IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received March 25, 2022, accepted April 28, 2022, date of publication May 2, 2022, date of current version May 9, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3171922

Binary Exploitation in Industrial Control Systems: Past, Present and Future

QI LIU, KAIBIN BAO, AND VEIT HAGENMEYER
Institute for Automation and Applied Informatics, Karlsruhe Institute of Technology, 76344 Eggenstein-Leopoldshafen, Germany

Corresponding author: Qi Liu (qi.liu@kit.edu)

This work was supported by funding of the Helmholtz Association (HGF) through the Energy System Design (ESD) program.
We also acknowledge support by the KIT-Publication Fund of the Karlsruhe Institute of Technology.

ABSTRACT Despite being a decades-old problem, binary exploitation still remains a serious issue in computer security. It is mainly due to the prevalence of memory corruption errors in programs written with notoriously unsafe but yet indispensable programming languages like C and C++. For the past 30 years, the nip-and-tuck battle in memory between attackers and defenders has been getting more technical, versatile, and automated. With raised bar for exploitation in common information technology (IT) systems owing to hardened mitigation techniques, and with unintentionally opened doors into industrial control systems (ICS) due to the proliferation of industrial internet of things (IIoT), we argue that we will see an increased number of cyber attacks leveraging binary exploitation on ICS in the near future. However, while this topic generates a very rich and abundant body of research in common IT systems, there is a lack of systematic study targeting this topic in ICS. The present work aims at filling this gap and serves as a comprehensive walkthrough of binary exploitation in ICS. Apart from providing an analysis of the past cyber attacks leveraging binary exploitation on ICS and the ongoing attack surface transition, we give a review of the attack techniques and mitigation techniques on both general-purpose computers and embedded devices. At the end, we conclude this work by stressing the importance of network-based intrusion detection, considering the dominance of resource-constrained real-time embedded devices, low-end embedded devices in ICS, and the limited ability to deploy arbitrary defense mechanism directly on these devices.

INDEX TERMS Binary exploitation, industrial control systems, cyber incidents, attack and defense.

I. INTRODUCTION

The recently increased prevalence of headline-making ransomware attacks on industrial control systems (ICS) is particularly worrisome for the society. These attacks often leverage program exploits as a stepping stone for getting the initial access. A remote exploit is perhaps the infection method hardest to develop and hardest to prevent, since it normally does not need any human interaction, unlike other social engineering techniques including phishing emails. Binary exploitation has long been a major cybersecurity issue and continues to pose a serious security threat to individuals and organizations around the world. Although a significant amount of efforts are dedicated to it, defeating and preventing successful binary exploitation remains an open problem.

Historically, the main target of binary exploitation has been devices running on Intel x86 architecture, predominantly Windows operating systems (OS), and mostly in common information technology (IT) systems. However, the target of binary exploitation is about to gradually shift to ICS as a result of various factors. Firstly, with hardened mitigation techniques in up-to-date commodity OS, successful binary exploitation is still possible, but becomes much harder. Secondly, the recent rapid integration of industrial internet of things (IIoT) unintentionally increases the number of entry points into ICS. Thirdly, many embedded devices are not only lack of dedicated mitigation techniques because of intolerable performance overhead and/or unfulfilled hardware requirements, but also inherently insecure due to vendors' unjustified prioritization of functionality over security. We argue that this will inevitably make attackers go for easier wins.

Apart from human factors, in general, security concepts in ICS can be divided into a few classes: secure end devices, secure communication protocols, and secure network architectures and configuration. In order to really thwart malicious actors' attempts to break into a system, all these three security

The associate editor coordinating the review of this manuscript and approving it for publication was Weizhi Meng.

requirements must be guaranteed at the same time. To reach a secure network architecture and configuration, there are many practical recommendations to follow, e.g., [1], [2]. Those recommendations include robust network segmentation, e.g., deploying demilitarized zones (DMZ) and virtual local area networks (VLAN), strong access control, e.g., implementing authentication and authorization mechanisms, secure remote access, e.g., using virtual private network (VPN) or secure shell (SSH). They are easy to implement, i.e., even without assistance of any security professionals, and also prove to be very effective especially against not very skilled attackers.

In the past, the lack of these basic security measures gave even unskilled attackers an easy chance to break into a system and cause real damage. With raised security awareness, this weakest link has been largely reinforced, making it much harder to attack an organization. However, a very skilled attacker (group) can still break into a system, often by finding one or several software vulnerabilities and developing (zero-day) exploits accordingly. This brings us back to the first two security concepts, i.e., secure end devices and secure communication protocols, and stresses their importance. Because a vulnerability normally results from either design errors or implementation errors of the implemented OS, firmware, application software or communication protocols. Speaking of exploiting a communication protocol vulnerability, what is really exploited during an attack and the direct target for an attacker is the vulnerable program that implements this protocol and runs on an end device. Hence, we can say that the most important point of secure end devices and secure communication protocols is development and implementation of secure programs.

As it shows in the later sections of this article, in the early days, i.e., before 2010, ICS were easily exploitable largely due to terribly insecure network architectures and configuration, and of course, the presence of vulnerable programs was also critical. By analyzing the major cyber incidents in ICS in the last decade, we see that the sophistication of attacks increased very much, and threat actors seem to be willing to invest lots of efforts to find software vulnerabilities and develop exploits and malwares.

Hence, we argue that both writing secure programs, and deploying program exploitation countermeasures, are very important topics in ICS. When developing large programs, oftentimes we can hardly verify that the programs are absolutely secure, which makes binary exploitation countermeasures become a crucially important second defense line. However, while this topic generate a very rich and abundant body of research in common IT systems, it is not well addressed in ICS. Instead, most studies on ICS defense strategies still either focus on how to properly isolate and configure networks, e.g., prevent network traffic from untrusted sources, or mainly tackle situations in which it is assumed that attackers already took foothold in an ICS network, e.g., behavior-based intrusion detection. The initial infection methods are often not discussed, making it appear to be a topic exclusive for common IT systems.

In this work, we analyze binary exploitation in the context of ICS, which consist of general-purpose computers, embedded automation devices, low-end IoT devices etc. At first, we provide a preliminary introduction of binary exploitation in Section II, including its causes, types and consequences. To show how and where it is used in real-world attacks, we present in Section III a brief overview and analysis of major ICS cyber incidents leveraging binary exploitation in the last decade. Then, we argue in Section IV why binary exploitation will likely become a more serious problem in ICS. Next, we summarize the contemporary mitigation techniques on both general-purpose computers and embedded devices, respectively, in Section V. Lastly, we discuss heuristics-based detection and remote runtime attestation techniques in Section VI, and conclude this work in Section VII.

II. BINARY EXPLOITATION–A PRIMER

Although memory corruption bugs in general become harder to spot and much harder to exploit in modern software including operating systems, there have always been examples showing that binary exploitation is still possible in practice, and often easier than perceived by many security defenders. It has been long the case that every time security experts claimed that with the newly designed mitigation technique, binary exploitation is (almost) not possible anymore, and then found out that their assertions are falsified by subsequent attacks [3]–[5]. The corresponding exploitation techniques are often demonstrated and discussed by offensive security researchers in technical security conferences like Black Hat[1] or periodicals like Phrack Magazine.[2] For example, as Intel presented its hardware-assisted Control-flow Enforcement Technology (CET) [6] as a solution to prevent binary exploitation, it was probably believed that the problem is solved, only to find out there are still ways to achieve control-flow hijacking in the presence of this modern mitigation technique, as it was demonstrated by security researchers from the security company McAfee [7].

Undoubtedly, memory corruption exploits remain a clear, present and persistent danger to modern software, as they are still used as an entry point for the most recent and advanced attacks [5]. Modern software written in unsafe programming languages like C and C++ is particularly concerning due to the requirement of manual memory management, which is a very challenging task especially for large programs with millions of lines of code. Any mistake in handling memory buffers could lead to a memory corruption vulnerability, which consequently gives an attacker an opportunity to access the otherwise inaccessible memory location of a vulnerable program. Besides, even memory-safe languages like Java rely on virtual machines that are in turn implemented in C/C++

[1]https://www.blackhat.com/briefings/
[2]http://phrack.org

for performance reasons. As modern software becomes more complex, it is less likely to see the end of memory errors in the near future [8].

Moreover, in the world of embedded devices, memory errors in programs and examples of binary exploitation are also observed in the wild. That is to say, except from x86-based platforms, a variety of other hardware platforms can also be targeted by attackers, e.g., ARM [9], [10], MIPS [11], [12], PowerPC [13], [14], Atmel AVR [15], [16], SPARC [17], [18], Zilog Z80 [19], SuperH [20]. The prevalence of binary exploitation in ICS will likely continue to increase with accelerated integration of IIoT devices and augmented connectivity, and become a more serious problem. Compared to general-purpose computers, embedded systems are way less protected, e.g., because of a lack of hardware support required for modern defense techniques. Hence, it can be confidently assumed that attackers will go for easier wins by focusing more on embedded systems in ICS.

In order to help to understand the attack techniques and defense strategies discussed in the following sections, a preliminary introduction of binary exploitation is given in this section, including its causes, types and consequences.

A. CAUSES OF BINARY EXPLOITATION

The causes of a program being exploitable can be interpreted in several ways, and understood from different angles.

1) MEMORY UNSAFETY – A DANGEROUS CONDITION

A lack of memory safety in programming languages like C and C++ is arguably the root cause of exploitable vulnerabilities in software, and is deemed as the foundation of numerous attack vectors. The need for manual memory management makes it appear to be a never-ending problem, in particular, as modern software written in C and C++ becomes more complex. Memory safety is a program property that guarantees objects in memory can only be accessed with the corresponding capabilities, i.e., well-defined memory access [8]. That is to say, memory pointers should always point to valid allocated memory of the correct size and type, to prevent unintended and/or undefined behavior.

Memory safety includes spatial memory safety, temporal memory safety, and type safety. They ensure that pointer dereferences are restricted to data inside the corresponding memory object, a pointer can only reference a currently allocated memory object, and only pointers with the correct type can access related memory objects, respectively [8]. Normally, an exploit starts by triggering a memory error, in which it first makes a pointer invalid and then dereferences this pointer. A pointer becomes invalid, if it goes out of the bounds of its pointed object, or its pointed object is deleted, which is known as out-of-bounds pointer or dangling pointer, respectively. While dereferencing an out-of-bounds pointer raises a spatial memory error, dereferencing a dangling pointer results in a temporal memory error [4]. That is to say, the first case makes accessing neighboring memory objects possible, and the second case allows accessing memory location that was used for a deleted object, but now may contain another (unrelated) object.

It is worth noting that a single memory error can have cascade effect, i.e., invoking many more other memory errors. Memory corruption errors should be prevented by enforcing both spatial and temporal memory safety policies. In memory-safe languages, spatial and temporal memory safety are ensured by automatic bounds-checking of objects during memory access and garbage collection, respectively.

2) SOFTWARE DEVELOPERS – THE UNINTENDED TRIGGER

Despite the above-mentioned memory safety issue, C and C++ still remain popular among many software developers mainly for performance reasons. Programming in these memory-unsafe languages will often inevitably result in program bugs, especially when writing complex programs. It is arguably nearly impossible to write memory-safe code using C and C++ at scale due to the need of burdensome manual memory management, meaning that software written in unsafe languages is inherently error-prone.

When writing software, software developers may often implicitly or explicitly make questionable assumptions. Both design flaws and implementation flaws can appear in buggy programs written by software developers who are often not security experts. The aforementioned invalid pointers can be easily produced through mistakes like incautious pointer arithmetic. Historically, performance was probably the only key metric for many software programmers, mostly unconcerned about security. Today, the situation has greatly improved, with raised security awareness in general, proposed secure coding guidelines, compilers-enforced safety checks, and intense code review etc. However, some flaws still stay unnoticed, and some attack surface remains.

3) PROGRAM BUGS – THE OBSERVED AVALANCHE

We categorize frequently encountered program bugs into two classes, namely spatial safety-related bugs and temporal safety-related bugs. Whereas spatial safety-related bugs arise with a breach of memory objects' boundaries, temporal safety-related bugs occur when memory objects are accessed at the wrong time due to incorrectly tracked memory usage. An avalanche of both kinds of exploitable bugs can originate from using memory-unsafe languages.

a: SPATIAL SAFETY-RELATED BUGS

i) BUFFER OVERFLOW AND UNDERFLOW

As an instantiation of out-of-bounds write, a buffer overflow occurs when an input is allowed to be written beyond the boundaries of an allocated buffer during program execution, which corrupts the data of adjacent objects. Buffer overflow bugs range from the *classic* stack-based buffer overflow bugs, i.e., allowing copying a user-supplied over-sized input into a fixed-size buffer on the stack, to more recent and complicated heap-based buffer overflow bugs. In fact, today, heap-based buffer overflows are more dominantly exploited, and they are exploitable also due to intermingled user data and control data in the heap as in the stack [21]. Whereas stack-based overflows are often used to overwrite function return addresses, stack pointers, or stack base pointers, heap-based overflows are often used to overwrite function pointers [22], [23].

Another instantiation of out-of-bounds write is buffer underflows [24], a somewhat unfairly underestimated program bug due to too much attention given to buffer overflows. Whereas buffer overflows occur when memory access goes beyond the end of the targeted memory object, buffer underflows happen if the boundary at the object's beginning is breached.

ii) BUFFER OVERREAD AND UNDERREAD

As the incarnation of out-of-bounds read, both buffer overreads [25] and buffer underreads [26] can help attackers to reveal sensitive information like cryptographic keys or memory addresses to bypass security mechanisms, e.g., address space layout randomization (ASLR), which will be discussed in Section V. Similar to out-of-bounds writes, these bugs are typically caused by excessively incrementing or decrementing an array pointer in a loop to a memory location after or before the valid buffer, respectively, or by erroneous pointer arithmetic leading to a position beyond the bounds of the buffer.

iii) INTEGER OVERFLOW AND UNDERFLOW

Once the classic buffer overflows were largely avoided, integer overflows and underflows have become a serious security problem, and is frequently involved in binary exploitation [21]. An integer overflow appears when an integer arithmetic operation results in a value that is too large to be stored in the declared variable, triggering a wraparound and making the value become very small or negative. On the contrary, an integer underflow can turn a very small number into a very large number by producing a value that is smaller than the minimum allowable integer value during integer subtraction. Both integer overflows and integer underflows can be very dangerous, especially when they can be invoked by user-supplied inputs, and attackers exploit the math involved in resource management like calculating memory allocation sizes. A buffer overflow can be introduced by an integer overflow [27] or an integer underflow [28] during arithmetic operations of an array index, in which it causes less memory or much more memory to be allocated than expected, respectively.

iv) OFF-BY-ONE ERROR

An off-by-one bug [29] is introduced when a maximum or minimum value is incorrectly calculated to be one more or one less than the correct value. This often occurs when programmers fail to take into account that an array index starts at zero, making a loop operation iterate one more time than it should. Or they forget that a terminating null byte is automatically appended to a string variable taken as input by some C library functions like `strncat()`, making the input potentially corrupt the stack base pointer close to the allocated buffer and hence creating an exploitable condition.

v) TYPE CONFUSION

A type confusion bug [30], or type violation, appears when resources, e.g., objects in memory, are accessed using an incompatible type. That is, during program execution, an object is first initialized using one type, but then accessed using another type. Type confusion often arises due to incorrect downcasting of a base object to a subtype without proper checking, in particular in C++ programs with complex inheritance hierarchies. This can result in logical errors, because the object with a wrong type does not have the expected properties when accessed, and can be misinterpreted. Ultimately, it could lead to out-of-bounds memory access, if the allocated buffer is smaller than an object with the correct type, or if any data is misinterpreted as a pointer. Note that type-safe programming languages are practically also memory safe due to static or dynamic error checks, i.e., not only spatial safety errors but also temporal safety errors can be avoided.

vi) FORMAT STRING BUG

A format string bug [31] exists, when an externally submitted input string can be interpreted as a command by some functions like `printf()` that perform formatting and accept a variable number of arguments. As with many other bugs, format string bugs are exploitable due to directly accepting user-supplied inputs without validation or sanitization. A programmer may omit a format string specifier, such as `%s`, when using format functions in a program. An attacker can exploit this weakness by inserting multiple format string specifiers into a single input string, and supply it to a format function to subvert its normal behavior. When this input string is taken as an argument and parsed by a format function, it may mistakenly interpret this argument as instructions, ultimately leading to a so-called *write-what-where* condition, in which the attacker has the ability to write an arbitrary value to an arbitrary location. Fortunately, format string bugs are now rare thanks to easy discovery and elimination, hence no longer pose a serious threat to security.

b: TEMPORAL SAFETY-RELATED BUGS

i) USE-AFTER-FREE

The most common and infamous temporal safety-related bugs are use-after-free bugs, in which a previously freed memory location is accessed using a dangling pointer, i.e., a pointer that points to a deallocated object. This can happen due to some error conditions and/or when it is unclear which part of a program is responsible for freeing the memory location [32]. Sometimes, attackers can make a pointer become a dangling pointer by exploiting an incorrect exception handler, which deletes an object, but does not reinitialize the corresponding pointer [4]. Depending on the implementation of memory management system and the program code, the consequences of a use-after-free bug vary from data corruption to arbitrary code execution. After identifying or invoking a use-after-free bug, to exploit it, attackers could reallocate the freed memory position to themselves and insert a crafted object into it, before this memory position will be accessed through the dangling pointer. Later, the inserted object is accessed through the dangling pointer, tricking the program into executing attackers-specified code. Note that dangling pointers can point to not only heap but also other memory regions like stack, but heap-based attacks are the most concerning ones [33].

ii) DOUBLE FREE

Considered as a special case of use-after-free bugs, double-free bugs appear in programs in which a previously freed memory location is mistakenly freed again, instead of being accessed using a dangling pointer. This potentially leads to prematurely freeing a new object, or even worse, corruption of the program's memory management data structures. To reliably exploit it, sometimes attackers leverage a combination of double-free bugs and use-after-free bugs, such as [34]. Like use-after-free bugs, double-free bugs are also easy to create but difficult to spot during testing, because, on the one hand, the first `free()` operation and the second `free()` operation or the dangling pointer dereference are two separate events that may occur far apart in time and/or be invoked by code far apart in program space, and, on the other hand, the second `free()` operation or the dangling pointer dereference may not appear in every program execution [33], [35].

iii) INVALID FREE

Similar to double-free bugs, a `free()` operation can be wrongly used to introduce invalid-free bugs when trying to free an object in memory. But instead of freeing a memory object twice, a vulnerable program with invalid-free bugs calls the `free()` function to free an object on the heap with a pointer that does not point to the start of the object due to erroneous pointer arithmetic [36]. Or it frees an object not allocated on the heap at all, i.e., not allocated using heap memory allocation functions like `malloc()`, but rather automatically allocated on the stack as a local variable or allocated on the data segment as a global variable [37]. Depending on the implementation of the `free()` function, the consequences range from simply crashing a program to corrupting the program's memory management data structures, ultimately allowing attackers to modify critical program variables and gain arbitrary memory access abilities.

iv) MISMATCHED FREE

When trying to return memory resource to the system, it can go wrong not only by erroneous handling of the `free()` function as in invalid-free bugs, but also by using an incompatible memory deallocation function, producing a mismatched-free bug. For instance, in a C++ program, a memory buffer is first allocated with the `malloc()` function, but then released using the `delete` operation, instead of the `free()` function. When the memory management functions are mismatched, the consequences can be as problematic as those of above-mentioned temporal safety-related bugs [38].

v) USE OF UNINITIALIZED VARIABLE

Use of uninitialized variables [39], or use of uninitialized resources [40], may produce program bugs that cause undesired effects. This happens either when a variable is accessed before being assigned with a value, or when a variable is accessed before it should be reinitialized. If a variable is accessed, but not assigned previously, it can result in program crash or invalid memory access. If a variable is accessed prior to a necessary reinitialization, it can sometimes just return junk data, or, in other circumstances, leak sensitive information.

vi) WILD POINTER DEREFERENCE

Not only is use of uninitialized variables bad, but also use/dereference of uninitialized pointers. When pointers are created without necessary initialization, they are called wild pointers, because they simply point to arbitrary memory location, which can be valid or invalid. In many C and C++ programs, pointers are often declared as a wild pointer. If a wild pointer pointing to an invalid memory location is dereferenced, it could result in program crash. If the wild pointer happens to point to the start of an arbitrary function in memory, i.e., acting like a valid function pointer, this function may get executed. If attackers can somehow access the memory location to which the wild pointer points, this may result in arbitrary code execution, making it particularly dangerous [41].

vii) Null POINTER DEREFERENCE

To avoid wild pointers, programmers may initialize pointers to NULL when not possible to assign a more appropriate value at the point of declaration. However, if a pointer is initialized to NULL and dereferenced prior to being assigned a proper value, it also introduces a bug. Whereas a wild pointer has an undefined value, a NULL pointer keeps an implementation-dependent defined value, which is guaranteed not to be a valid memory address. Dereferencing a NULL pointer means trying to access a memory location that does not exist, typically resulting in a crash or exit. This may be less harmful than dereferencing a wild pointer, and such a bug is easier to spot during testing. However, NULL pointer dereference occurs sometimes due to race condition or other rarely encountered error conditions, making it less easier to detect [42]. Sometimes, attackers may leverage intentionally triggered NULL pointer dereference to bypass security mechanisms. Under exceptional circumstances, malicious code execution may also be possible, if NULL pointers are defined to hold the 0×0 memory address, and attackers happen to be able to trick the kernel into accessing it.

viii) RACE CONDITION

Another kind of bug arising only due to memory access at a bad time is race condition bugs, which exist in different concrete forms with varying reasons, such as while using signal handlers [43], in the middle of a switch statement [44], between time-of-check and time-of-use (TOCTOU) of a resource [45], and during context switching [46]. A race condition appears when two different *execution contexts* are able to affect a shared resource without proper synchronization, and hence interfere with each other. It is often caused by programmers thinking that system calls will execute *atomically*. But most system calls end up executing a large number of instructions and hence having longer than expected execution time, letting another concurrently running thread or process have a chance to intervene [21]. A race condition can have security implications, in particular when it takes place during execution of security-relevant code. It may be exploited by attackers to corrupt or modify important state information, potentially leading to program crash or access to resources otherwise unavailable to unauthenticated and unauthorized users. Besides, in some cases, race condition bugs are the cause of other temporal safety-related bugs like use-after-free, double-free, and NULL pointer dereference bugs.

According to the MITRE CWE ranking [47], out-of-bounds writes and out-of-bounds reads bugs are among the top three most dangerous software weaknesses in 2021, while use-after-free bugs are among the top 10 and NULL pointer dereference bugs are among the top 15. That is to say, spatial safety-related bugs are still the most severe ones, but the focus of memory corruption location has shifted from stack to heap [48]. Besides, it is worth noting that spatial safety-related bugs and temporal safety-related bugs are not unrelated to each other, and in fact, temporal safety-related bugs can often result in spatial safety-related issues. For instance, out-of-bounds writes and reads are sometimes enabled by use-after-free bugs.

In summary, the fundamental problem is that a lack of memory safety in programming languages allows software developers to introduce certain types of exploitable program bugs. Apart from invoking undefined behavior, i.e., mostly a stability problem, these program bugs can lead to leakage of sensitive information from memory and/or arbitrary code execution.

B. TYPES OF BINARY EXPLOITATION

By leveraging one or more of the above-mentioned memory corruption bugs, attackers can conduct different types of attacks. To alter a running program's behavior to the benefit of attackers, there are generally two types of exploitation, i.e., control-oriented exploitation (also known as control-flow hijacking [49]) and data-oriented exploitation (also known as non-control-data attacks [50]). In control-oriented exploitation, attackers seek to modify control data of a running program, i.e., the return address of a function call, or the code pointer of an indirect call or jump, to redirect the control-flow to execute attackers-injected code [49], attackers-selected library functions [51], or attackers-selected small code snippets [52]. In data-oriented exploitation, attackers intend to tamper with non-control but security-critical data, such as user ID, to bypass access control mechanisms and grant the root privilege [50], to leak sensitive information [53], or to indirectly influence the control flow [54], without even violating control-flow integrity polices in place. While this section also briefly mentions some defense methods, more detailed information about defense methods is provided in Section V and Section VI.

1) CONTROL-ORIENTED EXPLOITATION

The first known type of binary exploitation is control-oriented, which is still the one receiving most of the attention in attacks exploiting memory bugs today. Beginning with the stack-smashing attacks [49], the early generation of control-oriented exploitation centered on code-injection attacks. Only few years later, the focus shifted to code-reuse attacks with a number of increasingly sophisticated variants surfacing one after another. In any of these cases, in order to successfully exploit a program bug, an attacker needs to be able to first inject code into a running program or carefully pre-select existing code from that program, then redirect the control flow to the injected or pre-selected code, hence obtaining the control of the process. The injected code or the pre-selected code is simply a user input containing machine instructions or memory addresses of pre-selected instructions, respectively. By means of the program bug, part of the user input is used to change the target address of an indirect branch instruction, e.g., `ret`, `jmp`, `call`, essentially controlling which instruction is executed next. Regarding to whether the control-flow diversion is achieved through a return instruction, or an indirect jump or call instruction, control-oriented exploitation is further categorized into backward-edge control-flow hijacking and forward-edge control-flow hijacking, which differ very much in used methods for designing attacks.

a: BACKWARD-EDGE CONTROL-FLOW HIJACKING

Backward-edge control-flow hijacking attacks include all code-injection attacks [49] and some code-reuse attacks, i.e., return-into-libc attacks [51], [55], [56] and return-oriented programming (ROP) attacks [52], [57], due to their great reliance on the return instruction `ret`.

i) CODE INJECTION ATTACKS

Initially, in order to execute arbitrary code of attackers' choice, they solely relied on injection of simple code. The highly influential paper first to describe stack-smashing attacks [49] has spurred a considerable amount of research both in attacks and in defenses over time. The described stack-smashing attacks leverage a classic stack-based buffer overflow bug to inject code occupying the target buffer and its neighboring memory area. The injected code is carefully chosen and arranged, so that part of it overwrites a return address, i.e., a saved instruction pointer pointing to the next to-be-executed instruction, on the stack with a memory address of the attacker's choice. Hence when the vulnerable function returns, it points to the code injected by the attacker rather than returning to the caller that invoked the vulnerable function. The injected code that is executed after control-flow diversion launches a command shell, from which the attacker can control the compromised machine, and hence is called *shellcode*.[3]

Code-injection attacks are possible due to the fact that, on x86-based platforms with von Neumann architecture, code and data are stored in the same memory space, and x86 instruction set architecture itself does not distinguish between code and data, meaning that the same piece of raw bytes can be referred to as instructions or data depending on the context.

ii) RETURN-INTO-LIBC ATTACKS

Not long after code-injection attacks had grown their popularity, the W⊕X (write xor execute) policy [59] was introduced and has been widely enforced on modern operating systems, asserting that a memory page can be either writable or executable. By doing so, it marks all data regions, such as the stack, as non-executable, and hence effectively thwarts all code-injection attacks. In response, attackers have turned to code-reuse attacks, i.e., reuse of existing legitimate code in a vulnerable program or its linked libraries for malicious purposes. The first generation and also the simplest form of code-reuse attacks are the return-into-libc (RILC) attacks initially introduced in [51], which leverage functions from the standard C library libc.[4] This library is dynamically linked to nearly all Unix programs and loaded in their memory spaces. Thus, instead of injecting malicious code into the stack by means of a user input, attackers inject malicious data interpreted as code pointers pointing to exported functions in libc during execution.

The first generation of RILC attacks [51] used a single-call to the `system()` function of libc, with specified function arguments, to spawn a command shell. As a single function call is used to perform a specific operation, it alone has limited ability or expressiveness. Afterwards, the second generation of RILC attacks [55] appear to be more advanced, i.e., capable of arbitrarily chaining multiple libc functions together and invoking them one after another for performing a more powerful operation. A method called *esp lifting* [55] was introduced to glue multiple functions by means of common short instruction sequences like `pop esp; ret`. The stack pointer `esp` serves as a "virtual" program counter. By overwriting the return address in the current stack frame with the memory location of such a sequence, an attacker can move the stack pointer to the next stack frame, hence chaining multiple functions together [56]. That is to say, the attacker can first populate the stack with carefully crafted and arranged malicious function call frames containing function entry points and parameters, then use the stack pointer to redirect the execution to the next function of the attacker's choosing.

However, the second generation of RILC attacks still can hardly support conditional branching, an essential operation for a system to be Turing complete, meaning that they cannot freely alter the control flow during program execution. Subsequently, the third generation of RILC attacks [56] were presented to prove that Turing completeness is achievable with RILC attacks, and thereby attackers can perform arbitrary computations in the target program through selected and arranged function calls. Some commonly available specific functions, dubbed *widgets*, can be chosen and misused to induce conditional branching and hence arbitrary behavior in the target program.

iii) RETURN-ORIENTED PROGRAMMING

As the x86-64 architecture becomes more prevalent, RILC attacks get more difficult to conduct, because most function arguments are passed into CPU registers instead of the stack, due to increased space available in registers in the x86-64 architecture. Moreover, removing certain functions from libc, as a defense strategy, may restrict the capabilities of RILC attacks. To overcome these restrictions, attackers moved to a more general and advanced attack technique called return-oriented programming (ROP) [52]. ROP extends code-reuse attacks greatly, and the name results from the fact that it is the return instruction triggering the processor to continue executing what attackers put or specified in memory, like in code-injection attacks and RILC attacks. Whereas the building blocks of RILC attacks are libc functions, ROP attacks take as building blocks only short instruction sequences ending with a return instruction, dubbed *gadgets*. Such gadgets are discovered offline through static analysis of a vulnerable program, and allow an attacker to carry out arbitrary computations possible with x86 code, i.e., Turing complete, when appropriately arranged [52].

Each gadget performs a well-defined specific task, e.g., loading a value into a register, reading a value from memory, some arithmetic operation, or a conditional branching. To find these gadgets is not difficult in a large code base like libc. In particular on x86-based platforms, finding appropriate gadgets is made even easier by the fact that instructions are of variable length and unaligned memory access is supported, meaning that unintended instruction sequences can be found and acquired by starting from an offset of some intended instructions. That is, every x86 program contains many unintended instruction sequences which can be leveraged by attackers. ROP is essentially all about finding useful gadgets located anywhere in memory, and connecting them sequentially to perform desired operations. The location of every gadget is written into the stack, and the return instruction

[3]The term shellcode is nowadays used to refer to exploit payload, irrespective of whether it launches a shell [58].

[4]Note that a C runtime library provides many useful low-level routines, e.g., wrappers for system calls, that can be called by a compiled C program during execution. In most of research papers the GNU libc is used for demonstration, but the presented techniques should be also applicable with C runtime libraries implemented in other OS like Microsoft Windows OS, as per [52], [56].

at the end of each gadget helps to get the next gadget to be executed, effectively allowing them to be chained. A perhaps more concrete way to view ROP is that the selected gadgets form a "virtual" instruction set that can be used to write a program of arbitrary complexity, and the stack pointer acts as a "virtual" instruction pointer [57], [60].

b: FORWARD-EDGE CONTROL-FLOW HIJACKING

Since the advent of ROP attacks, defenders have presented a variety of potent defense methods. Driven by the insight that ROP has a great reliance on the return instruction, such defense methods [61]–[63] either detect abnormal use of the return instruction, e.g., too frequent or unusual according to calling convention, or prevent compilers from producing code containing return instructions. This, though, has prompted attackers to come up with other code-reuse variations, extending the approach from solely relying on return instructions to capable of leveraging any indirect branching instruction. Hence attackers further generalized code-reuse attacks to include forward-edge control-flow hijacking attacks such as jump-oriented programming (JOP) [64], [65] and call-oriented programming (COP) [60].

i) JUMP-ORIENTED PROGRAMMING

The term of jump-oriented programming was coined by [65], while independent techniques leveraging the jump instruction `jmp`, instead of the return instruction `ret`, were first presented in [66] and [64]. JOP attacks can bypass above-mentioned defense approaches against ROP attacks, because such obvious inherent characteristics of ROP attacks like violation of calling convention, which can be taken as a detection indicator, do not exist in JOP attacks. Given that certain instruction sequences behave like a return instruction, it is proposed in [64] that the `ret` instruction can be replaced with the `pop x; jmp *x` sequence. Such a sequence can be acquired from a target program, and is called *trampoline*. In this case, gadgets are certain selected instruction sequences ending in an indirect jump instruction, whose target is the trampoline. That is, the trampoline is responsible for chaining short instruction sequences and redirecting execution, essentially behaving like the "glue" and thereby making JOP also Turing complete.

However, the techniques in [64] still rely on the stack to steer the control flow among gadgets, and the `pop x; jmp *x` sequence is rather rare [64], [65], i.e., not always present in a target program. Another JOP approach [65] does not suffer from such limitations. A special kind of gadget is introduced to chain other gadgets ending with an indirect jump, and to govern the control flow without relying on the stack. This kind of gadget is called *dispatcher gadget*,[5] in order to differentiate it from other gadgets performing certain primitive operations such as arithmetic operations or conditional branching, which are considered as functional gadgets. The control transfer between functional gadgets is achieved through maintaining an internal dispatch table by the dispatcher gadget and ensuring that the jump instruction at the end of each functional gadget will always give the control back to the dispatcher gadget.

[5]Sometimes it is also called *gadget dispatcher* like in [67].

ii) CALL-ORIENTED PROGRAMMING

Not only can an indirect jump instruction take the role of the return instruction in code-reuse attacks, but so does an indirect call instruction, as it is proposed in call-oriented programming [60]. Similar to ROP and JOP, arbitrary computations with malicious purposes are performed in COP by finding, chaining, and executing some useful gadgets ending with an indirect call instruction. Even when some control-flow integrity (CFI) [68] policies, a powerful defense technique discussed in detail in Section V, are enforced, COP may still be possible, given that indirect call instructions must be always allowed to jump to certain functions [60].

c: HYBRID EXPLOITS

Although aforementioned code-reuse attacks prove to be Turing complete, the enforcement of CFI policies can make it much more challenging to successfully conduct these type of attacks. That is, when a CFI policy is enforced in a target program, attackers have to find *CFI-compliant gadgets*, which are usually much scarcer in the target program. This prompted attackers, again, to adapt their techniques, leading to hybrid exploits, in which they combine code-reuse attacks and code-injection attacks, such as [69] and [60]. In fact, hybrid exploits are shown to be more dominant than pure code-reuse attacks in real world [70]. Recall that the reason why attackers turned from carrying out code-injection attacks to code-reuse attacks is that the enforcement of the W⊕X policy forbids injected code from being executed. However, the W⊕X policy only states that a memory page cannot be both writable and executable at the same time, but still allows a memory page to be first only writable and afterward only executable. This lets attackers come up with a multi-stage attack procedure. They first write shellcode into a buffer in a memory page that is now only writable. Subsequently, they launch a code-reuse attack, in which they use gadgets to invoke a function call or system call to change the permissions of that memory page from only writable to only executable, effectively bypassing the W⊕X protection. Then, the shellcode can be executed, because neither the W⊕X policy nor the CFI policy will prevent it. Note that a CFI policy is enforced only in existing code, not in injected code, as CFI checks are inserted in a binary program either during compilation by a compiler, or through binary rewriting following static analysis.

d: AUXILIARY TECHNIQUES

To pave the way for a successful exploitation, except requiring one or more program bugs, attackers also need to make use of some supporting techniques. As mentioned previously, heap is now the front-line of the battle in memory [21], [48].

When exploiting heap-based vulnerabilities, attackers usually employ a number of techniques to reliably and successfully perform the exploitation, such as heap-spraying [71], [72], heap-Feng-Shui [73], [74].

Due to inherent randomness of memory allocation in the heap, and especially when the effective defense technique ASLR is employed, it is not easy for an attacker to reliably redirect the execution to attacker-supplied code. To increase the likelihood of jumping to the correct memory location of attacker-supplied code, and hence facilitate a successful exploitation, the attacker makes lots of copies of the malicious code and "sprays" them into yet unoccupied memory space of the heap, thereby dubbed heap-spraying. Next, the attacker tries to overwrite a code pointer on the heap making it point to the attacker-supplied code.

The heap-spraying technique is effective, but it alone might not be sufficient to underpin a reliable heap-based exploitation [73]. Because, on the one hand, the state and layout of the heap are dynamic and hard to predict, meaning that the overwritten code pointer is not guaranteed to always contain the attacker-controlled data. On the other hand, there might not always be enough free space left in the heap for being sprayed. Therefore, for a successful exploitation, it is often necessary to control the heap state and layout before triggering a program vulnerability. Heap-Feng-Shui is a technique for manipulating heap layouts by first finding some primitives to interact with heap allocators, and subsequently carefully assembling them through allocating or freeing objects of selected sizes [73], [74].

e: AUTOMATIC EXPLOIT GENERATION

Carrying out a successful code-reuse attack is usually a complex task, especially when defense mechanisms are in place. In order to reduce time and human efforts, attackers gradually managed to automate various steps of designing a code-reuse attack, e.g., automatic discovery of gadgets, ultimately leading to automatic exploit generation (AEG). A variety of algorithms, techniques or tools are designed to semi-automatically or fully-automatically find program bugs and craft corresponding (defense-resilient) exploits, respectively.

Early attempts of AEG target stack-based program bugs. Automatic patch-based exploit generation proposed in [75] can generate an exploit for an unpatched program without first knowing the vulnerability, when provided with the patch as a guidance. In [76], control-flow-hijacking exploits are automatically generated also using *dynamic taint analysis* paired with an algorithm that generates candidate exploits from a constraint formula. Though, the first end-to-end fully automatic exploit generation, i.e., from automatic vulnerabilities disclosure to generating exploits that spawn a command shell, is considered to be the approach presented in [77], which makes use of *symbolic execution* for exploring program paths and checking their exploitability. However, these earliest AEG solutions do not assume any defense mechanisms are deployed in the target systems. To enhance the practicability and usability of AEG, another AEG approach is demonstrated in [78], which automatically generates ROP payloads. It is applicable even in the presence of *loosely* implemented defenses mechanisms W⊕X and ASLR, resulting in hardened exploits. As an extension, during automatic ROP payload generation, the presented solution in [79] can also deal with gadgets containing pointer dereferences. Furthermore, AEG solutions are extended to include the one working solely on raw binary programs without debugging information or source code [80].

Another group of AEG approaches are designed to abuse heap-based program bugs, which are more common in modern software, and generally more difficult to exploit, e.g., due to involved interaction with the heap manager. Comparing to stack-based AEG, heap-based AEG is much harder to realize, and yet remains an open challenge [74], [81], [82]. That is, existing heap-based AEG solutions can only automate one or more steps of exploit generation, but still fall short of a generic end-to-end fully AEG, even in the absence of modern defense mechanisms. Nonetheless, some advancements were made in recent years by representative heap-based AEG techniques. For instance, [83] presented a framework for discovering exploit primitives[6] in heap managers using symbolic execution, and generating usable exploits. Another framework demonstrated in [81] focuses on automated exploitability assessment of heap-based program bugs, in which it makes use of both *fuzzing* and symbolic execution to explore exploitable states from a crashing input, i.e., an input which triggers a program bug, and to generate functioning exploits. Note that transforming a crashing input into an exploitable state is in general not an easy task. Similarly, based on *bounded model checking* and symbolic execution, another tool is designed in [84] to assess the exploitability of several heap allocators given a memory corruption bug. Besides, the first AEG for heap overflows in language interpreters, e.g., PHP and Python interpreters, is proposed in [82], which utilizes fuzzing-based input generation, rather than relying on symbolic execution. Furthermore, automated heap-Feng-Shui is proposed in [74] for automatically manipulating heap layouts, hence to facilitate heap-based AEG, as many heap-based program bugs are exploitable only given certain heap layouts.

2) DATA-ORIENTED EXPLOITATION

As control-oriented exploitation gets harder, mostly due to widely deployed defense mechanisms like, in particular, CFI, data-oriented exploitation gains popularity thanks to its immunity to CFI checks. Data-oriented attacks do not violate the control-flow graph (CFG) of a target program during its execution amid control-flow transfers, but still can cause significant damage and pose a considerable threat [50], [53], [67]. This advantage is provided by the fact that some

[6]Exploit primitives are basic operations like a write-what-where operation, and they are considered as exploitation building blocks used to achieve arbitrary code execution.

program data is not directly used in control-flow transfer instructions, i.e., never loaded into the instruction pointer register, but can yet indirectly influence program's execution to the benefit of attackers. As per [67], data-oriented exploitation can be grouped into two categories, namely direct-data-corruption attacks [50], [53], and data-oriented programming (DOP) attacks [67].

iii) DIRECT-DATA-CORRUPTION

Direct-data-corruption attacks represent the first generation of data-oriented exploitation, in which attackers directly manipulate a variety of non-control, yet security-sensitive data [50] like user ID or configuration data, in order to mislead a program's execution for malicious purposes. A real-world example is the Heartbleed[7] exploitation, which allows attackers to steal cryptographic keys and other credentials from a vulnerable device, hence to eavesdrop on its communications or impersonate it. Early direct-data-corruption attacks [50] are relatively straightforward, and can succeed with only a single *data-flow edge* compromised.

To further demonstrate the power of data-oriented exploitation, a systematic approach called *data-flow stitching* [53] is developed to automatically generate data-oriented exploits when provided with memory corruption bugs. Data-flow stitching aims to chain multiple existing data-flow edges in a data-flow graph to create new unintended data-flow paths (from the perspective of normal program execution). A further generalization of data-oriented attacks is proposed in [54], which introduces the notion control-flow bending (CFB). In CFB, modifications of both control data and non-control data are allowed as long as the enforced CFI policy is not breached. That is, CFB refers to any attack, in which the entire execution trace looks legitimate with respect to the control-flow graph.

iv) DATA-ORIENTED PROGRAMMING

The aforementioned data-oriented attacks suffer from limited expressiveness, that is, they cannot provide attackers the ability to perform arbitrary computations in a vulnerable program's memory space. Besides, they can be easily prevented by enforcing some access control policies in memory, which largely restrict unauthorized access to security-critical data. With the advent of the second generation of data-oriented exploitation, i.e., data-oriented programming [67], attackers can design Turing-complete data-oriented attacks, also without relying on corrupting specific security-critical data. Similarly to ROP, in DOP, the first step is to discover useful gadgets, i.e., short sequences of instructions used to perform the basic operations like arithmetic operations and conditional branching. Note that data-oriented gadgets are, by definition, CFI-compliant, and hence generally more difficult to find. Next, gadget dispatchers need to be identified, and are used to chain those functional gadgets, in order to achieve desired functionality or computations. As mentioned previously, a gadget dispatcher is a special kind of gadget which selectively and consecutively invokes functional gadgets, allowing arbitrary recursive computations. An example of a typical data-oriented gadget dispatcher can be acquired from code which implements a loop statement [67].

Another type of advanced data-oriented attack is demonstrated in [85], in which the term block-oriented programming (BOP) is introduced. BOP gadgets are comprised of entire basic blocks, instead of instructions. The authors presented a framework for automatically constructing defense-resilient exploits against programs hardened with CFI policies, surpassing previous DOP attacks in the sense that they heavily rely on manual analysis.

According to whether a bug in user-space programs or operating system kernels is exploited, we can categorize it as user-space program exploitation or kernel exploitation, respectively. Whereas user-space program exploitation represents the majority of binary exploitation, due to easy accessibility etc., kernel exploitation is less common and more difficult, but very appealing to attackers. Real-life examples like EternalBlue and EternalRomance [86] show that kernel exploitation poses a serious and realistic threat. Note that commodity OS kernels are almost exclusively written in the memory-unsafe C language, providing attackers a better chance to conduct runtime attacks against kernels. Except above-mentioned software-based exploitation, attackers can also carry out sophisticated hardware-based exploitation, which is particularly dangerous, e.g., the infamous Rowhammer—a hardware bug—exploitation on various hardware platforms [87]–[89].

C. CONSEQUENCES OF BINARY EXPLOITATION

Depending on specific program bugs, targeted applications, defense mechanisms in place etc., the consequences of binary exploitation vary from the severest ones—arbitrary code executions—to the mildest ones—harmless program crashes of non-critical programs.

1) ARBITRARY CODE EXECUTION

The ability to execute arbitrary code in a target program's memory space gives attackers the complete control of it, and the freedom of causing any possible damage to the system if the controlled program has the highest privileges. This is arguably attackers' first objective. They can force the running program to execute at their bidding by invoking arbitrary system calls, and behave differently from the programmer's intention. Besides, they can leverage credentials stored in the victim system to further attack other devices. Even if the controlled program does not have all the permissions in the system, a limited set of system calls, i.e., confined code execution, may also allow attackers to do enough damage.

2) INFORMATION LEAKAGE

Sometimes, an exploitable bug cannot directly grant an attacker the right to take over the system and execute any code, but rather the ability to reveal some sensitive

[7]https://heartbleed.com/

information, e.g., by copying it to an output stream. There are two types of information, which are of special interest of attackers, i.e., security credentials and memory layout-related information. Leaking security credentials like password hashes or cryptographic keys helps attackers to easily pass access controls, and eventually take over control of the target system. Disclosing memory layout-related information, e.g., the memory address of a specific object, let an attacker reliably assess the memory layout of a target process. As a result, it renders some deployed defense mechanisms like ASLR useless, and helps the attacker to further exploit another memory bug (or sometimes the same bug), then leading to arbitrary code execution.

3) PRIVILEGE ESCALATION

If an exploited program itself does not have the root privilege (in Unix parlance) or the administrator right (in Windows parlance), or the leaked credentials do not provide all the permissions in the system, it may not satisfy an attacker's goal. The attacker may try to further leverage another bug to corrupt security-critical data [50], which can result in elevated privileges.

4) DENIAL OF SERVICE

Crashing a target program, as a result of binary exploitation, is not uncommon during real-life attacks, not to mention during the test of an exploit under development. Due to various factors, even a well-tested exploit may not work every time on every system containing the corresponding program bug. Causing this kind of denial-of-service (DoS) is sometimes intended, but at other times rather unintended. For instance, in a past ICS cyber attack, attackers exploited a software vulnerability for causing DoS, as we will discuss it in Section III-F. Nonetheless, more often than not, attackers would choose to make better use of a vulnerability like in the first three cases, when possible.

D. EXPLOIT-BASED MALWARE

Traditionally, malware is encapsulated inside some executable needed to be manually executed, and requires no exploitation of program bugs. In today's cyber attacks, the target of an attacker is usually not a single computer, but rather a network with lots of computing devices. In order to save time and have a higher impact on the target by attacking as many devices as possible in a short period of time, an attack process is now largely automated by designing exploit-based malware, as it shows in Section III. In this way, user interaction is often not a prerequisite, and unnecessary intervention from command-and-control (C&C) servers can also be reduced to evade network-based intrusion detection. Besides, for systems that forbid outbound Internet access, this would be the only way to attack them. That is to say, nowadays, an exploit is mostly used in combination with some malware, which serves as payload following an exploit and/or has integrated exploit(s). In some cases, an attack can be completely automated, meaning that all the functionality needed to sabotage a system is embedded directly in the corresponding exploit-based malware.

When binary exploitation is used as an initial infection vector, it often employs one of the two most typical forms, i.e., direct remote exploitation, and phishing emails with an attachment containing an exploit. Direct remote exploitation mostly targets network service programs, i.e., programs implementing communication protocols, e.g., Server Message Block (SMB) protocol and Remote Desktop Protocol (RDP). An exploit embedded in an attachment of a phishing email typically targets a user application program, which is used to open and process the corresponding attachment, e.g., Microsoft Word document.

III. A BRIEF OVERVIEW OF MAJOR ICS CYBER INCIDENTS

In this section we aim to provide a brief overview and analysis of major ICS cyber incidents leveraging binary exploitation in the last decade. The past real-world examples demonstrate that binary exploitation is a very important launch pad for malware. For every cyber incident, we primarily discuss the consequence/damage of the attack, the technical attack procedure, and the exploits involved.

Note that this may only be a small sample of cyber incidents, because, according to MITRE [90], the vast majority of discovered incidents are not reported publicly. It is also worth mentioning that there are ICS cyber incidents that are not listed in this paper, because, to the best of our knowledge, there are no evidences published yet, which indicate that binary exploitation is involved. This could be due to a lack of forensic evidences in victim networks. For instance, as per US ICS-CERT [91], [92], the initial access vectors in more than a third of total incidents reported in 2014 and in 2015 were not identified, because of a lack of detection and monitoring capabilities within the compromised networks.

Given that cyber incidents are usually characterized and represented by the corresponding malware, and often even named after these malware, and in order to avoid lengthy title for each incident, we also use the malware name as the title for each incident.

A. STUXNET

The attack against Iranian nuclear centrifuges uncovered in 2010 [93] is widely perceived as a watershed moment in ICS security. This is not only due to the unprecedented use of four zero-day exploits [94], but also due to the willingness and the ability of the attackers to study the industrial devices and the physical processes. The corresponding malware Stuxnet is deemed as the first confirmed example of ICS-tailored malware. According to the security firm Symantec [94], it includes a Windows OS rootkit that hides its binaries, and the first ever programmable logic controller (PLC) rootkit that hides modified code on PLC.

It is widely believed that the initial access was introduced by removable USB drives. These removable drives contain a specially crafted Microsoft Windows shortcut file, which exploits the then zero-day vulnerability CVE-2010-2568[8] (patched with the security update MS10-046[9]) in Windows Explorer. When a removable drive is plugged into a Windows computer and viewed, the contained shortcut file will automatically execute the first hidden file in the removable drive. This hidden file will hook some important DLL[10] API and replace the original code of exported functions with some files-checking code. Then it loads the second hidden file from the removable drive, which contains the main Stuxnet DLL. After the main DLL is extracted into memory and executed, Stuxnet is installed.

In the targeted system, many computers were non-networked, and the data exchange between them were done with removable drives. Hence one of Stuxnet's propagating ways is also through removable drives. Whenever an uninfected removable drive is inserted into a already compromised computer, Stuxnet will copy itself and its supporting files to the removable drive [94]. Other Stuxnet's spreading methods include exploiting the then zero-day vulnerability CVE-2010-2729 (patched with MS10-061) in Windows Print Spooler, and exploiting the vulnerability CVE-2008-4250 (patched with MS08-067) in Windows Server Service. Through the vulnerability in Windows Print Spooler, which allows a file to be written to a system directory of a vulnerable computer, Stuxnet spreads itself easily in a local area network (LAN), and executes itself in the infected machines. By means of the SMB protocol and the vulnerability in Windows Server Service, Stuxnet is able to copy itself to unpatched remote computers through sending a malformed path string that allows arbitrary code execution [94].

The other two then zero-day exploits are used for privilege escalation. The first one is used to exploit a vulnerability in Windows Task Scheduler, which makes the main DLL file run as a new process with Adminstrator rights. The second privilege escalation exploit targets the CVE-2010-2549 vulnerability (patched with MS10-073) in the Windows Win32k.sys kernel-mode device driver [94].

B. DUQU

A global cyber espionage especially targeting ICS was disclosed in 2011 with the discovery of a new piece of malware, which uses many of the same techniques from Stuxnet. The malware, named Duqu and discovered by the CrySyS Lab [95], targeted a number of organizations around the world for stealing information, which can be used for planning another destructive attack against them.

The initial access was likely introduced by spear-phishing emails with a Microsoft Word document as attachment containing a then zero-day kernel exploit [96]. This exploit abuses the CVE-2011-3402 vulnerability (patched with MS11-087) in the Win32k.sys kernel-mode device driver of Windows OS, and helps to install Duqu onto targeted computers unbeknownst to users. A malicious embedded TrueType font file can allow code auto-execution in kernel mode, because of erroneous font-parsing of the Windows kernel-mode driver [97].

When such a Word document is opened on a computer, the exploit is triggered, and will first check if the computer is already compromised by looking into the Windows Registry. If the computer has already been compromised, it exits. Otherwise the shellcode contained in the exploit will decrypt a driver file and an installer DLL file from within the Word document. The execution is then passed to the extracted driver file, which is signed with a stolen legitimate digital certificate allowing it to bypass default restrictions on unknown drivers and common security policies. The driver file injects the installer DLL into a running benign services.exe process. Then the installer DLL gets executed, and starts extracting other components from it. These components are injected into other benign processes to hide Duqu's activities, and to bypass security programs. An information-stealer program is installed by Duqu to collect system information [96].

C. FLAME

Another complex global cyber espionage targeting ICS especially in Middle Eastern countries was revealed in 2012. The corresponding malware, named Flame, was identified, analyzed and disclosed by multiple entities including Iranian National CERT, Kaspersky Lab [98] and the CrySyS Lab [99]. This modular malware shares many characteristics with the above-mentioned malware Stuxnet and Duqu, but its modules are different. Comparing with Duqu, Flame is not only a lot more complex, but also much more widespread, making it redefine the notion of cyber espionage [98].

It is assumed that the initial access is achieved by exploiting the CVE-2010-1879 vulnerability (patched with MS10-033), and a then zero-day exploit is used to compromise then fully-patched Windows 7 OS [98]. In order to spread itself to non-infected computers, Flame includes two exploits used by Stuxnet, i.e., for the CVE-2010-2729 vulnerability in Windows Print Spooler and for the CVE-2010-2568 vulnerability in Windows Explorer [98], [99]. The first exploit is used to spread itself through network shares, and the second is used with removable drives for non-networked computers.

D. HAVEX

The third confirmed widespread cyber espionage followed in 2013 with the ICS-tailored malware Havex. It first targeted defense and aviation companies in the US and Canada, and then shifted its focus to US and European energy firms [100]. The stolen information gives the attackers the ability to more easily sabotage operations in the targeted critical infrastructures.

According to the security firm Symantec [100], the attacker group has at least three infection tactics for getting initial access to victim systems, i.e., spear phishing emails with

[8]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2568
[9]https://docs.microsoft.com/en-us/security-updates/securitybulletins/2010/ms10-046
[10]Dynamic-link library (DLL) is Windows OS' implementation of the shared library concept.

malicious PDF attachment, watering hole attacks, and supply chain attacks. In the watering hole attacks, the attackers first compromised a number of energy-related websites, and injected an iFrame[11] into them. This iFrame redirects visitors to another compromised website hosting the LightsOut exploit kit,[12] or its updated version. The LightsOut exploit kit abuses vulnerabilities in the Java Runtime Environment (JRE) component in Oracle Java SE 7 and in Internet Explorer browser to deliver the Havex malware, and in some cases the Karagany malware [100].

In the supply chain attacks, the attackers compromised a number of legitimate software packages from three different ICS equipment providers, by inserting the Havex malware into them [100]. When users download these trojanized software packages from ICS vendor websites, they would then unknowingly download the malware themselves, allowing the malware to bypass some security measures.

E. BlackEnergy

In 2015, a major hours-lasting blackout in Ukraine was caused by a complex targeted attack consisting of a set of actions, e.g., disrupting electricity distribution, destroying IT systems, flooding call centers, and inhibiting incident response [101]. The attackers deployed a piece of malware dubbed BlackEnergy 3, which is the second update of its original version BlackEnergy appeared in 2007.

To deliver the malware, various initial infection vectors were employed [102]. The first kind is spear-phishing emails with an executable file as attachment, which is disguised with a Microsoft Word document icon tricking receivers to click it and execute it. The second kind is spear-phishing emails with some Microsoft Power Point slides as attachment, which contain hidden objects exploiting the CVE-2014-4114 vulnerability (patched with MS14-060) in Windows Object Linking and Embedding (OLE). The last one is spear-phishing emails with a Word document as attachment, which contains hidden objects exploiting the CVE-2014-1761 vulnerability (patched with MS14-017) in Microsoft Word and Office Web Apps.

After the initial infection and delivery of the core module, a number of plug-ins, i.e., DLL files, are downloaded and successively executed [103]. For instance, a file-system operations plug-in is used for early reconnaissance. A network scanner plug-in explores the network perimeter, and disguises itself as a Windows service program. Credentials are stolen with some password stealer and key logger plug-ins, and then used to help attackers to move to other computers in the network.

After the attackers have gained the required credentials, they start infecting Windows servers using BlackEnergy 2, i.e., the direct predecessor of BlackEnergy 3. BlackEnergy 2 appears as a kernel-mode driver. In order to disable the driver signature check in Windows OS and evade detection, a tool called DSEFix[13] is used, which exploits the CVE-2008-3431 vulnerability [103]. Besides, BlackEnergy 2 also contains exploits for human-machine interface (HMI) applications from ICS vendors like Siemens and General Electric. Hence an Internet-connected HMI can easily give attackers an initial access and a foothold in the ICS central location, for information gathering and conducting further attacks [104].

F. INDUSTROYER

Another large-scale cyber attack on Ukraine's power grid took place in 2016 with a piece of even more advanced malware dubbed Industroyer. This malware is considered as the first malware designed and deployed specifically to target power grids. It has the ability to directly control switches and circuit breakers, showing that the malware creators have a good knowledge of ICS and communication protocols used in power grids [104]. The initial infection vector remains unknown [105].

This modular malware has a core component being the main backdoor, which receives commands from its remote C&C server via HTTPS, and controls all other components [105]. A launcher is installed by the main backdoor as a program responsible for launching several payloads and a data wiper module, which are DLL files and export a function named Crash for the launcher. The payloads partly implement the communication protocols specified in the standard IEC 60870-5-101, IEC 60870-5-104, IEC 61850, and OPC Data Access specification, respectively [105]. By leveraging the intended functionality in these protocols, the attackers are capable of enumerating and possibly taking over control of all remote terminal units (RTU), intelligent electronic devices (IED), and open platform communications (OPC) servers etc. The destructive data wiper module is used in the final stage of an attack, which hides its traces and makes recovery difficult, e.g., by deleting all files.

Besides, the CVE-2015-5374 vulnerability was exploited by sending crafted packets to Siemens SIPROTEC digital relays rendering them unresponsive [105]. This denial-of-service (DoS) attack against SIPROTEC protective relays was performed by the attackers right after opening circuit breakers and removing operators' visibility into system operations through the data wiper module [106].

G. WannaCry

In May 2017, numerous organizations across the world including railway companies, manufacturing companies, and energy firms, were attacked by a cryptoworm known as WannaCry [107]. This fast-spreading malware continued to infect thousands of computers globally for at least two years [108]. WannaCry is a piece of ransomware, i.e., malware containing an encryption plug-in and a ransom note.

[11]An iFrame is a HTML element allowing embedding documents, videos, and interactive media within a web page.

[12]The LightsOut exploit kit contains exploits for the CVE-2012-1723, CVE-2013-2465 and CVE-2013-1347. https://www.mcafee.com/enterprise/de-de/threat-center/threat-landscape-dashboard/exploit-kits-details.lights-out-exploit-kit.html

[13]https://github.com/hfiref0x/DSEFix

The initial access into a network can be achieved by selecting random IP addresses across the Internet and exploiting vulnerable Internet-facing Windows computers. It then spreads itself rapidly in local networks. It does this by exploiting the CVE-2017-0144 vulnerability (patched with MS17-010) in Windows SMB Server. Before infecting new computers through an infected computer, it checks if they are already compromised. If not, it proceeds to use the infamous SMBv1 exploit EternalBlue [86] to drop its payload to those vulnerable computers.

According to Kaspersky Lab [107], during the investigation, WannaCry was able to infect computers in ICS due to the following facts: (1) use of dual-homed computers, e.g., engineering workstation,[14] acting as a bridge between an enterprise network and a local industrial network; (2) a lack of properly configured secure network perimeter devices between segments of an industrial network, which are connected through the Internet via VPN channels[15] due to large distances; (3) use of devices, e.g., USB modem, for setting up direct mobile Internet connections for computers on an industrial network, bypassing the network perimeter.

H. NotPetya

Another cryptoworm affecting ICS globally appeared only one month later, in June 2017. According to Kaspersky ICS CERT [109], at least half of the companies attacked by this malware, dubbed NotPetya, are manufacturing and energy firms. The disruption caused by NotPetya resulted in a halt of operation. Despite appearing to be a piece of ransomware, encrypted files cannot be restored, even after victims have paid the ransom. NotPetya is more like a piece of data-wiping malware pretending to be ransomware [109].

One initial infection method is conducting watering hole attacks, i.e., first compromising websites presumably frequently visited by targeted victims and starting infecting victims from there. Once NotPetya has infected one computer in a network, it starts to spread itself to other computers inside the same network using several propagation methods. Two infamous exploits are used for this purpose, i.e., EternalBlue and EternalRomance exploits [86], which abuse the CVE-2017-0144 vulnerability and the CVE-2017-0145 vulnerability in SMBv1 Server, respectively. These exploits are used for different Windows OS versions [110]. Using the same exploits, NotPetya is able to propagate from corporate networks to industrial networks. The exploitation process can be observed in network traffic, i.e., a series of specifically crafted SMB packets.

[14]Note that an engineering workstation is a device used to write program and configuration data for PLC. Engineering workstations using Windows OS are commonly seen in ICS.

[15]Note that a VPN channel is used to prevent unauthorized eavesdropping and data tampering in the data transfer channel. It cannot prevent a computer from being attacked, when another computer at the other end of the channel is already compromised.

I. BAD RABBIT

In October 2017, another large-scale ransomware campaign took place, and affected organizations across Russia and eastern Europe, e.g., the transportation sector in Ukraine [111]. This, again, demonstrates that the historically ransomeware-free ICS networks are now a very focused target of ransomware.

According to Kaspersky Lab [112], the ransomware, dubbed Bad Rabbit, also used a watering hole attack as initial infection method. But in this case, no exploit was used to gain initial access. Instead, when a target victim visits a compromised legitimate website, the malware is downloaded and pretends to be an Adobe Flash installer. This tricks the victim into executing the malware. After infecting one computer, it starts to spread itself within the corporate network and to the industrial network. It does so by using a modified version of the EternalRomance exploit, which abuses the the CVE-2017-0145 vulnerability in SMBv1 Server. Whereas in NotPetya this exploit is used to install the DoublePulsar[16] backdoor, Bad Rabbit employs it to overwrite a kernel's session security context, which enables it to launch remote services [114].

J. TRITON

Another watershed moment in ICS security occurred with an attack against a petrochemical plant in Saudi Arabian reported in December 2017 [115]. This is considered as a watershed moment, because the malware, named Triton, distinguishes itself from other ICS-tailored malware like Stuxnet in two ways. Firstly, Triton targets a safety instrumented system (SIS), which serves to put a critical infrastructure in a safe state or shut it down to prevent any physical harm. A SIS is deemed as the last defense line for critical infrastructures. It is often totally isolated from all other networks and hence hard to reach. Secondly, sabotaging a SIS not only causes economical loss for operators, but also directly risks human lives.

In this attack, the Triconex safety controllers made by Schneider Electric were specifically targeted [14]. Triton consists of two parts, i.e., a malicious Windows program compiled from a Python script and a malicious program for the safety controllers. The attackers first compromised a Windows computer within a SIS network, and the malicious Windows program is executed on the compromised computer. This program leverages a custom implementation of an internal TriStation protocol, through which the compromised computer connects to a safety controller. The malicious program for the safety controllers includes an injector and a backdoor, and both of them are downloaded into the safety controller by the compromised computer.

The injector executes automatically on the safety controller. After it verifies that the controller can be

[16]DoublePulsar backdoor is a sophisticated SMB backdoor, and the primary payload used in SMB and RDP exploits in the FuzzBunch framework [113].

TABLE 1. Summary of exploitation purposes of exploit-based Malware.

Malware	Exploitation for initial access	Exploitation for privilege escalation	Exploitation for detection evasion	Exploitation for lateral movement	Exploitation for denial-of -service
Stuxnet	✓	✓	✗	✓	✗
Duqu	✓	✓	✗	✗	✗
Flame	✓	✗	✗	✓	✗
Havex	✓	✗	✗	✗	✗
BlackEnergy	✓	✗	✓	✗	✗
Industroyer	✗	✗	✗	✗	✓
WannaCry	✓	✗	✗	✓	✗
NotPetya	✓	✗	✗	✓	✗
Bad Rabbit	✗	✗	✗	✓	✗
Triton	✗	✓	✓	✗	✗
VPNFilter	✓	✗	✗	✗	✗

compromised, it exploits a then zero-day vulnerability in the device firmware[17] for privilege escalation [115] [14]. Before injecting the backdoor into the firmware memory region, it disables a firmware consistency check for detection evasion. The backdoor is enabled by the injector through changing a jump table entry to point to the injected code. It gives the attackers the capability of reading, writing memory and executing arbitrary code on the controller [14].

K. VPNFilter

In 2018, at least half a million public-facing routers and network attached storage devices across the globe were infected by a piece of malware known as VPNFilter [116]. This concerning malware has the capability of spying on traffic routed through infected devices, and has dedicated code for targeting ICS. It is also destructive in that it can make infected devices unusable. Besides, unlike most other internet of things (IoT) malware, VPNFilter is capable of maintaining its persistence even after a reboot [116], [117].

As per security researchers in Cisco Talos [116], since all of the affected devices have publicly known vulnerabilities, it is very likely that the initial infection vector is the exploits targeting those vulnerabilities. This multi-stage, modular malware is used both for cyber espionage and for destructive attacks. The stage 1 module works as a backdoor providing a persistent foothold, and multiple redundant mechanisms are employed to connect it to its C&C server. The stage 2 module serves to gather and exfiltrate data, and overwrite a critical portion of the device's firmware making it inoperable. Additional functionality is provided by multiple stage 3 modules, which include Modbus/TCP traffic monitor, exploits deliverer, man-in-the-middle (MitM) component capable of intercepting and manipulating network traffic [116]–[118].

1) EXPLOITATION PURPOSES

According to the MITRE ATT&CK for ICS Matrix [119], the purposes of binary exploitation can be categorized into four groups: exploitation for initial access, exploitation for privilege escalation, exploitation for detection evasion, and exploitation for lateral movement. All these exploitation purposes can be observed in the cyber incidents mentioned above. Besides, we observed and add another exploitation purpose after analyzing these cyber incidents, i.e., exploitation for denial-of-service. It is worth mentioning that in this section we analyze the purposes of binary exploitation from a network point of view, whereas in Section II-C the attention is solely on a single host.

A summary of exploitation purposes of above-mentioned exploit-based malware is given in Table 1. Note that in the case of protocol vulnerability, the same exploit could be used both for getting initial access and for lateral movement, e.g., the EternalBlue exploit, making the corresponding exploit-based malware spread very fast and hence especially dangerous.

IV. ATTACK SURFACE ON ICS

The ICS landscape is very diverse in terms of involved physical processes, devices, software platforms, hardware platforms, communication protocols, vendors etc. The increasing complexity and connectivity in these heterogeneous environments inevitably result in an increased attack surface. Devices commonly found in ICS encompass engineering workstations, HMI, OPC servers etc. running commodity OS like Windows OS and based on x86 hardware platforms, and specialized embedded devices such as PLC, RTU, IED, smart sensors running a (tiny) embedded OS or even bare-metal applications (i.e., without an OS), and based on a variety of hardware platforms like ARM, MIPS, PowerPC, AVR etc. Evidences showing that all these platforms are potentially susceptible to binary exploitation are provided in Section V. As discussed in Section III, only one Internet-facing vulnerable device in an ICS suffices to grant an attacker a strong initial foothold in its network and a great opportunity to further attack other devices in the network. Any compromised device could be utilized to provide rogue services, such as disrupting network operation by masquerading as another device, eventually leading to safety issues. To promote a better understanding on why binary exploitation has become a more serious problem in ICS, we provide in this section primarily a brief overview of history of ICS network structure and a succinct analysis of its attack surface transition.

A. A BRIEF HISTORY OF ICS

Looking back in the history of ICS, we have witnessed a transition from the very early air-gapped, well-isolated, hardware-centric systems to some more connected, advanced, well-segmented systems with distinct IT and OT (operational technology) networks based on the Purdue Model [120], and then a drastic shift to systems with converged IT and OT networks, driving the connectivity to a much higher level and eventually breaking the (boundaries introduced in the) Purdue Model.

[17] Note that Triconex firmware versions 10.0–10.4 running on a PowerPC processor are vulnerable to Triton. The newer Triconex safety controllers use ARM processors, meaning that a different version of exploit would be required [14].

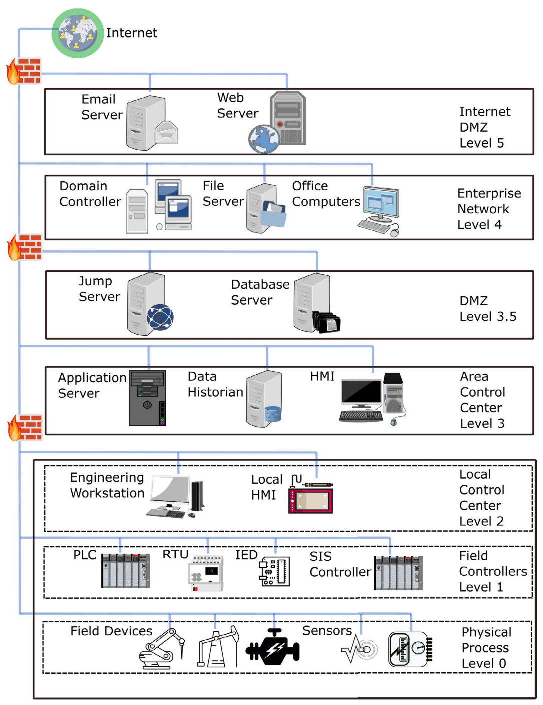


FIGURE 1. ISA's representation of the purdue model.

1) HARDWARE-CENTRIC

Initially, physical security was more of a concern than digital security in ICS due to their air-gapped nature, i.e., physically completely isolated from other systems [121]. They were hardware-centric in the sense that there was not much software involved, but primarily hardware systems. The control was done mainly through electrical, pneumatic or other physical signals instead of digital signals, and the reliance on human operator in the field was very high. Besides, a reconfiguration of a control system would require re-wiring of those hardware devices. Even after the introduction of remote control via a point-to-point connection, the increased exposure was very low, because the physical medium for communication was normally owned and exclusively used by the same company and isolated from other networks.

2) THE PURDUE MODEL WAS BORN

In order to fulfill the requirement for rapid information exchange between production and business segments in an industrial sector company, and efficiently manage the relationship between its ICS and enterprise resource planning (ERP) system, a hierarchical model called the Purdue Model [120] was created in the early 1990s. This model has a high impact on the standard ISA-95 from the International Society of Automation (ISA),[18] and was extended by ISA-95, which results in a clear-cut, well-segmented network structure with six levels, i.e., from level 0 to level 5. This network structure became highly influential for ICS design and quickly found a broad acceptance in industry. Subsequently,

[18]https://www.isa.org/standards-and-publications/isa-standards/isa-standards-committees/isa95

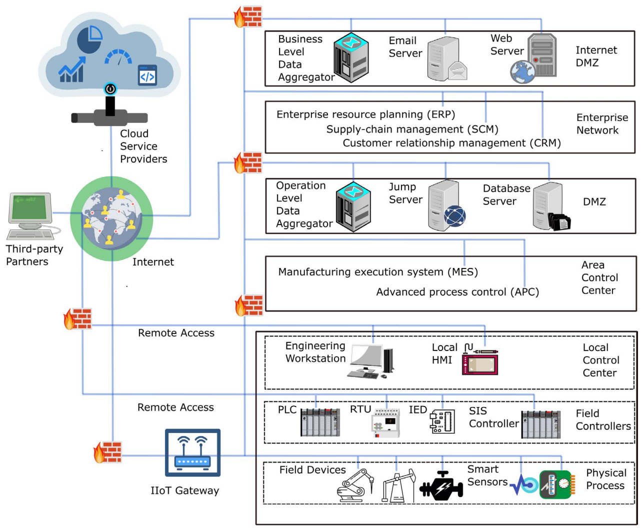


FIGURE 2. A modern IIoT-based ICS network.

for security purpose, the ISA-99 / IEC 62443 standard[19] proposed a new level, which is a demilitarized zone (DMZ) and is positioned between level 3 and level 4, hence often called level 3.5. The ISA's representation of the Purdue Model is shown in Figure 1.

The DMZ is deployed to better control information flow between the operation network and the enterprise network, or to re-introduce the air gap for the OT network by removing the direct communication link between the OT network and the IT network. In some ICS environments, there may be an extra DMZ or firewall between any two adjacent levels. However, a true air gap no longer exists, as a DMZ or firewall does not fully stop information exchange between the OT network and the IT network, but rather only mediate the communication between them, not to mention when a DMZ or firewall is not properly set up or configured.

3) THE PURDUE MODEL IS "BROKEN"

As new technologies progressively find their way into ICS, it becomes, little by little, clear that the Purdue Model has been "broken", or at least the strict boundaries between those levels start being bypassed, replaced, or disappearing [122]–[124]. This rigid, hierarchical model is greatly challenged by the proliferation of IIoT and cloud services, in which new untrusted third-party entities/levels are incorporated in the network structure, all kinds of industrial devices are equipped with Internet connectivity and smart sensors, and data is transmitted directly or through an IIoT gateway outside the company to a cloud service provider for big data analysis.

The reasons why we may now think that the Purdue Model was inevitably torn apart can be summarized into two key points. Firstly, it is due to performance and management

[19]https://www.isa.org/standards-and-publications/isa-standards/isa-standards-committees/isa99

requirements. Increased connectivity and intelligence in industrial devices (down to the level 0 of the Purdue Model) allow improved data sharing both in quantity and in quality within the OT network and to the enterprise network, which results in a more detailed view of individual devices and a more comprehensive overview of the entire industrial ecosystem, enables continuous monitoring of the performance and conditions of these devices and the ecosystem, and eventually simplifies the management. Secondly, it is caused by business requirements. The data captured from everywhere in the ecosystem is analyzed with big data technologies, and used for predictive maintenance, optimization of operation conditions, fine-tuning of production processes, and better market strategies. All these compelling benefits ultimately contribute to a huge cost saving [123], [125]. For instance, offshore oil and gas organizations can reduce more than a third of their losses due to unplanned downtime with an agile information sharing, analysis, and response process [125].

However, ICS operators often do not have the expertise and resources for big data analytics, hence need to transmit their data to third-party cloud computing providers for timely predictive analytics. The growing desire for real-time data from (remote) industrial devices, and for timely control and configuration of these devices, drives the convergence of IT and OT networks, and thereby creates a modern IIoT-based ICS network like the one in Figure 2.

B. ATTACK SURFACE TRANSITION

Due to raised security awareness in ICS in general, and more commonly applied good security practices, the attack surface should have been minimized. However, from the perspective of network entry points, the attack surface on ICS has inevitably steadily expanded with the transition from hardware-centric systems to more connected, hierarchical systems, and then to highly networked IIoT-based systems. The newly introduced initial access points into ICS include any Internet-accessible hardware device, in particular when the OS or firmware, application software or associated libraries running on it have unpatched program vulnerabilities.

1) PRE-2010

Before the attack wave on ICS in the 2010s started with the Stuxnet, attacks on ICS were possible mainly due to a lack of security awareness, misconfigured network perimeter devices, few to no security countermeasures in the network and on end devices. At that time, attacks weren't really sophisticated, only required scanning the Internet for finding ICS devices, for which default or guessed credentials would suffice to log into them, and using them to access and manipulate other devices in the network, potentially causing serious damage very easily. As a little more technical as it could get, attackers would have to exploit a vulnerability to gain the initial access into an ICS, yet would not necessarily need to develop an exploit by themselves. As it shows in the study [126], many Internet-connected vulnerable ICS devices contained vulnerabilities, for which the exploits were already present in online exploit databases.

2) THE LAST DECADE

As discussed in Section III, the last decade has seen a number of ICS attacks with increased complexity. The entry points were either an Internet-facing device in an enterprise network or an ICS device with remote access functionality. The attack surface became bigger, especially because more and more hardware products and software solutions from various vendors were integrated in an ICS, and a direct or indirect remote access to the network is required by those third-party vendors. This can also be reflected in the fact that more watering hole attacks and supply chain attacks were observed in the last decade. Once attackers find an initial access point into a system, they may quickly pivot into a critical path in the network and reach all kinds of devices. This is made possible because of the reality that ICS networks have been increasingly upgraded to TCP/IP-backed routable networks.

Real-world examples analyzed in Section III show that the "IT conduit" [127] in both IT and OT networks is frequently exploited and traversed in order to reach the final targets. For instance, Stuxnet leverages an engineering workstation to get into a target PLC. BlackEnergy exploits an Internet-connected HMI to get an overview of the target ICS, and facilitate disrupting or destroying all its target devices. Industroyer can take control of an application server to directly manipulate or damage its target RTU and IED. An example of a complete attack path is shown in Figure 3, which is adapted from [128], [129].

3) AFTER-2020

With continuous integration of IT technologies into OT networks, and the convergence of IT and OT networks, ICS networks become increasingly connected. Remote access directly to ICS is enabled for many participants like technicians, maintenance engineers, third-party vendors and contractors. According to a survey by SANS in 2019 [130], about 12% of ICS are directly connected to the Internet, 23% of ICS are indirectly (through a DMZ) connected to the Internet, and around 10% are directly connected to a third-party private infrastructure. These numbers should increase in the near future. In some cases, the attackers' initial access into an ICS is achieved through a compromised third-party vendor or contractor.

As a large amount of data is collected from field devices, and pushed directly into the cloud, points of exposure spread further down to the lowest level of the Purdue Model. The more interconnected devices there are, the more targets attackers can have and the quicker they can compromise devices. IIoT gateways clearly become an important target for attackers. Like other third-party partners, cloud service providers are not immune to cyber attacks. A security breach in a cloud service provider may also provide an initial access

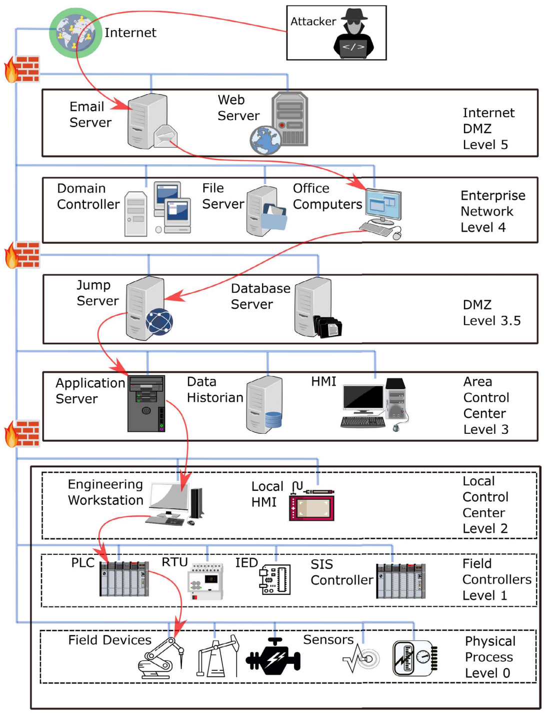


FIGURE 3. An example of a complete attack path.

into an ICS network. Besides, the growing popularity of embedded web servers in ICS field devices or controllers may provide additional attack vectors. All these added entry points in modern ICS, as shown in Figure 2, greatly expand the attack surface.

V. ATTACK TECHNIQUES AND DEFENSE TECHNIQUES

Since ICS consist of both general-purpose computers, e.g., engineering workstations, data servers, and a variety of embedded systems, e.g., PLC, IED, IoT gateways, in this section we discuss the attack techniques and mitigation techniques in both fields, respectively. In the arms race between attackers and defenders till now, it seems that the attackers have *always* managed to successfully exploit computer systems despite various defense mechanisms in place. The reason for it is that these adversaries not only keep exploring and exploiting vulnerabilities in software, but also have evolved to exploit weaknesses in most of defense techniques, i.e., weaknesses in their design or in their implementation. Furthermore, a hard truth is that, to secure a system, the developer or defender must eliminate *all* vulnerabilities and consider *all* possible attacks, while to attack a system only *one* flaw may suffice. This makes it particularly challenging or impossible to guarantee or formally prove the security of a system. All defense mechanisms, at the end, may only serve to reduce the attack surface to a limited extent, or raise the bar to a certain degree.

A. ON GENERAL-PURPOSE COMPUTERS

Defense techniques are often classified into program integrity-based, W⊕X-based, randomization-based, flow integrity-based, memory sanitization-based, and memory isolation-based ones. Each type of defense technique introduces some security policy enforced against a specific exploitation stage [4], which serves to prevent attackers from corrupting any code or data (cf. Section V-A1, Section V-A5), from executing injected code (cf. Section V-A2), from finding correct memory addresses (cf. Section V-A3), from accessing critical memory regions (cf. Section V-A6), or from tampering program execution flow (cf. Section V-A4). Once a security policy is violated at runtime, the process may be forced to terminate. The reference monitor of a security policy can be implemented either with hardware support, e.g., W⊕X, or purely by embedding some integrity checks into program code, e.g., flow-integrity checks [4].

1) PROGRAM INTEGRITY

Program integrity-based defense mechanisms prevent certain memory locations of a running program from being tampered, including code integrity policies, return-address integrity checking, and code-pointer integrity policies. These defense mechanisms also provide an important support for other more advanced defense mechanisms, e.g., CFI. For instance, CFI could be easily bypassed, if memory regions containing code can be modified, meaning that CFI checks can be removed by attackers [8].

a: CODE INTEGRITY

A code integrity policy prevents attackers from modifying code regions in memory by setting the corresponding memory pages as not writable. Existing hardware support renders this policy effective and efficient. However, it cannot be fully enforced for programs that have features like self-modifying code, dynamic library loading or just-in-time (JIT) compilation. The necessity of changing, loading, or generating code at runtime leaves attackers a small time window to corrupt code on a writable page [4], [8].

b: RETURN ADDRESS INTEGRITY

If code cannot be overwritten, an attacker may instead try to corrupt and control some security-critical data in the memory, in particular a return address in the stack, which determines what to be executed next when the called function returns. To prevent this from happening, several defense schemes were proposed, e.g., stack canaries [131], shadow stack [132], zipper stack [133].

As one of the first solutions against the classic stack-based buffer overflow attacks, a stack canary [131] is a secret value inserted between the return address and the local variables in a call frame. The integrity of this secret value is checked before the return instruction. A tampered value signifies memory corruption in the stack and will result in process termination. Despite being popular and widely implemented due to low performance overhead, stack canaries can prohibit only the classic stack-based buffer overflow attacks, but not attacks capable of overwriting a return address without touching the secret value before it.

A shadow stack [132] improves the defense effectiveness by moving to directly ensure the integrity of a return address. A shadow stack is a separate, safe memory region, onto which a copy of every saved return address is pushed during a function call. Before a return instruction, the saved return address on the stack is compared with its copy on the shadow stack. A mismatch will cause process termination. Yet, the shadow stack's own safety cannot always be guaranteed, leaving attackers a harder but still possible way to successfully attack a system by corrupting a return address.

To overcome the reliance on safe memory isolation, zipper stack [133] instead makes use of message authentication codes (MAC) function for protecting return addresses. It authenticates a return address by a chain structure using MAC, and provides a more robust return address protection. However, like the previous two defense schemes, this one is also inherently ineffective against attacks that do not need to manipulate return addresses and/or have no reliance on the stack, e.g., heap-based code-reuse attacks.

c: CODE-POINTER INTEGRITY

Whereas above-mentioned techniques intend to only ensure the integrity of return addresses on the stack, another defense mechanism called code-pointer integrity (CPI) [134] aims to guarantee the integrity of all code pointers including indirect jump targets, indirect call targets and return addresses. A code-pointer integrity (CPI) policy can be enforced to prohibit (both stack-based and heap-based) control-oriented exploitation (cf. Section II-B1), by protecting the integrity of all code pointers as well as data pointers used to access code pointers. These sensitive pointers can be identified by a compiler, and then stored in a protected memory region at runtime, which is isolated from the rest of the memory address space [134]. The access to the protected pointers is only given to the program code that is supposed to have the need. However, the memory isolation sometimes relies on information hiding, which is fundamentally susceptible to information disclosure [135]. Furthermore, CPI does not prohibit data-oriented exploitation (cf. Section II-B2).

2) W⊕X

Instead of preventing the corruption of code or data like program integrity-based defense mechanisms, another type of defense allows memory corruption and code injection, but prevents its execution. The non-executable (NX) data policy [59], or data execution prevention (DEP) policy [136] in Windows parlance, makes all memory pages containing data non-executable. Together with the code integrity policy, this creates the widely enforced W⊕X (writable xor executable) policy, meaning that a memory page is either writable or executable. However, as mentioned in Section II-B1, the W⊕X policy can be rendered useless by code-reuse attacks or hybrid exploits.

3) RANDOMIZATION

A fundamental requirement of all types of binary exploitation is the ability to find the correct memory addresses of target code and/or data. For efficient execution, the memory addresses of some code and data were deliberately fixed, meaning that they stay the same at every program execution on every host. This makes it straightforward for attackers to find the correct memory addresses with binary analysis, and thereby to successfully exploit a system. Hence, to make binary exploitation much harder and more unreliable, a variety of randomization-based defense techniques were proposed.

a: ADDRESS SPACE LAYOUT RANDOMIZATION

A well-accepted and widely deployed countermeasure is address space layout randomization (ASLR) [137], which generates random start addresses of code segments, (e.g., libraries,) and data segments, (e.g., stack and heap,) in memory address space, for each program execution. This increases the resiliency against both code-injection and code-reuse attacks, by making it more difficult for attackers to reliably find the location of injected code or existing program code. Trying to access a wrong memory address will likely result in a segmentation fault and failed exploitation.

However, only randomizing the start address of a segment means that, once attackers find this address, they can find the address of a target object (in this segment), which has a fixed offset to the start address. Hence, ASLR can be defeated by information disclosure of only a few memory addresses. To make randomization-based defense more robust, more fine-grained randomization techniques were proposed.

b: INSTRUCTION LOCATION RANDOMIZATION

In order to thwart such (two-stage) attacks, in which an attacker first finds the memory addresses of target objects by leaking the start address of the corresponding segment, and then conducts the real attack, instruction location randomization (ILR) [138] is introduced. ILR generates unpredictable code layout to create randomness in a code segment itself, i.e., instructions are not placed sequentially but rather distributed randomly across the memory address space. To execute a program with randomly scattered instructions properly, ILR creates, during static analysis, a so-called fallthrough map specifying the execution order of those instructions. Despite being more effective in hiding instructions locations, ILR suffers from not only high performance overhead, but also code coverage deficiencies due to imprecision of the static analysis [70].

c: DATA SPACE RANDOMIZATION

Another approach providing a higher level of randomness and stronger resilience to information leak is data space randomization (DSR) [139]. Instead of randomizing the location of some data, DSR randomizes the representation of all data by encrypting it using different keys. When variables are encrypted with different keys, a memory location (of a variable) overwritten by an attacker through a different variable will have an unexpected value after decryption, making it useless for the attacker. As a result, DSR provides a strong protection against not only control-oriented exploitation but also data-oriented exploitation. Nevertheless, fine-grained DSR may cause high performance overhead [67]. Furthermore, DSR is not binary compatible, meaning that instrumented binaries are not compatible with unmodified libraries [4].

d: AUTOMATED SOFTWARE DIVERSITY

The essence of randomization-based approaches is protecting programs by introducing program diversity,[20] and this idea goes back to nearly 30 years ago [140]. The proposed program evolution aims to create syntactically different but semantically equivalent versions of an original program. A generalization of randomization-based approaches is automated software diversity [141], [142], which presents various forms of randomization/diversification, with distinct diversification targets, at different levels of granularity, and introduced in individual phases of the software life-cycle.

Other influential research works on randomization-based defense include [143]–[146] [147]–[150] [151], which fit to one of above-listed types, but are not discussed in this paper due to space constraint. Nevertheless, like the discussed ones, all randomization-based defense techniques suffer from a fundamental problem. That is, they rely on secrets, which can be eventually guessed (through brute-force) or leaked (through some flaws). Furthermore, even without directly disclosing the memory layout, just-in-time ROP [152], just-in-time spraying [153], [154], side channel attacks [155], [156], and address-oblivious code-reuse attacks [157] etc. prove, again and again, that randomization-based defenses cannot provide sufficiently strong protection as promised or expected. That is to say, randomization-based defense approaches, at best, add only an extra necessary hurdle for attackers to overcome.

4) FLOW INTEGRITY

Apart from program integrity-based defenses, there is a large body of research focusing on another type of integrity enforcement, i.e., flow integrity of a running program, including control-flow integrity (CFI) and data-flow integrity (DFI).

a: CONTROL-FLOW INTEGRITY

When a code pointer, e.g., a return address, is corrupted and then loaded into the instruction pointer register through an indirect control-flow transfer instruction, e.g., a return instruction, the control flow of the program execution will be unknowingly diverted. Predetermining the control-flow graph (CFG) of a program during static analysis and then inserting a stateless check before each indirect control-flow transfer,[21] can detect the deviation of the intended control-flow at runtime.

CFI, proposed in [68], [158], is a key defense mechanism that relies on a CFG, which can be seen as some form of finite state machine. It restricts the target of each indirect control-flow transfer to only a few locations, as these locations are considered as legitimate during static analysis and marked with a unique label. At runtime, before an indirect control-flow transfer, it is checked whether the transfer target has the right label. If not, the process is forced to terminate. By doing so, most control-oriented exploitation will fail, as a control-flow hijacking attempt will most likely fail the CFI check, and results in program execution termination. Note that the effectiveness of CFI heavily relies on enforced W⊕X policy, as it ensures that code (including CFI checks) cannot be tampered, and injected code (for which there is no CFI check) cannot be executed.

i) FORWARD-EDGE CFI

Forward-edge CFI inserts checks for indirect jump and indirect call instructions. Whereas an indirect jump instruction can be introduced by a switch statement in source code, an indirect call instruction is normally resulted from a function call. At runtime, for an indirect jump or call, there is only one correct jump target or call target, respectively. However, during static analysis, the jump target or call target cannot be completely determined, but rather restricted to a few possible locations. Hence, all these possible locations are considered as valid and receive the same unique label. This, though, leaves an attacker a slight possibility to hijack the control flow, when one of these locations happens to be the one to which the attacker aims to redirect the control flow. As a result, the program execution is compliant to the enforced CFI policy, but deviated from the expected program behavior. The more statically determined valid targets an indirect jump or call has, the better the attacker's chance is.

ii) BACKWARD-EDGE CFI

Backward-edge CFI aims to prevent a return instruction from being used for control-flow hijacking. Similarly, CFI cannot guarantee that, at runtime, a return instruction really returns to its true invoking site, but rather to one of its seemingly legitimate callers, as this function may be called from (many) different places of the code at different times. The number of callers of a function may be noticeably higher than the number of locations to which an indirect jump/call may lead, making backward-edge CFI even less precise than

[20]It is often said that security by obscurity is a bad strategy. However, the idea of program diversity is to introduce some obscurity and thereby to enhance the complexity and cost for attackers. This may sound contradictory. We stress that security by obscurity is a bad thing, only when the obscurity mechanism is the solely deployed countermeasure, and even not properly designed and implemented.

[21]In contrast, a direct control-flow transfer instruction has a fixed target memory address/offset, which cannot be manipulated by attackers due to enforced code integrity.

forward-edge CFI. That is, statically computed CFG alone inevitably leads to *coarse-grained CFI*.

Coarse-grained CFI [159]–[162] are considered as practical and suitable for real-world deployment due to comparatively low overhead. Nevertheless, due to imprecision and inherent limitations of static analysis, the number of allowed target locations for an indirect control-flow transfer is normally larger than necessary, resulting in some degree of over-approximation [54]. The effectiveness of a coarse-grained CFI depends on the degree of over-approximation. As shown in [54], coarse-grained CFI in general can be easily defeated by advanced code-reuse attacks.

To enhance the precision of CFI, in the original CFI works [68], [158], statically computed CFI checks are coupled with an above-mentioned shadow stack, moving towards more *fine-grained CFI*. Whereas the statically inserted CFI checks are stateless, a shadow stack adds some state-awareness during program execution, and guarantees that each return instruction really goes back to the initial caller. However, having to keep a shadow stack for each execution thread introduces non-negligible overhead. Furthermore, problem may occur in certain programming constructs in which a return instruction does not always need to return to its direct caller, that is, correctly tracking a shadow stack is not trivial [60], [70].

To reduce performance degradation, hardware-assisted CFI [163]–[166] [6], [167]–[169], which requires extending an instruction set architecture, i.e., adding new CPU instructions, becomes more relevant and attractive. Intel CET [6], [169] showcases a real-world deployment of hardware-assisted CFI. Although neither software-only CFI nor hardware-supported CFI can completely prevent control-oriented exploitation [7], [170], they do make most code-reuse attacks highly unreliable and unsuccessful. As a result, data-oriented exploitation has attracted much attention of attackers in recent years [171].

b: DATA-FLOW INTEGRITY

An effective way to defend against both control-oriented exploitation and data-oriented exploitation is enforcing a DFI policy, which is originally proposed in [172] short after the proposal of CFI. To enforce a DFI policy, a data-flow graph (DFG) is constructed during static analysis for a protected program, and the program code is instrumented to ensure that the data flow does not deviate from the DFG at runtime. But instead of checking indirect branching instructions, as it is done in CFI, DFI checks each read operation, to prevent any corrupted data from being used. That is, DFI aims to detect the corruption of the data that is the next to be accessed. A DFG may include all program data for enhanced precision [172], or only consider a small portion of security-critical data for reduced performance overhead [173]. Like a statically computed CFG, a DFG obtained from static analysis is only an over-approximation of a complete DFG, leading to coarse-grained DFI. More fine-grained DFI may provide a stronger protection, but suffers from very high performance overhead and is not yet very practical [54], [67].

5) MEMORY SANITIZATION

Whereas above-mentioned defenses aim to detect or prevent attacks at some late stage of a bug's exploitation, another line of research focuses on fighting against the underlying root cause, i.e., preventing memory corruption from ever happening. This is done by memory sanitization [174], which includes finding and removing vulnerable code [175] or unnecessary code [176]; extending C language to safer versions [177], [178]; integrating spatial safety checks [179], temporal safety checks [180] to provide stronger policies. Like above-mentioned defenses, memory sanitizers also rely on program instrumentation, e.g., inserting reference monitors at some level. Memory sanitizers normally provide stronger defenses and potentially stop any binary exploitation, at the cost of very high performance overhead.

To uncover memory corruption bugs before a program's release and hence to scale down the attack surface, defenders propose frameworks for conducting inter-procedural, context-sensitive analysis in not only user application programs but also operating system kernels [175]. To further lower attackers' chance to find and exploit a bug, some developer-intended program features and the code, which are not necessarily useful for every program user or not used in a particular deployment context, can also be removed individually during program instrumentation and before the program's deployment [176]. To make C programs safer, a program transformation system [178] is proposed to extend C with type safety, in which type safety is either statically verified or guaranteed by inserted checks at runtime. Another C extension [177] also combines static analysis and runtime checks to rule out (almost) all safety violations in a C program, while preserving programmers' control over low-level details.

A strong defense can also be provided by embedding spatial memory-safety checks and temporal memory-safety checks in a program. Complete memory safety can be ensured by maintaining and correctly tracking both bounds information and allocation information as metadata for every pointer. Whenever a pointer is dereferenced, this metadata is used to examine whether the pointer stays inside the bounds of the pointed object, and whether the pointed object is still valid. The original pointer extended with the associated metadata turns into a so-called *fat pointer*. With fat pointers, an instrumented program cannot interact with uninstrumented libraries, as they cannot correctly interpret and update fat pointers. To address this compatibility problem, the metadata can also be separated from the original pointer and stored in a disjoint metadata space like in SoftBound [179] and CETS [180]. Whereas SoftBound ensures spatial memory safety for a C program, CETS provides temporal memory safety for it. When combined, a complete memory safety can be guaranteed, but with a performance overhead over 100% [180].

6) MEMORY ISOLATION

Lastly, memory isolation-based defenses are also widely studied and implemented for countering binary exploitation, by restricting an attack to a confined memory address space. Disjoint memory regions are used for trusted and untrusted program code, respectively, to contain the memory access and control-flow transfer behavior of untrusted program code. Software-based Fault Isolation (SFI) techniques [181]–[184] [185], [186], also referred to as sandboxing techniques, establish logical protection domains within a process, and constrain the untrusted code's memory access and control-flow transfer to a designated logical memory address space, referred to as the fault domain. Similarly, Intel Software Guard Extensions (SGX) [187] allows program code to designate a protected memory region, called enclave, for hosting security-critical code and data, in order to prevent compromised code from accessing it. However, memory isolation-based countermeasures can, at best, limit the damage that an attacker can cause, but not prevent it from happening. Besides, these defense mechanisms are also breakable [188], [189].

B. ON EMBEDDED SYSTEMS

According to Embedded Markets Study [190], [191], ICS are the largest host of embedded devices. As it shows in the past [9]–[12] [13]–[16] [17]–[20], despite their limited computation capabilities, embedded systems are also prone to fall victim to binary exploitation. One of the main reasons is that embedded systems are predominantly developed in unsafe programming languages like C, C++ or assembly languages [190], owing to the fact that they support high performance and low-level hardware control. As defense mechanisms are widely deployed on modern general-purpose computers, which can raise the bar for conducting binary exploitation, attackers now typically focus on targets that either do not enable defense mechanisms or cannot implement them. Embedded devices often cannot directly implement many of above-mentioned defense mechanisms due to, in particular, performance overhead and hardware limitation. Besides, embedded systems are normally difficult to update, and it is not uncommon that many of them contain unpatched publicly known software vulnerabilities for a long time. Hence, attackers will more likely shift their focus to the low-hanging fruit—embedded devices in ICS—to yield easy wins. In this section, we first discuss the constraints that prohibit the direct implementation of some above-mentioned defense techniques on embedded systems. Next, we review some defense mechanisms designed for embedded systems and with the limitations of embedded systems in mind.

1) CONSTRAINTS

Even on general-purpose computers, some effective defenses are normally not implemented or enabled due to high overhead. According to [4], defenses causing over 10% performance overhead will unlikely get widely deployed. When applied to embedded systems with their tight constraints, the performance penalty issue is further exacerbated. Solely considering this, it makes many defenses developed for general-purpose computers already practically infeasible.

Another important issue which makes many defenses technically infeasible is hardware limitation. That is, most embedded systems (except a few high-end embedded systems like smartphones) do not have a memory management unit (MMU), and hence do not support virtual memory and memory paging, which underpin many necessary and universally deployed security mechanisms on general-purpose computers. For instance, both the enforcement of code integrity policy and the enforcement of DEP policy rely on the presence of a MMU. Without these defenses, the most basic code corruption attacks and code injection attacks become easily achievable. Although CFI itself does not require a MMU, it can be rendered useless by code corruption attacks (through manipulating CFI checks) and code injection attacks (since for injected code there is no CFI check).

Above-mentioned randomization-based defenses like ASLR also become much weaker and less relevant, when it comes to embedded systems. Because embedded systems typically run on 32-bit, 16-bit or even 8-bit architectures, for which the memory address space is much smaller and thereby the entropy sources are very limited. This makes defenses relying on information hiding, like randomization-based defenses or shadow stack, generally more easily breakable. For example, as it shows in [192], the protection provided by ASLR is limited on 32-bit architectures, not to mention on 16-bit and 8-bit architectures.

2) AVAILABLE DEFENSE TECHNIQUES

Although defense techniques for general-purpose computers are not directly applicable to embedded systems, they are highly influential for developing defenses for embedded systems. That is, as shown in the following, similar approaches are proposed for embedded systems, but with the limitations of embedded systems in consideration. Besides, in comparison to software-only solutions, hardware-based solutions appear to be more favored in embedded systems due to relatively low overhead.

a: CODE INTEGRITY

The easiest way to carry out binary exploitation would be directly corrupting program code in memory. On general-purposes computers, this is straightforwardly prevented by a code integrity policy, which leverages a MMU to realize access control, i.e., all memory pages containing code cannot be modified. However, a MMU causes unacceptable performance overhead, i.e., due to management of page tables, for most of embedded systems. To overcome this, a memory protection unit (MPU) [193], [194], often seen as a lightweight MMU, is implemented in a large number of (high-end) embedded systems. A MPU does not support virtual memory, but provides access control by arranging the memory address space into several memory regions with associated access permissions. Consequently, not only the enforcement of a code integrity policy is feasible, but so are other defenses heavily relying on memory access control.

b: RETURN ADDRESS INTEGRITY

At first glance, a return address is safely stored in many embedded systems due to a dedicated register called link register. In many embedded devices employing RISC instruction set architectures, e.g., ARM, PowerPC, SPARC, a link register is used to hold the return address of a function call. This is not only more efficient (as accessing a register is much faster than accessing the stack on the main memory) but also safer, as an attacker cannot directly tamper a value on a register. However, nested function calls are very typical in a program, and in this case some return addresses have to be stored on the stack. As a result, attackers typically try to overwrite a return address on the stack to hijack the control flow.

As a countermeasure on embedded systems, defenders aim to protect return addresses by either implementing a dedicated return stack [195], or deploying a shadow stack [196], [197], or directly ensuring the integrity of return addresses [198]. To safeguard low-end embedded systems, a light hardware modification is introduced in [195] (for AVR microcontrollers), which splits the stack into a normal data stack (without return addresses) and a return stack (containing solely return addresses). The return stack is stored at a different location in memory, and the access to the return stack is restricted to return and call instructions to prevent unauthorized modification, which is realized by the hardware modification.

Like on general-purpose computers, a shadow stack can be implemented to protect return addresses of a program running on embedded devices. But preventing a shadow stack itself from being tampered is more difficult on embedded systems due to limited entropy sources and memory isolation. To make a shadow stack safer, an approach called Silhouette [197] is designed for various ARM architectures, which enforces an efficient intra-address space isolation dubbed *store hardening*. It relies on some special store instructions of ARM processors and a MPU to set memory access permissions. By doing so, it creates a logical separation between code associated to shadow stack operations and other program code. Another approach ensuring return address integrity is μRAI proposed in [198], which turns a general-purpose register into a dedicated register called status register. The value in this register is used in combination with (during program instrumentation) inserted direct jump instructions to resolve the correct return addresses, and it is guaranteed that this value is never spilled to the main memory. As a result, it removes the need to ever store return addresses on the stack, and makes sure that return addresses are never writable except by an authorized instruction.

c: POINTER INTEGRITY

Enforcing return address integrity can only prevent backward-edge control-flow hijacking. In order to thwart forward-edge control-flow hijacking, other code pointers like function pointers also need to be safeguarded. High-end embedded systems running on ARM processors with ARMv8-A architecture [199] now have instructions for pointer authentication to resist attacks leveraging corrupted pointers. It makes use of cryptographic MAC, referred to as pointer authentication codes (PAC), and places a PAC into unused bits of a pointer value (in 64-bit architectures), before the pointer is written to memory. Upon using this pointer, its integrity is first verified. As a result, an attacker has to find the correct PAC (in addition to a memory corruption bug), in order to corrupt a protected pointer. However, pointer authentication may be vulnerable to pointer substitution attacks, in which an authenticated function pointer is replaced with another authenticated pointer pointing to a different, attacker-intended memory location [200]. As a countermeasure, an enhanced scheme is proposed in [201], which enforces pointer integrity for all code and data pointers, coupled with pointer type safety ensuring a pointer is of the correct type.

d: eXecute-ONLY-MEMORY

By setting memory pages or regions as either writable (and readable) or executable (and readable), i.e., W$\oplus$X policy, through a MMU or MPU, respectively, it prevents attackers from both corrupting code and executing injected code, thereby largely restricting attackers' ability. However, this does not thwart memory disclosure attacks, in which attackers disclose sensitive information from code regions, e.g., in order to conduct code-reuse attacks. To defeat memory disclosure attacks, eXecute-Only-Memory (XOM) is proposed, which states that some code regions are solely executable, i.e., neither writable nor readable. Although XOM is supported in high-end processors by extending memory access permissions to include execute-only (XO) permission [202], it is not built into most of embedded systems. To enable this feature in (low-end) embedded systems, researchers present uXOM [203] for ARM Cortex-M[22]—one of the most popular processors in low-end embedded devices. In addition to a MPU, uXOM also leverages unprivileged load and store instructions offered by ARM Cortex-M processors. It converts most memory instructions into unprivileged ones and sets code regions as privileged, so that these instructions cannot access code regions. For memory instructions that cannot

[22]Note that XOM cannot be implemented simply by configuring the MPU in a Cortex-M processor, since read permission is needed for a memory region to be executable [203].

be turned into unprivileged ones, they are instrumented with verification routines ensuring that uXOM's protection will not be broken.

e: RANDOMIZATION

Randomization-based defenses increase the resilience against, in particular, code-reuse attacks. To address unique challenges of implementing effective randomization mechanisms on low-end embedded systems, a hardware-based fine-grained randomization technique called MAVR [16] is proposed for 8-bit AVR microcontrollers. MAVR extends a main processor with an external flash memory and an additional low-cost processor dubbed master processor. The external flash memory is used to store the unrandomized program code and symbol information. The master processor is responsible for the randomization, and uploading randomized program code to the main processor. In order to make it harder for attackers to find out gadgets locations, it shuffles the function blocks within the code segment, instead of just start addresses of all segments.

f: CFI

In order to enforce CFI on real-time embedded systems, but not create extra unpredictable workload on the processor, a hardware-based solution called OCFMM is presented in [18]. OCFMM introduces a dedicated hardware module to perform CFI checks for the program running on a 32-bit SPARC processor, in which it directly hooks into the processor and tracks the control flow of the program. This module has its own isolated memory unit for preventing itself from being manipulated. Another hardware-assisted CFI approach for embedded systems proposed in [164] does not add an additional hardware module, but rather introduces new instructions through hardware modification on a 32-bit Intel processor designed for embedded systems. Besides, software-only CFI, such as RECFISH [196] targeting ARM Cortex-R processors, is also proposed for real-time embedded systems.

g: MEMORY ISOLATION

To constrain the ability of an attacker who already partly or completely exploited an unprivileged program, and to prevent the attacker from further exploiting other security-critical programs and resources, memory isolation-based defenses can be deployed not only on general-purpose computers, e.g., with Intel SGX [187], but also on embedded devices, e.g., with ARM TrustZone [204]. However, ARM TrustZone offers a limited degree of isolation (and security) in comparison to Intel SGX, because it only separates programs into two areas, i.e., a normal world and a secure world, but provides no strong compartmentalization within the areas. As a result, the attack surface of the secure world expands with increased number of trusted (but potentially vulnerable) programs [205]. To make ARM TrustZone more comparable to Intel SGX, i.e., establishing user-space enclaves and enabling unrestricted use of trusted execution environments (TEE), a new security architecture [206] is proposed, which suggests moving trusted programs to the normal world and introducing strongly isolated compartments in the normal world.

h: MEMORY TAGGING

Another important hardware-based feature is memory tagging, which is designed to provide a very promising and robust defense against binary exploitation. Memory tagging is currently supported in some SPARC processors and ARM v8.5 architecture, in which the implementation is called SPARC ADI [207] and ARM MTE [208], respectively. For instance, two types of metadata are used in ARM MTE, i.e., address tags and memory tags. Whereas an address tag is a 4-bit value inserted to the topmost unused byte of each pointer (only in 64-bit architecture), a memory tag is also a 4-bit value associated with every aligned 16-byte memory region [209]. These tags are handled by newly introduced instructions. When a heap region is allocated, a random 4-bit value is selected to mark both the pointer and the heap region. Upon accessing the memory region, it is verified whether the address tag and memory tag match. A mismatch will cause a hardware exception.

Note that defenses like ARM pointer authentication, ARM TrustZone and ARM MTE rely on advanced processor features, and hence mostly apply to high-end (64-bit) embedded systems. Low-end embedded systems are in general less protected due to more strict resource constraints and hardware limitations.

VI. DETECTION TECHNIQUES AND EVASION TECHNIQUES

Fully enforced memory safety is rare in real-world systems, in particular due to performance and compatibility issues [8]. This often makes another line of defense become necessary, i.e., intrusion detection. Often used as a secondary defense, intrusion detection can lessen the trust needed in and the reliance on a front-line defense mechanism. In this section we mainly discuss detection techniques against binary exploitation, and evasion techniques. They differ from defenses discussed in Section V, in that they are either heuristics-based detection, and/or the detection is conducted from a connected device in the same physical/virtual network (rather than on the monitored device itself), i.e., remote runtime attestation. Heuristics-based detection approaches have the advantage that they can apply machine learning techniques, e.g., rule learning based IDS [210]. Remote runtime attestation is in particular suitable for (low-end) embedded systems due to little to no extra performance overhead on a monitored device itself. Hence we see a great potential in these detection techniques, especially when combined.

A. HEURISTICS-BASED DETECTION

1) SHELLCODE DETECTION

Early detection-based solutions focus on shellcode detection, which allows detecting not only known exploits but also unknown exploits, irrespective of underlying program bugs being exploited. The first generation of shellcode detection bases on static analysis of network traffic [211], [212], in which network input data is first disassembled and then compared with predefined pattern(s). A pattern can be a combination of heuristics-based features, e.g., *NOP-sled*, identified through studying a large sample of shellcode. However, static analysis-based detection approaches are limited in that they cannot correctly deal with code obfuscation tricks employed by shellcode authors to evade detection.

Hence, researchers later turned to dynamic analysis-based detection techniques coupled with a processor emulator, implemented either directly on the monitored/protected host [213] or on a network-level monitor/detector [214]–[216]. As a result, obfuscated/encrypted shellcode can be detected, as the shellcode needs to be first deobfuscated/decrypted during its execution. For instance, in [216], each network input, i.e., a byte sequence, is mapped to the memory space and executed by the processor emulator for several times (each time beginning with a different byte in the input). The execution results are matched against patterns based on runtime heuristics and identified to be inherent in different types of shellcode. Nonetheless, the aforementioned approaches may be still susceptible to evasion, due to insufficient context information only obtainable from a real target process. To solve this issue, a new approach [217] suggests frequently taking a snapshot of a real target process's memory space containing a to-be-examined input, and sending the snapshot to a shellcode detector for more realistic examination.

2) ROP DETECTION

After the advent of code-reuse attacks, especially ROP attacks, shellcode detection-based approaches seem to become much less relevant. That is, shellcode detection may be effective against code-injection attacks and hybrid exploits, but not against pure code-reuse attacks, as they do not contain any code. Consequently, defenders proceeded to develop methods for detecting ROP exploits. One of these methods is ROPscan [218], which aims to catch ROP payloads, i.e., a byte sequence consisting of gadgets addresses, from monitored network traffic. The emulator in ROPscan is initialized with a snapshot of the memory space of a targeted/protected process. ROPscan speculatively drives the execution of code existing in the memory space, to which a to-be-examined network input (presumably interpreted as valid memory addresses) points to. The execution results are examined according to some runtime heuristics, which will classify the input data as benign or malicious. Besides, such a ROP detector can be integrated into a shellcode detector for creating a stronger defense.

Some other runtime-behavior heuristics-based ROP detection techniques, like DROP [61], DynIMA [62], ROPdefender [219], embed the checks directly in program code, instead of relying on a dedicated emulation-based (network) monitor. An alarm is triggered, if a predefined condition is met. These detection techniques base on runtime heuristics obtained from observed ROP attacks, e.g., an excessive use of return instructions in a short time [61], [62], a violation of the last-in, first-out stack invariant [219]. To make these kind of techniques more generic and practical, i.e., with the ability to detect also JOP and COP attacks, and no reliance on side information and binary instrumentation, ROPecker [220] presents a new detection system, which detects ROP attacks by leveraging a combination of offline binary analysis results, runtime execution flow information, and records in last branch record (LBR) registers.

3) PROGRAM EXECUTION ANOMALY DETECTION

In order not to be restricted to any specific type of program exploitation, another group of detection techniques aim to detect all kinds of attacks by focusing on a program's normal execution behavior, rather than relying on the characteristics of a specific kind of attack. A normal execution behavior is defined in a normal program execution model, and any deviation from it, i.e., an anomaly, is deemed as malicious. For example, a normal execution model is defined in [221] as a set of conditions on program states. An execution behavior is considered as benign, if the program counter and the call stack prove to be valid and consistent with each other for all program states. Nevertheless, data-oriented exploitation is generally harder to detect, as it may not cause any control-flow anomaly during program execution. To improve the effectiveness of anomaly-based detection against data-oriented attacks on control programs in cyber-physical systems, a new approach [222] also incorporates runtime execution semantics checking, including event identification and dependence analysis with respect to physical environments.

B. REMOTE RUNTIME ATTESTATION

To detect the occurrence of a runtime attack on a (resource-constrained) embedded system, but without introducing non-negligible performance overhead on it, remote runtime attestation is a popular choice, as the detection process, i.e., the computation, is mainly performed on the remote server. Initially, remote attestation is introduced to detect code modification or malware on an embedded device, in which the trusted server, called *verifier*, requires the *prover* running on the embedded device to compute a checksum of its code memory (with a nonce included to prevent replay attacks), and the verifier will receive it and check it. Conventional remote attestation is referred to as static attestation, because it only aims to detect the presence of modified code at load-time, but is unable to detect binary exploitation at runtime. To complement static attestation, a number of runtime attestation schemes are proposed to catch runtime attacks by attesting not only control flow [223]–[225], but also data flow [226] of a program running on a monitored/protected embedded system.

C-FLAT [223] is an early runtime attestation scheme that verifies the control-flow integrity of a running program on an embedded system. It requires the remote verifier to generate the CFG of the attested program beforehand, and needs to instrument all branch instructions of the program, to let them be intercepted by a runtime tracer on the prover during execution. Besides, it has a trust anchor, i.e., ARM TrustZone, on the prover to securely measure and report the program's runtime behavior, i.e., its execution paths, to the verifier. However, C-FLAT has limited practicality due to its reliance on the hardware extension ARM TrustZone and program instrumentation, and thereby noticeable performance penalty on the embedded system itself. To improve the efficiency and usability of runtime attestation, in particular by eliminating the need of software instrumentation, a hardware-assisted approach LO-FAT [224] introduces dedicated hardware modules, i.e., a *branch filter* and a *loop monitor*, to track the attested program's control flow in parallel to the main processor. By doing so, it does not stall the attested program's execution and allows efficient control-flow attestation.

Nevertheless, both C-FLAT and LO-FAT are susceptible to two types of TOCTOU attacks, provided that the attacker has physical access to the embedded device [225]. This is due to the fact that they only attest indirect control transfers, but instructions within a basic block are not verified, and hence allowed to be replaced by malicious code. A more resilient runtime attestation scheme ATRIUM [225] mitigates these attacks by tracking and attesting both the executed instructions and the control flow of a target program, with an attestation engine separated from the processor. Although ATRIUM can detect additional memory manipulation attacks, it is incapable of identifying DOP attacks, which do not change a victim program's control flow. As a countermeasure against sophisticated DOP attacks, LiteHAX [226] suggests accurately and continuously tracking not only control-flow but also data-flow information of an attested program, which is realized by its introduced hardware modules tightly integrated with the main processor.

C. EVASION

It is often said that the easiest way to break system security usually is to circumvent it rather than defeat it. A fundamental problem of detection-based defenses is that they are prone to evasion. The early generation of shellcode detection mechanism [211] aims to detect *plain* shellcode using static analysis. As attackers always seek a way to evade detection, code obfuscation techniques such as polymorphism and metamorphism [227] are employed to circumvent such defense. Various forms of the same plain shellcode can be generated by encrypting it with individual random keys and appending a decryption routine to it, so that it can self-decrypt when executed. In the face of polymorphic and metamorphic shellcode, defenders moved to developing dynamic analysis-based detection approaches, which run suspicious code in an emulator and check the execution results with some runtime heuristics, and hence are not hindered by code obfuscation tricks. However, attackers, again, proved to be capable of achieving a higher level of evasiveness, in which they exploit both design flaws and implementation flaws in emulation-based detection systems [228]. For instance, there is always an *emulation gap*, i.e., some difference between a real target system and an emulator, which can abused to circumvent detection. An attacker may construct shellcode which will be executed completely and correctly only when it is on a real target system.

As it shows in [229], ROP detection techniques based on execution heuristics or anomalies [218]–[221] can also be evaded by a strong adversary. For instance, a length-based detection approach classifies a code sequence as normal or ROP gadget depending on its length. A stealth ROP attack will contain only code sequences that are longer than expected, and hence would be incorrectly classified as normal. Furthermore, dispatcher-like behavior or intensive use of indirect jumps or calls may be considered as an execution anomaly and taken as an indicator to detect JOP or COP attacks. However, this kind of detection could be circumvented by selecting long-running functions and changing dispatchers periodically [65]. Hence, evasion-resilient detection techniques are of special interest. It is undoubtedly critical to identify properties that are truly representative and indispensable in an attack vector or during an attack, but non-existent in a benign program or during a normal execution. These fundamental differences need to be captured and used to design a reliable and sound detection scheme that cannot be circumvented.

VII. CONCLUSION

With the aim of providing an appropriate assessment, and raising situation awareness adequately, we present a systematic study concerning binary exploitation in ICS. In this work, we first provide a comprehensive analysis of binary exploitation. We analyze its causes from different angles and give an overview of the most critical program bugs. The two types of binary exploitation, i.e., control-oriented and data-oriented, are extensively discussed, so are the varying consequences. The connection to modern malware is explained and illustrated with many real-world examples in the last decade, which serve to indicate the popularity and severity of binary exploitation and exploit-based malware in ICS. Besides, a number of exploitation purposes are summarized for the past ICS cyber incidents.

To further argue that binary exploitation will tend to be a more serious problem, we reveal a drastic shift from a well-isolated and -segmented network structure to a more interconnected and complex structure with converged IT and OT networks, and point out the added network entry points as well as the attack surface transition. Next, an extensive list of available defense techniques for general-purpose computers is presented, as well as another list of defense mechanisms designed for embedded systems and with the limitations of embedded systems in mind.

Lastly, we discuss detection-based defenses, often used as a secondary defense, as we see a great potential in heuristics-based detection and remote runtime attestation techniques for ICS. In order to reduce the damage that an attacker can cause, it is ideal to catch and halt the attack as early as possible, i.e., detect and prevent the program exploits. An early-stage detection and mitigation of an attack also means a less dramatic clean-up of the victim system. In particular, ICS asset owners should prioritize employing network-based detection techniques, as it is impractical to deploy resource-intensive defenses on many embedded devices, especially when the systems are already under operation. However, the biggest challenge is still how to identify fundamental properties in attack vectors and design evasion-resilient detection techniques.

REFERENCES

[1] Homeland Security. (2016). *Recommended Practice: Improving Industrial Control System Cybersecurity With Defense-in-Depth Strategies*. [Online]. Available: https://us-cert.cisa.gov/sites/default/files/recommended_practices/NCCIC_ICS-CERT_Defense_in_Depth_2016_S508C.pdf

[2] J. McCarthy, M. Powell, K. Stouffer, C. Tang, T. Zimmerman, W. Barker, T. Ogunyale, D. Wynne, and J. Wiltberger. (2020). *Securing Manufacturing Industrial Control Systems: Behavioral Anomaly Detection*. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8219.pdf

[3] V. van der Veen, N. D. Sharma, L. Cavallaro, and H. Bos, "Memory errors: The past, the present, and the future," in *Proc. 15th Int. Symp. Res. Attacks, Intrusions Defenses*, 2012, pp. 86–106.

[4] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 48–62.

[5] P. Larsen and A.-R. Sadeghi, Eds., *The Continuing Arms Race: Code-Reuse Attacks and Defenses*. New York, NY, USA: Association for Computing Machinery and Morgan & Claypool Publishers, 2018.

[6] Intel. (2017). *Control Flow Enforcement Technology (CET)*. [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/catc17-introduction-intel-cet-844137.pdf

[7] B. Sun, J. Liu, and C. Xu. *How to Survive the Hardware-Assisted Control Flow Integrity Enforcement*. Accessed: Dec. 7, 2021. [Online]. Available: https://i.blackhat.com/asia-19/Thu-March-28/bh-asia-Sun-How-to-Survive-the-Hardware-Assisted-Control-Flow-Integrity-Enforcement.pdf

[8] M. Payer, "How memory safety violations enable exploitation of programs," in *The Continuing Arms Race*, P. Larsen and A.-R. Sadeghi, Eds. New York, NY, USA: Association for Computing Machinery and Morgan & Claypool Publishers, 2018, pp. 1–23.

[9] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. *Return-Oriented Programming Without Returns on ARM*. Accessed: Dec. 13, 2021. [Online]. Available: https://www.ais.ruhr-uni-bochum.de/media/trust/veroeffentlichungen/2010/07/21/ROP-without-Returns-on-ARM.pdf

[10] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," in *Proc. 13th Int. Conf. Inf. Secur.*, 2011, pp. 346–360.

[11] A. Cui, J. Kataria, and S. J. Stolfo, "Killing the myth of Cisco IOS diversity: Recent advances in reliable shellcode design," in *Proc. 5th USENIX Conf. Offensive Technol.*, 2011. [Online]. Available: https://www.usenix.org/conference/woot11/killing-myth-cisco-ios-diversity-recent-advances-reliable-shellcode-design

[12] E. Itkin. (2020). *Don't be Silly—It's Only a Lightbulb—Check Point Research*. [Online]. Available: https://research.checkpoint.com/2020/dont-be-silly-its-only-a-lightbulb/

[13] F. Lindner. (2009). *Router Exploitation*. [Online]. Available: https://www.recurity-labs.com/research/FX_Router_Exploitation.pdf

[14] *MAR-17-352-01 HatMan—Safety System Targeted Malware (Update B)*, Cybersecur. Infrastruct. Secur. Agency, Arlington, VA, USA, 2019.

[15] A. Francillon and C. Castelluccia, "Code injection attacks on harvard-architecture devices," in *Proc. 15th ACM Conf. Comput. Commun. Secur. (CCS)*, 2008, pp. 15–26.

[16] J. Habibi, A. Gupta, S. Carlsony, A. Panicker, and E. Bertino, "MAVR: Code reuse stealthy attacks and mitigation on unmanned aerial vehicles," in *Proc. IEEE 35th Int. Conf. Distrib. Comput. Syst.*, Jun. 2015, pp. 642–652.

[17] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to RISC," in *Proc. 15th ACM Conf. Comput. Commun. Secur. (CCS)*, 2008, pp. 27–38.

[18] F. A. T. Abad, J. V. D. Woude, Y. Lu, S. Bak, M. Caccamo, L. Sha, R. Mancuso, and S. Mohan, "On-chip control flow integrity check for real time embedded systems," in *Proc. IEEE 1st Int. Conf. Cyber-Phys. Syst., Netw., Appl. (CPSNA)*, Aug. 2013, pp. 26–31.

[19] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham, "Can DREs provide long-lasting security? The case of return-oriented programming and the AVC advantage," in *Proc. Electron. Voting Technol. Workshop/Workshop Trustworthy Elections*, 2009. [Online]. Available: https://www.usenix.org/conference/evtwote-09/can-dres-provide-long-lasting-security-case-return-oriented-programming-and

[20] S. Checkoway, D. McCoy, B. Kantor, and D. Anderson, "Comprehensive experimental analyses of automotive attack surfaces," in *Proc. 20th USENIX Conf. Secur.*, 2011. [Online]. Available: https://www.usenix.org/conference/usenix-security-11/comprehensive-experimental-analyses-automotive-attack-surfaces

[21] M. Howard, D. LeBlanc, and J. Viega, *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. New York, NY, USA: McGraw-Hill, 2010.

[22] MITRE. *CWE—CWE-121-Stack-Based Buffer Overflow*. Accessed: Oct. 13, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/121.html

[23] MITRE. *CWE—CWE-122-Heap-Based Buffer Overflow*. Accessed: Oct. 13, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/122.html

[24] MITRE. *CWE—CWE-124-Buffer Underwrite ('Buffer Underflow')*. Accessed: Oct. 13, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/124.html

[25] MITRE. *CWE—CWE-126-Buffer Over-Read*. Accessed: Oct. 13, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/126.html

[26] MITRE. *CWE—CWE-127-Buffer Under-Read*. Accessed: Oct. 13, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/127.html

[27] MITRE. *CWE—CWE-680-Integer Overflow to Buffer Overflow*. Accessed: Oct. 13, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/680.html

[28] MITRE. *CWE—CWE-191-Integer Underflow (Wrap or Wraparound)*. Accessed: Oct. 13, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/191.html

[29] MITRE. *CWE—CWE-193-Off-By-One Error*. Accessed: Oct. 13, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/193.html

[30] MITRE. *CWE—CWE-843-Access of Resource Using Incompatible Type ('Type Confusion')*. Accessed: Oct. 13, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/843.html

[31] MITRE. *CWE—CWE-134-Use of Externally-Controlled Format String (4.5)*. Accessed: Oct. 13, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/134.html

[32] MITRE. *CWE—CWE-416-Use After Free*. Accessed: Oct. 13, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/416.html

[33] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, 2012, pp. 133–143.

[34] Blackngel, "Malloc des-maleficarum," *Phrack Mag.*, vol. 13, no. 66, 2009. [Online]. Available: http://phrack.org/issues/66/10.html

[35] MITRE. *CWE—CWE-415-Double Free*. Accessed: Oct. 13, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/415.html

[36] MITRE. *CWE—CWE-761-Free of Pointer Not at Start of Buffer*. Accessed: Oct. 13, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/761.html

[37] MITRE. *CWE—CWE-590-Free of Memory Not on the Heap*. Accessed: Oct. 13, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/590.html

[38] MITRE. *CWE—CWE-762-Mismatched Memory Management Routines*. Accessed: Oct. 13, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/762.html

[39] MITRE. *CWE—CWE-457-Use of Uninitialized Variable*. Accessed: Oct. 13, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/457.html

[40] MITRE. *CWE—CWE-908-Use of Uninitialized Resource*. Accessed: Oct. 13, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/908.html

[41] MITRE. *CWE—CWE-824-Access of Uninitialized Pointer*. Accessed: Oct. 13, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/824.html

[42] MITRE. *CWE—CWE-476-Null Pointer Dereference*. Accessed: Oct. 13, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/476.html

[43] MITRE. *CWE—CWE-364-Signal Handler Race Condition*. Accessed: Oct. 13, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/364.html

[44] MITRE. *CWE—CWE-365-Race Condition in Switch*. Accessed: Oct. 13, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/365.html

[45] MITRE. *CWE—CWE-367-Time-of-Check Time-of-Use (TOCTOU) Race Condition*. Accessed: Oct. 13, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/367.html

[46] MITRE. *CWE—CWE-368-Context Switching Race Condition*. Accessed: Oct. 13, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/368.html

[47] MITRE. *2021 CWE Top 25 Most Dangerous Software Weaknesses*. Accessed: Oct. 13, 2021. [Online]. Available: https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html

[48] M. Miller, "Trends and challenges in the vulnerability mitigation landscape," in *Proc. 13th USENIX Workshop Offensive Technol.*, 2019. [Online]. Available: https://www.usenix.org/conference/woot19/presentation/miller

[49] A. One, "Smashing the stack for fun and profit," *Phrack Mag.*, vol. 7, no. 49, pp. 14–16, 1996. [Online]. Available: http://phrack.org/issues/49/14.html

[50] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Proc. 14th Conf. USENIX Secur. Symp.*, 2005, pp. 177–191.

[51] Solar Designer. *Getting Around Non-Executable Stack (and Fix)*. Accessed: Sep. 18, 2021. [Online]. Available: https://seclists.org/bugtraq/1997/Aug/63

[52] H. Shacham, "The geometry of innocent flesh on the bone," in *Proc. 14th ACM Conf. Comput. Commun. Secur.*, 2007, pp. 552–561.

[53] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits," in *Proc. 24th USENIX Secur. Symp.*, 2015, pp. 177–192.

[54] N. Carlini, A. Barresi, M. Payer, and D. Wagner, "Control-flow bending on the effectiveness of control-flow integrity," in *Proc. 24th USENIX Secur. Symp.*, 2015, pp. 161–176.

[55] Nergal, "The advanced return-into-lib(c) exploits," *Phrack Mag.*, vol. 11, no. 58, 2001. [Online]. Available: http://www.phrack.org/archives/issues/58/4.txt

[56] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*, 2011, pp. 121–141.

[57] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, pp. 1–34, 2012.

[58] I. Arce, "The shellcode generation," *IEEE Secur. Privacy Mag.*, vol. 2, no. 5, pp. 72–76, Sep. 2004.

[59] A. van de Ven. (2004). *Exec Shield*. [Online]. Available: https://people.redhat.com/mingo/exec-shield/docs/WHP0006US_Execshield.pdf

[60] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Proc. IEEE Symp. Secur. Privacy*, May 2014, pp. 575–589.

[61] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "Drop: Detecting return-oriented programming malicious code," in *Proc. Int. Conf. Inf. Syst. Secur.*, 2009, pp. 163–177.

[62] L. Davi, A.-R. Sadeghi, and M. Winandy, "Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks," in *Proc. ACM Workshop Scalable Trusted Comput. (STC)*, 2009, pp. 49–54.

[63] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, "Defeating return-oriented rootkits with 'return-less' kernels," in *Proc. 5th Eur. Conf. Comput. Syst. (EuroSys)*, 2010, p. 195.

[64] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proc. 17th ACM Conf. Comput. Commun. Secur. (CCS)*, 2010, pp. 559–572.

[65] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proc. 6th ACM Symp. Inf., Comput. Commun. Secur. (ASIACCS)*, 2011, pp. 30–40.

[66] S. Checkoway and H. Shacham. (2010). *Escape From Return-Oriented Programming: Return-Oriented Programming Without Returns (on the X86)*. [Online]. Available: https://hovav.net/ucsd/papers/cs10.html

[67] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 969–986.

[68] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proc. 12th ACM Conf. Comput. Commun. Secur.*, 2005, p. 340.

[69] D. D. Zovi. (2010). *Practical Return-Oriented Programming*. [Online]. Available: https://people.csail.mit.edu/hes/ROP/Readings/rop-talk.pdf

[70] L. V. D. Davi, "Code-reuse attacks and defenses," Ph.D. dissertation, Tech. Univ. Darmstadt, Darmstadt, Germany, 2015.

[71] P. Ratanaworabhan, B. Livshits, and B. Zorn. (2008). *Nozzle: A Defense Against Heap-Spraying Code Injection Attacks*. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2008-176.pdf

[72] F. Gadaleta, Y. Younan, and W. Joosen, "Bubble: A Javascript engine level countermeasure against heap-spraying attacks," in *Proc. Int. Symp. Eng. Secure Softw. Syst.*, 2010, pp. 1–17.

[73] A. Sotirov. (2007). *Heap Feng Shui in JavaScript*. [Online]. Available: http://www.mathcs.richmond.edu/~dszajda/research/summer_2014/papers/sotirov_heap_feng_shui_javascript_paper.pdf

[74] Y. Wang, C. Zhang, Z. Zhao, B. Zhang, X. Gong, and W. Zou, "MAZE: Towards automated heap Feng Shui," in *Proc. 30th USENIX Secur. Symp.*, 2021, pp. 1647–1664.

[75] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2008, pp. 143–157.

[76] S. Heelan, "Automatic generation of control flow hijacking exploits for software vulnerabilities," M.S. thesis, Univ. Oxford, Oxford, U.K., 2009.

[77] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "Automatic exploit generation," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2011, pp. 74–84.

[78] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy," in *Proc. 20th USENIX Conf. Secur.*, 2011. [Online]. Available: https://www.usenix.org/conference/usenix-security-11/q-exploit-hardening-made-easy

[79] A. Follner, A. Bartel, H. Peng, Y.-C. Chang, K. K. Ispoglou, M. Payer, and E. Bodden, "PSHAPE: Automatically combining gadgets for arbitrary method execution," in *Proc. Int. Workshop Secur. Trust Manage.*, 2016, pp. 212–228.

[80] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proc. IEEE Symp. Secur. Privacy*, May 2012, pp. 380–394.

[81] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou, "Revery: From proof-of-concept to exploitable: (one step towards automatic exploit generation)," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 1914–1927.

[82] S. Heelan, T. Melham, and D. Kroening, "Gollum: Modular and greybox exploit generation for heap overflows in interpreters," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 1689–1706.

[83] D. Repel, J. Kinder, and L. Cavallaro, "Modular synthesis of heap exploits," in *Proc. 12th Workshop Program. Lang. Anal. Secur.*, Oct. 2017, pp. 25–35.

[84] M. Eckert, A. Bianchi, R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Heaphopper: Bringing bounded model checking to heap implementation security," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 99–116.

[85] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 1868–1882.

[86] Microsoft Defender Security Research Team. (2017). *Analysis of the Shadow Brokers Release and Mitigation With Windows 10 Virtualization-Based Security*. [Online]. Available: https://www.microsoft.com/security/blog/2017/06/16/analysis-of-the-shadow-brokers-release-and-mitigation-with-windows-10-virtualization-based-security/

[87] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *ACM SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 361–372, 2014.

[88] M. Seaborn and T. Dullien, "Exploiting the dram rowhammer bug to gain kernel privileges: How to cause and exploit single bit errors," *Black Hat*, vol. 15, 2015. [Online]. Available: https://www.blackhat.com/docs/us-15/materials/us-15-Seaborn-Exploiting-The-DRAM-Rowhammer-Bug-To-Gain-Kernel-Privileges-wp.pdf

[89] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic rowhammer attacks on mobile platforms," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 1675–1689.

[90] B. E. Strom, A. Applebaum, D. P. Miller, K. C. Nickels, A. G. Pennington, and C. B. Thomas. (2020). *MITRE ATT&CKr: Design and Philosophy*. [Online]. Available: https://attack.mitre.org/docs/ATTACK_Design_and_Philosophy_March_2020.pdf

[91] National Cybersecurity and Communications Integration Center. (2015). *ICS-CERT Monitor September 2014-February 2015*. [Online]. Available: https://us-cert.cisa.gov/sites/default/files/Monitors/ICS-CERT_Monitor_Sep2014-Feb2015.pdf

[92] Homeland Security. (2015). *NCCIC/ICS-Cert Year in Review 2015*. [Online]. Available: https://us-cert.cisa.gov/sites/default/files/Annual_Reports/Year_in_Review_FY2015_Final_S508C.pdf

[93] R. Langner. (2013). *To Kill a Centrifuge: A Technical Analysis of What Stuxnet's Creators Tried to Achieve*. [Online]. Available: https://www.langner.com/wp-content/uploads/2017/03/to-kill-a-centrifuge.pdf

[94] N. Falliere, L. O. Murchu, and E. Chien. (2010). *W32.Stuxnet Dossier*. [Online]. Available: https://www.wired.com/images_blogs/threatlevel/2010/11/w32_stuxnet_dossier.pdf

[95] Laboratory of Cryptography and System Security. (2011). *Duqu: A Stuxnet-Like Malware Found in the Wild*. [Online]. Available: https://www.crysys.hu/publications/files/bencsathPBF11duqu.pdf

[96] Symantec Security Response. (2011). *W32.Duqu: The Precursor to the Next Stuxnet*. [Online]. Available: https://docs.broadcom.com/doc/w32-duqu-11-en

[97] Kaspersky Lab. (2015). *The Duqu 2.0 Technical Details*. [Online]. Available: https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2018/03/07205202/The_Mystery_of_Duqu_2_0_a_sophisticated_cyberespionage_actor_returns.pdf

[98] A. Gostev. (2012). *The Flame: Questions and Answers*. [Online]. Available: https://securelist.com/the-flame-questions-and-answers/34344/

[99] Laboratory of Cryptography and System Security. (2012). *Skywiper (a.k.a. Flame a.k.a. Flamer): A Complex Malware for Targeted Attacks*. [Online]. Available: https://www.crysys.hu/publications/files/skywiper.pdf

[100] Symantec Security Response. *Dragonfly: Cyberespionage Attacks Against Energy Suppliers*. Accessed: Sep. 3, 2021. [Online]. Available: https://docs.broadcom.com/doc/dragonfly_threat_against_western_energy_suppliers

[101] J. Styczynski and N. Beach-Westmoreland. (2016). *When the Lights Went Out: A Comprehensive Review of the 2015 Attacks on Ukrainian Critical Infrastructure*. [Online]. Available: https://www.boozallen.com/content/dam/boozallen/documents/2016/09/ukraine-report-when-the-lights-went-out.pdf

[102] R. Lipovsky. (2016). *CVE-2014-4114: Details on August Blackenergy Powerpoint Campaigns*. [Online]. Available: https://www.welivesecurity.com/2014/10/14/cve-2014-4114-details-august-blackenergy-powerpoint-campaigns/

[103] A. Cherepanov and R. Lipovsky, "Blackenergy—What we really know about the notorious cyber attacks," in *Proc. Virus Bull. Conf.*, 2016. [Online]. Available: https://www.virusbulletin.com/conference/vb2016/abstracts/blackenergy-what-we-really-know-about-notorious-cyber-attacks/

[104] Dragos. (2017). *Crashoverride: Analyzing the Threat to Electric Grid Operations*. [Online]. Available: https://www.dragos.com/wp-content/uploads/CrashOverride-01.pdf

[105] A. Cherepanov. (2017). *Win32/Industroyer: A New Threat for Industrial Control Systems*. [Online]. Available: https://www.welivesecurity.com/wp-content/uploads/2017/06/Win32_Industroyer.pdf

[106] J. Slowik. (2019). *Crashoverride: Reassessing the 2016 Ukraine Electric Power Event as a Protection-Focused Attack*. [Online]. Available: https://www.dragos.com/wp-content/uploads/CRASHOVERRIDE.pdf

[107] Kaspersky ICS CERT. (2017). *Wannacry on Industrial Networks-Error Correction*. [Online]. Available: https://ics-cert.kaspersky.com/reports/2017/06/22/wannacry-on-industrial-networks/

[108] P. Mackenzie. (2019). *Wannacry Aftershock*. [Online]. Available: https://www.sophos.com/en-us/medialibrary/PDFs/technical-papers/WannaCry-Aftershock.pdf

[109] Kaspersky ICS CERT. (2017). *More Than 50% of Organizations Attacked by Expetr (Petya) Cryptolocker are Industrial Companies*. [Online]. Available: https://ics-cert.kaspersky.com/alerts/2017/06/29/more-than-50-percent-of-organizations-attacked-by-expetr-petya-cryptolocker-are-industrial-companies/

[110] S. Hurley and K. Sood. (2017). *Notpetya Technical Analysis Part II: Further Findings and Potential for MBR Recovery*. [Online]. Available: https://www.crowdstrike.com/blog/petrwrap-technical-analysis-part-2-further-findings-and-potential-for-mbr-recovery/

[111] M.-E. M. Léveillé. *Bad Rabbit: Not-Petya is Back With Improved Ransomware*. Accessed: Sep. 10, 2021. [Online]. Available: https://www.welivesecurity.com/2017/10/24/bad-rabbit-not-petya-back/

[112] O. Mamedov, F. Sinitsyn, and A. Ivanov. (2017). *Bad Rabbit Ransomware*. [Online]. Available: https://securelist.com/bad-rabbit-ransomware/82851/

[113] Zerosum0x0. (2017). *Doublepulsar Initial SMB Backdoor Ring 0 Shellcode Analysis*. [Online]. Available: https://zerosum0x0.blogspot.com/2017/04/doublepulsar-initial-smb-backdoor-ring.html

[114] C. Talos. (2017). *Threat Spotlight: Follow the Bad Rabbit*. [Online]. Available: https://blog.talosintelligence.com/2017/10/bad-rabbit.html

[115] A. Di Pinto, Y. Dragoni, and A. Carcano. (2018). *Triton: The First ICS Cyberattack on Safety Instrument Systems. USA*. [Online]. Available: https://i.blackhat.com/us-18/Wed-August-8/us-18-Carcano-TRITON-How-It-Disrupted-Safety-Systems-And-Changed-The-Threat-Landscape-Of-Industrial-Control-Systems-Forever-wp.pdf

[116] C. Talos. (2018). *New VPNFilter Malware Targets at Least 500K Networking Devices Worldwide*. [Online]. Available: https://blog.talosintelligence.com/2018/05/VPNFilter.html

[117] Symantec Security Response. (2018). *VPNFilter: New Router Malware With Destructive Capabilities*. [Online]. Available: https://symantec-enterprise-blogs.security.com/blogs/threat-intelligence/vpnfilter-iot-malware

[118] C. Talos. (2018). *VPNFilter Update—VPNFilter Exploits Endpoints, Targets New Devices*. [Online]. Available: https://blog.talosintelligence.com/2018/06/vpnfilter-update.html

[119] O. Alexander, M. Belisle, and J. Steele. (2020). *MITRE ATT&CKr for Industrial Control Systems: Design and Philosophy*. [Online]. Available: https://collaborate.mitre.org/attackics/img_auth.php/3/37/ATT&CK_for_ICS_-_Philosophy_Paper.pdf

[120] T. J. Williams, "The Purdue enterprise reference architecture" *Comput. Ind.*, vol. 24, no. 2, pp. 141–158, Sep. 1994.

[121] E. D. Knapp, *Industrial Network Security: Securing Critical Infrastructure Networks for Smart Grid, SCADA, and Other Industrial Control Systems*, 2nd ed. Waltham, MA, USA: Elsevier, 2014.

[122] D. Peterson. (2019). *Is the Purdue Model Dead?* [Online]. Available: https://dale-peterson.com/2019/02/11/is-the-purdue-model-dead/

[123] M. Giles. *Triton is the World's Most Murderous Malware, and It's Spreading*. (2019). [Online]. Available: https://www.technologyreview.com/2019/03/05/103328/cybersecurity-critical-infrastructure-triton-malware/

[124] Mission Secure. (2021). *A Comprehensive Guide to Operational Technology (OT) Cybersecurity*. [Online]. Available: https://www.missionsecure.com/resources/comprehensive-guide-to-operational-technology-security-ebook

[125] K. G. Crowther, R. M. Lee, and K. R. Wightman. (2018). *Building Security to Achieve Engineering and Business Requirements*. [Online]. Available: https://www.dragos.com/resource/design-and-build-productive-and-secure-industrial-systems-paper-2-of-3/

[126] E. P. Leverett, "Quantitatively assessing and visualising industrial system attack surfaces," Ph.D. dissertation, Univ. Cambridge, Cambridge, U.K., 2011.

[127] S. Miller, N. Brubaker, D. K. Zafra, and D. Caban. (2019). *TRITON Actor TTP Profile, Custom Attack Tools, Detections, and ATT&CK Mapping*. [Online]. Available: https://www.mandiant.com/resources/triton-actor-ttp-profile-custom-attack-tools-detections

[128] B. Meixell and E. Forner, "Out of control: Demonstrating SCADA exploitation," Cimation, Kuala Lumpur, Malaysia, Tech. Rep., 2013.

[129] C. Foreman. (2015). *Cyber-Security in Industrial Control Systems*. [Online]. Available: https://engineering.purdue.edu/VAAMI/ICS-modules.pdf

[130] B. Filkins, D. Wylie, and J. Dely. *Sans 2019 State of OT/ICS Cybersecurity Survey*. [Online]. Available: https://www.forescout.com/resources/2019-sans-state-of-ot-ics-cybersecurity-survey/

[131] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *7th USENIX Secur. Symp.*, 1998.

[132] T.-C. Chiueh and F.-H. Hsu, "RAD: A compile-time solution to buffer overflow attacks," in *Proc. 21st Int. Conf. Distrib. Comput. Syst.*, 2001, pp. 409–417.

[133] J. Li, L. Chen, Q. Xu, L. Tian, G. Shi, K. Chen, and D. Meng, "Zipper stack: Shadow stacks without shadow," in *Proc. 25th Eur. Symp. Res. Comput. Secur.*, vol. 12308, 2020, pp. 338–358.

[134] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *Proc. 11th USENIX Symp. Oper. Syst. Design*, 2014, pp. 81–116.

[135] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, "Missing the point(er): On the effectiveness of code pointer integrity," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 781–796.

[136] S. Andersen and V. Abella. (2004). *Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies, Data Execution Prevention*. [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-xp/bb457155(v=technet.10)

[137] PAX Team. (2003). *PAX Address Space Layout Randomization (ASLR)*. [Online]. Available: https://pax.grsecurity.net/docs/aslr.txt

[138] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd my gadgets go?" in *Proc. IEEE Symp. Secur. Privacy*, May 2012, pp. 571–585.

[139] S. Bhatkar and R. Sekar, "Data space randomization," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2008, pp. 1–22.

[140] F. B. Cohen, "Operating system protection through program evolution," *Comput. Secur.*, vol. 12, no. 6, pp. 565–584, 1993.

[141] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, Feb. 2013, pp. 1–11.

[142] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *Proc. IEEE Symp. Secur. Privacy*, May 2014, pp. 276–291.

[143] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits," in *Proc. 12th USENIX Secur. Symp.*, 2003, pp. 105–120.

[144] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proc. 10th ACM Conf. Comput. Commun. Secur. (CCS)*, 2003, pp. 272–280.

[145] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in *Proc. 22nd Annu. Comput. Secur. Appl. Conf. (ACSAC)*, Dec. 2006, pp. 339–348.

[146] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy X86 binary code," in *Proc. ACM Conf. Comput. Commun. Secur. (CCS)*, 2012, p. 157.

[147] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Proc. 21st USENIX Secur. Symp.*, 2012, pp. 475–490.

[148] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proc. IEEE Symp. Secur. Privacy*, May 2012, pp. 601–615.

[149] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2015, pp. 268–279.

[150] P. Chen, J. Xu, Z. Lin, D. Xu, B. Mao, and P. Liu, "A practical approach for adaptive data structure layout randomization," in *Proc. 20th Eur. Symp. Res. Comput. Secur.*, 2015, pp. 69–89.

[151] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 763–780.

[152] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 574–588.

[153] D. Blazakis, "Interpreter exploitation," in *Proc. 4th USENIX Workshop Offensive Technol.*, 2010, pp. 1–9.

[154] R. Gawlik and T. Holz, "SoK: Make JIT-spray great again," in *Proc. 12th USENIX Workshop Offensive Technol.*, 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/gawlik

[155] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 191–205.

[156] J. Seibert, H. Okhravi, and E. Söderström, "Information leaks without memory disclosures," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2014, pp. 54–65.

[157] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz, A.-R. Sadeghi, and H. Okhravi, "Address oblivious code reuse: On the effectiveness of leakage-resilient diversity," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/address-oblivious-code-reuse-effectiveness-leakage-resilient-diversity/

[158] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, pp. 1–40, Oct. 2009.

[159] Z. Wang and X. Jiang, "HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *Proc. IEEE Symp. Secur. Privacy*, May 2010, pp. 380–395.

[160] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 559–573.

[161] B. Niu and G. Tan, "Modular control-flow integrity," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2014, pp. 577–587.

[162] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *Proc. 23rd USENIX Secur. Symp.*, 2014, pp. 941–955.

[163] M. Budiu, Ú. Erlingsson, and M. Abadi, "Architectural support for software-based protection," in *Proc. 1st Workshop Archit. Syst. Support Improving Softw. Dependability (ASID)*, 2006, pp. 42–51.

[164] L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation," in *Proc. 51st ACM/EDAC/IEEE Design Automat. Conf. (DAC)*, Jun. 2014, pp. 1–6.

[165] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "HAFIX: Hardware-assisted flow integrity extension," in *Proc. 52nd Annu. Design Autom. Conf.*, Jun. 2015, pp. 1–6.

[166] D. Sullivan, O. Arias, L. Davi, P. Larsen, A.-R. Sadeghi, and Y. Jin, "Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity," in *Proc. 53rd Annu. Design Autom. Conf.*, Jun. 2016, pp. 1–6.

[167] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis, "HCFI: Hardware-enforced control-flow integrity," in *Proc. 6th ACM Conf. Data Appl. Secur. Privacy*, Mar. 2016.

[168] J. Zhang, B. Qi, Z. Qin, and G. Qu, "HCIC: Hardware-assisted control-flow integrity checking," *IEEE Internet Things J.*, vol. 6, no. 1, pp. 458–471, Feb. 2019.

[169] V. Shanbhogue, D. Gupta, and R. Sahita, "Security analysis of processor instruction set architecture for enforcing control-flow integrity," in *Proc. 8th Int. Workshop Hardw. Archit. Support Secur. Privacy*, Jun. 2019, pp. 1–11.

[170] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrdia, "The dynamics of innocent flesh on the bone," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 1675–1689.

[171] L. Cheng, H. Liljestrand, M. S. Ahmed, T. Nyman, T. Jaeger, N. Asokan, and D. Yao, "Exploitation techniques and defenses for data-oriented attacks," in *Proc. IEEE Cybersecur. Develop. (SecDev)*, Sep. 2019.

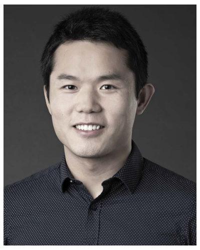
[172] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proc. 7th Symp. Oper. Syst. Design Implement.*, 2006, pp. 147–160.

[173] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016.

[174] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "SoK: Sanitizing for security," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 1275–1295.

[175] D. Gens, S. Schmitt, L. Davi, and A.-R. Sadeghi, "K-miner: Uncovering memory corruption in Linux," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018.
[176] M. Ghaffarinia and K. W. Hamlen, "Binary control-flow trimming," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 1009–1022.
[177] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *Proc. USENIX Annu. Tech. Conf.*, 2002, pp. 275–288.
[178] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 3, pp. 477–526, May 2005.
[179] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "SoftBound: Highly compatible and complete spatial memory safety for c," *ACM SIGPLAN Notices*, vol. 44, no. 6, pp. 245–258, May 2009.
[180] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "CETS: Compiler-enforced temporal safety for C," *ACM SIGPLAN Notices*, vol. 45, no. 8, pp. 31–40, 2010.
[181] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proc. 14th ACM Symp. Oper. Syst. Princ. (SOSP)*, 1993, pp. 203–216.
[182] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC architecture," in *Proc. 15th USENIX Secur. Symp.*, 2006, pp. 209–224.
[183] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software guards for system address spaces," in *Proc. 7th Symp. Oper. Syst. Design Implement.*, 2006, pp. 75–88.
[184] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast byte-granularity software fault isolation," in *Proc. ACM SIGOPS 22nd Symp. Oper. Syst. Princ. (SOSP)*, 2009, p. 45.
[185] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *Proc. 30th IEEE Symp. Secur. Privacy*, May 2009, pp. 79–93.
[186] G. Tan, "Principles and implementation techniques of software-based fault isolation," *Found. Trends Privacy Secur.*, vol. 1, no. 3, pp. 137–198, 2017.
[187] Intel. (2013). *Software Guard Extensions Programming Reference*. [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/329298-002-629101.pdf
[188] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi, "The guard's dilemma: Efficient code-reuse attacks against Intel SGX," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 1213–1227.
[189] T. Cloosters, M. Rodler, and L. Davi, "Discovery and exploitation of memory corruption vulnerabilities in SGX enclaves," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 841–858.
[190] *2015 Embedded Markets Study: Changes in Today's Design, Development & Processing Environments*, UBM Electron. Group, Vienna, Austria, 2015.
[191] AspenCore. (2019). *2019 Embedded Markets Study*. [Online]. Available: https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf
[192] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proc. 11th ACM Conf. Comput. Commun. Secur. (CCS)*, 2004, pp. 298–307.
[193] ARM Limited. (2014). *ARMv7-M Architecture Reference Manual*. [Online]. Available: https://developer.arm.com/documentation/ddi0403/eb/
[194] *Memory Protection Unit (MPU) for Keystone Devices User's Guide*, Texas Instrum., pp. 1–30. [Online]. Available: https://www.ti.com/lit/ug/sprugw5a/sprugw5a.pdf?ts=1638233271706
[195] A. Francillon, D. Perito, and C. Castelluccia, "Defending embedded systems against control flow attacks," in *Proc. 1st ACM Workshop Secure Execution Untrusted Code*, 2009, pp. 19–26.
[196] R. Walls, F. N. Brown, T. Baron, C. Shue, H. Okhravi, and C. B. Ward, "Control-flow integrity for real-time embedded systems," in *Proc. 31st Euromicro Conf. Real-Time Syst.*, 2019, pp. 1–24.
[197] J. Zhou, Y. Du, Z. Shen, L. Ma, J. Criswell, and R. J. Walls, "Silhouette: Efficient protected shadow stacks for embedded systems," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 1219–1236.
[198] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "μRAI: Securing embedded systems with return address integrity," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/murai-securing-embedded-systems-with-return-address-integrity/
[199] ARM Limited. (2021). *ARM Architecture Reference Manual, ARMv8, for ARMv8-A Architecture Profile*. [Online]. Available: https://developer.arm.com/documentation/ddi0487/ga
[200] Qualcomm Technologies Inc. (2017). *Pointer Authentication on ARMV8.3: Design and Analysis of the New Software Security Instructions*. [Online]. Available: https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf
[201] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan, "PAC it up: Towards pointer integrity using ARM pointer authentication," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 177–194.
[202] Y. Chen, D. Zhang, R. Wang, R. Qiao, A. M. Azab, L. Lu, H. Vijayakumar, and W. Shen, "NORAX: Enabling execute-only memory for COTS binaries on AArch64," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2017, pp. 304–319.
[203] D. Kwon, J. Shin, G. Kim, B. Lee, Y. Cho, and Y. Paek, "uXOM: Efficient execute-only memory on ARM cortex-M," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 231–247.
[204] ARM Limited. (2009). *ARM Security Technology: Building a Secure System Using Trustzone technology*. [Online]. Available: https://developer.arm.com/documentation/PRD29-GENC-009492/c/TrustZone-Hardware-Architecture
[205] G. Beniamini. (2017). *Trust Issues-Exploiting Trustzone TEEs*. [Online]. Available: https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html
[206] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "SANCTUARY: ARMing TrustZone with user-space enclaves," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/sanctuary-arming-trustzone-with-user-space-enclaves/
[207] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov, "Memory tagging and how it improves C/C++ memory safety," 2018, *arXiv:1802.09517*.
[208] *ARMV8.5—A Memory Tagging Extension*, ARM, Cambridge, U.K., 2019.
[209] K. Serebryany, "ARM memory tagging extension and how it improves C/C++ memory safety," *Login USENIX Mag.*, vol. 44, no. 2, pp. 12–16, Summer 2019.
[210] Q. Liu, V. Hagenmeyer, and H. B. Keller, "A review of rule learning-based intrusion detection systems and their prospects in smart grids," *IEEE Access*, vol. 9, pp. 57542–57564, 2021.
[211] T. Toth and C. Kruegel, "Accurate buffer overflow detection via abstract pay load execution," in *Proc. 5th Int. Symp. Res. Attacks, Intrusions Defenses*, 2002, pp. 274–291.
[212] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "STILL: Exploit code detection via static taint and initialization analyses," in *Proc. Annu. Comput. Secur. Appl. Conf. (ACSAC)*, Dec. 2008, pp. 289–298.
[213] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda, "Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2009, pp. 88–106.
[214] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos, "Network–level polymorphic shellcode detection using emulation," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2006, pp. 54–73.
[215] Q. Zhang, D. S. Reeves, P. Ning, and S. P. Iyer, "Analyzing network traffic to detect self-decrypting exploit code," in *Proc. 2nd ACM Symp. Inf., Comput. Commun. Secur. (ASIACCS)*, 2007, pp. 4–12.
[216] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos, "Comprehensive shellcode detection using runtime heuristics," in *Proc. 26th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, 2010, pp. 287–296.
[217] B. Gu, X. Bai, Z. Yang, A. C. Champion, and D. Xuan, "Malicious shellcode detection with virtual memory snapshots," in *Proc. IEEE Annu. Joint Conf. INFOCOM*, Mar. 2010, pp. 1–9.
[218] M. Polychronakis and A. D. Keromytis, "ROP payload detection using speculative code execution," in *Proc. 6th Int. Conf. Malicious Unwanted Softw.*, Oct. 2011, pp. 58–65.
[219] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: A detection tool to defend against return-oriented programming attacks," in *Proc. 6th ACM Symp. Inf., Comput. Commun. Secur. (ASIACCS)*, 2011, pp. 40–51.
[220] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, "ROPecker: A generic and practical approach for defending against ROP attacks," in *Proc. 21st Netw. Distrib. Syst. Secur. Symp.*, 2014. [Online]. Available: https://www.ndss-symposium.org/ndss2014/programme/ropecker-generic-and-practical-approach-defending-against-rop-attacks/

[221] E. R. Jacobson, A. R. Bernat, W. R. Williams, and B. P. Miller, "Detecting code reuse attacks with a model of conformant program execution," in *Proc. Int. Symp. Eng. Secure Softw. Syst.*, 2014, pp. 1–18.
[222] L. Cheng, K. Tian, and D. Yao, "Orpheus: Enforcing cyber-physical execution semantics to defend against data-oriented attacks," in *Proc. 33rd Annu. Comput. Secur. Appl. Conf.*, Dec. 2017, pp. 315–326.
[223] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-FLAT: Control-flow attestation for embedded systems software," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 743–754.
[224] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A.-R. Sadeghi, "LO-FAT: Low-overhead control flow ATtestation in hardware," in *Proc. 54th Annu. Design Autom. Conf.*, Jun. 2017, pp. 1–6.
[225] S. Zeitouni, G. Dessouky, O. Arias, D. Sullivan, A. Ibrahim, Y. Jin, and A.-R. Sadeghi, "ATRIUM: Runtime attestation resilient under memory attacks," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, Nov. 2017, pp. 384–391.
[226] G. Dessouky, T. Abera, A. Ibrahim, and A.-R. Sadeghi, "LiteHAX: Lightweight hardware-assisted attestation of program execution," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2018, pp. 1–8.
[227] P. Szor, *The Art of Computer Virus Research and Defense*. Upper Saddle River, NJ, USA: Addison-Wesley, 2005.
[228] A. Abbasi, J. Wetzels, W. Bokslag, E. Zambon, and S. Etalle, "On emulation-based network intrusion detection systems," in *Proc. Int. Symp. Res. Attacks, Intrusions Defenses*, 2014, pp. 384–404.
[229] N. Carlini and D. Wagner, "ROP is still dangerous: Breaking modern defenses," in *Proc. 23rd USENIX Secur. Symp.*, 2014, pp. 385–399.



QI LIU received the B.E. degree from Beijing Information Science and Technology University, China, in 2016, and the M.S. degree from the Karlsruhe Institute of Technology, Germany, in 2019, where he is currently pursuing the Ph.D. degree. His current research interests include industrial control systems (ICS) security, energy systems security, machine learning based intrusion detection systems (IDS), information and communications technologies in ICS, binary exploitation, and malware analysis.

KAIBIN BAO received the Ph.D. degree from the Karlsruhe Institute of Technology, Germany, in 2021. His dissertation was on the topic of non-intrusive appliance load monitoring for domestic buildings using convolutional neural networks. He is currently the Head of the Working Group Robust Secure Automation (RSA), which is part of the Research Area Advanced Automation Technologies (A2T) with the Institute for Automation and Applied Informatics (IAI), Karlsruhe Institute of Technology (KIT). His current research interests include center around cybersecurity for critical infrastructures and automation systems using machine learning methods.

VEIT HAGENMEYER received the Ph.D. degree from Universite Paris XI, Paris, France, in 2002. He is currently a Professor of energy informatics with the Faculty of Computer Science, and the Director of the Institute for Automation and Applied Informatics, Karlsruhe Institute of Technology, Karlsruhe, Germany. His research interests include modeling, optimization and control of sector-integrated energy systems, machine-learning based forecasting of uncertain demand and production in energy systems mainly driven by renewables, and integrated cybersecurity of such systems.