

When GANs meet LLMs: Bridging the Feature-Problem space gap for efficient adversarial ML-based malware generation

Bhavan Dondapati
bhavan.dondapati@tamu.edu

Manoj Reddy Gurram
manoj850@tamu.edu

Marcus Botacin
botacin@tamu.edu

Texas A&M University, College Station, Texas, USA

Abstract

Malware detection is a major security problem that has been promisingly tackled via Machine Learning (ML) approaches. The viability of using ML for malware detection has been challenged by the construction of Adversarial Examples (AEs) that can bypass ML-based detectors. The State-Of-The-Art (SOTA) in AE generation is the use of Generative Adversarial Networks (GANs). The major drawback of GAN-based malware generation is that changes in the feature space rarely map back to the problem space, i.e., the proposed changes cannot be converted into actual binaries. At the same time, the world was introduced to new Large Language Model (LLM) solutions. Previous work has demonstrated that LLMs can generate malware from behavioral descriptions. We here propose coupling the two technologies to make the creation of viable AEs efficient. We believe that the adversarial feature space generated by a GAN can be converted to a real binary in the problem space by an LLM-powered generative AI. To test this hypothesis, we implemented 3 malware-generation GANs and integrated them with GitHub Copilot to generate and compile code. We let the GANs inform GitHub Copilot about the need to add or remove functions to/from a binary to implement a *mimicry* attack. We generated thousands of malware variants informed by GAN's decisions and discovered that they can bypass ML detectors and real AVs.

Keywords: This is the archived version of the paper.

1 Introduction

Malicious Software (Malware) is a major security threat to modern computer systems, motivating the development of myriad detection solutions. The most successful solutions for malware detection currently rely on machine learning (ML) for binary classification [15]. The successful use of ML, however, has been challenged by the creation of Adversarial Examples (AEs) [21] that fool detection systems, i.e., malicious constructions classified as benign by ML systems.

Multiple AE creation strategies have been proposed over time [17, 20], with Generative Adversarial Networks (GANs) being currently the State-Of-The-Art (SOTA) approach. GANs consist of two neural networks (a generator and a discriminator) that fight each other for the best learning [23]. In the AE malware case, generators produce new malware samples, and discriminators detect them. A well-trained generator can point out a set of features that, if present in a sample, make it bypass a detector similar to the trained discriminator (attack transference [22]).

AE generation using GANs has a major drawback: its manipulation occurs in feature space (feature vectors), whereas the malware detector operates on the binary file (the problem space), creating a gap (the feature-problem space gap [42]). This gap means that some constructions possible in the feature space cannot be transferred to the problem space. For instance, a GAN might suggest that incorporating two given libraries could make the sample evasive, but if these two libraries are incompatible (e.g., different endianness, architectures, dependencies, etc.), a working binary cannot be generated. GAN’s limitation to operate in the feature space has led to the wrong impression that GAN-generated malware was not a primary source of concern in practice, such that whereas many security actors, such as AntiVirus (AVs) companies, were aware of GAN malware [11], they had believed so far that the widespread use of GANs is not practical yet given the need for inverse feature generation.

However, at present, we observe the emergence of Large Language Models (LLMs) [10, 31, 27], powerful ML models capable of generating text and even code in many programming languages [19]. Previous work on LLM suggested that LLMs could be used to write malware [25]. In this case, many security players, including AV companies, believe that LLMs lower the barriers for attacks [47]. Our concern in this paper is even beyond that. We believe that attackers might couple LLMs and GANs to bridge the feature-problem space gap, i.e., using LLMs to generate on-demand code for the feature vector produced by a GAN. Once feasible, attackers could launch large-scale campaigns powered by automated malware generation. This scenario becomes increasingly plausible as Anthropic reports the first AI-generated malware campaign in the wild [4].

Therefore, in this work, we investigate the malware generation process by a combination of GANs and LLMs to understand **(i)** the potential and the limits of their use by attackers when evading Machine Learning (ML)-based and AntiVirus (AV) detection systems; and **(ii)** how to protect users from these threats, such as via using these same technique for data augmentation in adversarial retraining procedures.

Specifically, we test the GAN-LLM hypothesis by implementing a framework to automatically generate malware using GANs and LLMs. Whereas multiple works proposed GANs for malware generation [28, 49, 48], no previous work generated malware on-demand but only adapted existing malware. Also, whereas previous work proposed using GPT-3 for malware generation [7] as a viable strategy, this approach presented multiple constraints, which we here relax by moving the implementation from GPT-3 [10] to GitHub Copilot [26]. We developed an automation tool for the Copilot VSCode extension that automatically inserts prompts and retrieves generated code. It allowed us to couple GitHub Copilot with the GANs as if it were an API. We generated AEs by **(i)** taking the GAN’s boolean vector of which functions the malware should present and **(ii)** generating prompts from it to ask GitHub Copilot to include or not these functions in the generated code. We added functions as dead code and removed them via dynamic loading. We

performed large-scale tests to evaluate our proposal.

Contributions. In summary, our contributions are the following:

- We propose a framework that uses GitHub Copilot as a code-generation API and evaluate its use as a malware-generation tool, complementing prior work that considered only the pure GPT-3 model. We open-source the framework for the community.
- We propose the combined use of GANs and LLMs to bridge the gap between the feature-problem space for automatic malware generation. We deploy this idea within an open-source framework.
- We conduct a systematic evaluation of the proposed solution to characterize the use of GANs and LLMs for malware generation, thereby complementing and broadening the pioneering work in the field.

Findings. In sum, we discovered that:

- GitHub Copilot still requires multiple attempts per malware building block to generate functional code, as with prior GPT-3 work. However, GitHub Copilot generates more blocks and more diverse blocks per request, making it more effective in the long tail. GitHub Copilot-generated samples exhibit a wide range of AV detection rates (from low to high), as do GPT-3’s samples.
- GANs are effective AE generators (up to 99% ASR) even in high-dimensional cases. We extended the generation problem from 150 functions (previous work) to 1500. However, in most cases, naive GANs remove binary functions, thereby breaking their functionality and necessitating our GAN-LLM approach to fix them.
- Coupling GitHub Copilot and GANs for malware generation effectively evades ML models and real AVs (up to 20) by generating binaries with function insertions and removals that preserve their functionality.

Organization. This paper is organized as follows: In Section 2, we present background information on automated malware generation to highlight our contributions; In Section 3, we presented related works to better position our contributions; In Section 4, we introduce our developments for coupling GANs and LLMs for malware creation; In Section 5, we evaluate the proposed solution; In Section 6, we draw our conclusions.

2 Background

In this section, we recap the working of GANs, LLMs, and malware generators to foster the understanding of our proposed solution.

2.1 GANs for malware generation

The general structure of any AE generation process consists of a generator that takes malware samples detected as such as input and modifies them so they are detected as goodware by a target detector. The more guided the generation process is, the more efficient the process, i.e., the more evasive attacks are generated with fewer attempts. A challenge in real scenarios is obtaining information about the target detector to guide operations. In most cases, target detectors are black-box models with limited query access. To overcome this limitation, substitute models are used aiming attack transference.

Figure 1 shows the overall architecture of GAN in the context of malware generation. In sum, the generator generates new malware samples/features, and the discriminator detects them. The

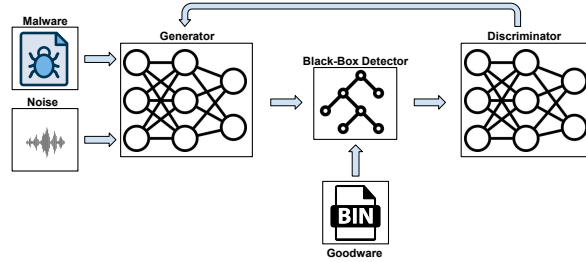


Figure 1: **GAN Architecture.** MalGAN [28] used as a reference.
GAN Architecture.

generator takes a malware sample and some noise as input. The noise is used to perturb the malware sample/feature vector. The new malware sample and the goodware samples are labeled by a target detector. The labels provided by the detector are used to train a discriminator. The discriminator works as a substitute model for the target model. The discriminator’s information is used as feedback to guide the generator’s perturbations. The trained discriminator may later be used as a standalone malware detector.

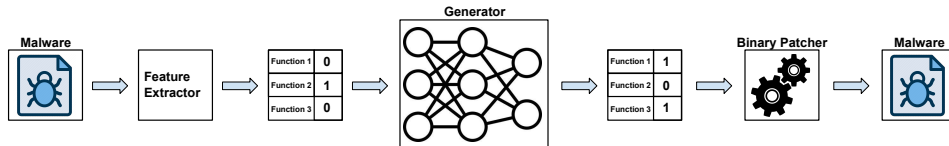


Figure 2: **Malware generation using the generator.** The functions present in the feature vector generated by the generator must be added to the binary via a patching mechanism.

Malware generation using the generator.

The inputs to GANs are typically features extracted from malware and goodware binaries. However, the modifications performed by the generator happen in the feature vector. To create real malware, the feature vector must be converted back into a valid binary, as shown in Figure 2. A major drawback of this approach is that there is no guarantee that such mapping exists. For instance, if the GAN represents binary functions in a feature vector, a generator might identify that removing that feature makes the binary evasive. This modification does not map to a viable real-world binary if removing the function would break the binary’s functionality. The solution for this problem in the literature so far has been to avoid removing features, but this limits the GAN’s power. Similarly, problems arise when adding new features, since functions and libraries present in a feature vector might not be compatible with the actual binaries. We proposed solving these problems by using an LLM to generate code for the feature vector’s functions on demand, preserving both functionality and stealth.

2.2 LLMs for malware generation

Many works proposed LLMs for malware generation (Sec. 3). Their common challenge is prompt engineering, i.e., defining the instructions to provide to LLMs to guide code generation. In this work, we adopt and extend the prompt engineering proposed by Botacin [7], as it is the first strategy formalized in the academic literature. Thus, it is key to understand the assumptions of this previous work to understand our methodology, so here we present our understanding of the key findings of Botacin’s work:

- It designed prompts to generate representative malware behaviors. We reuse these prompts as seeds for our generation.

- It focused on generating C code that relies on the native Windows API. We follow this same practice.
- It developed a divide-and-conquer strategy to split malware generation into 10 building blocks that don't trigger LLM's alignment violations. We use this same strategy in this work.
- It built, compiled, and tested the building blocks both individually and as part of a complete malware sample. We repeat this strategy here.
- It eliminates hallucination issues in functional testing. We adopt the same strategy.
- It is assumed that the LLM-based malware generation strategy would be used as an exploit kit, relying on post-processing scripts to fix minor generation bugs and assemble the final malware code. Here, we repeat this strategy and adopt the post-processing fixes.
- It creates a strategy in which individual building blocks are combined exponentially to produce malware variants, which are then compiled into a functional binary. We also adopt this same strategy.

In sum, we repeat the generation strategy of previous work, but instead of using a fixed prompt, as in previous work, we propose that the developed GANs refine our prompts using a list of (non-)allowed functions.

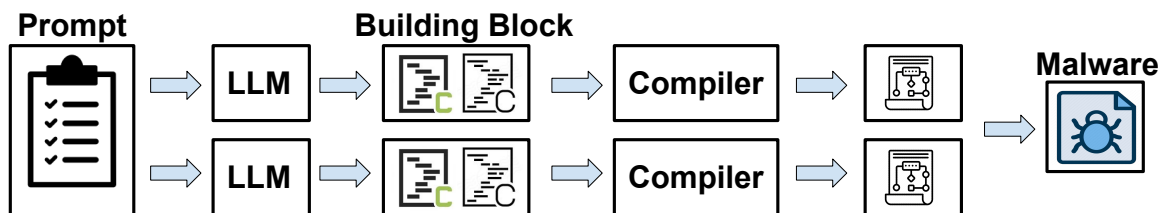


Figure 3: **LLM Malware Generator.** A prompt describing the malware behaviors is used to describe building blocks that are individually generated and then assembled into a binary.

LLM Malware Generator.

Figure 3 illustrates the malware code generation process. An attacker specifies a list of behaviors to be present in the malware sample. The behaviors are mapped to pre-established, specially crafted prompts. The prompts are passed through the LLM to generate code. Multiple requests are made to ensure that multiple code versions are generated. Each generated code piece is compiled into a malware building block. The building blocks are assembled into a single sample using different combinations, yielding malware variants. The most evasive variant is identified. The major limitations of this initial approach are **(i)** the need for manually specifying the prompts and **(ii)** the code generation power of the LLM. In this work, we overcome the first limitation by coupling a GAN to the pipeline to generate the prompt request, and the second by automating the use of a programming-specific code-generation LLM as an API.

2.3 Removing Functions from Static Analysis Sight via Runtime Loading

To understand the idea of making a sample evasive via function removal, it is key to understand how a function can be removed from view in static binary analysis. When no instructions are provided to GitHub Copilot (or any other LLM), it tends to generate code that uses functions from libraries linked at compile time. This causes the function and library names to be included in the PE file header, enabling static detectors to identify them. Code 1 illustrates the real case where, during one of our requests to write code to decode base64 strings, GitHub Copilot produced code for compile-time linking.

```

1 // Write in C
2 // Use the Windows API
3 // Write a function that takes a buffer and its size as arguments and
  // returns the buffer content decoded from base64
4 // The function prototype is int Decode_Base64(void *encoded, int size,
  // void *decoded);
5
6 int Decode_Base64(void *encoded, int size, void *decoded){
7     DWORD dwDecodedSize = 0;
8     BOOL bResult;
9     bResult = CryptStringToBinary(encoded, size, CRYPT_STRING_BASE64, NULL,
  &dwDecodedSize, NULL, NULL);
10    if (!bResult){
11        printf("CryptStringToBinary failed: %d\n", GetLastError());
12        return 0;
13    }
14    bResult = CryptStringToBinary(encoded, size, CRYPT_STRING_BASE64,
  decoded, &dwDecodedSize, NULL, NULL);
15    if (!bResult){
16        printf("CryptStringToBinary failed: %d\n", GetLastError());
17        return 0;
18    }
19    return dwDecodedSize;
20 }

```

Code 1: Original Prompt and Code. `CryptStringToBinaryW` is called directly as it is linked statically.

An alternative implementation of this function would replace compile-time linking with runtime linking, in which libraries are only added to the process's address space on demand. Code 2 exemplifies a real case from our experiments in which GANs requested that GitHub Copilot generate the same function as in Code 1, but now using runtime linking. GAN inserts an additional prompt instruction automatically (last line) based on its identified feature vector. Since it states that the `CryptStringToBinaryW` function should be removed from the PE header, one way to do so without breaking binary functionality is to move the function from compile-time to runtime linking. As a result, the function and library names will not appear in the PE file header. In this case, the `LoadLibrary` (line 9) and `GetProcAddress` (line 12) functions are used to dynamically retrieve the library and function handles, which are then used normally (line 18).

```

1 // Write in C
2 // Use the Windows API
3 // Write a function that takes a buffer and its size as arguments and
4 // returns the buffer content decoded from base64
5 // The function prototype is int Decode_Base64(void *encoded, int size,
6 // void *decoded);
7 // Load the CryptStringToBinaryW function from the crypt32.dll library
8 // dynamically
9
10 int Decode_Base64(void *encoded, int size, void *decoded){
11     int decoded_size = 0;
12     HMODULE hModule = LoadLibrary("crypt32.dll");
13     if (hModule != NULL){
14         typedef BOOL(WINAPI *CryptStringToBinaryW_t)(LPCWSTR pszString, DWORD
15             cchString, DWORD dwFlags, BYTE *pbBinary, DWORD *pcbBinary, DWORD
16             *pdwSkip, DWORD *pdwFlags);
17         CryptStringToBinaryW_t CryptStringToBinaryW = (CryptStringToBinaryW_t
18             )GetProcAddress(hModule, "CryptStringToBinaryW");
19         if (CryptStringToBinaryW != NULL){
20             WCHAR *encoded_w = malloc((size + 1) * sizeof(WCHAR));
21             MultiByteToWideChar(CP_UTF8, 0, encoded, size, encoded_w, size);
22             encoded_w[size] = 0;
23             DWORD dwFlags = CRYPT_STRING_BASE64;
24             CryptStringToBinaryW(encoded_w, size, dwFlags, NULL, &decoded_size,
25                 NULL, NULL);
26             if (decoded_size > 0){
27                 BYTE *decoded_b = malloc(decoded_size);
28                 if (decoded_b != NULL){
29                     if (CryptStringToBinaryW(encoded_w, size, dwFlags, decoded_b, &
30                         decoded_size, NULL, NULL)){
31                         WideCharToMultiByte(CP_UTF8, 0, (WCHAR *)decoded_b,
32                             decoded_size / sizeof(WCHAR), decoded, decoded_size, NULL,
33                             NULL);
34                     }
35                     ...
36                 }
37                 return decoded_size;
38             }
39         }
40     }
41 }

```

Code 2: Modified Prompt and Code. CryptStringToBinaryW must be resolved before it is called, as it is linked dynamically.

3 Related Works

In this section, we discuss related work on adversarial machine learning, GAN-based malware generation, and LLM-assisted malware generation to better position the contributions of this paper.

Adversarial Machine Learning. The interaction between attackers and defenders has historically evolved as an arms race [14]. As malware detection progressively adopted machine learning techniques, this conflict naturally extended into the adversarial machine learning domain [29], where attackers started crafting adversarial examples (AEs) to evade ML-based detectors.

Over time, multiple AE generation strategies have been proposed in the literature. Existing approaches include embedding malicious payloads into otherwise benign executables [17], perturbing executable binaries through gradient-guided optimization [20], and patching binary instructions to alter detector-visible characteristics [51]. Each of these strategies presents advantages and limitations, but most rely on manually engineered transformations and domain expertise, limiting scalability and automation.

The current state of the art in automated AE generation for malware is the use of Generative Adversarial Networks (GANs) [13]. GANs provide a principled mechanism for learning which

feature perturbations increase the probability of evading a detector. In this work, we build upon this automated perturbation capability and investigate whether GAN-guided transformations can be translated into functional evasive malware binaries in a fully automated pipeline.

GANs for malware generation. The applicability of GANs to security problems was recognized early in the development of adversarial learning techniques. One of the most common defensive uses of GANs is the construction of more robust malware detectors [50], including approaches adopted in industrial settings [12]. In these cases, the discriminator is typically used as the final detector, while the generator is primarily employed for adversarial training or data augmentation.

Subsequent work explored offensive applications of GANs, such as adapting network traffic to evade firewalls and intrusion detection systems [44]. In these scenarios, the focus shifts toward the generator, which learns how to produce evasive inputs against third-party detectors.

GAN-based malware generation has been proposed for both desktop [41] and mobile [30] environments. In this work, we focus on desktop malware. The literature presents multiple GAN architectures suitable for adversarial malware generation [23], each with different optimization characteristics and stability properties. We rely on three representative architectures: MalGAN [28], one of the earliest and most widely adopted GANs for malware generation; LSganAT [49]; and LSMalGAN [48], which introduce variations intended to improve training stability and adversarial effectiveness. By evaluating multiple architectures, we seek to distinguish effects associated with the GAN design itself from those associated with adversarial malware generation more generally.

Despite their effectiveness, GAN-based approaches suffer from a major limitation: they typically operate in feature space rather than problem space. In practice, GANs manipulate vector representations of binaries rather than the binaries themselves. As a result, a GAN may identify a feature vector as evasive even when the corresponding modification cannot be realized in a functional executable. For example, zeroing a feature in a vector representation may correspond to removing functionality from a compiled binary, which is generally infeasible without access to source-level control.

This limitation is known as the *problem-space/feature-space gap* [42] and represents one of the main barriers preventing adversarial example generation techniques from being reliably applied to real-world malware binaries. We observe that this gap exists largely because most adversarial learning techniques were originally designed for domains where the underlying source representation is directly modifiable, which is often not the case for malware analysis, where only compiled binaries are available.

Our key observation is that this limitation can be mitigated if the adversarial modifications suggested by the GAN are mapped to source-level constructs under attacker control rather than directly applied to the original binary. This can be achieved by embedding the malicious payload into a wrapper or dropper whose implementation can be synthesized and modified programmatically. In this work, we investigate whether Large Language Models (LLMs) can provide this missing realization capability.

LLMs and security. Since the emergence of Large Language Models, researchers have explored their applicability to a wide range of security tasks. Proposed applications include automated bug fixing [39], reverse engineering assistance [40], and support for software analysis tasks. At the same time, researchers have investigated the security risks associated with these models, including the generation of insecure or unsafe code [38]. One major concern is the possibility that malicious actors may leverage LLMs to automate malware development [19].

LLMs for malware generation. The abuse of LLMs for offensive purposes was envisioned shortly after the introduction of modern generative models [10]. Early work discussed the possibility of using language models for exploit generation and malware development [25]. Initial evaluations, however, suggested that general-purpose LLMs did not substantially simplify malware creation [19].

As models evolved, researchers and attackers progressively discovered methods to improve the viability of LLM-assisted offensive tooling. Many early demonstrations appeared in gray literature [46] and underground communities [2]. More recent academic work has investigated malware generation techniques via bypassing safety guardrails [37] or using intentionally unrestricted models [1].

One of the first academic efforts to systematically evaluate malware generation capabilities in LLMs was the GPT-3 study by Botacin [7]. While pioneering, that work presents two important limitations: **(i)** malware behaviors still required manual specification by the operator; and **(ii)** the generated code quality was constrained by the use of a general-purpose text model rather than a programming-specialized code generator. These limits are overcome in our solution with the use of a specialized (GitHub Copilot) model coupled to an automated solution (GAN-based) for prompt generation.

Subsequent work investigated the generation of malware variants using LLMs [3], which is also our goal. However, prior studies were limited to relatively small-scale evaluations, such as generating (six) hundreds of variants from a small seed set (10 samples). In contrast, our goal is to investigate whether evasive malware generation can be performed at substantially larger scales (10 thousand variants per sample) through a fully automated pipeline.

Bridging feature-space and problem-space generation. Our proposal should not be interpreted as a simple sequential concatenation of a GAN and an LLM. Our contribution emerges from integrating these two complementary capabilities into an iterative malware generation process. In our framework, the GAN operates as an optimization mechanism that discovers evasive feature-level transformations, while the LLM acts as a realization mechanism capable of instantiating those transformations into compilable malware variants under source-level control. The generated samples are then re-evaluated against detectors, allowing the generation pipeline to iteratively refine evasive behaviors and produce increasingly effective malware variants.

This integration is necessary to bridge the long-standing problem-space/feature-space gap in adversarial malware generation [42], which prior GAN-only and LLM-only approaches address only partially. Making this integration practical, however, required overcoming multiple technical challenges beyond simply connecting two existing tools. First, we needed to design a representation capable of mapping GAN-generated feature perturbations into coherent source-level constructs that could be realized by the LLM. We therefore developed a boolean-vector representation encoding behavioral building blocks such as functions, libraries, and execution mechanisms. Second, we needed to ensure semantic consistency between generated features and the produced malware source code, preventing infeasible or contradictory combinations of APIs and behaviors. Third, existing GAN architectures were not originally designed to operate at the scale required by our feature representations, requiring adaptations to support feature vectors one order of magnitude larger than those commonly explored in prior work.

Additionally, fully automating the malware generation pipeline required overcoming several practical engineering barriers. In particular, code generation, compilation, packaging, execution, and detector evaluation needed to be orchestrated automatically despite dependencies on external development environments and GUI-driven tooling. These challenges are critical because the effectiveness of an iterative generation framework depends on the ability to repeatedly generate, compile, execute, and evaluate large numbers of malware variants without manual intervention.

Accordingly, the novelty of this work does not lie in proposing a fundamentally new adversarial learning theory, but rather in demonstrating that a tightly integrated GAN-LLM iterative generation framework can bridge the problem-space/feature-space gap and enable scalable generation of functional evasive malware variants. We further provide a systematic empirical characterization of this process across multiple GAN architectures, detector configurations, binary formats, and deployment settings.

4 Connecting GANs and LLMs

Overview. The typical process for generating adversarial malware is for an attacker to first create a malware sample with the desired malicious behavior, then have a GAN modify it to make it more evasive. We follow the same approach, but replace the manual attacker generator with the LLM-powered generation procedure described by Botacin [7]. Thus, the whole malware generation process is automated.

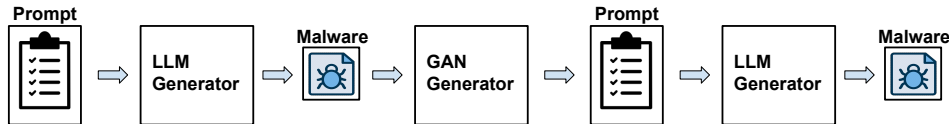


Figure 4: **GAN-LLM malware generation.** A first malware sample is generated following the ordinary LLM-based malware generation procedure. The generated malware then passes through the GAN to determine which functions to keep, add (as dead code), or remove (obfuscate). The resulting feature vector is re-inputted to the LLM to generate the final evasive malware sample. GAN-LLM malware generation.

Figure 4 summarizes the fully automated malware generation process. The pre-defined prompts are used as seeds for the LLMs. LLMs generate code for the Building Blocks (BBs), which are then compiled into malware variants. The compiled malware variants are then fed into the GAN, which modifies each binary to become more evasive. The feature vector produced by the GAN is used as a prompt for the LLM, which outputs code for new BBs, which are then compiled into binaries (see prompt modifications in Sec. 2.3).

Attack rationale and threat model. In this work, we assume that attackers target function-name-based PE detectors [17], i.e., detectors that classify function-name strings present in binary headers, such as those of imported and exported functions. This type of detector/feature is widely adopted in the literature [45, 35] and by AVs [16]. The reliance on function names makes classifiers stronger than byte-based classifiers because it is easier for an attacker to append bytes to the binary body without breaking it than to the header of the binary, which makes this type of classifier robust to adversarial attacks to the point of winning previous malware evasion challenges [17]. The rationale for the attacks here is that GANs may indicate which functions should be present in the binary header to evade detector classification. In the proposed pipeline, the first malware sample generated by the LLM serves as a template for the functions and libraries needed to implement the desired malicious behaviors. The GAN is then allowed to: **(i)** keep the functions; **(ii)** add new functions (as dead code); or **(iii)** remove functions—i.e., moving them to runtime (see Sec. 2.3). We target static detectors, such that a successful evasion should implement the same malware semantics in runtime with a different static binary shape. Thus, “removing” a function means reimplementing the same functionality with **(i)** other functions or **(ii)** a runtime function loading to preserve semantics while displaying a binary without the “red flag” static function import.

The base LLM. We adopted GitHub Copilot as a base LLM as it is powered by a more advanced model (GPT-4) than the previous study (GPT-3). We used GPT-4 in its GitHub Copilot version, as it is tailored for code generation and thus more likely to be used by developers (and attackers).

The targeted GANs. GANs are the basis of our proposal. Therefore, we implemented (3) representative examples of GANs for malware generation to perform our experiments. The literature is full of proposals for GAN architectures, but only a limited number address malware generation. While prior work has examined the pros and cons of GAN architectures for general applications [23], no prior work has examined GAN architectures for malware generation as we propose. Therefore, in this work, we considered MalGAN [28] as the first representative example, as it was the pioneering work in the general field and the most-cited on GAN-based malware

generation. To demonstrate generalization, we also reproduced two versions of MalGAN based on alternative GAN architectures: LSganAT [49] (attention) and LSMalGAN [48] (Least Squares). **Handling functions with GANs.** MalGAN-like GANs operate on binary vectors that indicate whether a given feature is present in the (input or output) binary. We used this characteristic to represent the presence of statically extracted binary strings in the array. To that end, we trained a one-hot encoder that maps each symbol name (either a function name, library name, or exported symbol name) to an array index. The GAN assigning 1 to that array position means that the symbol should be present in the final binary, thus leading to an LLM request to generate code that uses that function.

The Limits of the feature vector. In our modified GANs, the Boolean feature vector is populated with information from the functions statically referenced in the binaries. In practice, binaries may have a large number of imports (e.g., hundreds), making GAN optimization an exponential problem and thus intractable. To avoid this problem, the original GAN implementation [28] proposed limiting the feature vector to the most prevalent functions in the dataset. Here, we repeat this strategy by identifying the most prevalent information to populate the feature vector. Our implementation includes not only the top functions but also the top libraries and the top exported symbols. Unlike the original approach [28] that used a fixed feature vector size ($K=150$), we here identify the maximum viable vector size and characterize its impact on the GAN operation (Sec. 5). Our final feature size ($K=1500$) is sufficient to support all function imports and exports of any native Windows binary and/or library.

Keeping the feature vector consistent. A challenge for building the set of the top- K most popular features is that it encompasses multiple types of interdependent data. When a function is identified as prevalent, the library that hosts it should also be included in the binary to ensure it remains functional. Thus, after identifying the prevalence, we ensure feature-vector consistency by adding the functions' hosting libraries to the feature vector. The identification process is performed automatically by parsing the function exports of each DLL included in the binary (via PEfile [24]). Thus, our feature vector of the most prevalent feature is composed of a mix of top functions, libraries, and exports (DLL-only).

Handling PEs and DLLs. The proposed GAN implementation processes the symbols in PE files, including imported and exported functions. Although the PE file format supports simultaneous imports and exports, in practice, different applications use it in different ways. Typical applications (EXE) tend to import more functions/libraries and not export any symbol, whereas libraries (DLL) tend to export functions. Allowing the GAN to add exports to typical EXEs would likely increase the likelihood of the binary being detected. To avoid this case, we added information on whether the PE file is an EXE or a DLL to our solution. If the parsed file is a DLL, the GAN is allowed to produce function exports as output. In practice, this solution is deployed by training two different GANs, one for DLLs and one for EXEs.

From GANs to LLM Prompts. GANs' outputs are boolean feature vectors that must be translated into English to become LLM prompts. This is done deterministically by a slot-filling script that converts the presence or absence of a function (0 or 1 in the feature array) into a sentence. For instance, we represent the removal of a function by a slot-filling prompt in the format of `// Load the <function name> function from the <library name> library dynamically`, as in the example shown in Code 2 (line 5) from Section 2. The function and library names are identified by their index (fixed) in the feature array.

Adding Functions. When GANs indicate that an API function should be added to a binary, we can add it as a dead code without affecting the binary functionality. Thus, we traverse the GAN's output and ask GitHub Copilot to generate a code snippet that uses each specified function. All the generated snippets are added at the end of the malware code via a never-taken branch-implemented via an implicit (e.g., opaque constant [34]) false condition.

Removing Functions. When GANs indicate that a function should be removed, we cannot simply ask GitHub Copilot not to generate them because it would break the malware function-

ality. The key insight is that the function should not be removed, but only hidden from static analysis. In such cases, we instruct GitHub Copilot to generate the same code by dynamically loading the function [32]. Note that we do not generate a new library in this process. Still, we apply it to the import of the existing native Windows libraries referenced by the malware samples (an assumption of prior work—Sec. 2.2).

Automating LLM code generation. After the GAN generates the new prompt, the final step in malware generation is to ask the LLM to code it. The GPT-3 model used in previous work has its own API [36], which facilitates automation. As the GitHub Copilot model used in this work does not present an API, we created one ourselves. Access to GitHub Copilot is provided via a Visual Studio Code (VSCode) extension [33]. Users should type prompts in VSCode, and the LLM’s response is displayed directly in the VSCode canvas. To automate the prompt insertion and code collection, we created an AutoIT [6] script that takes the prompt from the GAN and pastes it into VSCode. Our script mimics pressing the Copilot hotkeys to trigger its operation, collects data from the canvas, and writes it to a target file. We can specify a number of pressed keys in the script. Each time the script presses a key, a different Copilot response is captured, allowing us to create variants of the same prompts. With this script, we can provide prompts and retrieve generated code as if the GitHub Copilot provided an API. Once inside the VSCode environment, we also added the ability to automatically compile the generated code into the script. We deployed the script inside a Virtual Machine (VM), thereby creating a malware-as-a-service environment in which the GANs send prompts to the VM and retrieve compiled binaries from it.

4.1 Operational Example

To better understand the operational flow of the proposed malware generation pipeline, consider the following (real) example of a ransomware generation request:

1. **Initial Malware Specification and Generation.** The operator provides a high-level prompt describing the intended behavior: *“generate ransomware-like software capable of enumerating and encrypting files.”* The LLM-based generation framework translates the prompt into a set of behavioral building blocks following Botacin’s automated malware composition approach. The generated implementation includes:

- Filesystem enumeration through `FindFirstFileW()` and `FindNextFileW()`,
- File access primitives such as `CreateFileW()`, `ReadFile()`, and `WriteFile()`,
- Cryptographic operations using `CryptAcquireContextW()`, `CryptGenKey()`, and `CryptEncrypt()`,
- Directory traversal using `SetCurrentDirectoryW()`,
- Execution orchestration using `CreateThread()` and `Sleep()`.

The generated source code is compiled into an executable binary, producing the tuple: $(\text{source}_0, \text{binary}_0)$

2. **GAN Refinement Round 1: Initial Binary Analysis.** The initial binary binary_0 is submitted to the GAN-based optimization framework. The generator extracts the feature representation used by the target malware detector and evaluates it against the discriminator. The discriminator classifies the sample as malicious based on ransomware-associated API usage patterns, particularly the co-occurrence of `FindFirstFileW()`, `CryptEncrypt()`, and `WriteFile()` operations. The GAN then proposes feature-space perturbations intended to move the sample toward the benign region of the classifier space. Example modifications include:

- Replacing `FindFirstFileW()` and `FindNextFileW()` with `NtQueryDirectoryFile()`,

- Introducing benign file-management APIs such as `GetFileAttributesExW()`,
- Adding synchronization-oriented functionality through `CopyFileExW()`,
- Substituting standard I/O patterns with `CreateFileTransactedW()`,
- Introducing registry-access routines such as `RegOpenKeyExW()` commonly observed in legitimate management software.

The framework computes the feature delta between the original and optimized feature vectors, producing: $\Delta_1^+ = \{\text{functions to add}\}$ and $\Delta_1^- = \{\text{functions to remove}\}$.

3. **LLM Refinement Round 1: Source-Level Regeneration.** The tuple $(\text{source}_0, \Delta_1^+, \Delta_1^-)$ is provided to the LLM. The LLM regenerates the source code while preserving the original semantics. The transformations include:

- Replacing `FindFirstFileW()` with `NtQueryDirectoryFile()`,
- Inserting auxiliary calls to `CopyFileExW()` and `GetDiskFreeSpaceExW()`,
- Restructuring execution flow through worker threads created with `CreateThread()`,
- Modifying static imports associated with `Advapi32.dll` and `Crypt32.dll`.

The regenerated source is compiled, producing: $(\text{source}_1, \text{binary}_1)$.

4. **GAN Refinement Round 2: Secondary Adversarial Optimization.** The updated binary binary_1 is re-evaluated by the GAN discriminator. Although the malicious score is reduced, the discriminator still identifies ransomware-related characteristics. In this iteration, the GAN determines that statically imported cryptographic APIs such as `CryptEncrypt` and `CryptGenKey` remain highly discriminative malicious features. The optimizer, therefore, proposes the additional modifications:

- Removing direct imports of `CryptEncrypt()`,
- Dynamically resolving cryptographic routines through `LoadLibraryW()` and `GetProcAddress()`,
- Delaying cryptographic API invocation until runtime,
- Reducing suspicious import-table artifacts associated with `Advapi32.dll`,
- Replacing direct entropy-generation calls with wrapper functions.

The resulting feature-space modifications Δ_2^+ and Δ_2^- are produced.

5. **LLM Refinement Round 2: Behavioral Refactoring.** The tuple $(\text{source}_1, \Delta_2^+, \Delta_2^-)$ is passed back to the LLM. The model regenerates the implementation to incorporate the requested feature modifications while preserving the intended ransomware behavior. In particular, cryptographic APIs are no longer statically imported and are instead dynamically resolved during execution using `LoadLibraryW(L"Advapi32.dll");` and `GetProcAddress(..., "CryptEncrypt");` The resulting implementation is compiled, forming the tuple: $(\text{source}_2, \text{binary}_2)$.

6. **Convergence and Evasion Success.** The optimized binary binary_2 is submitted once more to the discriminator. At this stage, the discriminator classifies the sample as benign-like due to the reduced presence of strongly discriminative malware features and the increased similarity to legitimate backup or synchronization software. The iterative optimization process terminates. The final output is semantically equivalent to the initial request, but in an evasive form.

5 Evaluation

The following Research Questions (RQs) evaluate three dimensions of the malware generation process, providing a complete treatment of the topic. **First**, we characterize the transition from GPT-3 to GitHub Copilot. Our goal is to demonstrate that migrating to bigger models produces superior code generation results, but it is not enough alone to produce naturally evasive code, which justified the adoption of our proposed GAN-LLM generator; **Second**, we prove previous GAN approaches as promising approaches for identifying evasive APIs, but we also prove them to break binary functionality if applied alone in a naive strategy, which justifies once again the adoption of our proposed GAN-LLM approach to “fix” the GAN result into a realizable malware sample; and **Third**, we show how integrating GANs to LLMs solves the two above identified problems, being this the evaluation of our proposal itself.

5.1 RQ1. Is GitHub Copilot’s malware generation superior to that of GPT-3?

Since a key difference between this work and previous work [7] is the use of a different baseline (GitHub Copilot vs. GPT-3), it is important to evaluate its raw performance (i.e., without GAN support) to establish a new ground truth. To that, we repeated the malware generation experiment presented in previous work [7]. The test was performed on the same ground, i.e., we provided GitHub Copilot with the same prompts as in [7] for the same 10 original building blocks and characterized the code-generation accuracy and variety. Note that we are not re-running the GPT-3 experiments, but re-running the same reference prompts in GitHub Copilot and comparing the results with the described GPT-3 baseline. Note also that the goal of this experiment is to quantify the evasion improvements obtained by shifting to a more powerful LLM vs. the further addition of an LLM orchestrator on top of it.

RQ1.1 How many different building blocks can GitHub Copilot generate? The first aspect to evaluate when considering the use of an LLM for malware generation is whether it actually generates the requested code. Then, it is key to evaluate whether the generated code is diverse, since the main hypothesis behind LLMs for malware generation is that they can generate different implementations of the same building blocks, and such variety might result in evasive constructions. We evaluate GitHub Copilot’s malware code generation by repeatedly asking it to generate code for known malware prompts and measuring the diversity of the generated code.

There are various metrics for measuring the complexity of code snippets. We here consider two of them; (1) unique functions; and (2) unique paths. First, we consider the number of unique functions among the building blocks a good indicator of diversity, since the GAN detectors used consider these functions in their feature vectors. For the second, we consider that adding information about the order in which the function appears in the code helps characterize the diversity of the generated code.

Figure 5 shows the number of unique building blocks’ code excerpts generated by GitHub Copilot for each building block type for a growing number of requests. We stopped querying GitHub Copilot after 50 requests per building-block prompt, as we empirically observed that 10^{50} combinations are on the boundary of computational intractability. A general characteristic of the generated building-block excerpts is that they exhibit positive variation: the more one asks, the more diverse the outputs become. We qualitatively characterize the generated code excerpts in Appendix A. However, growth is not linear across all building block types; it depends on their complexity. More complex blocks (e.g., DLL Injection, AutoRun, and so on) require more functions to be implemented and thus offer more room for different implementations. Simpler blocks that can be solved with a few functions (e.g., deleting a single file) have little to no room for variation. In this case, they repeatedly present the same implementation, thereby contributing the same building blocks. For this reason, the actual number of possible variants is not 10^{50} , but it is limited by the actual number of building blocks that one can generate for a

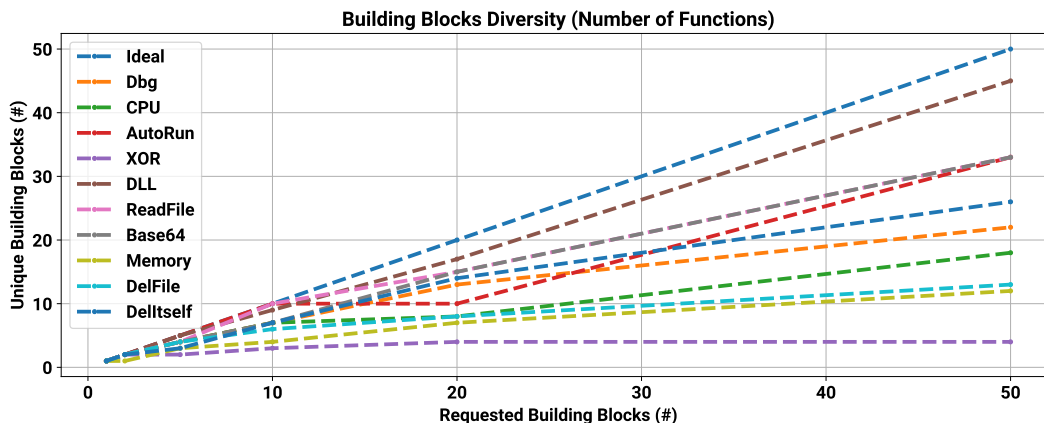


Figure 5: **GitHub Copilot’s Malware Building Blocks Generation.** Number of generated unique building blocks (functions) vs. number of requested building blocks.
 GitHub Copilot’s Malware Building Blocks Generation.

given number of attempts. In no scenario did GitHub Copilot produce as many building blocks as requests (the Ideal line). such that the ideal line can be considered an upper bound for the number of BBs.

To confirm the upper bound for the building block generation process, we evaluated the same building blocks using the second metric. Therefore, we considered the unique paths generated in the building blocks. For instance, whereas we previously considered two building blocks that had only the `IsDebuggerPresent` function as unique, regardless of their control flow, we now consider them distinct if one uses an IF-ELSE structure and the other doesn’t. This type of variation cannot defeat our function-name-based GAN detector, but it might defeat call-graph-based detectors and thus should be considered for transferability (e.g., from GANs to real AVs).

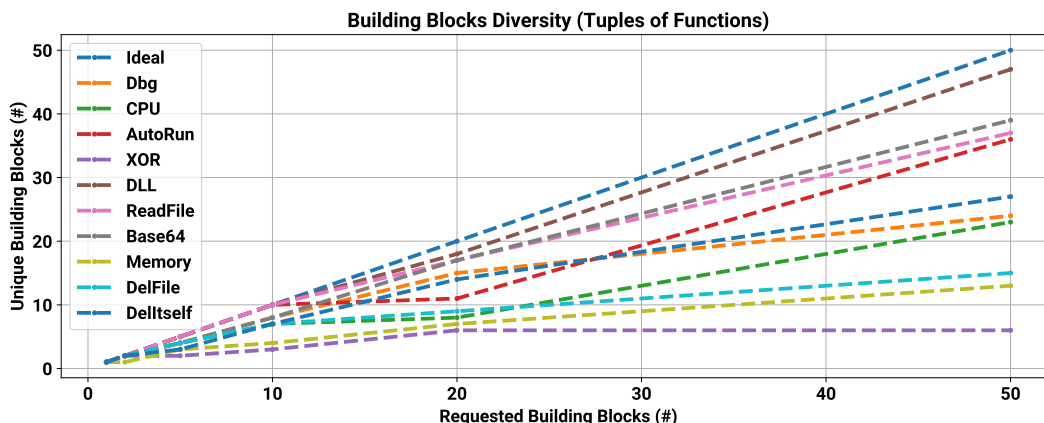


Figure 6: **GitHub Copilot’s Malware Building Blocks Generation.** Number of generated unique building blocks (Paths) vs. number of requested building blocks.
 GitHub Copilot’s Malware Building Blocks Generation.

Figure 6 shows the number of unique building blocks according to their paths for the different numbers of requests. Once again, positive growth is observed, but it is limited by the complexity of the building blocks. The number of unique constructions is greater when we consider differences in paths than in function names. However, in this case, once again, at no moment was the number of generated unique building blocks equal to the number of different requests. Therefore, we conclude that GitHub Copilot can generate multiple distinct malware building blocks, but

fewer than the number of requests. This result is also observed in the original GPT-3 study, which shows that the sub-linear growth of the building blocks is a characteristic of the malware generation problem, not only of the model.

Verdict: *The more requests one makes, the more unique building blocks GitHub Copilot produces. The code is more diverse as the building blocks become more complex. However, GitHub Copilot always produces fewer unique building blocks than the number of attempts. This is in line with the GPT-3’s results.*

RQ1.2 How many compilable malware samples can GitHub Copilot generate? The previous results demonstrate GitHub Copilot’s maximum potential for generating malware building blocks. However, in addition to the presented code-generation challenges that limit variability, the compilation process required to generate the malware files introduces further limitations. A code excerpt might be theoretically unique but impractical for compilation and deployment, as it might rely on wrong API flags, incompatible library versions, and so on. Therefore, it is key to evaluate GitHub Copilot’s feasibility for producing compilable, functional binaries.

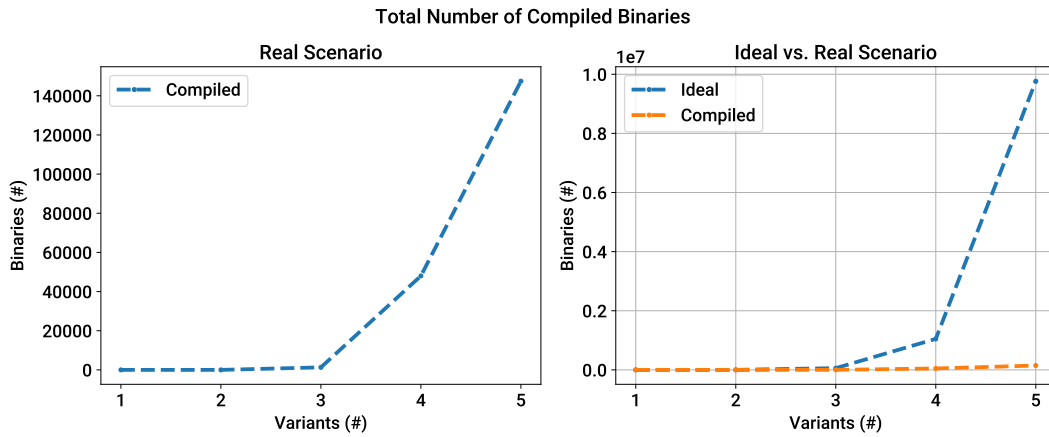


Figure 7: **Total Number of Compiled Binaries.** (a) Absolute number of compiled binaries and (b) the comparison with the expected number for a perfect (ideal) solution.

Total Number of Compiled Binaries.

To test how compilable are the malware generated by GitHub Copilot, we asked it to generate an increasing number of variants for each building block. Figure 7 (a) shows the number of compilable binaries vs. the number of requested variants. The number of generated binaries remains at 0 until 3 variations per building block are requested, because at least one building block failed to compile in all previous attempts. Afterward, we observe exponential growth in the number of generated binaries, as each newly compiled building block combines with all versions of the other building blocks. Despite exponential growth, it occurs on a smaller scale than the ideal number of binaries that could be generated. Figure 7 (b) shows the actual number of compiled binaries vs. the number of binaries expected in the ideal case, i.e., if all building blocks are compiled successfully. This result highlights that the absolute number of generated malware variants is substantial, whereas the relative number indicates that LLMs still have substantial room for improvement in generating correct code.

Although the previous experiment characterizes the overall malware compilation process, individual building blocks exhibit distinct characteristics. Figure 8 shows the fraction of successfully compiled building blocks for our multiple, incremental attempts. Initially, the success rate is 0 for many building blocks, meaning the first code GitHub Copilot suggests is not compilable in most cases. The success rate increases with the number of building-block variants requested, indicating that at least some are functional. Only the simplest building block remained consistently compilable throughout the experiment. In general, the number of compilable building

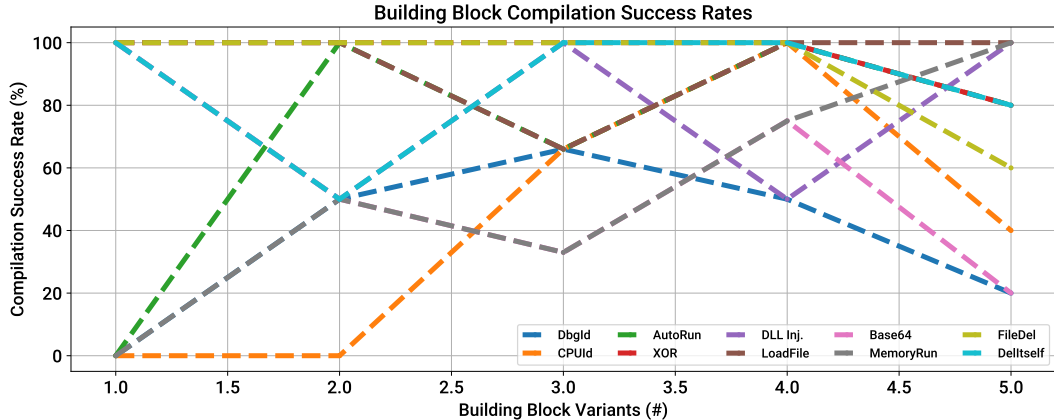


Figure 8: **Building Block Successful Compilation Rate.** Individual rates for each building block vs. the number of requested building blocks.

Building Block Successful Compilation Rate.

blocks increases within a certain limit as more variants are requested, and then decreases if a larger number of variants is requested, as the generated code tends to be more random/diverse and thus more error-prone. In only two cases did the fraction of correct samples increase with the number of requests, due to the randomness inherent in querying the model multiple times.

After characterizing GitHub Copilot’s code-generation capabilities, it is important to determine whether GitHub Copilot produces more compilable code than GPT-3. When the previous study asked GPT-3 for 10 variants of each building block, only 10K (10^4) compiled pieces of code were produced out of the 10^{10} ideal possibilities. As a comparison, Figure 7 (a) shows that this same number of compiled binaries is achieved between 3 and 4 variant requests per building block. With 5 variants per building block, more than 10 times more binaries are compiled (120K). Therefore, we conclude that GitHub Copilot can generate more compilable malware building blocks than GPT-3. To provide a fair ground truth for comparison, we did not compare GPT-3 (10 variants) vs. GitHub Copilot (10 variants); instead, we limited the GitHub Copilot comparison to the identified ranges to normalize results based on the total number of produced binaries. Table 1 compares GPT-3 and GitHub Copilot via the metrics proposed in the previous work.

Table 1: **Comparing GitHub Copilot and GPT-3.** Top-10 malware building blocks generation statistics.

Combinations: Metric: Behavior	5 Requests				10 Requests				First Binary		Time (s)	
	GPT-3	GitHub Copilot	GPT-3	GitHub Copilot	GPT-3	GitHub Copilot	GPT-3	GitHub Copilot	GPT-3	GitHub Copilot	GPT-3	GitHub Copilot
String XORing	88%	80%	70%	100%	88%	80%	70%	100%	-/4	1/1	2.49	44s/9s
Debugger Identification	84%	80%	10%	75%	84%	20%	10%	63%	-/2	2/2	2.63	44s/9s
Remove File	95%	90%	90%	100%	95%	60%	90%	92%	-/2	1/1	2.17	44s/9s
Payload Loading	91%	100%	40%	50%	91%	100%	40%	23%	-/2	1/2	3.21	44s/9s
CPUID check	83%	80%	30%	90%	83%	40%	30%	51%	-/2	3/3	3.45	44s/9s
Delete Itself	94%	93%	40%	80%	94%	80%	40%	76%	-/3	1/1	2.36	44s/9s
Memory Run	60%	80%	20%	51%	60%	100%	20%	51%	-/2	2/2	2.11	44s/9s
AutoRun	99%	100%	20%	60%	99%	80%	20%	17%	-/5	2/3	2.41	44s/9s
Base64	60%	80%	10%	60%	60%	20%	10%	14%	-/3	1/2	3.31	44s/9s
DLL Injection	60%	100%	30%	55%	60%	100%	30%	4%	-/2	1/5	3.41	44s/9s
Compilable Baseline	<10K		140K		10K		>140K					
Functional Baseline	<4.8K		5.6K		4.8K		>5.6K					

1. First Binary means the generation round in which the first functional BB was generated out of the first generation round in which a set of all functional BBs was generated together for the same time, i.e., when the first complete sample was produced.

2. Effective Compilable and Functional Baselines refer to the fact that the presented rates (%) reflect different baselines, as one model generates substantially more functional samples than the other when considering the combination of BBs.

Table 1 shows that GitHub Copilot code is, in general, relatively less compilable than GPT-3 (success rate decreased for 7 of 10 building blocks). However, in absolute terms, this is offset by

GitHub Copilot generating greater variation in the building blocks. In contrast, GPT-3 always compiled the same limited number of building blocks. Notably, GitHub Copilot surpassed GPT-3 on the most complex building blocks. Surprisingly, the code for DLL injection was generated and compiled in 100% of the cases for GitHub Copilot but not for GPT-3. Our manual investigation revealed that, in cases where compilation success rates decreased (e.g., Base64), this was because GitHub Copilot generates more variations than GPT-3, given the limited number of variations generated by GPT-3. Therefore, GitHub Copilot was more likely to generate incorrect code when attempting to diversify it. This allows us to infer that GitHub Copilot is more creative (in the sense of generating diverse code) than GPT-3. Regarding malware generation, we conclude that GitHub Copilot was more effective (producing more binaries) than GPT-3, at the cost of individual BB efficiency (a higher compilation rate but fewer blocks for GPT-3).

Verdict: *GitHub Copilot produces an absolute number of compilable building blocks much higher than GPT-3. 5 variations of GitHub Copilot-generated malware correspond to 10 variations of GPT-3-generated malware. The relative rate of compiled GPT-3-generated malware is higher, but GitHub Copilot outperforms GPT-3 in the long tail. Moving from GPT-3 to GitHub Copilot trades compilation efficiency for variant-generation effectiveness.*

RQ1.3 How many functional malware samples can GitHub Copilot generate? Beyond successfully compiling the generated code, LLM-based malware generation is successful only if the resulting sample is functional. We consider functional samples that (1) do not crash and (2) execute the designed building-block actions. We evaluated all compiled binaries from the previous experiments via sandbox executions to evaluate if they exhibit the expected IOCs.

Table 1 shows that GitHub Copilot produces more functional code than GPT-3 in relative terms. The rate of functional building-block implementation is higher for 7 of 10 building blocks in GitHub Copilot. Among the reduced rates, the most significant case is that of DLL injection. Whereas GitHub Copilot produces 100% compilable code with high implementation diversity, we observe that most of the generated code fails to achieve the goal, indicating that increasing code diversity does not necessarily improve correctness. Despite the challenges of implementing effective DLL injection, at least one version of each building block was successfully generated by GitHub Copilot, indicating that, in the long tail, both GitHub Copilot and GPT-3 can automatically produce fully functional malware code. Beyond relative numbers, GitHub Copilot also produces more functional variants in absolute terms because it generates more code. In previous work, when GPT-3 was asked to generate malware from 10 building-block variants, it produced 4820 functional binaries. In the current work, the request to generate malware based on 5 building-block variants yielded 5600 functional binaries.

It is important to note that the results presented in Table 1 represent the average functionality across all building blocks, whereas individual building-block variants may be more or less functional. When a variant is not functional, it affects all compiled binaries embedding that code. Figure 9 shows the rate of functional samples broken down by the variant embedded in the final malware binary. Although the overall functionality rate of the DLL injection building block is 4%, one of its implementations achieved 20% functionality.

This result shows that an attacker may need to wait to generate malware variants until obtaining a fully functional one. Table 1 indicates, for GitHub Copilot, when the first compilable and when the first functional code was generated. In the previous study, only information on the first functional binary was available. We observe that, in agreement with previous work, the first complete malware generated is not functional. However, unlike previous work, which also did not generate functional building blocks on the first attempt, GitHub Copilot now generated 3 compilable, functional building blocks as its first output. In 8 out of 10 building blocks, the first functional one was produced by GitHub Copilot before GPT-3. The overall performance

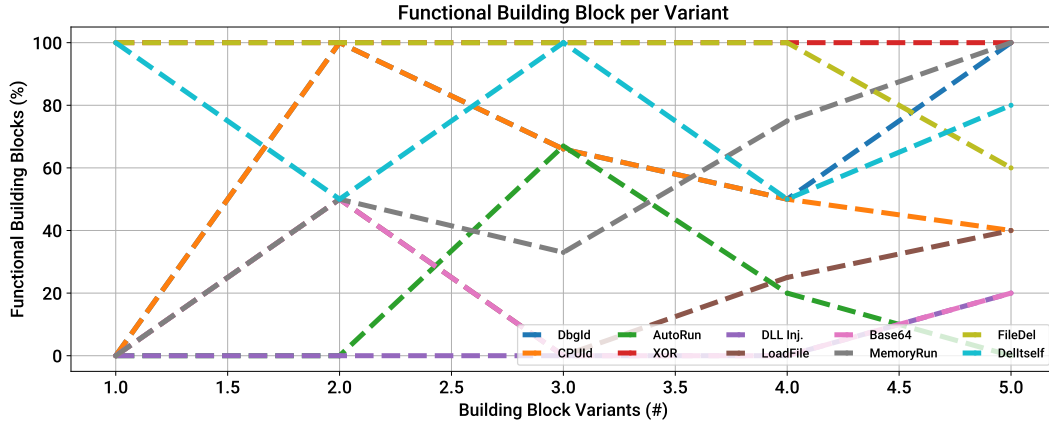


Figure 9: **Functional Building Blocks.** The rate of functional binaries changes with the building block’s variants.

Functional Building Blocks.

is not superior because the first functional DLL-injection building block appears late, causing a general delay (a compilation barrier).

Verdict: *GitHub Copilot produces a higher absolute number of functional binaries than GPT-3. The functionality rate depends on the building block variant. GitHub Copilot consistently struggles to generate complex DLL-injection code. Even though GitHub Copilot once again outperforms GPT-3 in the long tail.*

RQ1.4 How long does it take for GitHub Copilot to generate the malware samples? The last dimension along which GitHub Copilot code generation can be evaluated is the time required to generate the building blocks. Timing is important because it determines how quickly attackers can modify their distributed payloads (e.g., server-side polymorphism). A significant difference between GitHub Copilot and GPT-3 is that whereas GPT-3 presented a well-defined API to be invoked on-demand, GitHub Copilot required us to create the API abstraction by automating the VSCode application, which is expensive due to the need to simulate multiple user clicks. Thus, starting the query process becomes expensive, although generating additional queries is fast (cold-start problem).

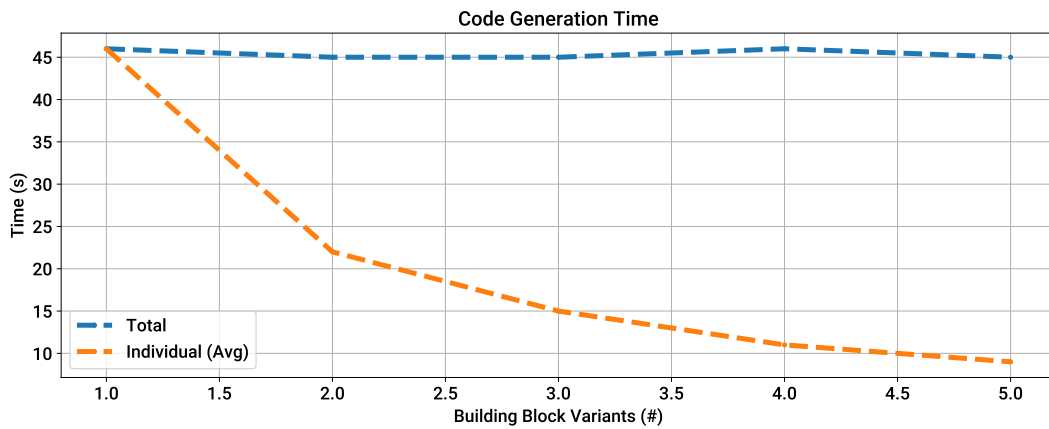


Figure 10: **Code Generation Time.** Total time for generating all and single building blocks. Code Generation Time.

Figure 10 clearly illustrates this phenomenon by displaying the time taken to generate the

building blocks in the previous experiments. We observe that the total time to generate the building blocks is relatively constant, indicating that the process bootstrap accounts for most of the overhead. The high initial cost is diluted as more variants are generated, since generating additional variants is easy (one additional key is sent per request to trigger GitHub Copilot’s autocomplete, compared to the total number of clicks and keys required to initially activate GitHub Copilot). The best cost-benefit is achieved when more variants are generated per building block request.

Table 1 shows the average time taken for GPT-3 and GitHub Copilot to generate each building block in the comparable range. Even when the cost of generating variants via GitHub Copilot is diluted, the actual time taken is at least 3 times higher (from $\approx 3s$ to $\approx 9s$). This result reveals a trade-off between GPT-3’s speed and GitHub Copilot’s ability to generate increasingly diverse building blocks.

Verdict: *Due to the lack of an API, querying GitHub Copilot via GUI automation is a magnitude order more expensive than querying GPT-3 via API. However, the cost is concentrated in the bootstrapping of the automation. Thus, the total cost is diluted as more variants are generated.*

RQ1 Summary. The above results demonstrate (and quantify) that GitHub Copilot is superior to GPT-3 in many dimensions and is therefore suitable as the new baseline for malware generation. The variations observed across building blocks provide additional support for our hypothesis that GAN-based control over which functions are included in binaries might serve not only as an evasion-guidance mechanism but also as a means to increase binary production by guiding GitHub Copilot to produce code using the constructions it performs best.

5.2 RQ2. Do GAN-based approaches produce viable binaries?

As for the transition to GitHub Copilot, adopting a GAN-based malware generation approach also necessitates establishing a new ground truth for the experiments. Our goal is to evaluate the inherent potential of GANs to generate evasive malware and subsequently assess the extent to which LLMs contributed to increasing it. To that end, we evaluate GANs independently and select the best configuration for further experiments. While previous GAN works made significant contributions to the field, they did not characterize the GAN operation with respect to the parameters that affect it. Thus, before answering our key question, we stepped back to present a characterization of GAN operation.

Datasets and Training. To perform the characterization experiments, we considered a dataset composed of (i) 55594 malicious EXE files and 19307 benign EXE files; and (ii) 15964 malicious DLLs and 15964 benign DLLs. This dataset was collected by a partner CSIRT from infected machines between 2012 and 2020. The dataset was characterized in a previous study [8]. We relied on this dataset both to train the GAN’s substitute detector and to test its evasion capabilities.

Testing Methodology. We present results for different GAN detectors to compare their performance. All GANs were trained using the same malware and goodware dataset. We trained and tested EXEs and DLLs separately. To evaluate the impact of the dataset size on the performance, we conducted our evaluation using different splits of the main datasets. The EXE dataset was split into three subsets (5K, 10K, and 19K samples, corresponding to 100% of the dataset). The DLL one was split into two subsets (1K and 3K samples). The split criteria were the availability of goodware samples for a balanced (50-50) training. All samples not included in a given training set were used as testing samples for that same split (tests are the average of 5 runs).

RQ2.1 What is the maximum potential for GAN-based malware generation? In our first experiment, we evaluate the ideal performance of a GAN for evasive malware generation. The tested scenario is considered ideal in two dimensions: **First**, because we use a surrogate model

equal to the target model, thereby ensuring attack transference. **Second**, because we allow the GANs to both add and remove binary features without worrying about whether it is viable in the problem space or not. The experiment aims to demonstrate the potential achievable when GANs are coupled with a solution that bridges the feature-problem space gap.

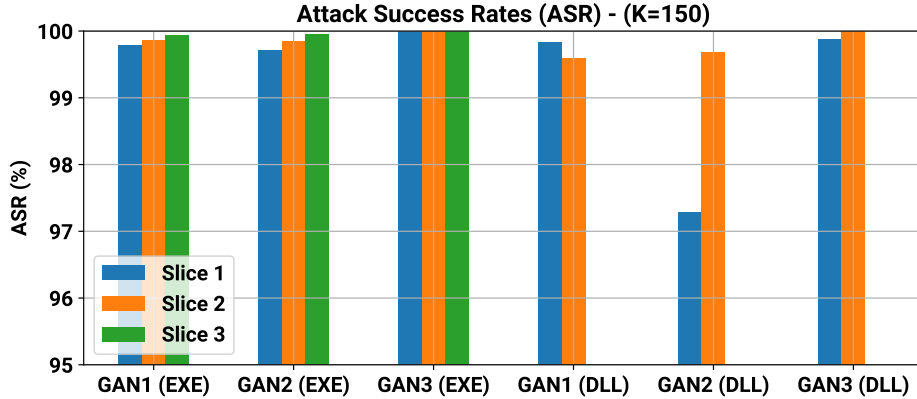


Figure 11: **Attack Success Rates.** Results for multiple GANs and datasets (K=150).
Attack Success Rates.

Figure 11 shows the Attack Success Rates (ASR) for all tested GANs, dataset splits (Slices), and file types for the case where the top (K) 150 features (the same value used in the previous works) are considered. Overall, GANs were shown to be highly effective evasion-generation mechanisms. Despite minor individual variations across scenarios, the GANs consistently achieved ASRs exceeding 97% in all cases. These results highlight GAN’s significant **theoretical** potential for generating evasive malware. This result in our replication study is relevant not only because it validates our GAN implementation, but also because it constitutes the largest GAN malware experiment (largest dataset) to date.

Verdict: *GANs can achieve evasion rates superior to 95% in all scenarios when allowed to freely add and remove binary features.*

RQ2.2 How are GAN-based attacks affected by the size of the training datasets? Breaking down the tests into different dataset slices allows us to evaluate how GAN’s performance is affected by the amount and diversity of data in the training and test sets. A natural hypothesis is that the evasion rate could decrease the more (and more diverse) data a scenario has, as it might make the problem more complicated. The results presented in Figure 11 show that no significant impact could be observed at this test scale. For the EXE dataset, the evasion rate increased slightly with more data across all GANs. For the DLL dataset, the evasion rate increased for 2 out of the 3 GANs. This result shows that GANs exploit the inherent weaknesses of classification models, regardless of the data they are trained on. It guarantees that we can scale up our experiments to the real-world.

Verdict: *GANs achieve high evasion rates regardless of the dataset size, which shows that an intrinsic property of the classification problem is being exploited.*

RQ2.3 How are GAN-based attacks affected by the different file types? Breaking down the tests by file type allows us to evaluate how GAN’s performance is affected by feature types (imports and exports). A natural hypothesis is that the evasion rate for DLLs could decrease following the addition of function exports, as this might make the problem more complex. The results presented in Figure 11 show that GANs are also effective in generating evasive DLLs. In most cases, the evasion rate for DLLs is similar to that for EXEs. Although performance decreases

for the smallest slice in the second GAN, its evasion rate remains above 97%. This result shows that GANs exploit the inherent weaknesses of classification models, regardless of feature type (im/ex)ports.

Verdict: *GANs achieve high evasion rates regardless of the file type, thus being able to operate both with function imports and exports.*

RQ2.4 How are GAN-based attacks affected by the different GAN architectures? Breaking down the tests by GAN implementation allows us to evaluate how GAN performance is affected by architectural choices (e.g., the impact of least squares vs. other minimization functions). A natural hypothesis is that some functions are better fitted to evasion than others. The results presented in Figure 11 show that the third GAN (MalGAN) slightly outperforms the other ones (LSGAN and MalLSGAN) for both EXEs and DLL. This GAN achieved a 100% evasion rate for EXEs. Despite this minor advantage, all GANs achieved evasion rates exceeding 97% in all cases. This result shows that GANs exploit the inherent weaknesses of classification models that rely less on the internal fit function.

Verdict: *GANs' architectures have a noticeable but minor impact on increasing the evasion rates in comparison to the base evasion phenomenon.*

RQ2.5 How much do GAN-based attacks affect the binaries? Once we evaluate the ideal potential of GANs as malware generators, it is key to evaluate their actual feasibility in the problem space—i.e., to evaluate if the proposed modifications are feasible. To that, we here characterize what are the changes the GANs proposed for the binaries from the previous experiment in comparison to their original, non-evasive versions.

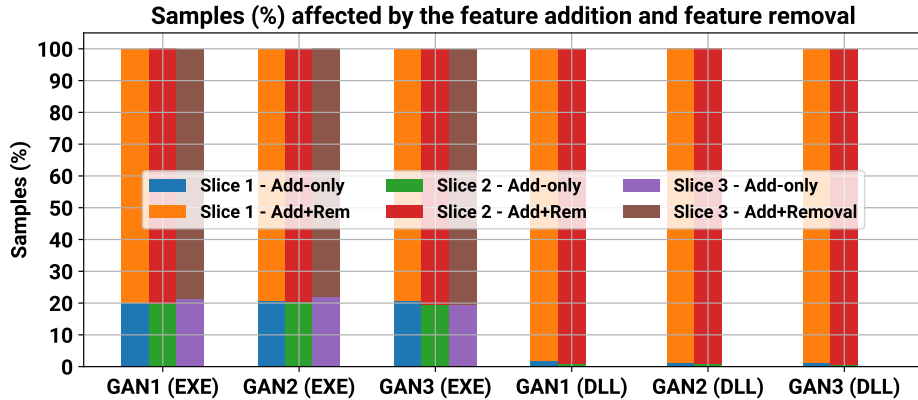


Figure 12: **GAN effect over samples.** Percentage of samples with added and removed features. GAN effect over samples.

Figure 12 shows the fraction of the GAN-generated evasive samples affected by feature removal and addition for each experimental setting. Our initial hypothesis was that GANs could individually add and remove features as needed to make each sample evasive. We found that, in practice, samples are affected in two ways: (1) they receive only feature additions, or (2) they receive both feature additions and removals simultaneously. The first case accounts for $\approx 20\%$ of EXEs and only 1% of DLLs (this difference arises from the presence of function exports as features). Therefore, in most cases ($\approx 80\%$ and 99% , respectively), GANs not only suggest adding new features to the binaries but also, in most cases, removing features, which significantly affects the attack’s feasibility. We observed no difference in results across multiple GAN architectures and dataset sizes, indicating that feature removal is intrinsic to the nature of the classification problem.

The previous results show that most samples had features removed from their feature vectors, thereby affecting the practicality of the attacks by potentially breaking the samples’ functionality. Thus, it is key to understand the extent to which feature addition and removal affect the samples. If only a single feature is removed, the sample may remain largely functional, whereas if many functions are removed, the sample is almost certainly broken.

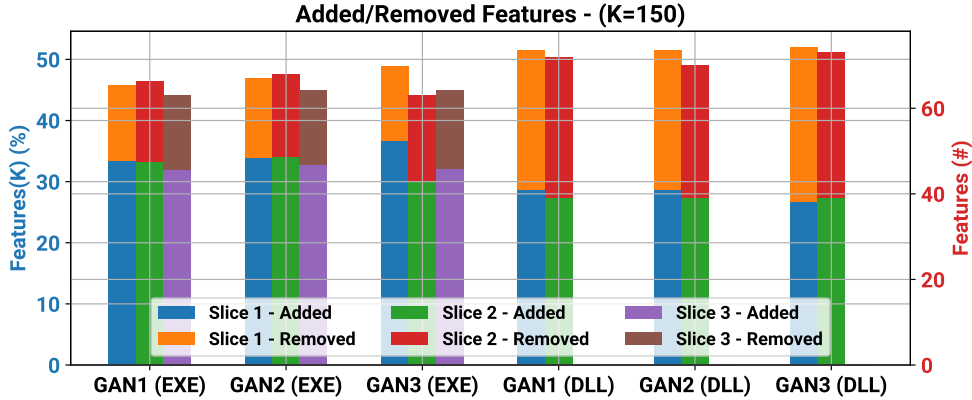


Figure 13: **GAN effect over samples.** Absolute and relative number of features affected in each sample.

GAN effect over samples.

Figure 13 shows the average absolute and relative numbers of features affected in each sample for each experimental setting. On average, 70 distinct features are modified in each sample ($\approx 50\%$ of the top-150 considered features). When we break down the results by type of change, we find that, on average, 50 features are added to the samples ($\approx 33\%$ of the top-150), and 20 features are removed ($\approx 13\%$). No significant difference was observed in the multiple settings, which indicates that the nature of the problem requires a significant number of changes. These results indicate that the number of removed features is non-negligible, thereby significantly affecting attempts to transition the attacks from feature space to problem space.

Verdict: *The large majority of GANs-generated evasive binaries require the removal of features, which might break their original functionality.*

RQ2.6 What is the achieved potential for GAN-based viable malware generation? A common approach to avoid breaking the generated binaries is to limit GANs to only add new features, i.e., even when GANs suggest removing features, these features are kept in the original binary. This approach was used by the three original GANs considered in this paper. However, our previous result demonstrated that most samples are affected by removal. Therefore, it is plausible to hypothesize that, if we adopt this approach, most samples will remain non-evasive. We here evaluate this hypothesis to characterize how evasive the **viable** GAN-generated binaries are.

Figure 14 shows the ASR for all settings when not removing the features from the GAN-generated binaries. For the EXEs, the ASR decreased from almost perfect ($\approx 99\%$) in the ideal scenario (with removals) to $\approx 70\%$. This result corroborates our hypothesis that retaining features significantly reduces sample evasiveness. The case for DLLs is even more significant in highlighting the impact of keeping features. Since almost all ($\approx 99\%$) samples are affected by the removal, retaining these features significantly improves detection rates. Samples that kept the GAN-removed features had their ASR drop from the original over 95% value to the 20% to 30% range. This result shows not only that feature removal is a significant aspect of evasive malware generation, but also that, in some scenarios (e.g., DLL), supporting feature removal is key to producing **viable** evasive binaries. This motivates our contribution: coupling an LLM with a GAN to generate viable binaries on demand from feature vectors.

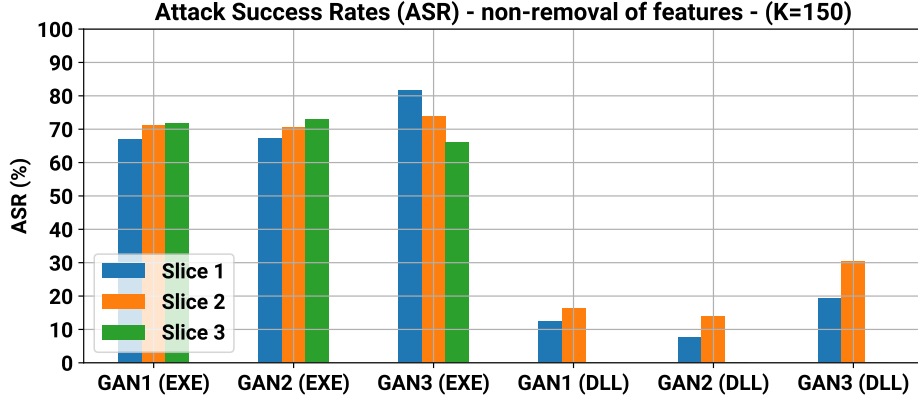


Figure 14: **GAN ASR when feature removal is not allowed.** The values are lower than in the ideal scenario.

GAN ASR when feature removal is not allowed.

Verdict: Preventing GANs from removing features to not harm malware functionality largely decreases their evasion capabilities, making DLL evasion impractical.

RQ2.7 How are GAN-based attacks affected by the size of the feature vector? The previous experiment demonstrated GAN performance in the same setting as in prior work, using a 150-dimensional feature vector. However, the size of the feature vectors (i.e., the number of functions and exports considered) may affect GAN performance. Since it was not evaluated in previous work, we here repeated the experiment with different feature vector sizes to characterize its impact on GAN performance. We varied the feature set size from 150 (as in previous work) to 1500 (10 times larger), which we considered a reasonable increment.

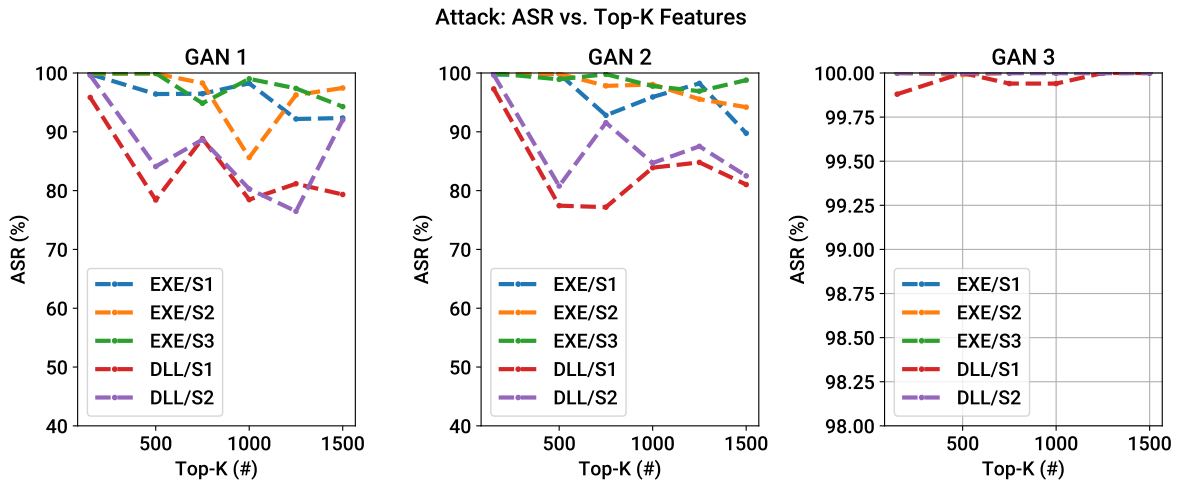


Figure 15: **Attack Success Rates vs. feature vector sizes.**

Attack Success Rates vs. feature vector sizes. Subplot (c) intentionally in a different scale to highlight its superior performance via the minimal variation margins.

Figure 15 shows the ASR for the different vector sizes when feature additions and removals are allowed. Compared with the original scenario, increasing the feature set size makes the problem more difficult. For some datasets and GAN configurations, the ASR decreases from nearly 100% to 80%. The third GAN architecture has proven more resilient to increases in vector size and has continued to present ASR close to 100%. Therefore, we conclude that increasing the feature size makes the problem more difficult, but GANs remain largely effective at generating evasive

samples.

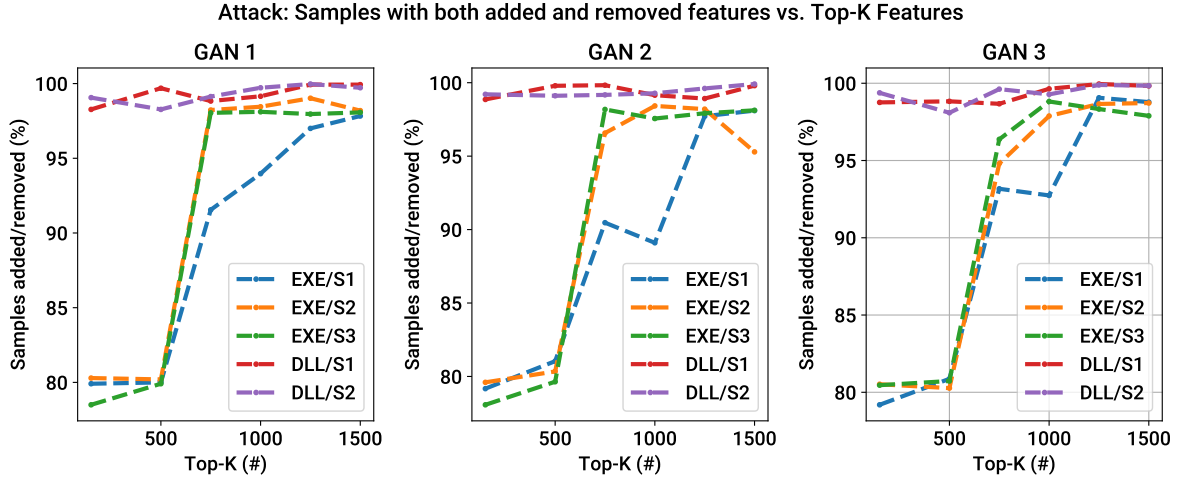


Figure 16: **Feature Removal Rates** vs. feature vector sizes.
Feature Removal Rates vs. feature vector sizes.

As in the previous experiments, we evaluated the fraction of samples affected by each type of feature operation. Fig. 16 shows the fraction of samples simultaneously affected by feature addition and removal. This covers most of the cases, as no sample exhibited only feature removal in any scenario. In the original scenario, most but not all samples were affected by the removal. As feature sizes increased, all samples were affected by the removal. Thus, although GANs remain effective at scale, not all of their generated samples may be functional due to feature removal.

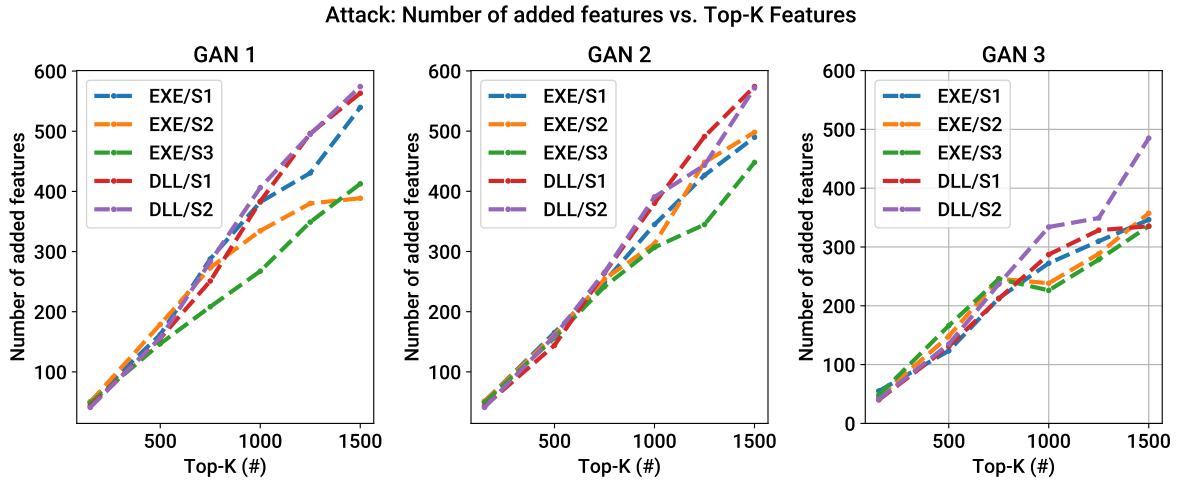


Figure 17: **Added Features** vs. feature vector sizes.
Added Features vs. feature vector sizes.

Figure 17 shows the number of added features to each sample for each GAN. The growth is linear: although larger feature vector sizes require a greater absolute number of features, the relative proportion of the added features remains the same. Interestingly, the third GAN architecture is the one that adds fewer features, despite being the most evasive generator.

Figure 18 shows the number of removed features for each sample for each GAN. It is notable that for all GANs, the number of features initially increases with the size of the feature set, then stabilizes. The third GAN architecture removed the most features, which explains its superior evasion capabilities.

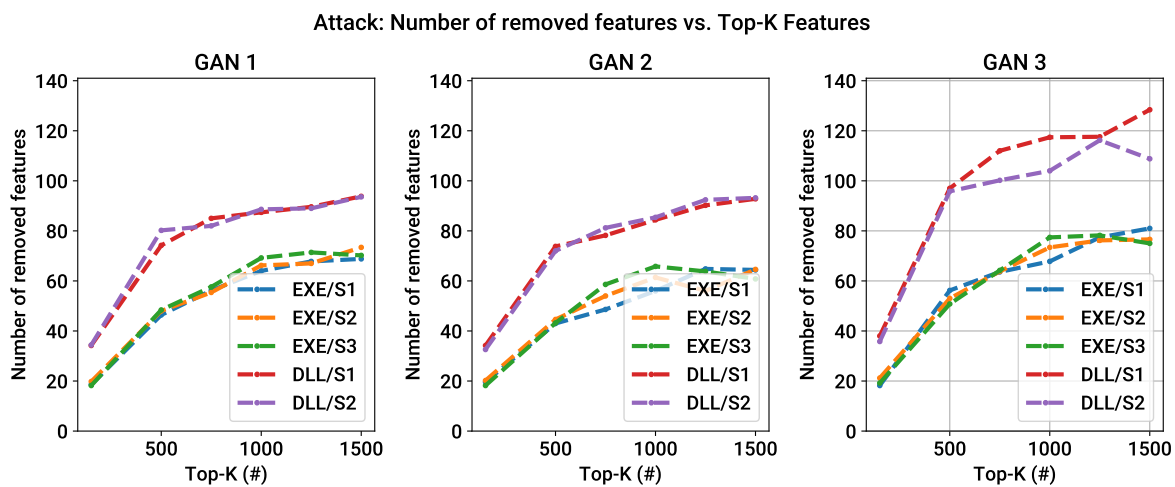


Figure 18: **Removed Features** vs. feature vector sizes.
Removed Features vs. feature vector sizes.

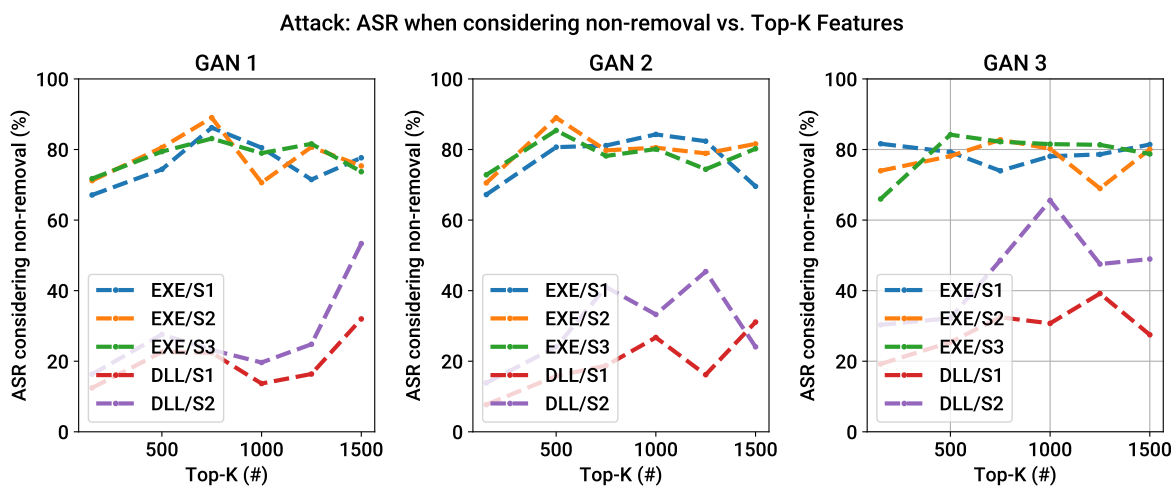


Figure 19: **Attack Success Rates** for feature addition only.
Attack Success Rates for feature addition only.

We also evaluated the strategy of allowing only feature additions to avoid breaking the final binaries. Figure 19 shows the ASR achieved by all GANs while employing this strategy. The removal restriction significantly affects all GANs, reducing their evasion capabilities. Even the third GAN, which was previously close to 100% ASR, now exhibits an ASR of at most 80%. Again, DLL evasion becomes impractical in all cases.

Verdict: *Increasing the feature vector size makes the evasion problem more difficult, although the GANs can still generate a significant number of evasive samples. The number of added and removed features grows at most linearly with the size, remaining stable in relative terms. However, because almost all samples are affected by feature removal, the generated samples may also be functionally broken. Limiting GANs to feature addition significantly reduces GAN performance across all cases and renders evasion of DLLs impractical.*

RQ2.8 How do GAN-based attacks transfer between detectors? We proceeded with our investigation by adding real-world constraints to the experiments. Our previous experiments assumed an ideal scenario in which the substitute detector had the same architecture as the black-box detector. However, in the real world, targets (e.g., AVs and ML detectors) might have architectures that differ from the GAN’s surrogate model. Thus, it is key to understand how GAN attacks transfer across detectors to assess their generalization.

We tested this by generating attacks with one GAN and detecting them with the substitute detector of a different one. We also tested the generated samples against a detector of a different architecture (the NFS model [17]). We selected this model as a reference due to its open-source availability and its validation in a malware evasion challenge, which makes it very aggressive at classifying samples as malware. We coupled the GANs with the NFS model by directly feeding the GAN feature vector into the NFS feature extractor. The NFS features not considered by the GAN remained unchanged in the NFS model and were therefore still used for classification.

Table 2: **Attack Transference.** EXE files.

	G1/S	G2/S	G3/S	G1/L	G2/L	G3/L	NFS
G1/S	99%	3%	16%	21%	0%	45%	0%
G2/S	33%	99%	12%	10%	0%	45%	0%
G3/S	100%	99%	100%	0%	0%	97%	0%
G1/L	5%	12%	83%	95%	0%	0.5%	0%
G2/L	0%	1%	97%	0%	100%	0%	0%
G3/L	0%	100%	0%	0%	0%	100%	0%

Table 2 shows the ASR for the EXE files across three GANs (1, 2, 3) in their (S)mall (150) and (L)arge (1500) configurations. The first column shows the generator, and the remaining ones show the detectors. The diagonals exhibit high values because this is the ideal scenario in which substitute and black-box models share the same architecture. The other cells represent attack transferability rates. It is noticeable that the different GAN architectures significantly affect ASRs. no GAN-generated attacks that completely transferred to all other architectures simultaneously. Increasing the feature set size did not affect transferability, as the architecture plays a greater role. GAN3 achieved the biggest number of transferability cases, reaching 100% ASR in some scenarios. When we consider the NFS model, no attack transfer is observed, thus reinforcing that the model architecture and features are key to attack transfer.

Table 3 shows the ASR results for the same experiments but for the DLL files. Once again, the results vary substantially with the detector architecture. In this case, the impact of the feature set size is clearly observed: GAN3 exhibited significant ASR on the small set and negligible performance on the large one. Also, in this case, the GAN attacks transferred to NFS, even at

Table 3: **Attack Transference.** DLL files.

	G1/S	G2/S	G3/S	G1/L	G2/L	G3/L	NFS
G1/S	99%	73%	43%	66%	2%	43%	24%
G2/S	38%	99%	37%	99%	0%	38%	31%
G3/S	87%	95%	100%	99%	99%	9%	0%
G1/L	82%	22%	80%	98%	0%	7%	98%
G2/L	24%	11%	50%	4%	78%	1%	94%
G3/L	0%	0%	0%	99%	100%	100%	98%

a high rate (98%). The increased attack transfer rate observed for DLLs relative to EXEs is hypothesized to result from DLLs having more data to consider (their function exports), thereby making the GAN’s output closer to the NFS model.

Verdict: *Attack transference is possible, even completely (100%), but this is rare. In most cases, the attacks only partially transfer. Attack transference is largely architecture-dependent. Transference to different models was observed for DLLs but not for EXEs.*

RQ2 Summary. Our experiments demonstrate that GANs are efficient evasive generators at all scales, but the bigger the scale, the more functions are removed, and thus the more samples are broken. Thus, the only way to make GANs effective in the problem space is to couple them with the LLM generators, as we propose. This coupling should differ between DLLs and EXEs, as their behaviors are distinct.

5.3 RQ3. Does the GAN-LLM combined approach produce more viable binaries?

Once we characterize GitHub Copilot and GANs, we can use their identified best strategies and parameters in combination. In further experiments, we ask GitHub Copilot to generate malware and test it with the GANs. We select only the **previously validated functional GitHub Copilot samples** for this test to establish ground truth. The samples are submitted to the GANs for detection using surrogate models and for transferability evaluations. The detected samples are then modified by the GANs under their best settings (e.g., wrt. feature set size), and a request to code the proposed modifications is forwarded to GitHub Copilot. The resulting modified binary is then re-tested in the GANs and substitute models in an iterative process.

RQ3.1 How evasive are the samples originally created by GitHub Copilot? We tested all GitHub Copilot-generated samples against the previously presented GANs and the NFS model. The first line in Table 4 shows the results for the GANs trained with the previously presented datasets. We observe low detection rates across all GANs, indicating that GitHub Copilot can generate evasive malware.

Table 4: **Attack Detection.** GitHub Copilot-generated malware.

	G1/S	G2/S	G3/S	G1/L	G2/L	G3/L	NFS
Original	27%	69%	0%	37%	74%	56%	100%
Retrained	100%	100%	100%	100%	100%	100%	100%

Low detection rates result in a few samples triggering the GAN generation processes. Thus, we adopted an Adversarial Retrain (AR) strategy to exercise it as much as possible by complementing the previous GAN training with GitHub Copilot-generated malware. The second line of Table 4 shows the detection rate for the hardened models. We conclude that AR is an effective defense against GitHub Copilot-generated malware.

Verdict: *GitHub Copilot-generated malware is evasive against naive GAN detectors. Adversarial retraining is an effective defense.*

RQ3.2 What is the detection rate of GitHub Copilot-generated malware after GAN-based function addition? We proceed by measuring the impact of GAN’s feedback on the detection of the generated code. We initially evaluated the impact of allowing GANs to add only new functions to the binary, as this is how GANs have traditionally been used. This experiment considered only samples initially detected as malware by the GANs. The attacker aims to add the GAN-suggested functions to make samples evasive.

Table 5: **Evasion Rate.** GAN-based dead code addition.

	GAN1	GAN2	GAN3
1 Iteration	+13%	+14%	+16%
3 Iterations	+15%	+21%	+22%

Table 5 shows the rate of modified samples that can evade detection compared to the original detection rate for each classifier in their best settings. The first line shows the increase rate at the first time (first iteration) when a GAN is applied to a given binary. We notice that all GAN+GitHub Copilot combinations could generate concrete evasive samples to some extent.

Verdict: *The addition of GAN-suggested dead code to the GitHub Copilot-generated malware increases their evasiveness, thus demonstrating the benefits of coupling GANs to the LLMs.*

RQ3.3 What is the success rate in dead code addition? Previous results (RQ2.6) showed that GANs were effective (>80%) in generating AEs via dead code addition. However, the previous result (RQ3.2) showed that although still effective, the numbers achieved by GitHub Copilot+GAN are smaller in practice. The root cause is that GitHub Copilot cannot always generate compilable code for the required functions (RQ1.2), so we skipped these cases during generation. This results in the final binary not presenting all functions suggested by the GAN, but an approximation of them.

Table 6: **Code Gadgets.** Code generation success rate.

Affected	GAN1	GAN2	GAN3
Samples (%)	60,00%	99,00%	64,00%
Samples (# Funcs)	2,44 (5.5%)	3,22 (30%)	4,04 (15.6%)

Table 6 shows the rate of samples affected with at least one failed dead function generation attempt, the rate of skipped functions per sample, and the relative rate of skipped functions over the total amount of dead functions requested. The results show that GitHub Copilot is a bottleneck in the integrated malware generation process, as GANs are easier to train to suggest functions to add than to generate code for them.

Verdict: *Different GAN architectures result in the suggestion of different functions (and at different numbers) to be added to the GitHub Copilot-generated malware. The GitHub Copilot and GAN’s error causes are nested, as some of the GAN suggestions cause GitHub Copilot to produce non-compilable code in the first iteration.*

RQ3.4 Do more GAN iterations help? Since GitHub Copilot may not produce compilable code with a single request, our key insight is to make the malware construction iterative by applying GANs on top of a previously modified sample with GAN feedback. There are two hypotheses

behind it: **First**, if the GAN requests the same function as in a previous round, GitHub Copilot will have a second chance to produce compilable code. **Second**, the functions added in previous rounds might change the probability that GANs add functions, so the next GAN request might be for a different function, with a new chance for GitHub Copilot to learn to generate compilable code for it. We evaluated these hypotheses by submitting the previous samples to multiple (3) GAN iterations (limited due to computational resource constraints). Tables 5 and 7 show that more iterations indeed increase the ASR. We investigated the root causes of evasion and found that most of it (97.7%) is due to GANs repeatedly requesting the same functions over iterations, thereby giving GitHub Copilot additional opportunities to succeed. In other cases (2.3%), this occurs because the GANs request different functions, thereby indicating the potential for iterative malware generation.

Verdict: *More GAN iterations improve malware-generation efficiency. Their effect is twofold: increasing evasiveness by suggesting alternative implementations and giving GitHub Copilot a second chance to generate a compiled building block that previously failed.*

RQ3.5 What is the detection rate of GitHub Copilot-generated malware after GAN-based function removal? To conclude, we evaluate the effect of letting GANs also suggest removing functions (the ideal scenario from RQ2.1). Note that we are not removing functionality but allowing GANs to suggest LLMs to prevent the inclusion of detectable functions. Also note that we are not breaking the binary functionality as done by naive GAN approaches (RQ2.5), but using LLMs to generate code with the same original semantics but without using the functions suggested by the GANs, as designed.

Table 7: **Evasion Rate.** GAN-based function removal.

	GAN1	GAN2	GAN3
1 Iteration	+21%	+40%	+42%
3 Iterations	+22%	+44%	+54%

Table 7 shows that the evasion increase via function removal is greater than via dead-code insertion (Table 5). Thus, we conclude that integrating GitHub Copilot and GANs allows GANs’ performance to reach values closer to the ideal (RQ2.1) than limiting their operation to addition-only (RQ2.6).

Verdict: *GAN’s guidance makes GitHub Copilot’s generated samples more evasive (compared to their standalone GitHub Copilot baseline). The more iterations, the greater the stealthiness.*

RQ3.6 Does the GAN-LLM method produce more evasive malware than other AE generation methods? We also evaluated how our GAN-LLM solution compares with traditional AE generation strategies: byte-based [20], FGSM, header modifications, and Dropper (packer) embedding [17]. The comparison is not straightforward, as GAN-LLM operates at the source-code level, whereas traditional AE generators operate only at the binary level. We compared our solution with theirs in a restricted scenario in which ours operated only on the dropper layer of the binary, not on the embedded payload, to demonstrate that we still achieved superior performance. We conduct classic AE testing using two classic detectors: byte-based Malconv [43] and PE function-based NFS [17]. In all cases, we started with samples detected as malware and aimed to flip their labels to benign.

Table 8 clearly demonstrates how each attack targets a different threat model and thus performs better against a different model. Byte-based attacks are completely effective against Malconv but not against NFS, and vice versa. GAN-LLM is positioned as a solution closer to

Table 8: **Attack Success Rate (ASR);**

Defenses	Attacks			
	FGSM	Drop.	G-G(1)	G-G(10)
MalConv	100%	44%	34%	54%
NFS	22%	100%	88%	99%

PE-based operation; thus, its performance with a single iteration is similar to the dropper’s. With more iterations (e.g., 3), GAN-LLM achieves the same performance as the dropper, since it generates a dropper with dead functions to bypass NFS. Interestingly, with a large number of iterations (e.g., 10), GAN-LLM can perturb the PE bytes to a significant extent, causing flips even in the Malconv model, which operates under a different threat model. While GAN-LLM is not efficiently tuned for byte-flipping, it still performs, demonstrating the power of changing internal function implementations to generate diverse binaries.

Verdict: *AEs’ success depends on the type of feature (threat model) they target. Our GAN-LLM achieves performance comparable to the SOTA AE for its own threat model (PE header detectors) and, as a side effect, perturbs detectors for a different threat model (byte detectors).*

RQ3 Summary. Our experiments demonstrate that guiding GitHub Copilot malware generation with a GAN indeed yields more (and more evasive) samples, either because GANs suggest more diverse function implementations, or because the iterative malware generation process gives GitHub Copilot a second chance to re-implement previously-failed building blocks in a different way. This yields evasion rates comparable to the SOTA for AE generation against PE header-based classifiers. The number of added functions varies according to the GAN architecture.

5.4 RQ4. How do the created AEs transfer to the real world?

To evaluate the impact of the generated malware samples, it is key to verify their detection by the market detection solution. To that, we perform attack transferability testing against AVs. We remind that all samples are functional according to the sandboxing tests previously reported in Table 1. The reader can check samples execution in VirusTotal’s sandboxes^{1 2 3}.

RQ4.1 What is the AV detection rate of GitHub Copilot-generated malware? We followed the strategy from related work [7] and submitted all generated (and functionally validated) samples to VirusTotal over 2 weeks, thereby simulating a server-side polymorphic distribution. This test aims to verify whether AVs recognize the generated samples as real threats (rather than prototype artifacts) and to establish a baseline for the AV detection evasion rates induced by the GAN-suggested manipulations, which are evaluated in subsequent experiments.

Figure 20 shows the distribution of the detection rate for the multiple (20K+) samples (all variants/queries) in their multiple formats (EXEs and DLLs). All generated samples were detected by at least one AV, indicating that GitHub Copilot can generate binaries that are actually recognized by AVs. Whereas most samples were detected by multiple AVs, hundreds of samples were detected by only a few (or none), indicating that the strategy is viable for creating evasive variants in the long tail—i.e., that the attackers will succeed in creating a final

¹<https://www.virustotal.com/gui/file/f4e5097d2ea3cf0beb099e32e7261578590a7a3c40c3b1c8d71c3ddf1e5bf220/behavior>

²<https://www.virustotal.com/gui/file/ab4330b8aaa16861231816aa760b50de91733325d92202cbceb963189f678dfe/behavior>

³<https://www.virustotal.com/gui/file/d9ce06b2cdf15163adf648178bc22101d05b0c4c9a9acee684b02703c5d9bb19/behavior>

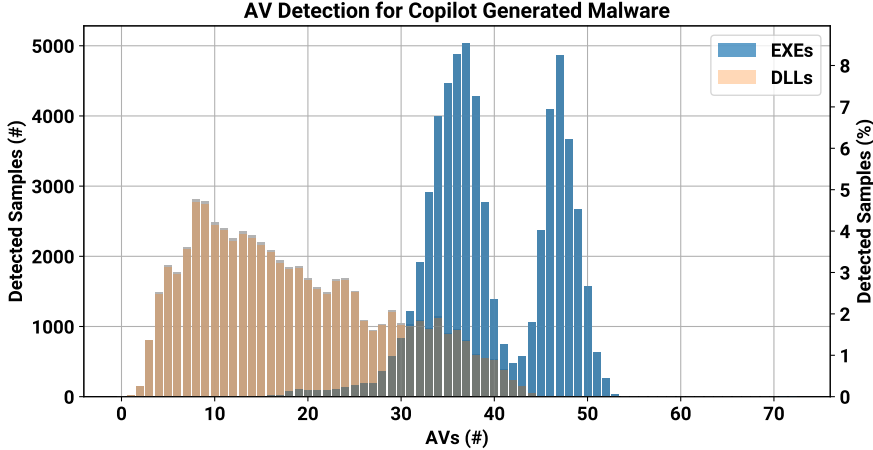


Figure 20: **VirusTotal Detection.** Original GitHub Copilot malware. VirusTotal Detection. Note that hundreds of samples are detected by no AV (scaling issue).

evasive malware instance if they keep repeating the process until this point. While the process might not be necessarily query-efficient, this is not a requirement for attackers, who optimize for effectiveness. Remember the lemma: *defenders must cover all breaches, while attackers need only find a single gap*. The presented EXEs curve with 2 Gaussians is due to the AV updates occurring in between [9]. This result aligns with the ones obtained for GPT-3 [7].

Verdict: *GitHub Copilot produces binaries that AVs recognize as malware. Most samples are detected, but hundreds have low detection rates. This long-tail effect is consistent with observations of GPT-3 malware.*

RQ4.2 Are the DLL detection rates different? Previous results revealed that typical EXEs and DLLs present different characteristics for both attacks and defenses. Therefore, it is key to evaluate if these same differences are present in AV detection rates. To that, we regenerated the same malware building blocks as before, but now compiled them as DLLs to test their detection by AVs. Figure 20 shows detection rates broken down by EXEs and DLLs. It is noticeable that DLL samples are detected by fewer AVs than EXEs, a systemic AV weakness. The Gaussians are not observed as the detection of DLLs did not grow over time. We highlight that this difference in results across binary structures is often overlooked in many ML-for-systems papers, yet it should not be neglected, given its significant impact.

Verdict: *GitHub Copilot-generated DLL malware samples are less detected by VirusTotal's AVs than EXEs.*

RQ4.3 Are the GAN-adversarial samples detection rates lower? We also measured attack transference for the GAN-generated samples to evaluate the hypothesis that the samples modified by GitHub Copilot after GAN instructions are more evasive to AVs than the original GitHub Copilot-generated samples.

Table 9: **AV Detection (#)** vs. GAN Iterations.

	Iteration 0	Iteration 1	Iteration 2	Samples (#)
GAN1	48	48 (-0%)	47 (-2.08%)	-416
GAN2	56	55 (-1.78%)	55 (-0%)	-356
GAN3	54	53 (-1.85%)	46 (-14.81%)	-3277

Table 9 shows the average number of Virustotal AVs detecting the set of samples that were modified by the GANs (we discarded the originally evasive samples). We notice that the samples

are on average more evasive after GAN-instructed modifications. We also notice that more iterations make the samples become more evasive on average. The effect is more notable for the GAN3.

Verdict: *GitHub Copilot-generated samples after GAN instructions are more evasive to AV detection than baseline GitHub Copilot’s malware. The more GAN iterations are performed, the more evasive the samples become. The GAN architecture also affects attack transference rates to AVs.*

Individual samples were affected beyond the average/ Since an attacker might focus on specific variants, we here take a further look at the extreme effects, i.e., the most affected samples. It is important to note that the average is significantly affected by samples whose detection rates were not influenced by the GANs (44%, 55%, and 37% for GANs 1 to 3, respectively).

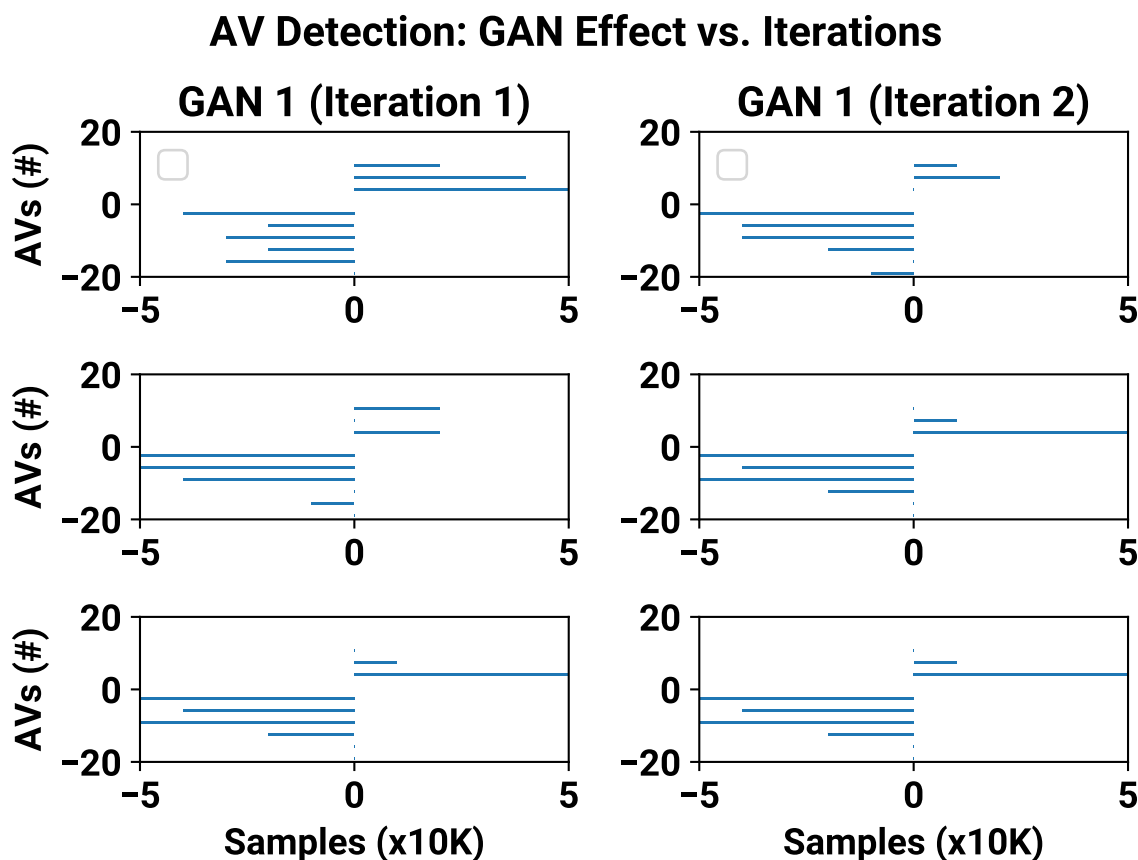


Figure 21: AV Detection rates. (In/De)crease vs. GANs.
AV Detection rates.

Figure 21 shows the distribution of samples (in multiples of 10K) according to the number of AVs that changed their minds over the multiple iterations, discarding the ones that remained stable (the 0 line). We report results as the difference between the number of detected AVs in the first GAN iteration over the original sample and in the second GAN iteration over the first GAN iteration. We observe that, for affected samples, most saw a decrease in detection rates across 20 VirusTotal AVs, indicating that GANs are effective against real detectors. Some samples, however, exhibit increased detection rates with the addition of functions. The results also hold across iterations. The second iteration reduced the aAV detection rate compared to the first round, and the original samples were included.

Verdict: *Individual samples are significantly more affected than the average. While the GAN modifications might increase AV detection for some samples, they can completely evade AVs (>20 evasions per iteration) for many (thousands) samples.*

RQ4 Summary. Both pure GitHub Copilot-generated malware and GAN-modified malware can be evasive to AV detection at significant numbers. GAN-modified samples indeed become more evasive, as expected. The previously observed evasion pattern extends to AVS: the more GAN iterations, the more evasive the samples become. However, the effects are observed at different scales, both across all samples and in individual samples, reinforcing that the attacker’s advantageous position stems from the approach’s long-tail strategy.

5.5 Application: Predicting Future Threats

We believe the main reason a defender uses a synthetic malware generator is to retrain adversarial classifiers. We hypothesize that our approach can “*predict the future*” by generating novel malware with features (i.e., not previously observed by a classifier) and that it enables classifiers to become more robust via retraining on these samples. We tested this hypothesis using the classic experiment that compares the classifier’s performance with and without adversarial retraining on the generated samples. More specifically, we conducted a classic concept-drift experiment on our dataset by splitting it into disjoint training and test sets to avoid data leakage [5]. We ran the experiments on the NFS detector [17], which served as the reference in previous experiments (RQ2.8).

Table 10: **Adversarial Retraining** as a Mitigation.

Model	Train	Test	Model	Train	Test
Original (AUT):	99.3%	78.54%	Retraining (AUT):	98.7%	89.45%

Table 10 shows the performed experiments’ results. Both classifiers learned the training dataset well, and their performance degraded over time (train vs. test set results), as expected. However, the classifier without adversarial retraining degraded substantially more, thereby demonstrating that generating synthetic yet valid malware samples is an effective strategy for mitigating the studied threats.

Verdict: *The GAN-LLM generated samples can be used not only for adversarial retraining strategies aiming for increased robustness to evasion, but also as a temporal data augmentation approach for retraining classifiers to resist concept drift.*

6 Conclusion

We proposed coupling GANs and LLMs to bridge the gap between the problem and feature spaces and to efficiently produce AEs that outperform ML-based malware detectors. We implemented 3 GANs and integrated them with GitHub CoPilot for code generation and compilation. We found that CoPilot samples are evasive against many GANs and AVs, and that adding a GAN feedback loop increases the likelihood of evasion. The more GAN iterations, the greater the evasion.

Ethical Concerns. Our concern is that attackers integrate GANs and LLMs to deploy malware in massive numbers through automated creation. We do not intend to teach attackers how to generate malware. They will learn on their own [18]. Our position is that it is better to know an attack is possible before the attackers do so, so we can anticipate rather than catch up with them. Thus, the strategies for generating malware and defending against it are hereby disseminated to a wider academic community, and the generated samples have been shared on VirusTotal for community AV detection.

Future Work. Current works are limited to relying on externally defined malicious targets, either malware behaviors in previous works or GAN-specified functions in ours. In future work, we will investigate the LLM’s ability to generate malware from more general definitions than those manually specified by attackers.

Reproducibility Limitations. We generated the samples using Copilot 1.250 with GPT-4o in VSCode 1.96, always using the default configurations. No compiler optimization was performed. All VirusTotal scans were performed fresh over a continuous two-week period, immediately after the sample compilation. IOCs were checked for 100% coverage using VirusTotal’s own sandboxes. We acknowledge that LLM research is time-sensitive and that attempts to reproduce our results may not yield exactly the same numbers due to the evolution of tools and models. We do expect our conclusions to hold for the processes to which malware generation is subject if the same methodology is employed.

Reproducibility Effort. To allow the full reproduction of our presented results and mitigate the above-discussed limitations, we are making all prompts, scripts, samples, and sandbox logs available at: <https://github.com/marcusbotacin/Automated.Malware.Generation>

Acknowledgments. We thank the National Science Foundation (NSF) for the support via the CNS 2327427 grant.

References

- [1] Raúl Acosta-Bermejo, José Alexis Terrazas-Chavez, and Eleazar Aguirre-Anaya. Automated malware source code generation via uncensored llms and adversarial evasion of censored model. *Applied Sciences*, 15(17):9252, 2025.
- [2] Sharon Adarlo. Hackers create chatgpt rival with no ethical limits. <https://news.yahoo.com/hackers-create-chatgpt-rival-no-142712150.html>, 2023.
- [3] Md Ajwad Akil, Adrian Shuai Li, Imtiaz Karim, Arun Iyengar, Ashish Kundu, Vinny Parla, and Elisa Bertino. Llmalmorph: On the feasibility of generating variant malware using large-language-models, 2025.
- [4] Anthropic. Disrupting the first reported AI-orchestrated cyber espionage campaign — anthropic.com. <https://www.anthropic.com/news/disrupting-AI-espionage>, 2025.
- [5] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. Dos and don’ts of machine learning in computer security. In *Proc. of USENIX Security Symposium*, volume 1, page 1, USA, 2022. USENIX.
- [6] AutoIt. Autoit. <https://www.autoitscript.com/site/>, 2023.
- [7] Marcus Botacin. Gpthreats-3: Is automatic malware generation a threat? In *2023 IEEE Security and Privacy Workshops (SPW)*, pages 238–254, USA, 2023. IEEE.
- [8] Marcus Botacin, Hojjat Aghakhani, Stefano Ortolani, Christopher Kruegel, Giovanni Vigna, Daniela Oliveira, Paulo Lício De Geus, and André Grégio. One size does not fit all: A longitudinal analysis of brazilian financial malware. *ACM Trans. Priv. Secur.*, 24(2), jan 2021.
- [9] Marcus Botacin, Fabricio Ceschin, Paulo de Geus, and André Grégio. We need to talk about antiviruses: challenges & pitfalls of av evaluations. *Computers & Security*, 95:101859, 2020.
- [10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini

- Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901, online, 2020. Curran Associates, Inc.
- [11] Thomas Bühler. <https://web.archive.org/web/20240529031531/https://www.avira.com/en/blog/gan-made-malware>, 2019.
- [12] Thomas Bühler. Applying gans to malware detection. an introduction. <https://web.archive.org/web/20240811013651/https://www.avira.com/en/blog/applying-gans-to-malware-detection>, 2019.
- [13] Zhipeng Cai, Zuobin Xiong, Honghui Xu, Peng Wang, Wei Li, and Yi Pan. Generative adversarial networks: A survey toward private and secure applications. *ACM Comput. Surv.*, 54(6), jul 2021.
- [14] Luca Cavaglione, Michał Choraś, Iginio Corona, Artur Janicki, Wojciech Mazurczyk, Marek Pawlicki, and Katarzyna Wasielewska. Tight arms race: Overview of current malware threats and trends in their detection. *IEEE Access*, 9:5371–5396, 2021.
- [15] Fabrício Ceschin, Marcus Botacin, Albert Bifet, Bernhard Pfahringer, Luiz S. Oliveira, Heitor Murilo Gomes, and André Grégio. Machine learning (in) security: A stream of problems. *Digital Threats*, 1(1):1, sep 2023. Just Accepted.
- [16] Fabrício Ceschin, Marcus Botacin, Heitor Murilo Gomes, Luiz S. Oliveira, and André Grégio. Shallow security: on the creation of adversarial variants to evade machine learning-based malware detectors. In *Proceedings of the 3rd Reversing and Offensive-Oriented Trends Symposium*, ROOTS’19, New York, NY, USA, 2020. Association for Computing Machinery.
- [17] Fabrício Ceschin, Marcus Botacin, Gabriel Lüders, Heitor Murilo Gomes, Luiz Oliveira, and André Grégio. No need to teach new tricks to old malware: Winning an evasion challenge with xor-based adversarial samples. In *Reversing and Offensive-Oriented Trends Symposium*, ROOTS’20, page 13–22, New York, NY, USA, 2021. Association for Computing Machinery.
- [18] Checkpoint. Opwnai : Cybercriminals starting to use chatgpt. <https://research.checkpoint.com/2023/opwnai-cybercriminals-starting-to-use-chatgpt/>, 2023.
- [19] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, S. Arun Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374:1, 2021.
- [20] Luca Demetrio, Battista Biggio, Giovanni Lagorio, Fabio Roli, and Alessandro Armando. Functionality-preserving black-box optimization of adversarial windows malware. *IEEE Transactions on Information Forensics and Security*, 16:3469–3478, 2021.

- [21] Luca Demetrio, Scott E. Coull, Battista Biggio, Giovanni Lagorio, Alessandro Armando, and Fabio Roli. Adversarial examples: A survey and experimental evaluation of practical attacks on machine learning for windows malware detection. *ACM Trans. Priv. Secur.*, 24(4), sep 2021.
- [22] Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks. In *Proceedings of the 28th USENIX Conference on Security Symposium, SEC'19*, page 321–338, USA, 2019. USENIX Association.
- [23] Aeryn Dunmore, Julian Jang-Jaccard, Fariza Sabrina, and Jin Kwak. Generative adversarial networks for malware detection: a survey, 2023.
- [24] erocarrera. Pefile. <https://github.com/erocarrera/pefile>, 2016.
- [25] Jennifer Fernick. On the malicious use of large language models like gpt-3. <https://www.nccgroup.com/research/on-the-malicious-use-of-large-language-models-like-gpt-3/>, 2021.
- [26] Github. Your ai pair programmer. <https://copilot.github.com/>, 2020.
- [27] Google. Pathways language model (palm): Scaling to 540 billion parameters for breakthrough performance. <https://ai.googleblog.com/2022/04/pathways-language-model-palm-scaling-to.html>, 2022.
- [28] Weiwei Hu and Ying Tan. Generating adversarial malware examples for black-box attacks based on gan. In Ying Tan and Yuhui Shi, editors, *Data Mining and Big Data*, pages 409–423, Singapore, 2022. Springer Nature Singapore.
- [29] Deqiang Li, Qianmu Li, Yanfang (Fanny) Ye, and Shouhuai Xu. Arms race in adversarial malware detection: A survey. *ACM Comput. Surv.*, 55(1), nov 2021.
- [30] Heng Li, ShiYao Zhou, Wei Yuan, Jiahuan Li, and Henry Leung. Adversarial-example attacks toward android malware detection system. *IEEE Systems Journal*, 14(1):653–656, 2020.
- [31] Microsoft. Turing-nlg: A 17-billion-parameter language model by microsoft. <https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/>, 2020.
- [32] Microsoft. Run-time dynamic linking. <https://learn.microsoft.com/en-us/windows/win32/dlls/run-time-dynamic-linking>, 2021.
- [33] Microsoft. Visual studio code and github copilot. <https://code.visualstudio.com/blogs/2023/03/30/vscode-copilot>, 2023.
- [34] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430, USA, 2007. IEEE.
- [35] Nitin Naik, Paul Jenkins, Nick Savage, Longzhi Yang, Tossapon Boongoen, and Natthakan Iam-On. Fuzzy-import hashing: A static analysis technique for malware detection. *Forensic Science International: Digital Investigation*, 37:301139, 2021.
- [36] OpenAI. Openai python library. <https://github.com/openai/openai-python>, 2020.

- [37] Yin Minn Pa Pa, Shunsuke Tanizaki, Tetsui Kou, Michel van Eeten, Katsunari Yoshioka, and Tsutomu Matsumoto. An attacker’s dream? exploring the capabilities of chatgpt for developing malware. In *2023 Cyber Security Experimentation and Test Workshop*, CSET 2023, page 10–18, New York, NY, USA, 2023. Association for Computing Machinery.
- [38] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. An empirical cybersecurity evaluation of github copilot’s code contributions. *CoRR*, abs/2108.09293:1, 2021.
- [39] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Can openai codex and other large language models help us fix security bugs? *CoRR*, abs/2112.02125:1, 2021.
- [40] Hammond Pearce, Benjamin Tan, Prashanth Krishnamurthy, Farshad Khorrami, Ramesh Karri, and Brendan Dolan-Gavitt. Pop quiz! can a large language model help with reverse engineering? *CoRR*, abs/2202.01142:1, 2022.
- [41] Xiaowei Peng, Hequn Xian, Qian Lu, and Xiuqing Lu. Semantics aware adversarial malware examples generation for black-box attacks. *Applied Soft Computing*, 109:107506, 2021.
- [42] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1332–1349, USA, 2020. IEEE.
- [43] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles Nicholas. Malware detection by eating a whole exe, 2017.
- [44] Maria Rigaki and Sebastian Garcia. Bringing a gan to a knife-fight: Adapting malware communication to avoid detection. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 70–75, USA, 2018. IEEE.
- [45] Ashkan Sami, Babak Yadegari, Hossein Rahimi, Naser Peiravian, Sattar Hashemi, and Ali Hamze. Malware detection based on mining api calls. In *Proceedings of the 2010 ACM Symposium on Applied Computing, SAC ’10*, page 1020–1025, New York, NY, USA, 2010. Association for Computing Machinery.
- [46] Shimony and Tsarfati. Chatgpt please write me a piece of polymorphic malware. <https://www.youtube.com/watch?v=kRF3CoQ9jfk>, 2023.
- [47] TrendMicro. Hype vs. reality: Ai in the cybercriminal underground. https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/hype-vs-reality-ai-in-the-cybercriminal-underground?utm_source=trendmicroresearch&utm_medium=smk&utm_campaign=0723_AIcybercriminal-TW, 2023.
- [48] Jianhua Wang, Xiaolin Chang, Jelena Mišić, Vojislav B. Mišić, Yixiang Wang, and Jianan Zhang. Mal-lsgan: An effective adversarial malware example generation model. In *2021 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, Spain, 2021. IEEE.
- [49] Jianhua Wang, Xiaolin Chang, Yixiang Wang, Ricardo J. Rodríguez, and Jianan Zhang. Lsgan-at: enhancing malware detector robustness against adversarial examples. *Cybersecurity*, 4(1):38, Dec 2021.
- [50] Dong-Ok Won, Yong-Nam Jang, and Seong-Whan Lee. Plausmal-gan: Plausible malware training based on generative adversarial networks for analogous zero-day malware detection. *IEEE Transactions on Emerging Topics in Computing*, 11(1):82–94, 2023.

- [51] Javier Yuste, Eduardo G. Pardo, and Juan Tapiador. Optimization of code caves in malware binaries to evade machine learning detectors. *Computers & Security*, 116:102643, 2022.

A Building Blocks Implementation

In this section, we qualitatively detail the implementation choices proposed by GitHub Copilot for the requested malicious Building Blocks (BBs).

BB1. Debugger Identification:

- **IsDebuggerPresent.** The most frequent detection method is to call this API in the context of one's own process.
- **CheckRemoteDebuggerPresent.** The remote version of the above API. Common variations involve retrieving the process handle to pass as a function argument.
- **IsDebuggerPresent + CheckRemoteDebuggerPresent.** A common variation that attempts to check the presence of the debugger using the two APIs.
- **PEB check via API.** This method checks the debug flag in the Process Environment Block (PEB) struct by parsing it at a high level. This code explicitly declares the PEB struct.
- **PEB check in ASM.** This method checks the debug flag in the PEB structure by directly accessing the PEB struct via inline ASM, requiring no struct declaration.
- **NtQuerySystemInformation.** Checks for the presence of a debugger by retrieving system information from the API. This method, in fact, checks for the presence of a kernel debugger.
- **Debug Registers.** This method emits inline assembly to check if debug registers are set or not.
- **Exception Triggering.** This method emits inline assembly to purposely cause a division by zero and trigger its exception mechanism. The exception is not triggered if a debugger first traps it.

BB2. CPU Identification:

- **CPUID Intrinsics.** This method emits compiler intrinsics for the CPUID instruction to retrieve the processor vendor. The variants for this method consist of multiple ways to handle the resulting vendor string.
- **CPUID Inline ASM.** This method is a variation of the above method in which the compiler intrinsic is replaced by the inline assembly for dispatching the same CPUID instruction.
- **GetSystemInfo.** This method invokes a system API to retrieve system information. The variations involve checking different struct fields, such as the `ProcessorInformation`, `ArchitectureInformation`, or `OEM` information.
- **IsFeaturePresent.** This method checks if the processor has a specific feature. Intel features are used as a proxy for identifying an Intel processor.
- **GetEnvironmentVariable.** This method performs a check for a `ProcessorIdentifier` variable to identify the host CPU. This method might fail, as this variable might not always be set on a system.

- **Register Key Check.** This method performs a check for a `CentralProcessor` key under the `Hardware` tree for CPU information.

BB3. AutoRun Setting:

- **RegSetValueKey.** All variations used this API function call as it is the interface to write in the registry. Variations from these methods are: (1) in the way keys are created (directly using the `CreateKey` function or later invoking the `OpenKey` function); (2) the API versions used to manipulate the register (`OpenKeyEx` vs. `OpenKey`); and (3) clean-up actions (some code versions do not close key handles).

BB4. XOR Encoding:

- **Bytewise-XOR.** All variations of this method implemented a byte-wise XOR in a for loop. The observed variations are due to the way the stop condition is identified. Some versions assumed that the key was a string of fixed size (hardcoded); some measured it using `strlen`, and others evaluated it via an `IF` inside the loop.

BB5. DLL Injection:

- **CreateRemoteThread.** All variations of the building block were implemented using the same traditional DLL injection approach. No other injection technique was generated by GitHub Copilot. The variants for this implementation are: (1) how to get the process handle; (2) how (and if) verify for errors; (3) thread management, i.e., if suspend the processes threads or not, if resume threads or not, or if wait for thread termination or not; and (4) clean up routines, i.e., if the code closes handles or not and if it free allocated temporary buffers or not. An important aspect to highlight is that, in many cases, the implementation variants are in the use of static or dynamic linking. In many cases, the resulting code relied upon dynamic function imports (e.g., via `GetProcAddress`). Dynamic function resolution is common in malware samples — especially those that inject DLLs. We hypothesize that GitHub Copilot generated code in this manner because it might have been trained on educational malware code databases containing the code to mimic obfuscated DLL injectors.

BB6 Load From File:

- **C API.** This method uses the native C APIs (e.g., `fseek` and `fgetc`) to manipulate the files. Variations of this approach concern whether the generated code closes the open handles.
- **WinAPI.** This method differs from the above by using the Windows API functions (e.g., `OpenFile` and `ReadFile`) instead of the native C ones. Variations from this method are in the different manners to handle a file (e.g., `CreateFile` and `OpenFile` can be used interchangeably).

BB7. Decode Base64:

- **Manual Algorithm Implementation.** In this method, the GitHub Copilot outputs code to implement the base64 algorithm directly by subsequently manipulating the encoded string.
- **CryptStringToBinaryA.** The invocation of this API is the natural way to decode base64 via the Win API. Variations from this method involve the different flags passed as arguments and the use (or not) of an auxiliary buffer. It is important to note that, as with the DLL injection building block, GitHub Copilot also generated dynamic function-resolving code in this case. Our hypothesis for that is, once again, based on its frequent use by malware and the training on educational security repositories.

BB8. Shellcode Execution:

- **CreateThread.** This method consists of creating a new thread to run the shellcode as a new function.
- **Pointer Execution.** This method consists of changing the permission of the memory page hosting the shellcode to executable and running it from the memory address. Variations from this method include the use or not of auxiliary buffers.

BB9. Delete File:

- **WinAPI.** This method consists of using the WinAPI functions (e.g., `DeleteFile`) to manipulate the file. Variations are in the check for corner conditions (e.g., if the file is NULL).
- **C API.** This variation consists of using the C API functions (e.g., `remove`) to manipulate the file.

BB10. Delete Itself:

- **GetModuleHandle + DeleteFile.** This method used the WinAPI functions to identify the current process's file and remove it.
- **CMD Argument.** This method identifies the current process's file as above, but removes it by passing a removal argument to a newly spawned shell (CMD).
- **BAT file.** This method works similarly to the above one, but the removal command is written to a new `.bat` file. The file is executed via the `system` API call.
- **MoveFile to NULL.** This method identifies the current process's file as above but removes the file by moving it to a NULL location via setting the file attributes flags in the function invocation parameters.