universidad de león

Escuela de Ingenierías I.I.

Industrial, Informática y Aeroespacial

MÁSTER UNIVERSITARIO EN INVESTIGACIÓN EN CIBERSEGURIDAD

Trabajo de Fin de Máster

FIRMWARE VULNERABILITY DISCOVERY VIA LLVM LIFTING AND MACHINE LEARNING

Autor: Erik Cembreros Otero

Tutor: Ricardo J. Rodríguez

Co-Tutora: Camino Fernández Llamas

(Septiembre, 2026)

UNIVERSIDAD DE LEÓN

Escuela de Ingenierías I.I.

MÁSTER UNIVERSITARIO EN INVESTIGACIÓN EN CIBERSEGURIDAD

Trabajo de Fin de Máster

ALUMNO: Erik Cembreros Otero

TUTOR: Ricardo J. Rodríguez

CO-TUTORA: Camino Fernández Llamas

TÍTULO: Firmware Vulnerability Discovery via LLVM Lifting and Machine Learning

CONVOCATORIA: Septiembre, 2026

RESUMEN:

The software driving routers, cameras, and industrial controllers, commonly referred to as *firmware*, is rarely audited for security. It is distributed as compiled binary images across diverse hardware architectures, without source code or standardised vulnerability datasets for machine learning. This work addresses this gap with a scalable, end-to-end pipeline that moves from vendor download pages to vulnerability prediction without source code or manual reverse engineering. The pipeline scrapes real-world firmware images and automatically labels them by matching vendor versions against affected ranges in the National Vulnerability Database. Binaries extracted from these images are lifted into an architecture-independent low-level virtual machine intermediate representation (IR), embedded into fixed-length vectors, and used to train classifier models. This process yields a unique corpus of real-world firmware paired with image-level vulnerability labels. Models trained on the lifted-IR embeddings detect vulnerable firmware with statistically significant performance on a controlled benchmark (validated via label-permutation tests), demonstrating that vulnerability signals can be recovered directly from binary-derived IR at scale.

Palabras clave: firmware, vulnerability detection, LLVM lifting, machine learning, cybersecurity.

Firma del alumno:	**VºBº Tutor:**	**VºBº Co-Tutora:**

Contents

List of Figures

List of Tables

Declaration of Generative AI and AI-Assisted Technologies

During the preparation of this work, the author used Claude Opus 4.8 and Claude Opus 5.0 to improve the readability and language of the text, to write and debug parts of the pipeline implementation, and to produce the diagrams in this document. The author also used Undermind to search the literature. After using these tools, the author reviewed and edited the content as needed, verified every work cited against its original source, and assumes full responsibility for the content of this work.

The important decisions are the author's: the research question, the architecture of the pipeline and the stages it is built from, the tools chosen for each of them, the experiments and the controls applied to them, the interpretation of the results, and what this document argues.

Glossary of terms

ABI: Application Binary Interface. The low-level runtime convention governing how compiled code interfaces, including calling conventions, register usage, and data-type sizes. Non-standard MIPS ABIs (N32, O64) are a source of lifting failures here.

Binwalk: An open-source tool for extracting and analysing firmware images through signature-based recursive decomposition.

CPE: Common Platform Enumeration. A structured naming scheme for software, hardware, and operating systems, used in CVE records to identify affected products.

CVE: Common Vulnerabilities and Exposures. A publicly available dictionary of known cybersecurity vulnerabilities, each assigned a unique identifier.

CVSS: Common Vulnerability Scoring System. A standard for rating the severity of security vulnerabilities on a 0–10 scale, recorded in CVE entries and used here at labelling.

Def-use chain: The link that joins the definition of a value in a program to each of the places where that value is later used.

DIR / EIR: Decompiler-style IR and Emulation-style IR, the two lifter paradigms. DIR (e.g., RetDec) recovers variables, types, and control flow; EIR (e.g., Remill) preserves explicit machine state per instruction.

ELF: Executable and Linkable Format. The standard container format for executables, object code, and shared libraries on Unix-like systems; its header identifies the target architecture.

FA: Flow-aware. The IR2Vec encoding that augments the symbolic one with information propagated along def-use chains and the control-flow graph.

Firmware: Embedded software stored in non-volatile memory of a hardware device, responsible for controlling low-level device operations.

HPC: High-Performance Computing. The use of clusters of powerful compute nodes for computationally intensive workloads, such as the Priscilla cluster used here for lifting and embedding.

IoT: Internet of Things. The class of network-connected embedded devices, such as routers, cameras, and industrial controllers, whose firmware this work analyses.

IR2Vec: An embedding scheme that maps LLVM IR programs to continuous vector representations by encoding semantic relationships between opcodes, types, and operands.

Lifting: The process of translating binary machine code into a higher-level intermediate representation, such as LLVM IR, without access to the original source code.

LLVM IR: Low-Level Virtual Machine Intermediate Representation. A language-agnostic, architecture-independent code representation.

NVD: National Vulnerability Database. The U.S. government repository of standards-based vulnerability management data, providing CVE records and associated metadata.

RBF: Radial Basis Function. A kernel used by support vector machines to define a non-linear decision boundary from distances between points.

Remill: Binary lifter by Trail of Bits following the EIR paradigm, translating each machine instruction into LLVM IR that reproduces its effect on processor state. Used here for x86-64, x86-32, and AArch64.

RetDec: Retargetable decompiler developed by Avast. An open-source static lifter that produces LLVM IR following the DIR paradigm from executables of various architectures.

ROC-AUC: Area under the receiver operating characteristic curve. The probability that a model ranks a random vulnerable sample above a random non-vulnerable one; 0.5 corresponds to chance. It is the primary metric used in Chapter 5.

SDK: Software Development Kit. The toolchain, libraries, and reference code that a chipset vendor supplies to device manufacturers, and a common route by which the same third-party components reach the firmware of many vendors.

SVM: Support Vector Machine. A classifier that separates two classes by the widest possible margin, either with a hyperplane or, through an RBF kernel, with a curved boundary.

SYM: Symbolic. The IR2Vec encoding that represents each instruction from its opcode and operand types alone.

WAL: Write-Ahead Logging. A journaling mode, used here by SQLite, that records changes to a log before applying them, enabling concurrent-safe reads and writes.

Chapter 1

Introduction

1.1 Problem Statement

When vulnerabilities are discussed, the picture that comes to mind is almost always an application. A bug in a browser, an injection in a website, ransomware arriving through an email, a phone app leaking what it should not. That is where the security industry has settled its attention, since that is where the users are.

The infrastructure all of that rests on, however, is not made of applications. The core routers moving a telecommunications operator's traffic, the firewalls at the edge of every corporate network, the controllers regulating a substation or a water treatment plant, the equipment keeping a hospital running: none of these is a program installed on a general-purpose computer. They are hardware, and the software governing them lives inside them, as firmware [20, 49]. The backbone of modern society runs on code, and it is firmware code that very few people ever look at.

The reason has less to do with how that code is written than with who can get at it. An application can be run by anyone who wishes to study it: a researcher installs it on an ordinary machine, feeds it malformed input, attaches a debugger, and fuzzes it. An entire ecosystem of bug bounties, continuous fuzzing infrastructure, and commercial auditing is aimed at such software because reaching it is easy. However, firmware is harder to reach. It is bound to the physical system it was written for, expecting particular peripherals at particular addresses, sensors that answer in particular ways, and timing that only the real device exhibits. Removed from its device the code will often not run, and left inside it can only be observed by someone who owns the device and is skilled enough to open it [11, 21]. The population able

to examine a given firmware image therefore shrinks to those few, plus the vendor, which is a hardware company whose engineering effort goes into the next product rather than into the security of the last one.

What also makes firmware hard to audit is that it does not look like the software the analysis community is equipped for. It arrives as a monolithic binary image, without source code, without debug symbols, frequently compressed or packed [19], and compiled for whichever instruction set suited the device, so that MIPS, ARM32, PowerPC, and AArch64 all appear in the same corpus [20]. Understanding a single image is a reverse-engineering exercise that can take days. The corpus, meanwhile, is measured in thousands of images, which makes the manual approach infeasible.

Automation has been attempted along two main lines. One runs the firmware under whole-system emulation in order to observe how it behaves, but simply getting an image to boot demands per-device effort and fails on anything whose peripherals are not modelled [11, 21]. The other reasons symbolically about many execution paths at once, but the number of paths grows explosively and the analysis stalls well short of a complete firmware image [8, 37]. Neither offers a way to decide which images are worth analysing in the first place.

Over the same period, machine learning has become a default instrument across essentially every discipline with data to learn from, and the analysis of code is no exception. A substantial body of work now learns to recognise vulnerable code directly from labelled examples instead of from hand-written rules, either by classifying code fragments outright [27, 51] or by first mapping code to vectors on which ordinary classifiers can operate [1]. The results are encouraging, and they share an assumption: they are built on source code.

That assumption is not a matter of taste but a consequence of where the data is. Labelled vulnerability datasets exist at source level because open-source repositories and their patch histories can be mined at scale: a commit fixing a disclosed vulnerability points at the flawed code and at its repaired form simultaneously, and millions of such commits are public. Firmware offers no equivalent, and the corpora that do exist each stop short in a different place. Large collections of real firmware have been assembled with careful provenance and metadata [20], but they attach vulnerability either by inventorying which known-vulnerable third-party components an image happens to contain [49], which describes its ingredients instead of judging the image, or by matching a handful of devices against public exploits, which yields ground truth for a hundred samples rather than for a corpus. Other collections

do pair code with vulnerability labels and with an intermediate representation, but obtain that representation by compiling source the authors held [25], which is precisely the situation firmware is not in. What does not exist is a corpus of real-world firmware images carrying vulnerability labels at the level of the image and expressed in a representation a classifier can consume.

Two obstacles therefore remain. The first is labelling: deciding which images are vulnerable and which are not. The second is heterogeneity: raw binaries are not uniform, and a model trained on MIPS has learned nothing about ARM32.

The contribution of this work is a pipeline that aims to remove both obstacles end to end. It discovers firmware on vendor sites and content delivery networks, downloads it, labels it by correlating its version against public vulnerability records, extracts the executables inside, lifts them to the Low-Level Virtual Machine intermediate representation (LLVM IR) across the six architectures its two lifters cover between them, unifying their vocabulary, and embeds the result into vectors ready for training, with no source code and no human reverse engineer at any point along the way. Existing firmware corpora stop at acquisition, and the existing work on learning from IR begins after compilation; what has been missing is the chain that joins them. The dataset this pipeline produces is the second contribution: real-world firmware images, labelled at the level of the image and represented as vectors drawn from a single architecture-independent IR, a combination no public dataset currently offers. The third is the evidence that the pipeline works. On a benchmark holding both vendor and architecture fixed, so that nothing but the vulnerability label remains to be learned, linear models reach an area under the receiver operating characteristic curve (ROC-AUC) of 0.72–0.76 on the symbolic space and 0.72–0.78 on the flow-aware one, against a permutation-test baseline that rules out chance.

1.2 Objectives

The global objective of this work is to design, build, and evaluate a scalable pipeline that goes from real-world firmware images to a vulnerability prediction, without access to source code and without manual reverse engineering at any stage. This global objective is pursued through five specific objectives:

1. Build a large-scale, real-world firmware collection and labelling infrastructure correlating firmware images with Common Vulnerabilities and Exposures (CVE) records.

2. Develop a binary lifting pipeline capable of handling heterogeneous embedded architectures, producing LLVM IR as a normalised representation.
3. Generate vector embeddings from architecture-independent LLVM IR using IR2Vec, an embedding scheme that maps IR to fixed-length vectors.
4. Construct a labelled dataset of vulnerable and non-vulnerable firmware samples.
5. Train and evaluate a classifier for firmware vulnerability prediction.

1.3 Structure of the Work

The remainder of this document is organised as follows. Chapter 2 introduces the background concepts the work relies on: firmware and its vulnerabilities, binary lifting and LLVM IR, and code embeddings and classification. Chapter 3 reviews the state of the art in firmware security analysis, binary lifting, and machine-learning-based vulnerability detection. Chapter 4 describes the methodology, covering the construction of the labelled dataset and the design of the vulnerability detector. Chapter 5 discusses the results. Finally, Chapter 6 draws conclusions and outlines future work. Appendix A records the project planning and the deviations from the initial schedule.

Chapter 2

Background

This work sits at the intersection of three different fields: embedded systems, binary analysis, and machine learning. In this chapter, we explain the fundamentals the rest of the document depends on: what firmware is and why its vulnerabilities are so durable, how public vulnerability records describe the software they affect, what binary lifting does and why LLVM IR is the representation of choice for it, and finally how code can be turned into vectors that a classifier is able to learn from. A reader already fluent in any of these areas can safely skip the corresponding section.

2.1 Firmware and Embedded Systems

Firmware is the software held in the non-volatile memory of a hardware device, where it provides the lowest layer of control over that device's operation [20]. The term covers a wide spectrum. At one extreme sits a few kilobytes of hand-written C on a microcontroller with no operating system, driving a sensor in a loop. At the other sits a complete Linux system, with a kernel, a shell, a web server, and a filesystem full of ordinary binaries, which is what ships inside a modern router, a network camera, or a home NAS [11, 21]. We are concerned with the second kind of firmware, because those images contain conventional executables that a binary analysis can be pointed at, and because it is those devices that sit on the network and are worth attacking.

Such firmware is distributed as a single image, downloaded from the vendor's website as an opaque blob of a few megabytes. Inside, an image typically concatenates a bootloader, a compressed kernel, and one or more root filesystems, with no standard governing the layout; each vendor packs its image as it sees fit [20].

Recovering the executables inside means identifying and unpacking those structures by signature, using tools such as Binwalk [19], and this recursive, signature-driven extraction is the first stage of any firmware analysis.

What makes firmware distinctive from a security standpoint is not its content but its lifecycle. A router bought today may still be routing traffic in ten years, on the image it shipped with, because embedded devices are rarely configured to update themselves and their owners rarely do it by hand. Vendors declare products end-of-life while large numbers of units remain deployed, at which point no patch will ever be issued regardless of what is found. The same image, moreover, is commonly reused across an entire product line [20], and the components inside it (BusyBox, OpenSSL, etc.) are reused across vendors through shared chipset software development kits (SDKs) [49].

These properties compound: a vulnerability in a widely reused component is inherited by every image built on it; every image is deployed on many devices; and every device stays vulnerable for as long as it stays plugged in. Therefore, a single flaw expands into hundreds of thousands of exposed hosts, which is the mechanism behind the large embedded botnets of the last decade [4]. This also explains why the ability to tell, cheaply and at scale, which images are likely to be vulnerable has practical value beyond any individual device.

2.2 Firmware Vulnerabilities

The vulnerabilities found in firmware are, to a large extent, the classes of vulnerability that server software faced two decades ago [42, 49]. The reason is environmental: this code is written in C or C++ for cost and footprint reasons, compiled with old toolchains, and deployed without the mitigations that have become standard elsewhere, so that address space layout randomisation, stack canaries, and non-executable memory are frequently absent or only partly enabled [42, 47].

Firmware then adds failure modes of its own. Command injection is common because embedded web interfaces routinely implement their functionality by building shell command strings from request parameters [11]. Hardcoded credentials survive in shipped images, whether debug accounts left enabled or keys shared across an entire product line [13]. Services intended for a local network are exposed to the internet in default configurations [4]. And because images bundle third-party components frozen at whatever version the SDK provided, a device inherits every

vulnerability disclosed in those components since it was built [49], without any mistake being made by the vendor's own developers.

When such a vulnerability is disclosed it is recorded publicly, and the shape of that record matters for this work because it makes labelling possible. Each disclosed vulnerability receives a CVE identifier, and the National Vulnerability Database (NVD) publishes the associated metadata, including a description, a severity score, and a list of affected products [32]. Those products are named using Common Platform Enumeration (CPE), a structured naming scheme that identifies a vendor, a product, and a version [10], and a CVE record expresses the affected set either as individual CPE names or as version ranges.

This is the connection we exploit. A firmware image downloaded from a vendor also carries a vendor, a product, and a version, whether in its filename, its download URL, or its release notes. If we can match that data against the CPE ranges of the CVE record, then we can label the image as vulnerable or not without anyone examining its contents, which is precisely what supervised learning requires and what no existing firmware dataset provides.

2.3 Binary Lifting and LLVM IR

Analysing firmware means analysing machine code, and machine code is a poor subject for analysis. It is specific to one instruction set, so any technique developed for one architecture must be redeveloped for another; it is untyped, having lost the variables and structures the programmer wrote; and its control flow is implicit in jumps to addresses rather than stated in any structure [37].

The response is to translate machine code into a single representation that is independent of the processor it came from. LLVM IR [24] is the usual target. Originally the internal language of the LLVM compiler, it has become an analysis substrate in its own right [34, 41]. Three of its properties matter here: it is typed, so values carry types instead of being anonymous bit patterns; it is in static single assignment form, meaning each value is assigned exactly once, which makes the flow of data through a program explicit and easy to follow; and it is architecture-independent, so the binary, once translated, is expressed in the same language and can be handled by the same downstream analysis.

Translating machine code back up into LLVM IR is called *lifting*, and the tool that does it is a *lifter* [2, 28]. Lifting inverts what a compiler does, without the

information the compiler discarded. It is not decompilation to source code, which aims at human readability; the goal is a representation a machine can reason over.

A lifter proceeds in stages. It first loads the binary, parsing the executable format to determine the target architecture and locate the sections and the entry point. It then decodes the bytes of the code sections into individual machine instructions, and each instruction is translated into LLVM IR that reproduces its semantics, which requires the lifter to define a runtime model of the machine environment: the registers, the condition flags, the stack, and the program counter must all be represented explicitly, because the source language of this translation is a processor. A final refinement stage transforms this initial, verbose IR into something more analysable [2, 28].

We can distinguish two families of lifters depending on what to lift: which bytes in the image are code, and how those bytes group into functions [28]. A *decompiler-style lifter* (DIR), such as RetDec [5], answers this itself, reconstructing the program's control-flow graph as an inseparable part of its design, because it cannot recover variables and types without first knowing the shape of the program. An *emulation-style lifter* (EIR), such as Remill [44], answers none of it. Its scope is instruction semantics: it translates whatever bytes it is handed and asks no questions about where they came from. The work is delegated to a separate front-end, typically a disassembler, which need only supply function boundaries and not a full graph. This is where the modularity difference between the two comes from, and it is a design decision rather than an oversight. Separating the decision of what to lift from the act of lifting lets the front-end be replaced, and lets a binary be carved into functions and lifted piecemeal.

The refinement stage is where lifters diverge again, and the divergence defines the two paradigms. An EIR lifter keeps the machine model explicit: the IR it emits reads as a faithful simulation of the processor, updating registers and flags instruction by instruction. A DIR lifter instead applies heuristics to recover what the compiler threw away, promoting stack slots back into local variables, inferring types, and reconstructing function prototypes, so that its output resembles compiled code rather than an emulator.

The choice between them is a real trade-off, not a question of quality, and it is judged along several axes [28]. Functional correctness asks whether the lifted IR can be recompiled into a working executable that still passes the original tests, and it is the axis on which DIR lifters struggle, since a wrong type inference produces a wrong

program. Discriminability asks whether IR from different algorithms actually looks different once embedded, which is exactly what a learning-based analysis depends on and where EIR suffers, because emulation boilerplate dominates the IR and makes unrelated programs look alike. Structuredness measures how readable and well-formed the output is, typically by counting `goto` statements and lines of code per function. Static analysis support asks how well the IR serves the pointer and value-flow analyses built on top of it, where distinguishing pointers from ordinary integers remains the central difficulty.

2.4 Learning from Code

The task we address here is supervised binary classification, the most conventional setting in machine learning. A model is shown a collection of examples, each a fixed-length vector of numbers accompanied by a label, and it searches for a rule that reproduces the labels from the vectors. If the rule generalises, it will assign correct labels to examples it has never seen. Here each example is a firmware image, and the label states whether it is known to be vulnerable.

The difficulty, therefore, is that programs are not vectors. A model requires a fixed-length sequence of numbers, whereas a program is a variable-length structure of instructions related by control and data flow, and the entire problem lies in crossing that gap without discarding whatever it is that makes vulnerable code vulnerable. The bridge is an *embedding*: a function that maps a program, or a fragment of one, to a point in a continuous vector space, arranged so that geometric proximity reflects semantic similarity, with programs that do similar things landing near each other.

The idea is borrowed from natural language processing, where words are embedded by exploiting the observation that a word is characterised by the company it keeps, so that words appearing in similar contexts receive similar vectors. Applied to code, the same principle can be driven by the contexts in which instructions or operands appear, and the representation chosen then determines what the resulting vectors can express. Embedding source code binds the result to a language and requires source in the first place [1]; embedding an intermediate representation such as LLVM IR yields vectors that are comparable across languages and, when the IR came from a lifter, across architectures [6, 45]. Once each program is a vector, the vulnerability question reduces to an ordinary classification problem over vectors, and the standard tools of machine learning become available.

The classifiers we use are deliberately conventional. Logistic regression fits a linear boundary between the two classes and reports a calibrated probability. A support vector machine (SVM) looks for the boundary that separates the classes by the widest possible margin, either as a straight hyperplane in the linear case, or, with a radial basis function (RBF) kernel, as a curved boundary obtained by measuring distances between points rather than working in the original coordinates. A random forest takes an entirely different route, growing many decision trees on random subsets of the data and features and combining their votes, which lets it capture interactions between features that a linear model cannot express.

Establishing whether the embeddings carry any vulnerability signal requires a detailed evaluation, because a single split of a small dataset into training and test parts gives an estimate that swings wildly with the luck of the split. Cross-validation addresses this by dividing the data into folds, training on all but one and testing on the one held out, rotating through the folds, and reporting the mean and spread of the results, with a stratified variant preserving the class proportions in every fold. The choice of metric matters just as much when the classes are unbalanced: accuracy is misleading, since a corpus that is three-quarters vulnerable can be classified at 75% accuracy by a model that answers "vulnerable" unconditionally. ROC-AUC avoids this trap by measuring how well the model ranks a random vulnerable sample above a random non-vulnerable one [14], and we report it alongside precision, recall, and F1 throughout Chapter 5.

Chapter 3

State of the Art

This work draws on three research areas that have developed largely independently. We first discuss how firmware vulnerabilities are found today, and why the automation built so far stops short of the scale the problem demands. We then turn to how machine code can be lifted into LLVM IR, and which of the available lifters produce IR worth learning from. Finally, we discuss what has been built on top of LLVM IR, and how machine learning has been brought to code more generally.

3.1 Automated Vulnerability Discovery

Firmware analysis has received sustained attention over the last decade, driven by Internet of Things (IoT) security and by research into network appliances [13, 20]. The approaches divide naturally according to how they obtain information about a program: by running it, by reasoning about it without running it, or by learning from examples of it.

Dynamic analysis runs the firmware and watches what happens. Firmadyne [11] and FirmAE [21] are the representative systems, automating extraction, emulation, and dynamic analysis so that an image can be exercised without possessing the device it came from. How far that scales is a separate question. On a corpus of 1,124 router and camera images, Firmadyne boots only 183 of them (16%); FirmAE's arbitrated emulation raises that to 892 (79%), enough to uncover twelve previously unknown vulnerabilities. The cost is high, however. Both are difficult to install, resource intensive, and demanding of considerable manual effort per image, since emulation succeeds only to the extent that the device's peripherals have been modelled.

Static analysis inspects code without executing it, reasoning about all possible executions through program models such as control-flow graphs, data-flow relations, and abstract states, and not through individual test inputs [34, 41]. Data-flow and taint analyses track how values move from untrusted sources, such as network input or files, into dangerous sinks, such as `memcpy` or a system call, and flag the paths that connect them as candidate overflows or injections [26, 36]. Abstract interpretation approximates program states, tracking the possible ranges of variables or the aliasing relationships between pointers, and uses those approximations to detect out-of-bounds accesses, integer overflows, null dereferences, and use-after-free [38, 46]. Pattern-based checkers take a more direct route, encoding rules about what correct code should look like, for instance that a security-sensitive operation must be guarded by a check, and scanning for violations of them [39]. This approach is fast and easy to extend, but produces false positives unless it is combined with some data-flow reasoning about whether the match is actually reachable.

The last branch of the static family learns its patterns instead of being given them: code fragments are represented as feature vectors or embeddings, and a classifier is trained on labelled examples to predict whether a fragment is vulnerable [27, 51]. This is the branch to which the present work belongs, applying it to embeddings derived from lifted LLVM IR.

3.2 Binary Lifting to LLVM

Many binary lifters exist, spanning the EIR and DIR paradigms introduced in Section 2.3. In this work, we concentrate on two: RetDec [5], a DIR lifter, and Remill [44], an EIR lifter. Our choice is guided by a comparative study of lifters [28]: because we embed the IR rather than execute it, discriminability is the axis that matters, and it is the axis on which DIR output leads, with RetDec reaching 81.9% on that study's discriminability benchmark.

RetDec [5] is a static lifter developed by Avast that follows the DIR paradigm, and its advantage traces back to a single design decision: rather than emulating a physical stack through global arrays, it recovers local variables and types, so that what it emits resembles compiled code rather than a simulation of a processor.

These qualities come at a price, and the price is architectural. Because RetDec rebuilds the control-flow graph and operates on the binary as a whole, its work cannot be partitioned into individual functions, a division that would otherwise be available to make lifting faster or merely feasible. On a large binary whose control-

flow reconstruction demands too much of it, the tool simply fails, and there is no way to recover the functions it might have handled.

Remill [44], developed by Trail of Bits, occupies the opposite corner of the design space. Following the EIR paradigm, it translates each machine instruction into LLVM IR reproducing that instruction's exact effect on processor state, updating registers, condition flags, and memory instead of recovering the variables and types a decompiler would. Its output is verbose and carries the emulation boilerplate that costs EIR lifters their discriminability.

What Remill offers in exchange is modularity. Because instructions are lifted in isolation, with no assumptions made about the stack or about a function's arguments, a binary can be split into functions and only the interesting ones lifted, skipping whatever is unimportant or infeasible. Large binaries that would defeat a whole-binary approach therefore remain tractable, and the two tools end up complementary: RetDec produces the better IR, Remill produces IR where RetDec produces none.

3.3 LLVM Vulnerability Discovery

LLVM IR long ago outgrew its origin as a compiler-internal representation and became an analysis substrate in its own right, used for vulnerability discovery [34,41], malware analysis [5], and program hardening [35] alike. The work built on it falls into four broad groups.

The first group instruments the IR to install dynamic defences. Sanitizers, control-flow integrity, and temporal memory safety are implemented as instrumentation passes in Clang and LLVM, which add checks at compile time that detect violations at run time [16,18,23,35,43,50]. The second group analyses the IR statically and across procedures, implementing taint tracking, value-flow analysis, integer and IO2BO checking, and missing-check detection over LLVM IR that may have been compiled from source or lifted from a binary [26,36,38–40,46]. A third group builds hybrid compiler-level security stacks, letting a developer mark selected variables as sensitive and combining taint tracking, control-flow integrity, and memory safety to guarantee their confidentiality or integrity [7, 9]; the approach is interesting in itself but aimed at protecting code rather than at discovering flaws in it, and so falls outside the present scope.

The fourth group is infrastructural, and it is what makes the rest practical. Reusable frameworks such as SVF [41] and PhASAR [34], code property graphs over LLVM IR [22] and lifting frameworks themselves [2] provide control-flow graph construction, data-flow analysis, and pointer analysis as building blocks, so that a new analysis need not reimplement the foundations. The present pipeline depends on this ecosystem in the same way, taking LLVM IR as a given and building only what is specific to its own question.

3.4 Machine Learning over Code

The idea of learning from code instead of reasoning about it has developed along two lines that this work needs to bring together: how code is turned into vectors, and what is done with those vectors once vulnerability is the question.

The embedding line began at source level. code2vec [1] represents a method as a bag of paths through its abstract syntax tree and learns vectors from them, showing that a program fragment can be mapped to a point in a continuous space where semantic similarity becomes geometric proximity. Its dependence on source, and on one language's syntax, is exactly the limitation that motivated the move to intermediate representations. inst2vec [6] takes that step, embedding LLVM IR instructions using contextual flow graphs built from data and control dependencies, which yields a representation independent of the source language. IR2Vec [45], the scheme this work adopts, refines the approach: it encodes LLVM IR as a knowledge graph over opcodes, types, and operands, learning a seed vocabulary that is then composed into instruction, function, and program vectors, and offers both a symbolic mode and a flow-aware mode that propagates information along def-use chains and the control-flow graph. Because these vectors are derived from IR and not from source, they are in principle comparable across architectures once a lifter has supplied the IR, which is the property this work relies on.

The detection line asks what a classifier can do with such representations. VulDeePecker [27] learns from "code gadgets", sequences of semantically related source statements, and demonstrated that a neural network trained on labelled examples could outperform hand-written rules. Devign [51] replaces sequences with graphs, applying a graph neural network to a composite representation of a function's syntax, control flow, and data flow, on the argument that vulnerability is a structural property rather than a lexical one. Both work at source level and rely on datasets mined from open-source repositories. More recent work moves to LLVM IR as the input

representation and applies large language models [29] and Transformer architectures [30], exploiting the fact that IR normalises away surface differences between source languages. Rust-IR-BERT [25] is the clearest instance of the pattern this work also follows: a pretrained model, GraphCodeBERT, supplies the embedding, and a separate classifier, CatBoost, decides on it, so that the representation and the decision are learned by different components rather than by one network end to end. TACSan [48] pursues the graph-based route with a graph neural network but over three-address code instead of LLVM IR, illustrating that the choice of IR remains open.

Closest to this work is Transformer-based detection over IoT firmware binaries [31], which shares the target, firmware, and the premise that vulnerability can be predicted from a learned representation of compiled code. It operates directly on opcode sequences, however, which ties the representation to the instruction set the opcodes belong to. Lifting to LLVM IR before embedding is precisely an attempt to remove that dependence, so that one model can span the MIPS, ARM32, PowerPC, and AArch64 images that a real firmware corpus mixes together.

3.5 Existing Datasets

A learning-based method is bounded by the data available to train it, so the corpora in this space deserve examination in their own right. They divide into those that collect real firmware and those that supply code paired with an IR, and the division is the point: no dataset does both.

On the firmware side, LFwC [20] is the most rigorous corpus available. Its contribution is methodological as much as empirical: it distils a framework of three goals, six requirements, and sixteen practical measures for what makes a firmware corpus scientifically sound, applies that framework to forty-four top-tier papers, and finds no common ground among them, with 30% failing to describe deduplication and 52% failing to document how they unpacked. It then demonstrates the framework by building a corpus of 10,913 deduplicated and fully unpacked Linux images, drawn from 14,583 unique packed samples across ten manufacturers, 2,365 devices, and 22 device classes. Its vulnerability ground truth, however, is deliberately modest: mapping devices to public exploit databases yielded some 800 candidate matches, which manual inspection against vendor advisories reduced to 105 samples with possibly applicable remote exploits. That is ground truth for validating an analysis, not a training signal, and the corpus carries no code representation. For ethical rea-

sons the images themselves are not published, only metadata, links, and the scripts needed to re-acquire them.

FirmSec [49] labels at greater scale but at a different granularity. From 11,086 public firmware images and 23,050 private ones, it identifies third-party components at version level through syntactic and control-flow features, deriving 128,757 vulnerability instances from 429 CVEs. The result is an inventory: it records which known-vulnerable components an image ships, rather than judging the image against the vulnerability record of the product it belongs to. An image-level label can only be recovered from it by taking a logical disjunction over components, and the corpus is not paired with any representation on which a model could be trained.

On the representation side the situation is inverted. ComPile [17] assembles 2.4 TB of LLVM IR from production sources across C/C++, Rust, Julia, and Swift, proving that IR corpora can be built at scale, but carries no vulnerability labels and obtains its IR from a compiler with the source in hand. CveBinarySheet [12] provides pre-built binaries for 1,033 CVEs across sixteen components and five architectures at two optimisation levels each; the labelling is exact, because the authors chose which vulnerable version to compile, but that control is also the limitation, since these are laboratory builds, not firmware as vendors ship it. Closest of all is the Rust work in [25], which pairs roughly 2,300 CVE-labelled samples with LLVM IR. The difference from the present work lies in where the IR comes from: those authors hold the source and compile it, whereas firmware ships without source, so our IR has to be recovered from the binary by a lifter. Its labels are also rougher than the CVE identifiers attached to them suggest, since every `.rs` file in the source tree of an affected crate version inherits that version's advisory, so most of the files marked vulnerable contain nothing that the advisory describes. The unit that carries the label is larger than the unit that carries the flaw, which is the same compromise the present work makes at the level of the firmware image, and it appears to be what building a labelled corpus from public advisories costs, whatever the representation.

No public dataset therefore occupies, to our knowledge, the intersection these two lines define: real-world firmware, downloaded as vendors ship it and not compiled by the authors of the dataset; vulnerability labels attached to the image itself and not to its ingredients; and an architecture-independent representation a classifier can consume.

Chapter 4

Methodology

Building a firmware vulnerability detector from scratch means turning raw vendor download pages into labelled, machine-learnable vectors, with no source code at any point. This chapter describes the pipeline that does so, shown in Figure 4.1. It has two parts. The first builds the dataset and comprises eight stages: (i) firmware URL discovery; (ii) CVE retrieval from NVD; (iii) CPE-to-firmware matching; (iv) firmware download; (v) firmware image extraction; (vi) binary lifting into LLVM IR; (vii) IR to embeddings; and (viii) dataset construction. The second part describes the detector placed on top of the dataset: the embedding the firmware vectors are drawn from, and the classifiers trained on them. Similarly, it describes the corpus soundness. In what follows, we describe each stage of the pipeline in more detail.

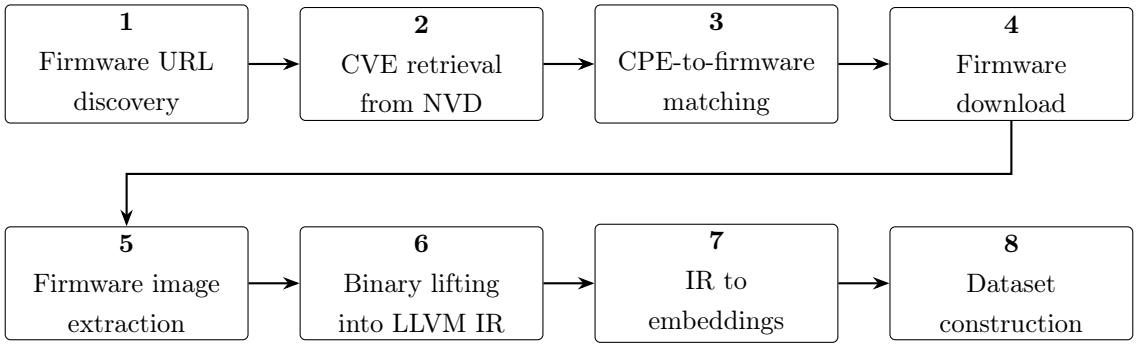

Figure 4.1: Overview of the firmware processing pipeline. Its eight stages run from the discovery of firmware URLs to the construction of the labelled dataset, and correspond one to one with Sections 4.1 to 4.8.

4.1 Firmware URL Discovery

We perform web scraping with the Scrapy framework as our primary data collection tool, discovering firmware download links across multiple online sources. We adopt an exhaustive crawling strategy to maximise link coverage and enumerate as many firmware resources as possible. We consider multiple types of source, including vendor websites, content delivery networks, and public repositories. Since these platforms expose heterogeneous structures and delivery mechanisms, each source needs a dedicated crawler. In particular, we handle dynamically generated pages (e.g., React-based interfaces with client-side rendering) with Selenium, which enables JavaScript execution and DOM interaction.

An independent scraper processes each website, and we store the collected firmware metadata in a MongoDB database. We assign each entry a unique identifier and record fields such as the firmware source URL and a timestamp of when we retrieved the resource.

Many links are not firmware (documentation, drivers, web assets). We filter these with a two-layer scheme that keeps only firmware-looking URLs:

1.- **Layer 1:** URL-only rules. We keep a candidate when a firmware extension appears at the end of the path, and reject it when the path carries blacklisted tokens or extensions.

2.- **Layer 2:** for URLs that pass Layer 1, a single HEAD request (falling back to a ranged GET) inspects the response. Network errors leave a URL pending for retry.

Each candidate keeps a filter status (`pending` → `firmware`/`rejected`).

4.2 CVE Retrieval from NVD

Labelling requires the vulnerability record to be queryable offline: we test thousands of firmware versions against version ranges, which is not a workload the public NVD API is meant to serve. We therefore build the index once from the NVD JSON data feeds and query it locally. The feeds themselves come from the community reconstruction maintained by FKIE-CAD [15], which continues to publish the legacy JSON format that NVD itself retired.

We organise the index as two related tables that mirror the structure of NVD, shown in Figure 4.2. A CVE describes a single vulnerability but typically affects

many products and version ranges, so each CVE fans out into a list of affected-platform entries. The `cves` table holds one row per vulnerability: its identifier, publication and modification dates, description, and Common Vulnerability Scoring System (CVSS) scores (v2 and v3, of which we keep the higher as the labelling score). The `cpe_matches` table holds one row per affected-platform entry (many per CVE, linked back by the CVE identifier), each storing the CPE string, its parsed vendor, product, and version, and the four version-range bounds it may carry (`version_start_inc`/`exc`, `version_end_inc`/`exc`). These bounds later allow a firmware version to be tested against an affected range rather than matched only on a shared product token.

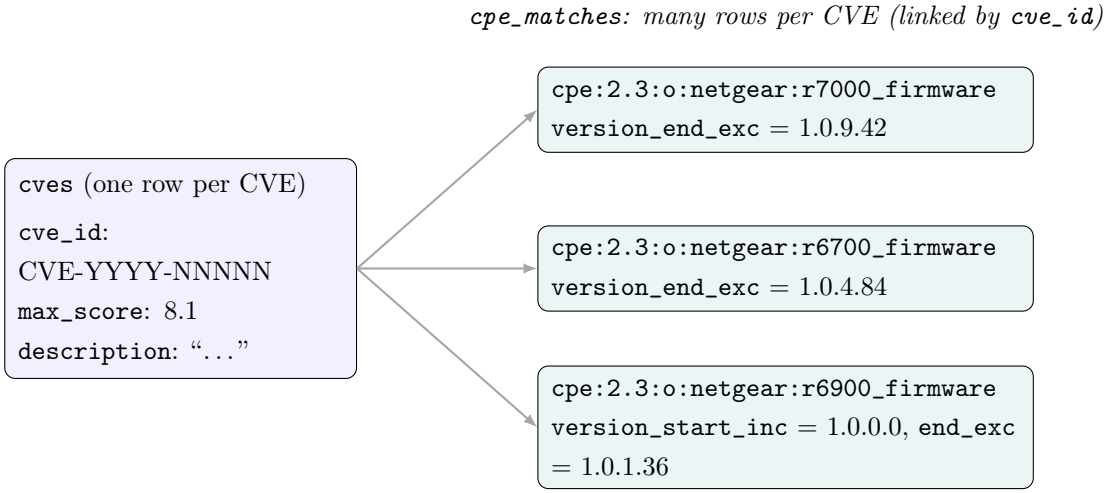

Figure 4.2: Index data model: a single CVE (one `cves` row) expands into many affected-platform entries (`cpe_matches` rows), each carrying its own CPE and version-range bounds. The identifier, score, and version bounds shown are illustrative and do not reproduce a particular NVD record.

4.3 CPE-to-Firmware Matching

Our first version of this stage matched CVEs to firmware URLs by token intersection: we extracted tokens from each CVE's CPE strings, compared them against tokens from the firmware URL, used the size of the intersection as a score, and applied per-vendor score thresholds to decide which matches to keep. That approach labelled most firmware "vulnerable", because the vendor token alone hits hundreds of unrelated CVEs; it encoded vendor identity rather than vulnerability. We replace it with a version-aware method.

We turn each firmware URL into one of three labels (`vuln`, `nonvuln`, or `unknown`) by comparing the firmware's version against the version ranges the CVE records declare as affected. The comparison runs in three steps.

The first step reads the firmware: we parse its vendor, one or more product tokens, and its version string out of the URL, which at this stage is the only description of the firmware we have. The second step retrieves the candidates: we query the index for every CPE belonging to that vendor and product, and cache the results per (vendor, token) so that repeated lookups within a vendor cost nothing. The third step is the version test, where token matching alone no longer suffices. A CPE entry records not only which product a CVE affects but which of its versions, and it does so in one of a few shapes. Most give a lower and an upper bound, and the firmware is affected when its version falls inside that range. Some fix a single version, which the firmware matches only by being exactly it. Others name the product with no version; these confirm that the product is mentioned but not that this particular firmware is affected, and are treated as an "all-versions" match to be handled with caution. Entries describing physical hardware (`cpe:2.3:h:`) carry no software version, and we skip them.

The label then follows from these tests in a fixed order of precedence, so that the strongest available evidence decides. If any CPE places the version inside an affected range, the firmware is `vuln` when the highest associated CVSS score reaches the severity threshold (≥ 6.0) and `nonvuln` otherwise (`version_in_range`). Failing that, we trust an "all-versions" match only when it implicates at most a capped number of CVEs (in particular, 25): within the cap the same CVSS threshold decides, and above it the match is too broad to trust and the firmware becomes `unknown` (`too_broad`). Failing that, a version that lies outside every affected range makes the firmware `nonvuln` (`version_out_of_range`). And when none of these apply, because the product is absent from the index, the version is not parsed, or only hardware CPEs remain, the firmware is left `unknown` (`no_cpe`).

The severity threshold of 6.0 is a pragmatic choice rather than a standard CVSS boundary (the bands break at 4.0, 7.0, and 9.0): we set it to restrict the vulnerable class to CVEs of upper-medium severity and above. The `nonvuln` class therefore also holds firmware whose only known CVEs are below 6.0, not just firmware with none.

Validating on real data exposed three ways the matcher could be fooled, and we closed each. A firmware product token must match a whole word of the CPE

Table 4.1: Entry counts after offline index build and version-range labelling.

Stage / store	**Number of entries**
NVD CVEs (offline index)	290,739
Firmware links (scraped URLs)	27,526
Labelled `vuln`	3,902
Labelled `nonvuln`	696
Labelled `unknown`	13,236

product and not merely a substring, which rejects spurious hits such as `mst` against `mstp` or `r7000` against `r7000p`. Known abbreviations we map to their real CPE products through an alias table, for example `jbundle` to `junos`. Finally, the all-versions cap already described keeps an unbounded match from being trusted when it implicates too many CVEs, sending it to `unknown` instead of `vuln`. The `unknown` label is the point of the whole scheme. It is an honest abstention when a version cannot be tied to a CVE range, whereas the token method forced a label on every URL, so a firmware could end up `nonvuln` either from a real match or merely from an absence of information.

We write the labelled URLs to the per-URL state store (the `firmware_urls` queue), each carrying its label and the basis for it. Table 4.1 reports the resulting counts. Of the 27,526 scraped links, only the 17,834 that the two-layer filter above confirms as `firmware` reach the matcher; the remainder are still `pending` (the Layer-2 probe never resolved, e.g., a network error) or were `rejected`, and never reach CVE matching.

4.4 Firmware Download

To enable large-scale analysis, we retrieve firmware images from the database with a dedicated download stage. To avoid overwhelming system resources and remote servers, we download in fixed-size batches. In each iteration we select a subset of URLs by their download status, taking only rows not previously attempted or not yet successfully downloaded, which makes the stage resumable, and we map each selected URL to a local filename with a hash function (SHA-256).

We delegate the actual downloading to aria2, a high-performance multi-source downloader configured for concurrent downloads, segmented transfers, automatic retries, and resume capabilities. After each batch completes, we verify the results.

Rather than relying on the downloader's exit status alone, we inspect the filesystem directly to confirm the presence and size of each expected file, treating only files above a minimum size threshold as valid and thereby filtering out error pages, truncated transfers, and empty files. We then record verified downloads in the database with their local path and file size.

Before fetching, we HEAD-probe each URL and skip any resource above a size cap (default 1 GB). Multi-gigabyte hits in this dataset are operating-system or virtual-appliance images (e.g., ReadyNAS, vMX/vEOS), not the embedded firmware we can lift. We can also exclude a named vendor from the download and lift stages, which stops a single vendor from dominating the corpus. For instance, we excluded Cisco on these grounds: its classic IOS images are almost uniformly labelled vulnerable (leaving no non-vulnerable samples to balance against) and their platform-code encoding collides with genuine architecture detection during lifting.

4.5 Firmware Image Extraction

The firmware images are stored as raw binary blobs, so we must unpack them to expose the embedded filesystems and executable binaries they contain. We implement this extraction stage with Binwalk v3, a widely adopted tool for recursive signature-based decomposition of binary firmware images.

For each image, we invoke Binwalk inside an isolated Docker container, both to ensure environment reproducibility and to contain any side effects of recursive extraction. The container receives the firmware image and writes all extracted artefacts to a separate directory. We parallelise extraction with a multiprocessing pool, taking the number of workers as a tunable parameter.

We track processed images through an explicit state file and skip already-extracted firmware in subsequent runs, which makes the stage resumable in the event of interruption. Not all firmware images yield successful extractions: encrypted firmware, proprietary container formats, or highly non-standard images may fail to produce any usable filesystem content, and we log these cases for downstream awareness.

4.6 Binary Lifting into LLVM IR

The lifting phase uses two open-source tools, RetDec [5] and Remill [44], each applied to the architectures it handles best. We invoke both through isolated Docker containers to ensure reproducibility across executions.

4.6.1 Lifter Selection

RetDec is our preferred lifter. As a decompiler it produces the more expressive representation and it covers the embedded architectures that dominate firmware (MIPS, PowerPC, ARM32). We intended to lift the whole dataset with RetDec, and use Remill only where RetDec does not work: the x86-64/x86-32 images, on which RetDec aborts with `Missing basic info about input file`, and AArch64, where RetDec failed on the larger binaries of our corpus. The reason is granularity: RetDec decompiles a binary as a single unit, so one function it cannot handle costs the whole binary. Remill lifts each function separately, so the functions it cannot handle are skipped and the remainder are linked into the binary's bitcode. The ELF header selects the back-end (as seen in Table 4.2).

The two paths differ in more than the tool invoked. As described in Section 2.3, an EIR lifter translates the bytes it is handed and decides nothing about which bytes those are, so Remill needs an external front-end to tell it what to lift. Radare2 fills that role here, and we use it both to obtain the function inventory and, in most cases, to read out the function bytes themselves. RetDec needs no such companion, since a decompiler reconstructs the program's shape itself, which is also why we cannot partition its work across functions the way we can on the Remill path.

Table 4.2: Architecture routing across the two lifting back-ends.

Lifter	**Architectures**
RetDec	MIPS, PowerPC, ARM32
Remill	x86-64, x86-32, AArch64

Relying on two different IR paradigms is not ideal for the embedding stage, since DIR and EIR produce structurally different code and the resulting vectors are, strictly speaking, drawn from two different spaces. We accept the inconsistency as a deliberate trade-off, in exchange for the architecture coverage that makes the dataset large enough to be useful. Each binary records the tool it was lifted with, so we can reconstruct consistent single-paradigm subsets later.

4.6.2 RetDec Path

We invoke RetDec as a single decompilation pass. It produces the bitcode early and then spends considerable time on a C-emitting back-end that we do not need. To avoid that cost, we poll for the emitted `.bc` and, once it is present and complete,

stop the container, keeping only the bitcode. A container-level watchdog bounds the whole step at 600 s as a safety net. We validate the bitcode for a minimum size before passing it to the embedding stage.

We encountered a significant practical limitation: a substantial fraction of MIPS binaries in the dataset use non-standard ABIs (specifically, MIPS with N32 or O64 calling conventions) which RetDec 5.0 does not support, returning errors for unsupported target format and architecture combination. We permanently classify these binaries as `skip` in the database.

4.6.3 Remill Path

Remill lifts one function at a time, so the path has to establish what the functions are before any lifting happens. We invoke Radare2 [3] inside a Docker container with the appropriate analysis commands to return a JSON list of function addresses and sizes. We drop two kinds of entry from that list before lifting: functions smaller than 8 bytes, which are stubs or trampolines with no meaningful IR content, and functions belonging to the import table, discarded for the same reason.

We recover the bytes of each surviving function with the Radare2 command `p8`, which reads them independently of the architecture, and fall back to parsing the hex-byte column of an `objdump -d` listing for each function Radare2 leaves without bytes. We pass the bytes individually to `remill-lift`, which produces one `.bc` file per function. Lifting function by function means that a single pathological function can stall the whole binary, a behaviour we observed in practice on functions that were infeasible to lift, so we run two independent guards against this. The first wraps each invocation in a `timeout` inside the container: a function that exceeds the limit is skipped, and the binary receives an embedding built from whatever subset did lift. The second is a watchdog thread that, after 1,200 s, issues `docker kill` to the container and abandons the local client, freeing the worker slot and bounding wall-clock time. A lifter genuinely stuck in uninterruptible sleep is immune to `SIGKILL` and can survive as an orphaned container until its blocking I/O completes, so the watchdog guarantees the pipeline keeps making progress rather than the immediate death of the stuck task.

4.6.4 State Tracking

We record all lifting decisions in a SQLite database with WAL journal mode for concurrent-safe writes. Each binary is assigned one of the following statuses (seen

in Figure 4.3): `pending` (liftable, not yet processed), `lifted` (has `.bc`, awaiting embedding), `done` (has `.bc` and `.npy`), `skip` (unsupported architecture or name filter), `toobig` (input binary exceeds 100 MB), `error` (tool failure), or `timeout` (watchdog fired). The embedding stage applies the same 100 MB limit to the lifted `.bc`, skipping any bitcode above it, since IR2Vec's memory grows with the size of the IR.

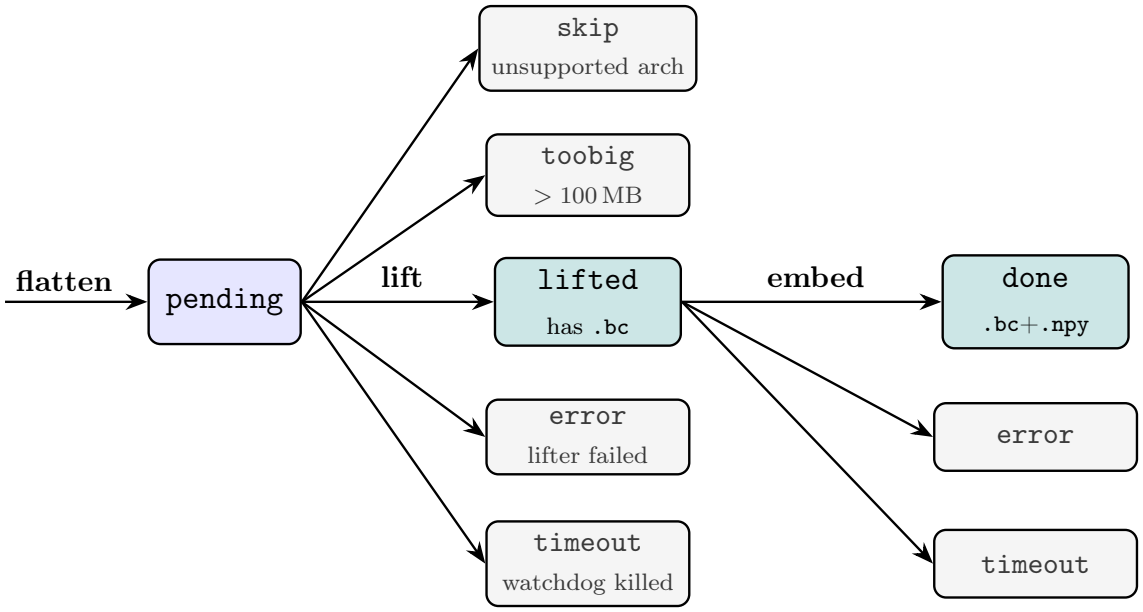

Figure 4.3: Status lifecycle of a binary across the three stages.

4.6.5 Content-Addressed Deduplication of Lifting and Embedding

Firmware images share a large amount of common code, both within a vendor and across vendors. Lifting is the most expensive stage of the pipeline, so lifting these shared binaries once per image would be redundant.

To avoid this, we content-address the lifting. Before lifting a binary, we compute the SHA-256 hash of its contents. We store the lifted bitcode in a cache keyed by that hash, and any later binary whose contents hash to the same value reuses the cached `.bc` instead of being lifted again. We share the cached bitcode by hard link rather than copying it.

Measured on the flattened corpus, the binaries deduplicate roughly four to one (about 571,000 binaries reduce to about 139,000 unique contents), so the cache avoids on the order of three quarters of the lifting work with no effect on the resulting embeddings. The per-binary state database still records every binary independently,

so the firmware-level dataset is unchanged; only the redundant lifting is skipped. We apply the same content-addressed cache to the embedding stage.

4.7 IR to Embeddings

Once we have produced LLVM bitcode files for each binary, we convert them into fixed-dimensional vector representations suitable for machine learning. This stage employs IR2Vec [45], a tool that maps LLVM IR programs to continuous vector spaces by encoding the semantic relationships between opcodes, types, and operands using a vocabulary derived from a pre-trained seed embedding.

IR2Vec operates in two modes: *symbolic* and *flow-aware*. The symbolic mode encodes each instruction based solely on its opcode and operand types, whereas the flow-aware mode additionally propagates information through the program's def-use chains and control-flow graph, producing richer representations at the cost of increased sensitivity to IR irregularities.

The embedding stage iterates over all `.bc` files produced for a firmware image, linked into one module per binary, and runs IR2Vec on each, producing two 300-dimensional vectors per binary: one flow-aware (FA) and one symbolic (SYM). These per-binary vectors are the input to the dataset stage.

4.8 Dataset Construction

The final stage takes the `.npy` files inside a firmware folder and turns them into a dataset. For each firmware, we combine all of its binary embedding vectors by mean aggregation into a single 300-dimensional vector. We discard malformed vectors (wrong dimensionality, containing `NaN`, or all zeros) before aggregation. We apply the procedure independently to the symbolic and flow-aware spaces, producing two parallel datasets.

We take labels from the matching stage, keyed by the firmware hash that also names each embedding directory. We keep only confidently labelled firmware, `vuln` and `nonvuln`, and drop the `unknown` class so the model trains only on trustworthy labels. We discard identical embeddings so they do not leak between the training and test splits. Recording the vendor enables vendor-grouped train/test splits, so we can test the model on vendors unseen in training, which matters given the vendor skew of the corpus.

4.9 Corpus Soundness

A corpus assembled by its own author is a corpus assembled by an interested party, and the reader is entitled to ask on what grounds it should be trusted. The question is not rhetorical: the work in [20] surveyed forty-four firmware papers from the top four security conferences and found that 30% never describe whether they deduplicated, 52% never document how they unpacked, and 50% give no or incomplete information about the known vulnerabilities in their own corpora. Rather than assert soundness, we assess the corpus we built here against the framework proposed in [20], which organises six requirements under the goals of replicability and representativeness. The framework was designed for corpora serving vulnerability research and not model training, so some measures land differently here, but it is the closest thing the field has to an agreed standard. Table 4.3 summarises the outcome, and the discussion that follows does not spare the places where this corpus falls short.

Table 4.3: Assessment of the corpus built in this chapter against the six requirements for sound firmware corpora proposed in [20].

Requirement	**Verdict**	**Basis**
R1 Ground Truth	Exceeds	4,598 automatically labelled firmware, against the 105 manually confirmed in [20]; but derived, not verified
R2 Relevance	Partial	Versions recorded, release dates not
R3 Clean Data	Meets	SHA-256, content-addressed, and embedding-level deduplication; packed and unpacked quantities reported
R4 Rich Meta Data	Partial	Links, hashes, versions, vendors; no dates, models, or device classes
R5 Documentation	Meets	Acquisition, unpacking, and selection documented; no explicit unpacking success criterion
R6 Heterogeneity	Falls short	Three vendors and three architectures in the corpus, with acknowledged skew

Ground truth (R1) is the requirement the field satisfies worst and the one this corpus satisfies best, though not unconditionally. The work in [20] establishes its ground truth by mapping devices to a public exploit database, which produced some 800 candidate matches that manual inspection against vendor advisories reduced to

105 samples carrying possibly applicable remote exploits. We label 4,598 firmware confidently, roughly forty times as many, and do so without human intervention. The comparison favours this work only until the nature of the labels is considered: the 105 in [20] were confirmed by a person, whereas these 4,598 are the output of version-range matching against CPE records and have never been verified against the images they describe. The trade is scale for certainty. Supervised learning needs many labels and tolerates some noise, so the trade is worth making here; a tool being validated needs few labels and no noise, so it would not be. The labelling is still the component of this pipeline with no counterpart in the corpora reviewed in Section 3.5, and its noise is kept bounded by the three-way scheme, which sends doubtful matches to *unknown* instead of guessing.

Relevance (R2) asks whether the samples can be placed in time and in context. Versions are captured, since the labelling cannot function without them, but release dates are not, which leaves the corpus unable to say whether its vulnerable and non-vulnerable samples are drawn from comparable periods, an unexamined confound alongside the vendor and architecture confounds that our results (discussed in Chapter 5) do control for.

Clean data (R3) is met by three independent stages of deduplication: firmware are stored under a SHA-256 of their contents, so byte-identical downloads collapse before extraction; lifting is content-addressed, so a binary recurring across images is lifted once, cutting the work roughly four to one; and byte-identical firmware vectors are collapsed before the dataset is written, removing a further 42%. The funnel of Figure 5.2 and Table 5.1 report both packed and unpacked quantities at every stage, which is the measure on which [20] found 81% of surveyed papers to report precise figures, and which matters here because the attrition is severe enough that reporting only the final figure would misrepresent the corpus entirely.

A further limitation concerns near-duplicate images. The corpus in [20] deliberately excludes open-source firmware distributions, having found that two factory images of the same OpenWrt release, built for different routers from different manufacturers, overlap in 95% of their contents, and noting that earlier work removed 4,020 OpenWrt and DD-WRT samples from Firmadyne's corpus, 42% of it. We apply no such filter, and we have not quantified whether such images are present in the corpus. The deduplication stages would collapse byte-identical cases, but images that overlap in 95% of their contents are not byte-identical. This matters more than raw duplication suggests. Almost everything in a Linux firmware image is shared across devices, so the vulnerability-relevant signal lives in the small fraction

that differs, and a single changed binary can flip an image from safe to vulnerable. Near-duplicates can thus legitimately carry opposite labels, which leaves them neither pure duplicates to discard nor safe to keep: held apart they inflate the apparent size of the corpus and blur the train/test boundary, collapsed together they erase exactly the difference the classifier must learn. This is an open question about the corpus, not a known defect, and settling it requires a content-level comparison between images, which this work does not attempt.

Rich meta data (R4) asks for models, device classes, and firmware types alongside the links, hashes, versions, and vendors that are recorded here. What the corpus holds is what the download URL yields: the vendor, the product token, and the version are parsed from it and stored, and later stages draw on them, as the dataset stage does when it balances the corpus by vendor. Models, device classes, and firmware types cannot be recovered from a URL, so they never enter the pipeline in the first place.

Documentation (R5) is met in its substance: acquisition through Scrapy and Selenium is described in Section 4.1, unpacking through Binwalk in an isolated container in Section 4.5, and the reasoning behind sample selection, including the size caps and the vendor exclusion policy, in Section 4.4. It falls short in one specific and remediable way: [20] verifies unpacking success by checking that the extracted file paths contain common Linux components such as `/bin`, `/etc`, and `/lib`, whereas we treat extraction as successful when Binwalk exits without error and produces output. The weaker criterion admits images that unpacked partially, and the 19% extraction loss reported here is therefore an optimistic figure, not a conservative one.

Heterogeneity (R6) is the plainest shortfall: three vendors against the ten in [20], three architectures against nine, and a vendor distribution skewed heavily towards Netgear, all of it a consequence of the processing-time ceiling rather than of any design decision, and all of it quantified in Figures 5.3 and 5.4.

One more caution about heterogeneity. The routing of Table 4.2 covers six architectures, but only three of them reach the final corpus: no PowerPC, x86-32, or x86-64 image survives to the evaluation set. Support for those three is implemented and exercised by the pipeline, but nothing in our results (discussed in Chapter 5) demonstrates it, and claims about the breadth of the lifting stage should be read as claims about its capability rather than about the corpus it produced here.

The sixteen measures proposed in [20] cover acquisition, unpacking, metadata, and ground truth, but say nothing about the representation a corpus offers a model,

since none of the corpora surveyed there provides one. Lifting and embedding therefore fall outside what the framework can assess.

4.10 The Vulnerability Detector

The pipeline described so far ends with a labelled vector per firmware. What consumes that vector is deliberately the least elaborate part of this work, and we explain both parts of the choice here: the representation it inherits, and the models we place on top of it.

The representation is IR2Vec [45], chosen among the schemes reviewed in Section 3.4: code2vec [1] learns from source syntax and so cannot apply to a binary, while inst2vec [6] and IR2Vec both embed LLVM IR and are therefore reachable from a lifted binary. IR2Vec is preferred because its vocabulary is learned over opcodes, types, and operands rather than over identifiers, which survives the absence of debug symbols, and because it offers the flow-aware mode that propagates information along def-use chains, giving two representations of the same IR to compare instead of one to trust. We adopt the embedding unchanged, so that what gets tested is the lifted IR and the labels, and not an embedding tuned until it worked.

A firmware is not a single binary, and the embedding operates per binary, so we combine the per-binary vectors by mean aggregation into one 300-dimensional vector per image, as described in Section 4.8. Mean aggregation makes no assumption about which binary matters, which is what is not known in advance; its cost is that the contribution of any single binary is diluted by the others. Chapter 6 returns to this.

We place four classifiers on that vector: logistic regression, a linear support vector machine, a support vector machine with an RBF kernel, and a random forest. We compare them to see which performs best, but they also probe the geometry of the embedding space, and for that we arrange them as two opposed pairs. The linear models ask whether the two classes are separated by a hyperplane, that is, whether vulnerability corresponds to a direction in the space. The non-linear models ask whether the classes are separable by a boundary of some more complicated shape: the RBF kernel through distances between points, the random forest through thresholds on individual dimensions and their interactions. Either outcome is informative. If only the linear models work, the signal is a direction and the sample is too small to support anything richer; if only the non-linear models work, there is structure that a direction cannot express.

We attempt nothing more expressive, for the reason set out in Section 2.4: at a few hundred samples in 300 dimensions a high-capacity model has enough freedom to fit its training set and report a number that means nothing. We planned a Transformer-based classifier at the outset, when we expected the corpus to be considerably larger than the one the pipeline finally produced. A model of that capacity becomes appropriate only once there are enough samples to constrain it, so at the scale actually reached we set it aside.

Chapter 5

Discussion of Results

This chapter examines the datasets the pipeline produces and the results of training classifiers on them; it serves as a proof of concept for the practical possibilities the pipeline opens up. Section 5.1 describes those datasets and the controlled subsets we derive from them, Section 5.2 the experimental setup, and Section 5.3 the results themselves.

5.1 Evaluation Datasets

Because dataset generation is resource-intensive, we could not complete the full lifting and embedding for the entire corpus; the full dataset used in this training phase is therefore neither vendor-agnostic nor balanced, and should be read as preliminary rather than representative. To run the experiments, we derive several subsampled datasets from it that achieve class balance and vendor- and architecture-agnostic evaluation.

The CPE version-range matcher classifies every candidate firmware as *vulnerable*, *non-vulnerable*, or *unknown*. Figure 5.1 shows the outcome: the *unknown* class (version unparseable, no CPE match, or a match too broad to trust) dominates at 74%, leaving only 3,902 vulnerable and 696 non-vulnerable. The `nonvuln` label should be read with one caveat in mind for the results that follow: because the severity threshold sends firmware whose only known CVEs score below 6.0 to this class, it mixes firmware with no known vulnerabilities and firmware with only mild ones, so the boundary a classifier must learn is softer than a clean split between vulnerable and safe firmware.

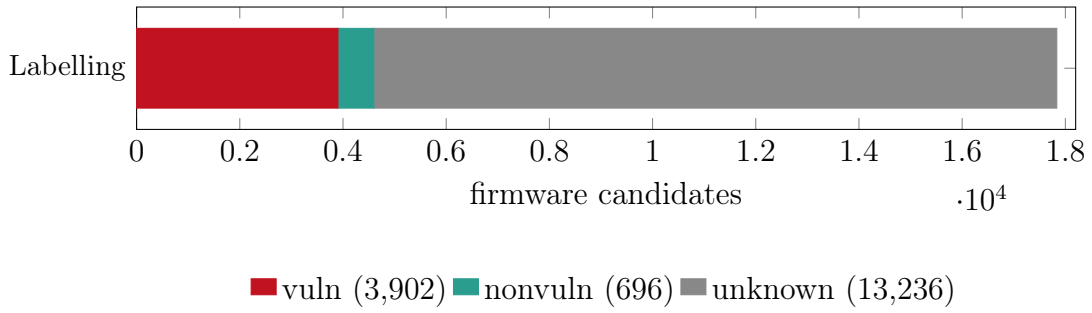

Figure 5.1: Three-way labelling outcome of the version-range matcher over 17,834 candidates.

The build of the actual dataset implements another filter, first in what counts as liftable (the extraction and flattening stages) and then in what the lifters can actually process. Figure 5.2 shows this gap. Extraction discards the images that cannot be unpacked, or that are encrypted, proprietary, or corrupted, removing 19%; flattening then keeps only the file types that can be lifted: the ELF binaries, dropping scripts, resources, and files above the size cap. The firmware that survive both stages (2,024) are the ones counted as liftable. The second filter is what the lifters can actually process: a binary is dropped if its architecture is unsupported or undetectable, if it exceeds the size or function caps, or if the lift times out. A firmware reaches the dataset only if at least one of its binaries lifts and embeds successfully. This is where the gap is widest: of the 2,024 liftable firmware only 353 produce an embedding, a drop of 83%, much of it a processing-time limit (firmware still pending) rather than a structural one. Deduplication finally collapses byte-identical firmware vectors, removing a further 42% (353 → 206), and the dataset stage keeps only the unique firmware that produced *both* embedding types, leaving the 204 evaluation samples.

Part of the gap between the 4,598 confidently labelled firmware and the 2,925 promoted for download reflects the vendor-exclusion policy described in Section 4.4, together with firmware already promoted and downloaded in an earlier pass of the pipeline.

As seen in Table 5.1, the class imbalance in the last stage is real: vuln dominates the dataset. This imbalance is not the only problem: the corpus is also heavily skewed vendor-wise (Figure 5.3) and architecture-wise (Figure 5.4). Such a skew is dangerous because a model can exploit it as a *confound*: a feature correlated with the label that lets a classifier score well without learning anything about vulnerability, for instance a single vendor's compiler fingerprint rather than a flaw itself. To

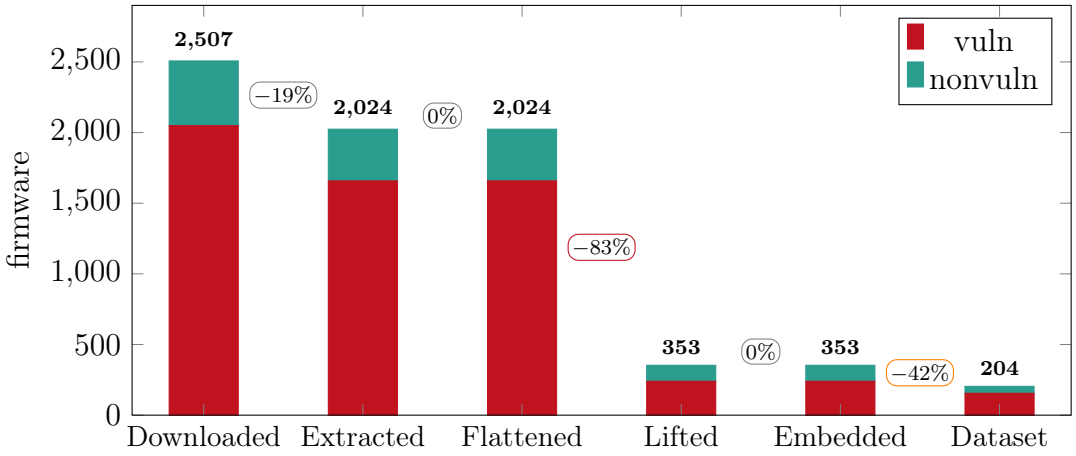

Figure 5.2: Version-matched firmware lost from download to the dataset, split vuln/nonvuln. The annotated −42% is deduplication of byte-identical firmware vectors (353 → 206); the dataset stage then keeps only the firmware with both SYM and FA embeddings (206 → 204).

address these problems, we downsample the evaluation corpus to create smaller, properly balanced datasets. In particular:

1.- **Full matched dataset ($n = 204$).** The complete version-matched evaluation corpus with its natural class distribution (155 vuln / 49 nonvuln), kept imbalanced and used as the baseline.

2.- **Globally balanced ($n = 98$).** Built by downsampling the majority class over the whole corpus so the overall vuln/nonvuln ratio is 1, without imposing balance within any vendor or architecture.

3.- **Per-vendor balanced ($n = 96$).** Built by combining every vendor after downsampling each one so its vuln/nonvuln ratio is 1.

4.- **Per-architecture balanced ($n = 92$).** Built by combining every architecture after downsampling each one so its vuln/nonvuln ratio is 1, so the architecture cannot be exploited as a shortcut for the label.

5.- **Vendor×architecture ($n = 90$).** Built by downsampling within each vendor × architecture group so that the vuln/nonvuln ratio is 1 in every group, decorrelating *both* vendor and architecture from the label at once so neither confound remains.

6.- **Netgear balanced ($n = 66$).** Restricted to the Netgear samples and downsampled to a vuln/nonvuln ratio of 1, holding the vendor fixed.

Table 5.1: Firmware at each pipeline stage (data behind Figures 5.1–5.2). "Labelled (in pipeline)" is the 2,925 matched firmware promoted into the `firmware_urls` download queue, out of the 4,598 confidently labelled firmware (3,902 vuln, 696 nonvuln); the matcher's 13,236 *unknown* never enter the pipeline.

Stage	vuln	nonvuln	Total	Step
Labelled (in pipeline)	2,414	511	2,925	N/A
Downloaded	2,049	458	2,507	−14%
Extracted	1,658	366	2,024	−19%
Flattened	1,658	366	2,024	+0%
Lifted	240	113	353	−83%
Embedded	240	113	353	+0%
Deduplicated	156	50	206	−42%
Dataset (SYM ∩ FA)	155	49	204	−1%

7.- Netgear/ARM32 ($n = 48$). Restricted to Netgear *and* ARM32 and downsampled to a vuln/nonvuln ratio of 1; with both vendor and architecture held fixed, this is the cleanest benchmark.

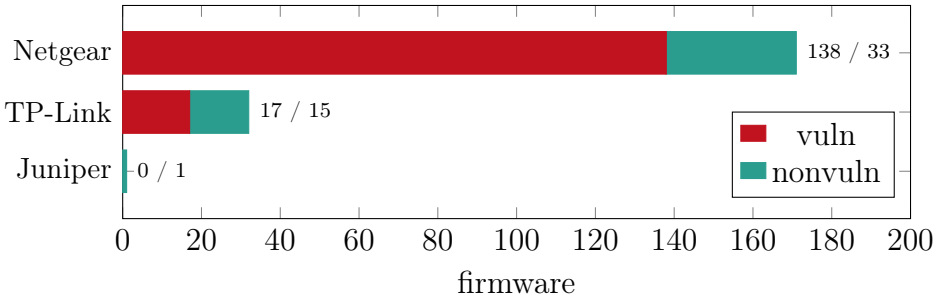

Figure 5.3: Evaluation corpus ($n = 204$) by vendor, split vuln / nonvuln (counts at bar end).

5.2 Experimental Setup

We evaluate the four classifiers described in Section 4.10 under a common protocol, implemented with scikit-learn [33]. We standardise features with a scaler fitted inside each training fold, so that nothing from the held-out fold reaches the model, and we run every experiment under stratified 5-fold cross-validation, reporting metrics as mean ± standard deviation across folds. ROC-AUC is the primary metric, for the reason given in Section 2.4: the corpus is small and unbalanced, and accuracy

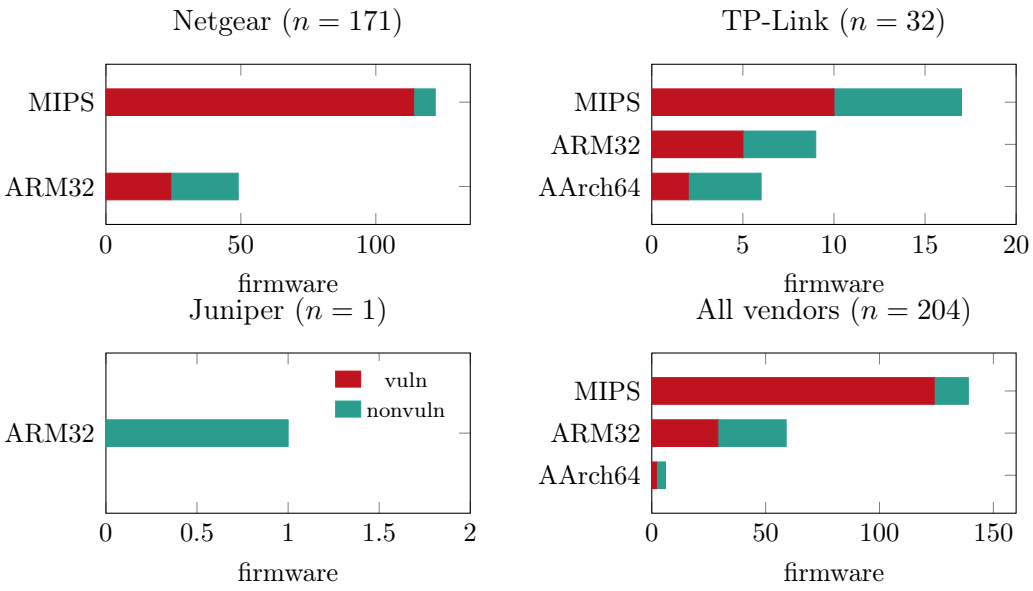

Figure 5.4: Matched dataset by architecture within each vendor and general, split vuln / nonvuln.

rewards a model that simply follows the majority class. We report it alongside accuracy, F1, precision, and recall.

Writing TP and TN for the number of correctly classified vulnerable and non-vulnerable firmware, and FP and FN for the two kinds of mistake, these metrics are defined as follows. Accuracy, $(TP + TN)/(TP + FP + TN + FN)$, is the fraction of firmware classified correctly. Precision, $TP/(TP + FP)$, is the fraction of the firmware flagged as vulnerable that really are. Recall, $TP/(TP + FN)$, is the fraction of the vulnerable firmware that the model finds. F1 is the harmonic mean of the last two, $2\,\mathit{precision} \cdot \mathit{recall}/(\mathit{precision} + \mathit{recall})$, and summarises them in a single number. ROC-AUC is the area under the curve that the true-positive rate traces against the false-positive rate as the decision threshold is swept, and it equals the probability that the model scores a randomly chosen vulnerable firmware above a randomly chosen non-vulnerable one, so 0.5 corresponds to chance and 1.0 to a perfect ranking [14].

We evaluate both embedding spaces produced by the pipeline: SYM and FA. To keep the two spaces directly comparable, we build every dataset from the deduplicated corpus (206 unique firmware) restricted to those that have *both* embeddings, leaving the 204 unique samples (155 vuln / 49 nonvuln) recorded in the final stage of Table 5.1.

5.3 Results

On the full matched dataset a random split looks strong for every model (ROC-AUC 0.77–0.83). A vendor-grouped split, where the test vendor is never seen during training, tells a very different story (Table 5.2): the linear models fall below chance on SYM (0.36 for logistic regression, 0.29 for the linear SVM), meaning the decision directions they learn are vendor-specific and transfer negatively to an unseen vendor. The non-linear models retain part of their performance (0.70–0.73). With only three vendors (one of them contributing a single sample), the grouped estimate is noisy, which is why the controlled subsamples discussed below are the more reliable evidence.

Table 5.2: Full matched dataset ($n = 204$): ROC-AUC under a random stratified split versus a vendor-grouped split.

	SYM		**FA**	
Model	**Random**	**Vendor-grouped**	**Random**	**Vendor-grouped**
logreg	0.827 ± 0.101	0.363	0.785 ± 0.106	0.635
lin_svm	0.811 ± 0.053	0.293	0.790 ± 0.066	0.507
rbf_svm	0.773 ± 0.109	0.728	0.782 ± 0.116	0.725
rand_forest	0.810 ± 0.077	0.724	0.782 ± 0.097	0.704

A direct probe measures how much vendor identity the embeddings expose (Table 5.3): trained to predict the *vendor* rather than the label, every model exceeds 0.66 one-vs-rest ROC-AUC, and the random forest decodes the vendor almost perfectly in both spaces (mean ≈ 0.96). Vendor identity is therefore trivially available to any classifier, so random-split results on the full dataset cannot be read as vulnerability detection on their own.

The same probe applied to *architecture* gives a starker answer (Table 5.4). To keep the question clean, we run it on the Netgear subset alone, so that vendor is held fixed, and restrict it to MIPS and ARM32, which are both routed to RetDec, so that the lifter is held fixed as well. What remains to be decoded is the instruction set and nothing else. Every model separates the two architectures perfectly, at ROC-AUC 1.000, and a multiclass model recovers the architecture with accuracy 0.994 ± 0.012 against a majority-class baseline of 0.713.

Two facts together explain most of what this chapter reports. The architecture is perfectly readable from the embedding, and within Netgear the architecture almost determines the label: its MIPS samples are 114 vulnerable against 8 non-vulnerable,

Table 5.3: Vendor-identification probe (one-vs-rest ROC-AUC, full matched dataset, $n = 204$): how well each model predicts the *vendor* from the embedding. Only vendors with at least 8 samples are probed. The random forest decodes the vendor almost perfectly.

Space	Vendor	Pos	logreg	lin_svm	rbf_svm	rand_forest
SYM	Netgear	171	0.815	0.865	0.819	**0.957**
SYM	TP-Link	32	0.753	0.775	0.791	**0.959**
SYM	mean		0.784	0.820	0.805	**0.958**
FA	Netgear	171	0.735	0.667	0.740	**0.951**
FA	TP-Link	32	0.732	0.661	0.718	**0.960**
FA	mean		0.734	0.664	0.729	**0.956**

Table 5.4: Architecture-identification probe (one-vs-rest ROC-AUC, SYM, Netgear only, $n = 171$): how well each model predicts the *architecture* from the embedding. Vendor and lifter are held fixed, so only the instruction set is left to decode. The two rows are identical because with two classes the one-vs-rest problems differ only in the sign of the label.

Architecture	Pos	logreg	lin_svm	rbf_svm	rand_forest
MIPS	122	**1.000**	**1.000**	**1.000**	**1.000**
ARM32	49	**1.000**	**1.000**	**1.000**	**1.000**

while its ARM32 samples are 24 against 25 (Figure 5.4). A classifier given the full corpus therefore has a shortcut available that has nothing to do with vulnerability: read the instruction set off the vector, and answer *vulnerable* if it is MIPS. The performance that disappears when architecture is decorrelated from the label is the performance that shortcut was providing.

This also settles what the lifting stage does and does not achieve. LLVM IR gives one vocabulary for every architecture, which is what lets a single model consume MIPS and ARM32, and that much the pipeline delivers. It does not give one distribution: the instruction set survives the translation, in whatever RetDec encodes of register widths, calling conventions, and architecture-specific idioms, and IR2Vec preserves it. The embeddings are drawn from an architecture-independent representation without themselves being invariant to architecture, and any claim about the breadth of the pipeline should be read with that distinction in mind. Chapter 6 takes up the question of how the two might be brought closer together.

5.3.1 Controlled Subsampled Datasets

Downsampling to create balanced datasets shrinks the data but yields honest numbers. Table 5.5 reports ROC-AUC for every model on the seven datasets and both embedding spaces, and Figure 5.5 plots the same numbers. Three patterns stand out:

1.- **Performance degrades as vendor and architecture are decorrelated, but stays above chance.** On SYM, logistic regression goes from 0.827 on the full corpus to 0.799 (globally balanced), 0.782 (per-vendor), 0.696 (per-architecture), and 0.667 on the fully agnostic vendor×architecture set; the linear SVM follows the same path, ending at 0.674. The vulnerability signal is therefore weak but present. The last two datasets stand outside this progression: they hold the vendor fixed rather than tightening the vendor and architecture controls. Netgear balanced rises to 0.878, the highest value in the table, because within Netgear the architecture almost determines the label (114/8 on MIPS against 24/25 on ARM32), and balancing the vendor alone leaves that shortcut open; only Netgear/ARM32 closes it, at 0.762.

2.- **The RBF SVM falls below chance and the random forest degrades.** Once architecture can no longer be exploited, the RBF SVM collapses (SYM: 0.391 per-architecture, 0.274 vendor×architecture), while the random forest degrades to 0.58–0.63 (0.598–0.631 on SYM, 0.583–0.617 on FA) without falling below chance.

3.- **SYM dominates FA.** On nearly every controlled dataset the symbolic space outperforms the flow-aware one for the linear models (per-vendor balanced: 0.782 vs 0.631 for logistic regression; vendor×architecture: 0.667 vs 0.553), and FA is only competitive through the non-linear models. We therefore treat SYM as the primary space in the analysis below.

5.3.2 The Cleanest Benchmark: Matched Netgear/ARM32

This set holds vendor and architecture fixed, so nothing but the vulnerability label is left to learn. Table 5.6 gives the full metric panel on SYM. The linear models reach ROC-AUC 0.72–0.76 and precision 0.70, whereas the non-linear models perform substantially worse, with the RBF SVM close to chance and the random forest remaining below the linear models.

Table 5.5: ROC-AUC (mean ± sd over stratified 5-fold CV) for all models on the seven datasets and both embedding spaces. Chance is 0.50. The linear models degrade gracefully as controls tighten and stay above chance; the RBF SVM falls below chance on the symbolic space once architecture is controlled.

Dataset	n	**logreg**	**lin_svm**	**rbf_svm**	**rand_forest**
Symbolic (SYM)					
Full matched	204	0.827 ± 0.101	0.811 ± 0.053	0.773 ± 0.109	0.810 ± 0.077
Globally balanced	98	0.799 ± 0.116	0.745 ± 0.072	0.770 ± 0.112	0.816 ± 0.076
Per-vendor balanced	96	0.782 ± 0.109	0.786 ± 0.085	0.739 ± 0.136	0.778 ± 0.105
Per-architecture balanced	92	0.696 ± 0.080	0.744 ± 0.086	0.391 ± 0.071	0.631 ± 0.082
Vendor×architecture	90	0.667 ± 0.099	0.674 ± 0.088	0.274 ± 0.106	0.598 ± 0.140
Netgear balanced	66	0.878 ± 0.075	0.803 ± 0.077	0.847 ± 0.110	0.848 ± 0.069
Netgear/ARM32	48	0.762 ± 0.111	0.722 ± 0.154	0.554 ± 0.147	0.640 ± 0.036
Flow-aware (FA)					
Full matched	204	0.785 ± 0.106	0.790 ± 0.066	0.782 ± 0.116	0.782 ± 0.097
Globally balanced	98	0.728 ± 0.124	0.684 ± 0.153	0.797 ± 0.136	0.796 ± 0.075
Per-vendor balanced	96	0.631 ± 0.087	0.619 ± 0.133	0.737 ± 0.120	0.749 ± 0.131
Per-architecture balanced	92	0.708 ± 0.120	0.656 ± 0.140	0.561 ± 0.171	0.617 ± 0.106
Vendor×architecture	90	0.553 ± 0.086	0.521 ± 0.138	0.407 ± 0.086	0.583 ± 0.086
Netgear balanced	66	0.695 ± 0.098	0.722 ± 0.109	0.814 ± 0.088	0.844 ± 0.085
Netgear/ARM32	48	0.778 ± 0.177	0.720 ± 0.210	0.682 ± 0.145	0.638 ± 0.144

To verify that the observed performance reflects a relationship between the embeddings and the vulnerability labels, and is not an artefact of chance in a small sample, we performed a label-permutation test. We shuffled the labels randomly 1,000 times while keeping the cross-validation folds and random seed identical to those used in the main sweep, so the *real AUC* column below reproduces the Netgear/ARM32 row of Table 5.5. Destroying the correspondence between features and labels reduced the AUC to approximately 0.5 for every model, as expected. The empirical p-value, computed as the fraction of shuffled datasets whose AUC matched or exceeded the observed value, indicates that only the linear models (logistic regression and linear SVM) achieve statistically significant performance, whereas the RBF SVM and random forest do not reach the conventional 0.05 significance level (Table 5.7).

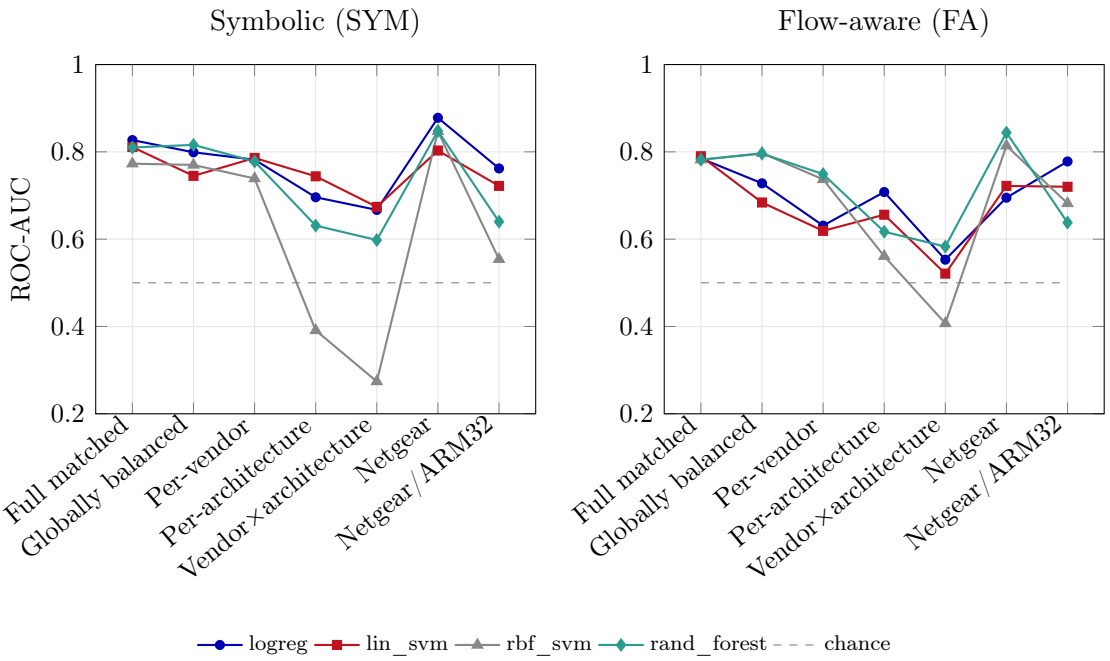

Figure 5.5: ROC-AUC of the four models across the seven datasets, for both embedding spaces. The first five are ordered left to right by increasing control; the last two stand outside that progression, holding the vendor fixed instead of tightening the vendor and architecture controls.

Table 5.6: Matched Netgear/ARM32 (SYM, $n = 48$, 24/24): stratified 5-fold metric panel (mean ± sd).

Model	ROC-AUC	Accuracy	F1	Precision	Recall
logreg	0.762 ± 0.111	0.647 ± 0.081	0.614 ± 0.100	0.700 ± 0.179	0.580 ± 0.133
lin_svm	0.722 ± 0.154	0.627 ± 0.077	0.567 ± 0.152	0.700 ± 0.179	0.540 ± 0.196
rbf_svm	0.554 ± 0.147	0.567 ± 0.103	0.569 ± 0.121	0.564 ± 0.116	0.590 ± 0.169
rand_forest	0.640 ± 0.036	0.651 ± 0.110	0.669 ± 0.141	0.610 ± 0.075	0.760 ± 0.233

Table 5.7: Label-permutation test on the matched Netgear/ARM32 dataset ($n = 48$, 1,000 shuffles).

Embedding	**Model**	**Real AUC**	**Shuffled AUC (mean±sd)**	p**-value**
SYM	logreg	0.762	0.509 ± 0.112	0.0070
	lin_svm	0.722	0.508 ± 0.109	0.0250
	rbf_svm	0.554	0.509 ± 0.114	0.3826
	rand_forest	0.640	0.503 ± 0.113	0.1209
FA	logreg	0.778	0.511 ± 0.108	0.0080
	lin_svm	0.720	0.511 ± 0.106	0.0240
	rbf_svm	0.682	0.510 ± 0.110	0.0509
	rand_forest	0.638	0.511 ± 0.113	0.1459

Chapter 6

Conclusions and Future Work

We set out to design, build, and evaluate a scalable pipeline that takes real-world firmware from acquisition to a vulnerability prediction, without source code and without manual reverse engineering, and we pursued it through the five specific objectives stated in the Introduction. We met all five, the last in reduced form: the Transformer originally scoped for the classifier gave way to classical models once the dataset proved too small to constrain one. With vendor and architecture held fixed, linear models recover a vulnerability signal from the lifted-IR embeddings (ROC-AUC 0.72–0.76 on the symbolic space and 0.72–0.78 on the flow-aware one, all significant under a permutation test), whereas on the full corpus the same embeddings also encode architecture and vendor almost perfectly, so only the tightly controlled subsets give an honest reading. The main limitation is scale: only 204 of the 2,925 promoted firmware reached the final dataset, a processing-time ceiling rather than a structural one.

The pipeline is this work's central contribution. It carries firmware the whole way from a vendor's download page to a labelled vector drawn from a single architecture-independent IR, through discovery and download, three-way CVE/CPE version-range labelling, extraction, dual-lifter coverage, and IR2Vec embedding. Content-addressed deduplication cuts the lifting work by roughly three-quarters. The lifting stage is implemented for six architectures, but only three of them reach the corpus evaluated here, since no PowerPC, x86-32, or x86-64 image survives to the evaluation set, so its breadth should be read as implemented capability rather than demonstrated coverage. No stage is new on its own; the combination is. The dataset the pipeline produces is the second contribution, and the demonstration that the pipeline runs: real-world firmware images labelled at the level of the image and

represented as architecture-independent vectors, a combination no public dataset offers.

The most immediate priority is to finish lifting and embedding the firmware the pipeline has not yet processed, which would enlarge the corpus enough to justify moving beyond the classical models used here, beginning with the Transformer originally scoped for the classifier. A second and larger source of data lies upstream, in the labelling stage: around 13,000 firmware are currently classified as unknown by the NVD CPE matcher and excluded before the pipeline begins, because their version cannot be parsed, they match no CPE, or their match is too broad to be trusted. Recovering all of them would take the labelled corpus, currently 4,598 firmware, to close to four times that size, and even a modest share of that would widen the pipeline's input substantially. Moving deduplication ahead of the lifting stage would remove redundant firmware earlier and cheapen the whole run.

Two design choices also deserve revisiting. Mean-aggregating the binary embeddings into one firmware vector weights every binary equally and can wash out a signal carried by a single component; a more principled alternative would be to fine-tune IR2Vec with a representation-learning objective that maps bitcode into a space in which the vulnerability-relevant features are linearly separable and confounds such as vendor and architecture are factored out, which would let the simple average preserve the signal. And because different lifters emit different IR styles, building separate, lifter-specific datasets (Remill versus RetDec) and comparing them would isolate the effect of mixing paradigms, in the same spirit as the vendor and architecture controls introduced in Chapter 5.

Acknowledgements

This work is part of the project UPV/EHU Cybersecurity Chair - INCIBE (C08823), funded by the European Union NextGeneration-EU, Recovery, Transformation and Resilience Plan, through INCIBE.

References

[1] Alon, U., Zilberstein, M., Levy, O., Yahav, E.: code2vec: Learning Distributed Representations of Code. Proceedings of the ACM on Programming Languages **3**(POPL), 1–29 (2019)

[2] Altinay, A., Nash, J., Kroes, T., Rajasekaran, P., Zhou, D., Dabrowski, A., Gens, D., Na, Y., Volckaert, S., Giuffrida, C., Bos, H., Franz, M.: BinRec: Dynamic Binary Lifting and Recompilation. In: EuroSys. pp. 36:1–36:16 (2020)

[3] Alvarez, S., the Radare2 Community: Radare2: Open Source Framework for Reverse Engineering and Binary Analysis. `https://github.com/radareorg/radare2` (2026), accessed: 2026-06-12

[4] Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J.A., Invernizzi, L., Kallitsis, M., Kumar, D., Lever, C., Ma, Z., Mason, J., Menscher, D., Seaman, C., Sullivan, N., Thomas, K., Zhou, Y.: Understanding the Mirai Botnet. In: 26th USENIX Security Symposium (USENIX Security 17). pp. 1093–1110. USENIX Association (2017)

[5] Avast Software: RetDec: A Retargetable Machine-Code Decompiler. `https://github.com/avast/retdec` (2022), accessed: 2026-06-12

[6] Ben-Nun, T., Jakobovits, A.S., Hoefler, T.: Neural Code Comprehension: A Learnable Representation of Code Semantics. In: Advances in Neural Information Processing Systems 31 (NeurIPS). pp. 3589–3601. Curran Associates, Inc. (2018)

[7] Brahmakshatriya, A., Kedia, P., McKee, D.P., Garg, D., Lal, A., Rastogi, A., Nemati, H., Panda, A., Bhatu, P.: ConfLLVM: A Compiler for Enforcing Data Confidentiality in Low-Level Code. In: EuroSys. pp. 4:1–4:15 (2019)

[8] Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: Proceedings

of the 8th USENIX Conference on Operating Systems Design and Implementation. pp. 209–224. OSDI'08, USENIX Association, USA (2008)

[9] Carr, S.A., Payer, M.: DataShield: Configurable Data Confidentiality and Integrity. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security. pp. 193–204. ASIA CCS '17, Association for Computing Machinery, New York, NY, USA (2017)

[10] Cheikes, B.A., Waltermire, D., Scarfone, K.: Common Platform Enumeration: Naming Specification Version 2.3. Tech. Rep. NIST Interagency Report 7695, National Institute of Standards and Technology (2011)

[11] Chen, D., Egele, M., Woo, M., Brumley, D.: Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In: Network and Distributed System Security Symposium (NDSS) (2016)

[12] Chen, L.: CveBinarySheet: A Comprehensive Pre-Built Binaries Database for IoT Vulnerability Analysis. arXiv preprint arXiv:2501.08840 (2025)

[13] Costin, A., Zaddach, J., Francillon, A., Balzarotti, D.: A Large-Scale Analysis of the Security of Embedded Firmwares. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 95–110. USENIX Association (2014)

[14] Fawcett, T.: An Introduction to ROC Analysis. Pattern Recognition Letters **27**(8), 861–874 (2006)

[15] Fraunhofer FKIE, Cyber Analysis and Defense: nvd-json-data-feeds: Community Reconstruction of the Legacy JSON NVD Data Feeds (2023), `https://github.com/fkie-cad/nvd-json-data-feeds`

[16] Gaidis, A.J., Moreira, J., Sun, K., Milburn, A., Atlidakis, V., Kemerlis, V.P.: FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking. In: Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses. pp. 527–546. RAID '23, Association for Computing Machinery, New York, NY, USA (2023)

[17] Grossman, A., Paehler, L., Parasyris, K., Ben-Nun, T., Hegna, J., Moses, W.S., Monsalve Diaz, J.M., Trofin, M., Doerfert, J.: ComPile: A Large IR Dataset from Production Sources. Journal of Data-centric Machine Learning Research (2024)

[18] Haller, I., Jeon, Y., Peng, H., Payer, M., Giuffrida, C., Bos, H., van der Kouwe, E.: TypeSan: Practical Type Confusion Detection. In: Proceedings of the 2016

ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 517–528. Association for Computing Machinery (2016)

[19] Heffner, C.: Binwalk: Firmware Analysis Tool. `https://github.com/ReFirmLabs/binwalk` (2024), accessed: 2026-07-16

[20] Helmke, R., Padilla, E., Aschenbruck, N.: Mens Sana In Corpore Sano: Sound Firmware Corpora for Vulnerability Research. In: Proceedings of the Network and Distributed System Security Symposium (NDSS). The Internet Society, San Diego, CA, USA (2025)

[21] Kim, M., Kim, D., Kim, E., Kim, S., Jang, Y., Kim, Y.: FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis. In: Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC). pp. 733–745. Association for Computing Machinery (2020)

[22] Küchler, A., Banse, C.: Representing LLVM-IR in a Code Property Graph. In: Information Security Conference (ISC). Lecture Notes in Computer Science, vol. 13640, pp. 360–380. Springer (2022)

[23] Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., Song, D.: Code-Pointer Integrity. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI). pp. 147–163 (2014)

[24] Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In: Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 75–88. IEEE Computer Society, San Jose, CA, USA (2004)

[25] Lee, Y., Boshra, S.J., Yang, J., Cao, Z., Liang, G.: Machine Learning-Based Vulnerability Detection in Rust Code Using LLVM IR and Transformer Model. Machine Learning and Knowledge Extraction **7**(3), 79 (2025)

[26] Li, B., Ma, R., Wang, X., Wang, X., He, J.: DepTaint: A Static Taint Analysis Method Based on Program Dependence. In: Proceedings of the 2020 4th International Conference on Management Engineering, Software Engineering and Service Sciences. pp. 34–41. ICMSS 2020, Association for Computing Machinery, New York, NY, USA (2020)

[27] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y.: VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In: Proceed-

ings of the 25th Annual Network and Distributed System Security Symposium (NDSS). The Internet Society (2018)

[28] Liu, Z., Yuan, Y., Wang, S., Bao, Y.: SoK: Demystifying Binary Lifters Through the Lens of Downstream Applications. In: 2022 IEEE Symposium on Security and Privacy (SP). pp. 1100–1119 (2022)

[29] Mahyari, A.A.: Harnessing the Power of LLMs in Source Code Vulnerability Detection. In: MILCOM 2024 – 2024 IEEE Military Communications Conference (MILCOM). pp. 251–256. IEEE (2024)

[30] McCully, G.A., Hastings, J.D., Xu, S., Fortier, A.: Bi-Directional Transformers vs. word2vec: Discovering Vulnerabilities in Lifted Compiled Code. In: 2024 IEEE Cyber Awareness and Research Symposium (CARS). pp. 1–8. IEEE (2024)

[31] Nandish, M., Kumar, J., Mohan, H.G., Manoj Kumar, M.V.: Transformer-Based Vulnerability Detection in IoT Firmware Binaries Using Opcode Sequences. IEEE Access **13**, 124250–124263 (2025)

[32] National Institute of Standards and Technology: National Vulnerability Database (2026), `https://nvd.nist.gov/`

[33] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, É.: Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)

[34] Schubert, P.D., Hermann, B., Bodden, E.: PhASAR: An Inter-Procedural Static Analysis Framework for C/C++. In: TACAS. pp. 393–410 (2019)

[35] Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: AddressSanitizer: A Fast Address Sanity Checker. In: 2012 USENIX Annual Technical Conference (USENIX ATC 12). pp. 309–318. USENIX Association, Boston, MA (2012), `https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany`

[36] Shimchik, N., Ignatyev, V.: Vulnerabilities Detection via Static Taint Analysis. Proceedings of the Institute for System Programming of the RAS **31**(3), 177–190 (2019)

[37] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In: IEEE Symposium on Security and Privacy. pp. 138–157 (2016)

[38] Silva, G.Q., Pereira, F.M.Q.: Static Detection of Address Leaks. In: Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg). pp. 225–238. Sociedade Brasileira de Computação (2011)

[39] Situ, L., Wang, L., Liu, Y., Mao, B., Li, X.: Vanguard: Detecting Missing Checks for Prognosing Potential Vulnerabilities. In: Proceedings of the 10th Asia-Pacific Symposium on Internetware. pp. 1–10. Association for Computing Machinery (2018)

[40] Song, D., Zhao, J., Burke, M., Sbirlea, D., Wallach, D., Sarkar, V.: Finding Tizen Security Bugs through Whole-System Static Analysis. arXiv preprint arXiv:1504.05967 (2015)

[41] Sui, Y., Xue, J.: SVF: Interprocedural Static Value-Flow Analysis in LLVM. In: Proceedings of the 25th International Conference on Compiler Construction (CC). pp. 265–266. Association for Computing Machinery (2016)

[42] Szekeres, L., Payer, M., Wei, T., Song, D.: SoK: Eternal War in Memory. In: IEEE Symposium on Security and Privacy. pp. 48–62 (2013)

[43] Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., Pike, G.: Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In: USENIX Security. pp. 941–955 (2014)

[44] Trail of Bits: Remill: A Static Binary Translator for LLVM. `https://github.com/lifting-bits/remill` (2024), accessed: 2026-06-12

[45] VenkataKeerthy, S., Aggarwal, R., Jain, S., Desarkar, M.S., Upadrasta, R., Srikant, Y.N.: IR2VEC: LLVM IR Based Scalable Program Embeddings. ACM Transactions on Architecture and Code Optimization **17**(4), 32 (2020)

[46] Xu, L., Xu, M., Li, F., Huo, W.: ELAID: Detecting Integer-Overflow-to-Buffer-Overflow Vulnerabilities by Light-Weight and Accurate Static Analysis. Cybersecurity **3**(1), 18 (2020)

[47] Yu, R., Del Nin, F., Zhang, Y., Huang, S., Kaliyar, P., Zakto, S., Conti, M., Portokalidis, G., Xu, J.: Building Embedded Systems Like It's 1996. In: Pro-

ceedings of the Network and Distributed System Security Symposium (NDSS). The Internet Society (2022)

[48] Zeng, Q., Xiong, D., Wu, Z., Qian, K., Wang, Y., Su, Y.: TACSan: Enhancing Vulnerability Detection with Graph Neural Network. Electronics **13**(19), 3813 (2024)

[49] Zhao, B., Ji, S., Xu, J., Tian, Y., Wei, Q., Wang, Q., Lyu, C., Zhang, X., Lin, C., Wu, J., Beyah, R.: A Large-Scale Empirical Analysis of the Vulnerabilities Introduced by Third-Party Components in IoT Firmware. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA). pp. 442–454. ACM (2022)

[50] Zhou, J., Criswell, J., Hicks, M.: Fat Pointers for Temporal Memory Safety of C. Proceedings of the ACM on Programming Languages **7**(OOPSLA1), 316–347 (2023)

[51] Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In: Advances in Neural Information Processing Systems 32 (NeurIPS). pp. 10197–10207. Curran Associates, Inc. (2019)

Appendix A

Master's Thesis Progress Tracking

This appendix records the project-management view of the thesis: the initial task breakdown and schedule, and a comparison of the planned timeline against what was actually achieved. The project spanned from October 2025 to August 2026 and was carried out by the author of this work, acting as a single researcher working part-time, with access to the Priscilla HPC cluster (UPV/EHU) for the computationally intensive lifting and embedding stages.

Table A.1 lists the identified tasks, their planned durations and dependencies, and Figure A.1 shows the corresponding nine-month schedule (October 2025 to June 2026).

Figure A.2 shows the actual execution timeline, and Table A.2 compares the planned and actual completion dates for each task, as of August 2026.

Table A.1: Task identification and duration (initial planning).

ID	Task	Weeks	Dependencies
T1	State of the art & paper review	4	none
T2	Firmware web scraping	8	none
T3	NVD/CVE retrieval	4	none
T4	CPE–firmware matching & tuning	4	T2, T3
T5	Server setup (Priscilla)	2	none
T6	Binwalk extraction	4	T2, T5
T7	Binary lifting to LLVM IR	8	T6
T8	IR2Vec embedding generation	6	T7
T9	Dataset assembly	4	T8
T10	Transformer model design	4	T9
T11	Model training	6	T10
T12	Evaluation & benchmarking	4	T11
T13	Methodology chapters (writing)	8	T6–T9
T14	Conclusions & final editing	2	T12, T13

Table A.2: Planned vs. actual schedule comparison (Aug 2026).

ID	Task	Planned end	Actual end / status	Dev.
T1	State of the art	Oct 2025	Oct 2025	0
T2	Firmware scraping	Nov 2025	Dec 2025	+1 month
T3	CVE retrieval	Dec 2025	Jan 2026	+1 month
T4	CPE matching & tuning	Dec 2025	Jan 2026	+1 month
T5	Server setup (Priscilla)	Nov 2025	Feb 2026	+3 months
T6	Binwalk extraction	Dec 2025	Feb 2026	+2 months
T7	Binary lifting	Feb 2026	Jul 2026	**+5 months**
T8	IR2Vec embeddings	Mar 2026	Jul 2026	**+4 months**
T9	Dataset assembly	Apr 2026	Jul 2026	**+3 months**
T10	Model design	Apr 2026	Jun 2026 (rescoped)	+2 months
T11	Training	May 2026	Jul 2026	+2 months
T12	Evaluation	Jun 2026	Jul 2026	+1 month
T13	Methodology (writing)	May 2026	Aug 2026	**+3 months**
T14	Conclusions & editing	Jun 2026	Aug 2026	+2 months

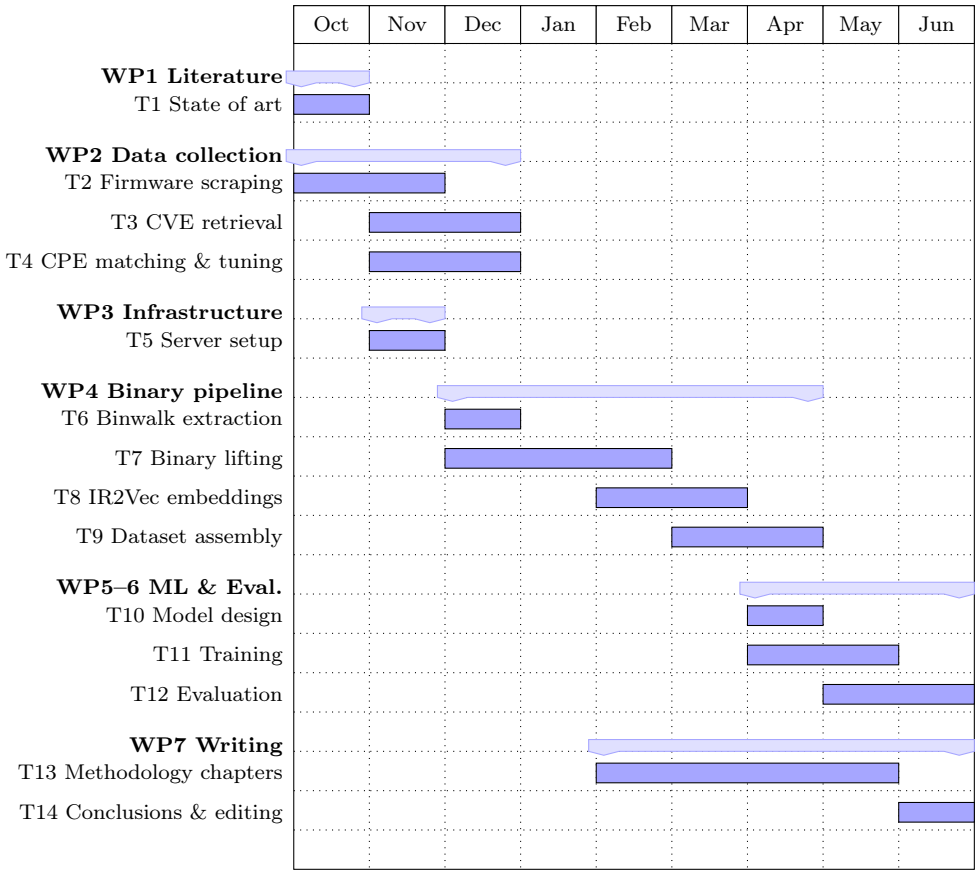

Figure A.1: Original project planning (Oct 2025 – Jun 2026).

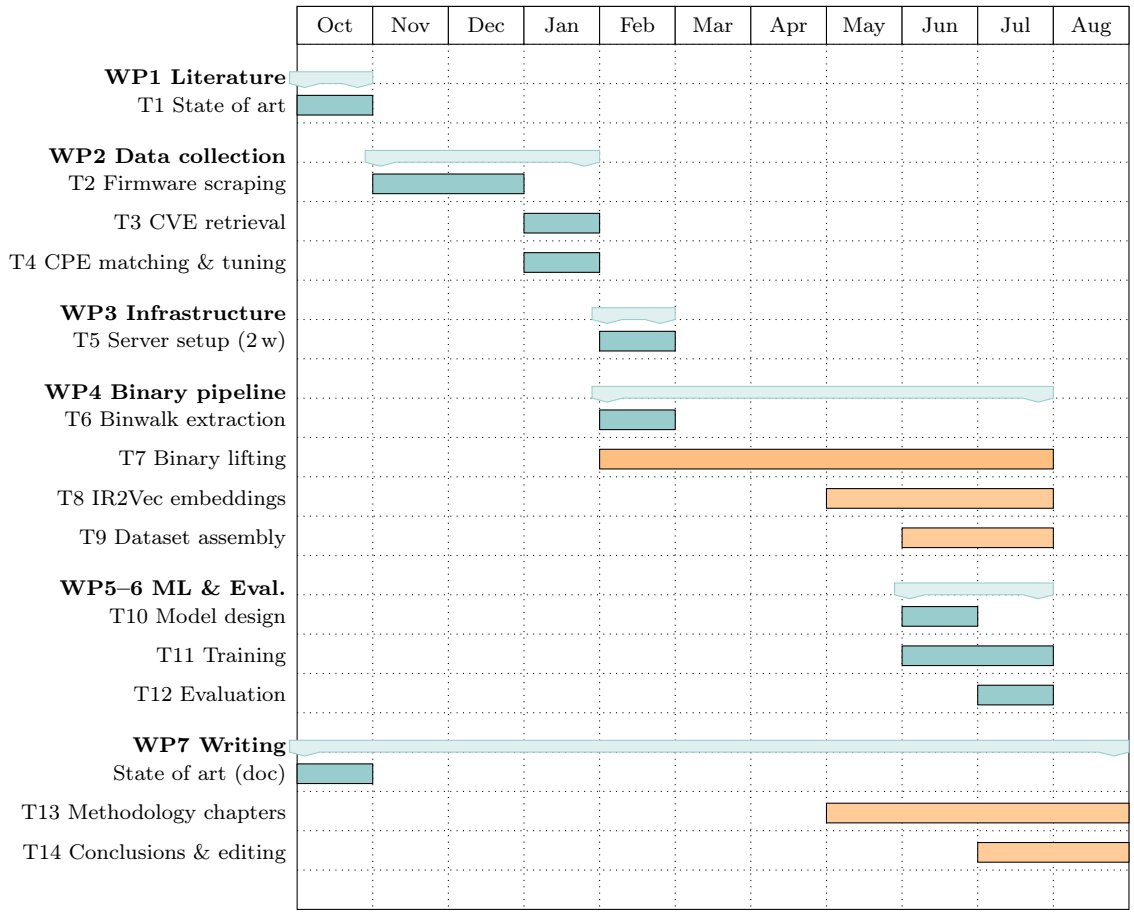

Figure A.2: Real project planning (Oct 2025 – Aug 2026). Orange bars represent tasks that finished later than planned.