Binary Decompilation And Wavelet Analysis Were Used To Detect Avionics Supply Chain Non-Control-Flow Malware

Sakshi Srivastav[1], Mr. Sambhav Agarwal[2]

[1]M.Tech, Computer Science and Engineering, SR Institute of Management & Technology, Lucknow, India
[2]Associate Professor, Computer Science and Engineering, SR Institute of Management & Technology, Lucknow

---

Abstract - *Non-control-flow Trojans present in embedded systems provide a risk to the data utilized in the decision-making process because of the risk they pose to that data. In order to adjust for the bias in the data, the output data is slanted in such a way that judgments are made either slightly earlier or slightly later than what was originally intended. This is done by keeping the system running continuously and limiting the amount of input data that the system adjusts to a certain geographic area. This approach eliminates the need for the customary testing carried out by a third party. The functional behavior of a binary may be extracted, shown in a waveform, and anomalies—also known as localized behaviors—can be found by using the Ghidra decompile in combination with the discrete wavelet transform. To do this, decode the data using Ghidra and then depict the functional behavior using discrete wavelet transform. This idea is made possible by the fact that the people who created the Ghidra decompile also created the discrete wavelet transform. With the help of Ghidra, one can do a Monte Carlo simulation of phase-shifted Bessel functions of the first type with a Gaussian Trojan that has random magnitude (also known as amplitude), location (also known as mean), and breadth (also known as variance). Ghidra may be used to better understand the behavior of a simple programmed in operation. Ghidra may also be used to finish out a Monte Carlo simulation of second-order phase-shifted Bessel functions using a Gaussian distribution. The discrete wavelet transform may identify anomalies that are situated in a very specific focal area on the map.*

Key Words: Malware, Binary, Wavelet, Avionics Supply, Gaussian Trojan

1. INTRODUCTION

Due to the specialised nature of military avionics applications and missions, they are easy prey for Trojan assaults that bypass control flows. Consider an unmanned aerial vehicle (UAV) equipped with GPS-guided missiles and an autonomous weapon system. The vendor writes and supplies the guided missiles' decision-making software in binary executable form. In order to ensure the safety of the mission, this procedure is activated when the UAV is above the intended target.

However, the organisation was breached before the binary was delivered, and it contains a non-control flow Trojan in the form of a Gaussian function; we'll call this a Gaussian Trojan. The missile decision system incorporates this function by appending it to the received coordinate before sending it on. The value added is zero for almost all GPS coordinates outside of the targets because the function is Gaussian. The GPS guidance is manipulated by this addition, and the payload is fired in the wrong direction. One possible outcome is that the Trojan simply diverts the payload so that it cannot harm anyone else. However, the Trojan may change the target, causing collateral damage. However, please explain the meaning of a Trojan that does not interfere with the control flow. Much of the discussion about anti-malware measures revolves around preventing an interruption in the normal execution of a programme. While a programme is being executed, the stack pointer follows a path known as the control flow. Figure 2 shows the result. Control flow is the process of modifying data that is used to direct execution of an application. Trojans alter the program's execution by rerouting control to malicious scripts or unneeded library code [2]. View Figure 2. The dynamics of flow regulation The system-agnostic nature of Trojans makes it possible for hackers to exploit a wide range of platforms with the same malicious code (i.e. granting the attacker equal or higher privileges than the victim).

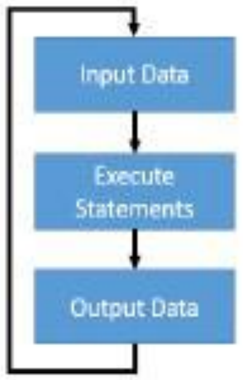


Figure-1: Program control flow diagram of Malware free Diagram.

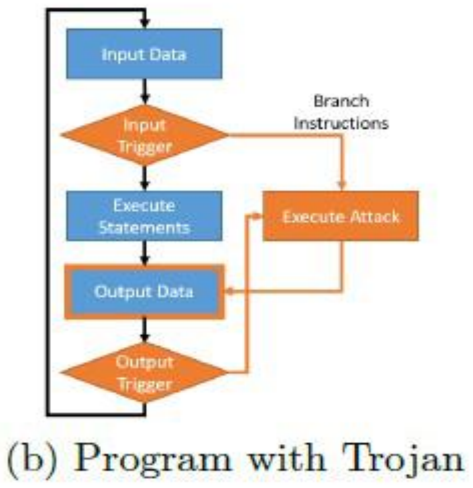


Figure-2: Program control flow diagram of Program with Trojan

1.1. Feature Selection Methods

Below we discuss two statistical techniques to feature selection, principal component analysis (PCA) and logistic regression (LR), that were selected and used due to their good performance in feature selection work for the detection and identification of malware. Through the use of principal component analysis (PCA) and logistic regression techniques, the most emblematic features of network traffic from the CICAndMal2017 dataset were selected.

1.2. Concepts of Security.

The nature of software is established by the kind and extent of the information it contains. One may forecast the input and output data's sequence and time thanks to determinacy. System designers may benefit from three separate resources—confidentiality, integrity, and availability—as they work to build a safe and reliable end result.

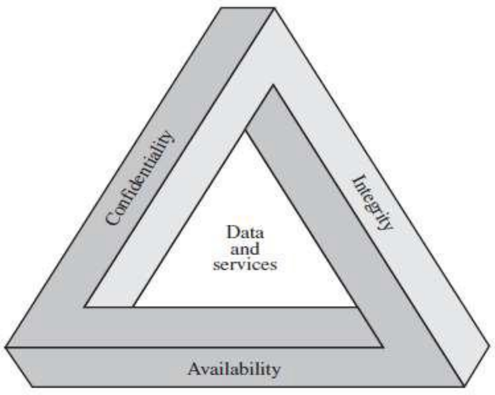


Figure-3: CIA Triad

The above diagram shows the interdependence of three core concepts: privacy, security, and accessibility. This is known as the CIA triad, and it is widely acknowledged as the three most important tenets of information security. Any vulnerability in even one part of the system might allow the whole thing to be compromised.

1.3. Key Terminologies

There is some terminology used in the malware:

1.3.1. Attack Vector.

An attack vector is a technique for gaining unauthorized access inside a computer or network for a criminal purpose by exploiting the system's weaknesses.

1.3.2. Risk.

It is the potential for harm to come about as a result of any danger in the threat landscape exploiting the system for its own ends, including the disclosure of sensitive information like user names and passwords, the loss of proprietary data, and the tarnishing of the company's good name. Damage to or loss of data stored on computer systems is another definition of risk.

1.3.3. Threat.

Anything that can take advantage of vulnerability, either intentionally or unintentionally, to gain access to, harms, or destroy an asset.

1.3.4. Vulnerability.

Different threats can exploit weaknesses or holes in a systems security programme, design policies, and execution to obtain unauthorized access to a computer system or network.

2. METHODOLOGY

Matlab code is located that may be used to run the simulation of the Chapter-01 scenario. In Figures 4 and 5, we see the instant the UAV shoots against the target (the structure outlined in black) from a circling perspective in Figure-4 and a birds-eye perspective in Figure 5. The unmanned aerial vehicle (UAV) is hovering right above the target, and the red GPS coordinates established beforehand indicate that the payload will actually reach the target. Figures 6 and 7 depict the inclusion of a Gaussian Trojan as a supply-chain embedded assault to the guided missiles' GPS readings from comparable perspectives. This allows the system to fire the payload on a fresh target located 400 miles distant after the UAV's GPS reading has already transmitted the fire instruction. By extending this situation by another degree of longitude, the distance would increase to 60 miles (calculated using [7]).

As discussed in Chapter I, localized behavior refers to the Gaussian Trojan's small, localized impact on the sensor output. However, not all regional deviations in behavior are harmful (i.e. the proper coordinates for firing the payload). So, it's impossible to predict the total number of local behaviors detected after accounting for noise, typical behavior, and malicious behavior. More local activities are likely to be considered normal or benign than malevolent. This is discussed in more depth elsewhere, however the phrase "localized behavior" is introduced here and used throughout in the text.

To appreciate the originality of the approach, it is necessary to first comprehend the shortcomings of existing control flow Trojan detection techniques. Coding from Trojans and other malicious programmes is often checked against signature databases of previously detected malware [8]. While control flow Trojan signatures are generally regarded dangerous, non-control flow Trojan signatures are not. As seen above, the output of GPS sensors may have been intentionally tampered with via the introduction of the Gaussian Trojan.

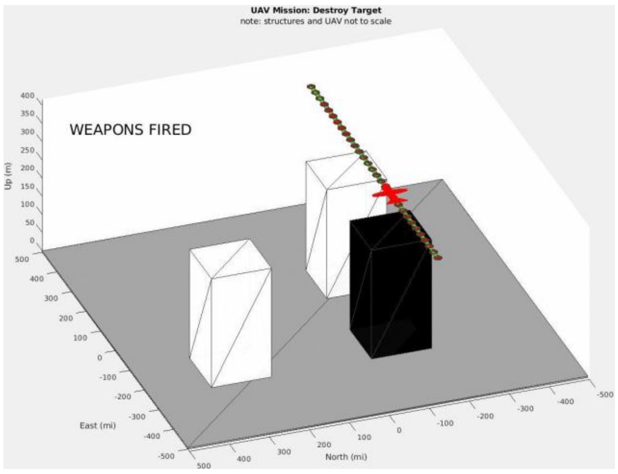


Figure-4: UAV Mission.-Circling view of mission

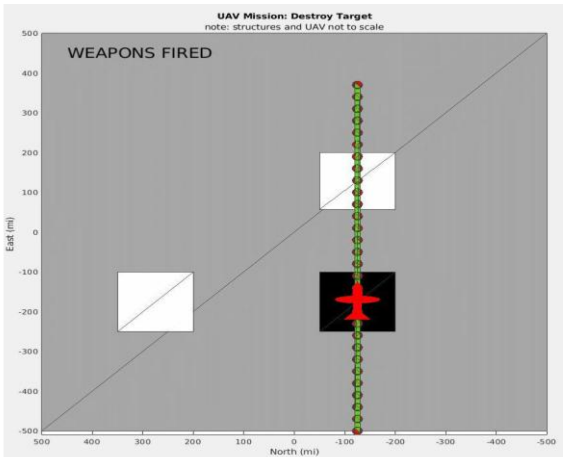


Figure-5: UAV Mission-Birds-eye view of mission

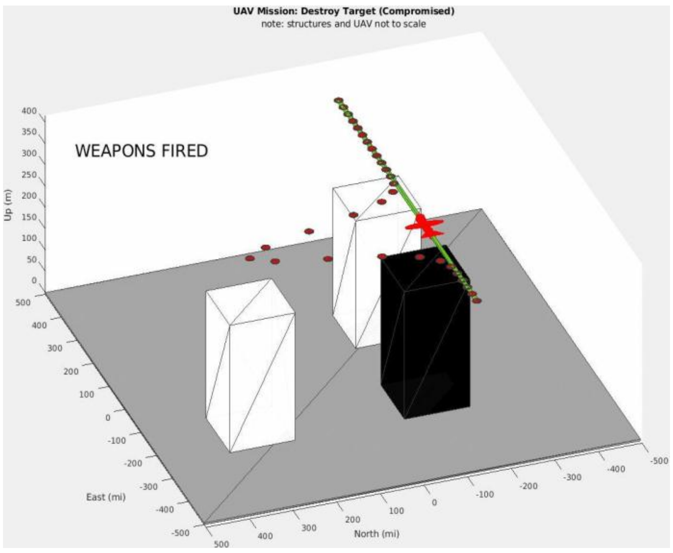


Figure-6: UAV Mission compromised by a Gaussian Trojan. Circling view of Mission

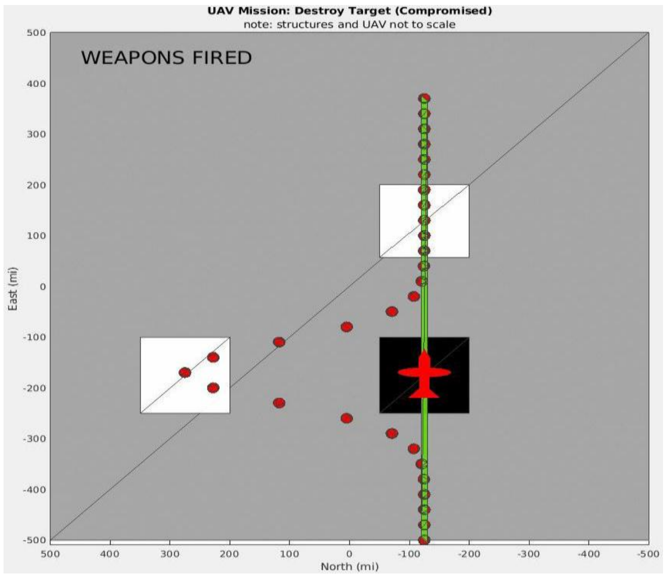


Figure-7: UAV Mission compromised by a Gaussian Trojan. Birds-eye view of Mission

2.1. Modern Decompilers

Investigation and improvement of decompilers is ongoing. Hex-Rays decompile has been the most popular commercial solution for a long time. University researchers create their own decompilers like Phoenix and FoxDec due to the high price tag of commercially available options and the desire to push decompile research ahead. In 2019, the NSA released Ghidra, its decompiler, providing reverse engineers access to a cutting-edge platform without the need for expensive hex-rays or scholarly studies.

2.1.1..Hex-Rays

When it comes to decompilers, Hex-Rays is the gold standard [19, 20]. Named after the firm that created and maintains it, Hex-Rays, the same people behind the gold standard in disassembly, IDA Pro [20]. Hex-Rays has a price tag, but the other decompilers we've covered here are open-source and so free to use and contribute to. Because of this limitation, Hex-Rays could not be used in this investigation.

2.1.2. Phoenix

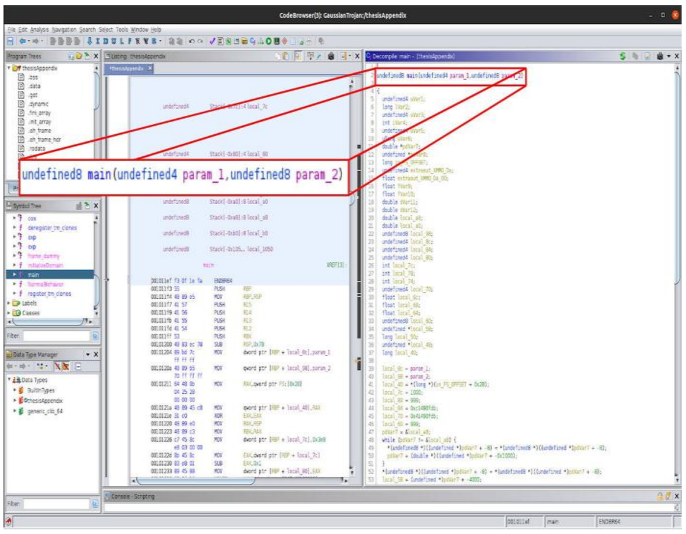
Schwartz et al. [19] introduced the Phoenix decompiler in 2013. When compared to Hex-Rays [19], Phoenix's structural analysis technique, which makes use of iterative refinement and semantics preservation, was shown to recover 28% more controlflow structure. Decompilations with more information retrieved are better at extracting the binary's functional behaviour and the consequences of the Trojan on the programme, but they won't be any better at detecting non-control flow Trojans.

3. RESULTS AND DISCUSSION

3.1. Test Program

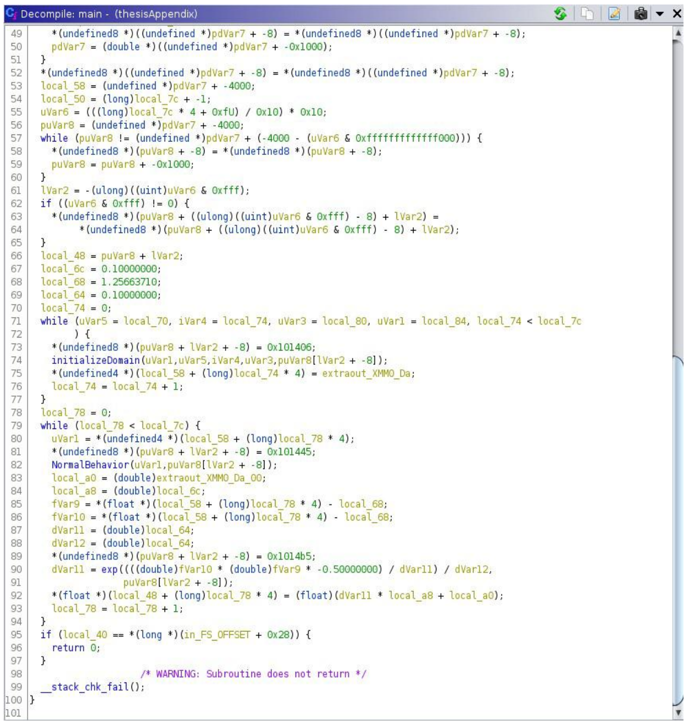
Due to the lack of information, it is assumed that the soft-trigger Trojan is an addition to the standard functioning code. The Gaussian Trojan takes its name from the fact that a Gaussian function with an amplitude of A, a mean of, and a standard deviation of 2 is quite close to 0 everywhere except for the range between 32 and + 32. So, to achieve the always-on state with minimal alterations to other outputs, it is just to add 0 to the standard signal output. This extension creates, by definition, a regionalized pattern in the system output.

Interesting enough, while initialise Domain is called with five parameters in main, the function signature only contains four input arguments. An unsigned long variable is initialised and returned unmodified by the function. The recompiled code is a poor representation of the original algorithm. This is clearly not the intended functional behaviour of initialise Domain, since the primary purpose of a function should never be to initialise an unused variable and return it.



```
undefined8 main(undefined4 param_1,undefined8 param_2)
```

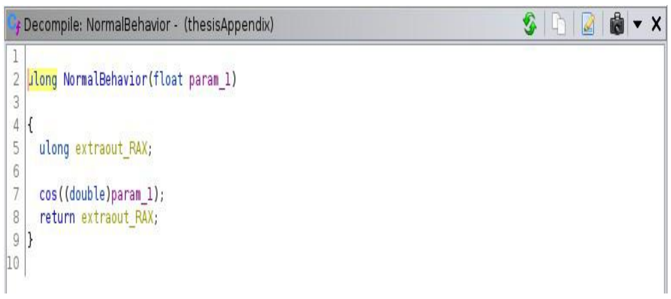
Figure-8: Full window view



```
    *(undefined8 *)((undefined *)pdVar7 + -8) = *(undefined8 *)((undefined *)pdVar7 + -8);
    pdVar7 = (double *)((undefined *)pdVar7 + -0x1000);
  }
  *(undefined8 *)((undefined *)pdVar7 + -8) = *(undefined8 *)((undefined *)pdVar7 + -8);
  local_58 = (undefined *)pdVar7 + -4000;
  local_50 = (long)local_7c + -1;
  uVar6 = (((long)local_7c * 4 + 0xfU) / 0x10) * 0x10;
  puVar8 = (undefined *)pdVar7 + -4000;
  while (puVar8 != (undefined *)pdVar7 + (-4000 - (uVar6 & 0xfffffffffffff000))) {
    *(undefined8 *)(puVar8 + -8) = *(undefined8 *)(puVar8 + -8);
    puVar8 = puVar8 + -0x1000;
  }
  lVar2 = -(ulong)((uint)uVar6 & 0xfff);
  if ((uVar6 & 0xfff) != 0) {
    *(undefined8 *)(puVar8 + ((ulong)((uint)uVar6 & 0xfff) - 8) + lVar2) =
         *(undefined8 *)(puVar8 + ((ulong)((uint)uVar6 & 0xfff) - 8) + lVar2);
  }
  local_48 = puVar8 + lVar2;
  local_6c = 0.10000000;
  local_68 = 1.25663710;
  local_64 = 0.10000000;
  local_74 = 0;
  while (uVar5 = local_70, iVar4 = local_74, uVar3 = local_80, uVar1 = local_84, local_74 < local_7c
        ) {
    *(undefined8 *)(puVar8 + lVar2 + -8) = 0x101406;
    initializeDomain(uVar1,uVar5,iVar4,uVar3,puVar8[lVar2 + -8]);
    *(undefined4 *)(local_58 + (long)local_74 * 4) = extraout_XMM0_Da;
    local_74 = local_74 + 1;
  }
  local_78 = 0;
  while (local_78 < local_7c) {
    uVar1 = *(undefined4 *)(local_58 + (long)local_78 * 4);
    *(undefined8 *)(puVar8 + lVar2 + -8) = 0x101445;
    NormalBehavior(uVar1,puVar8[lVar2 + -8]);
    local_a0 = (double)extraout_XMM0_Da_00;
    local_a8 = (double)local_6c;
    fVar9 = *(float *)(local_58 + (long)local_78 * 4) - local_68;
    fVar10 = *(float *)(local_58 + (long)local_78 * 4) - local_68;
    dVar11 = (double)local_64;
    dVar12 = (double)local_64;
    *(undefined8 *)(puVar8 + lVar2 + -8) = 0x1014b5;
    dVar11 = exp((((double)fVar10 * (double)fVar9 * -0.50000000) / dVar11) / dVar12,
                 puVar8[lVar2 + -8]);
    *(float *)(local_48 + (long)local_78 * 4) = (float)(dVar11 * local_a8 + local_a0);
    local_78 = local_78 + 1;
  }
  if (local_40 == *(long *)(in_FS_OFFSET + 0x28)) {
    return 0;
  }
                    /* WARNING: Subroutine does not return */
  __stack_chk_fail();
}
```

Figure-9: Remaining code in main

In order to do some last arithmetical manipulations on Normal Behavior before the programme exits, it loops back through main as shown before. There's a lot of clutter in the code, and it has to be organised. From the standpoint of the attacker, the Trojan will be concealed (obfuscation, evolved, etc). The analyst will assume that there are no functions with names like Normal Behavior since all of the program's functionality is considered normal. As a result, you shouldn't assume that the analysis will be as easy as this one.



```
ulong NormalBehavior(float param_1)

{
  ulong extraout_RAX;

  cos((double)param_1);
  return extraout_RAX;
}
```

Figure-10: Normal Behavior decompiled function in Ghidra.

4. CONCLUSIONS

Soft-trigger, always-on, non-control-flow Trojans may be detected in their early stages by recompilation and wavelet analysis. Ghidra is more of a reverse engineering tool than a generator of safe, recompilable source code, so keep that in mind if you plan on using it for the latter purpose. We can infer sufficient functionality from the tested basic binary. On the other hand, the static Ghidra decompile may not be enough for a more complicated and, thus, more realistic binary. Cleaning up the code might be difficult and time-consuming if you don't have a firm grasp of the algorithms included in the decompiled binaries.

The DWT is a powerful tool for finding small-scale aberrations in wavelet analysis. In all three of these extreme examples, the Trojan created a strong pulse in high frequency channels that were normally quite flat (or zero) (i.e. Levels 1 & 2). Since pulses do not appear in either the high- or low-frequency channels, it may be concluded that wavelet analysis does not function for more diffuse abnormalities. Because of this, finding them before they pass the third-party test suite becomes more challenging.

Last but not least, it is abundantly clear that non-control-flow Trojans that directly target decisionmaking data in embedded systems pose a significant risk to avionics and other embedded technology. These Trojans might infect everything from a weapon system's navigation algorithm to an autonomous vehicle's proximity sensor, causing fire alarms to go off, unmanned aerial aircraft to stray off course, and the public to be placed in danger. As the world continues to push the boundaries of autonomous systems, non-control-flow Trojans become a greater threat to the safety of innocent people.

REFERENCES

1. S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, “Non-control-data attacks are realistic threats,” Proceedings of the USENIX Security Symposium, pp. 177 – 192, 2005.

2. C. Cifuentes and K. J. Gough, “Decompilation of binary programs.” Software-Practice & Experience, vol. 25, no. 7, pp. 811 – 829, 1995.

3. “Cert-eu latest security advisories,”

4. https://cert.europa.eu/cert/newsletter/en/latest SecurityBulletins .html.

5. “Multiple vulnerabilities in citrix,” https://media.cert.europa.eu/static/ SecurityAdvisories/2021/CERT-EU-SA2021-027.pdf.

6. “Critical vulnerability in vmware vcenter server,” https://media.cert.europa.eu/static/ SecurityAdvisories/2021/CERT-EU-SA2021-025.pdf.

7. M. Tehranipoor and F. Koushanfar, “A survey of hardware trojan taxonomy and detection.” IEEE Design & Test of Computers, Design & Test of Computers, IEEE, IEEE Des. Test. Comput, vol. 27, no. 1, 2010.

8. “Latitude/longitude distance calculator,” https://www.nhc.noaa.gov/gccalc.shtml.

9. M. Sikorski and A. Honig, Practical Malware Analysis. William Pollock, 2012.

10. “Virustotal,” https://www.virustotal.com/gui/.

11. M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Control-flow integrity principles, implementations, and applications,” 12th ACM Conference on Computerand Commu- nications Security, 2009.

12. A. Seshadri, M. Luk, N. Qu, and A. Perrig, “Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses,” 12th ACM Conference on Computer and Communications Security, 2007.

13. W. Xu, D. C. DuVarney, and R. Sekar, “An efficient and backwards-compatible transformation to ensure memory safety of c programs.” ACM SIGSOFT Software Engineering Notes (ACM Digital Library), vol. 29, no. 6, pp. 117 – 126, 2004.

14. D. Dhurjati and V. Adve, “Backwards-compatible array bounds checking for c with very low overhead.” ICSE: International Conference on Software Engineering, pp.162–171, 2006.

15. S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, “Softbound: Highly compatible and complete spatial memory safety for c.” ACM SIGPLAN NOTICES, vol. 44, no. 6, pp. 245 – 258, 2009.

16. ets: Compiler-enforced temporal safety for c.” ACM SIGPLAN NOTICES, vol. 45, no. 8, pp. 31 – 40, 2010.

17. C. Cifuentes, “Reverse compilation techniques,” Ph.D. dissertation, Queensland University of Technology, 1994.

18. M. H. Halstead, Machine-independent computer programming. Spartan Books, 1962.

19. Data Based Advisor. [electronic resource]., ser. Lexis-Nexis Academic, n.d.

20. E. J. Schwartz, J. Lee, M. Woo, and D. Burnley, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring." Proceedings of the 22nd USENIX Security Symposium, 2013.

21. "Hex rays–state-of-the-art binary code analysis tools," https://hex-rays.com/.

22. F. Verbeek, P. Olivier, and B. Ravindran, "Sound c code decompilation for a subset of x86-64 binaries," Proceedings of the 18th International Conference on Software Engineering and Formal Methods, 2020.

23. "Ghidra webpage," https://ghidra-sre.org/.

24. "Ghidra blog," https://ghidra.re/.

25. R. Messier and M. Berninger, Getting Started with Ghidra. [electronic resource]. O'Reilly Media, Inc., 2019.

26. C. Eagle and K. Nance, The Ghidra Book. [electronic resource]. No Starch Press, 2020.

27. D. Sundararajan, Discrete wavelet transform : a signal processing approach. John Wiley & Sons, 2015.

28. M. Golgowski and S. Osowski, "Anomaly detection in ecg using wavelet transformation." 2020 IEEE 21st International Conference on Computational Problems of Elec- trical Engineering (CPEE), pp. 1 – 4, 2020.

29. O. Aydin and M. Kurnaz, "Wavelet-based anomaly detection on digital signals." 2017 25th Signal Processing and Communications Applications Conference (SIU), Signal Processing and Communications Applications Conference (SIU), 2017 25th, pp. 1 – 3, 2017.

30. H. V. Poor, An Introduction to Signal Detection and Estimation. Dowden & Culver, 1994.

31. "Avida by devosoft," https://avida.devosoft.org/.

32. S. Dolan, "mov is turing-complete," Computer Laboratory, University of Cambridge, pp. 1 – 4, 2013.