



Magika: AI-Powered Content-Type Detection

Yanick Fratantonio
Google

Luca Invernizzi
Google

Loua Farah
Google

Kurt Thomas
Google

Marina Zhang
Google

Ange Albertini
Google

Francois Galilee
Google

Giancarlo Metitieri
Google

Julien Cretin
Google

Alex Petit-Bianco
Google

David Tao
Google

Elie Bursztein
Google

Abstract—The task of content-type detection—which entails identifying the data encoded in an arbitrary byte sequence—is critical for operating systems, development, reverse engineering environments, and a variety of security applications. In this paper, we introduce MAGIKA, a novel AI-powered content-type detection tool. Under the hood, MAGIKA employs a deep learning model that can execute on a single CPU with just 1MB of memory to store the model’s weights. We show that MAGIKA achieves an average F1 score of 99% across over a hundred content types and a test set of more than 1M files, outperforming all existing content-type detection tools today. To foster adoption and improvements, we open source MAGIKA under an Apache 2 license on GitHub and we make our model and training pipeline publicly available. Our tool has already seen adoption by Gmail and Google Drive for attachment scanning, by VirusTotal to aid with malware analysis, and by prominent open-source projects such as Apache Tika. While this paper focuses on the initial version, MAGIKA continues to evolve with support for over 200 content types now available. The latest developments can be found at <https://github.com/google/magika>.

I. INTRODUCTION

Content-type detection is a fundamental computing task that identifies the data encoded in an arbitrary byte sequence. This allows an application to distinguish source code (C++, Python, etc.), media (PDF, JPG, etc.), binaries (EXE, ELF), and a variety of other file formats. As such, content-type detection impacts a wide range of downstream use-cases including software development tools, security tools, browsers, and media players. For example, development environments (e.g., Visual Studio Code, VS Code for short) rely on content-type detection to decide which syntax highlighters and plugins to use. Security applications rely on content-type detection for policy enforcement (e.g., email providers prohibiting executable attachments), for forensic analysis and recovery, and for routing samples to the most capable content-type-specific threat analysis scanners (e.g., anti-viruses often have specialized scanners for binaries, scripts, or PDFs; these scanners are too resource-intensive to be run on all samples).

The need for content-type detection stems from byte sequences lacking an intrinsic, trustworthy indicator of their underlying file format. While some approaches to built-in indicators exist—such as file extensions, MIME types, magic bytes, and metadata—they can easily be omitted when a file’s

	LibMagic	Magika
<pre>let myHeading = document.querySelector("h1"); function setUsername() { const myName = prompt("Please enter your name."); if (!myName) { setUsername(); } else { localStorage.setItem("name", myName); myHeading.textContent = `\${myName} is cool`; } }</pre>	ASCII text ❌	JavaScript ✅
<pre>__BUNDLE_START_TIME__ = "1"; function hi() { console.log("hi")}</pre>	ASCII text ❌	JavaScript ✅
<pre>__BUNDLE_START_TIME__ = "1"; ← Spaces have been removed function hi() { console.log("hi")}</pre>	JavaScript ✅	JavaScript ✅

Fig. 1: Example of the frailty of signature-based content-type detection when applied to distinct code snippets taken from “JavaScript Basics” of the Mozilla’s MDN [8]. `file`—which relies on regular expressions for content-type detection—imprecisely labels each snippet as ASCII text unless the spaces around the “=” sign are removed. As we show, our proposed content-type detector MAGIKA overcomes these robustness limitations.

content is copied or transmitted, or otherwise spoofed (e.g., to evade security systems that prohibit certain content types).

The task of content-type detection was first tackled with the `file` command in Bell Labs UNIX over five decades ago [1]. Since then, developers have created a variety of tools, the vast majority of which rely on manually-written signatures (e.g., rules or regular expressions) that are updated as new file types and versions emerge. Examples include the latest version of `file` [2] (powered by `libmagic` [3]); `exiftool` [4]; and `trid` [5]. Unfortunately, signature-based approaches are frail: a single whitespace change (or other subtle byte sequence modification) can result in inaccurate content-type detection as shown in Figure 1. Beyond signatures, `guesslang` [6] employs a simple TensorFlow text model to distinguish source code content types, which is integrated into VS Code [7].

In this paper, we discuss the design and implementation of a novel AI-powered content-type detection tool called MAGIKA. Built using a deep learning model, MAGIKA automatically infers the content type of a byte sequence without any reliance on human expertise with respect to the intricacies of different file formats. The model takes as input three sequences of 512 bytes drawn from the beginning, middle, and end of a file’s content; automatically identifies patterns unique to content types; and outputs the most probable content type. We trained

MAGIKA to identify 113 canonical content types using 24M file samples sourced from GitHub [9] and VirusTotal [10]. We find that MAGIKA achieves an average F1 score of 99% on a holdout test dataset of 1.2M samples.

For comparison, we benchmark MAGIKA against four existing tools—`file`, `exiftool`, `trid`, and `guesslang`—to compare their accuracy, speed, and overall resource usage using the same test dataset of 1.2M samples. We find that MAGIKA outperforms every existing tool, with a 4% F1 gain over the best tool for binary content types and a 22% F1 gain over the best tool for text content types, and 12% F1 gain overall. The F1 gain increases, respectively, to 9%, 47% and 27% when considering all content types in our benchmark, instead of just those supported by existing tools. For bulk inferences, MAGIKA requires 5.77ms to yield a decision per file, outperforming all other existing tools except for `file` (0.75ms). Likewise, MAGIKA requires only 1MB of memory to represent model weights and achieves the aforementioned performance with just a single CPU; no GPU is required.

To foster wider adoption and accelerate improvements, we open source MAGIKA under an Apache 2 license on GitHub [11]. In terms of real-world deployments, MAGIKA has already been integrated within Gmail and Google Drive to determine the content type for hundreds of billions of files every week [12], with VirusTotal to assist with malware analysis, with prominent open-source projects such as Apache Tika [13], and we are in discussions to integrate MAGIKA with VS Code as a replacement for `guesslang`. This positive reception reflects the real-world improvements MAGIKA provides over existing software engineering toolsets.

We note that this paper focuses on the initial iteration of our approach, and that MAGIKA continues to evolve with support for over 200 content types now available. The latest developments can be found at <https://github.com/google/magika>.

II. RELATED WORK

Content-type detection is a well-studied, longstanding problem. Before presenting our new approach, we discuss both traditional approaches used by existing tools as well as research proposals to integrate machine learning into content-type detection.

A. Traditional approaches to content-type detection

The conventional approach to content-type detection relies on manually-crafted signatures (e.g., regular expressions). The most prominent tool employing signatures is `file` [2], which is regarded as the default command-line utility for detecting content types. `file` provides support for binary file formats (which are particularly amenable to a signature-based approach), including highly-specialized formats (e.g., file formats employed in specific video games for storing saved games, textures, etc.). `file` also possesses a range of signatures for textual content types (e.g., C, Java, Python). Anecdotal evidence suggests that `file` exhibits higher accuracy for binary formats than textual ones, but its performance has not been systematically evaluated prior to our study.

By default, `file` outputs a textual description of the detected content type, and a number of additional metadata. For example, when processing a PDF file, `file` will output its content type: PDF, but also the PDF version number, how many pages it has, and so forth. Since this output is challenging to parse programmatically, `file` also supports outputting a MIME type. MIME types are easier to parse and more codified than human descriptions, and thus, it may be more appropriate when integrating `file` in automated pipelines. However, dealing with MIME types has its set of challenges, which we document in Section IV.

Other popular tools include `trid` [5] and `exiftool` [4]. The latter was originally designed to detect image content types exclusively, but it has since expanded its scope. Apart from these general purpose content-type detection tools, there are a number of specialized tools that trade breadth for depth on a specific domain. For instance, `PEid` [14] is a tool dedicated to detecting PE packers, whereas `Detect-It-Easy` [15] facilitates fine-grained inspection of PE files and other executable formats. We omit these special-purpose tools from our comparative analysis due to their limited support of the content types under evaluation.

Lastly, numerous libraries re-implement a subset of existing features to offer content-type detection to various programming languages, often aiming to be free of dependencies to simplify integration. Examples include: `mime-types` [16], a library specializing in a limited number of content types, particularly for JavaScript clients; `PolyFile` [17], a pure-Python implementation of `libmagic`; `filetype` [18], a GoLang library for file type identification; `filetype.py` [19], a Python library for file type identification; and `file-type` [20], another (binary) file format detection tool for the JavaScript ecosystem. Given the derivative nature of many of these tools, we omit them from our comparative evaluation as well.

B. Machine-learning approaches to content-type detection

Recently, researchers have proposed a number of approaches that leverage machine learning to replace manual signatures. Scaedan [21] proposes using support vector machines to model features based on unigram and bigram frequencies. Fitzgerald et al. [22] suggest natural language processing techniques for file fragment classification, whereas Wang et al. [23] propose using sparse coding and unsupervised learning to classify file fragments in the context of memory forensics.

More recent approaches employ appropriately state-of-the-art techniques such as recurrent and convolutional neural networks [24], [25]. These proposals have not found wide adoption, possibly due to their relatively low overall accuracy (ranging from 53% to 84%), limited support for different content types (between 18 and 75, primarily binary file formats), and competition from existing tools, which already provide adequate support for binary file formats. As such, we treat them as out of scope for our comparative evaluation.

The most recent and successful approach is `guesslang` [6], an open-source tool that detects

programming languages from a snippet of source code. `guesslang` focuses exclusively on 54 textual content types and employs a Wide & Deep TensorFlow model [26] for the detection task [27]. Notably, VS Code utilizes `guesslang` to infer the programming language when a user creates a new file without specifying a file extension [7]. While the original `guesslang` is currently unmaintained (last commit in September 2021) and relies on deprecated TensorFlow abstractions, VS Code developers maintain a Node.js client that facilitates the testing of the underlying model [28].

III. DATASET

We curate a novel dataset of 26M files drawn from a diverse set of 113 content types to act both as a benchmark for evaluating existing content-type detectors and to train our deep learning content-type detector. We describe the origin of this data, our validation steps, and limitations with our approach.

Dataset sources. In real-world settings, the distribution of content types varies from environment to environment. For example, social media uploads are dominated by videos and images, whereas digital-signing platforms predominantly receive PDFs. Rather than tailor our dataset to any single environment, we incorporate a sample of content types that may be present across a variety of environments including source code, executables, documents, media, archives, and more. We build a stratified dataset, in which every type has equal representation. When reporting performance metrics throughout this work, we break our results down per content type to account for this sampling strategy. In practice, any deployment environment can estimate the efficacy of `MAGIKA` by taking a weighted average of our reported results, with weighting taking into account the frequency of content types within a specific setting.

To gather our stratified samples, we identified GitHub [9] (a popular source code development platform) and VirusTotal [10] (a popular binary and file scanning platform) as two promising sources, as each covers a complementary set of content types: GitHub includes mostly text-based files such as source code (e.g., C/C++, Java, Python), configuration files (e.g., JSON, YAML, INI), and a variety of text files for documentation (e.g., Markdown, RTF). Conversely, VirusTotal includes file archives (e.g., ZIP, TAR), binaries (e.g., EXE, DLL, ELF), as well as documents (e.g., PDF, DOC, XLS).

Content types. We selected prevalent content types based on available public metrics. Using a public mirror of GitHub designed for large-scale analysis [29] and VirusTotal’s public API [30], we first calculate, for each, the top 50 most prominent file extensions across both platforms. We then identified the associated content type, if any, for each of these file extensions.¹ Given the potential biases of our data sources, due to which potentially relevant content types do not appear in the most-frequent file extensions, we augment this list

¹Not all file extensions are associated to a content type: for example, a significant portion of samples on VirusTotal have the `.file` or `.virus` extension which is not indicative of any content type.

by consulting existing resources [31]–[33] and adding file extensions if at least one of the authors believed to be critical to make a first release of practical utility.

In total, we consider 128 file extensions in our study. As a simplification, we manually group some extensions into a single category. For example, we group both JPEG and JPG into a JPEG content type. Similarly, we group both EXE and DLL into a PEBIN content type. As a notation mechanic, we format these canonical content types as `TYPE` throughout this paper, whereas we refer to file extensions as `TYPE`. This process reduces our final set of categories from 128 file extensions to 113 canonical content types. Of these, 70 are binary content types and 43 are text content types. The full list of selected content types is available on the Magika GitHub repository [11].

We consider our selection of content types to be sufficiently large and diverse to demonstrate the generalizability of our approach. We also note that a more recent version of `MAGIKA`, available on GitHub, already supports more than 200 content types.

Sampling & validating content types. For each content type, we query GitHub and VirusTotal to obtain a random sample of files associated with the content type’s file extensions. To minimize the risk of mislabeled content types in our ground truth, we perform a number of validation checks before adding a sample to our dataset. We specifically avoid using existing content-type detection tools as part of our validation, otherwise, we might oversimplify content-type detection (in the case of requiring agreement across all tools); or propagate detection errors (in the case of trusting one tool above others).

We use four heuristics for validation: file size, magic bytes (for binary files), character encoding (for text files), and file trustworthiness. For file size, we require any sample in our dataset to consist of at least 16 bytes. For magic bytes, we apply a set of *necessary* but not *sufficient* rules to validate a file extension. Note that these rules are, by design, straightforward, as we would otherwise risk the introduction of biases in our dataset. For example, all PEBIN (the main Microsoft Windows executable format) *must* start with the string `MZ` (`0x4D 0x5A`). Not all files starting with `MZ` are PEBIN (e.g., they could be textual files which happened to start with the characters `MZ`), but if a sample’s file extension claims to be an EXE or DLL, we verify this condition holds. For text files—where checks on magic bytes are not applicable—we merely verify the encoding to ensure the file contains only text characters. Finally, for samples from VirusTotal, we ensure that no anti-virus engines flags the sample as malicious due to such samples having an increased risk of obfuscated types.

Independent of these workflows, we also create *synthetic* samples for two content types: UNKNOWN and TXT. For UNKNOWN, each sample consists of a random byte sequence of arbitrary length. For TXT, each sample consists of a random string of text characters of arbitrary length. These synthetic samples are used as a form of data augmentation to train the model. We do so to help the model handle files with content

types it has not been trained on, following best practices: rather than forcing the model to choose 1 of N valid content types, it can select UNKNOWN. We empirically observed that this approach has a negligible impact on accuracy on our dataset ($<0.05\%$ accuracy improvement). Nevertheless, in adherence to best practices, we adopt this technique as it does not introduce any disadvantages. There are additional technical details we omit for space constraints, but the interested reader can find complete information in our open-source release.

Final dataset. Our final dataset consists of 26.5 million samples. We randomly split these into a training, validation, and testing dataset. The latter serves as a uniform benchmark for all content-type detection tools, including our deep learning model, but is never exposed to our model during training, nor has it been used to select the model hyperparameters or thresholds (for which we used the validation split, following best practices). In total, we selected 10K samples per content type for our testing benchmark (with more than 1M samples in total, making our testing dataset large enough to obtain robust evaluation metrics), 10K samples for model validation, and 10K+ samples for model training (capping at 1 million per content type). There are two exceptions to this: for ISO and ODP content types, we have only 14K samples total. As we show later, this does not have a material difference on our deep learning detection accuracy. In total, our testing benchmark consists of 1.2 million samples; our training dataset consists of 24 million samples; and our validation dataset consists of 1.2 million samples.

Limitations. As with any measurement or machine learning study, our methodology incurs a number of limitations. First, our dataset consists of only 113 content types. We acknowledge the selected list is necessarily incomplete and may not encompass content types relevant to all deployment environments. While there is a long tail of many other content types in use today, acquiring and validating a representative sample is prohibitively expensive—in terms of manual overhead. As such, we focus on selecting a diverse and large enough number of content types to prove the generalizability of our deep learning approach on the most popular content types, leaving the extension of our technique to a more comprehensive set of content types to future work with the open source community. Second, despite our best efforts at validation and sampling, our benchmark dataset may contain mislabeled content types or be biased towards the samples available on VirusTotal or GitHub. As such, performance metrics may vary for other deployments, though our sample size should be suitably large to provide insights into the limitations of existing content-type detection tools, as well as our own.

IV. BENCHMARKING EXISTING TOOLS

We benchmark the performance of existing content-type detection tools to motivate the need for more robust detection. Our measurements rely on the 1.2M samples in our test benchmark dataset. The dataset includes samples of various sizes, as shown in Figure 2. As part of our evaluation,

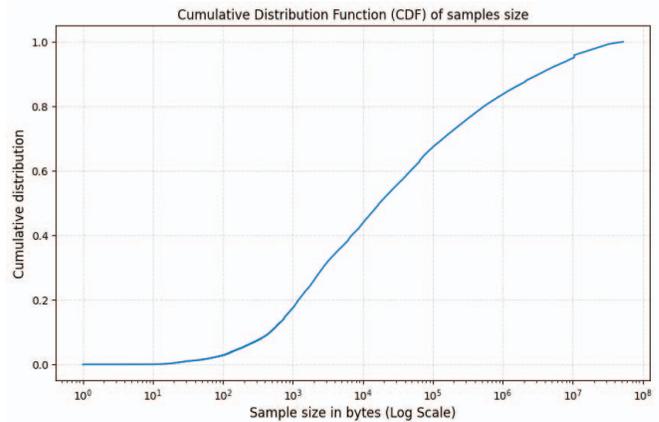


Fig. 2: CDF of the sample sizes in our benchmark dataset.

we assess both the number of content types supported per tool (overlapping with the types in our benchmark) and the accuracy of predictions per tool. As we show, the best existing tool achieves an F1 score of only 88%, with no single tool supporting all of the content types we evaluate.

A. Tools Selection

Our benchmark evaluates four popular content-type detection tools used for a variety of applications:

- `file` [2]: the default command line tool for detecting content types for a variety of files. We note that `file` can be optionally queried to return the MIME type, which we also benchmark as the two modes can yield distinct results (denoted `file-mime`).
- `exiftool` [4]: a tool originally developed for detecting image content types, but that has since expanded to a variety of binary and text content types.
- `trid` [5]: a tool for detecting a wide range of content types.
- `guesslang` [6]: a tool for detecting the programming language based on an excerpt of source code.

We select the first three because they represent the status-quo: they are based on well-established signatures-based approaches, readily maintained, and are widely adopted. We select `guesslang` as a representative of emerging content type detection tools based on machine learning. `guesslang` is unique as it focuses exclusively on textual content types, and is robust enough for real-world deployment in VS Code [7].

B. Metric selection

For each tool, our benchmark assesses the precision and recall of inferred content types compared to the golden labels of our test dataset. Given this is a multi-class classification problem, we estimate per-type *precision* as: $TP/(TP + FP)$. Here, a true positive (TP) indicates the tool predicts the correct

golden label,² while a false positive (FP) indicates the tool predicts a specific content type, but in fact the golden label is any other content type. We calculate per-type *recall* as: $TP/(TP + FN)$. Here, a false negative (FN) means a tool failed to detect a sample as the golden canonical content type, predicting any other label. For ease of presentation, we simplify these metrics into an F1 score, which provides equal weight to both per-type precision and recall:

$$F_1 = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

We calculate this F1 score per content type and then average F1 scores across text content types, binary content types, and overall content types to provide a variety of assessments on how well existing tools perform.

C. Automating Large-Scale Evaluations

The text string outputs of existing content type detection tools are not designed for cross comparison. As such, we develop a harness to programmatically query millions of samples while normalizing the outputs to match one of the 113 canonical content types in our benchmark.

Normalizing detected content types. Unfortunately, there are no canonical naming conventions for content types across tools—or even the same version of the same tool. Nor is there a hierarchy of how content types relate. For example, valid command line outputs of `file` for XML documents include `XML document`, `XML 1.0 document`, and `XML 1.0 document text`, among others. We also find that different versions of the same tool will silently update naming conventions. For example, `file` silently changed the output for JavaScript files from “Node.js script text executable” to “Node.js script executable,” dropping the “text” keyword, and potentially breaking automated workflows [34].

Similarly, we find that the MIME types generated by tools, despite being machine-oriented, also do not follow canonical naming conventions, for several reasons: MIME types change over time (e.g., `text/x-markdown` vs. `text/markdown` [35]); tools (silently) update their output (e.g., for `PEBIN`, `file 5.41` outputs `app/x-dosexec`, while `file 5.44` outputs `app/vnd.microsoft.portable-executable`); some content types have multiple valid MIME types (e.g., for XML, both `application/xml` and `text/xml` are valid); tools databases have typos (e.g., `text/pyton`), or arbitrary variations (e.g., `text/python37` vs. `text/python`).

We used a manual, iterative process to address all naming discrepancies. For each tool, we gathered the default command line outputs for every file in our benchmark. We then grouped these by output frequency, mapped the most popular types to

²Given multiple fine-grained content types may be associated with a canonical content type (e.g., JavaScript and TypeScript \rightarrow JAVASCRIPT), we treat a prediction as accurate if it matches any fine-grained content types in our mapping.

Tool name	# Binary types (Of 70)	# Text types (Of 34)	# All types (Of 113)
<code>file</code>	66	31	97
<code>file-mime</code>	64	27	91
<code>exiftool</code>	38	14	52
<code>trid</code>	68	23	91
<code>guesslang</code>	0	29	29

TABLE I: Number of content types in our benchmark that are supported by existing content-type detection tools.

one of our 113 canonical content types using a set of hand-written rules, and iteratively updated the set of rules until 99% of the default outputs mapped to a canonical type. In the case a tool produced a content type outside our canonical set, we flag it with a special error code.

Enumerating supported content types. Of the tools we evaluate, only `exiftool` and `guesslang` provide a list of supported content types. To determine which tool supports which content type, we use the normalized outputs over every sample as previously discussed. If a tool never flags any sample as one of our canonical content types, we make a simplifying assumption that it does not support that content type. This provides a fairer comparison with existing tools: we omit unsupported samples from some of our metrics, rather than giving a tool an F1 score of 0% for such content types. Table I shows a breakdown of the content types in our benchmark supported by existing tools across text, binaries, and overall.

D. Results

We report a high-level summary of our benchmark results for each existing tool in Table II. For space reasons, we share per-content type results for only a sample of the 113 content types we benchmark in Table III. Full results can be found on the GitHub repository [11]. Overall, if we restrict our view to content types supported by each tool, we find that `file-mime` achieves the best performance with an average F1 score of 88%—with `exiftool` and `trid` scoring just 1% lower. For text exclusively, `guesslang` achieves the best performance with an average F1 score of 77%. For binaries exclusively, `file-mime` achieves the best performance with an average F1 score of 96%. As we demonstrate shortly, `MAGIKA` is able to outperform all of these tools, achieving an overall average F1 score of 99%. We explore the performance of each existing tool in detail below.

file and file-mime. We find that `file` is highly performant for supported binary content types, with an average F1 score of 96% when using the MIME flag, and 92% otherwise. However, its accuracy decreases for some specific binary types, such as `APK` (80%), `JAR` (64%), or `DMG` (4%). For supported text types, the average F1 score drops to 68%, likely due to the limitations of signatures when applied to text. We find the accuracy of supported text content types can vary widely, with an F1 score of 88% for `RUBY`, 70% for `LISP`, and just 1% for `POWERSHELL`. Support is also missing for

Content Type Metric	file	file-mime	exiftool	trid	guesslang
Binary – supported types	92%	96%	93%	93%	-
Text – supported types	67%	68%	70%	68%	77%
Overall – supported types	84%	88%	87%	87%	77%
Binary – all types	87%	87%	50%	91%	0%
Text – all types	48%	42%	22%	35%	52%
Overall – all types	72%	70%	39%	70%	19%

TABLE II: Performance of existing content-type detection tools—measured as an F1 score—averaged across all binary, text, and overall content types. For fairness, we include two scopes for our metrics: performance restricted to supported content types, and performance on all content types.

some programming languages, such as RUST and SCALA. We note that the MIME flag is critical for some types, e.g., the accuracy drops from 100% to 1% for DOC without the flag.

exiftool and trid. We find that `trid` and `exiftool` provide similar overall accuracy, with an average F1 score of 87% each. However, `trid` supports far more content types (91) than `exiftool` (52). Both `trid` and `exiftool` see drops in accuracy for text content types, with average F1 scores falling to 70% and 68% respectively. For instance, `trid` produces inaccurate inferences for JSON (9%), SQL (6%), and YAML (<1%) despite supporting each content type. Likewise, `exiftool` struggles with PYTHON (22%) and RUBY (3%) among other text content types. As with `file`, the high variance of F1 scores makes it challenging for clients to understand the quality of content-type detection from a tool absent benchmarks.

guesslang. As previously mentioned, `guesslang` focuses exclusively on text content types, achieving an average F1 score of 77% for supported types. While `guesslang` is highly accurate for some programming languages like RUST (97%) or SCALA (92%), it nevertheless struggles with HTML (60%) and XML (31%) among others. One limitation specific to `guesslang` is that it does not support an “unknown” verdict. As such, it still predicts a random, incorrect content type for our synthetic TXT samples. We note that VS Code developers ran into this exact problem, as discussed in a GitHub issue [36]. Instead, VS Code implements a number of heuristics to artificially penalize automatic recognition of content types prone to false positives [37].

V. MAGIKA

In light of the limitations of existing content-type detectors, we design and train a new deep learning classifier called MAGIKA that distinguishes between content types without the need for signatures created by experts.

A. Requirements

We design MAGIKA to support two distinct deployment scenarios with a single model: 1) a command line tool that

Content Type	file	file-mime	exiftool	trid	guesslang
APK	80%	80%	-	83%	-
ASM	21%	21%	-	-	90%
ASP	-	-	-	80%	-
BATCH	65%	65%	-	-	38%
BMP	100%	100%	100%	98%	-
C	59%	59%	-	-	91%
CS	-	-	-	-	96%
CSS	-	-	-	-	84%
CSV	75%	75%	-	-	56%
DMG	0%	4%	-	77%	-
DOC	1%	100%	99%	98%	-
DOCX	99%	99%	65%	100%	-
ELF	100%	100%	-	63%	-
GO	-	-	-	-	95%
HTML	58%	58%	81%	73%	60%
INI	2%	2%	-	46%	28%
JAR	64%	64%	-	74%	-
JAVA	81%	81%	-	-	87%
JAVASCRIPT	81%	81%	-	-	88%
JPEG	100%	100%	100%	100%	-
JSON	98%	98%	98%	9%	84%
MACHO	100%	100%	-	100%	-
MAKEFILE	62%	62%	-	-	95%
MARKDOWN	-	-	-	-	48%
PDF	100%	100%	100%	100%	-
PEBIN	100%	100%	-	56%	-
PERL	74%	75%	75%	3%	80%
PNG	100%	100%	100%	100%	-
POWERSHELL	1%	-	-	-	91%
PYTHON	94%	87%	22%	-	90%
RUBY	88%	88%	3%	-	89%
SHELL	89%	89%	88%	61%	83%
SQL	-	-	-	6%	82%
TXT	17%	15%	10%	1%	0%
UNKNOWN	75%	31%	5%	6%	2%
VBA	-	-	-	-	51%
XLS	12%	99%	90%	91%	-
XLSX	100%	100%	97%	99%	-
XML	31%	31%	35%	64%	31%
YAML	-	-	-	0%	74%
ZIP	56%	56%	39%	77%	-

TABLE III: Performance of existing content-type detection tools—measured as an F1 score—per content type for a sample of content types in our benchmark.

can replace established utilities, and 2) a bulk inference tool that can scale to analyzing billions of samples per day. These scenarios constrain how we approach accuracy, speed, and resource utilization.

Accuracy. MAGIKA should provide equivalent or better accuracy at distinguishing between the 113 content types in our dataset. Furthermore, detection should depend exclusively on a file’s content—regardless of the presence of a file extension or metadata. As before, we use an F1 score to evaluate accuracy.

Speed. Bulk inference should return a decision within <10ms per file, excluding any initialization overhead for loading a model into memory. Command line users may be sensitive to such initialization costs. As such, the total overhead of

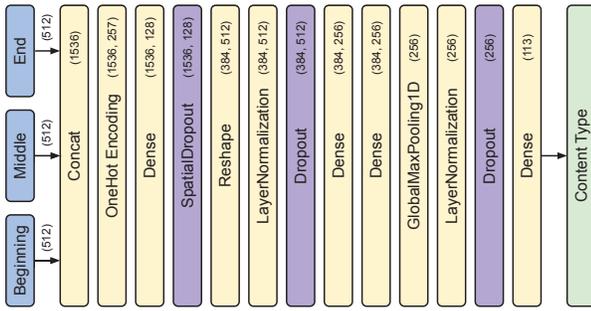


Fig. 3: Architecture of MAGIKA. The input and the output are depicted in blue and green, respectively. The model’s layers are in yellow. The layers in purple are used only in training. The numbers next to the layers’ names indicate the size of their outputs.

initialization and inference should remain $< 100\text{ms}$. This speed also includes the time spent reading a file, which has design implications for how to detect the content type of large files (e.g., $> 100\text{MB}$) without necessarily reading the whole file.

Resources. In order to maximize potential deployment scenarios, MAGIKA should meet our speed and accuracy requirements even on a single CPU. This resource constraint is atypical of machine learning, which often assumes access to a GPU. However, we cannot expect command line users to have access to a GPU. Likewise, requiring large-scale platforms to dedicate GPU resources for the simple task of content type detection would be prohibitively expensive, especially considering the required scale (e.g., potentially billions of files processed daily).

B. Model Architecture

We present the architecture of the deep learning model that powers MAGIKA in Figure 3. The overall flow consists of (1) transforming the contents of a file into a fixed-sized vector representation; (2) processing the vectors via a neural network; and (3) interpreting the model outputs to predict the most probable content type.

Inputs. The model’s inputs consists of three vectors that encode a sequence of 512 bytes selected from the beginning, middle (taken from the center of the file, with no randomness), and end of an input file. We encode each byte as an integer in the range $[0, 255]$. To maximize the utility of the inputs made available to the model, we strip any leading or trailing whitespace from the beginning and end of a file’s contents before selecting bytes for encoding. As the model’s input has constant size (i.e., 3×512 integers), if an input file is too small, we pad the encoding with a special character (represented by the integer 256). We then concatenate the three input vectors to form a single vector of 3×512 integers, which we one-hot encode and embed into a 128-float vector using a Dense layer.

While we considered using longer sequences, or more sequences of bytes from a file, in practice this incurs additional resource requirements and slows down processing due to having to seek over more portions of the input file. By using

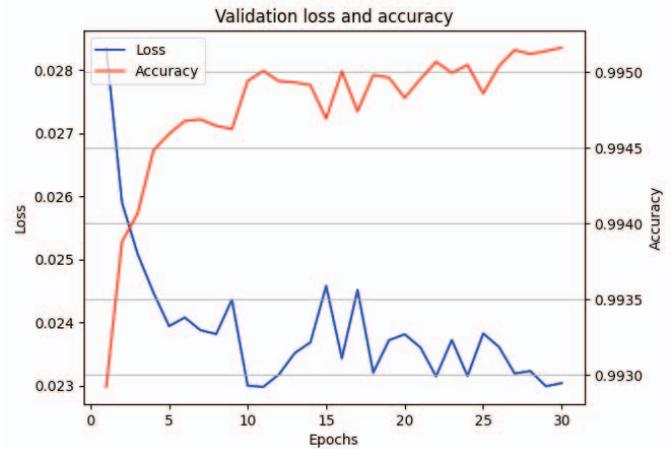


Fig. 4: Validation loss and validation accuracy as the training progresses in terms of the number of epochs. We find accuracy increases up to around 30 epochs.

a fixed size input rather than a whole file, we ensure inference in constant time. Despite limiting the model’s visibility, we demonstrate we can achieve robust accuracy.

Trunk. The model trunk consists of two components. First, the output of the previous Dense layer is reshaped to 384×512 dimensions, effectively reorganizing the previous output into small, four-byte chunks. Each chunk is represented by a 512 dimension vector that the model can treat as a single entity instead of having to consider each byte separately, which we found to improve both performance and efficiency. Then, the model contains two 256-dimensional Dense layers with *gelu* activation [38]. A global max pooling layer [39] performs downsampling and reduces the dimensionality back to 1D. During training, we apply layer normalization [40], a 10% dropout rate [41], and a 10% of spatial dropout rate [42] for regularization throughout the model.

This final model represents hundreds of design iterations across architectures that are simple enough to be small and fast on a CPU. We also ran extensive hyperparameter tuning experiments (using the validation dataset) and tested various model configurations that evaluated the embedding dimension; the size of the Reshape layer; the number, size, and activation of the Dense layers; the normalization type; and the amount of dropout applied before converging on this specific design.

Outputs. The final Dense layer uses *softmax* activation [43] with size equal to the number of canonical content types in our dataset. The output vector represents a probability distribution over each of the potential content types. We can select the most likely output with an *argmax* function [44], or apply individual confidence thresholds per content type, ultimately yielding the single most likely content type among those that meet a minimum confidence threshold.

C. Training

We trained the MAGIKA model using Categorical Cross-Entropy loss which is suited for multi-class classification

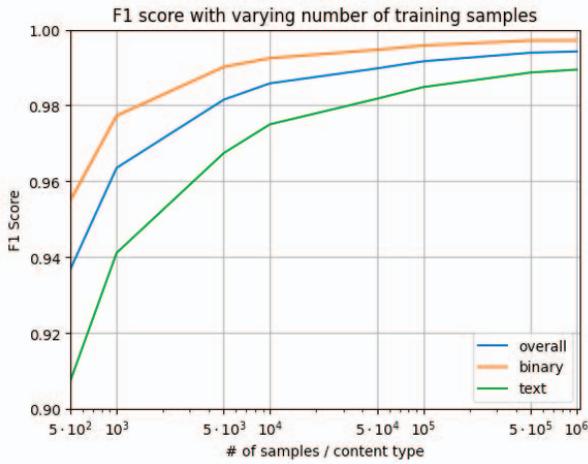


Fig. 5: Average F1 Score (after one epoch of training) with increasing number of samples per content type, across binary, text, overall content types.

settings, with Adam as the optimizer (batch size = 256, learning rate = 0.001, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 1e-07$). Additionally, we use CutMix [45] for data augmentation, although our experiments did not highlight a significant boost in validation accuracy. However, since CutMix is used only during training and does not affect model inference, we continue using a 5% rate for CutMix because it has no downsides and could make the model more robust to out-of-distribution samples.

We trained the model on a machine with a 16-Core AMD Ryzen 9 7950X CPU, one NVIDIA GeForce RTX 4090, and 126GB of RAM. We implemented the model and training pipeline in TensorFlow [46] and Keras [47]. The training pipeline handles hundreds of millions of examples in a scalable manner, with dataset sharding and shuffling to ensure that batches are balanced across the different content types. We train the final model for 30 epochs on the training dataset, which took 6 days and 21 hours total on our setup.

Figure 4 shows how the loss and accuracy progress with the number of epochs (computed on the validation split of the dataset). Note how the model achieves a very high validation accuracy already starting from the first few epochs. Figure 5 shows how the average F1 score increases with the number of training samples per content type. Note how the average F1 score for binary content types surpasses 99% with only 10K samples, but that more samples are needed to boost the average F1 score among text content types.

D. Setting Confidence Thresholds

A classification threshold is the cut-off point whereby we treat the probability output from our final Dense layer as suitably high-confidence to yield a content-type prediction. Selecting an optimal threshold involves balancing the trade-offs between precision and recall, which is made more complicated by a multi-class setting. For example, we empirically found that a probability value of 0.80 was robust to assert that a file

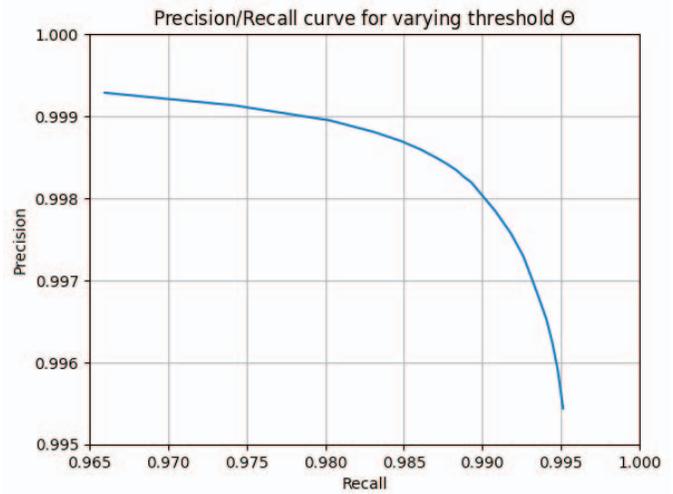


Fig. 6: Precision-recall curve for a fixed threshold Θ applied to all content type predictions. Note that the truncated scale for precision and recall is different.

is HTML, whereas the same threshold for PDF yielded poor accuracy.

We approached this problem by first considering a single threshold for all content types. Figure 6 contains the precision-recall curve for this scenario. The curve shows that while there is a trade-off between precision and recall, we can configure the model to obtain a precision and a recall that are both higher than 99.5%.

As a further optimization, we instead compute per-content-type thresholds. We determine each threshold by fixing precision at 99% and then choosing the maximum recall of the remaining thresholds. At inference time, we then select the *argmax* of content type predictions that exceed the type’s custom threshold. If the model yields no confident outputs, we output either TXT or UNKNOWN, depending on the nature of the content type the model was the most confident with. We discuss the overall performance of this approach shortly. In practice, clients using MAGIKA can set their own confidence thresholds, but by supplying tuned defaults we avoid a pain point encountered by users of other tools [36], [37].

E. Performance Optimizations

Per our design requirements, MAGIKA must also be performant in terms of the time it takes to load the model into memory and yield a prediction. We implement a number of performance optimizations, whereby a client can access our deep learning model either through a command line or API. As part of this, we use OnnxRuntime [48] (instead of Tensorflow and Keras, which we use for training), because it is roughly 15x faster in loading the model (while having a similar model inference time), thus significantly reducing the initialization cost, which is critical for command line use cases. Likewise, we support batching for clients performing multiple inferences to parallelize processing. Our first CLI prototype has been written in Python, but due to performance reasons, we have

Content Type Metric	F1	Δ (Supported)	Δ (All)
Binary	100%	4%	9%
Text	99%	22%	47%
Overall	99%	12%	27%

TABLE IV: Performance of MAGIKA—measured as an F1 score—averaged across all binary, text, and overall content types. To simplify comparison across tools, we calculate the delta (Δ) in F1 score between MAGIKA and the most accurate existing tool per metric.

also implemented a version in C++ for production use-cases, and a version in Rust as the iteration of our command line tool (already available on GitHub [11]). We have also implemented a prototype based on TensorFlowJS [49], which allows one to run MAGIKA entirely within a browser.

VI. EVALUATION

In order to demonstrate the value of MAGIKA, we benchmark it against existing tools in terms of accuracy and speed.

A. Accuracy

We report a high-level summary of our benchmark results for MAGIKA in Table IV. The reported accuracy metrics have been computed using the testing dataset, which, following best practices, has not been used for any step involving the creation or tuning of MAGIKA. As a source of further insights, we share per-content type results for a sample of the 113 content types in our benchmark in Table V. Full results can be found in our source repository [11]. Regardless the context—be it text, binary, or overall content types—we find that MAGIKA outperforms all existing tools. This demonstrates the generalizability of our architecture and approach.

Accuracy gains. For binary content types, we find that MAGIKA yields a modest F1 gain of 4% when compared to the best existing tool—`file-mime`—limited to content types supported by both MAGIKA and `file-mime`. This F1 gain extends to 9% if we consider all content types in our benchmark. For text content types, MAGIKA yields even better F1 gains of 22% over `guesslang`, limited to content types supported by both MAGIKA and `guesslang`. This extends to 47% for all content types in our benchmark. This is especially true of MARKDOWN (45% gain), VBA (49% gain), and XML (35% gain). Likewise, MAGIKA is better than all models at exhibiting uncertainty, with an F1 score of 94% for UNKNOWN (synthetic random byte sequences) and 84% for TXT (synthetic random text strings).³ We also note that MAGIKA has high accuracy (and often shows a double-digit % gain in F1) even for content types that were shown to be problematic for `guesslang`'s integration in VS Code, such as BATCH, CSV, INI, MAKEFILE, SQL, and YAML [37].

No need for preprocessing. We find that MAGIKA is able to handle various forms of packing and compression. For

³In practice, this TXT includes both TXT files as well as our synthetic examples. This label is only valid if no other text content types apply (e.g., CSV, HTML, etc..)

Cont. Type	F1	Δ	Cont. Type	F1	Δ
APK	99%	16%	MACHO	100%	0%
ASM	99%	9%	MAKEFILE	100%	5%
ASP	99%	19%	MARKDOWN	92%	45%
BATCH	97%	32%	PDF	100%	0%
BMP	100%	0%	PEBIN	100%	0%
C	99%	8%	PEM	100%	14%
CS	100%	4%	PERL	99%	20%
CSS	99%	15%	PNG	100%	0%
CSV	99%	24%	POWERSHELL	99%	8%
DMG	100%	23%	PYTHON	99%	5%
DOC	99%	-1%	RUBY	99%	11%
DOCX	99%	0%	RUST	100%	3%
ELF	100%	0%	SCALA	100%	8%
GO	100%	4%	SHELL	98%	9%
HTML	94%	13%	SQL	99%	18%
INI	98%	52%	TXT	84%	67%
JAR	98%	25%	UNKNOWN	94%	19%
JAVA	99%	12%	VBA	99%	49%
JAVASCRIPT	99%	12%	XLS	99%	0%
JPEG	100%	0%	XLSX	99%	-1%
JSON	99%	1%	XML	99%	35%
LATEX	100%	5%	YAML	99%	25%
LISP	100%	4%	ZIP	99%	21%

TABLE V: Performance of MAGIKA—measured as an F1 score—per content type for a sample of content types in our benchmark. To simplify comparison across tools, we calculate the delta (Δ) in F1 score between MAGIKA and the most accurate existing tool per content type.

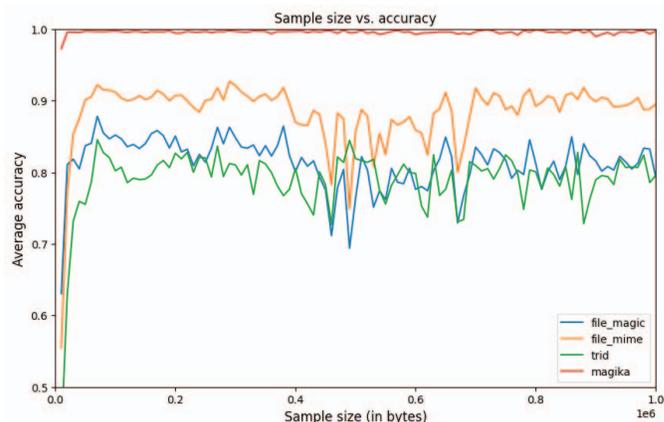


Fig. 7: Average accuracy vs. samples size for MAGIKA and existing tools (for simplicity, the figure only shows tools with an average accuracy higher than 50%).

example, DOC, XLS, PPT are all variants of the same Composite Document File binary format. Likewise, DOCX, XSLX, PPTX, APK, and JAR are all instances of Zip file formats. Most existing tools deal with these content types by first unpacking the file before applying any signature-based rules. Conversely, MAGIKA requires no pre-processing or domain knowledge to yield accurate predictions.

Accuracy vs. sample size. Figure 7 shows how the accuracy of MAGIKA and existing tools change depending on the samples size. The results show how MAGIKA achieves a stable 99%+ accuracy for most samples, with a slight degrade in accuracy for very small samples of less than ~ 100 bytes; upon manual

inspection, we find that this loss of accuracy is due to MAGIKA outputting a generic content type (such as TXT or UNKNOWN) due to lack of strong confidence. It is also interesting to see how the average accuracy of existing tools is also somewhat stable across samples size, but with much higher variance.

Limitations. Our evaluation is affected by a few limitations. First, our dataset is balanced across content types: this is standard practice (especially when there is no *one* reference real-world distribution), as it allows one to determine how each content type is supported regardless of whether they are rare in practice. This makes the results more transparent and easier to inspect (e.g., when using an imbalanced dataset, a model with poor support for a rare content type would still have a very high average accuracy). However, we acknowledge that the reported accuracy metrics are not directly applicable, but given that MAGIKA is open-source, a user with a very unique setting can tune and evaluate MAGIKA to its needs, for example by mapping our results to her distribution—this is the standard way to adapt models to an imbalanced environment.

The second limitation is that, as it is common with works based on deep learning on large datasets, we did not perform cross-validation, as it is computationally prohibitively expensive (as reported, a single training run takes about one week). However, cross-validation is deemed as critical only when dealing with very small datasets [50], in which a specific “pick” of the testing dataset may be accidentally very biased; this is not a concern in our situation, in which the testing set alone has more than 1M samples. We also performed an additional experiment in which we split our testing dataset in 10 random smaller sets: the average accuracy is $99.19\% \pm 0.03\%$, showing very low variance, which supports the robustness of our evaluation.

B. Speed

Per our design requirements, the time it takes to infer a content type is equally important to accuracy. There are two relevant dimensions to processing latency: initialization (e.g., loading a signature database or model) and inference (e.g., via matching signatures or running a model). For MAGIKA we can measure the breakdown between these two aspects directly, but it is challenging for the other tools. Thus, we measure the breakdown indirectly, by measuring the time required to evaluate one sample at a time, and to evaluate multiple samples all at once.

We evaluate all of the existing tools and MAGIKA as follows. We select a fixed subset of our benchmark dataset that consists of 1,130 samples: ten for each of the 113 content types. This avoids any bias in the event a tool is faster for one content type versus another (e.g., due to requiring decompression or more complex signatures). As a first experiment, we send a single request to each tool for each sample in our evaluation. As a second experiment, we pass all samples as a single request to each tool (e.g., we provide each sample’s path as arguments to the same command line invocation). In both cases, we limit the number of available CPUs to one (using the

Tool name	One sample per request (ms)	All samples in one request (ms)
<code>file</code>	5.36	0.75
<code>file-mime</code>	5.30	0.67
<code>exiftool</code>	102.56	6.36
<code>trid</code>	137.08	51.57
<code>guesslang</code>	958.61	307.66
MAGIKA	86.73	5.77

TABLE VI: Average execution speed of MAGIKA and existing tools when scanning 1,130 samples (113 content types, 10 samples per type). The first column captures the initialization and inference cost for content-type detection on a single sample (averaged across all samples). The second column captures the amortized initialization and inference costs of content-type detection for all samples in a single request, normalized to the total number of samples.

`taskset` [51] utility). We estimate the overall running time for both modes using `hyperfine` [52], executing each experiment ten times and averaging the results (with three warm up rounds, to minimize the impact of external factors, such as caches). Our environment consists of an isolated docker container within an idle virtual machine (an `e2-highmem-8` instance on Google Cloud, with 8x AMD Rome CPUs and 64GB of RAM).

Table VI shows the results. We find that `file` is the fastest tool (with or without the MIME flag), followed by MAGIKA, `exiftool`, `trid`, and `guesslang`. Apart from `file`, every tool has a non-negligible initialization cost on the order of 100ms or more. For clients scanning files in bulk, the amortized processing time of MAGIKA drops to 5.77ms, better than all existing tools apart from `file`. The significant performance gain of MAGIKA over `guesslang` (the only other tool to use a model) is likely due to the latter’s reliance on TensorFlowJS [49], whereas MAGIKA uses OnnxRuntime. We note that our implementation of MAGIKA allows clients to take advantage of multiple CPUs without having to wrap requests via parallelization, which is not true of existing tools. For example, our client achieves an average of 1.39ms per sample when running on 8 CPUs, while the amortized processing time of the other tools remains the same. While many other optimizations are feasible, our results demonstrate that MAGIKA can quickly infer content types even when executing a neural network exclusively via a CPU.

VII. REAL-WORLD ADOPTION

We released the MAGIKA model and a command line wrapper as open source under an Apache 2 license [11]. We share information on its reception and real-world adoption.

GitHub release and open-source community. Upon release, MAGIKA reached 4K stars on GitHub in less than a week, it was featured in GitHub trending projects, and it sparked interest from many external contributors. Likewise, a number of developers have reached out to integrate MAGIKA into their workflows, such as for validating datasets, or for enhancing security malware analysis pipelines (e.g., AssemblyLine [53]). MAGIKA was also recently integrated in a very prominent open-source framework, Apache Tika [13].

Attachments scanner. We have worked with Gmail and Google Drive to integrate MAGIKA into their attachment and file upload scanners that detect and block malicious samples [12]. These scanners process hundreds of billions of files every week, underscoring the computational performance of our approach. MAGIKA can be used to enforce content-type policies on potentially obfuscated files (e.g., prohibiting executable binary content types as attachments), and to route specific samples to anti-virus scanners based on the content type detected, which might otherwise be prohibitively expensive to run on all samples. For example, we have estimated that, on a daily basis, MAGIKA routes to MS Office-specific malware scanners several millions samples that would have otherwise not been scanned when using existing content type detection systems or file extensions. MAGIKA has now been running in production for several months, with no significant concerns to be reported.

VirusTotal file scanner. The VirusTotal service has recently integrated our work: every submission to VirusTotal is now processed with MAGIKA, whose result can be seen in the “Details” tab of each submission, alongside other content type detection tools. This can be used by VirusTotal for improved indexing and to optimize which files are sent to the platform’s Code Insight functionality [54], which employs generative AI to analyze and detect malicious code. For examples, MAGIKA’s accurate detection of POWERSHELL means that VirusTotal can now isolate such files and specialize any generative analysis exclusively to POWERSHELL.

VS Code. We have reached out to VS Code developers and we have presented our model for consideration as a potential replacement for `guesslang`. During our discussions, the developers acknowledged the limitations and challenges encountered with `guesslang`, expressing openness to consider alternatives. We subsequently engaged in a dialogue regarding current feature gaps, primarily focusing on the necessity for more fine-grained detection of specific content types (e.g., C vs. C++, JAVASCRIPT vs. TYPESCRIPT, INI vs. TOML), which we have been exploring with very promising results.

VIII. DISCUSSION

Our benchmark and integration stories highlight that content-type detection remains an important problem. We discuss future design directions for further improving MAGIKA and potential risks.

Handling new content types. We acknowledge that the current version of MAGIKA supports a limited number of content types, and that our selected list is not necessarily the most representative in all scenarios. Moreover, new content types are constantly emerging, requiring the authors of content-type detection tools to continuously maintain and add new signatures. MAGIKA sidesteps the necessity of understanding the intricacies of different file formats (e.g., to write signatures and to ensure they do not collide with previous signatures). Instead, handling new content types with MAGIKA requires two steps: (1) sourcing a sufficiently large number of training

samples; and (2) retraining the model with a new Dense output sized to the number of supported content types. We find in practice that MAGIKA can use as few as 10K samples to achieve robust accuracy for binary content types, though text content types require more samples. For our first iteration, we have been relying on GitHub and VirusTotal, but, for some content types, we have been exploring generating samples via large language models. While re-training a new model can take several days, it is not a concern as the process is automated.

Multiple valid content types. MAGIKA currently outputs only a single inferred content type, which is enough for the vast majority of files. However, one could craft so-called *polyglot* files, which are syntactically valid for multiple content types (e.g., interpretable as Python, PHP, and bash) [55]. Such polyglots use unspecified holes in the file format specifications, making accurate detection more challenging. More trivially, multiple valid content types might arise due to a file containing code snippets from different programming languages. In practice, users of MAGIKA may be able to examine the top N most likely content types, though we leave evaluating the accuracy of MAGIKA on multi-content-type files to future work.

Adversarial risk. As with other content-type detection tools, MAGIKA is potentially susceptible to evasion. Here, evasion means tricking a content-type detection tool to infer an incorrect type, but where the intended application can nevertheless process the file’s contents correctly. For signature-based rules, this might be as simple as adding whitespace per our earlier example in Figure 1. For MAGIKA, as the model is open source, attackers might instead add small perturbations to a file’s contents to cause our neural network to infer an incorrect content-type. Adding such noise in a programmatic way, while still yielding a file that is interpretable by the intended application may be non-trivial; however, this analysis falls under the area of adversarial machine learning, and significant work is required to make the approach resilient to attackers that specifically attempt to bypass the detection.

IX. CONCLUSIONS

In this work, we presented MAGIKA, a novel AI-powered content-type detection tool. Trained on 24M samples and 113 canonical content types, we showed how our deep learning architecture can achieve an average F1 score of 99%. Our approach outperforms all existing content-type detection tools, with a 4% F1 gain over the best tool for binary content types and a 12% F1 gain over the best tool for text content types. For bulk inferences, MAGIKA requires 5.77ms to yield a decision per file with just a single CPU, making it suitable for a variety of deployment scenarios. Our tool has already seen adoption in the real world in a number of critical pipelines and has drawn significant attention from the open-source community. We have been working on a second iteration with many improvements; the interested reader can follow the latest developments on GitHub [11].

REFERENCES

- [1] "File man page from PDP-11, 1973," <https://www.darwinsys.com/file/file-v4.1.txt>.
- [2] "File — A file type guesser," <https://www.darwinsys.com/file/>.
- [3] "Manual page for libmagic," https://man7.org/linux/man-pages/man3/magic_list.3.html.
- [4] "ExifTool," <https://exiftool.org/>.
- [5] "TrID," <https://mark0.net/soft-trid-e.html>.
- [6] "guesslang: tool to detect programming language of a given source," <https://github.com/yoeo/guesslang>.
- [7] "VSCode Blog: Automatic Language Detection," https://code.visualstudio.com/updates/v1_60#_automatic-language-detection.
- [8] "JavaScript Basics," https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/JavaScript_basics.
- [9] "GitHub," <https://github.com>.
- [10] "VirusTotal," <https://virustotal.com>.
- [11] "Magika: AI-Powered Content-Type Detection," <https://github.com/google/magika>.
- [12] E. Bursztein and Y. Fratantonio, "Magika: Ai powered fast and efficient file type identification," <https://opensource.googleblog.com/2024/02/magika-ai-powered-fast-and-efficient-file-type-identification.html>, 2023.
- [13] "Apache Tika – Magika integration," <https://github.com/apache/tika/pull/2036>.
- [14] "PEiD: packer detection tool," <https://www.aldeid.com/wiki/PEiD>.
- [15] "Detect-It-Easy GitHub page," <https://github.com/horsicq/Detect-It-Easy>.
- [16] "mime-types - The ultimate javascript content-type utility," <https://github.com/jshhttp/mime-type>.
- [17] "PolyFile GitHub page," <https://github.com/trailofbits/polyfile>.
- [18] "filetype Go Package," <https://github.com/h2non/filetype>.
- [19] "filetype.py Python Library," <https://github.com/h2non/filetype.py>.
- [20] "file-type GitHub Page," <https://github.com/sindresorhus/file-type>.
- [21] N. L. Beebe, L. A. Maddox, L. Liu, and M. Sun, "Sceadan: Using concatenated n-gram vectors for improved file and data type classification," *IEEE Transactions on Information Forensics and Security*, vol. 8, no. 9, pp. 1519–1530, 2013.
- [22] S. Fitzgerald, G. Mathews, C. Morris, and O. Zhulyan, "Using NLP techniques for file fragment classification," *Digital Investigation*, vol. 9, pp. S44–S49, 2012.
- [23] F. Wang, T.-T. Quach, J. Wheeler, J. B. Aimone, and C. D. James, "Sparse coding for n-gram feature extraction and training for file fragment classification," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 10, pp. 2553–2562, 2018.
- [24] G. Mittal, P. Korus, and N. Memon, "FiFty: large-scale file fragment type identification using convolutional neural networks," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 28–41, 2020.
- [25] K. Skračić, J. Petrović, and P. Pale, "ByteRCNN: Enhancing File Fragment Type Identification with Recurrent and Convolutional Neural Networks," *IEEE access*, 2023.
- [26] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir *et al.*, "Wide & deep learning for recommender systems," in *Proceedings of the 1st workshop on deep learning for recommender systems*, 2016, pp. 7–10.
- [27] "guesslang model," <https://github.com/yoeo/guesslang/blob/master/guesslang/model.py#L50>.
- [28] "GitHub issue on guesslang not supporting txt label." <https://github.com/yoeo/guesslang/issues/45>.
- [29] "GitHub Public Dataset on BigQuery," <https://cloud.google.com/blog/topics/public-datasets/github-on-bigquery-analyze-all-the-open-source-code>.
- [30] "VirusTotal API," <https://virustotal.readme.io/reference/overview>.
- [31] "Wikipedia page: List of file formats," https://en.wikipedia.org/wiki/List_of_file_formats.
- [32] "Mozilla Media type and format guide," <https://developer.mozilla.org/en-US/docs/Web/Media/Formats>.
- [33] "mimetype.io — List of all types," <https://mimetype.io/all-types>.
- [34] "File silently updates JavaScript output," <https://github.com/file/file/commit/a2756aa50fdf7d87ebb14002ffd7609373ea6839>.
- [35] "Markdown MIME type RFC," <https://www.rfc-editor.org/rfc/rfc7763.html>.
- [36] "VS Code implementation of heuristics to tackle false positives in GuessLang," <https://github.com/microsoft/vscode/blob/1b27c60b387be23a46f5d9d0b0c20708c12bd68b/src/vs/workbench/services/languageDetection/browser/languageDetectionSimpleWorker.ts#L165>.
- [37] D. Hendrycks and K. Gimpel, "Gaussian error linear units (gelus)," 2023. [Online]. Available: <https://arxiv.org/abs/1606.08415>
- [38] "Keras – GlobalMaxPooling1D," https://keras.io/api/layers/pooling_layers/global_max_pooling1d/.
- [39] "Keras – LayerNormalization," https://keras.io/api/layers/normalization_layers/layer_normalization/.
- [40] "Keras – Dropout Layer," https://keras.io/api/layers/regularization_layers/dropout/.
- [41] "Keras – SpatialDropout1D Layer," https://keras.io/api/layers/regularization_layers/spatial_dropout1d/.
- [42] J. Bridle, in *Advances in Neural Information Processing Systems*.
- [43] "Tensorflow – argmax function," https://www.tensorflow.org/api_docs/python/tf/keras/ops/argmax.
- [44] S. Yun, D. Han, S. J. Oh, S. Chun, J. Choe, and Y. Yoo, "CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features," in *ICCV*, 2019.
- [45] "TensorFlow," <https://www.tensorflow.org/>.
- [46] "Keras," <https://keras.io/>.
- [47] "OnnxRuntime," <https://onnxruntime.ai/>.
- [48] "TensorFlowJS: a library for machine learning in JavaScript," <https://www.tensorflow.org/js>.
- [49] L. Alzubaidi, J. Bai, A. Al-Sabaawi, J. Santamaria, A. S. Albahri, B. S. N. Al-dabbagh, M. A. Fadhel, M. Manoufali, J. Zhang, A. H. Al-Timemy *et al.*, "A survey on deep learning tools dealing with data scarcity: definitions, challenges, solutions, tips, and applications," *Journal of Big Data*, vol. 10, no. 1, p. 46, 2023.
- [50] "Taskset utility tool," <https://manpages.debian.org/testing/util-linux/taskset.1.en.html>.
- [51] "Hyperfine – A command-line benchmarking tool," <https://github.com/sharkdp/hyperfine>.
- [52] "AssemblyLine," <https://github.com/CybercentreCanada/assemblyline>.
- [53] B. Quintero, "Introducing virustotal code insight: Empowering threat analysis with generative ai," <https://blog.virustotal.com/2023/04/introducing-virustotal-code-insight.html>, 2023.
- [54] L. Koch, S. Oesch, A. Chaulagain, M. Adkisson, S. Erwin, and B. Weber, "Toward the detection of polyglot files," in *Proceedings of the 15th Workshop on Cyber Security Experimentation and Test*, 2022, pp. 120–128.