Mind, Unity and Software Security - Analysis of Functional Unity in Cases of Data-only Attack

Ziyuan Meng
Drew University
36 Madison Ave, Madison, NJ
zmeng@drew.edu

ABSTRACT: *The computer security research community today still lacks a theoretical understanding of the essence of security vulnerabilities. The article argues that the prevailing reductionism in computer science theory leads to insecure coding practice, and Immanuel Kant's philosophy of mind sheds light on what makes software secure. In particular, Kant's constructivist conceptualization of the mind and his theory on the unity of the mental faculties inspire us to develop a new, non-reductionist approach to software vulnerability problems. We argue that a computer program can possess some structural similarities to the human mind. Similar to the unity of human mind, there is also a functional unity or 'integration' in any given program. In light of this similarity, a cyber-attack can be viewed as operations to compromise a computer program's original function by violating its internal integration. To illustrate the point, we provide a detailed analysis of two examples of data-only attacks, a new emerging threat to software security. In each case study, we examine the internal, functional integration of the case program and how data-only attacks affect the integration. The result shows a direct correlation between functional integration and the security of software. In the end, we propose a new technical normativity of cultivating to supplement that of coding.*

Keywords: Philosophy of Mind, Philosophy of Technology, Software Security, Kant, Integration, Reductionism

Received: 3 July 2021, Revised 4 October 2021, Accepted 31 October 2021

DOI: 10.6025/isej/2021/8/2/65-74

1. Introduction

As information technologies are increasingly integrated into everyday life, cybersecurity has evolved into one of the most urgent challenges that society faces. What has become clear is that the software and hardware of the digital infrastructure have not been designed with sufficient security considerations. As a result, many computer systems contain zero-day vulnerabilities: unknown design or implementation flaws that can be exploited by cybercriminals. The new and evolving cyber threats of the past decade have shown the far-reaching impacts of security vulnerabilities in both physical and social realms. These vulnerabilities allow hackers to compromise a computer system and cause unintended behavior. In some circumstances, hackers can even control physical infrastructures. For example, a computer malware named Stuxnet found in 2010 is the first known malware that targets physical systems [1]. Stuxnet exploits multiple zero-day vulnerabilities in the industrial control

system of an uranium enrichment plant in Iran. The malware alters the control system 's program, resulting in the centrifuges being spun too quickly, eventually damaging them. The current computer security approach, which we believe is inadequate, is primarily based on user feedback. For example, a vulnerability in software is first reported, and the software vendor then fixes it and releases the patch to the users. It is the authors' belief that the digital civilization of the future demands an investigation of the *conditions* for the possibility of meaningful computations. In the context of computer security, we would like to ask what constitutes the intended form of computing and what constitutes a mis-computing because of vulnerabilities.

We believe that an investigation of potential similarities between machines and minds is required. Instead of asking how "accurately" computer systems simulate or conform to objective reality, we turn the investigation in a different direction and ask how the internal structure of computer systems conform to our mode of thinking, to our mind. It is our belief that looking at the connection between computer systems and the human mind can help us understand what constitutes meaningful computation and how to detect miscomputations due to vulnerabilities. In their work [2], Jon and Ziyuan have proposed that software developers and computer scientists can borrow concepts from the 18th century German Philosopher Immanuel Kant's theory of mind to have a holistic, anti-reductionist understanding of the conditions of secure computation. They propose the following thesis on the two necessary conditions of secure computation:

1. Similar to how the human mind uses a priori concepts to synthesize and unify its various sense perceptions, a computer program employs pre-defined concepts and logical rules (algorithms) to integrate information taken from inputs and produce meaningful results. "Therefore, for a system to be secure, an input can generate computational outcomes only after being processed within the system's pre-existing internal structure."

2. Furthermore, similar to the Kantian understanding of the mind as a unity of its mental faculties, different components in a secure program do not exist in isolation. Instead, they are functionally integrated during the runtime execution.

In this article, we provide an overview of their thesis and apply it to critique the prevailing reductionist assumptions in computer science theory and education (section 2). We then engage with recently emerged data-only attacks as the case studies to evaluate the above thesis (section 3). We include a detailed analysis of two examples: an attack against an authentication algorithm and an attack against a web server. Both attacks exploit the buffer-overflow vulnerability. Our analysis of these attacks illustrates how a secure program always integrates data inputs while maintaining its internal functional unity and how cyberattacks cause the violation of the unity. In the end, we propose a non-instrumental technical normativity that views software development processes as the cultivation of quasi bio-cognitive beings.

2. Computer and Mind

Computer and information science has gone through over six decades of rapid development. As with any other discipline, it did not emerge from a cultural, intellectual vacuum. In its theoretical development and practical application, reductionist ways of thinking have dominated the field. This reductionist tendency manifests at two levels. First of all, the historical development of formal logic makes modern computer science possible. These include ancient foundations of Aristotelian logic and modern advancement of formal logic in the 20th century. One character of formal logic is that the *form* of the logic gains its autonomy by abstracting away from the *content* of thought [3]. To see this, we could think of any example of traditional Aristotelian syllogistic logic, e.g., 'All A are B; All C are A; therefore, C are B.' Here, we can apply this logical form to the world where A is 'mammal', B is 'animal', and C is 'elephant'. This separation between logical form and content in formal logic gives rise to dualistic and reductionist ways of thinking in computer science. In the imperative, procedural programming paradigm, algorithmic logic and data are two separate components in a given program. In his celebrated work Algorithms + Data Structures = Programs, the famous computer scientist, Turing Award winner, Niklaus Emil Wirth, considers data and algorithms as two related yet distinct aspects of computing [4]. Second, there has been a long tradition of *atomism* in thinking about an algorithm as a sequence of computing operations. Each operation is considered to have a precisely defined, *fixed* meaning in itself regardless of the context of computation. The renowned computer scientist Brian W. Kernighan once famously said:

> "*An algorithm is a sequence of precise, unambiguous steps that perform some task and then stop; it describes a computation independent of implementation details. The steps are based on well-defined elementary or primitive operations.*" [5]

The reductionism permeating computer science neglects one necessary condition of meaningful computations: *unity*. In a *meaningful* algorithm, data input must be unioned with the algorithmic logic to produce correct results. Moreover, its computational steps are not isolated "logic atoms" within an algorithm. Instead, they are functionally *integrated*. To illustrate the point, consider the following simple program written in pseudo code:

```
1. read (n);
2. i := 1;
3. sum := 0;
4. while i <= n do
5.   { sum := sum + i;
6.     i := i + 1; }
7. write (sum);
```

Code 1. A program to add up integers from 1 to n

.

The program takes an input from the user, stores it in a variable named n. It then uses a while loop to calculate the sum of integers from 1 to n. The program uses a variable named sum to accumulate the calculation result. If we focus on the final state of sum, it is clear that a few elements influence its final state . First of all, there is influence from the user input n. However , this influence is indirect . The code in the program processes the input to produce the summation . Moreover , statements in the program are interdependent . We usually think of an algorithm as a sequence of instructions , each with a fixed meaning. In reality, the meaning of each instruction is alwayssituatedin the particular context of a program. In a given program, multiple statements are involved in determining a particular outcome . One statement 's meaning is inseparable from the meaning of others . For example , the execution of the statement sum := sum + i depends on the condition of the while loop: i <= n. The variable i must be appropriately updated in the statement i := i + 1 before adding to sum in the next iteration. The way the program initializes i and sum in lines 2 and 3 also influence all the subsequent operations in the loop. We can use aprogram dependence graph (PDG) to represent the dependency relationships among the statements [7]. A PDG of a program is a directed graph in which the nodes represent statements, and the edges represent dependencies. Figure 1 shows the PDG of the program from code 1.

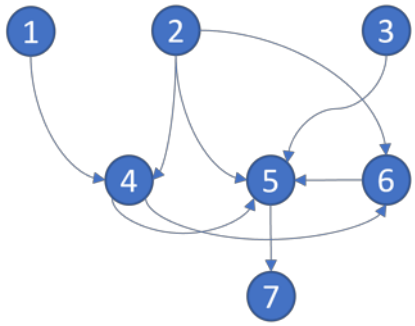

Figure 1. The program dependence graph of the program in code 1

Based on the PDG of a program, there is a well-known code analysis technique called program slicing to reason about the set of statements in a program that affects a given variable's state at any given location in a program [6]. If we focus on the variable sum, the PDG in Figure 1 clearly shows that all the statements in the above program contribute to its final state. Together with the user input, these statements form an integrated group to determine the final state of sum.

A program can generally have multiple groups of integrated statements, each determining a different computational result. Consider the following extended program written in pseudo code:

[1]In a formal representation, a PDG has an entry and makes the distinction between control dependence and data dependence. In this article, we omit the entry and difference between control and data dependencies.

```
1.  read (n);
2. i := 1;
3. sum := 0;
4. product := 1;
5. while i <= n do
6.   { sum := sum + i;
7.      product := product * i;
8.      i := i + 1; }
9.  write (sum);
10.write (product);
```

Code 2. A program to produce sums and products from 1 to n.

In addition to calculating sum, the above program also uses a variable named product to calculate the products of integers from 1 to n. From the final state of sum 's perspective , the code in lines 4 and 7 is irrelevant . Although these two lines of code are integrated with the computing process of determining the final state of product, they are not integrated with regards to the final state of sum.

How a computer program processes input from the external world and integrates the operations within its algorithm shares similarities with 18th century German philosopher Immanuel Kant's theory on how the mind functions. Kant famously criticizes the empiricist theory of mind in his time, which was proposed by thinkers such as David Hume and John Locke. According to the empiricist theory of mind, all human knowledge is derived only from sensory experience [8]. The mind functions like melting wax, passively and directly taking sensory impressions from the external world. In contrast to the empiricist approach to mind, Kant's theory of mind is *constructivist*. He argues that human knowledge arises only after the mind employs pre-existing concepts to synthesize/integrate the manifolds of received sense data into a single cognition. That is to say, knowledge is the result of the mind's active constructions using sense data as "raw material". If the mind had only sensory impressions without a stable conceptual order and structure to organize them, the impressions would be blind noise, and we could not perceive any meaningful patterns[8].

Furthermore, the mind's ability to synthesize the stream of sense data into meaningful experience implies that the mind itself is a unified system [8]. A *unifier* must be unified in itself. Kant's view of mind is anti-reductionism. The mind, which he refers to as "the original unity", is irreducible to the *sum* of its faculties [9]. Different faculties of the mind, such as the faculty of remembering, the faculties of comparing, the faculties of inferring, are not separable from each other. In any mental process, these faculties operate together in an integrated form. That is to say, the mind is an integrated cognitive system.

In their work[2], Jon and Ziyuan argue that computer security can learn from Kant's holistic view of mental functions. They postulate that computer programs as creations of the human mind share certain structural similarities with the mind. First, like the Kantian mind, a secure program never allows input data from an external environment to generate computational results directly. It always integrates the input data with its internal algorithm and memory state to determine the computing outcomes. Second, like the Kantian mind's unity, a secure program's different elements are functionally integrated during the runtime execution. Jon and Ziyuan argue that the above two conditions are necessary for any program to be secure. To illustrate the point, they use a case study of structured query language (SQL) injection attack against a web application to show that a secure web application always integrates its key components: data storage, logical controller, and user-inputs, while an insecure web application fails to maintain its unified form under a SQL attack.

In the next section, we will use case studies of data-only attacks to validate the above thesis on the conditions of secure computing. We restrict our attention to the imperative, procedural programming paradigm. Other programming paradigms, including the object-oriented and functional approaches, do not exhibit the same duality, structuring respectively around data and around algorithm logic. The degree to which Kantian duality applies to such paradigms, and the form(s) it may take, is a

consideration for future work.

3. Case Study Analysis

In this section, we will evaluate their thesis in the cases of data-only attacks. Data-only attacks are an emerging threat to vulnerable programs. The threat is seen most often in imperative procedural programming languages such as C. Unlike the conventional exploitation techniques which either inject executable code or alter the structure of the code in a target program, data-only attacks only manipulate the *data structure* of a target program[10]. By controlling the states of critical variables or data structures, a data-only attack can cause the target program to perform unintended operations, completely deviating from the original intent of its algorithm. In this paper, we will provide detailed analysis of two examples of data-only attacks: an attack against an authentication program written in C and an attack against a file server. Our analysis is informal and qualitative . Our goal in this paper is not to develop any new solution to mitigate data-only attack vulnerabilities. Instead, we will use data -only attacks as empirical case studies to examine the above thesis as steps toward a more general theory of secure computation. This study aims to show that Kantian philosophy of mind can illuminate the essence of software security. In each case study, we will examine the union of data and programming logic and the functional unity within the program . As illustrated below, the data-only attacks have destructive effects on the functional unity of target programs. We will also see that the countermeasure to mitigate the data- only attacks is, in its effect, to ensure such unity.

3.1 A Motivating Case Study: A Vulnerable Password Checker

Our motivating case study is a program that implements a simple login algorithm that verifies a user's password. The program receives a user-provided password through the network, then compares it with the pre-stored correct password. Depending on the comparison result, the program either accepts the login request or rejects it. The program is depicted in Code 3. The C program first declares a character array named buf; then an integer variable named auth with an initial value of 0. It reads a user input data from the network, stores it in the buffer (see line 3), and then compares the stored input data with "abcd" - the correct password (see line 4). If the user-provided input data matches the correct password, the code will update auth to 1 (see line 5). Otherwise , auth remains in its initial state of 0. Finally, the program uses the state of auth to decide whether to call the function authentication_pass() or the function authentication_fail() (see line 6, 7 and 8).

```
1. char buf[5];
2. int auth = 0;
3. readData(sockfd, buf);
4. if(strcmp(buf, "abcd") == 0)
5.         auth = 1;
6. if(auth != 0)
7.         authentication_pass();
8. else
           authentication_fail();
```

Code 3. A simple password checker written in C

We are interested in reasoning about the set of statements that affect auth's state in line 6. Figure 2 shows the program dependence graph of the program from line 1 to 6.

As illustrated in the PDG, statements in lines 1, 2, 3, 4 and 5 are all involved in computing the value of auth. That is to say, the meaning of auth's value in line 6 depends on all the preceding statements . The meaning of these lines of code is *integrated* to determine auth's state. The user-provided input also influences auth. However, it does not directly determine the value of auth. It is processed by and integrated with these lines of code before influencing the final state of auth.

Now, consider the situation of an attack to see how it affects the integration of the program. The attack which we will examine is a form of data-only attack. The character array buf has a limited capacity of size 5. Suppose that the function readData (see

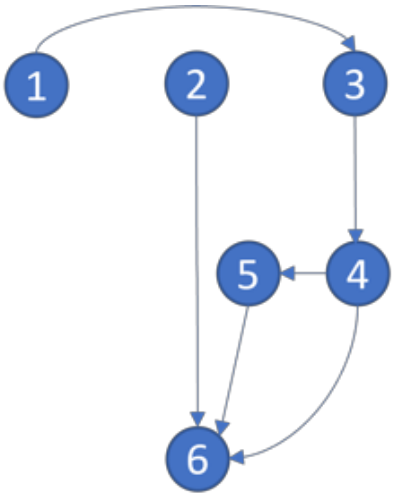

Figure 2. The program dependence graph of the program in code 3

line 3) does not check the size of the user input. A long user-provided input will overflow buf and *directly* overwrite the adjacent memory locations allocated for the variable auth. A simple way to attack the password checker program is to send it a user input consisting of five 'A' characters, appended by a 32-bit integer value of 1. The layout and contents of memory locations after the attack is illustrated as the following:

buf [ ]	auth
"AAAAA"	**0x00000001**

Here, 0x00000001 is the hexadecimal representation of the integer value of 1[2]. Due to the buffer overflow effect, auth is *directly* set to 1. This would allow the attacker to bypass the authentication without providing a correct password. This attack is data-only since it only modifies critical variables or data structure without injecting any executable code or altering the control flow structure of the program. The attack violates the functional integration of the target program. Since auth's value is directly set to 1 by the user input, the statements in lines 2, 4, and 5 are no longer relevant in determining the state of auth. That is to say, these lines of code are no longer *integrated* with the overall authentication process. The attack violates the original functional unity of the program and reduces it to the following simple form:

```
char buf[5];
readData(sockfd, buf);
if(auth != 0) pass();
else fail();
```

Code 4. The reduced form of the simple password checker during the attack

The attack also makes the user input have a more direct influence on auth. During the attack, the user input completely bypasses the code in lines 2, 4, and 5, having a *more* direct influence in determining the computational outcome. From a programmer's perspective, the solution to prevent data-only attacks is to have the readData function validate the user input's size before copying it to buf [11]. If the input size exceeds the size of the buffer, the program will throw an exception.

[2] The size of an integer depends on the CPU architecture. This article focuses on 32-bit Intel X86 CPU architecture, which represents an integer with a 32-bit size.

3.2. Data-only attack against a vulnerable FTP server

In this case study, we will use an example of a vulnerable file transfer protocol (FTP) server to illustrate more sophisticated data-only attacks. The example is first presented in Hu's work [10]. The purpose of this example was to show that data-only attacks are capable of expressing a rich set of computations. Code 5 shows the C source code snippet of the vulnerable FTP server:

```
1. struct server{ int total, cur_max, typ;} *srv;
2. int connect_limit = MAXCONN;
3. int *size, *type;
4. char buf[MAXLEN];
5. size = &buf[8]; type = &buf[12];
.....
6. while(connect_limit--) {
7.     readData(sockfd, buf); // stack buffer overflow
8.     if(*type == NONE ) break;
9.     if(*type == STREAM) // condition
10.        *size = srv->cur_max;
11.    else {
12.        srv->typ = *type; // assignment
13.        srv->total += *size; // addition
14.    }
15. ... (following code skipped) ...
16. }
```

Code 5. The code snippet of a FTP server written in C [10]

The server's primary purpose is to receive users' file transfer requests from the network and carry out different operations according to the type of request encoded in the received network packet. First, the program declares a pointer variable named srv of a structure type server (see line 1). The structure has three elements to describe the state of the server:

- Maximum size of the current user's file transfer request (int cur_max)
- The total number of bytes that the server has received so far (int total)
- The type of the current user request (int typ)

In line 2, a variable named connect _limit stores the maximum number of connections that the server can support simultaneously. In line 4, a character buffer named buf is declared with a maximum size of MAXLEN. It is used for the later storage of the incoming user input data. In line 3, the program declares two pointer variables : size which is defined as the reference to the size field of the user request and type which is defined as the reference to the type field of the request (see line 5).

In a while loop, the function readData receives a user request from a network packet and stores it in buf. The program examines the type and the size of the request by dereferencing the pointer variable type and size. If the type is "NONE", the loop will immediately terminate (see line 8). If the type is "STREAM", the program truncates the size of the user request (see line 10). If the request type is neither "NONE" or "STREAM ", the program updates the type of the current user request (srv->typ) and total size of received bytes (srv->total). It then proceeds to process the user request.

If we focus on the state of srv->total, it is clear that all the statements from line 1 to line 9 and line 11 affect its final state. They form a functional unity – an integrated group of statements — in the determination of the result. The user input also influences the state of srv->total. However, its influence is indirect. The input data is integrated with these lines of code to produce the result.

Let us examine how a data-only attack affects the functional integration of code. Suppose that the function readData in line 7 does not check the length of the incoming user input. Because of this vulnerability, if an attacker provides a long input that

exceeds the capacity of buf, she can cause a buffer-over and overwrite four variables: connect_limit, size, type and srv. With these variables under her control, the attacker can *invent* very expressive forms of computations that are not originally intended. Suppose that the system uses a linked list to store users' privilege information. Each node in the linked list uses an integer number to describe a user's privilege, with zero representing *low* access privilege and non-zero representing *high* access privilege. The following figure illustrates the linked list:

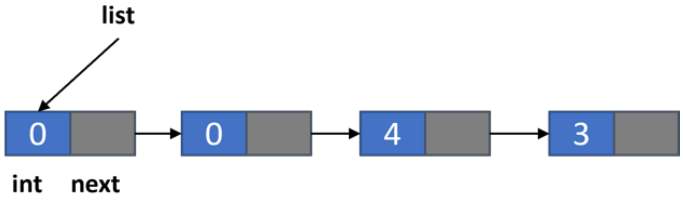

Figure 3. A linked list which stores users' privilege information

Here, a variable named list is a pointer that points to the first node of the linked list. That is to say, it stores the memory address of the first node. Suppose also that the attacker knows the address of the variable list and her goal is to escalate the access privilege of the first user stored in the linked list. She can use a long malicious input to overwrite the variables with the following parameters:

buf [ ]	type	size	connect_limit	srv
"AAAAAAAA... AAAAAA"	**&list**	**&addend**	**0x100**	**&srv-8**

The attacker *directly* sets the variable type to & list - the memory address of list, the variable srv to &srv-8 - its own memory address &srv minus 8. Since the assignment statement in line 12, srv->typ = *type, is equivalent to *(srv + 8) = *type, it is now transformed to the following assignment statement:

```
srv = list;
```

At this point, the attacker has successfully set srv to list. The code in line 13 srv->total += *size is transformed to the following addition operation:

```
*(list)  += addend
```

Here, addend is a variable that contains a non-zero integer. The operations above effectively escalate the first user's privilege stored in the first node of the linked list. The attacker also keeps the while running indefinitely by directly setting connect_limit to 0x100. With complete control of the loop's termination, the attacker can move to the other elements in the linked list and continue the subversion. In the next round of the while loop, she can use another long user-input to corrupt the variables with the following parameters:

buf [ ]	type	size	connect_limit	srv
"AAAAAAAA... AAAAAA"	**&STREAM**	**&list**	**0x100**	**list**

The attacker directly sets the variable type to &STREAM - the memory address of a variable that contains the same STREAM value. She also sets size to &list, srv to list. This forces the program execution to take the statement *size = srv->cur_max in line 10. Since the statement is equivalent to *size = *(srv + 4), it is now transformed to list = list -> next[3].

The attacker successfully moves to the next node in the linked list! As Hu argues in his paper [10], from the attacker's perspective, the while loop in this example is essentially a virtual CPU that dispatches different operations during its iterations . By alternating the above two exploits, the data-only attack can traverse all the nodes in the linked list, arbitrarily modifying any user's access privilege.

The above data-only attacks violate the functional integration of the code. Since the attacker directly sets the values of three variables: connect_limit, size, and type, the code in lines 2, 3 and 5 no longer participates in the computational process, the user-provided input data gains a more direct influence on the computational result by bypassing the code in lines 2, 3 and 5.

We have now completed the analysis of two data-only attack case studies. As illustrated above, Jon and Ziyuan's proposed thesis of two necessary conditions of secure computation is valid in both case studies. For a program to be secure, it must never allow the user input to bypass the code in the program to produce the computational outcomes. A secure program must also maintain its internal functional unity while processing the user input. It must never allow the user input to disrupt the unified form in the original algorithm. In other words, secure computation is an integrated computation.

It is worth noting that our thesis on the conditions of secure computation is congruent with the development of secure software engineering. One secure programming practice to prevent cyber-attacks due to malicious input data is input validation [14]. For example, buffer overflows mentioned above can be prevented by ensuring that input data does not exceed the limit of the size of the buffer in which it is stored[4] . From a Kantian perspective, the prevention technique essentially keeps input data from directly influencing the computational outcome. Consequently, and unwittingly, the approach protects the functional unity of the program from the potential disruptive power of the external influence. Since 2008, Microsoft has incorporated input validation as a critical secure coding practice in its secure software development life cycles [15].

4. The Implications to Computer Science Education

Computer science education has long adopted an instrumental view of digital technical objects such as software. The dominating technical normativity today is that of coding. This view is consistent with a culture that has alienated technical beings, reducing them to mere *usage*, as the word 'application' connotes. French philosopher of technology, Gilbert Simondon, describes the contemporary culture's limited view of technical beings in the introduction of *On the mode of existence of technical objects*: "Culture has become a system of defense against technics; now, this defense appears as a defense of man based on the assumption that technical objects contain no human reality. We should like to show that culture fails to take into account that there is a human reality in technical reality and that, if it is to fully play its role, culture must come to incorporate technical entities into its body of knowledge and its sense of values ... The most powerful cause of alienation in the contemporary world resides in this failure to understand the machine, which is not caused by the machine but by the non-understanding of its nature and essence ... " [12].

Due to such a narrowed view of technical being, software development tends to adopt a reductionist engineering attitude. Traditional computer science education encourages students to think of software as functions that take inputs and return desired outputs for users. In the process of solving a computational problem, students are trained to dissect the problem as a whole into parts, express each part in a computational way, then later find a way to connect them [13]. Little attention has been paid to the *internal structural necessity* of programs in their *own right*. As illustrated in the above case studies, such a reductionist, atomist approach has often led to insecure coding practices, resulting in vulnerable software.

In the age of cybersecurity, a new paradigm of technical development is needed. Software development should focus more on the unity of a program's internal structures as the necessary condition for its survival. The new paradigm recognizes that the structural and functional unity in a technical object has its origin in human reality. As Simondon points out, "what resides in the machines is human reality, human gestures fixed and crystallized into working structure" [12]. More precisely, for a computer program, its human reality is the mental process incarnated in coding. Following Kant's constructivist, anti-reductionist theory on the human mind, we argue that a computer program should 'inherit' the structural and functional unity from the mind - its creator.

---

[3] Here, we assume that the size of a pointer is 32-bit.

[4] Input validation is not limited to secure C programming. It is also used to prevent vulnerabilities in web applications written in other high-level programming languages such as PhP, Java.

Future software developers are the ones who are involved in maintaining the integrated form of a software system throughout its life-cycle as if *cultivating* a quasi-cognitive organism.

5. Conclusion

As the examples in this article demonstrate, a deep understanding of software security and vulnerability needs to consider the functional and structural unity within the program. The data-only attacks against vulnerable C programs show that a secure computation is always integrated. It integrates the input data with its internal algorithm and integrates the operations within the algorithm. A violation of these integrations violates the program's intended meaning and security. We believe that Jon and Ziyuan's thesis on the necessary conditions of secure computation is the right step toward a theoretical foundation of software security. By making connections between the Kantian theory of the human mind and computer programs, their thesis points to a non-reductionist approach to cybersecurity research. From a Kantian perspective, the attempts to create more secure computing environments would do well to take more cues from the nature of the mind. This study also has educational ramifications. Computer science education needs to reconnect to philosophical traditions and realize that the technocentric, reductionist approach to software development cannot provide a sustainable cyberinfrastructure in the future.

References

[1] Kushner, D. (2013). The real story of stuxnet, *IEEE Spectrum*, 50 (3) 48–53.

[2] Burmeister, J., Ziyuan Meng. (2021). Kant, Cybernetics, and Cybersecurity: Integration and Secure Computation." SYSTEMICS, CYBERNETICS AND INFORMATICS VOLUME 19 - NUMBER 4.

[3] Smith, R. (2019). Aristotle's Logic", The Stanford Encyclopedia of Philosophy (Summer 2019 Edition), Edward N. Zalta (ed.). Retrieved from https://plato.stanford.edu/archives/sum2019/entries/aristotle-logic.

[4] Wirth, N. (2008). Algorithms + data structures = programs. New Delhi, India: Prentice-Hall of India.

[5] Craig Kernighan, B. W. (2021). Wrapup on Software. In Understanding the digital world: What you need to know about computers, the internet, privacy, and security (pp. 117–118). essay, Princeton University Press.

[6] Tip, F. (1995). A survey of program slicing techniques. J. Program. Lang., 3.

[7] Ferrante, J., Ottenstein, K. J., Warren, J. D. (1987). The program dependence graph and its use in optimization. ACM Tra nsactions on Programming Languages and Systems, 9(3), 319–349. https://doi.org/10.1145/24039.24041

[8] Kant, I. (1965). Critique of Pure Reason (unabridged edition). St. Martin's Press.

[9] Kant, I. (1987). Critique of Judgment (1st ed.). Hackett Publishing.

[10] Hu, H., Shinde, S., Adrian, S., Chua, Z. L., Saxena, P., Liang, Z. (2016). Data-oriented programming: On the expressiveness of non-control data attacks. 2016 IEEE Symposium on Security and Privacy (SP). https://doi.org/10.1109/sp.2016.62

[11] Seacord, R. C. (2005). Secure Coding in C And C++ (1st ed.). Addison-Wesley Professional.

[12] Simondon, G. (2017). On the mode of existence of technical objects (C. Malaspina & J. Rogove, Trans.). Minneapolis, MN, United States: Univocal Publishing.

[13] Hunsaker, E., Hunsaker, Ottenbreit-Leftwich, A., Kimmons, R., & Enoch HunsakerEnoch Hunsaker is a Master's student at Brigham Young University. (1970, January 1). Computational thinking. The K-12 Educational Technology Handbook. Retrieved January 18, 2022, from https://edtechbooks.org/k12handbook/computational_thinking

[14] Seacord, R. C., Pethia, R. D. (2015). String. In *Secure coding in C and C++* (pp. 29–110). essay, Addison-Wesley.

[15] Sullivan, B. (2008, September). *Security briefs: SDL embraces the web*. Developer tools, technical documentation and coding examples. Retrieved April 23, 2022, from https://docs.microsoft.com/en-us/archive/msdn-magazine/2008/september/security-briefs-sdl-embraces-the-web