Multidisciplinary Journal of Technology
Volume 11 Issue 4, October-December 2023
ISSN: 2995-441X
Impact Factor: 7.73
https://kloverjournals.org/journals/index.php/Tech

INTRICACIES OF SAFETY: A MULTI-VARIANT ANALYSIS WITH K-VARIANT ARCHITECTURE FOR WEB SERVICE AND APPLICATION SECURITY ENHANCEMENT

Christopher Michael Adams
Department of Computer Science, Illinois Institute of Technology

Abstract
The K-Variant Architecture is proposed as a cost-effective approach for enhancing the security of web services and applications against memory exploitation attacks. Memory-related vulnerabilities continue to be a major concern, even for web services implemented in memory-safe languages. To address this, the K-Variant Architecture uses source code-level program transformations to generate variants, providing statistical security against memory exploitation attacks through critical data diversification in memory. Unlike the N-version architecture, which is limited to mission and safety-critical systems due to its high cost and difficulties in verifying versions, the K-Variant Architecture offers a low-cost alternative with diversity in critical data to improve system security against memory exploitation attacks.This paper presents the high-level design of the K-Variant Architecture, program transformation techniques, and implementation details for web services and applications. The proposed K-Variant Architecture is demonstrated as an object-oriented design utilizing three classes, namely Client, ServiceDirectory, and EngineMotor, to provide critical data diversification in memory for web services. The program transformation techniques used in the K-Variant Architecture and their suitability for web services are also discussed. The overall architecture's implementation details are provided, including the use of various program transformations, the deployment of variants on different operating systems, and the compilation of variants depending on the programming language.In conclusion, the K-Variant Architecture is proposed as a cost-effective approach for improving the security of web services and applications against memory exploitation attacks. The proposed architecture provides critical data diversification in memory, improving the system's survivability against memory exploitation attacks. The use of safe and automated program transformations in the generation of variants makes system development cost-effective.

Keywords: *K-Variant Architecture, program transformations, memory exploitation attacks, critical data diversification, web services, object-oriented design.*

INTRODUCTION

With the proliferation of e-commerce platforms, businesses must develop more reliable and secure systems to instill confidence in their consumers. Unreliable and insecure systems can result in the loss of a large number of current and prospective customers. Additionally, businesses' reputations may suffer due to unreliable and insecure web services. Additionally, businesses and organizations may incur legal and financial liabilities as a result of service disruptions or failures. For these reasons, more secure web services and applications are required.

Most web services and applications are written in memory-safe languages such as Java and C#. Therefore, the risk of memory exploitation attacks on those systems is low. However, highperformance websites and applications still use machine-compiled code like C++ and C to make services or applications fast to start up and execute. Memory-unsafe languages can improve the performance of web services and applications. On the other hand, they can be exposed to

memory exploitation attacks. Exploiting a buffer overflow vulnerability may allow adversaries to corrupt web services, expose confidential information, or execute malicious code.

Buffer overflow vulnerability attacks may still be possible even if a web service or application is coded in a safe memory language because the interpreter and libraries can be written in unsafe languages. For example, PHP is a scripting language that is not itself affected by memory exploitation attacks. However, the PHP interpreter is written in the memory-unsafe C programming language. Therefore, systems can be affected by memory-related attacks [1]. Another example is a memory-safe Java application that uses a compression library written in the memory-unsafe C programming language. The library can allow overwriting of the Java executable file by exploiting a buffer overflow vulnerability. These examples show that services and applications written in a memory-safe language can be exposed to memory exploitation attacks. Therefore, memory-related vulnerabilities should be considered in web services and applications requiring high security.

The CVE Details vulnerability database [2] shows that reported memory-related vulnerabilities have increased recently. Newly discovered memory-related vulnerabilities in popular web servers allow attackers to corrupt services and override existing files and executables. Although developers and communities provide patches for new vulnerabilities, many unreported vulnerabilities still exist and are exploitable by attackers. Therefore, architecture-level security may be required for web services and applications that require high security.

Fault tolerance architecture is one of the methods to improve the reliability and security of software systems through redundancy. Diversity in design, programming languages, and operating systems can be achieved to produce spare components and programs. The N-version architecture is one of the fault tolerance architectures that appeared in the 70s to improve mission-critical systems' reliability and security. However, high reliability and security are also required in web services and applications for companies and organizations to provide more confidence to their customers and users. Moreover, some companies may pay a monetary penalty when their systems fail because of unreliable components or cyber-attacks.

In the N-version architecture, multiple versions of a program are developed by different developers that usually do not share anything except software specifications. Each version may have different designs and may be developed in a different programming language. So, it is expected that each version has different vulnerabilities. If one of the versions fails because of the exploitation of a vulnerability, the other versions may continue to operate as expected. The apparent disadvantage of the N-version architecture is its high cost. The cost of a project may double or triple for the second and third variants, respectively. Generating more than three variants is unlikely if the system is not mission or safety-critical. Another side effect of the N-version architecture is the verification of each variant. Especially for large programs, it is challenging to verify that each version is functionally equivalent, especially for large programs.

The K-variant is an alternative architecture that takes advantage of the N-version architecture at a reasonable cost. Variants in the K-variant architecture are generated by simple and safe source-tosource program transformations. So, the cost of the variant-generation process is significantly reduced. Program transformations in the K-variant architecture provide the diversity of critical data in memory for each variant. In this way, the survivability of systems against memory exploitation attacks is significantly improved [3]. In addition to memory level diversity in the Kvariant architecture, diversity in the execution environment can also be achieved, similar to the Nversion architecture [4]. Variants in K-variant systems can be deployed to different web servers

that may run on different operating systems. That may provide better security against memory exploitation attacks.

The main contributions to this paper are as follows:

- A K-variant architecture for web services and applications is proposed.
- The high-level design of the K-variant architecture for web services and applications is explained.
- Implementation of K-variant systems and other diversities such as operating system and web server level in the K-variant architecture are discussed.

The remainder of this paper is structured as follows. Section 2 explains the related research. Section 3 presents the K-variant architecture for web services and applications. Section 4 describes the high-level design of the K-variant architecture for web services and applications. Section 5 briefly explains the program transformation techniques used in the K-variant architecture. Section 6 discusses the implementation details of the K-variant architecture for web services and applications. Finally, Section 7 concludes and discusses future work.

RELATED RESEARCH

The reliability of web services was improved with a single version of a program by redundant data and functions using SOAP [5]. This approach is based on procedure triplication [6], where important procedures are triplicated to have the same signature but different implementations. The fault tolerance is achieved by calling each procedure sequentially with similar inputs; then, all results are voted by a majority algorithm.

Diversity is an important concept for improving the reliability and security of computer systems. Diversity makes systems more robust against replicated attacks [7]. In addition, diversity may tolerate accidental faults [8]. Diversity can be achieved at different levels, such as the interface level, application level, execution level, hardware level, and operating system level. Address space randomization [9], instruction set randomization [10, 11], DLL based randomization [12], stack space randomization [13], heap randomization [13], calling sequence diversity [14], encrypted instructions [15] are some of the diversity techniques that are used to improve security.

Diversity in architecture is one of the valuable techniques in fault tolerance. N-version programming [16, 17] is a prominent architecture that is used to improve mission and safetycritical systems. In N-version programming, multiple versions or variants of a program run concurrently to perform a mission or an operation. Multiple versions or variants are generated by different developers, only sharing software specifications. Different designs and programming languages can be used when developing different versions or variants. Eventually, each variant will have different vulnerabilities. Thus, if one of the variants is compromised because of an attack or a bug, the other versions or variants may continue to operate successfully.

N-version programming is an expensive process. Developing the second and third versions can double and triple the project cost. Besides the high cost, verifying the functional equivalence of all versions is a challenging process. Even for two small programs, it is hard to prove that two programs are functionally equivalent. For all these reasons, N-version programming is used only in mission or safety-critical systems where very high reliability and security are required. However, today, many business-to-business systems and profit or nonprofit organizations require high reliability and security because failures of these systems may cause huge losses in profits and prestige for them. Thus, N-version programming has started to be used in general-purpose systems such as web servers. The source [4] proposes an architecture for dependable web services using N-version programming. The prosed architecture uses design diversity and WS-BPEL (Web Services Business Process Execution Language) to make systems more adaptable. A variety of

Multidisciplinary Journal of Technology
Volume 11 Issue 4, October-December 2023
ISSN: 2995-441X
Impact Factor: 7.73
https://kloverjournals.org/journals/index.php/Tech

options for operating systems, web servers, application servers, database servers, programming languages, and IDEs are provided to achieve design diversity. Another fault tolerance architecture for web services is proposed in [18]. The proposed architecture is called FT-Web. In that architecture, a request is sent to active replicas of services. A component responsible for managing variants in the system analyzes received responses and decides the final response. A similar architecture is also proposed in [19]. A transparent middle layer achieves fault tolerance by sending requests to all replicas in the system. The middle layer also manages all variants, provides consistency between variants, and decides the final response to clients.

The K-variant architecture was proposed in [20] to improve the security of time-bounded missioncritical systems. The K-variant architecture is an alternative to N-version programming. Unlike Nversion programming, all variants in the system are generated by automated, safe, inexpensive program transformations. Since the variant generation process is automated in the K-variant architecture, the cost of systems is significantly reduced.

K-variant architecture for web services and applications

In this section, the K-variant architecture for web services and applications and its components are described. The K-variant architecture uses active replication to enhance security. Different variants of a program are generated automatically by using program transformations. In this paper, two versions of the K-variant architecture for web services and applications are explained. The first version is static, in which all variants are generated and deployed when the system starts up. The variants are never updated during runtime. The second version is dynamic, in which variants can be updated during the execution time. The proposed architecture can easily be switched between static and dynamic versions.

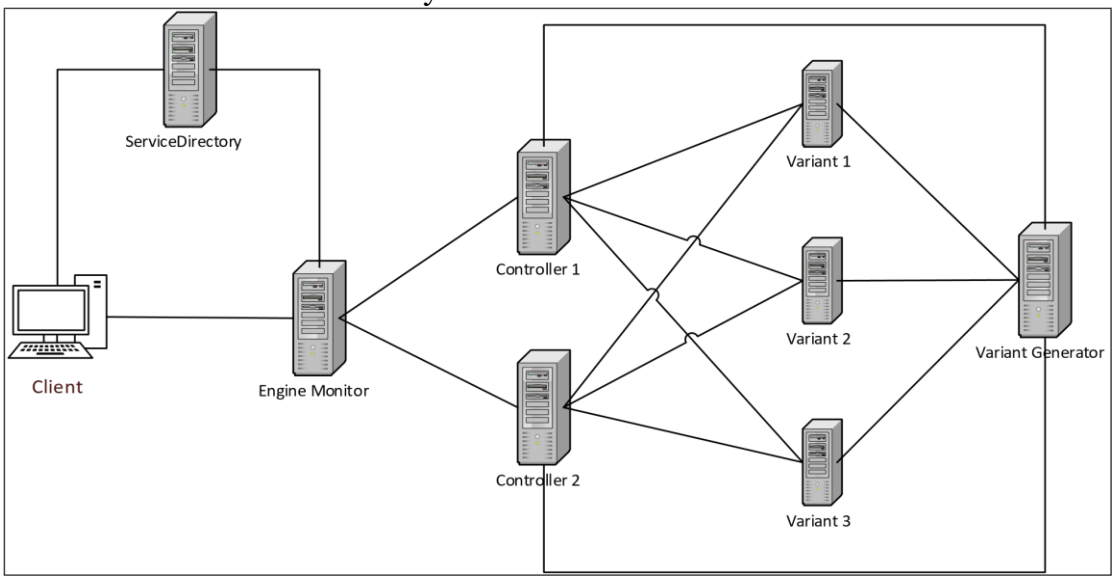


Figure 1 K-variant architecture for web services and applications.

The K-variant architecture for web services and applications consists of the following components. Some of the parts are similar to the traditional N-version architecture for web services that is shown in [4].

- ***Client*:** It is a user that requests service from the system. The *Client* may search for services from the *serviceDirectory,* an indexing engine, to find registered services.
- ***ServiceDirectory*:** It is a directory to keep registered/published services in the system. The *Client* looks up the *ServiceDirectory* to find available services and addresses of services. All services can be indexed with unique IDs, service names, interfaces, specifications, and addresses.

Multidisciplinary Journal of Technology
Volume 11 Issue 4, October-December 2023
ISSN: 2995-441X
Impact Factor: 7.73
https://kloverjournals.org/journals/index.php/Tech

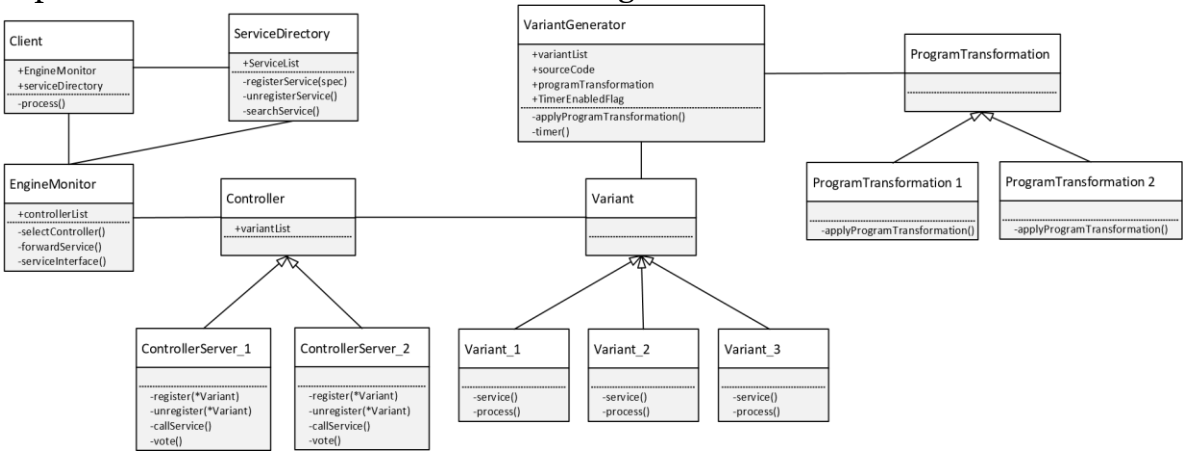
- ***EngineMonitor*:** It is an interface that clients interact with to get services. The *EngineMonitor* forwards requests to a controller to perform services. When there is more than one *Controller* in the system, the *EngineMonitor* also decides on a controller based on various factors, such as load balancing, the proximity of servers, etc. In the K-variant architecture, it is assumed that the *EngineMonitor* is safe because of the single point of failure.
- ***Controller*:** It is the core component of the K-variant architecture. It is responsible for managing variants, sending requests to variants, and voting for final results. The *Controller* receives clients' requests from the*EngineMotor*and sends them to all variants in the system simultaneously. In the *Controller*, a voting module is used to decide the final response to the *Client*. There may be more than one *Controller* in the K-variant architecture to prevent a single point of failure.
- ***VariantGenerator*:**It is a component that automatically generates variants using program transformations. The used program transformations are simple and safe. Thus, no additional software testing is required for automatically generated variants. The *VariantGenerator* deploys variants to different servers or locations after generating them. The *VariantGenerator*also has a timer for the dynamic model, in which variants are updated periodically during the runtime.
- ***Variant*:**It is a program that provides services. All variants are generated by the same program by applying source-to-source program transformations. Increasing the number of variants tends to improve the security of a K-variant system.

Figure 1 represents the high-level architecture of the K-variant architecture. The *Client* looks up a web service by using a service's specifications in the *ServiceDirectory*. Then, the Client sends its service request to the *EngineMonitor*. The service request can be synchronous or asynchronous depending on the *Client*'s application and the used protocol between the *Client* and the *EngineMonitor*. *EngineMonitor*decides on one of the *Controllers* in the system and forwards the service request to the selected Controller. The Controller sends service requests to all variants in the system simultaneously. After the *Controller* receives all of the results from variants, it votes for the final result. After that, the final result returns to the *EngineMonitor*. Finally, the Client received the result of the requested service from the*EngineMonitor*.

The *EngineMonitor* and *Controllers* form a middle layer, which is transparent to clients. The Client may not be aware of the number of variants, controllers, and engine monitors in the system.

The high-level design of K-variant architecture for web services

In this section, the object-oriented design of the K-variant architecture is demonstrated. Figure 2 shows the class diagram of the K-variant architecture for web services. The rest of the section explains each method in each class and design details.



Multidisciplinary Journal of Technology
Volume 11 Issue 4, October-December 2023
ISSN: 2995-441X
Impact Factor: 7.73
https://kloverjournals.org/journals/index.php/Tech

Figure 2. Class diagram of the K-variant architecture for web services.

Client Class: It has a *process()* that invokes *searchService()* from the *ServiceDirectory* to look up a service. Then, the *process()* calls *serviceInterface()* from *EngineMotor* to get the service.

ServiceDirectory Class: Offered services are published and unpublished by the *register()* and *unregister()* methods, respectively. Service specifications, including service name, ID, and interface, are provided when registering a web service. An internal data store (*serviceList*) may be used to keep service data. *searchSerive()* finds and returns the name of a published service from the *serviceList* by using service specifications.

EngineMotor Class:serviceInterface() provides an interface for the *Client* to get a service.Service name and parameters may also be sent when calling *serviceInterface(). selectController()* and *forwardService()* are called inside *serviceInterface(). selectController()* selects one of the controllers from the *controllerList*. That selection can be random. Also, different factors, such as load balancing, server proximity, etc., may be considered when selecting a controller. *forwardService()* forwards the service request with the required service parameters to the selected Controller.

Controller Class: *register()* and *unregister()* methods can add and remove variants from the *variantList,* which is a data structure to keep variants' information such as variant addresses, specifications, etc. *callService()* calls a requested service to all variants in the *variantList*. After receiving responses from all variants, the *vote()* method is called to decide the final result. Any voting mechanism that is used in the N-version architecture can also be used in the K-variant architecture.

VariantGenerator Class: It applies*ProgramTransformation()* method that takes the original source code and generates a variant by applying a program transformation. The strategy pattern [21] is used to apply different program transformations. The strategy pattern allows adding new program transformations with a minimum change in design. Program transformation classes include the implementation of different program transformations, which are called in *applyProgramTransformation()*. After generating new variants, they are deployed to different locations using addresses in the *variantList*. Depending on the programming language, the source codes of variants may need to be compiled. Thus, variant programs may be compiled, and executable files are deployed in different locations.

ProgramTransformation classes: These classes are related to the strategy pattern. Different program transformations are implemented in these classes. *applyProgramTransformation()* is a method that applies a specific program transformation.

Variant classes: They contain the implementation of web services. *service()*is a method that calls *process()* to perform a web service.

```
int main(){
    ...
    int val;
    int buffer1[5]; //definition of buffer1
    ...

    ...
    val++;
    buffer1[val] = 1; //the use of buffer1
    ...

    return 0;
}
```

Figure 3. Original Program

Multidisciplinary Journal of Technology
Volume 11 Issue 4, October-December 2023
ISSN: 2995-441X
Impact Factor: 7.73
https://kloverjournals.org/journals/index.php/Tech

Program transformations for the K-variant architecture

Variants in the K-variant architecture are generated by applying source-to-source program transformations. The goal of program transformations in the K-variant architecture is to shift the vulnerable memories in each variant. By applying memory shifter program transformations [22], the addresses of vulnerable memory will not be totally overlapped. This approach may improve the security against memory exploitation attacks. If one variant in the system is compromised because of an attack on vulnerable memory, the other variants may continue to deliver expected services. All program transformations that are used in the K-variant architecture have the following common features:

- They do not impact the functions or behaviors of programs.
- They shift the vulnerable data to different locations.
- They do not cause additional bugs in a program.
- The transformed program does not need significant software testing.
- The original source code is preserved as much as possible. ☐They have acceptable memory and runtime overheads.

In this section, program transformations that have been used in the K-variant architecture are briefly explained. These program transformations were explained in detail [22].

Inserting dummy buffers

It was the first program transformation for the K-variant architecture [20]. In this program transformation, a random number of dummy buffers with random sizes are inserted into the source code. A dummy buffer is a buffer that is defined but never used. A dummy buffer does not affect the program's execution, but it takes up space in the memory.

Dummy buffers can be inserted after the existing buffers in the source code. That may prevent potential buffer overflow vulnerabilities. An example program transformation of inserting dummy buffers is shown in Figure 4. In the example, a dummy buffer is inserted after the existing buffer in the original source code, which is shown in Figure 3. When a buffer overflow occurs in *buffer1*, the dummy buffer is manipulated instead of the critical data, which may affect the program's outcomes.

```
int main(){
    ...
    int val;
    int buffer1[5];
    int dummy_buffer1[5]; //definition of a dummy buffer
    ...

    ...
    val++;
    buffer1[val] = 1; //the use of buffer1
    ...

    return 0;
}
```

Figure 4. Inserting dummy buffers. The source code after the program transformation.

In this program transformation, any number of dummy buffers of any size can be inserted into the source code. The only limitation of this program transformation is the machine's memory size on which a variant runs.

Expanding the size of existing buffers

In this program transformation, random existing buffers are expanded by random sizes. Unlike inserting new dummy buffers, expanding the size of existing buffers provides more control over shifting vulnerable memory. New dummy buffers may not be placed next to existing buffers in some systems. In that case, inserting new dummy buffers may not prevent buffer overflows. On the

Multidisciplinary Journal of Technology
Volume 11 Issue 4, October-December 2023
ISSN: 2995-441X
Impact Factor: 7.73
https://kloverjournals.org/journals/index.php/Tech

other hand, the location of unused buffers can be easily determined with respect to the existing buffers by expanding the size of the existing buffers.

An existing buffer can be expanded to the right, left, or both directions. An example of the expanding buffer to the right is shown in Figure 5. In the expansion to the right, only the definition of the expanded buffers is updated. All the uses and references of the expanded buffer remain the same. In Figure 5, *buffer1* is expanded to the right by five units in its definition. The use of *buffer1* in the assignment statement does not change.

```
int main(){
    ...
    int val;
    int buffer1[5 + 5]; //definition of buffer1 is expanded to the right
    ...

    ...
    val++;
    buffer1[val] = 1; //the use of buffer1 remains the same
    ...

    return 0;
}
```

Figure 5. Expanding buffers to the right. buffer1 is expanded to the right by five units.

Figure 6 represents the expanding buffer to the left. Similar to the expansion to the right, the definition of the expanded buffer is updated in the left expansion. However, each use or reference of the expanded buffer must also be updated in its scope because the unused buffer is located at the beginning of the existing buffer. Therefore, the buffer's use or reference in the assignment statement is also updated in Figure 6. In this program transformation, all the values in the buffer are shifted to the right by the size of the expansion. In the example, all values in *buffer1* are shifted to the right by five. Thus, the index (*val*) of *buffer1* is incremented by five.

```
int main(){
    ...
    int val;
    int buffer1[5 + 5]; //definition of buffer1 is expanded to the
    left
    ...

    ...
    val++;
    buffer1[val + 5] = 1; //the use of buffer1 is updated
    ...

    return 0;
}
```

Figure 6. Expanding the buffers to the left. buffer1 is expanded to the left by five units.

In order to expand buffers in both directions, the left and right expansions can be applied consecutively. When buffers are expanded in both directions, the existing data is put in the middle of the expanded buffers.

The apparent disadvantage of program transformations of expanding buffers is that the number of expanded buffers is confined to the number of buffers in the program. On the contrary, there is no such limitation on inserting new dummy buffers.

Increasing dimensions of existing buffers

In the previous two program transformations, the data in the buffer was continuous. The goal of this program is to separate the data inside buffers and add dummy buffers between them. In this program transformation, the dimension of random buffers is expanded by random sizes. In this way, the data in the existing buffers are spread throughout the memory.

Multidisciplinary Journal of Technology
Volume 11 Issue 4, October-December 2023
ISSN: 2995-441X
Impact Factor: 7.73
https://kloverjournals.org/journals/index.php/Tech

In addition to updating the definition of the expanded buffer, any references or uses of the expanded buffer must also be updated in their scope. Because only one index is utilized in the new dimension, a random index needs to be selected for use or reference in the expanded buffers.
Figure 7 shows an example of increasing the dimension of existing buffers. The one-dimensional buffer is expanded to a two-dimensional buffer. The size of the new dimension is four. The use of the buffer in the assignment statement has also been updated. The new dimension index in the use of the buffer is two, a random number smaller than the size of the new dimension.

```
int main(){
    ...
    int val;
    int buffer1[5][4]; //a new dimension is added in the definition
    ...

    ...
    val++;
    buffer1[val][2] = 1; //the use of buffer1 is updated
    ...

    return 0;
}
```

Figure 7. Increasing the dimension of existing buffers. The dimension of buffer1 is expanded. The size of the new dimension is 4.
Index 2 is selected using the expanded buffer for the new dimension.

Converting primitive data type variables into buffers

One way to shift the vulnerable memory is by converting primitive data types, such as characters, floats, integers, and Booleans, into random-size buffers or arrays. In this program transformation, the value of the variable is placed in a random index in the new buffer. Other indexes in the buffer are unused. This program transformation involves updating both the definition and uses/references of the variables in the source code within its scope.
Figure 8 displays converting a primitive data type variable into a buffer. In this example, the integer variable *var* is converted into an integer array of size five. The value in variable *var* is moved to the second index of the buffer. Therefore, *var* is replaced with *var[2]* in the use of the variable in Figure 8.

```
int main(){
    ...
    int val[5]; //int is converted to an int array
    int buffer1[5];
    ...

    ...
    val[2]++; //the use of the val is updated
    buffer1[val[2]] = 1; //the use of the val is updated
    ...

    return 0;
}
```

Figure 8. Converting primitive data type variables into buffers. The integer variable val is converted to an integer array. The value of val is moved to the second index in the new buffer.

Implementation of K-variant architecture

This section explains and discusses some implementation details of the K-variant architecture. The modules in the K-variant architecture can be implemented within a single program and run on a single machine. Alternatively, a separate program can be developed for each module that runs on single or multiple machines. Although the K-variant architecture provides diversity in memory locations of critical data, other levels of diversity, such as operating systems, web servers, and messaging protocols between modules, can be achieved for additional security.

Multidisciplinary Journal of Technology
Volume 11 Issue 4, October-December 2023
ISSN: 2995-441X
Impact Factor: 7.73
https://kloverjournals.org/journals/index.php/Tech

In the N-version architecture, variants can be implemented in different programming languages. However, that is not possible in the K-variant architecture, in which all variants are generated by source-to-source program transformations. However, depending on the programming language, diversity on the webserver can be provided. Because each web server will have a different vulnerability, the security of the K-variant system will increase if the diversity of the web server is achieved for each variant.

Table 1 shows the most popular web servers with their supported operating systems, programming languages, the total number of reported vulnerabilities, and reported memory-related vulnerabilities. Supported languages and operating systems for the web servers are not limited to Table 1. With third-party support and new distributions, the coverage of the web servers keeps increasing. Therefore, more diversity can be achieved at the webserver level in the K-variant architecture. As seen in Table 1, an important percentage of the overall reported vulnerabilities are memory-related. Furthermore, a significant portion of these memory-related vulnerabilities has been reported in recent years. For example, 287 memory-related vulnerabilities have been reported since 1999 on the Apache Web Server. 133 of 287 memory-related vulnerabilities in the Apache Web Server have been reported since 2017 in the CVE Details vulnerability database [2]. That shows the potential threat of memory-related attacks on web servers and applications. For this reason, it is advised to have a diverse set of webservers in the K-variant design.

Table 1. Operating systems and web servers that can be used in K-variant systems to provide diversity. The vulnerability data numbers were retrieved from CVE Details [2].

Web Server	Operating system	Supported Languages	# of Reported vulnerabilities	# of memoryrelated vulnerabilities
Apache HTTP Server	Linux, Unix Windows, OpenVMS, Mac OS X	PHP, ASP.NET, Python, Prolog, Ruby, Perl, Lisp, Lua, JSP	1497	287
The Internet Information Server (IIS)	Windows	PHP, ASP.NET, Python, Prolog, Ruby, Perl	100	35
lighttpd	FreeBSD, Windows, Linux, Solaris, Mac OS X	PHP, Python, Perl, Ruby, Lua	29	4
Oracle iPlanet Web Server (OiWS)	Linux, Unix, Windows, Solaris,	PHP, Python, Perl	17	11
Jigsaw Server	Linux, Unix, Windows, Mac OS X, FreeBSD	PHP, JSP	3	0

Various program transformations can be utilized in *VariantGenerator* to generate new variants. The algorithms for four program transformations that can be used in the K-variant architecture are shown in [22]. By applying the strategy pattern, different program transformations can be selected at runtime. After generating the source code of variants, they may need to be compiled depending on the programming language. If variants are deployed on different operating systems, different compilers will be required to generate binary files for a specific operating system. In this case, the *Variant Generator* also needs to keep a list of compilers with their corresponding target operating systems and commands to generate binary files for a specific operating system. On the other hand, if the service is written in a scripting language, the implementation of the *Variant*

Generator may be much simpler. The transformed source code can be deployed to the server without any compilation.

The messaging protocol is another implementation detail that needs to be considered in K-variant systems. If variants are deployed on diverse operating systems and web servers, they need to be communicated efficiently and securely without any issues. SOAP (Simple Object Access Protocol) and REST (Representational State Transfer) are two standards that allow communication between diverse systems. Both SOAP and REST use the HTTP protocol available on all web servers. Therefore, SOAP and REST can be utilized in K-variant systems to communicate between modules. SOAP uses only the XML format, which may cause an additional burden of creating and parsing XML files. On the other hand, REST is not constrained to the XML file format. Any file format, including JSON, XML, CSV, etc., can be used with REST. REST can provide much faster communication than SOAP. However, creating requests and parsing responses is easy to implement with REST if a .NET language is used. So, the selected messaging protocol can depend on the performance and used programming language.

The *Controller* has one of the highest responsibilities in the K-variant architecture. Because of the single point of failure, multiple controllers can be introduced to a system. The *Controller* finds the service and requests all variants concurrently. The received responses need to be voted on to produce the final result. The voting module in the *Controller* can be a simple majority algorithm. Because all variants are generated from the same code, their trustworthiness is the same, and no weighted voting algorithm is required. An acceptance test module can also be introduced to the *Controller* to prevent voting responses from compromised variants.

CONCLUSIONS

In this paper, the K-variant architecture for web services and applications is described. The Kvariant is a multi-variant architecture that improves security against memory exploitation attacks. By applying source-to-source simple and safe program transformations, multiple variants are generated in the K-variant architecture. These program transformations shift the addresses of vulnerable data in memory so that the diversity of memory locations of critical data in each variant is achieved. This paper proposes designing a K-variant architecture for web services and applications. The proposed architecture aims to improve security against memory-related attacks.

Unlike other multi-execution architectures, the cost of the K-variant architecture is low because of the automation in generating multiple variants. In this paper, four program transformations: inserting dummy buffers, expanding the size of existing buffers, increasing the dimensions of existing buffers, and converting primitive data type variables into buffers are briefly explained. In addition, the added diversity at the webserver and operating system levels is discussed for Kvariant systems for web services and applications. Deploying variants to different web servers that run on different operating systems may provide additional security for K-variant systems.

In future work, the effectiveness of the K-variant architecture for web services and applications will be investigated experimentally for various types of memory attacks. Moreover, the overhead program transformations for web services and applications will be investigated.

REFERENCES

J. Dahse, "sonarsource," 20 6 2017. [Online]. Available: https://blog.sonarsource.com/security-flaws-in-the-php-core?redirect=rips. [Accessed 90 7 2021]. "cvedetails," [Online]. Available: https://www.cvedetails.com/. [Accessed 31 7 2021].

B. Bekirogluand BogdanKorel, "Source Code Transformations for Improving Security of Time-bounded K-variant Systems," *Information and Software Technology,* vol. 137, 2021.

E. Nourani, "A new architecture for Dependable Web Services using N-version programming," in *2011 3rd International Conference on Computer Research and Development*, Shanghai, China, March 2011.

G. K. Saha, "Single version fault tolerant web services," *Ubiquity,* no. 2007, p. 4:1, 2007.

G. K. Saha, "A Single-Version Scheme of Fault Tolerant Computing," in *JOURNAL OF COMPUTER SCIENCE & TECHNOLOGY*, 2006.

S. Forrest, A. Somayaji and D. Ackley, "Building diverse computer systems," in *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems*, Cape Cod, MA, USA, May 1997.

Y. Deswarte, K. Kanoun and J.-C. Laprie, "Diversity against accidental and deliberate faults," in *Proceedings Computer Security, Dependability, and Assurance: From Needs to Solutions*, York, UK & Williamsburg, VA, USA, 7-9 July 1998.

S. Bhatkar, Defeating memory error exploits using automated software diversity, Thesis State University of New York at Stony Brook, 2007.

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*, Washington D.C., USA, October 27, 2003.

G. Portokalidis and A. D. Keromytis, "Fast and practical instruction-set randomization for commodity systems," in *Proceedings of the 26th Annual Computer Security Applications Conference*, Austin, Texas, USA, December 6, 2010.

S. Bhatkar, D. C. DuVarney and R. Sekar, "Address obfuscation: an efficient approach to combat a board range of memory error exploits," in *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, Washington, DC, August 4, 2003.

S. Bhatkar, R. Sekar and D. C. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," in *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, Baltimore, MD, July 31, 2005.

D. Williams, W. Hu, J. W. Davidson, J. D. Hiser, J. C. Knight and A. Nguyen-Tuong, "Security through Diversity: Leveraging Virtual Machine Technology," *IEEE Security Privacy,* vol. 7, no. 1, pp. 26-33, January 2009.

M. Milenković, A. Milenković and E. Jovanov, "Using instruction block signatures to counter code injection attacks," *ACM SIGARCH Computer Architecture News,* vol. 33, no. 1, p. 108–117, March 1, 2005.

A. Avizienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE Transactions on Software Engineering,* Vols. SE-11, no. 12, pp. 1491-1501, December 1985.

L. Chen and A. Avizienis, "N-VERSION PROGRAMMINC: A FAULT-TOLERANCE APPROACH TO RELlABlLlTY OF SOFTWARE OPERATlON," in *Twenty-Fifth International*

Symposium on Fault-Tolerant Computing, 1995, ' Highlights from TwentyFive Years'., Pasadena, CA, USA, June 1995.

G. Santos, C. L. Lau and C. Montez, "FTWeb: a fault tolerant infrastructure for Web services," in *Ninth IEEE International EDOC Enterprise Computing Conference (EDOC'05)*, Enschede, Netherlands, Sep. 2005.

X. Ye and Y. Shen, "A middleware for replicated Web services," in *IEEE International Conference on Web Services (ICWS'05)*, Orlando, FL, USA, July 2005.

B. Korel, S. Ren, K. Kwiat, A. Auguste and A. Vignaux, "Improving operation time bounded mission critical systems' attack-survivability through controlled source-code transformation," Sydney, Australia, November 14, 2011.

R. H. R. J. J. V. Erich Gamma, Design Patterns: Elements of Reusable Object-Oriented Software, Pearson Education, 1994.

B. Bekiroglu and B. Korel, "Source Code Transformations for Improving Security of Timebounded K-variant Systems,"*Information and Software Technology (Elsevier),* vol. 137, 2021.

P. Brady, "Cross-Site Scripting (XSS)," [Online]. Available: https://phpsecurity.readthedocs.io/en/latest/Cross-Site-Scripting-(XSS).html. [Accessed 1 6 2020]. "Zend\Escaper," Zend, [Online]. Available: https://framework.zend.com/manual/2.1/en/modules/zend.escaper.introduction.html. [Accessed 1 6 2020].

U. Ladkani, "Prevent cross-site scripting attacks by encoding HTML responses," IBM, 30 7 2013. [Online]. Available: https://www.ibm.com/developerworks/library/se-prevent/. [Accessed 1 6 2020].

B. Hope, P. Hope and B. Walther, Web Security Testing Cookbook: Systematic Techniques to Find Problems Fast, Sebastobol, CA: O'Reilly Media, 2009-05-15.

B. Bekiroglu and B. Korel, "Survivability Analysis of K-Variant Architecture for Different Memory Attacks and Defense Strategies," *IEEE Transactions on Dependable and Secure Computing,* vol. 18, no. 4, pp. 1868-1881, 2021.