World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:16, No:9, 2022

Extending the Aspect Oriented Programming Joinpoint Model for Memory and Type Safety

Amjad Nusayr

***Abstract*—**Software security is a general term used to any type of software architecture or model in which security aspects are incorporated in this architecture. These aspects are not part of the main logic of the underlying program. Software security can be achieved using a combination of approaches including but not limited to secure software designs, third part component validation, and secure coding practices. Memory safety is one feature in software security where we ensure that any object in memory is have a valid pointer or a reference with a valid type. Aspect Oriented Programming (AOP) is a paradigm that is concerned with capturing the cross-cutting concerns in code development. AOP is generally used for common cross-cutting concerns like logging and Database transaction managing. In this paper we introduce the concepts that enable AOP to be used for the purpose of memory and type safety. We also present ideas for extending AOP in software security practices.

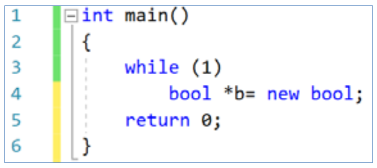
***Keywords*—**Aspect oriented programming, programming languages, software security, memory and type safety.

I. Introduction

TYPE safety and memory safety is regarded as one of the main pillars in secure systems. Systems are vulnerable once protected memory is accessed. Protected memory that is part of a secure system could contain Level 1 sensitive data or instructions. An attacker could instrument and write an interception segment to protected memory or create a trampoline jump to some unintended-unwanted malicious code. Programmers who use C and C++ languages must manage their use of memory to ensure safety. Any allocation, reference, or deletion to memory must be done explicitly via code. A simple mistake on the programmer's side could lead to unwanted consequences. See Fig. 1, for each iteration of the infinite while loop, the program allocates a new byte in memory. The pointer pointing to this byte will point to another newly allocated memory address on every remaining iteration. The previous byte will not be referenced any more. Something simple like this will cause total memory consumption in a small amount of time. Fortunately, newer systems will detect abnormal use of memory and will kill any process for that program and release back the memory. This was not the case in older operating systems. We were able to compile this code using an older Borland C++ Compiler and run it on windows 98. This resulted in the crash of the operating system and a hard restart to the machine.

AOP is relatively a newer paradigm in programming [2], in which, cross cutting concerns which are not part of the main code logic of some programs are written in separate segments called aspects and then are weaved into their proper location based on user defined expressions. Most of these expressions are code based (i.e., the code is weaved into specific code locations). AOP reinforces the concepts of encapsulation and abstraction by guaranteeing that base logic code and cross cutting concerns are both in written in separate programming modules.

Amjad Nusayr is with University of Houston-Victoria, United States (e-mail: nusayra@uhv.edu).



```
int main()
{
    while (1)
        bool *b= new bool;
    return 0;
}
```

Fig. 1 A continuous one-byte memory leak

This paper presents ideas for extending the use of AOP as a tool to detect possible memory violations and type safety. Our contribution in this paper is the use of the AOP for the general domain of security safety. The rest of the paper is structed as follows: Section II is a background. Section III presents what is needed to extend AOP for such use. Section IV presents related work followed by the conclusion and future work is in Section V.

II. Background

A. Memory and Type Safety

One of the most important principles of a secure system is memory safety [1], [3]. This is done by ensuring each memory access is bounded to a valid object. In C and C++ this is made by having pointers point to valid memory objects. In other memory managed languages like Java and the .Net based programming languages, this is made by guaranteeing that the garbage collector in the underlying framework is working at the right time and the right memory location. Functional programming languages on the other hand do not expose any direct pointer or memory location to the programmer, thus, guaranteeing memory and type safety to the programmer. Yet the biggest issue with memory and type safety is the fact that achieving this goal comes with at a price. Some of the safety checks can be done at a static level but most checks must be done at a dynamic level while code is executing. This means that the underlying system executing the code will have much overhead time to assure safety. This is one reason many programmers still prefer C or C++ over memory managed languages like Java and C#.

Memory and type safety have a greater impact when it comes to code that executes on customized operating systems and/or fewer hardware resources. We consider the AVR based MicaZ [4]. The CPU's register-space is mapped to the bottom of the address space where the register 28 and register 29 are frequently used for memory indexing. The code in Fig. 2 will cause a memory fault and memory corruption.

```
int c = 0;
struct { char a[28]; int b; } *ptr;
ptr = null;
ptr->b = c;
```

Fig. 2 Dereferencing a pointer after null in AVR MicaZ

Next, we discuss the two major categories in memory and type safety; spatial memory safety and temporal memory safety. Spatial memory safety ensures that all pointers are pointing to a valid and intended memory object (i.e., memory dereferences will be to a valid object in memory). This can happen for a variety of reasons. These includes, but not limited to, pointer arithmetic, singleton pointers, arrays out of bounds or unchecked arrays, pointers to offset of the start of a memory objects. Memory objects are allocated and destructed in an explicit way using function like malloc and free respectively.

Other languages like Java and C# enforce memory safety at runtime. Managing the memory without the use of pointers has its overhead time. Both languages utilize a garbage collector to reclaim unused memory.

Temporal memory safety is achieved by ensuring that all memory dereferences are valid at runtime. It also ensures that memory associated with a prior object that has been freed is totally reclaimed by the underlying system and no longer associated with the object.

Violations in temporal safety can happen in several ways. One of the most popular is when a location in memory is read before it is written. Another example is to return data from a prior object that had the same address as a current object in memory. Fig. 3 shows an example to a violation where a pointer is no longer pointing to a valid object in memory because the object has been freed. A statement to access the memory will cause this violation (i.e., *b = 0;). It is even more sophisticated when it comes parallel programming and shared memory. Violations can simply occur because of a race condition. Having the need to constantly monitor shared access space is imperative.

```
bool *a = malloc(17);
bool *b = a;
free(a);
```

Fig. 3 Example of a temporal violation

B. Aspect Oriented Programming

AOP is a programming paradigm used for constructing and implementing segments of code that are otherwise considered orthogonal to the underlying program code logic. These segments of code are referred to as cross cutting concerns and they are scattered into many locations in the base program in a non-local and a non-modular matter. AOP enables programmers to write this orthogonal code into their own encapsulated containers, called aspects, and then insert them into their proper locations in a step called weaving.

AOP includes several concepts that are explained:

1- Aspect: Similar to a class in object-oriented programming languages, this is the main container that holds all other instructions and definitions that are used to weave code into some underlying program. The word "aspect" in AspectJ, a popular AOP language, is a reserved keyword.

2- Advice: Is the code that will be weaved, or instrumented to some underlying program. An advice has a type. Usually there are three types of advices, the "before" advice, the "after" advice, and the "around" advice.

3- Joinpoint or point-cut designator: Is a well-defined point in a program where advice will be weaved in. This point could be code-based, like a function call, or a back-edge of a loop. It also could be a point in time; absolute or wall clock time or based on any time constraint. Joinpoints could be data based, where weaving of an advice is centered over concepts in data space

4- Point-cut expression: Is simply an expression that defines where advice is to be weaved. Weaving can occur at multiple source code locations or programs in one step. Fig. 5 shows two cutting concerns in four different programs.

A point-cut expression could have its own identifier and allows the program to specify one or more targets in the underlying code. The point-cut expression example in Fig. 4 targets two specific joinpoints. In this example, the point-cut expression is interpreted as a call to the method foo(int) of type Class C1, or the execution of the method bar(int) of the class C2.

```
  //point cut expression
  pointcut p1() :
      call (void C1.foo(int)) ||
      execution (voidC2.bar(int))
```

Fig. 4 Definition of a pointcut

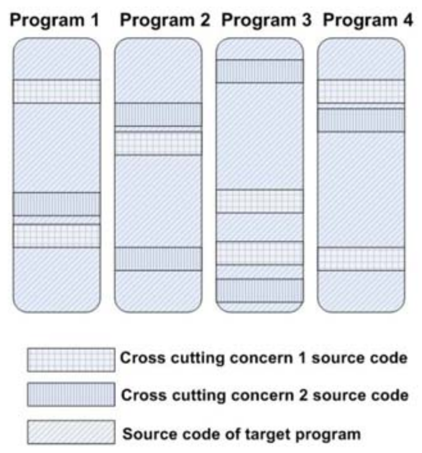

Program 1
Program 2
Program 3
Program 4
Cross cutting concern 1 source code
Cross cutting concern 2 source code
Source code of target program

Fig. 5 Two concerns being weaved into different programs

AOP concepts enable developers to isolate aspects like security and privacy concerns. This results in better code semantics, easier maintainability, more modularity, and code reuse. This makes AOP a natural fit to be used for such purpose. We still can observe that AOP can be extended to included custom designed point-cut expressions and joinpoints that target type and memory safety.

III. Extending AOP

There is a clear link in using AOP for the purpose of memory and type safety. Yet AOP does not fully support all needed functionalities. The major part of our contribution here is to detail how we plan to extend AOP joinpoint model and the point-cut designators that will enable AOP to support the many needs for memory and type safety.

The heart of what makes AOP so significant is the use of the joinpoint model. This model was created on a practical basis. For example, most of the joinpoints found in AspectC++ or AspectJ are code based joinpoint. That means the programmer needs to define his/her own set of "where in the code" the weaving must happen. An example is weave code before the execution of certain basic blocks in code. We find this model to be useful but on the other hand limiting when it comes to using AOP in the bigger picture for memory and Type safety.

Achieving goals in memory and type safety means that one must have some program monitoring for this cause while a program is in execution. We propose a more extensive set of point-cut designators to achieve these goals which fall into the following categories:

1 Code based: Joinpoints over the code space
2 Time based: Joinpoint that are based on time
3 Data based: Dynamic joinpoint based on memory constraints

Even though these three categories could be used for many memory and type safety features, we limit extending the AOP to be used in two domains; spatial memory safety and temporal memory safety. Next we detail extending AOP in the categories above.

1) Code Based: This is the traditional category that AOP already includes. Most AOP implementations like AspectJ already include a wide set of joinpoint definitions that are common across the domain of AOP. These include method calls, method executions.

In order to extend the AOP joinpoint code based model for memory and type safety, we propose including more fine-grained, custom made, joinpoints. The first two are in Fig. 6.

```
//1- dynamic joinpoint for
//upper boundary check
upperBoundCheck()
//2- dynamic joinpoint for
//lower boundary check
lowerBoundCheck()
```

Fig. 6 BoundCheck joinpoint

The two joinpoints would target the following scenarios illustrated in Figs. 7-9 respectively:

1- Dynamic memory buffer is allocated, and the buffer has an overflow.

```
a[MAX+10]=x;//MAX is the upper bound
```

Fig. 7 Upper bound violation

2- A violation via a direct write operation to the memory

```
a[-10]=x; //lower bound violation
```

Fig. 8 Lower bound violation

3- An invalid pointer that has been dereferenced

```
char *c=a+MAX;
b=b+10;
*a='A'; // violation
```

Fig. 9 Dereference violation

The third proposed joinpoint for this category is in Fig. 10.

```
// pointer or reference joinpoint
validObject(void *)
validObject()
```

Fig. 10 Joinpoint for each memory access

This joinpoint represents an access operation to a dynamically created object in memory. This joinpoint is overloaded and has two versions. The first version is parameter-less version. Advice that will be associated to this joinpoint will be weaved at every dereferencing operation. The second version is more precise and will target any dereferencing operation associated with a provided pointer. For example, adding "a[10]=0;" statement at the end of Fig. 3 is one example where this joinpoint in code resides.

2) Time Based: We have to cover situations in which a memory violation is, for example, consuming up memory either faster than it needs to or in some abnormal manner that otherwise, the program is not supposed to do so.

To address these points, we propose two joinpoints for this category as in Fig. 11.

```
// Runs every N units in time
everyNth(int)
// Runs at every some absolute
// time
timeAbs(time s, time e)
```

Fig. 11 Time Based joinpoint

Advice attached to any pointcut expression that uses the first joinpoint, everyNth(int), will execute every *nth time* unit. This joinpoint gives the programmer total control over how frequently data need to be collected for about a certain object or objects in memory. This will enable the programmer to have some capability for memory sampling.

The second joinpoint, timeAbs(time), is a simple and power addition. Advice attached to this joinpoint starts executing at a certain wall clock time until some end time. Scenarios for this model is when memory objects need to be monitored as a certain time.

3) Data Based: The current AOP joinpoint model that deals with data fields is only limited to object fields. Dynamic memory allocation happens in actual methods or functions. We propose extending the semantics for this model to include allocations that occur in the heap memory rather than the stack We introduce the following joinpoint in Fig. 12:

```
// Attached to a memory object
targetObject(void *)
```

Fig. 12 Data based joinpoints

This advice for this joinpoint is bound with some memory object. This allows the programmer to monitor the behavior of that object in memory and the references/pointer in bound to the object.

Extending the AOP for the purpose of memory and type safety can be illustrated in Fig. 13, where a program executing be can observed from three different perspectives.

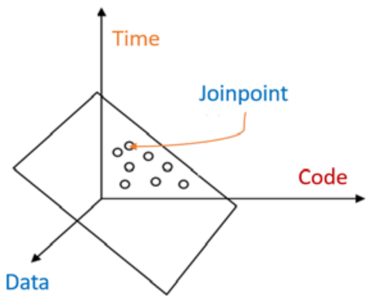


Fig. 13 AOP of proposed joinpoint model categories

IV. Related Work

Extending the AOP framework in general has been a reported in several research areas. Bodden & Havelund [5] have extended the pointcut model to create a Racer Algorithm to find subtle data races in memory. Llewellyn-Jones et al. [6] present a formalization for how to apply cross cutting concerns for larger networked systems. The earliest work found on extending the joinpoint model was made by Ubayashi et al. [7].

Memory and type safety is still in continuous research. This is because unmanaged-memory based program languages are still widely used. Duck and Yap [8] introduce the notion of dynamically typed C/C++ to detect such errors by dynamically checking the "effective type" of each object before the use at runtime. They also introduced a system for enforcing type and memory safety using a combination of pointers, type meta data and type/bounds check instrumentation. The work presented in [9], [3] show that compiler based dynamic checks can cause undesired overhead. They present a novel “toolchain” avoid these runtime costs.

Although some ideas fall in this general category [10], most of the work found the research does not fully utilize the capabilities and semantics of AOP to be used in memory and type-safety.

V. Conclusion

This paper presented ideas for extending the normal AOP concepts to support the notion of memory and type safety. We have presented the limitations in the current AOP joinpoint model for this case and introduced three main categories in which the joinpoint model can be extended for the purpose of achieving memory and type safety. We have proposed to add a time based and the data based categories that are necessary to facilitate the needs we are looking for.

The weaving process ideas required for the new joinpoint offered are out of the scope of this paper. This, by itself, is a bigger topic since both static and dynamic/run time weaving is necessary.

Our work is still in the “preliminary phase”. Newer tools for reverse engineering such as Ghidra and IDA pro could be utilized to find the new joinpoints proposed and insert the proper advice in these locations.

References

[1] S. L. Kanniah and M. N. ri bin Mahrin, “Secure software development practice adoption model: A delphi study,” *J. Telecommun. Electron. Comput. Eng.*, vol. 10, no. 2–8, pp. 71–75, 2018.

[2] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, “An Overview of AspectJ,” in *Proceedings of the 15th European Conference on Object-Oriented Programming*, 2001, pp. 327–353.

[3] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr, “Efficient memory safety for TinyOS,” in *SenSys’07 - Proceedings of the 5th ACM Conference on Embedded Networked Sensor Systems*, 2007, pp. 205–218.

[4] C. Technology, “MICAz: Wireless Measurement System,” *Prod. Datasheet*, pp. 4–5, 2008.

[5] E. Bodden and K. Havelund, “Aspect-oriented race detection in Java,” *IEEE Trans. Softw. Eng.*, vol. 36, no. 4, 2010.

[6] D. Llewellyn-Jones, Q. Shi, and M. Merabti, “Extending aop principles for the description of network security patterns,” in *Cyberpatterns: Unifying Design Patterns with Security and Attack Patterns*, vol. 9783319044477, 2014.

[7] N. Ubayashi, “An AOP Implementation Framework for Extending Join Point Models,” in *Proceedings of ECOOP 2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE’04*, 2004, pp. 71–81.

[8] G. J. Duck and R. H. C. Yap, “EffectiveSan: Type and memory error detection using dynamically typed C/C++,” *ACM SIGPLAN Not.*, vol. 53, no. 4, pp. 181–195, 2018.

[9] J. Regehr, N. Cooprider, W. Archer, and E. Eide, “Efficient type and memory safety for tiny embedded systems,” in *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, 2006.

[10] [A. usayr, J. Cook, and G. Rahnavard, “TEAMS: A special-purpose AOP framework for runtime monitoring,” in *Proceedings - 23rd IEEE International Symposium on Software Reliability Engineering Workshops, ISSREW 2012*, 2012.