Practical Protection of Binary Applications via Transparent Immunization

Xinyuan Wang, *Member, IEEE*

***Abstract*—In the past few years, massive data breach attacks on large organizations (e.g., Anthem Inc., Equifax) have compromised sensitive data of tens or even hundreds of millions of people. The 2017 Equifax data breach attack has compromised sensitive data of 148 million people and has costed Equifax $1.4 billion as of May 2019. Unfortunately the average time to detect, contain a data breach was 206 days and 73 days respectively in 2019. There is a pressing need to develop practical and deployable capability to detect and block previously unseen, application specific cyberattacks on vulnerable binary applications in real-time.**

In this paper, we present *AppImmu*, a practical cyber defense system that can detect and block previously unknown cyberattacks on vulnerable binary applications in real-time with no false positive. Given a potentially vulnerable ELF binary application , AppImmu can transparently and statically immunize it into an immunized version via binary rewriting. At run-time, AppImmu uses kernel level immunization based anomaly detection techniques to detect and block previously unknown cyberattacks on immunized binary applications without any prior knowledge of the attacks. We have successfully immunized real world large binary applications such as Apache Java execution environment, bash shell, Snort in Linux and have successfully detected and blocked real world data breach attacks (e.g., Apache Strut exploit used in 2017 Equifax data breach attack, Shellshock exploit) in true real-time. Our benchmark experiments show that AppImmu incurs less than 6% run-time overhead in overall system performance, 2.1% run-time overhead for applications under typical workload.

***Index Terms*—Intrusion Detection and Prevention, Application Immunization, Real-Time Attack Detection.**

I. Introduction

In the past few years, we have witnessed a number of massive data breach attacks [1] on various businesses and organizations (e.g., Target, OPM, JP Morgan Chase, Anthem Inc., Equifax) that have impacted tens or even hundreds of millions of people. Specifically, the 2015 Office of Personnel Management data breach has compromised not only personal data (e.g., SSN) of 21.5 million former and current government employees and contractors but also 5.6 million fingerprints[2]. The 2017 Equifax data breach attack has compromised sensitive data (e.g., SSN) of 148 million people. These massive data breach attacks resulted not only massive identity theft but also prohibitive loss to the business. According to the 2019 Cost of a Data Breach Report[3], the average cost of massive data breach of 50 million records is $388 million. Specifically, the cost of the 2017 Equifax data breach has reached $1.4 billion as of May 2019 [4].

Unfortunately, existing cyber defense mechanisms (e.g., firewall, authentication, IDS (intrusion detection system), IPS (intrusion prevention system), anti-malware) and procedures have been shown to be ineffective in protecting mission critical systems and sensitive data from increasingly sophisticated cyberatacks. Specifically, currently deployed cyber defense lacks the real-time detection and blocking capability against previously unknown, application specific cyberattacks on vulnerably binary applications. Such a deficiency in cyber defense allowed perpetrators to not only infiltrate and compromise mission critical systems, but also keep secret control of the compromised system and exfiltrate critical information for months before being detected and stopped. An independent study sponsored by IBM Security[3] shows that the average time to detect a data breach was 206 days and the average time to contain a data breach was 73 days in 2019.

Real-time detection and blocking of sophisticated cyberattacks are technically challenging as many cyberattacks exploit application specific vulnerabilities in applications. For example, the 2017 Equifax massive data breach attack exploited the Apache Struts vulnerability CVE-2017-5638 [5], [6] inside the Jakarta Multipart parser. It is impossible for any cyber defense system to have the application specific knowledge of all applications. Furthermore, the perpetrators could use polymorphism to disguise their application specific exploits. According to the Symantec Internet Security Threat Report Volume 24 [7], there were 669 millions new malware variants found in 2017 alone. That means on average over 1.83 million new variants appeared every day. Neither signature based nor machine learning based cyber defense approaches can keep up with over one million new malware variants per day!

In this paper, we present *AppImmu*, a practical cyber defense system that can detect and block previously unknown cyberattacks in real-time without any prior knowledge of the cyberattacks. Unlike existing cyber defense systems, AppImmu is built upon a novel combination of static binary instrumentation and run-time kernel level anomaly detection technique. The AppImmu immunization tool can transparently "immunize" potentially vulnerable binary applications such that the AppImmu run-time infrastructure can detect and block many previously unknown, otherwise working exploits on the immunized applications in real-time. Specifically, AppImmu uses dynamically generated, secret mark to tag all invocations of certain type of critical actions (e.g., system call) of the immunized binary applications at run-time. Consequently, all the critical actions invoked by the immunized code are tagged with the correct mark, any critical action invoked without the correct mark is illegal in that it must be from some attack rather than the properly immunized binary application. This enables AppImmu to effectively and efficiently detect and block attacker's illegal invocations of the critical actions with no false positive.

We have successfully immunized real world binary applications (e.g., the Apache Java execution environment, Bash

Xinyuan Wang is with the Department of Computer Science, George Mason University, Fairfax, VA 22030, USA. email: xwangc@gmu.edu

shell), and have empirically validated AppImmu's effectiveness with real world exploits (e.g., CVE-2014-6271, CVE-2017-5638, CVE-2019-17558). AppImmu is able to detect and block those real world exploits on the immunized binary application in real time. To the best of our knowledge, AppImmu is the first cyber defense system that can detect and block the CVE-2017-5638 Apache Struts exploit used in the 2017 Equifax massive data breach attack in true real-time without using any prior knowledge of the exploit. Our benchmark evaluation shows that AppImmu incurs less than 2.1% run-time overhead to applications under typical workload .

In this paper, we make the following contributions:

- **A generic immunization framework for binary applications.** We have developed and implemented the AppImmu tool that can transparently immunize ELF binary executables while keeping the original semantics of the binary applications. Our AppImmu immunization tool can automatically immunize proprietary ELF binary applications that have no symbol information.
- **An efficient and effective kernel level anomaly detection and blocking capability.** We have developed and implemented the kernel level AppImmu infrastructure that works closely with the immunized binary applications at run-time. By using dynamically generated random mark, AppImmu is able to detect and block previously unknown cyberattacks on immunized binary applications in real-time with no false positive.
- **Empirically validated protection of wide range of vulnerable binary applications.** Our empirical evaluation with real world exploits has confirmed that AppImmu is able to protect real world applications (e.g., Bash shell, Snort, Apache server) from previously unknown cyberattacks with no more than 6% run-time overhead.

The rest of this paper is organized as follows. Section II provides functional overview of AppImmu. Section III describes how binary software can be transparently immunized. Section IV evaluates the effectiveness and efficiency of AppImmu. Section V describes related work. Finally, section VI concludes the paper.

II. Functional Overview

A. Threat Model and Assumptions

We assume the to-be-protected binary software: 1) is not self modifying and contains no deliberate obfuscation; 2) can be either open source or commercial off-the-shelf (COTS) binaries with symbol information stripped; 3) may have known or unknown security vulnerabilities. We assume the target system has a clean start in that all the binary applications, libraries and the underlying operating system are benign in that they contain neither malware nor planted backdoor initially. Specifically, the operating system kernel in the target system is assumed trustworthy when it boots up.

When the adversary exploits the vulnerable binary application, he can inject and execute arbitrary code, read/write arbitrary content from/to those memory locations allowed by the vulnerable binary application and the underlying operating system.

These assumptions are reasonable as any mission critical system will be scrutinized to make sure only trusted software will be installed and any granted access will follow the principle of least privilege. Insider threat (e.g., the adversary has been granted privileged access to the target system at the beginning) and supply chain attack are beyond the scope of this work.

B. AppImmu Overview

The primary objective of AppImmu is to protect potentially vulnerable binary applications from various cyberatacks without any prior knowledge of the exploits. Specifically, AppImmu is designed to detect and block cyberattacks on "immunized" binary applications in real-time with virtually no false positive.

Traditional intrusion detection and anti-malware systems are not effective against previously unknown attacks (e.g., zero day attacks) due to their dependency on the incomplete knowledge (e.g., signature) of attacks. In order to detect previously unknown attacks, we build AppImmu upon novel anomaly detection techniques. Specifically, we "immunize" the given binary software such that every invocation of certain critical action (e.g., system call) by the given software binary will be tagged with some unique, dynamically generated random mark at run-time. Since no attacker knows the dynamically generated random mark, no attacker can invoke the critical action (e.g., system call) with the correct mark. This enables AppImmu to effectively and efficiently detect attacker's illegal invocation of the critical action by checking if the critical action has the correct mark at run-time.

If each run-time invocation of the critical action by the immunized software binary has been properly tagged with the correct random mark, then any run-time critical action without the correct mark must be invoked by something else (i.e., malware) rather than the immunized software binary. In other words, whenever AppImmu sees the immunized software binary invokes any critical action without the correct mark at run-time, there must be some attack. Therefore, AppImmu can detect previously unknown attacks on the immunized software binary with no false positive.

AppImmu system consists of two collaborating components: 1) AppImmu software immunization tool; and 2) AppImmu run-time infrastructure as shown in Figure 1 and Figure 2 respectively. The AppImmu software immunization tool transparently immunizes a given (potentially vulnerable) software binary into an immunized and semantically equivalent software binary offline. Any software binary just needs to be immunized once before it can be protected at run-time, and the updated version of software binary needs to be immunized again. The AppImmu software immunization tool is designed to immunize both binary executables and binary libraries.

The AppImmu run-time infrastructure provides run-time protection to the immunized software running upon it by 1) generating a random mark for each process/thread of the immunized software; 2) tagging each invocation of the chosen type of critical actions of each process/thread of the immunized software with the random mark; 3) checking each

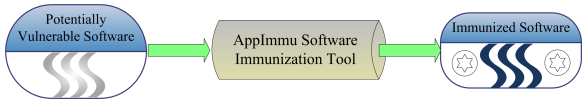


Fig. 1. AppImmu Software Immunization Tool

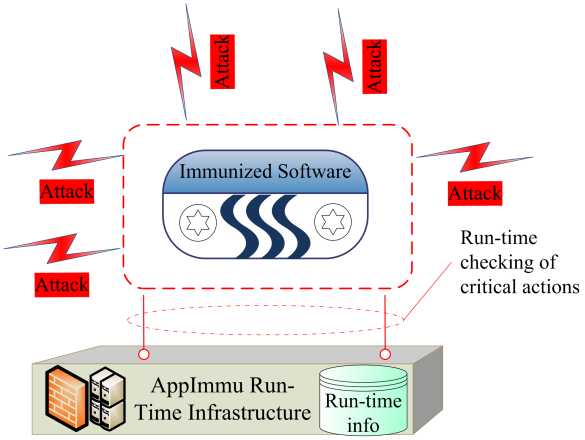


Fig. 2. AppImmu Run-time Infrastructure

invocation of each type of the chosen critical actions of immunized software for the correct mark. Since a random mark is dynamically generated for each process/thread at run-time, the same application will have a different run-time mark each time it is executed. This makes it very difficult for the adversary to learn the exact value of the random mark.

III. Immunizing Software Binaries

In this section, we describe the design and implementation of AppImmu software immunization tool for ELF binaries that include both ELF executables and ELF libraries, and we process ELF executable and ELF shared library separately.

A. Instrumenting ELF Executable

Since an ELF binary may have code and data that are expected to be at specific locations, we want to keep the existing ELF code and data at their original locations as much as possible. Specifically, we choose to place our instrumentation code in a new text section we create, and we only instrument the callsites and the start of the called functions. By keeping functions at their original locations, we avoid complications caused by indirect function calls (e.g., `call *%eax`) where the address of the called function is determined at run-time.

Given a Linux ELF executable, AppImmu software immunization tool transparently injects instrumentation code to the ELF executable such that every system call invoked from the instrumented ELF binary will be tagged with a dynamically generated random mark. Specifically, before the immunized ELF executable starts to run, the AppImmu infrastructure in the Linux kernel generates a random mark X and places it in a specific location of the running stack of the newly created process. Once the immunized ELF executable has obtained the dynamically generated mark X, it passes the random mark X to every function it calls, and each called function further

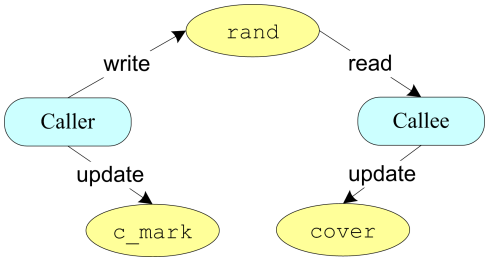


Fig. 3. Coordinated Update of `c_mark` and `cover` by Function Caller and Called Function

passes mark X to every function it calls. When the system call invoking function inside the run-time library is called, it tags the system call invocation with the passed mark X.

Instead of using run-time stack, we choose to use TLS (thread local storage) to pass the random mark X from the caller to the called function. Specifically, we use three TLS variables: `rand`, `cover` and `c_mark` to pass the random mark X from the function caller to the called function where `rand` is a random number generated at run-time, `cover` is the random cover derived from `rand` and `c_mark` is the covered mark: `cover` $\oplus X$. Using such three TLS variables enables us to transparently pass the random mark X to called functions without interfering existing function parameter passing in both 32-bit and 64-bit Linux environments. Since every thread has its own TLS, passing the random mark via TLS will support multiple threading.

1) Passing Random Mark X from the Function Caller to the Called Function: When the instrumented ELF binary starts to execute, it first obtains the random mark X from some specific stack location and overwrites that specific stack location with zero, then generates a random number r and initializes `cover` $\leftarrow r$; `c_mark` $\leftarrow r \oplus X$; `rand` $\leftarrow r$. Therefore, X = `c_mark` $\oplus$ `cover` when the instrumented ELF binary starts to run. This is the invariant we want to keep for every legitimate function call. Such a invariant is achieved by coordinating the function caller and the called function to update `c_mark` and `cover` separately. Figure 3 shows the coordinated update of `c_mark` and `cover` by the function caller and the called function. Specifically, the function caller 1) generates some random number r and updates `c_mark` with `c_mark` $\oplus r$; and 2) saves r to `rand` to be used by the called function. The called function 1) reads `rand` and updates `cover` with `cover` $\oplus$ `rand`; then 2) overwrites `rand` with some random value so that no one else will be able to get the original value of `cover`. After `c_mark` and `cover` have been updated by the function caller and the called function respectively, the invariant X = `c_mark` $\oplus$ `cover` holds. Eventually the system call invoking functions inside the run-time library can obtain the hidden random mark X via `c_mark` $\oplus$ `cover`, and it can tag the system call invocation by using the unused part of %rax or %eax.

Letting the function caller and the called function to update `c_mark` and `cover` respectively creates some desirable bond between the instrumented caller and the instrumented callee. In case some attack code calls some instrumented function without updating `c_mark` and `rand` as expected, the instru-

mented callee will update `cover` with some random value in `rand`. This will make `c_mark` $\oplus$ `cover` $\neq X$. Similarly, if the instrumented function somehow calls uninstrumented code (e.g., attack code) which does not update `cover` as expected, it will make `c_mark` $\oplus$ `cover` $\neq X$. Both cases will eventually result some system call or other critical action tagged with the wrong mark, which will be immediately detected by the AppImmu infrastructure. Therefore, our random mark passing scheme makes it difficult for the attacker to reuse instrumented code or execute attack code.

2) Immunizing ELF Binary Executable: Give an ELF binary executable, AppImmu software immunization tool first uses the compiler-agnostic function detection method [8] to identify the function callsites (i.e., location of legitimate function call instruction) and the starting address of functions. It then instruments the given ELF binary as illustrated by Table I.

AppImmu software immunization tool replaces the 5-byte instruction `call bar` at callsite_i inside function `foo` with the 5-byte unconditional jump instruction `jmp foo_callStub`. The call stub function `foo_callStub` will jump back to the next instruction at postcall_i inside function `foo` after updating `rand` and `c_mark`. In addition, AppImmu software immunization tool replaces the first few instructions of function `bar` with a 5-byte unconditional jump instruction `jmp bar_startStub`. The function start stub `bar_startStub` will 1) update `cover` with `rand` and overwrite `rand` with some random value; 2) execute the first few instructions that were at the beginning of function `bar` whose space was taken by the 5-byte jump instruction `jmp bar_startStub`; and 3) jump back to location `poststart_bar`.

Note, Table I only shows the handling of the most common cases of function calls in ELF binary. Instrumentation of short function calls (e.g., call %eax) requires moving extra instructions adjacent to the callsite to `foo_callStub` in order to make room for the 5-byte unconditional jump at `callsite_`i.

TABLE I
INSTRUMENTING FUNCTION CALLS AND CALLED FUNCTIONS

Original Binary Code	
foo: . . . callsite_i: call bar postcall_i: . . .	bar: push %rbp mov %rsp, %rbp bar_insns poststart_bar: . . .
Instrumented Binary Code	
foo: . . . callsite_i: jmp foo_callStub postcall_i: . . .	bar: jmp bar_startStub poststart_bar: . . .
Injected Instrumentation Code	
foo_callStub: rand $\leftarrow$ random number c_mark $\leftarrow$ c_mark $\oplus$ rand call bar jmp postcall_i	bar_startStub: cover $\leftarrow$ cover $\oplus$ rand rand $\leftarrow$ random number push %rbp mov %rsp, %rbp bar_insns jmp poststart_bar

IV. EMPIRICAL EVALUATIONS

We have empirically evaluated AppImmu with real world applications, real world exploits and three popular benchmark suites on one old machine with a quad-core 2.3 GHz Intel Core i7 CPU, 4GB RAM running 32-bit kernel 2.6.32.20 and one new machine with a quad-core 1.7 $\sim$ 4.4 GHz Intel Core i5-10310U CPU, 16GB RAM running 64-bit kernel 5.10.

A. Attack Detection and Prevention

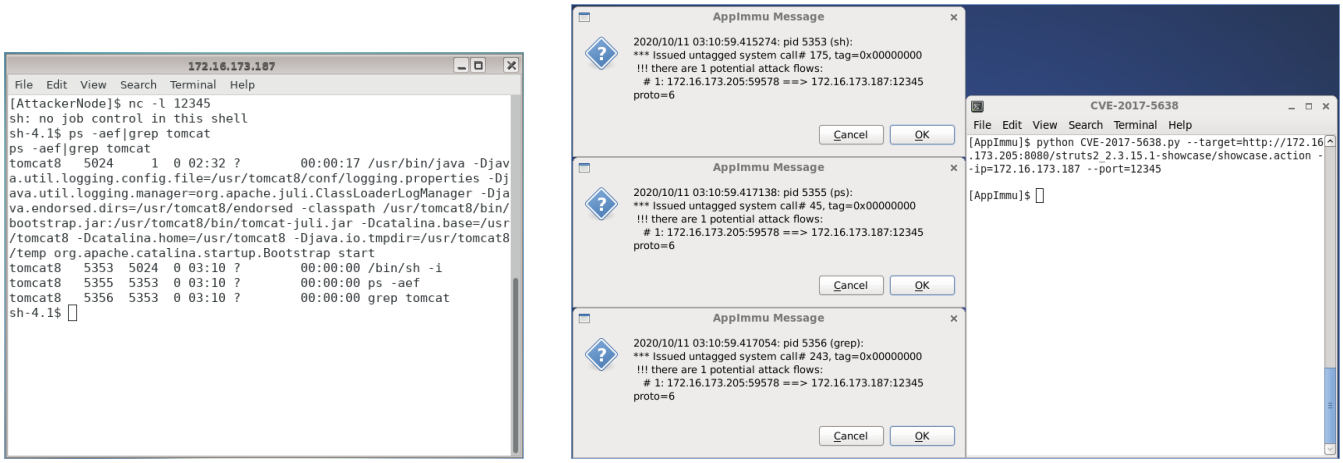
To evaluate the effectiveness of AppImmu in detecting and blocking previously unknown attacks, we have chosen the following real world exploits on some of the most popular applications:

1) Exploit of the CVE-2017-5638 vulnerability in Apache Struts – an open source web application framework for developing Java EE web applications. This is the exact exploit used in the 2017 Equifax massive data breach that has compromised sensitive data of 148 million people and has costed Equifax $1.4 billion as of May 2019. According to CISA's Top 10 Routinely Exploited Vulnerabilities [9], the Apache Struts vulnerability (CVE-2017-5638) is the third most exploited vulnerability during 2016–2019.
2) Exploit of the Shellshock (CVE-2014-6271) vulnerability in Apache server with CGI using Bash shell. Bash is the most popular shell in Linux and other Unix-like operating systems.
3) Exploit of Snort 2.6.1 DCE/RPC Preprocessor Remote Buffer Overflow that allows remote attackers to gain remote shell. Snort is the top free network intrusion detection system (NIDS) according to [10].

In our experiments, we have immunized the Java 8 execution environment which itself is an ELF executable, the Bash shell and the Snort executable. Note, even though the applications used in the experiments are open source, we treat them as binary applications and have stripped the symbol information from the ELF executables before we immunize the binary executables. AppImmu infrastructure is able to detect and block each of the above four exploits against corresponding immunized applications in real-time

To the best of our knowledge, there is no publicly known existing method that can detect the exploit of the Apache Struts vulnerability CVE-2017-5638 in real-time. Our AppImmu infrastructure can not only detect but also block the Apache Struts CVE-2017-5638 exploit in true real-time. In addition, it can automatically identify the TCP network flow used by the attacker. Such network flow actually points to the machine used by the attacker to gain the reverse shell!

We have experimented with an advanced exploit of the Apache Struts vulnerability CVE-2017-5638 that gives attacker a remoted shell at the machine of specified IP address. Figure 4(a) shows the reverse shell at machine of IP address 172.16.173.187. It also shows the result of command "ps –aef—grep tomcat" in the established reverse shell. Process 5353 is the reverse shell launched by the exploit, process 5355 executes command "ps –aef" and process 5356 executes



(a) Reverse Shell Established by the CVE-2017-5638 Exploit (b) Real-Time Detection of the CVE-2017-5638 Exploit with Attack Flow Information

Fig. 4. Real-Time Detection of Exploit of Apache Struts CVE-2017-5638 Vulnerability

commands "grep tomcat".

Figure 4(b) shows the real-time detection of the exploit of Apache Struts CVE-2017-5638 vulnerability. The 3 left AppImmu message boxes show that AppImmu infrastruture has identified not only those offending processes 5353, 5355 and 5356 that triggered illegal (untagged) system call, but also the TCP network flow used by the reverse shell. It clearly shows that the attacker used machine of IP address 172.16.173.187 to gain the remote shell. All such attack information have been collected the same time when the attack has been detected in real-time.

During the experiments with these real world exploits (CVE-2006-5276, CVE-2014-6271, CVE-2017-5638, CVE-2019-17558), AppImmu has never used any prior knowledge of these exploits to detect and block them in real-time. We expect AppImmu to be able to detect and block many other real world exploits – including previously unknown exploits in real-time.

B. Runtime Performance Overheads

To measure run-time performance overhead of AppImmu, we have used the original 64-bit kernel 5.10, the original GNU libraries 2.17. To measure the run-time performance overhead of AppImmu over the whole operating system, we have used the instrumented kernels loaded with the AppImmu infrastructure, and instrumented GNU library with AppImmu run-time checking all turned on.

To understand how applications would perform in the AppImmu environment under typical workload, we have used the latest SPEC CPU 2017 macro benchmarks to measure the performance overhead of the 64-bit AppImmu infrastructure with kernel 5.10. Table II shows the SPEC CPU 2017 benchmark measurements of 4 copies and 2 iterations. The lowest overhead is -4.1667% and the highest overhead is 6.0606%. The overall SPEC CPU 2017 integer rate overhead is 2.0548%. This indicates that the AppImmu infrastructure incurs very low run-time performance overhead for applications under typical real world workload.

TABLE II
APPIMMU RUN-TIME OVERHEAD BY SPEC CPU 2017 BENCHMARKS

SPEC CPU 2017	AppImmu	Baseline	Overhead
500.perlbench_r	17.0	17.80	4.4944%
502.gcc_r	17.0	17.0	0.0000%
505.mcf_r	12.70	12.40	-2.4194%
520.omnetpp_r	7.10	7.00	-1.4286%
523.xalancbmk_r	10.30	10.40	0.9615%
525.x264_r	32.40	34.40	5.8140%
531.deepsjeng_r	15.00	14.00	-4.1667%
541.leela_r	12.40	13.20	6.0606%
548.exchange2_r	24.00	25.10	4.3825%
557.xz_r	9.43	9.61	1.8730%
SPECrate 2017 int	**14.30**	**14.60**	**2.0548%**

C. Security Analysis

The security of AppImmu is based on the secrecy of the random mark dynamically generated at run-time. To increase the randomness of their locations in TLS, we can divide the multi-byte TLS variables `c_mark, rand, cover` into multiple parts of single byte and randomly allocating the single byte parts in TLS storage of $n \geq 20$ bytes. The attacker can not obtain the random mark unless he has figured out the exact location of each byte of those three TLS variables and has read them in the exact order. Since allocating 12 bytes from $n \geq 20$ bytes TLS storage has $\frac{n!}{(n-12)!} \geq \frac{20!}{8!} > 6 \times 10^{13}$ permutations, the chance for the attacker to happen to figure out the values of `c_mark, rand, cover` from the unknown locations within the 20 bytes TLS storage is exceedingly low ($\approx 1.7 \times 10^{-14}$).

By using dynamically generated random mark to tag every legitimate invocation of critical actions, AppImmu is able to detect and block previously unknown cyberattacks in real-time with no false positive. Since the random mark is unknown to the adversary, the initiator of sophisticated code reuse attacks such as ROP [11], JOP [12], COOP [13] and Control Jujutsu [14] can not provide the correct mark when reusing the code. This allows AppImmu to detect these code reuse attacks in real-time.

However, AppImmu may miss detecting certain cyberat-

tacks that do not trigger illegal critical actions. For example, AppImmu will not automatically detect the Heartbleed exploit on the vulnerable OpenSSL library even if the vulnerable OpenSSL library is immunized by AppImmu. In addition, AppImmu is not effective against supply chain attack such as the 2020 SolarWinds hack [15].

V. Related Works

Randomization based defense The idea of randomization has been widely used in many proposed cyber defense approaches [16], [17], [18], [19], [20], [21], [22], [23]. By randomizing the memory address of important run-time entities (e.g., stack, heap, pointer), address space randomization [16], [19], [20], [23] would cause the attack code to access the wrong address at run-time. Similarly, Instruction set randomization [17], [18] causes the attack code to use the wrong instructions at run-time. System call randomization [21] randomizes the system call number such that the attack code would use the wrong system calls at run-time. Existing randomization based defense approaches 1) tend to make debugging and diagnostic tasks more difficult as acknowledged by [21]; 2) would always crash the exploited applications; 3) can not detect or block exploits (e.g., CVE-2017-5638) on interpreted (e.g., Java) applications such as Apache Struts. In contrast, AppImmu does not impact debugging and diagnostics, and it can be configured to allow the exploited application to continue to run so that we can observe the actions of the exploit. In addition, AppImmu can detect and block exploit on interpreted Java applications.

System call based defense Forrest et al. [24] first applied the idea of immunology to building anomaly detection capability based on short sequence of system calls. A number of followup works [25], [26], [27] have improved the effectiveness of the system call sequence based anomaly detection. Instead of using sequence of system calls, Wang et al. [28] used system call tagging to eliminate run-time training required by previous system call sequence based anomaly detection systems [24], [25], [26], [27]. However, their approach requires the access of source code and it can not immunize binary applications. Sysfilter [29] limits the set of system calls an application can legitimately invoke at run-time thus it prevents attacks from invoking any system call outside the automatically constructed set of legitimate and application dependent system calls.

Binary instrumentation and rewriting

Since we only want to immunize our binary application statically, AppImmu does not use dynamical binary instrumentation/rewriting techniques [30], [31]. Static binary instrumentation [32], [33], [34], [35] and binary rewriting [36], [37], [38] statically instrument a given binary with desired security functionality at the binary code level. AppImmu combines its static binary instrumentation/rewriting with its kernel level anomaly detection in order to immunize and protect interpreted binary applications.

Control flow integrity (CFI) Control flow integrity (CFI) [39], [22], [40], [41], [42], [43], [44] seeks to prevent, detect control flow hijacking and code-reuse attacks by ensuring the indirect control flow transfers can only reach legitimate destinations in the control flow graph (CFG) at run-time.

Under the condition of fully-precise static CFI – the most restrictive stateless CFI policy, control-flow bending [45] has illustrated the fundamental challenges in using CFI to protect stripped binary executables from exploits. It shows that real world applications could be compromised by exploits without leaving CFG. For example, the Apache Struts exploit (CVE-2017-5638) used in the 2017 Equifax data brech attack does not violate any CFI of the Apache Struts while it tricks the flawed Jakarta Multipart parser to interpret the crafted HTTP message content as an Object Graph Navigation Library (OGNL) expression to launch arbitrary external program (e.g., shell /bin/sh). To the best of our knowledge, no existing cyber defense mechanism has been shown to be able to detect and block the Apache Struts CVE-2017-5638 exploit in real-time.

Unlike existing CFI based cyber defense, AppImmu does not try to ensure every indirect control flow transfer is legitimate, but focuses on detecting and blocking illegal critical actions (e.g., system call, program launching) based on checking if critical actions have the correct secret mark. This not only makes AppImm more lightweight, but also enables AppImmu to catch illegal control flow transfer at the interpreted language level, and those unintended but legitimate control flow transfers that are allowed by the program bug (e.g., CVE-2017-5638).

VI. Conclusions

In this paper, we have presented the AppImmu cyber defense system that can transparently immunize and protect many binary applications, and have demonstrated that (1) it is feasible to detect and block previously unknown cyberattacks in real-time with no false positive; and (2) transparent immunization is a promising and practical approach to protect vulnerable real world binary applications.

Based on a novel combination of binary rewriting and anomaly detection techniques, AppImmu can transparently immunize and protect real world binary applications from real world exploits. Our experiments show that AppImmu can effectively detect and block the Shellshock (CVE-2014-6271) exploit and the Apache Struts (CVE-2017-5638) exploit used in the 2017 Equifax massive data breach attack in real-time. At run-time, AppImmu imposes less than 2.1% performance overhead to applications under typical workload.

References

[1] D. Swinhoe, "The 15 biggest data breaches of the 21st century," April 2020, https://www.csoonline.com/article/2130877/the-biggest-data-breaches-of-the-21st-century.html.

[2] C. R. Center, "Cybersecurity Incidents," 2015, https://www.opm.gov/cybersecurity/cybersecurity-incidents/.

[3] I. Security, "Cost of a Data Breach Report 2019," 2019, https://www.all-about-security.de/fileadmin/micropages/Fachartikel_28/2019_Cost_of_a_Data_Breach_Report_final.pdf.

[4] M. J. Schwartz, "Equifax's Data Breach Costs Hit $1.4 Billion," May 2019, https://www.bankinfosecurity.com/equifaxs-data-breach-costs-hit-14-billion-a-12473.

[5] N. V. Database, "CVE-2017-5638 Detail," March 2017, https://nvd.nist.gov/vuln/detail/CVE-2017-5638.

[6] S. Mort, "CVE-2017-5638: The Apache Struts vulnerability explained," September 2017, https://www.synopsys.com/blogs/software-security/cve-2017-5638-apache-struts-vulnerability-explained/.

[7] Symantec, "Internet Security Threat Report Volume 24," February 2019, https://www.symantec.com/content/dam/symantec/docs/reports/istr-24-2019-en.pdf.

[8] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-Agnostic Function Detection in Binarie," in *Proceedings of 2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, April 2017.

[9] N. C. A. System, "Top 10 Routinely Exploited Vulnerabilities," May 2020, https://us-cert.cisa.gov/ncas/alerts/aa20-133a.

[10] U. Team, "Top 6 Free Network Intrusion Detection Systems (NIDS) Software in 2020," August 2020, https://www.upguard.com/blog/top-free-network-based-intrusion-detection-systems-ids-for-the-enterprise.

[11] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)*. ACM, October 2007.

[12] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-Oriented Programming: A New Class of Code-Reuse Attack," in *Proceedings of the the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2011)*. ACM, March 2011, pp. 30–40.

[13] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P 2015)*. IEEE, May 2015.

[14] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS 2015)*. ACM, October 2015, pp. 901–913.

[15] C. Timberg and E. Nakashima, "The U.S. government spent billions on a system for detecting hacks. The Russians outsmarted it," February 2021, https://www.seattletimes.com/nation-world/the-u-s-government-spent-billions-on-a-system-for-detecting-hacks-the-russians-outsmarted-it/.

[16] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits," in *Proceedings of the 12th USENIX Security Symposium*, Auguest 2003.

[17] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering Code-Injection Attacks with Instruction-Set Randomization," in *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS 2003)*. ACM, October 2003, pp. 272–280.

[18] E. Barrantes, D. Ackley, S. Forrest, T. Palmer, D. Stefanovic, and D. Zovi, "Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks," in *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS 2003)*. ACM, October 2003, pp. 281–289.

[19] J. Xu, "Intrusion Prevention Using Control Data Randomization," in *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN 2003)*. IEEE, June 2003.

[20] Z. K. Jun Xu and R. K. Iyer, "Transparent Runtime Randomization for Security," in *Proceedings of the 22nd Symposium on Reliable and Distributed Systems (SRDS 2003)*. IEEE, October 2003.

[21] X. Jiang, H. J. Wang, D. Xu, and Y.-M. Wang, "RandSys: Thwarting Code Injection Attacks with System Service Interface Randomization," in *Proceedings of the 26th Symposium on Reliable and Distributed Systems (SRDS 2007)*. IEEE, October 2007.

[22] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. A. Mccamant, D. Song, and W. Zou, "Practical Control Flow Integrity & Randomization for Binary Executables," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy (S&P 2013)*. IEEE, May 2013.

[23] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. D. Sutter, and M. Franz, "It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS 2015)*. ACM, October 2015, pp. 243–255.

[24] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A Sense of Self for Unix Processes," in *Proceedings of the 1996 IEEE Symposium on Security and Privacy (S&P 1996)*. IEEE, May 1996.

[25] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy (S&P 1999)*. IEEE, May 1999, pp. 133–145.

[26] R. Sekar, M. Bendre, and P. Bollineni, "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors," in *Proceedings of the 2001 IEEE Symposium on Security and Privacy (S&P 2001)*. IEEE, May 2001.

[27] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly Detection Using Call Stack Information," in *Proceedings of the 2003 IEEE Symposium on Security and Privacy (S&P 2003)*. IEEE, May 2003.

[28] X. Wang and X. Jiang, "Artificial Malware Immunization based on Dynamically Assigned Sense of Self," in *Proceedings of the 13th Information Security Conference (ISC 2010)*, October 2010.

[29] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis, "Sysfilter: Automated System Call Filtering for Commodity Software," in *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, September 2020, pp. 459–474.

[30] C. keung Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa, and R. K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*. Chicago, Illinois: ACM Press, June 2005.

[31] N. Nethercote and J. Seward, "Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2007)*. San Diego, California: ACM Press, 2007.

[32] B. Buck and J. K. Hollingsworth, "An API for Runtime Code Patchin," *International Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 317–329, November 2000.

[33] N. Aaraj, A. Raghunathan, and N. K. Jha, "Dynamic Binary Instrumentation-Based Framework for Malware Defense," in *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2008)*, July 2008, pp. 64–87.

[34] P. Saxena, R. Sekar, and V. Puranik, "Efficient Fine-Grained Binary Instrumentation with Applications to Taint-Tracking," in *Proceedings of the 2008 International Symposium on Code Generation and Optimization (CGO 2008)*, April 2008.

[35] M. Zhang, Q. Qiao, N. Hasabnis, and R. Sekar, "A Platform for Secure Static Binary Instrumentation," in *Proceedings of the 10th ACM International Conference on Virtual Execution Environments (VEE 20014)*, July 2014, pp. 129–140.

[36] B. Schwarz, S. Debray, G. Andrews, and M. Legendre, "PLTO: A Link-Time Optimizer for the Intel IA-32 Architecture," in *In Proceedings of the 3rd Workshop on Binary Rewriting (WBT 2001)*, 2001.

[37] M. Prasad and T. cker Chiueh, "A Binary Rewriting Defense against Stack Based Overflow attacks," in *Proceedings of the 2003 USENIX Annual Technical Conference (ATC 2003)*, June 2003.

[38] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Securing Untrusted Code via Compiler-Agnostic Binary Rewriting," in *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC 2012)*, 2012, pp. 299–308.

[39] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity: Principles, Implementations, and Applications," in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS 2005)*. ACM, November 2005, pp. 340–353.

[40] M. Zhang and R. Sekar, "Control Flow Integrity for COTS Binaries," in *Proceedings of the 22nd USENIX Security Symposium*. USENIX, August 2013.

[41] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque Control-Flow Integrity," in *Proceedings of the 22th Network and Distributed System Security Symposium (NDSS 2015)*, February 2015.

[42] V. van der Veen, D. Andriesse, E. Göktas, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical Context-Sensitive CFI," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS 2015)*. ACM, October 2015, pp. 927–940.

[43] V. van der Veen, E. Göktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level," in *Proceedings of the 2016 IEEE Symposium on Security and Privacy (S&P 2016)*. IEEE, May 2016.

[44] P. Muntean, M. Fischer, G. Tan, Z. Lin, and C. E. Jens Grossklags, " τCFI: Type-Assisted Control Flow Integrity for x86-64 Binaries," in *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID 2018)*, September 2018.

[45] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity," in *Proceedings of the 24th USENIX Security Symposium*. USENIX, August 2015.