Ghidra: Is Newer Always Better?

Jonathan Crussell
Sandia National Laboratories
jcrusse@sandia.gov

***Abstract*—Malware analysis relies on evolving tools that undergo continuous improvement and refinement. One such tool is Ghidra, released as open-source in 2019, which has seen 39 public releases and 13,000 commits as of October 2024. In this paper, we examine the impact of these updates on code similarity analysis for the same set of input files. Additionally, we measure how the underlying version of Ghidra affects simple metrics such as analysis time, error counts, and the number of functions identified. Our case studies reveal that Ghidra's effectiveness varies depending on the specific file analyzed, highlighting the importance of context in evaluating tool performance.**

We do not yet have an answer to the question posed in the title of this paper. In general, Ghidra has certainly improved in the years since it was released. Developers have fixed countless bugs, added substantial new features, and supported several new program formats. However, we observe that better is highly nuanced. We encourage the community to approach version upgrades with caution, as the latest release may not always provide superior results for every use case. By fostering a nuanced understanding of Ghidra's advancements, we aim to contribute to more informed decision-making regarding tool adoption and usage in malware analysis and other binary analysis domains.

I. Introduction

Because disassembling binary programs is hard [9], many binary software analysis tools sit downstream from one or more disassemblers and rely on their outputs to conduct their analysis. However, these disassemblers are far from finished tools – they are constantly changing as they add new features, make improvements, and fix bugs. One such disassembler is Ghidra [7], which was released as open-source in 2019 and, as of October 2024, has had 39 public releases and 13,000+ commits. This creates a conundrum for downstream users when a new version is released – should they upgrade to it or not? Is it better than the previous version for the binaries that they care about?

Quantifying "better" in the binary software analysis space is challenging as it tends to be situational. For example, a reverse engineer may want to know whether a particular binary has some capability (*e.g.*, to write to a file) to answer a broader question. In this situation, a version of Ghidra that fails to infer a call target to a function containing code which performs the capability is clearly worse than one that correctly infers the target of the call. Translating these situational questions into metrics is challenging. There are some simple metrics, like the number of errors, timeouts, instructions, and functions that we can use instead. However, for metrics like instructions and functions, its unclear whether we would like those to increase or decrease without ground truth.

In this paper, we will describe our work to conduct the same automated analysis (code similarity) across the publicly available versions of Ghidra. We will report trends in basic measurements like analysis time and error counts across Ghidra versions on a malware dataset. We will then look at how the underlying version of Ghidra affects code similarity at different granularities (*e.g.*, function-level, basic-block-level). We will then describe the notable differences in the code similarity results and explore some of the changes in Ghidra that produced them.

We select code similarity as our downstream analysis task because it has many applications and multiple approaches have been proposed in the academic literature [4]. Code similarity also serves as lens to measure changes between versions of Ghidra – for the same input, we would (likely) expect the same output and can use dissimilarities between versions to cue further analysis. Code similarity adds an additional layer to the concept of "is newer always better". In essence, code similarity tools store a "view" of input files from a particular version of the upstream tool. For maintainers of these tools, each new release of the upstream tool creates a conundrum: what should they do with their existing data? One option would be to start from scratch and rerun the new version across the historical data but this could be cost-prohibitive given a large dataset. Alternatively, they could start using the new version, intermixing old and new data. Lastly, they could simply ignore the new version.

Our contributions are as follows:

- We implement a test harness to run arbitrary versions of Ghidra on the same input that supports running arbitrary Ghidra scripts.
- We investigate how 30 versions of Ghidra analyze a malware dataset, reporting on trends in basic metrics like analysis time and error counts, as well as changes in our downstream code similarity task.
- We present four case studies, linking back changes in the downstream results to the originating change(s) in Ghidra.
- We release our test harness to encourage future research towards answering whether newer is always better for other downstream tasks.

Note: we do not intend for this analysis to serve as an indictment of the Ghidra in any manner. We greatly value the

Workshop on Binary Analysis Research (BAR) 2025
28 February 2025, San Diego, CA, USA
ISBN 979-8-9919276-4-2
https://dx.doi.org/10.14722/bar.2025.23003
www.ndss-symposium.org

contributions that the Ghidra team has made and continues to make to the binary analysis community and conduct this analysis to support Ghidra's ongoing improvement. We chose Ghidra for our analysis because it is broadly applicable and has numerous public versions, making it a valuable resource for researchers and practitioners alike. Our methodology can be applied to other disassemblers, and we expect to observe notable differences in their performance and capabilities across versions as well.

II. Related Work

Binary software analysis tools have been widely studied in the academic literature. We do not attempt to enumerate the complete space here and instead note several inspirational and closely related works.

Reinhold *et al.* [11] conduct experiments that are conceptually very similar to those presented in this paper. They explored how the outputs of CVE Binary Tool ("cve-bin-tool") [5] and cwe-checker [3] change across versions for the same set of input binaries. We note that cwe-checker sits downstream from Ghidra and the paper did not explore how the underlying version of Ghidra affected the results of the downstream tool. Possible directions for future research could include experiments where both the version of Ghidra and the downstream tool vary on the same set of inputs.

Pang *et al.* [9] systemize the algorithms and heuristics used by binary disassemblers and provides an evaluation of open-source disassembly tools. We draw inspiration from their work and seek to understand the evolution of a single disassembly tool across many versions rather a multiple disassemblers. Notably, in their work they look at Ghidra v9.0.4, which is several major releases behind. Future work could include using datasets that they provide with our test harness.

Shaila *et al.* [13] outline an approach to combine the results of multiple disassemblers to generate a consensus for a given binary. Again, this work looked at a single version of disassemblers (Ghidra v9.1 for MIPS and v9.0.4 for ARM after discovering a bug in v9.1 for ARM). The DisCo tool could be extended to leverage the consensus of multiple versions of each disassembler for a stronger consensus.

Haq *et al.* [4] provide a survey of binary code similarity techniques, documenting 20 years of approaches that have been proposed. The code similarity techniques that we use in this work are not new and would be categorized as syntactic similarity with multiple granularities and operand removal for normalization. We do not attempt to advance the state-of-the-art in the code similarity and repeating our experiments with more recent similarity tools is left to future work.

In v11.0, Ghidra added BSim [8], a plugin to find structurally similar functions using Ghidra's decompiler. BSim generates a feature vector for each function in a binary and indexes them in a database using locality-sensitive hashing to quickly identify nearest neighbors. Since BSim uses the decompiler to generate feature vectors based on Pcode (Ghidra's internal intermediate representation used by the decompiler), it too sits downstream from the disassembler. The Ghidra documentation for BSim[1] includes the following note:

> BSim fundamentally depends on the Ghidra decompiler, which steadily adds new analysis features that can affect compatibility over time. Many additions have no effect on BSim, have a small effect, or affect only a tiny percentage of functions. To minimize the impact to existing databases, the decompiler, independent of the Ghidra release, is assigned a major and minor version number. A change in the minor number of 1 or 2 should have little to no impact for most queries, but users will have to tolerate this rare degradation of results if they place queries using a client that doesn't match the BSim server's version. If the client and server differ by a major version, queries will return an error message.

Future work could explore how BSim feature vectors change across versions of Ghidra to quantify this degradation and whether recomputing BSim vectors is warranted for all major version changes in the decompiler.

III. Methodology

To understand how changes in upstream tools, like Ghidra, affect downstream tools, like our code similarity tool, we run multiple versions of the upstream tool across the same corpus of inputs (binaries), allowing us to compare Ghidra's outputs. We create a test harness that allows us to build Linux containers to simplify the management of multiple versions of the upstream tool (*i.e.*, we build one container per version of Ghidra). To simplify installation, we use the pre-built Ghidra releases published on Github[2]. Our test harness allows us to build, push, pull, save, and load these containers which facilitates moving them between machines to parallelize analysis. Our test harness also allows us to run Ghidra and collects two outputs: 1) standard output and error logs and 2) artifacts for downstream tools. To facilitate further research in this space, we release our test harness as open-source software to the community[3].

The standard output and errors logs provide a high-level summary of the analysis but are not easily parsed since they are intended to be read by a user. We use pattern matching to extract whether a fatal exception occurred and Ghidra's total analysis time. There is additional information, including the warnings that Ghidra reports as it does its analysis, that could be fruitful to extract for future work.

To produce artifacts for our downstream tool, we bind in a new volume with our downstream analysis scripts and an entrypoint to invoke them. This allows us to easily exchange different analysis scripts and compare artifacts for different downstream tools. In this work, we use Ghidra analysis scripts to dump the functions and instruction mnemonics found in each function from Ghidra's auto analysis. A limitation of

[1] https://github.com/NationalSecurityAgency/ghidra/blob/master/Ghidra/Features/BSim/src/main/help/help/topics/BSim/IngestProcess.html

[2] https://github.com/NationalSecurityAgency/ghidra/releases

[3] https://github.com/sandialabs/ghidra-galore

our current test harness is that these analysis scripts must work across all versions of Ghidra. If users wish to use more recently available APIs, they will be limited to those versions of Ghidra that support it.

Our choice of downstream analysis tool, a code similarity tool, enables a differential analysis of the artifacts for the same input binary. Specifically, we can process the same input binary using multiple versions of Ghidra and then use code similarity to compare the outputs. If the outputs between two versions of Ghidra produce no changes in the downstream tool (*i.e.*, their similarity is 100%) then there were likely no change of importance between those versions of Ghidra for that specific input file. Thus, we can use differences in the similarity to focus our attention to notable tuples of versions and input file. We can then compare the observed changes in the artifacts with the changelog in the upstream tool to identify likely causes of the difference.

Our code similarity tool supports multiple different feature representations which allows us to explore how changes in the upstream tool may affect similarity tools and other analysis tools that use different granularities. Specifically, we utilize three granularities in this work:

- NGRAMS: Sliding window across function mnemonics
- BASICBLOCKS: In-order hash of mnemonics within each basic block
- FUNCTIONS: In-order hash of mnemonics within each function

These three granularities capture the basic syntactic structure of the disassembly and are unaffected by changes to memory layouts, registry mappings, and function order.

IV. EXPERIMENT

Our primary experiment uses malware from the Malpedia dataset [10]. We chose this dataset because malware is notorious for its obfuscation, making disassembly harder. Conducting the same experiments over a benign dataset (*e.g.*, BinKit 2.0 [6]) would be illuminating for many Ghidra users.

Malpedia is curated by an invite-only community that focuses on the breadth of malware samples rather than volume of binaries. We use a snapshot of the dataset from 2023-08-11 which contains ∼9,000 samples across ∼2,400 families. We focus on the Windows "unpacked" samples (a total of 3,751 binaries). For many samples, Malpedia includes an "unpacked" version of the input but the unpackings are not always correct (*e.g.*, some of the unpacked binaries are SHA256-identical to the file they were supposedly unpacked from). We use this dataset as-is and do not try to curate it further.

We use our test harness to create containers for 30 versions of Ghidra (all versions available as of March 2024 with the exception of 9.0 which did not build and we did not attempt to fix). Due to changes in Java runtime requirements in Ghidra, we use JDK-11 for versions of Ghidra before 10.2 and JDK-17 after (note that starting in Ghidra 11.2, the required JDK version increases again to JDK-21 but these versions were not included in our experiments). We use *eclipse-temarin* containers as the base image and leave experimenting with different JDK distributions and JDK versions to future work (although we do not expect any significant findings other than timing due to the Java developers approach to compatibility[4]). We ran these 30 versions across our dataset of 3,751 "unpacked" Windows binaries, collecting the outputs as described in the previous section, totalling 112,530 results. We ran Ghidra with the default parameters, analysis flags, and memory limits for each version via the provided *analyzeHeadless* script.

We conducted secondary experiments with other subsets from Malpedia dataset including three sets of non-Windows files. This utilized the same harness, albeit with a different input file type. Ghidra's auto-import feature detects the file type of the input which simplifies this process. We analyze three different file types: *APKs* from Android Applications, *Mach-O* binaries from OSX, and *ELF* binaries from *nix operating systems. There are far fewer files in Malpedia for these file types than the "unpacked" Windows files. Specifically, there are 391 *APKs*, 220 *Mach-Os*, and 563 *ELFs*.

V. RESULTS

In this section, we detail the results of our experiment. We first explore how analysis time, timeouts, and fatal errors change across versions. These simple metrics give us a broad understanding of how different versions of Ghidra handle importing the same set of input files. Then, for a more detailed view, we select four input files to serve as case studies. We note how their pairwise similarity changes across versions, including how those similarities vary with different granularities. We note how our downstream analysis (*i.e.*, pairwise similarity) changes across versions and examine the differences in Ghidra's auto-analysis as case studies. Next, we look at the effects of these changes on a higher-level analysis, clustering. Finally, we briefly explore trends in our simple metrics for non-Windows binaries.

A. Analysis Metrics

In Figure 1, we show a summary for the analysis of the "unpacked" Windows binaries from Malpedia across the different versions of Ghidra. First, we show boxplots for the total time to process the binaries. Overall, we see a slight increase in the analysis time but the vast majority of input files take under a minute to analyze. We show the number of analysis timeouts with a generous maximum analysis time of 20 minutes. Ghidra has a vanishingly small number of timeouts (at most 7 in each of v10.3.3 and v10.4) across the dataset of 3,751 input files. Finally, we show the number of files with a fatal error (where an exception was thrown). There was a noticeable uptick in the number of fatal errors in v10.0.3 which we partially attribute to disabling the *ContinueInterceptor* which allowed Ghidra to catch exceptions while processing the file headers and continue past corrupt headers. Many of the bugs that crashed Ghidra were fixed in the subsequent versions.

We found a total of 152/3,751 files for which Ghidra was unable to produce an artifact for any version. We briefly

[4]https://wiki.openjdk.org/display/csr/Kinds+of+Compatibility

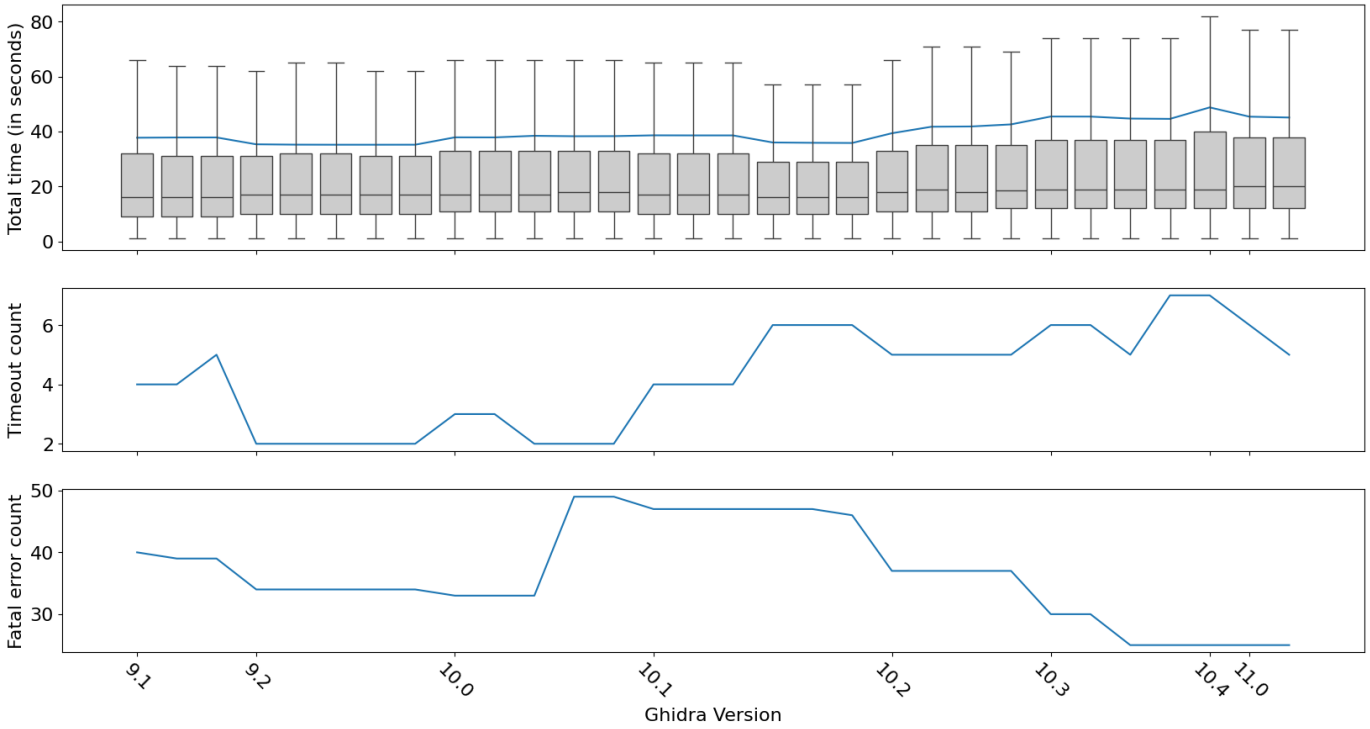


Fig. 1: Comparing summary metrics for "unpacked" Windows binaries across Ghidra versions. Top: boxplots of the total analysis time (in seconds) with the average in blue and outliers excluded. Middle: number of analysis timeouts (using a 20 minute timeout). Bottom: number of fatal errors. All plots share the same x-axis.

explored these errors and found that a majority of them were due to the input files not being binaries. Specifically, we found a number of scripts and other non-executable file types. These files fail to import but do not throw an exception. Only a subset of 13/152 were identified as executables by the "file" utility. We leave further analysis of these executables to future work.

B. Pairwise Similarities

In Figure 2, we show the pairwise similarities for four input files across the different versions of Ghidra, as measured by our code similarity tool (using basic block granularity). These four input binaries were selected from the full dataset of samples because they were representative of several of the different observed patterns. We will explore each as a case study in the following sections.

We note the pattern shown in Figure 2b, represents the pattern that we expected to see at the outset – artifacts are identical for some number of versions of Ghidra with periodic discontinuities when Ghidra's analysis changes for the file. We can see that as we get further from the diagonal the similarity decreases (*i.e.*, when comparing across many versions).

1) Case Study: win.redsalt: Figure 2a shows the pairwise similarities for a file from the *win.redsalt* family[5]. This PE32 file was added to Malpedia on 2020-01-17 (*ca.*, Ghidra v9.1.1).

The artifacts for this input file are identical across all versions of Ghidra – meaning that Ghidra has either (optimistically) completely solved this input file since its early versions or (pessimistically) has made no progress. We manually reviewed the file using Ghidra 11.0.2 and found that there was very little obfuscation in the file. Specifically, there were obfuscated strings and what appears to be a routine to deobfuscate them but the control flow was relatively trivial to follow. The sample's imports were also not obfuscated. This means that Ghidra has solved this input file and the file could be used to check for regressions (*i.e.*, when new versions stop producing the same artifact).

2) Case Study: win.pitou: Figure 2b shows the pairwise similarities for a file from the *win.pitou* family[6]. This PE32+ file was added to Malpedia on 2019-04-17 (*ca.*, Ghidra v9.0.3). There are several discontinuities for this file – at versions 10.2, 10.3, 10.3.1, 10.4, and 11.0.

Between v10.2.3 and v10.3, we found that Ghidra changed how it handled the LOCK prefix in x86. Specifically, in v10.2.3, the LOCK prefix was listed as a separate "instruction" (*i.e.*, LOCK, INC) and in v10.3, the prefix was correctly applied to the modifier to the instruction (*i.e.*, INC.LOCK)[7].

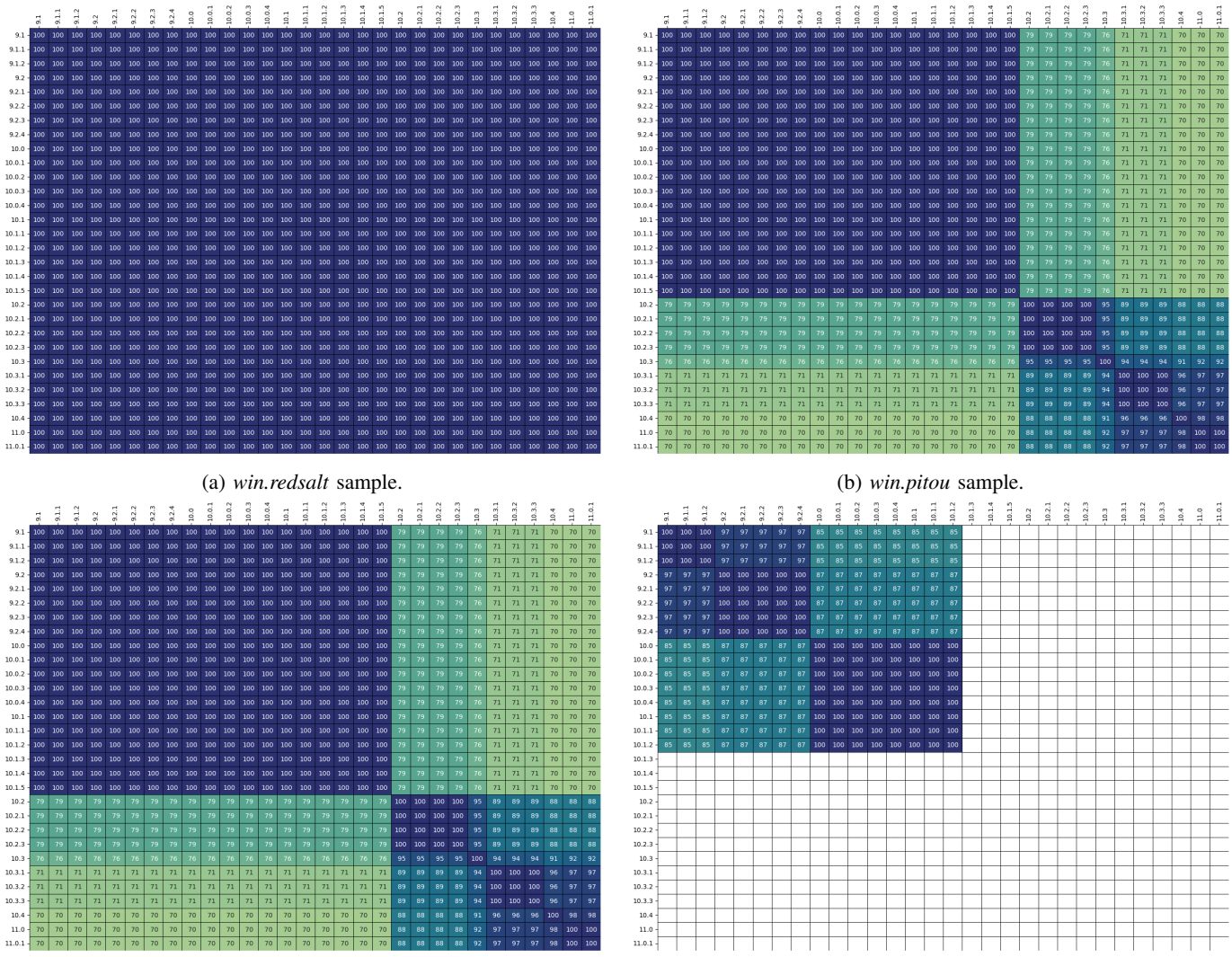
Between v10.3 and v10.3.1, we found that Ghidra fixed a bug when decoding POPF and PUSHF instructions[8]. This contributed to the change in similarity between the two versions.

[5]https://malpedia.caad.fkie.fraunhofer.de/details/win.redsalt

[6]https://malpedia.caad.fkie.fraunhofer.de/details/win.pitou

[7]https://github.com/NationalSecurityAgency/ghidra/pull/2033

[8]https://github.com/NationalSecurityAgency/ghidra/issues/4980



(a) *win.redsalt* sample.

(b) *win.pitou* sample.

(c) *win.bazarbackdoor* sample.

(d) *win.ghambar* sample.

Fig. 2: Heatmaps of pairwise similarities for the same input binaries as analyzed by multiple versions of Ghidra (using basic block granularity). Similarities range from 70% in green to 100% in dark blue. Cells are empty when Ghidra did not produce an output for one or both versions.

Between v10.1.5 and v10.2, v10.3.3 and v10.4; and v10.4 and v11.0, we found multiple changes in the number of functions and which functions the basic blocks "belonged" to. We explored the source of these change and traced it back to a change in Ghidra's options for how it handles non-contiguous functions. Specifically, in v10.2, Ghidra changed the default option[9] from FALSE to TRUE[10]. This changes how Ghidra interprets tail calls and creates an additional function for target of a tail call. In v10.4, Ghidra further refined function body creation when functions overlap to favor contiguous functions.

3) Case Study: win.bazarbackdoor: Figure 2c shows the pairwise similarities for a file from the *win.bazarbackdoor* family[11]. This PE32+ file was added to Malpedia on 2021-09-09 (*ca.*, Ghidra v10.0.3). The discontinuities for this file are at v10.0, v10.0.3, v10.2, and v10.3.

Between v9.2.4 and v10.0, the similarity drops to 95%. When we looked at the artifacts, we found several changes that contributed to this drop. First, we found an additional instruction in functions from v9.2.4 that was not present in v10.0 (*e.g.*, an INT3 after a CALL). We found that the call was to a non-returning function (*e.g.*, *Kernel32:ExitThread*) that was added to Ghidra in v10.0[12], meaning that it was unreachable and was added as padding. These instructions were correctly omitted in v10.0. Second, we found a difference in the basic block boundaries that Ghidra identified within

[9]OPTION_DEFAULT_ASSUME_CONTIGUOUS_FUNCTIONS_ENABLED

[10]https://github.com/NationalSecurityAgency/ghidra/pull/4573

[11]https://malpedia.caad.fkie.fraunhofer.de/details/win.bazarbackdoor

[12]https://github.com/NationalSecurityAgency/ghidra/pull/2111

several functions. Specifically, in v10.0, Ghidra splits a basic block in several functions into two basic blocks. We were unable to find the specific cause of this change but note that it only affects the basic block granularity.

In v10.0.3, Ghidra started to crash on this particular input file, producing no artifact for downstream analysis. Specifically, the PE loader encountered a null pointer exception while parsing the PE header. In previous versions of Ghidra, the *ContinuesInterceptor* would have allowed Ghidra to continue past this error but it was disabled by default in v10.0.3 and removed in v10.2. The specific exception that caused this crash was fixed in v10.2[13].

Between v10.2.3 and v10.3, we again attribute the drop in similarity to changes with the LOCK prefix in x86 that we discussed in Section V-B2.

4) Case Study: win.ghambar: Figure 2d shows the pairwise similarities for a file from the *win.ghambar* family[14]. This PE32 .NET file was added to Malpedia on 2023-02-16 (*ca.*, Ghidra v10.2.3). There are several discontinuities for this file – at v9.2, v10.0, and v10.1.3.

Between v9.1.2 and v9.2, the similarity of the artifacts drops to 97%. We dug into the artifacts and discovered a "new" mnemonic in v9.2 – INT3. INT3 is a one-byte-instruction used by debuggers to set a breakpoint as well as for padding. In Ghidra v9.1.2, the instruction was listed as INT 3 (note the space), conflating the disassemble of 0xCC (INT3) with 0xCD03 (INT). This was a relatively trivial change[15] to disentangle these two instructions.

Between v9.2.4 and v10.0, there was an even larger drop in similarity – to 87% (cumulatively, 85% between v9.1 and v10.0). In Ghidra v10.0, there were multiple improvements to the handling of .NET binaries that resulted in fairly significant changes to the function boundaries that were extracted by our analysis script. Further analysis is required to identify the root cause of these changes.

Finally, between v10.1.2 and v10.1.3, the similarity drops to 0%. This is because Ghidra started to time out when analyzing the file (with a timeout of 20 minutes), producing an empty artifact. Specifically, the "Decompiler Parameter ID" analyzer jumps from 146 second in v10.1.2 to 902 seconds in v10.1.3. It continues to increase from there, maxing out in the last tested version of Ghidra (v11.0.1) at 1458 seconds.

C. Similarity Granularities

In the previous section, we explored how similarities changed for several case studies using the basic block granularity for similarity. But the changes in Ghidra that we explored affect the input files at multiple granularities which we wish to explore further. Fortunately, our code similarity tool supports multiple granularities (*i.e.*, NGRAMS, BASICBLOCKS, and FUNCTIONS) as described in Section III.

In Table I, we show how the artifacts compare across our three granularities for each of the case study samples. We can see that, on average, the FUNCTIONS granularity is the most sensitive to changes in the Ghidra version with the similarity dropping as low as 6% for a *win.pitou* pair. In this limited testing, NGRAMS and BASICBLOCKS both demonstrate lower sensitivity, with BASICBLOCKS being the least sensitive.

D. Clustering

In the results explored so far, we have shown how changes in Ghidra affect individual input files. In this section, we explore how the changes affect a higher-level task, namely, clustering. Specifically, we build clusters for the artifacts produced by each version of Ghidra and then compare the clusterings across the versions. We use the BASICBLOCKS granularity, as it was the least sensitive to variations in Ghidra versions based our limited case studies in the previous section. We use DBSCAN [1] with a min points parameter of 2 (a file must have at least one neighbor to form a cluster) and a threshold of 75% to produce our clusterings. DBSCAN is a density-based clustering algorithm and these parameters allow us to explore the effects of Ghidra versions even on relatively low-density clusters. In a real-world deployment, the parameters would likely be set to identify clusters with higher density but would have many more data points to contribute to that density. In this analysis, we are interested in the stability of our clusters across Ghidra versions for a particular value of min points and threshold. In future work, we could vary and/or tune these parameters within/across versions to better understand the parameter stability across versions.

In Table II, we can see a summery of the clusterings across the notable versions of Ghidra (omits micro versions for the sake of space). The number of clusters is quite stable (varying by at most 4 between adjacent versions). The number of clusters with a single family label is also quite stable. The overall trend shows fewer clusters (and cluster members) across the different versions of Ghidra. We leave the investigation of this phenomenon to future work.

In Table III, we show the summary of the changes between the clusterings from Ghidra v10.1.5 and v10.2. We generated these summaries for each pair of adjacent versions in order to understand which individual samples affected our clustering results. This figure shows that between these two versions, there are two new clusters, one lost cluster, and one lost cluster member. From our case study in Section V-B3, we know that at least one of the samples in the *win.bazarbackdoor* family all had fatal errors in v10.1.5 that was fixed in v10.2. We confirmed that the other three samples had similar crashes. The samples in the *win.colony* family are in-feature-space identical to each other in v10.2 but not v10.1.5 which causes our code similarity tool to count them as the same file. This means that they fail to meet the min points threshold (a surprising corner case for in-feature-space deduplication). Finally, the file from the *win.dbatloader* family drops out of the cluster because its similarity of its nearest neighbor drops 2% in v10.1.5 from 75.5% to 73%, just missing the threshold (these files represent slightly different versions of the same tool from the family and

[13]https://github.com/NationalSecurityAgency/ghidra/pull/4161
[14]https://malpedia.caad.fkie.fraunhofer.de/details/win.ghambar
[15]https://github.com/NationalSecurityAgency/ghidra/pull/872

Family	Version	NGRAMS	BASICBLOCKS	FUNCTIONS	Primary Change
win.pitou	10.2.3 → 10.3	96%	95%	92%	LOCK Prefix
win.pitou	10.3 → 10.3.1	91%	94%	89%	POPF/PUSHF
win.pitou	10.1.5 → 10.2	41%	79%	6%	Contiguous functions
win.pitou	10.3.3 → 10.4	82%	96%	62%	Contiguous functions
win.pitou	10.4 → 11.0	93%	98%	89%	Contiguous functions
win.bazarbackdoor	9.2.4 → 10.0	96%	95%	95%	*INT* to *INT3*, Non-returning functions, Block boundaries
win.bazarbackdoor	10.2.3 → 10.3	98%	98%	93%	LOCK Prefix
win.ghambar	9.1.2 → 9.2	99%	97%	94%	*INT* to *INT3*
win.ghambar	9.2.4 → 10.0	94%	87%	77%	.NET improvements

TABLE I: Similarities across multiple granularities for case study samples from Section V-B along with a brief description of the primary change. Omits rows where the change was due to Ghidra crashing or timing out.

Version	Clusters	Members
9.1	376 (298)	1,446
9.2	374 (299)	1,444
10.0	375 (300)	1,447
10.1	372 (298)	1,441
10.2	368 (294)	1,433
10.3	366 (294)	1,434
10.4	367 (295)	1,433
11.0	365 (294)	1,430

TABLE II: Clustering metrics for the notable versions of Ghidra (omits micro release versions). The number of clusters includes the number of clusters with a single family label (*e.g.*, *win.bazarbackdoor*) in parenthesis.

Δ	cluster	hash	family	matches	identical
+	64	94719f43	*win.bazarbackdoor*	N/A → 1	N/A → 0
+	64	132e2d9a	*win.bazarbackdoor*	N/A → 1	N/A → 0
+	94	38dbc338	*win.bazarbackdoor*	N/A → 1	N/A → 0
+	94	935383f2	*win.bazarbackdoor*	N/A → 1	N/A → 0
-	181	d3625494	*win.dbatloader*	3 → 3	0 → 0
-	257	03af897d	*win.colony*	1 → 1	0 → 1
-	257	e66656ba	*win.colony*	1 → 1	0 → 1

TABLE III: Summarized clustering changes between 10.1.5 and 10.2. This shows two new clusters (64, 94), one lost cluster (257), and one lost cluster member (181).

changes to the contiguous functions as seen in Section V-B2 could have produced the 2% difference in similarity).

Overall, we find that the Ghidra version only affects a small subset of the clusters. This makes sense as input files that are similar will be analyzed similarly, even if there are bugs and/or deficiencies that affect the analysis. For example, for the *win.pitou* sample discussed in Section V-B2, all files in the family that include the LOCK prefix as a separate instruction in v10.2.3 will include it as a prefixed instruction in v10.3. If we do not mix versions of Ghidra, then the artifacts will produce consistent analyses of the files.

In Figure 3, we show a high-level depiction of the changes to the clusterings across all versions of Ghidra. Each node represents a cluster with the X-position determined based on the version of Ghidra. The Y-locations are arbitrary and ordered based on the input files contained within the cluster. Edges show an overlap in the input files between adjacent versions. The longer arcs indicate a cluster that disappeared for some number of versions of Ghidra (*i.e.*, crashed for some number of versions like the *win.bazarbackdoor* sample explored in Section V-B3). There are also clusters that begin "late" or end "earlier" in the sequence of Ghidra versions (*i.e.*, timed out for some versions like the *win.ghambar* sample explored in Section V-B4).

We also wished to explore a more real-world example of clustering where we use the artifacts produced by the version of Ghidra available at the time the input file was added to Malpedia. Specifically, we use the metadata in Malpedia to determine when the input file was added to Malpedia and then find the most recent release of Ghidra to use the artifact from that version. This assumes that we always update Ghidra versions as soon as a new release is available and that the time input files were added to Malpedia is a reasonable proxy for when they were discovered and first analyzed. We note that a large portion of the Malpedia dataset predates the Ghidra v9.1 release and thus use the artifacts from the v9.1 release for those 1,896 (out of 3,751) input files. We found that the number of clusters (366) was relatively consistent with those produced by a single version of Ghidra but the number of cluster members was significantly less (1,336). We leave the investigation of this to future work.

E. Non-Windows Files

In the previous sections, we explored Ghidra's behavior on "unpacked" Windows files. In this section, we briefly investigate whether there are notable differences for three other file types: *APKs*, *Mach-Os* ("osx"), and *ELFs*.

In Figure 4, we first show the average analysis time for each file type. We see a slight increase in the analysis time for *Mach-Os* while there is a slight decrease in the analysis time for *APKs* and *ELFs*. The overall analysis time is longer for *Mach-Os* and *ELFs* than the "unpacked" Window files. Next, we show the number of analysis timeouts (again with a generous maximum analysis time of 20 minutes). Again, the number of timeouts was vanishingly small (and non-existent for *APKs*). Finally, we show the number of files with a fatal error for each file type. We had to use a logarithmic scale for the y-axis because in v10.2, Ghidra added support for a new

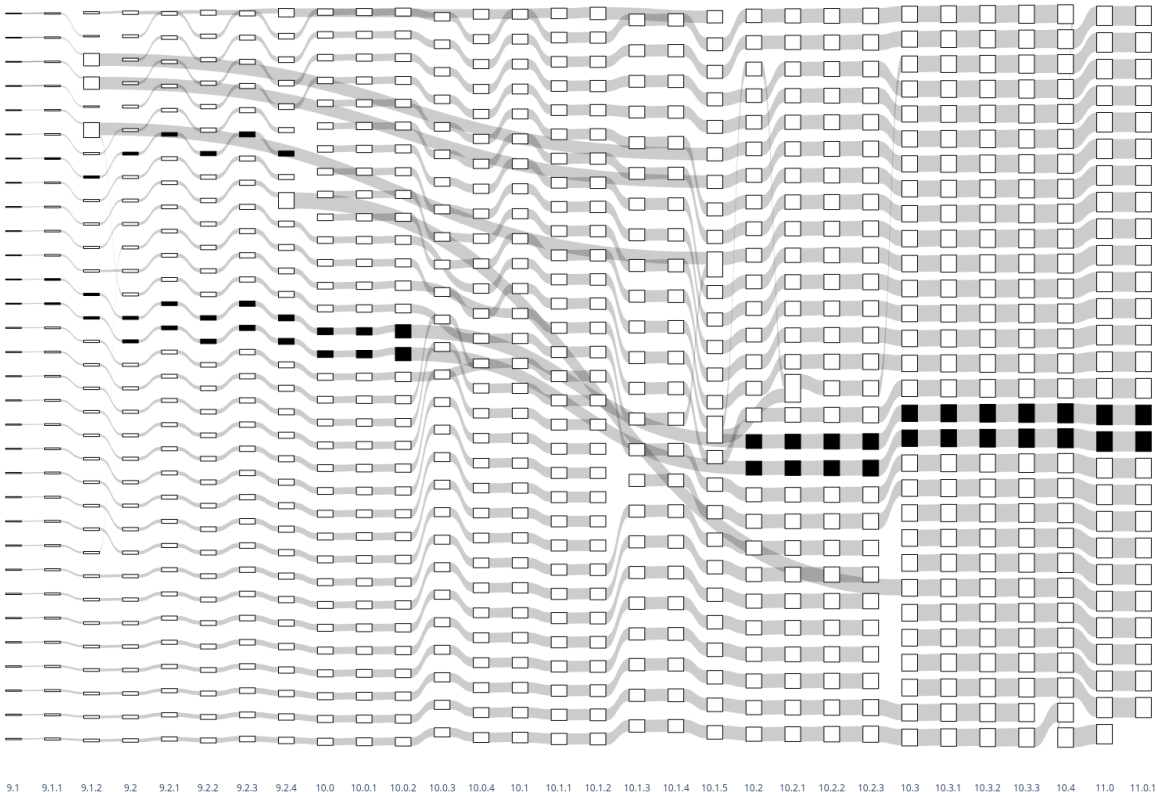


Fig. 3: Subset of clusters with changes across all versions. Edges show sample overlap between clusters with X position based on Ghidra version (shown at the bottom of the figure). The black clusters contain samples from the *win.bazarbackdoor* family studied in Section V-B3.

APK loader to load all DEX files but that crashed on many of the *APKs* in Malpedia due to a *NullPointerException*. This was later fixed in v10.3.0[16] when they refactored the *Loaders* and *AutoImporter* to support loading more than one program.

This terse investigation into Ghidra's behavior on non-Windows files aims to highlight a challenge that Ghidra (and other disassemblers) face – it seeks to disassemble a diverse set of executable formats, from multiple compilers, across multiple architectures and progress in one domain may cause regressions in another. Therefore, the "best" version of Ghidra may be dependent on the input file.

VI. Future Work

There are many future directions for this work, broadly grouped into three categories: 1) repeat with different datasets, 2) repeat with different upstream tools, 3) repeat with different downstream tools.

A high priority for datasets is the dataset from Pang *et al.* [9]. For the input files in their dataset, they embed ground truth which makes it possible to quantify whether or not Ghidra (and other disassemblers) are improving over time. We spent some time exploring their code and trying to integrate it with their own but found that their use of Protocol Buffers[17] and their mechanism for installing the library were difficult to replicate in containers across all versions of Ghidra. This is surmountable with additional time. Datasets to explore how changes in Ghidra affect specific workflows for specific file types is another area to explore.

There are numerous other disassemblers and lifters that would be worth exploring. Notably, IDA Pro [12], a commercial disassembler, has over 50 versions as of May 2024 and is popular in the malware reverse engineering space. Code similarity tools are just a small part of the downstream tools. Other downstream tools (*e.g.*, CAPA [2] and cwe-checker [3]) should be explored as well.

Ghidra and many other upstream tools support different options which provides yet another possible direction to explore – how different options in the upstream tool affect the

[16]https://github.com/NationalSecurityAgency/ghidra/issues/4929

[17]https://developers.google.com/protocol-buffers

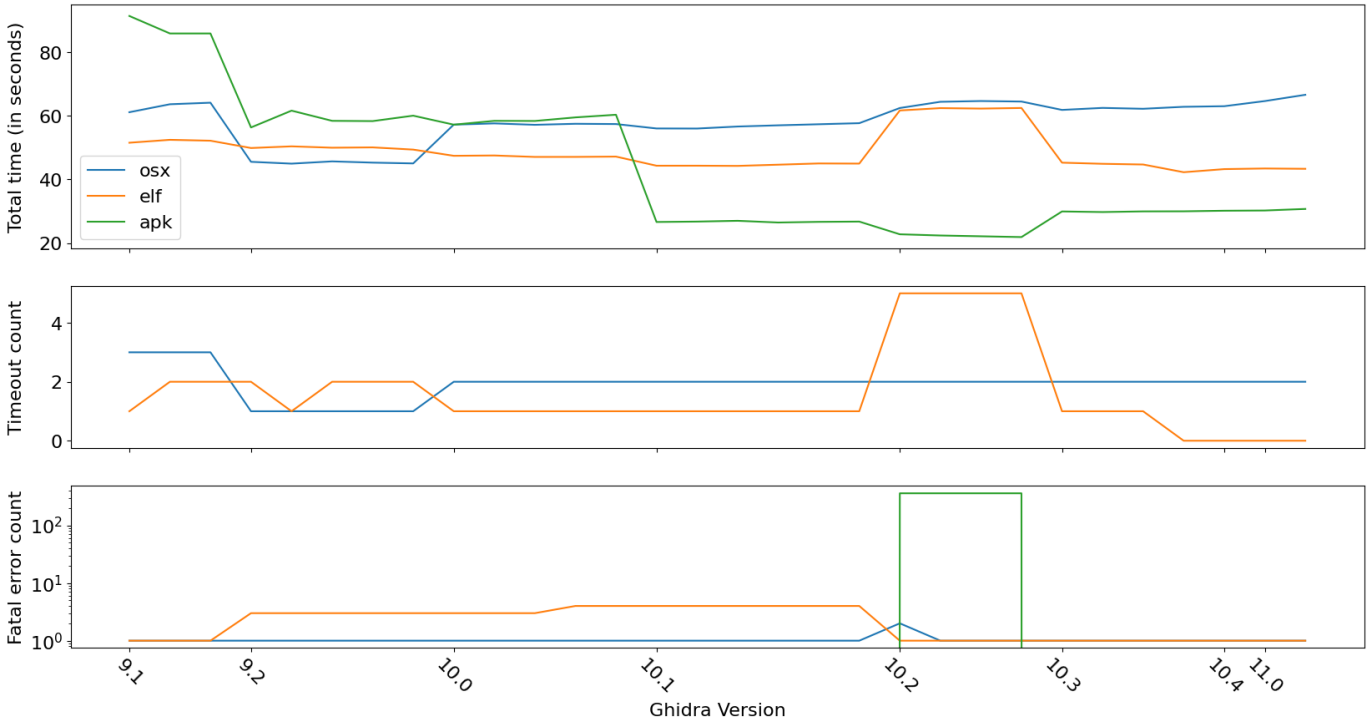


Fig. 4: Comparing summary metrics for non-Windows files across Ghidra versions. Top: average analysis time (in seconds). Middle: number of analysis timeouts (using a 20 minute timeout), there were no *APK* timeouts. Bottom: number of fatal errors (note: log scale for y-axis). All plots share the same x-axis.

downstream tools? There are far too many permutations to explore and every organization will have different priorities – this will take a community-level effort to address. We envision a common test harness (or set of test harnesses) to enable organizations to plug in their tools and datasets of choice.

An additional separate area for future work is in mitigation. Specifically, can we detect input files that would have been analyzed differently in a previous version of the upstream tool? This could help alert the reverse engineer that something about the input file "smells" off without prescribing a fix. We envision something akin to "code smells"[18] for binaries. At best, these "binary smells" could be helpful to identify a subset of files to re-analyze with newer versions of the upstream tool. Alternatively, enhanced tracing of the decision making process in the upstream tools would further enable reverse engineers to build mitigations outside of the upstream tool itself. However, this would require some consistency in the tracing format across versions.

VII. Conclusion

In this paper, we examined the evolution of Ghidra through its various releases and assessed how these changes impact the performance of our downstream code similarity tool. Our analysis of the same input files revealed significant variations in basic metrics such as analysis time, error counts, and the number of functions processed across different versions of Ghidra. While Ghidra has generally improved over time, the nuanced nature of "better" in the context of malware analysis is evident.

The case studies we conducted highlighted that Ghidra's effectiveness can vary depending on the specific file being analyzed. This variability emphasizes the importance of context when evaluating tool performance. We encourage practitioners in the field to approach version upgrades with caution, recognizing that the latest release may not always yield superior results for their specific use cases.

Ultimately, our findings contribute to a more informed discourse within the community regarding the adoption of new tools and versions. By emphasizing the need for careful evaluation and consideration of the unique characteristics of each version, we aim to foster a more nuanced understanding of how advancements in disassembly tools like Ghidra can influence downstream analysis processes. Future work should continue to explore these dynamics, ensuring that the tools we rely on for binary software analysis evolve in a way that truly enhances their utility and effectiveness.

Acknowledgements

We appreciate the feedback from our anonymous reviewers, as well as from Philip Kegelmeyer, the MTEM community, and the SUNS community, who provided input on earlier iterations of this work.

This work was supported by DHS S&T and IARPA at Sandia National Laboratories, a multi-mission laboratory man-

[18] https://martinfowler.com/bliki/CodeSmell.html

aged and operated by National Technology & Engineering Solutions of Sandia, LLC (NTESS), a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration (DOE/NNSA) under contract DE-NA0003525. This written work is authored by an employee of NTESS. The employee, not NTESS, owns the right, title and interest in and to the written work and is responsible for its contents. Any subjective views or opinions that might be expressed in the written work do not necessarily represent the views of the U.S. Government. The publisher acknowledges that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this written work or allow others to do so, for U.S. Government purposes. The DOE will provide public access to results of federally sponsored research in accordance with the DOE Public Access Plan.

REFERENCES

[1] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, volume 96, pages 226–231, 1996.

[2] Inc. FireEye. Capa. https://github.com/fireeye/capa, 2024.

[3] FKIE-CAD Fraunhofer FKIE. cwe-checker. https://github.com/fkie/cwe_checker, 2024.

[4] Irfan Ul Haq and Juan Caballero. A survey of binary code similarity. *Acm computing surveys (csur)*, 54(3):1–38, 2021.

[5] Intel. Cve binary tool (cve-bin-tool). https://github.com/intel/cve-bin-tool, 2024.

[6] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *IEEE Transactions on Software Engineering*, 49(4):1661–1682, 2022.

[7] National Security Agency. Ghidra. https://github.com/NationalSecurityAgency/ghidra, 2024.

[8] National Security Agency. Ghidra: Bsim tutorial. https://github.com/NationalSecurityAgency/ghidra/blob/master/GhidraDocs/GhidraClass/BSim/README.md, 2024.

[9] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE symposium on security and privacy (SP)*, pages 833–851. IEEE, 2021.

[10] Daniel Plohmann, Martin Clauss, Steffen Enders, and Elmar Padilla. Malpedia: a collaborative effort to inventorize the malware landscape. *The Journal on Cybercrime and Digital Investigations*, 3(1):1–19, 2017.

[11] Ann Marie Reinhold, Travis Weber, Colleen Lemak, Derek Reimanis, and Clemente Izurieta. New version, new answer: Investigating cybersecurity static-analysis tool findings. In *2023 IEEE International Conference on Cyber Security and Resilience (CSR)*, pages 28–35. IEEE, 2023.

[12] Hex-Rays SA. Ida pro. https://www.hex-rays.com/products/ida/, 2024.

[13] Sri Shaila, Ahmad Darki, Michalis Faloutsos, Nael Abu-Ghazaleh, and Manu Sridharan. Disco: Combining disassemblers for improved performance. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 148–161, 2021.

[14] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.