CS 4994 Fall 2022
Nanomites as an Anti-dumping Technique: An Extended Reality Case Study
Literature Review

Justin Perez*
Department of Computer Science
Virginia Tech

ABSTRACT

Often times, there is a need for developers to hinder dynamic and static analysis of their software. This form of cybersecurity is becoming more common for the video game industry, where video game publishers wish to prevent piracy and cheating. Typically, this is done with commercial-off-the-shelf software, that can prevent debugging, provide control flow obfuscation, and encrypt and decrypt code on-demand. In this paper, we present a technique for obfuscation on Windows and Linux `x86-64` binaries labeled *Nanomites* that provides Anti-Debugging, Anti-Dumping, and effectively hinders dynamic and static analysis, with minimal overhead. Additionally, we provide an open-source proof of concept under a GNU General Public License. We also investigate how this technique can help protect Extended Reality (XR) applications, in the video gaming discipline.

1 INTRODUCTION

Cybersecurity is a serious problem spanning a wide range of domains, from critical infrastructure, to personal computers, personal data, and even people's lives. One of these domains is the protection of code, that may contain intellectual property or trade secrets, and can be reverse engineered. When it comes to XR applications, there's often a need for the protection of proprietary algorithms. Our paper focuses on a novel implementation of a software-based protection, titled *nanomites*. *Nanomites* hinders static analysis by inserting junk opcodes, and dynamic analysis by forcing the target program to run with a specially-crafted debugger. Previously, *Nanomites* has only been implemented publicly by a commercial, proprietary protector Armadillo [6]. Here, we provide information on the technique, including a complete proof-of-concept written in Rust. We explore the Intel x86-64 Architecture and *Nanomites* in this instruction set architecture (ISA).

When compiled languages, like C and C++ are compiled, the compiler transforms the source code into machine code. This machine code is interpreted by the CPU. The machine code is often verbose for simple lines of code, but reverse-engineers can piece this information together to create an unwatned understanding of an application. Our focus for this paper is on the Intel Instruction Set, which has the largest market share of 93% of all processors [2].

This machine code can be disassembled, and turned into human-readable instructions through a tool called a disassembler. Other tools, like decompilers found in reverse engineering tools, can turn these series of instructions into psuedo-C code, which makes it trivial to reverse engineer compiled programs.

This creates a problem for developers, whose code can contain information that they wish to keep secret. Take for example patented algorithms, whose implementation is proprietary. Companies have a need to protect their intellectual property from unauthorized copy and reverse engineering. The potential for them to lose revenue if a competitor can recreate their intellectual property is real, and a threat to most businesses.

*e-mail: justinmp@vt.edu

In the XR domain, there are sometimes proprietary algorithms used by the developers to provide a more immersive experience for the user. Such algorithms can be localization, determining the exact position of the player in the room. Others like 3D motion recognition, near-zero wireless communication also exist and are valuable to competitors.

In addition, there is a need for privacy of personal private information in XR applications. XR applications haven the potential to collect sensitive data, such as biometrics, eye movement data, and images or videos of a person. Biometric data can be extremely valuable to a malicious actor, as users cannot change it. There may also be private information in these applications, such as names, emails, addresses, phone numbers, etc.

2 LITERATURE REVIEW

Little research has been done in protecting XR applications. The most common software used to build XR applications are Unity and the Unreal Engine [7]. An XR application is a game built with these technologies. These games often allow for *scripts*, or custom code a developer uses, to automate actions on *entities* or *game objects* in response to events. This can include things like moving the player based on keyboard events.

2.1 Security & Privacy

Kenwright, in his paper about *The future of Extended Reality*, discusses the metaphysical issues of XR, as well as the cybersecurity aspects that must be protected [11]. Kenwright is careful about precisely defining the domain of XR applications, as the future of XR is still much in the early phase of development.

One of the security issues is the emotional impact of XR applications, and the content within the application. Consider an XR application with user-generated content. There must be systems in place to moderate this content, as some users, particularly minors, should have discretion on the content seen.

In addition, Kenwright discusses the protection of users, and the user's data. Consider the amount of personal data an XR application could collect, store, and transit. This data could include biometric details, as well as private personal information [11]. This data could be compromised by a malicious actor.

Unfortunately, this data cannot be protected by *nanostorm*. *nanostorm* can only protect the application itself, not the content within the XR application.

2.2 Detecting Personality Traits from Eye-Tracking Data

In Berkovsky's land-mark paper, they discovered personality traits can be classified through only eye-tracking data in the form of images and videos from an XR device [4]. Here, they use machine learning to train a Naive-Bayes classification model on a data set of images and videos.

With machine learning models, it's common for the model to not be shipped inside consumer applications, and instead hosted through a REST API [5]. In fact, most popular apps host internal machine learning models through a machine-learning-as-a-service API. This

poses challenges for the privacy of the data, as requests to the REST API can be intercepted over the internet.

An attacker could steal eye tracking data by sniffing the data if they're connected to the local area network, or even reading the memory of the XR application capturing the eye movements. To prevent this, the data must be encrypted in transit, and resistant to man-in-the-middle attacks (most commonly through HTTPS & TLS). However, this data still must be secured in rest on the machine the XR application is running on.

Nanostorm and nanomites cannot protect this data, as it does not provide any memory protection to the application it is protecting.

Moreover, it is even possible for an attacker to steal machine learning models, and therefore recreate their own model [22]. This would allow for an attacker to intercept the raw eye-tracking data and run it themselves.

Personality traits are private personal data, and this information could be used to create a psychological model of a person to use in a social-engineering attack.

2.3 False Data Points and Timeliness

With XR applications, the risk of false data points also needs to be considered. Old or inaccurate data can lead to improper profiling or potential wrongful actions against an individual [14]. For example, certain applications might determine that an individual is moving slower compared to other individuals. If these individuals are participating in a competition that requires precision and micro-movements, and compares this data to others also in the competition, the affected individual could lose and not even know [14].

Thus, this data needs integrity, which cannot be provided by nanomites. Integrity of this data needs to come through other means, or a combination of nanomites and other memory integrity protections. We can only prevent the dumping of the executable over a machine an attacker controls.

It's also possible for an XR device to cryptographically sign the movement data, and verify this data on the server side to ensure authenticity and integrity of this data. However, the private key can be extracted from the onboard chip, if an adversary is sophisticated enough [18]. Techniques need to be explored to protect and provide integrity and confidentiality of this data.

2.4 Self-Modifying Code

Mavrogiannopoulos describes in the paper *A taxonomy of self-modifying code for obfuscation* that one can add anti-debugging and tamper resistance to a binary by modifying the target's code to implement breakpoints, by replacing the target code with software interrupt instruction (on x86 architectures) [12]. Likewise, software tamper resistance methods have been improved by using self-modifying code.

This technique thwarts debugging by already having a debugger attached, as only a single debugger can be attached to a process at a time. Due to the self-modifying nature, analysis of the code cannot be done statically, and must be performed dynamically, increased in difficulty by not being able to attach a debugger.

This technique is implemented, in a method not for obfuscation by the gdb to modify the target's code with breakpoints [21]. However, gdb is meant to aid debugging, not prevent analysis of code.

2.5 Software Cracking

Kanzaki mentions the importance of protecting software cracking, in relation to the copyright protection of software [9]. An example of a typical case is when a cracker examines a copy protection checking code and alters the program to disable the routine [9].

Crackers, who perform those illegitimate and illegal actions, create a serious threat to the software industry. This is especially relevant to the XR industry, as these companies are often start-ups attempting to pave the way with a new metaverse.

Kanzaki, et al., propose a method to add a self-modifying mechanism to the original program, to increase the cost of understanding the original program [9]. But these attempts only increase the difficulty in reverse-engineering the software, and eventually the software will be cracked by a well-informed reverse engineer.

The mechanism they propose modifies the code through an actual modification of the software's instructions at runtime. The code executes a deobfuscation routine that dynamically replaces other code in the process with the correct code, and then reapplies the obfuscation after the camouflaged instruction executes.

3 Problem Description

Our goal is to increase the difficulty for a reverse engineer to gain an understanding of the code contained in a Windows or Linux binary. Windows and Linux support a variety of instruction sets, notably both Intel x86-64 and ARM. A complete solution to prevent reverse engineering would have to be able to support all of these architectures, and use architecture-specific techniques to hinder analysis.

Here, we focus specifically on the Intel architecture. However, the nanomites technique can be applied to other ISAs, such as ARM (through the `BKPT` instruction [19]). The x86-64 architecture currently accounts for over 94% of processors currently in use. [2]. In addition, most XR applications are most vulnerable on consumer personal computers, where a malicious actor can easily inject code into other processes [13]. On devices like mobile phones, running unsigned code is challenging and requires jailbreaking [3].

Thus, the Intel architecture has the most widespread use, and the biggest impact and the focus point for our study.

The Intel x86-64 Architecture is a complex instruction set. Instructions in this set have variable instruction lengths, and can range from 1 byte to 15 bytes [8]. In this architecture, there is a special instruction, with the mnemonic `INT 3`. This instruction, is a one byte opcode (`0xCC`) that executes a call to the debug exception handler [8].

4 Proposed Approach

With Nanomites, we take advantage of the debug exception handler, the `INT 3` opcode is only 1 byte long, and the restriction that only one process can debug a given process at a time.

A *nanomite*, in the x86-64 architecture, is an `INT 3` instruction. [8]. Normally, `INT 3` instructions are placed by debuggers to set a breakpoint, and once stepped over, the debugger restores the original byte, and resumes normal execution of the program.

To employ this technique, we disassemble the target binary, and replace all or a portion of conditional jump instructions with an `INT 3`. The original conditional jump instruction's data, including the type of conditional jump (`jz, jnz, jg,` etc.), the size of the jump instruction, and the jump's offset relative to the instruction pointer are stored in a *Jump Data Table* (JDT), which is indexed by the original instruction's virtual address. Since jump instructions are often multiple bytes long, the rest of the bytes are replaced with random bytes.

Because we replace extra bytes in the jump instruction with random data, and the x86-64 architecture is a multibyte instruction-set architecture, we can effectively prevent static analysis of the binary with nanomites in it. This because the information about the original length of the jump instruction is lost. A reverse engineer does not know the length of the original disassembly, and thus the reverse engineer does not know how many bytes to skip and continue disassembling the binary.

To make matters more complicated, any instruction that contains the `INT 3` opcode is also encoded in the JDT. This prevents a reverse engineer from studying the JDT and statically replacing all entries in the table with the original jump instruction.

To complete this technique, a runtime (or called a stub) is strapped to the binary with nanomites. The runtime's function is first executed

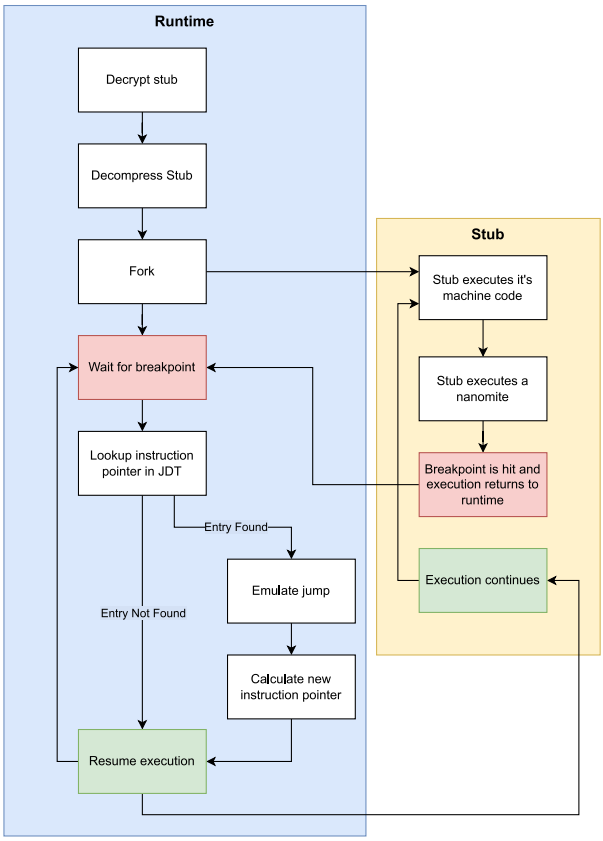

Figure 1: System overview of executing a nanomite protected binary

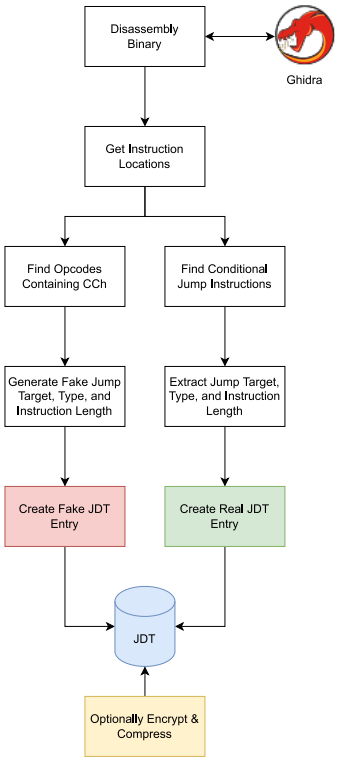

Figure 2: System overview of adding nanomites to a binary

in the nanomite binary, located in the runtime's data section. Then the runtime debugs the binary to be alerted when a nanomite is reached. Next, the runtime looks up the entry from the JDT, emulates the jump instruction by checking the `EFLAGS` register, and updates the instruction pointer.

4.1 Disassembly of the Target Binary

Disassembling a binary is not a trivial problem [15]. A disassembler has to account for various optimization techniques, such as jump tables. To account for this problem in our technique, we use the National Security Agency's Ghidra software that combines linear sweep disassembly with recursive disassembly, and other heuristics to get an accurate disassembly. This disassembly is then parsed by our tooling to find conditional jump instructions and instructions that contain an `INT 3` opcode.

5 Tool: Nanostorm

Nanostorm is our proof of concept tool that places nanomites in a target binary, and wraps a *stub* over the binary after nanomites has been replaced to allow normal execution and protection of the target binary [16].

5.1 Disassembly

First, *Nanostorm* uses *Ghidra* to dump the virtual addresses of all locations in the binary. As Ghidra uses a heuristic process to disassemble the binary, we get an accurate and mostly complete disassembly. This is done with a python script.

The python script outputs the virtual address of every instruction. This data is then read in by Nanostorm, and converted into the offset of the instruction in the binary.

5.2 Decoding Instructions

Our Ghidra script only gets the location of the instructions and by parsing the ELF or PE headers of the binary, we can extract the offsets into the binary of where the location of the instruction is.

To decode the instructions, we use a library, `iced-x86`, that is a blazingly fast Rust library to decode `x86-64` instructions.

With this library, we then extract the metadata of each instruction, whether the instruction is a conditional or unconditional jump, the length of the instruction in bytes, etc.

In particular, we're looking for conditional jump instructions, and instruction whose opcode contains a `0xCC`, the opcode for an `INT 3` instruction. When a conditional jump instruction is found, we take the type of conditional jump (`jnz`, `jz`, `jb`, etc...), the jump displacement, the length of the jump, and the virtual address of the instruction, and add this information to the *Jump Data Table*.

When an opcode that contains an `0xCC` is found, we create a fake entry in the JDT with the address of the `0xCC` byte. The jump type, displacement, and length are generated at random.

At the end, the JDT is optionally compressed and encrypted. The JDT get serialized to disk, to be included in a *stub* later.

5.3 Adding Nanomites

During this process, we are also modifying the original binary, and placing `INT 3` instructions. When a conditional jump instruction is found, the first byte of the instruction is replaced with an `INT 3`. Since jump instructions are often multiple bytes long, the rest of the bytes are replaced with randomly generated bytes.

Finally, the modified binary is again optionally compressed and encrypted, and then written to disk to be included in a *stub* later.

5.4 The Stub

The stub is the runtime for the new binary with nanomites. Since the new binary now contains `INT 3` instruction, if this binary was to be

executed without our custom runtime, the program would crash, as nothing would handle the breakpoint exception.

The stub is compiled with the JDT and the nanomite binary in the stub's `.rodata` section. At runtime, the stub determines if both the JDT and nanomite binary are compressed and encrypted, and then appropriately decrypts and decompresses.

Following this, the stub breaks down into two different ways for properly executing the nanomite binary, dependent on the Operating System.

5.4.1 Windows

Currently, *Nanostorm* has a less than ideal implementation on Windows. Since there is no built-in support for executing a program from memory, we first write the nanomite binary to a temporary folder, as specified in the environment variable `%TEMP%`. This is because techniques for running a PE File from memory are commonly used by malware, and Windows Defender and other Endpoint Detection & Response programs detect this a malicious action [20].

First, the file is written to the `%TEMP%` directory, queried from the Windows API function `GetTempPath`. Then a temporary file name is generated with `GetTempFileName`. The nanomite binary is written to disk with `CreateFile` and `WriteFile`. Lastly, the binary is executed with `CreateProcess`, with the `DEBUG_PROCESS` flag. The `DEBUG_PROCESS` flag "starts and debugs the new process and all child processes created by the new process." [10].

Next, the image base of the nanomite binary needs to be determined to properly compute the next `RIP`. This is achieved by querying the Process Environment Block (PEB). The PEB's location is queried with `NtQueryInformationProcess`, with the `ProcessBasicInformation` flag. The PEB contains the field `ImageBase`.

Using `WaitForDebugEvent`, we wait for an `INT 3` instruction. Next, we open a handle to the thread, with the thread ID provided by the `DEBUG_EVENT` structure provided by `WaitForDebugEvent`. The thread is then suspended, as to determine the `RIP` with `GetThreadContext`, the thread must be suspended. With the `RIP` from the thread's context, the entry in the JDT is located. Then, the jump is emulated, and the new `RIP` is computed, and finally set with `SetThreadContext`. The thread is then resumed with `ResumeThread`, and the exception is stepped over with `ContinueDebugEvent`.

Finally, once the `WaitForDebugEvent` informs us the process has exited, any handles are cleaned up, and the temporary nanomite binary we created is deleted.

5.4.2 Linux

Unlike Windows, the Linux runtime supports complete in-memory execution of the nanomite binary. It does this by first calling `fork()` to create a new child process. The child process creates a memory file descriptor with `memfd_create`. This "creates an anonymous file and returns a file descriptor that refers to it." [1]. Next, using `write`, and the file descriptor from `memfd_create`, we write the decompressed and decrypted nanomite binary to the file. Lastly, `fexecve` is used to run the binary from memory. `fexecve` is similar to `execve`, except it accepts a file descriptor to launch from, rather than a path. The arguments and environment variables are passed through to the nanomite binary.

The parent then uses `ptrace` with the `PTRACE_ATTACH` flag to attach a debugger to the child process. Next, a call to `waitpid` is issued to wait for any debug events by the child process. If the signal is a `STOPPED` signal, this means that the nanomite binary has executed a nanomite. Next, if we have not computed the base Virtual Address of the child, it is determined by reading `/proc/PID/maps` to determine the base address the binary was loaded into. This value is then saved for future nanomites.

Next, the registers are fetched with `ptrace` with the `PTRACE_GETREGS` flag. This returns the registers for the process' current state, including `RIP` and `EFLAGS`. This value is then looked up in the JDT, and if there is an entry, the conditional jump is emulated, and the RIP is updated with using *ptrace* with the `PTRACE_SETREGS` flag.

Lastly, the process is resumed using `ptrace` with the `PTRACE_CONT` flag. The process then eventually hits another nanomite or exits, repeating the steps here or exiting.

5.5 System Analysis

We also analyzed the performance of *nanostorm* itself, and measured the time it takes to infect a binary with nanomites. This was tested with a variety of binaries, and different size binaries commonly found on a Linux installation.

Notably, both Ghidra and nanostorm are multithreaded, so the performance and speed is dependent on the amount of cores on the system. These benchmarks were performed on a system with an AMD Ryzen 3900XT, which has 12 cores.

Performance was benchmarked with hyperfine, a command-line benchmarking tool. Hyperfine features statistical analysis across multiple runs, warm up runs that are executed before the actual benchmark, and statistical outlier detection to detect interference from other programs and caching effects [17].

The benchmarks were performed ten times over, and the mean and 95% confidence interval were recorded and displayed below.

Table 1: Nanostorm Time on Common Linux ELFs

Binary	**Size**	**Mean**	$\pm\sigma$
true	28 KB	8.493s	0.185s
ls	136 KB	13.616s	0.354s
openssl	1 MB	57.516s	0.970s

Nanostorm's time to process a binary scales with the size of the binary. This can be attributed to the parsing of address locations from Ghidra, which is written to the console and then parsed. In the future, a better way of passing this data should be implemented to decrease times.

6 Case Study: Superposition VR Benchmark

To measure the expected performance decrease with *nanostorm*, tests were done on the following system:

Table 2: Benchmarking System

CPU	AMD Ryzen 3900XT
Cores	12
GPU	AMD Radeon 6800
GPU VRAM	16 GB
RAM	32 GB DDR4-3200

We used the UNIGINE Superposition Benchmark, with the VR Optimum settings [23]. This benchmark evaluates the VR performance of an Oculus Rift VR headset, which has an eye resolution of 1332×1586.

We ran two tests, one on an unmodified version of Superposition, and another one where we ran *nanostorm* on Superposition, and benchmarked these. We measured performance with Frames per Second (FPS). We collected the Minimum FPS (FPS Min), the Average FPS throughout the benchmark (FPS Avg) and the maximum FPS obtained. (FPS Max).

Table 3: Nanostorm Superposition Benchmarks

Benchmark	FPS Min	FPS Avg	FPS Max
Unmodified	80.25	124.11	205.38
Nanomites	75.92	121.07	198.23

Through these benchmarks, we saw a 2%-6% performance decrease in the benchmarks with nanomites. Nanomites offer protection without a major sacrifice in performance. The main cause of this performance drop was due to the number of context switches caused by each nanomite. Notably, performance is also strong because nanomites does not affect the GPU. The CPU is only responsible for uploading data to the GPU, while the GPU renders the scene.

Additionally, nanostorm prevented us from attaching a debugger to the running benchmark. On Linux, there was some difficulty involved in dumping the binary with nanomites from memory. Memory could still be written and read from the running benchmark.

7 Discussion

nanostorm doesn't come without limitations. Notably, on Windows, the protected nanomite executable is dropped to disk to execute. This circumvents the encryption of the process in memory, and makes it trivial for a reverse engineer to analyze the nanomite binary.

In addition, performance of applications is reduced when they are protected with nanomites. This is because in order to function, the nanomite binary needs a special debugger to execute it normally. The additional system calls (and their associated context switches), jump emulation adds overhead to the application.

To further protect XR applications, which are typically creates with game engines, *nanostorm* needs to be extended to include protection for any library files (`.dll`s or `.so`s). To do this, *nanostorm* would first have to resolve what external libraries the application requires (ignoring system libraries like `kernel32.dll` or `libc.so`). Then, these files would have to be added to *Ghidra* for disassembly. The JDT would then need to become a hash map of hash maps. The outer hash map is keyed by the module, either the actual executable binary or a library. The inner hash map is the typical JDT, as discussed.

Lastly, nanomites do not offer protection against modification of memory in the nanomite binary. This means that, for instance, in an XR application, in-memory values like the orientation and position of objects are susceptible to modification.

To counter these weaknesses, one would have to combine nanomites with other techniques. For the context of XR applications, which are typically Windows applications, this would include DLL Injection detection, segment integrity checks, function hooking detection, etc.

8 Conclusion

In conclusion, *nanostorm* is an effective tool to hinder reverse engineering. It can prevent both static and dynamic analysis. When applied to XR applications, *nanostorm* is not a complete solution to protect the privacy and security of these applications. With the changes discussed previously, *nanostorm* can provide anti-analysis features to the entire XR application. However, the integrity of the application can still be compromised. Further research needs to be done in protecting the unique aspects of XR applications, such as commercial-off-the-shelf anti-cheat and anti-tampering software commercially available.

References

[1] *memfd_create(2) Linux User's Manual*, 5.10 ed., November 2020.

[2] P. Alcorn. Amd sets all-time cpu market share record as intel gains in desktop and notebook pcs. https://www.tomshardware.com/news/intel-amd-4q-2021-2022-market-share-desktop-notebook-server-x86, Feb 2022.

[3] Z. Ali. Developer talks about difficulties in ios 15 jailbreak development. https://ioshacker.com/news/developer-talks-about-difficulties-in-ios-15-jailbreak-development, Mar 2022.

[4] S. Berkovsky, R. Taib, I. Koprinska, E. Wang, Y. Zeng, J. Li, and S. Kleitman. Detecting personality traits using eye-tracking data. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, p. 1–12. Association for Computing Machinery, New York, NY, USA, May 2019. doi: 10.1145/3290605.3300451

[5] A. Cetinsoy, F. J. Martin, J. A. Ortega, and P. Petersen. The past, present, and future of machine learning apis. In L. Dorard, M. D. Reid, and F. J. Martin, eds., *Proceedings of The 2nd International Conference on Predictive APIs and Apps*, vol. 50 of *Proceedings of Machine Learning Research*, pp. 43–49. PMLR, Sydney, Australia, 06–07 Aug 2016.

[6] M. Fyrbiak, S. Wallat, J. Déchelotte, N. Albartus, S. Böcker, R. Tessier, and C. Paar. On the difficulty of fsm-based hardware obfuscation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, p. 293–330, Aug 2018. doi: 10.13154/tches.v2018.i3.293-330

[7] S. N. B. Gunkel, E. Potetsianakis, T. E. Klunder, A. Toet, and S. S. Dijkstra-Soudarissanane. Immersive experiences and xr: A game engine or multimedia streaming problem? (arXiv:2201.05552), Jan 2022. arXiv:2201.05552 [cs]. doi: 10.48550/arXiv.2201.05552

[8] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, April 2022.

[9] Y. Kanzaki, A. Monden, M. Nakamura, and K. Matsumoto. Exploiting self-modification mechanism for program protection. In *Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003*, p. 170–179, Nov 2003. doi: 10.1109/CMPSAC.2003.1245338

[10] Karl-Bridge-Microsoft. Process creation flags (winbase.h) - win32 apps. https://learn.microsoft.com/en-us/windows/win32/procthread/process-creation-flags.

[11] B. Kenwright. The future of extended reality (xr). *Communication Article. January*, 2020.

[12] N. Mavrogiannopoulos, N. Kisserli, and B. Preneel. A taxonomy of self-modifying code for obfuscation. *Computers & Security*, 30(8):679–691, Nov 2011. doi: 10.1016/j.cose.2011.08.007

[13] OWASP. Code injection owasp foundation. https://owasp.org/www-community/attacks/Code_Injection.

[14] S. Pahi and C. Schroeder. Extended privacy for extended reality: Xr technology has 99 problems and privacy is several of them. (4202913), Aug 2022. doi: 10.2139/ssrn.4202913

[15] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE Symposium on Security and Privacy (SP)*, p. 833–851, May 2021. doi: 10.1109/SP40001.2021.00012

[16] J. Perez. Nanostorm. `https://github.com/melotic/nanostorm`, 2022.

[17] D. Peter. hyperfine. https://github.com/sharkdp/hyperfine, Sep 2022.

[18] K. Ryan. Hardware-backed heist: Extracting ecdsa keys from qualcomm's trustzone. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, p. 181–194. Association for Computing Machinery, New York, NY, USA, Nov 2019. doi: 10.1145/3319535.3354197

[19] D. Seal. *ARM Architecture Reference Manual*. A Pearson education book. Addison-Wesley, 2001.

[20] M. Security. Uncovering cross-process injection with windows defender atp. https://www.microsoft.com/en-us/security/blog/2017/03/08/uncovering-cross-process-injection-with-windows-defender-atp/, Mar 2017.

[21] R. Stallman, R. Pesch, and S. Shebs. *Debugging with GDB: the GNU source-level debugger*. Free Software Foundation, 2002.

[22] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Stealing machine learning models via prediction APIs. In *25th USENIX Security Symposium (USENIX Security 16)*, pp. 601–618. USENIX Association, Austin, TX, Aug. 2016.

[23] UNIGINE. Performance benchmarks by unigine. https://benchmark.unigine.com.