Evaluating the Effectiveness of Obfuscated Instruction Codes for Malware Resistance

Lucas L Ritzdorf, Colter Barney, Christopher M Major, Tristan Running Crane,
Hezekiah Austin, Benjamin Macht, Clemente Izurieta and Brock J LaMeres
Montana State Unviersity

***Abstract*—Malware and maliciously crafted user input represent serious threats to modern computer systems. Many attacks begin with difficult-to-prevent vulnerabilities, such as code injection or memory corruption, usually achieved by exploiting known bugs in specific programs. We introduce a novel processor architecture which utilizes obfuscated hardware in order to effectively detect altered memory contents before they can be executed. This detection system also uses hardware reconfigurability to its advantage, providing the flexibility to counteract other attack vectors relying on code execution and demonstrating significant resistance to brute-force attacks. In addition, we present a prototype architectural implementation as an initial demonstration of feasibility.**

***Index Terms*—Computing, CPU central processing unit, OS command injection, cyber security, endpoint security, cyber-attack, attack detection, FPGA**

I. Introduction

In 2022, The MITRE Corporation tracked over twenty-five thousand new Common Vulnerabilities and Exposures (CVEs) across a variety of categories. Vulnerabilities which allow an attacker to "execute code" and "overflow" memory regions were the first and third most common such categories, respectively [1] (see Fig. 1). While there exist a variety of methods to guard against these types of vulnerabilities, such techniques are insufficient to fully prevent their occurrence and subsequent exploitation.

Modern programmable logic devices have enabled the practical implementation of redundant processors on a single chip. This facilitates the compile-time assignment of obfuscated instruction codes to otherwise functionally equivalent processor cores. In this paper, we present an evaluation of a novel processor architecture, utilizing heterogeneous obfuscated cores to prevent the execution of malware injected into memory. This architecture would confer immunity to all known attacks which rely on the improper insertion of executable code into memory. In addition, such hardware diversity has been shown to produce increased resilience to system failures in an information technology monoculture environment [2].

II. Background

A. Memory Overflows

Memory overflow vulnerabilities are particularly infamous for both their relative commonality and potentially devastating effects. Several distinct classes of software development mistakes can lead to an overflow, but all give rise to the same general condition: data over which a user has control is written to an improper memory location, which the user may also be able to influence. (Specific relevant vulnerabilities include "write-what-where" conditions and other out-of-bounds writes [3], all of which enable the same or similar attacks compared to the "classic buffer overflow.") Utilizing this access, a malicious user with sufficient knowledge of the vulnerable program can craft their input such that it causes the program to write addresses, executable instructions, or both into memory.

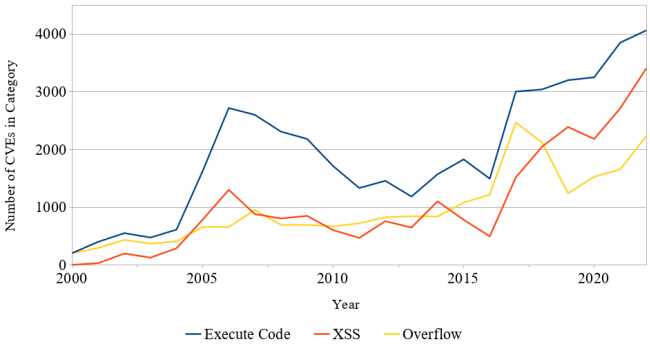


Fig. 1. Incidences of the three most common (in 2022) CVE classes over time. Data captured from [1].

Overflows can be exploited to enable a variety of attack mechanisms, such as initiating low-level system procedures, launching a reverse shell, or manipulating the return address stack to cause the execution of instructions elsewhere in the program. All such vulnerabilities fall into the category of arbitrary code execution (ACE), meaning that an attacker could exploit them to run any code they desire on an affected system. The situation is worsened when vulnerable input channels are accessible via a network, leading to remote code execution (RCE) scenarios, in which potentially devastating attacks can be conducted by an attacker with nothing more than network access.

B. Redundant and Reconfigurable Hardware

Processor redundancy has been used effectively within aerospace flight computers as a way to overcome faults caused by environmental factors, and serves as a foundation for this work. Redundant computing has proven especially effective against faults caused by radiation strikes, which can, for example, flip bits in memory or otherwise disrupt the configuration parameters and operational state of a computer system. These problems can be mitigated by running multiple processor cores

in lockstep, executing identical instructions and comparing their outputs (a technique known as modular redundancy). If a single core is affected by a radiation strike, its output will differ from that of the other cores, allowing for error detection and correction by way of an "output voter" module and memory-scrubbing systems. An architecture implementing this approach, as well as a sophisticated fault recovery system, is described by Hane, LaMeres, Kaiser, *et al.* [4], and forms part of the conceptual and functional basis for the architecture proposed here.

In addition to redundancy, this research demonstrates another important concept — the use of field-programmable gate arrays (FPGAs) to implement custom, reconfigurable hardware. An FPGA is a programmable logic device composed of two main component classes: configurable logic blocks and programmable interconnect points. Configurable logic blocks can be configured to emulate a variety of combinational and sequential logic circuits, typically with up to six inputs [5]. A single FPGA might contain several thousand such blocks, which are wired together using programmable interconnects. These act as configurable switches, knitting logic blocks together into a cohesive "fabric" and routing signals between blocks as needed.

The relevant result of this is that an FPGA is capable of being configured to essentially *become* an arbitrary digital system (up to and including a computer processor), as long as the chip model in use has sufficient resources to implement the desired hardware. FPGA configuration is performed by an automated synthesis program, which typically reads hardware description language (HDL) code as input, generating a bitstream which is written to the FPGA's configuration memory. The effects of this reconfigurability are discussed in Sections III-B and IV-C.

III. Approach

The primary exploit channel of concern in this investigation is the injection of executable code (i.e. binary instructions) into memory, and the subsequent execution of this code. Preventing development errors which give rise to such vulnerabilities has proven to be quite difficult, as evidenced by their continued occurrence and the high proportion of relevant CVEs described in Section II-A and illustrated in Fig. 1. Instead, we focus on detecting the presence of injected code in memory, and preventing its execution.

A. Architectural Concept

As the mechanism for this defense, we propose a customized processor architecture, composed of multiple uniquely designed compute cores. These cores run in lockstep, similarly to existing modularly redundant systems, but with a key distinction: the control units in each are designed such that they understand different instruction codes from any other core in the system. This yields a heterogeneously obfuscated processor where, for instance, an `ADD` instruction might be represented by the opcode `0x53` in one core, and by `0x09` in another. This is accomplished by applying an offset O_c to the decoding logic of each core c, and modifying the compiler such that any original opcode N is translated into $N_c = N + O_c$. Note that the "obfuscated opcode" N_c will be decoded correctly only by a core c whose internal logic incorporates the appropriate offset O_c. Furthermore, since each processor will contain multiple obfuscated cores, the opcode translation step is performed for each core (with its unique offset O_c during compilation, in order to generate an appropriate number of obfuscated instruction memory blocks.

One notable effect of this obfuscation technique is that, in order to successfully execute, a given program must be compiled especially for the exact hardware on which it is to be used. Any improperly compiled program will fail to decode on at least one core, causing an error. This should substantially increase the difficulty of creating a binary executable that will run on this obfuscated architecture to begin with, providing some measure of protection against malware.

While such "security through obscurity" measures should not be relied upon, the true strength of our architectural concept lies in its heterogeneity, rather than its obfuscation. Since all cores run in parallel and receive the same operational inputs, they execute in a functionally equivalent manner, despite each receiving differently obfuscated instruction codes. As a consequence of this, in the event that a memory injection attack does occur, *each core will load the same injected instructions* from shared data memory. Upon execution of said instructions, they will be correctly decoded by at most one core (or, more probably, none at all), creating an easily detectable attack signature.

To further improve the strength of this approach, we envision a separate "voter" hardware module designed to constantly monitor the instruction register in each core, which would raise a security flag upon detecting identical opcodes in all, or just some, of the cores. If implemented using combinational logic, independently from the remainder of the processor, such a module could facilitate injection detection and recovery within a single clock cycle of attempted execution.

B. Proposed Implementation

Recent advances in field-programmable gate array technology should facilitate the design, synthesis, and implementation of the architecture described here. FPGA-based hardware has the advantage of avoiding the rather involved process of semiconductor chip design, which we imagine most researchers would not be prepared to perform, particularly on the scale required to construct such a novel multi-core processor. The use of an FPGA would also significantly ease the implementation of the voter hardware module just described, as each core could easily be augmented with data lines to feed the contents of its instruction register into the module.

In addition, an FPGA-based system could be reconfigured on-demand, for instance to recover from accidental disclosure of instruction offset values by generating new hardware with different configuration parameters. Especially in this regard, we view FPGA design as more than simply a research and development tool — its use in a production-ready version of

our proposed architecture could significantly improve the level of security achievable in such a system.

C. Practical Design

In order to better evaluate our proposed design, we are currently developing a basic functional prototype based on the open RISC-V instruction set architecture (ISA). Among other considerations, this prototype has allowed us to validate that the implementation of a complex redundant processor system on an FPGA is indeed feasible. The current iteration includes four functional cores, each with 4 KiB of obfuscated instruction memory and 4 KiB of data memory. Figure 2 illustrates various features of the prototype, implemented on a Xilinx Artix-7 35T device.

In addition to assessing basic design feasibility, we are developing a sample application for the system, based on image processing tasks which can be made to leverage the strengths of our custom hardware. This will include such functions as I^2C and SPI communication, while also testing the system's ability to perform real-time data processing and eventually recover from a proper memory injection attack.

This application will be implemented not as a binary executable in the typical sense — instead, our prototype system is designed to behave more like an embedded microprocessor than a standard user-facing computer. The compilation toolchain consists of the RISC-V `gcc` compiler, its accompanying `readelf` program, and a custom Python script which translates binary disassembly into hardware description language constants. These are output in the form of a VHSIC Hardware Description Language (VHDL) package, the language in which our prototype is developed. VHDL files are converted into an FPGA programming bitstream via the device manufacturer's synthesis chain, which in this case is provided by Xilinx's Vivado software. The exported VHDL package from our own toolchain is incorporated into the synthesis process as a source of compile-time constants, including per-core obfuscation offsets (which affect the cores' control state machines) and instruction memory contents.

IV. Conceptual Evaluation

In addition to the basic infeasibility of code injection attacks, our proposed architecture has properties that significantly impede other known exploit techniques. While we again emphasize that "security through obscurity" should not be relied upon alone, we believe that it can serve as a useful layer within a more complex security strategy. We now consider the strengths and potential weaknesses of FPGA-based hardware obfuscation against a few relevant attack vectors.

A. Code Execution

As noted in Section III, a heterogeneous obfuscated processor is substantially more difficult to program for than a standard, homogeneous processor of known architecture. This is primarily because it will successfully execute only binaries which have been explicitly compiled to work with its particular architectural parameters — specifically, the number of parallel cores configured in the system, and the opcode offset for each. Thus, without access to these parameters, a potential attacker would be unable to compile malware or other binaries for an obfuscated system.

In practice, we anticipate that this could protect against such techniques as DLL injection, adding binaries to `PATH`, and many other methods of running malicious programs. Further, while strong defense postures are of course necessary in any security-sensitive context, obfuscated architecture stands in a category of its own, implementing deeply-integrated hardware attack countermeasures and the potential for detection within a single clock cycle.

There are, however, attack vectors against which hardware measures are likely to be less effective — they may not protect against scripting attacks implemented using, for instance, Python or Bash. Though community opinions on the relevant terminology vary, we will use the word "script" in reference to code which is not used to generate processor-level instructions, but is instead executed by way of an interpreter. This interpreter is typically implemented as a binary executable which (as before) must be compiled for the particular target system, so an adversary who attempts to download a generic interpreter as part of their attack would be unable to utilize it. If a usable interpreter has been installed on the system, however, the environment is no more secure than a standard computer. The interpreter will execute normally, loading and running the malicious script.

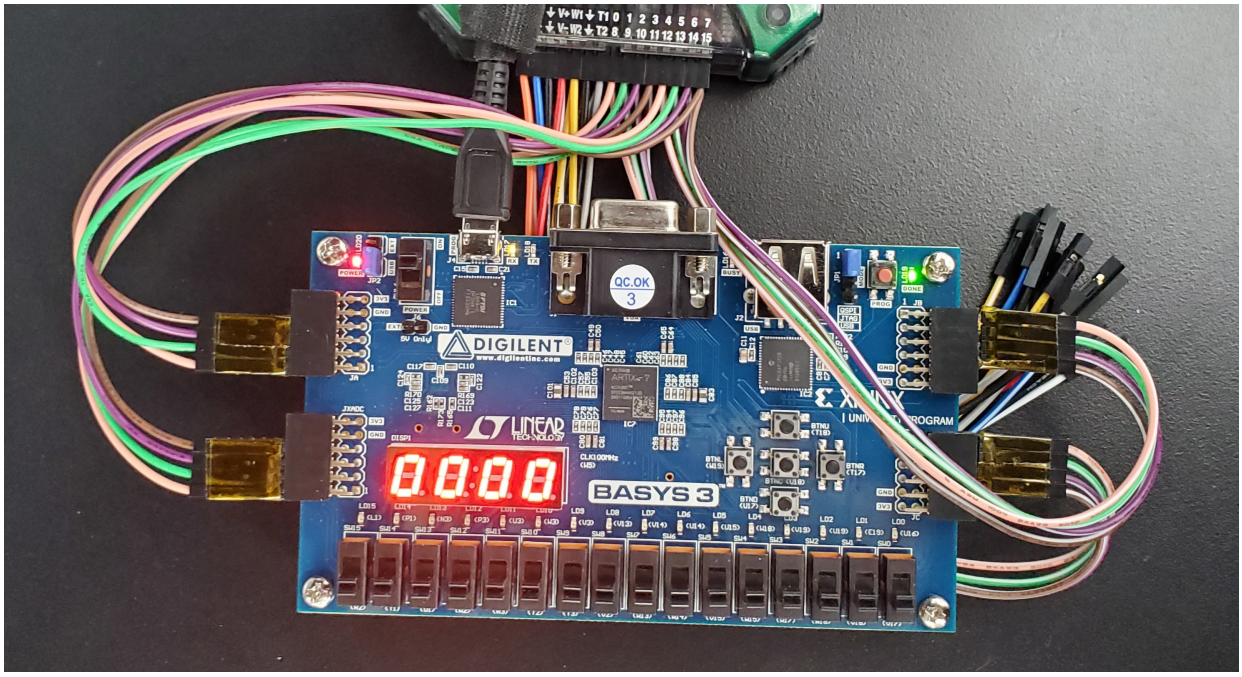
As with all secure computing systems, it is important that only required binaries be installed, and all nonessential functions be disabled. This will reduce the availability of scripting environments, as well as preventing attackers from "living off the land" — a technique wherein an adversary utilizes legitimate system functions in unintended ways in order to carry out an attack.

In addition, a compiler configured with obfuscated architectural parameters should never be installed on a system implementing that same architecture, since this could allow attackers to generate valid obfuscated binaries for use in an attack on the system itself. Beyond this, obfuscation parameters in any form should be protected (including any preconfigured compilers). However, unconfigured or generic compilers should pose no risk to an appropriately obfuscated system, as we describe in Section IV-B.

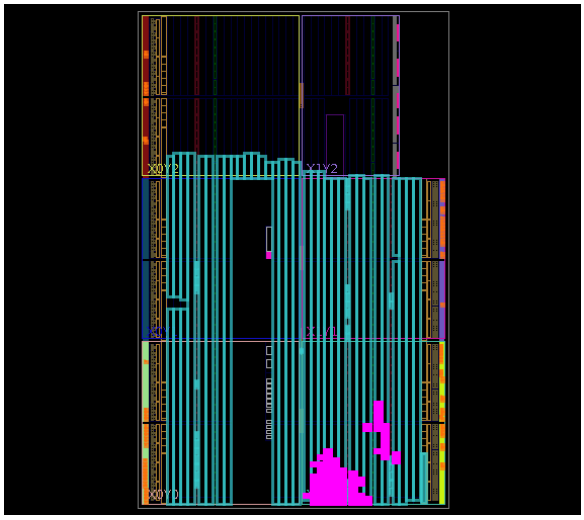
B. Brute-Forcing Obfuscation Offsets

When system synthesis is performed, each core in the obfuscated processor is assigned a unique opcode offset, within a range determined by the details of the instruction set architecture from which the processor is derived. For example, the popular x86-64 ISA (also known as Intel 64, AMD64, or simply x64) utilizes opcodes ranging from one to four bytes, so any given instruction will contain at least one obfuscable opcode byte. This yields a choice of $2^8 = 256$ possible obfuscation offsets for an individual core.

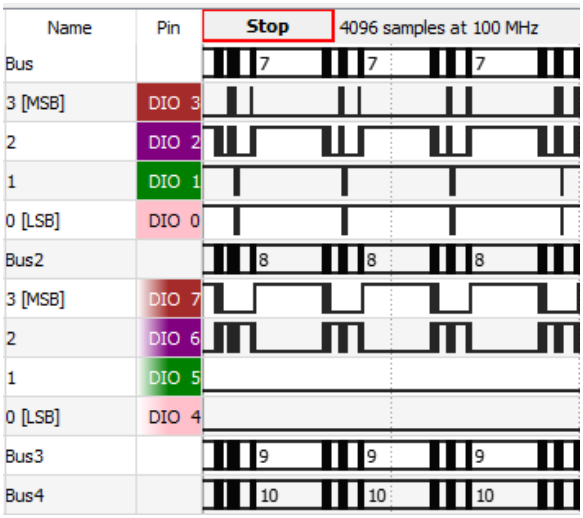
In any security context, when a situation arises in which combinations of parameters are involved, the first consid-



(a) Our prototype system, implemented and running on a Xilinx Artix-7 35T FPGA.



(b) Floorplan for the prototype processor with four obfuscated cores, as generated by Xilinx Vivado. Even as a midrange device, the Artix-7 35T is able to accommodate a basic four-core system.



(c) Obfuscated opcodes being executed in each of the four cores. Each "Bus$N$" signal displays the contents of a core's instruction register.

Fig. 2. Selected examples of prototype system implementation and functionality.

eration should be whether the correct combination can be brute-forced. If too few offset combinations are possible, it might be possible for an attacker with sufficient resources to simply generate a binary executable for each possible case, and attempt to run them in sequence until one succeeds.

We begin by determining the total number of distinct processor configurations available on a given system. Note that since each core loads instructions from its own region of instruction memory, the order in which offsets are mapped to cores does matter (i.e. we care about permutations, not combinations thereof). Assuming an ISA with one obfuscable byte per instruction, we find that generating a four-core processor in which every core has a unique offset yields $\prod_{n=0}^{3}\left(2^8-n\right) \approx 4.195 \times 10^9$ possible system variations.

In order to give a reasonable impression of attack feasibility against such a system, we estimate the compute time required to compile an executable for every possible configuration. Assuming a generation time of one second for each obfuscated binary, we find that 132.9 Gregorian calendar years of compute time would be required to generate every such executable. (In practice, the *average* time required would be half as long, assuming executables are being tested continually so that it is

immediately obvious when success is achieved.)

This is already impractical for an attacker without significant computing resources, but actually executing such a brute-force attack would also require inducing the obfuscated system to run arbitrary binaries — a difficult task on any properly secured system, much less one in which instruction injection is not possible. In addition, uploading each executable file to the targeted system will impose further overhead, and the volume of network traffic generated would increase the likelihood of attack discovery by monitoring systems. In combination, we feel that these obstacles represent a significant challenge to any would-be attacker, providing an extra layer of protection not found in traditional processor architectures.

C. FPGA Vulnerabilities

When introducing new technology into a secure environment, it is prudent to evaluate any new vulnerabilities that might be introduced as a consequence. As discussed in Section II-B, FPGAs are explicitly engineered to be capable of in-place reconfiguration, effectively rewiring themselves to become new digital devices. This flexibility immediately suggests a potential attack vector: if an adversary gains the ability to program the FPGA which hosts an obfuscated processor, they could easily replace it with arbitrary hardware of their own design and synthesis.

In the trivial case, this "injected hardware" might do nothing at all, as a means of carrying out a denial-of-service attack, for instance. However, a more strongly motivated or maliciously inclined adversary might wish to reconfigure the FPGA to act as a "Trojan horse" in the network, carrying out functions of their own choosing.

In the case of a particularly advanced attacker, one might ask whether this assumed programming access could be used to analyze the FPGA's existing configuration and determine the obfuscated core offsets. If so, that information could facilitate the generation of a malicious binary without requiring the effort of brute-forcing all possible system configurations. (As noted in Section IV-B, inducing the execution of an additional binary would still pose a challenge, though perhaps less so for an attacker with direct access to the device's programming interface.) The possible advantage of such an approach lies in its stealth — if the attacker were to reconfigure the system entirely, they would need to replace it with a very similar one of their own design in order to evade detection. If it were possible to simply compromise the original device in a more traditional manner, they might be able to simply add their own programs to the system without disrupting its intended functionality.

Fortunately, this turns out not to be a realistic attack vector, since FPGAs are also designed to be heavily resistant to reverse engineering. This is partially done in order to protect the intellectual property of the FPGA manufacturers themselves, so significant incentives exist for them to make this protection as strong as possible. Thus, even with programming access to the device, the adversary's only option would be to completely reconfigure the system, risking detection in the process.

In addition, reconfiguring an FPGA requires specific physical connections to be made, and thus is best defended against by a strong physical security system, rather than by architectural mechanisms. We also note that, given the impracticality of stealthily compromising such an obfuscated system, an attacker with physical access might achieve more effective results with less effort by simply destroying the device.

V. Related Work

Our effort to counter code injection attacks is far from unique in its goal. Several techniques already exist, and in fact are commonly present in modern computing, to defeat some forms of injection. However, they also operate in very different ways to the method proposed here, and thus present unique comparative strengths and weaknesses, which we now examine.

A. StackGuard

When considering countermeasures to overflow-based code injection attacks, the venerable StackGuard system inevitably comes to mind. StackGuard, proposed at USENIX '98 by Cowan, Pu, Maier, *et al.*, is a simple change to the `gcc` compiler which adds special values to the program stack when a function call is made, and checks their values upon returning from the function [6]. Such a "canary" will be overwritten along with the function's return address as part of a stack-smashing attack, allowing the program's return logic to detect tampering and abort execution. While generally a very effective countermeasure, StackGuard does nothing to protect against overflows that occur in heap memory, and can also be bypassed by more advanced attack methodologies [7].

In addition, StackGuard has a fundamentally different focus than our proposed architecture. It is a simple compiler tweak that facilitates stack tampering detection, whereas our architecture is designed to completely prevent the execution of binaries not built for its specific configuration. StackGuard is far more straightforward to implement, and integrates seamlessly with existing systems, but does not *directly* protect against unauthorized code execution. In comparison, our architecture is more complex to implement, but has the advantage of explicitly preventing the execution of improperly formatted instructions, backed by hardware that accomplishes attack detection within a single clock cycle.

B. Non-Executable Stack

Closer to the realm of hardware-based protection, the concept of a "non-executable stack" has also been widely deployed in modern computer systems. Initially developed by Solar Designer [8], it relies on kernel and hardware mechanisms to detect the attempted execution of instructions from specially marked memory areas. Similarly to our proposed architecture, this non-executable flag (or "NX bit") offers no protection against attacks that "live off the land," assembling code from other sources (especially common libraries such as `libc`) to perform desired actions. Memory regions containing this code will be marked as executable, and such attacks can

thus proceed uninhibited. Other bypass mechanisms that do allow for code injection also exist, commonly focusing on injection into unprotected heap memory [6].

As previously noted, the non-executable stack depends on hardware and kernel support in order to function. This constitutes an important distinction from our architecture, which implements its detection mechanism purely in hardware, and was concieved with embedded systems applications in mind. Embedded processors frequently run compiled binaries directly, without the presence and management authority of a kernel, a convention that renders the NX bit largely inapplicable to the embedded domain.

VI. Conclusions

While the heterogeneous obfuscated processor architecture proposed here will not remove the need to follow other security practices, it has the potential to serve as an additional component in a layered defensive strategy. In this capacity, we believe that it would provide an extremely effective barrier to high-severity memory injection attacks by entirely preventing the execution of injected instructions at the architectural level.

While such systems would most likely still be vulnerable to other attack vectors, particularly those involving interpreted programming languages, there are other known defense mechanisms for these — most obviously, refraining from installing interpreters on the system to begin with. Meanwhile, even for the fairly simple case of an x86-64 instruction set obfuscated across four compute cores, we believe that sufficiently many potential system variations exist to strongly discourage attempts to determine the processor's obfuscation parameters by means of brute force. Lastly, we conclude that no serious risk is posed by the introduction of field-programmable gate array technology into secure computing environments which might benefit from this new architectural concept.

References

[1] S. Özkan. "Vulnerability distribution of CVE security vulnerabilities by types." (May 2, 2010), [Online]. Available: https://cvedetails.com/vulnerabilities-by-types.php (visited on 01/04/2022).

[2] J. H. Lala and F. B. Schneider, "IT monoculture security risks and defenses," *IEEE Security & Privacy*, vol. 7, no. 1, pp. 12–13, 2009. DOI: 10.1109/MSP.2009.11.

[3] The MITRE Corporation. "CWE Category: Memory buffer errors." (Apr. 28, 2022), [Online]. Available: https://cwe.mitre.org/data/definitions/1218.html (visited on 01/04/2022).

[4] J. S. Hane, B. J. LaMeres, T. Kaiser, R. Weber, and T. Buerkle, "Increasing radiation tolerance of field-programmable-gate-array-based computers through redundancy and environmental awareness," *Journal of Aerospace Information Systems*, vol. 11, no. 2, pp. 68–81, 2014. DOI: 10.2514/1.I010106.

[5] B. J. LaMeres, *Introduction to Logic Circuits & Logic Design with VHDL*, 2nd ed. Springer Cham, Mar. 27, 2019, ISBN: 978-3-030-12488-5. DOI: 10.1007/978-3-030-12489-2.

[6] C. Cowan, C. Pu, D. Maier, *et al.*, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," *7th USENIX Security Symposium*, vol. 98, no. 7, pp. 63–78, Jan. 1998.

[7] G. Richarte, "Four different tricks to bypass stackshield and stackguard protection," May 2002.

[8] Solar Designer. "Linux kernel patch from the Openwall Project." (2001), [Online]. Available: https://www.openwall.com/linux (visited on 04/15/2023).