigen Eigentums durch Hardware-Software-Bindung." German, SCCH Software Competence Center Hagenberg. (2025), [Online]. Available: https://youtu.be/tkt14dR2Y-8 (visited on 01/12/2026).

[178] S. Cottinet, *scottinet/espresso-logic-minimizer: A Node.js bridge to the Espresso heuristic logic minimizer original C code*, 2016. [Online]. Available: https://github.com/scottinet/espresso-logic-minimizer.

[179] B. Cottier, R. Rahman, L. Fattorini, N. Maslej, T. Besiroglu, and D. Owen, *The rising costs of training frontier AI models*, 2025. DOI: 10.48550/ARXIV.2405.21015.

[180] F. Boenisch, "A systematic review on model watermarking for neural networks," *Frontiers in Big Data*, vol. Volume 4 - 2021, 2021. DOI: 10.3389/FDATA.2021.729663.

[181] L. Fan, K. W. Ng, C. S. Chan, and Q. Yang, "DeepIPR: Deep neural network ownership verification with passports," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 10, pp. 6122–6139, 2022. DOI: 10.1109/TPAMI.2021.3088846.

[182] F. Tramèr and D. Boneh, "Slalom: Fast, verifiable and private execution of neural networks in trusted hardware," in *7th International Conference on Learning Representations, ICLR 2019*, OpenReview.net, May 2019. [Online]. Available: https://openreview.net/forum?id=rJVorjCcKQ.

[183] R. Ding, T. Xu, A. A. Ding, and Y. Fei, "Graph in the vault: Protecting edge GNN inference with trusted execution environment," in *2025 62nd ACM/IEEE Design Automation Conference (DAC)*, 2025, pp. 1–7. DOI: 10.1109/DAC63849.2025.11132467.

[184] Z. Á. Mann, C. Weinert, D. Chabal, and J. W. Bos, "Towards practical secure neural network inference: The journey so far and the road ahead," *ACM Computing Surveys*, vol. 56, no. 5, Nov. 2023. DOI: 10.1145/3628446.

[185] Y. Gong, X. Chang, J. Mišić, V. B. Mišić, J. Wang, and H. Zhu, "Practical solutions in fully homomorphic encryption: A survey analyzing existing acceleration methods," *Cybersecurity*, vol. 7, no. 1, p. 5, 2024. DOI: 10.1186/S42400-023-00187-4.

[186] X. Ni, H. Huttunen, and E. Rahtu, "On the importance of encrypting deep features," in *2021 IEEE/CVF International Conference on Computer Vision Workshops (ICCVW)*, 2021, pp. 4125–4132. DOI: 10.1109/ICCVW54120.2021.00460.

[187] A. Chakraborty, A. Mondai, and A. Srivastava, "Hardware-assisted intellectual property protection of deep learning models," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218651.

[188] M. Alam, S. Saha, D. Mukhopadhyay, and S. Kundu, "NN-Lock: A lightweight authorization to prevent IP threats of deep learning models," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 18, no. 3, Apr. 2022. DOI: 10.1145/3505634.

[189] R. Alur, *Principles of Cyber-Physical Systems*. MIT Press, 2015, ISBN: 978-0-262-02911-7.

[190] D. Seto, B. Krogh, L. Sha, and A. Chutinan, "The Simplex architecture for safe online control system upgrades," in *Proceedings of the 1998 American Control Conference*, vol. 6, 1998, pp. 3504–3508. DOI: 10.1109/ACC.1998.703255.

[191] D. Dorfmeister, F. Ferrarotti, B. Fischer, M. Schwandtner, and H. Sochor, "A PUF-based approach for copy protection of intellectual property in neural network models," in *Software Quality as a Foundation for Security*, P. Bludau, R. Ramler, D. Winkler, and J. Bergsmann, Eds., Springer, Apr. 2024, pp. 153–169. DOI: 10.1007/978-3-031-56281-5_9.

[192] H. Sochor, M. Schwandtner, D. Dorfmeister, B. Fischer, F. Ferrarotti, and J. Fuß, "Locki: Neural networks that only behave when unlocked via runtime decryption," to appear in *Proceedings of 37th International Conference on Database and Expert Systems Applications (DEXA 2026)*.

[193] D. J. Bernstein, "ChaCha, a variant of Salsa20," in *Workshop Record of SASC 2008: The State of the Art of Stream Ciphers*, 2008. [Online]. Available: https://cr.yp.to/papers.html#chacha.

[194] Y. Nir and A. Langley, *ChaCha20 and Poly1305 for IETF Protocols*, RFC 8439, Jun. 2018. DOI: 10.17487/RFC8439.

[195] Y. G. Achamyeleh, S.-Y. Yu, G. Q. Araya, and M. A. Al Faruque, "Bridging the binary analysis gap: A cross-compiler dataset and neural framework for industrial control systems," in *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, ser. KDD '25, ACM, 2025, pp. 5260–5269. DOI: 10.1145/3711896.3737373.

[196] A. Akshintala, B. Jain, C.-C. Tsai, M. Ferdman, and D. E. Porter, "x86-64 instruction usage among C/C++ applications," in *Proceedings of the 12th ACM International Conference on Systems and Storage*, ser. SYSTOR '19, ACM, 2019, pp. 68–79. DOI: 10.1145/3319647.3325833.

[197] A. Adiletta, Z. Weissman, F. K. Dana, B. Sunar, and S. Tajik, "Rubber Mallet: A study of high frequency localized bit flips and their impact on security," in *Fifth Workshop on DRAM Security (DRAMSec)*, Jun. 2025. [Online]. Available: https://dramsec.ethz.ch/dramsec25-papers/rubbermallet-dramsec25.pdf.

[198] A. Adiletta, M. C. Tol, K. Derya, B. Sunar, and S. Islam, "LeapFrog: The Rowhammer instruction skip attack," in *2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P)*, IEEE, 2025, pp. 1067–1081. DOI: 10.1109/EUROSP63326.2025.00065.

[199] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ECDSA)," *International Journal of Information Security*, vol. 1, no. 1, pp. 36–63, 2001. DOI: 10.1007/S102070100002.

[200] D. F. Aranha, F. R. Novaes, A. Takahashi, M. Tibouchi, and Y. Yarom, "LadderLeak: Breaking ECDSA with less than one bit of nonce leakage," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20, ACM, Oct. 2020, pp. 225–242. DOI: 10.1145/3372297.3417268.

[201] M. Godbolt. "The BTB in contemporary intel chips, Matt godbolt's blog." (2016), [Online]. Available: https://xania.org/201602/bpu-part-three (visited on 01/12/2026).

[202] R. Qiao and M. Seaborn, "A new approach for rowhammer attacks," in *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, IEEE, May 2016, pp. 161–166. DOI: 10.1109/HST.2016.7495576.

[203] M. T. Aga, Z. B. Aweke, and T. Austin, "When good protections go bad: Exploiting anti-DoS measures to accelerate rowhammer attacks," in *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, IEEE, May 2017, pp. 8–13. DOI: 10.1109/HST.2017.7951730.

[204] Z. Zhang, Y. Cheng, D. Liu, S. Nepal, Z. Wang, and Y. Yarom, "PThammer: Cross-user-kernel-boundary rowhammer through implicit accesses," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, Oct. 2020, pp. 28–41. DOI: 10.1109/MICRO50266.2020.00016.

[205] Z. Zhang, W. He, Y. Cheng, W. Wang, Y. Gao, D. Liu, K. Li, S. Nepal, A. Fu, and Y. Zou, "Implicit Hammer: Cross-privilege-boundary rowhammer through implicit accesses," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–18, 2022. DOI: 10.1109/TDSC.2022.3214666.

[206] P. Rajasekaran, S. Crane, D. Gens, Y. Na, S. Volckaert, and M. Franz, "CoDaRR: Continuous data space randomization against data-only attacks," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '20, ACM, Oct. 2020, pp. 494–505. DOI: 10.1145/3320269.3384757.

[207] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis, "Compiler-assisted code randomization," in *2018 IEEE Symposium on Security and Privacy (SP)*, IEEE, May 2018, pp. 461–477. DOI: 10.1109/SP.2018.00029.

[208] S. Bhatkar and R. Sekar, "Data space randomization," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, D. Zamboni, Ed., Springer, 2008, pp. 1–22. DOI: 10.1007/978-3-540-70542-0_1.

[209] "Program memory usage," PlatformIO. (May 2022), [Online]. Available: https://docs.platformio.org/en/latest/faq/program-memory-usage.html (visited on 01/23/2026).

[210] L. Deng, "The MNIST database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012. DOI: 10.1109/MSP.2012.2211477.

[211] H. Xiao, K. Rasul, and R. Vollgraf, *Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms*, Aug. 2017. DOI: 10.48550/ARXIV.1708.07747.

[212] P. Warden, *Speech commands: A dataset for limited-vocabulary speech recognition*, 2018. DOI: 10.48550/ARXIV.1804.03209.

[213] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, D. Lin, Y. Matsumoto, and R. Mihalcea, Eds., ACL, Jun. 2011, pp. 142–150. [Online]. Available: https://aclanthology.org/P11-1015/.

[214] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *2017 IEEE Symposium on Security and Privacy (SP)*, IEEE, May 2017, pp. 3–18. DOI: 10.1109/SP.2017.41.

[215] G. Liu, T. Xu, R. Zhang, Z. Wang, C. Wang, and L. Liu, "Gradient-Leaks: Enabling black-box membership inference attacks against machine learning models," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 427–440, 2024. DOI: 10.1109/TIFS.2023.3324772.

[216] M. Nasr, R. Shokri, and A. Houmansadr, "Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning," in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 739–753. DOI: 10.1109/SP.2019.00065.

[217] M. Fredrikson, S. Jha, and T. Ristenpart, "Model inversion attacks that exploit confidence information and basic countermeasures," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, I. Ray, N. Li, and C. Kruegel, Eds., ACM, 2015, pp. 1322–1333. DOI: 10.1145/2810103.2813677.

[218] Y. Zhang, R. Jia, H. Pei, W. Wang, B. Li, and D. Song, "The secret revealer: Generative model-inversion attacks against deep neural networks," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020*, CVF/IEEE, 2020, pp. 250–258. DOI: 10.1109/CVPR42600.2020.00033.

[219] L. Zhu, Z. Liu, and S. Han, "Deep leakage from gradients," in *Advances in Neural Information Processing Systems*, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, Eds., vol. 32, Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2019/file/60a6c4002cc7b29142def8871531281a-Paper.pdf.

[220] A. Ghimire, S. R. Lingala, J. Zhang, F. Alsulami, and F. Amsaad, "A survey on application of AI on reverse engineering for software analysis and security," *IEEE Access*, vol. 13, pp. 152 903–152 913, 2025. DOI: 10.1109/ACCESS.2025.3593456.