InvisiGuard: Data Integrity for Microcontroller-Based Devices via Hardware-Triggered Write Monitoring

Dongliang Fang, Anni Peng, Le Guan, Erik van der Kouwe, Klaus v. Gleissenthall, Wenwen Wang, Yuqing Zhang, Limin Sun

***Abstract*—Deeply embedded devices powered by microcontrollers are widely deployed. To protect them from exploitation, many lightweight defense mechanisms, such as control flow integrity, have been proposed. However, these defenses cannot provide data integrity—a security property of particular interest in mission-critical tasks. Conversely, existing defenses that provide data integrity are too expensive to deploy in the resource-constrained context of deeply embedded devices. In this paper, we propose *InvisiGuard*, a hardware-assisted, low overhead approach for data integrity. *InvisiGuard* leverages data watchpoints—a commonly available debug feature on microcontrollers—to automatically intercept write operations to critical variables. *InvisiGuard* then checks the legitimacy of the write instruction against an allowlist stored in a trusted execution environment (e.g., ARM TrustZone-M). By relying on the hardware to automatically intercept potentially dangerous instructions, *InvisiGuard* avoids heavy code instrumentation, as required by traditional solutions, making it suitable for resource-constrained microcontroller devices. We have implemented *InvisiGuard* on an ARM Cortex-M based development board and evaluated it with seven real-world firmware samples. Our experiments show that *InvisiGuard* reduces the runtime overhead by 56.99% and memory overhead by 77.37% compared with state of the art.**

***Keywords*—Deeply Embedded Devices, Data-integrity, Hardware-assisted solutions.**

I. Introduction

DEEPLY embedded systems powered by microcontroller units (MCUs) are increasingly infiltrating our everyday lives, for example, in healthcare, autonomous driving, and home security. With the wide adaption of the Internet-of-Things (IoT) technology, billions of these devices are deployed today [84]. This results in a huge attack surface, which makes it challenging to secure them. It therefore comes as no surprise that attackers routinely exploit firmware vulnerabilities—mostly based around memory safety—e.g., to remotely hijack the target device or launch denial of service (DoS) attacks [15], [28], [78], [42], [44].

To combat attacks in production, devices can be equipped with runtime security defenses. But, while there are many on-device defense mechanisms for traditional computing platforms [1], [30], [18], [67], [51], [5], these defenses are not suitable for an embedded setting, as they tend to consume more resources than can be afforded (e.g., MCU devices commonly have less than 256 kB of RAM). In response, several lightweight mechanisms specifically designed for MCUs have been proposed. For example, Silhouette [85] and μRAI [7] effectively prevent corrupted return addresses on the stack, thus enforcing backward control flow integrity (CFI) of the firmware. Although CFI is an important security property, it alone cannot defeat the dynamic and sophisticated cyberattacks we are experiencing today [41], [65], [50].

This problem is particularly pressing for Cyber-Physical Systems (CPSs) and Industrial Control Systems (ICSs), where attackers can wreak havoc using *data-only attacks* [41], [19], [16], [35], [36], even when CFI is perfectly honored. For instance, attackers can gain unauthorized access by overwriting variables used for authentication without knowing the credentials. An example of this is the successful attack on the Schneider Electric Modicon M221 PLC controller, where the attacker overwrote the password hash [41]. In industrial control applications, corrupting semantically critical variables or memory has become a preferred exploitation method [29], [32], [2], [20]. Such attacks can cause devastating physical malfunctions, resulting in explosions, collisions, or false medical treatment [73]. Worse, in a data oriented programming (DOP) attack [36], [35], attackers can fully control the target device by chaining together multiple data-oriented gadgets, using a gadget dispatcher. Although there have been several proposals for data integrity protection, they are geared towards traditional computing platforms and have quite limited real-world adoption due to the prohibitive performance overhead [27]. It is therefore unsurprising to observe even worse adoption in resource-constrained embedded systems [49].

Lightweight Selective Protection With *InvisiGuard* In this paper, we present *InvisiGuard*, a lightweight, hardware-assisted defense mechanism to defeat data-only attacks on MCU devices. Unlike remote attestation solutions [4], [57], [58], [54], which are "offline" and detect attacks **after the fact**. *InvisiGuard* provides proactive lightweight "online" protection. Compared to traditional software-based online protection mechanisms(e.g., Samurai [61], OAT's CVI [76], etc.), *InvisiGuard* further significantly reduces the overhead by leveraging hardware features(i.e., DWT and TrustZone). To remain feasible in a low-resource setting, *InvisiGuard* only selectively protects *security-critical data* (we will refer to this data as *critical variables*), e.g., hash and flags used in an

Dongliang Fang and Limin Sun are with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China, and also with the School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China

Anni Peng and Yuqing Zhang are with National Computer Network Intrusion Protection Center, UCAS, Beijing, China

Le Guan and Wenwen Wang are with University of Georgia, USA

Erik van der Kouwe and Klaus v. Gleissenthall are with Vrije Universiteit Amsterdam, The Netherlands

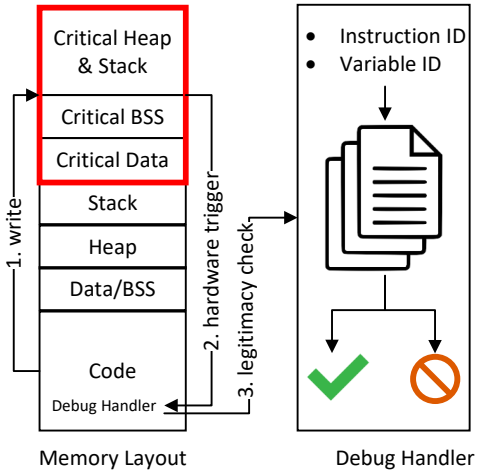


Fig. 1: High-level idea.

authentication process, as well as the variables determining the force and direction of an actuator. The idea of selective protection is not new, but existing solutions offer insufficient performance benefits, mainly because existing work heavily relies on software-based instrumentation.

InvisiGuard avoids costly instrumentation by proposing a lightweight data integrity mechanism, which combines two hardware features in a novel way: 1) data watchpoints (a commonly available debug feature on microcontrollers that enables automatic interception of chosen write or read operations), and 2) Trusted Execution Environments (TEE). At a high level, *InvisiGuard* leverages the watchpoint mechanism to construct a single continuous memory region called *GuardedZone*, where *InvisiGuard* places critical variables. Whenever writing a variable in the *GuardedZone*, the hardware automatically traps the execution via a monitor exception handler (see Section II), allowing *InvisiGuard* to transparently check the legitimacy of the write request against an allowlist stored in the TEE. This idea is illustrated in Figure 1, where we highlight the *GuardedZone* in a red box. By relying on the hardware to automatically intercept potentially dangerous instructions, *InvisiGuard* thus avoids heavy code instrumentation, as required by traditional solutions, making it suitable for resource-constrained microcontroller devices.

To create a design with this approach, we need to solve a number of new challenges. For example, the data watchpoint module cannot monitor an arbitrary number of variables scattered across the address space. We address this challenge by relocating all critical variables to a single memory region (i.e., *GuardedZone*). In particular, we divide the memory region into three sub-regions based on the variable's type and track the bounds of variables accordingly (see Section V-C). Another challenge we face is the fact that, when the hardware intercepts potentially dangerous instructions, its exception context does not contain all the necessary information (e.g., the responsible write instruction and target address). *InvisiGuard* needs to recover them through additional analysis to assist legitimacy checks. In particular, it uses stacked context to decode the write instruction to retrieve the base register and immediate offset, which are then used to calculate the target memory address (see Section V-D). To address challenges due to the asynchronous nature of exceptions and the multiple addressing modes offered by ARM, we use a compiler backend plugin that injects NOP instructions and regulates the emitted instructions (see Section V-D). We show that we can address these challenges with minimal hardware requirements.

We built a prototype of *InvisiGuard* on the STM32L562E-DK development board and evaluated it using seven real-world firmware samples. Our results show that *InvisiGuard* is effective in preventing data-only attacks, and has better performance with less memory overhead compared to state-of-the-art solutions. In particular, compared to critical variable Integrity (CVI) proposed in OAT [76] (see Background II for a brief introduction), *InvisiGuard* reduces the runtime overhead by 56.99% and memory overhead by 77.37%. We also find that *InvisiGuard* has no false positives and only a few false negatives. In summary, we make the following contributions.

- We propose a lightweight data integrity mechanism that selectively protects critical data only and minimizes instrumentation overhead with hardware assistance, thereby fulfilling the tight resource budget on resource-contained MCU devices.
- We implement our idea on an ARM Cortex-M based MCU by addressing a set of non-trivial challenges unique to our design and the ARM Cortex-M platform.
- We evaluate our prototype on real-world firmware, and the results show substantially lower overhead compared to the state-of-the-art.

II. Background

Deeply Embedded Systems Embedded systems are widely used in IoT, industrial control, automotive, healthcare, and other fields. Following the categorization proposed by Abbasi *et al.* [3], in this paper, we focus on *deeply embedded systems*, which are a subset of embedded systems with the following characteristics: (1) They are powered by a low-cost system-on-chip (SoC) (e.g., MCU), which integrates a processor, memory, and other peripheral devices onto a single chip with constrained resources; (2) The applications often lack a user interface or have uncommon ones to reduce resource consumption; and (3) run on extremely minimal OSs (such as FreeRTOS) or no OS at all (i.e., baremetal);

Data Watchpoint Data watchpoints are a common hardware-enabled debugging feature in MCU-based devices. With this feature, whenever an interesting variable is accessed, the hardware halts execution and generates a signal to the external debugger (e.g., a USB dongle), which then controls the debugging target. This feature is widely supported by mainstream MCUs, including ARM [11], RISC-V [71], and MIPS [47]. For Cortex-M MCUs, data watchpoints are implemented by the *Data Watchpoint and Trace Unit* (DWT), an ARM Intellectual Property for data watchpoints, execution tracing, system profiling, and many helpful debugging functions.

DWT monitors a contiguous memory region based on the access type (i.e., read, write, or execute). The range is configured using a pair of comparators. The board we used for our experiments supports 16 pairs of comparators, indicating that 16 memory regions can be monitored simultaneously. A match can either trigger an external debugger (halt mode), or raise a **debug monitor exception** (monitor mode), depending on the configuration of the *Debug Exception and Monitor Control Register* (DEMCR). *InvisiGuard* configures DWT to work in monitor mode. Specifically, *InvisiGuard* performs the legitimacy check of variable accesses in the handler of the debug monitor exception (i.e., `DebugMon_Handler`).

ARM TrustZone-M ARM TrustZone-M is specifically designed for ARM Cortex-M series MCUs, providing a hardware-based TEE for high-assurance tasks. The feature divides the system into two execution environments: the secure world and the non-secure (or normal) world. The processor is time shared between the two environments. Correspondingly, the system resources (such as memory and peripherals) are split into secure ones and non-secure ones. When the processor runs in the normal world (non-secure state), it can only access non-secure resources. When the processor runs in the secure world (secure state), it can access all the resources.

The transition between the two worlds is realized in two ways: secure calls or exceptions. Secure calls allow the normal-world firmware to invoke functions implemented in the secure world via well-defined entry points. The entry is guarded by a special instruction called *Secure Gateway* (`SG`). Exceptions can also cause cross-world transitions depending on whether they are banked or not. Banked exceptions have separate resources in both worlds to handle their own exceptions, so world transition is unnecessary. Non-banked exceptions such as debug monitor exception can only target one security state. As such, by configuring the DEMCR [10], we can also force the debug monitor exception to always target the secure world, even if the processor is currently running in the normal world.

CVI Mechanism One important contribution of OAT [76] is to propose selective data protection – CVI (Critical Variable Integrity). The core idea is to check if the value of a critical variable at the use-site (i.e., read instructions) remains the same as recorded at the last legitimate define-site (i.e., write instructions). The advantage of the CVI design is that it does not need to instrument non-critical instructions. However, both the define-site and use-site of critical variables must be instrumented to record the written value and to verify the data integrity, respectively. This instrumentation incurs considerable overhead. We can significantly reduce the overhead using our proposed hardware-supported mechanism.

III. Challenges

Problem Setting We use a code snippet of a real-world MCU firmware as an example to explain the problem and illustrate how our solution advances the state of the art. This firmware turns an Arduino MCU into a syringe pump, which is used to control the quantity of fluid to dispense or withdraw [62]. Syringe pump has been widely used in chemical and biomedical applications and this firmware has also been evaluated in related work [4], [76], [77]. In Listing 1, the core program logic is implemented in the function `userOperation()`. The code snippet is mostly copied as is, except that we added user authentication logic at line 12 based on [19]. Adding authentication is reasonable because some commercial syringe pump products can be controlled from remote smartphones [25], and authentication processes are commonly used in embedded devices [13].

```
1   typedef struct {
2       char direction;
3       long amount;
4   } cmd_t;
5   long ustepsPerML = 100;
6   void userOperation(){
7       cmd_t cmd;
8       int authenticated = 0;
9       char buf[100]; // input buffer
10      while (!authenticated){
11          getInput(buf);  // BUGGY!!!
12          if (auth(buf)){
13              authenticated = 1;
14              extractCommand(&cmd, buf);
15          }
16      }
17      if (authenticated){
18          bolus(cmd.direction, cmd.amount);
19      }
20  }
21  void bolus(char direction, int amount){
22       /* set operation direction */
23       /*......*/
24      /* change units to steps, move stepper */
25      long steps = amount * ustepsPerML;
26      for(long i=0; i < steps; i++){
27         /* ...... */
28      }
29  }
```

Listing 1: Code snippet of the open source Syringe Pump firmware.

This function receives user input and stores it in a buffer (line 11). The user input includes the credential and the command. Then, if authentication passes, the flag `authenticated` is set (line 13) and the command is extracted to `cmd` (line 14). Finally, based on direction indicated by `direction` and the amount indicated by `amount`, the firmware drives the syringe pump to dispense or withdraw liquid (lines 21-29). There is a buffer overflow vulnerability in the function `getInput()` which allows attackers to overwrite the **semantically critical** variable `authenticated`. Since the attacker does not know the credentials, the firmware will fail the authentication at line 12. However, if the attacker overwrites `authenticated`, in the subsequent `while` iteration the firmware directly jumps to line 17, followed by the execution of arbitrary attacker-specified commands. Therefore, assuming that the attacker can send data to the syringe pump, they can exploit this buffer overflow bug to issue any commands without knowing the user credential. This threat model is realistic even in an isolated network. As shown in a recent survey [31], insider attacks are top security challenges in an air-gapped environment. Note that this is a data-only attack that does not violate control-flow integrity, thereby bypassing existing

TABLE I: Comparison with related work in selective write integrity protection (✓: need instrumentation, -: no instrumentation needed, □: implicit instrumentation).

Name	Instrumentation Points				Protection Scope	
	Allowed Writes	Other Writes	Allowed Reads	Other Reads	Target	Property
WIT [5]	✓	✓	-	-	all data	write integrity
Datashield [17]	✓	✓	-	-	critical data	write integrity
Samurai [61]	✓	-	✓	-	critical data	write integrity
OAT [76]	✓	-	✓	-	critical data	write integrity
InvisiGuard	□	□	-	-	critical data	write integrity

CFI protection. To prevent these attacks from happening, the key research question is how to protect the integrity of **semantically critical data** in the system.

Selective Protection Full protection of non-control data offers strong security, but the required instrumentation and legitimacy checks at almost all memory access sites are too costly (e.g., 44%-103% overhead reported in DFI [18], 116% overhead reported in CETS [52]), which is infeasible for practical usage, especially in resource-constrained embedded devices. Indeed, low overhead is the most important prerequisite for a practical solution. To make it lightweight, a common solution is to selectively secure only the essential subset of non-control data for a specific program. Selective protection involves sacrificing strong security guarantees for practicality, but it has been shown to be valuable by many studies [61], [60], [17], [76]. Selective protection is reasonable, because 1) non-control data attacks are more likely to target specific types of data (e.g., data with a long lifetime and application-specific security-critical data [19]), and 2) safeguarding a subset of data can offer indirect protection to other data. For the example in Listing 1, instead of protecting all non-control data, only selectively protecting long-lived global data (i.e., `ustepsPerML`) and partly semantic critical data (i.e, `authenticated` and `cmd`) will greatly raise the bar for successful exploitation. Meanwhile, we notice that other non-control data (e.g., `steps`) in the function `bolus` will be indirectly protected based on integrity constraints on the function's inputs (i.e., the global variable – `ustepsPerML` and the function parameter – `amount`). Determining which variables should be selected generally requires domain knowledge of specific application semantics (see Section V-A).

Write Integrity Enforcement We distinguish between two security properties: *read integrity* and *write integrity*. The former means there is no unintended memory read operation while the latter means there is no unintended memory write operation. The notion of write integrity was first proposed in WIT [5], which enforces write integrity. In the setting targeted by our approach, write integrity is more important than read integrity because 1) DOP attacks [36] require violations of write integrity; 2) read accesses are far more frequent than write accesses [72], which makes their protection more expensive and less practical; and 3) existing work on general platforms shows that write integrity can be enforced, but incurs considerable overhead. Specifically, the reported performance overhead of WIT [5] is around 5-25% for the SPEC benchmark. Reducing this overhead makes our approach more likely to be deployed in practice.

However, reducing overhead through selective write integrity enforcement is far from trivial. Given a critical variable, besides the legitimate instructions that are allowed to write to it (i.e., "allowed writes" in Table I), its attack surface includes all other write instructions (i.e., "other writes" in Table I). As a result, a solution needs to ensure that modifications made by *all* the write operations are checked. In the example of Listing 1, if we want to ensure the write integrity property of the variable `authenticated`, only instrumenting lines 8 and 13 is not enough. Rather, we need to instrument all *other writes* (i.e., lines 11, 12, 14, etc.,) so that illegal modifications made by them can be detected. In C/C++, since it is hard to statically determine whether a pointer can or cannot point to a critical variable, widespread instrumentation seems to be inevitable.

Comparision With Prior Efforts In Table I, we summarize prior solutions on selective write integrity protection and compare them with *InvisiGuard*. As a baseline, WIT [5] performs system-wide program analysis and instruments all memory write instructions so that each of them only accesses a predetermined set of objects. DataShield [17] (in write protection mode) improves WIT by only protecting critical data. It also distinguishes *other writes* from *allowed writes* and only enforces a coarse-grained check for *other writes*. However, it still needs to apply system-wide instrumentation. OAT [76] and Samurai [61] completely avoid instrumentation at *other writes*, at the expense of instrumenting *allowed reads*. The basic idea is to selectively track the data flow of critical variables and check if the value at a use-site remains the same as that of the last legitimate define-site. To this end, they need to instrument both the *allowed writes* (to record the originally written values) and the *allowed reads* (to compare values). In the example, to protect `authenticated`, it needs to instrument both lines 8,13 to record a value copy at allowed writes, and lines 10,17 to perform integrity checks at allowed reads. Lastly, in *InvisiGuard*, we remove *all* the explicit instrumentation at *all* the memory access points. Therefore, no instrumentation is added in the example. Only when a critical variable is written would a check be triggered by the data watchpoint unit. In this sense, the instrumentation in *InvisiGuard* is considered implicit and binds to write operations. As we will discuss in Section VI-B, this can significantly reduce run-time overhead because 1) checks are performed only when necessary and 2)

the number of write operations is much fewer than that of read operations.

Why Not Place Critical Variables in TEE Since our solution relies on a TEE, one may question why not directly place critical variables in the secure world so that the compromised firmware cannot access them. There are two options following this idea. First, we can instrument critical instructions so that they are redirected to the secure world via secure calls. This approach necessitates instrumenting all define/use sites of critical variables, incurring a similar overhead of OAT's CVI. Second, we can partition the program logic into a secure part and a non-secure part, and then put the secure part in the TEE so that bugs in the non-secure part cannot infect the rest. Such partitioning requires a thorough refactoring of the existing code and oftentimes involves substantial manual effort. Moreover, it inevitably enlarges the Trusted Computing Base (TCB) since more code is placed in the TEE. In contrast, *InvisiGuard* provides a compiler-based framework that automatically transforms MCU firmware into a secure version, thereby avoiding re-engineering efforts needed to protect a large legacy C code base for MCU devices.

IV. Overview

InvisiGuard provides a lightweight data integrity mechanism by selectively protecting critical data and minimizing instrumentation overhead using hardware assistance. This allows *InvisiGuard* to respect the tight resource budget on resource-contained MCU devices. Figure 2 shows the workflow of *InvisiGuard*. Given the firmware's source code, with minimal annotations, our custom LLVM-based compiler automatically compiles it into a hardened firmware.

Our compiler pass first extracts an initial set of critical variables, marked as such through annotations. Then, it expands the set by adding variables based on a data dependence graph. The full set of critical variables includes all variables that might have a data flow into the initial ones, and therefore should also be protected (Section V-A). Then, for each write instruction, we calculate a list of data objects that it can access. If this list contains any critical variable, the instruction is considered critical. For each critical instruction, we select all the critical variables and store them in an *allowable write target table*, which is later compiled into the TEE software (Section V-B). To be able to repurpose DWT for efficient selective write integrity enforcement, all critical variables must be stored in contiguous memory ranges, regardless of whether they are on the stack, on the heap, or in global memory. Our *critical variable relocation* module (Section V-C) achieves this. At runtime, accesses to critical variables trap to the secure world exception handler, which checks whether the write is legitimate according to the allowable write target table (Section V-D).

Threat Model We consider an attacker who can feed arbitrary data to an I/O interface of the target device, either remotely (e.g., over the network) or locally (e.g., over the UART interface). The firmware contains memory-related vulnerabilities in either the kernel or application code. Therefore, attackers can bypass mitigations enforced by kernel, e.g., memory isolation with MPU. We focus on protecting *critical variables*, which are critical for the correct operation of many cyber-physical systems and IoT applications [19]. Numerous existing studies [41], [29], [32], [2], [20] have shown that this is a real threat. We do not consider protecting code pointers, as considerable orthogonal work on CFI of MCU devices exists [85], [7]. We assume such protection is already in place. We also assume the trusted software in the TEE is bug-free and isolated from the normal-world firmware, as important metadata such as the allowlist is stored there. Considering the small code base of the TEE-side software and the limited attack surface, this is a reasonable assumption, well accepted by existing TEE-based work [76], [59]. We do not consider low-level physical attacks, such as connecting to a JTAG debugger to re-program the firmware. Since MCU devices typically run on SRAM, RowHammer attacks that mainly target dynamic memory (DRAM) are out of scope. Finally, we assume our compiler passes are free of bugs.

V. Design

A. Critical Variable Selection

Full data flow integrity is expensive, and not viable on resource-constrained embedded systems. To enable efficient data integrity, our approach employs a selective mechanism. Generally, selecting variables heavily depends on application-specific semantics and protection goals. For example, DynPTA [60] proposes to selectively protect secrets (e.g., private key) in SSL programs, Samurai [61] proposes to protect heap metadata in malloc/free APIs, and KENALI [72] proposes to protect access-control related data in a kernel. In our design, we assume *critical variables* have been provided as such, either manually by the programmer using explicit annotations, or using an automated system. Determining which variables are critical and should be selected is orthogonal to our work, and this is not part of our contribution.

Existing works offering selective protection [19], [17], [76], [82] offer empirical guidance to help select critical variables. Among these, two notable semi-automatic tools [17], [76] are available. The approach involves two steps: 1) building an initial set based on manually specified variables, and 2) expanding this set by adding new variables that *may* influence the initial set using the data dependency graph. In our implementation, we have integrated one such tool – OAT [76], which is open source and also targets embedded programs, for ease of development. We still need to specify an initial variable set manually to use the tool. For this goal, we use the following high-level guidance: 1) all global variables are selected for their long lifetime, 2) variables with important semantics, e.g., authentication data, control command, etc., are also selected. Other details are elided as it is the same with existing work.

B. Allowable Write Target Extraction

Our next step is to determine which parts of the code can legitimately write to critical variables (CVs). We then create a lookup table that will be used subsequently for enforcing write integrity.

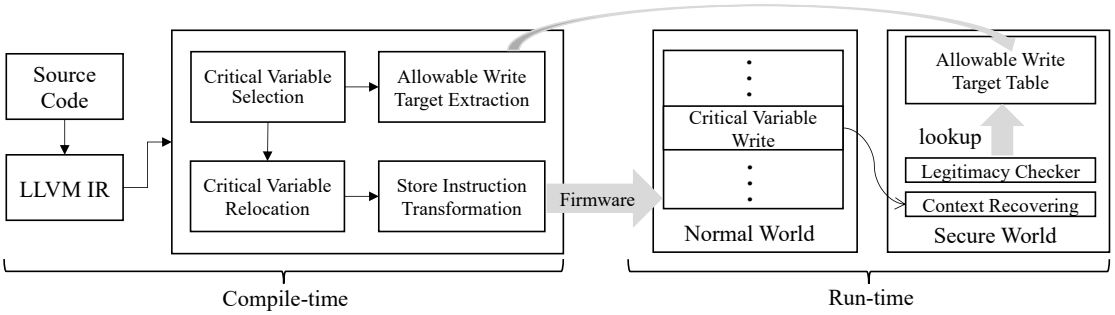


Fig. 2: Workflow of *InvisiGuard*.

This step starts by extracting an initial set of critical variables (CVs) from a combination of manually annotated variables and global variables (see Section V-A). Our compiler automatically expands this initial set of CVs. The expansion process primarily takes into account other variables that can influence the value of CVs, such as directly assigning another variable to a CV. Specifically, our custom compiler first constructs a data dependency graph, which is then used to extract CVs' dependent variables, following a backward traversal for each CV. These dependent variables will be included in the set. We further use standard Andersen's points-to analysis to construct a global points-to map. If any pointers in the map can point to CVs, we also include these pointers in the set. The expansion process is iterative and continues until the set of CVs stops growing.

We then iterate through all write operations in the program's intermediate compiler representation, including both direct memory stores and compiler intrinsics that perform stores (such as `memcpy` and `strcpy`). For each write operation, we determine whether it can target a critical variable (CV) and, if so, we consider the instruction to be critical. We construct an *allowable write target table* that maps each critical instruction to a set of critical variables they can write to. This is a hash table indexed, by the address of the critical instruction. We represent critical variables by assigning to each variable a static identifier, denoted as *VarID*. Note that at runtime, a *VarID* can represent multiple dynamically allocated instances of an object. Our table contains a list of permissible *VarIDs* for each write instruction.

We use Listings 2 and 3 to demonstrate our approach. In Listing 2, we first identify `criVar` as a critical variable because of the manual annotation. Then, our points-to analysis performs precise tracking of four types of operations, namely *"address"(p=&q), "copy"(p=q), "load"(p=*q), and "store"(*p=q)*, to propagate and update points-to information. We perform this analysis at both inter-procedural and intra-procedural levels, which allows us to construct a comprehensive global points-to map. Using this map, we can determine that pointers `ptr1`, `ptr2`, and `ptr3` point to the same memory location as the variable `criVar`. Because they can write to `criVar`, *InvisiGuard* considers the store instructions at lines 2 and 12 to be critical stores. In Listing 3, we can determine that `example->ptr` can point to the same

```
1  void modify_ptr(int *ptr3) {
2    *ptr3 = 10;
3  }
4
5  int __attribute__((annotate("critical"))) criVar;
6  int *ptr1 = NULL, *ptr2 = NULL;
7  ptr1 = &criVar;
8
9  modify_ptr(ptr1);
10
11 ptr2 = ptr1;
12 *ptr2 = 10;
```

Listing 2: Code snippet to demonstrate how indirect access works when pointers from both intra-procedural and inter-procedural scenarios.

```
1  int __attribute__((annotate("critical"))) criVar;
2  int b;
3  struct example { int *ptr; } *example;
4  b = 100;
5  example = malloc(sizeof(struct example));
6  example->ptr = &cv;
7  /* ... */
8  *example->ptr = b;
```

Listing 3: Code snippet to demonstrate how to obtain allowable write target table.

memory location as `criVar` using the global points-to map. We expand the set of critical variables to include variable `b` due to line 8, where `criVar` can be set to `b`. Consequently, *InvisiGuard* safeguards two critical variables, namely `b` and `criVar`, assigning each a unique *VarID*. *InvisiGuard* then determines the allowable write targets for each critical store instruction. Specifically, the allowable write target of store instruction in line 4 is *VarID* of the variable `b`, and the target of line 8 is *VarID* of the variable `criVar`. Using *InvisiGuard* to analyze the code, we successfully obtain the expected results.

C. Variable Relocation

Due to the limited number of comparators in DWT (see Section II), we cannot monitor a large number of variables if they are scattered across the address space of the firmware. We address this issue by relocating all critical variables to a single memory region called *GuardedZone*. *InvisiGuard* prevents any

illegal write attempts to this zone. A critical variable can appear in any of the stack, heap, or data/BSS segments. Correspondingly, we split *GuardedZone* into three sub-regions, as shown in Figure 1.

Critical Variables on Stack and Heap We treat critical stack variables as dynamic objects. That is, they are allocated on the heap on function entry and de-allocated on function exit. Together with the original critical heap variables, they are stored in a sub-region of *GuardedZone*. The heap manager which manages both critical stack variables and heap variables is implemented based on FreeRTOS heap_2 [8], a lightweight heap implementation. The metadata used to manage the heap is always marked as critical and is not an allowable write target for any non-TrustZone code.

Critical Variables in Data and BSS Segments Critical global variables are allocated in a dedicated data segment (`_critical_data`) or a BSS segment (`_critical_bss`). These two segments are mapped to two separated sub-regions of the *GuardedZone*. Following the common practice in MCU programming, during booting, the chip initialization code needs to copy `_critical_data` from flash to the corresponding sub-region and to zero out the sub-region for `_critical_bss`.

Tracking Object Bounds The secure world needs to maintain the memory ranges for each critical variable. For global variables in the data/BSS segments, we directly get their static boundary information $\{varID : (base, bound)\}$ from the symbol table. For critical variables on the heap or stack, since they are allocated on demand, their boundary information must be reported to the secure world dynamically. As such, we implement `trusted_malloc` and `trusted_free` as secure functions in TrustZone, which we use to handle both the heap and stack objects. Note that, although these functions run in the secure world, they manage non-secure heap memory for the normal-world firmware. This design leads to an important performance advantage: since a heap manager has to access heap memory to manage metadata, if we implement it in the normal world, numerous DWT exceptions would be unnecessarily triggered. In contrast, implementing it in the secure world suppresses such exceptions. This is because, in the secure world, we are free to manage DWT watchpoint configurations. Specifically, before storing the metadata, we configure the DWTs to disable the watchpoint mechanism on the memory being written to and enable it once the writes are completed.

D. Write Integrity Check in TEE

We configure watchpoints to deliver an exception to our secure-world handler on any write attempt to the *GuardedZone*. *InvisiGuard* performs a legitimacy check in the debug monitor handler in the secure world. The secure world can access the monitored memory without triggering exceptions. Our threat model assumes a powerful attacker who can arbitrarily write anywhere. To prevent them from disabling DWT at runtime, we use TrustZone to restrict access to DWT registers in the normal world after system initialization. We also use TrustZone to prevent attackers from manipulating the secure runtime, including the implementation of the debug monitor handler and the runtime data such as the allowable write target table and boundary information of critical variables.

Inside the handler, we need to 1) recover the addresses of responsible write instruction as well as the write target, and 2) look them up in the allowable write target table to decide whether the access should be allowed or not. Since the handler runs in the secure world and maintains runtime information, we call it and the associated code and data structures *secure runtime* in this work.

Recovering Addresses of Write Instructions & Targets From Exception Context As with any other architecture, exception context contains rich information related to the pre-exception processor state. We need to recover the address of the responsible write instruction as well as the write target from it. In ARM Cortex-M processors, on taking an exception, the hardware stores processor context onto the pre-exception stack. This operation is referred to as *stacking*, and the processor context stored in the stack is referred to as an *exception stack frame*. The exception stack frame includes the following register values at the time when an exception is taken: `R0–R3`, `R12`, `LR`, `PC`, `xPSR`, and floating point registers if the floating point is active. Here, the value corresponding to `PC` register is the address to return to when the exception is completed.

It seems straightforward to use the value corresponding to `PC` register as the address of the write instruction, as it should point to the instruction responsible for the exception. Unfortunately, the debug monitor exception is asynchronous, so the timing of the exception is not necessarily right after the monitored memory is accessed. Therefore, the processor context stored on the exception stack frame may not directly contain the information we are interested in. Based on our observation, there is a delay of at most two instructions. This also agrees with the pipeline design of the Cortex-M core; the manual [9] states explicitly that the maximum delay is two instructions (in §4.7). As such, the address of the faulting write instruction is within three instructions backward along the execution trace from the `PC` value obtained from the exception stack frame. If there exists a branch instruction among these instructions, it becomes hard to backtrace the original jump site and thus the write instruction.

We address this challenge by injecting two `NOP` instructions (i.e., `MOV R0, R0`) after every critical write instruction. As a result, the secure runtime only needs to linearly search for the nearest write instruction backward from the address obtained from the exception stack frame. We argue that it is very difficult, if not impossible, to launch an effective attack within two delayed instructions in practice. For example, it is rare that a critical variable is used immediately after a data corruption point.

There are a few corner cases to consider when the debug monitor exception is triggered by a non-critical write instruction that writes to a critical variable (due to memory errors). First, if a critical write instruction follows the non-critical write instruction immediately, our backward searching algorithm may fail to locate the real write instruction. Consider

the code snippet in the Listing 4. Normally, the first `write_non_critical` would write to non-critical variables, and second `write_critical` would write to critical variables. However, due to data corruption, the first `write_non_critical` now writes to a critical variable, which will then trigger a DWT interrupt and incur a delay, and stop at the first `NOP` instruction for example. As a result, *InvisiGuard* would recover and identify the responsible write instruction being the second `write_critical` instruction, which is allowed to modify the critical variable. This may lead to a possible bypass. To solve this problem, for each critical write, we check whether its previous two instructions contain a non-critical write. If so, we also insert two NOPs after the non-critical write instruction. For this example, *InvisiGuard* adds two NOPs after the first `write_non_critical` instruction. Second, since non-critical write instructions are not instrumented, our method may fail to locate a write instruction at all. If this happens, an alert is given. Third, if we find a write instruction, we check whether it is indeed a critical write instruction. Otherwise, it generates an alert. The user can also configure what to do depending on the specific scenario, including aborting the program, dropping privileges or just recording the alert.

```
1 write_non_critical value1 addr1
2 write_critical value2 addr2
3 NOP
4 NOP
```

Listing 4: An example for potential bypass.

After obtaining and confirming the address of the critical write instruction, we recover the address of the write target. We disassemble the write instruction and calculate the target address based on its addressing mode. ARM offers multiple addressing modes [10] (e.g., scaled, pre/post-indexed, etc.), some of which involve complex address calculations (e.g., barrel shifter). To avoid unnecessary complexity, we enforce restrictions in the compiler backend when emitting code for critical write instructions. In particular, the addressing mode is always the simplest one, in which the target address is calculated by adding an immediate offset to a base register. Assuming this encoding, we can quickly recover the target address. It is worth noting that only using simplified encoding generates less efficient code. However, this only applies to critical write instructions.

Checking Write Targets Against the Allowable Write Target Table To validate a write instruction, the secure runtime needs to check the write target against the allowable write target table. However, there is a gap between the two—the written targets obtained previously are actual addresses while the allowable write targets are a list of *VarIDs*. To bridge this gap, the secure runtime maintains the memory ranges for each *VarID* at runtime. As such, the check can be performed by matching the address against a list of memory ranges.

We maintain the memory ranges for each *VarID* using different strategies depending on the variable types. For global variables, a variable always maps to a fixed location. Therefore, the memory range is static and extracted from the symbol table. For heap and stack variables, since they are dynamically managed, we instrument all their allocation points, so that the memory ranges are always reported to the secure runtime. Note that one *VarID* could map to multiple instances of the same object when they are dynamically allocated.

Library Functions In addition to individual memory write instructions, libc functions such as `memcpy` and `strcpy` also access critical variables. They are recognized as memory intrinsic functions in LLVM. We do not instrument store instructions inside these functions and run them in the normal world, because doing so will generate an excessive number of exceptions. We instead replace these library functions with secure functions implemented in the secure world, leading to two performance improvements. First, the security check can be conducted at function entry for the entire target buffer instead of individual bytes. Second, individual store instructions will not generate exceptions since they run in the secure world. When performing legitimacy checks, the secure runtime can directly obtain the address of the write target from the parameters. To recover the address of the call to memory intrinsic functions (i.e., `BL memcpy`), we leverage the *intra-procedure call scratch register* `R12`, which is safe to use before any function calls.

VI. Evaluation

We implemented a prototype of *InvisiGuard* on top of LLVM 10.0, with about 3,300 source lines of code (SLOC), excluding the reused code. Our points-to analysis is based on SVF [75], and we reused some code for selecting critical variables from OAT [76] and backend analysis from Silhouette [85]. Our prototype runs on a popular ARM development board, the STM32L562E-DK discovery kit. It is equipped with an ARM Cortex-M33 core with TrustZone and DWT support, integrating a 512 kB Flash memory and a 256 kB SRAM. We did not use any board-specific features. Therefore, our prototype can be easily ported to other ARM Cortex-M chips.

Experiment Setup We evaluated our prototype against seven real-world firmware samples. The results are compared with OAT (with only CVI enabled) [76] and the baseline measurement obtained from running the firmware without any protection. CVI is the state-of-the-art solution for protecting critical variables in deeply embedded systems. We extracted the reported performance data from related work [5], [17], [61] and found CVI has the lowest overhead. All experiments were conducted on the same development board.

For a fair comparison, we used the same static analysis tool to identify and extend critical variables. In this way, both OAT and *InvisiGuard* protect the same set of critical variables. We first measured how our compiler pass performs in extending the initial critical variables and associating them with the critical store instructions (Section VI-A). Then, we measured the imposed performance and memory overhead (Sections VI-B and VI-C). Finally, we validated the effectiveness of *InvisiGuard* by running manually crafted exploits to corrupt critical variables. Our attack also tried to write to DWT registers to disable *InvisiGuard* directly (Section VI-E).

Firmware Samples We tried to evaluate with the same set of firmware samples that come with OAT. Unfortunately, this turned out to be more challenging than expected. Specifically, while OAT was designed for Cortex-M devices, the authors implemented their prototype on a Cortex-A device due to the lack of TrustZone-M devices at that time. As a result, two of their firmware samples that involve some POSIX APIs cannot be easily ported to any MCU runtime. Nevertheless, we included the remaining three samples in our evaluation. We additionally collected four more firmware samples commonly used in related work (e.g., [7], [4], [21]). The average size of these samples is 107.63 kB (compiled with default compiler options). They appear to be small in size compared with traditional software. However, we found in the source code very complex operation logic, common C/C++ programming features such as pointers, indirect calls, aliases, and also some complex algorithms such as SHA256. Based on another recent study on firmware security [83], the average size of real-world MCU-based BLE firmware is less than 122.7 kB. Therefore, these samples represent a reasonable level of variety, covering application scenarios such as ICS and smart homes. The details of each sample are explained below.

- **Syringe Pump** [62] has been explained in our motivating example.
- **Light Controller** [40] is used to control the switches in smart homes. It parses user input to locate the device and take actions (e.g., turn on/off).
- **Alarm** [23] is used for automated environmental monitoring. It reads data from presence-detection sensors. When user-specified conditions are met, it will trigger an alarm.
- **PinLock** [21] runs on a smart lock. It receives a PIN input from a user and performs a SHA256 hash calculation to get a 32-byte hash value. Then the hash value is used to compare with a pre-computed value. If they match, it performs `unlock` function.
- **RF_Door_Lock** [34] is similar to PinLock but with a different control logic. It directly compares user inputs with the stored password and performs the requested operation (e.g., opening the door, changing the password, etc.).
- **Steering Control** [48] is used in the field of autonomous driving. It receives commands from a user to control the angle of the steering wheel and the throttle of a vehicle.
- **PID controller** [24] is widely used in industrial control systems. In each scan cycle, it reads inputs from sensors, calculates proportional, integral, and derivative responses, and uses the responses to drive the actuators.

A. Accuracy of Static Analysis

Our analysis pass identifies critical variables to be protected and correlates them with store instructions. Therefore, its soundness and completeness are critical for the secure operation of firmware at runtime. If we miss checking a critical variable, the attacker could corrupt it without being detected. On the other hand, if our allowlist is incomplete, legitimate access could be blocked. Considering that MCUs are commonly used in mission-critical applications that cannot tolerate unexpected suspensions (due to false positives), our implementation follows a conservative construction that may relax the allowlist. In general, static analysis including alias analysis is an undecidable problem [46]. Therefore, rather than formal proof, this section presents our empirical study on the performance of *InvisiGuard*'s analysis pass in expanding critical variables and constructing the allowlist and discusses its implication to security.

TABLE II: Statistics on critical variables (CVs) and critical stores (CSs).

Firmware	# Initial CVs	# Extended CVs	# CSs Identified by *InvisiGuard*	# CSs Identified by OAT
Syringe Pump	31	42	55	51
Light Controller	16	22	21	19
Alarm	10	10	11	10
PinLock	13	18	84	42
RF_Door_Lock	10	10	35	14
Steering Control	8	8	15	15
PID controller	4	12	54	23
Avg.	13.14	17.57	39.29	24.86

TABLE III: Statistics on # of all writes, # of allowable writes and percentage of reduced writes that may be maliciously used to tamper the critical variable.

Firmware	# Allowable writes	# All writes	# Reduced writes (%)
Syringe Pump	55	4701	98.83%
Light Controller	21	4659	99.55%
Alarm	11	4600	99.76%
PinLock	84	11,814	99.29%
RF_Door_Lock	35	4682	99.87%
Steering Control	15	4917	99.69%
PID controller	54	11,295	99.52%
Avg.	39.29	6667.14	99.42%

In Table II, we show the number of critical variables that we manually annotated (e.g., authentication hash, PID parameters, etc.), the number of extended critical variables after analysis, along with the store instructions associated with them. As shown, our implementation added a moderate number of additional critical variables (4.43 on average) due to variable expansion. Meanwhile, *InvisiGuard* identifies more critical store instructions than OAT does (39.29 vs. 24.86). Note that *InvisiGuard* needs to recover critical store instructions to construct the allowlist, while OAT needs it to trace the data flow. This suggests that there are deficiencies in OAT's implementation, which we will elaborate on in the analysis of false positives.

False Negatives Our current static analysis is neither context-sensitive nor field-sensitive, which means a write instruction could be associated with multiple write targets, not all of which are legitimate in every context. As a result, attackers can illegally corrupt any critical variable in the allowlist using the same critical write instruction. For example, in the `Pinlock` firmware, the function `mbedtls_sha256` is invoked twice. The two outputs are stored in `key` and `key_in`, which are

supposed to hold the hash values calculated from the expected (and thus correct) `PIN` and the received `PIN`, respectively. Since our analysis cannot differentiate the calling context, *InvisiGuard* has to allow `mbedtls_sha256` to write to both `key` and `key_in` at all time. As another example, in `PID controller`, the `TPID` variable includes 19 different fields. Since our analysis is field-insensitive, we allow any store instructions that can access `TPID` to access any field. In both cases, attackers can bypass *InvisiGuard* because they can write to critical variables that should not be accessible. Fortunately, attackers are still limited to using an existing critical store instruction to exploit this weakness. Table III demonstrates the security performance in analysis considering false negatives. Even in the worst-case scenario, where the attackers can exploit all the allowable critical stores, InvisiGuard can still detect any non-critical stores attempting to write to the critical variables. On average, InvisiGuard reduces the attack surface of critical variables by 99.42%. This indicates that in practice, attackers can hardly launch an effective exploitation against this weakness. Given that both OAT and *InvisiGuard* rely on context-insensitive and field-insensitive analysis, they exhibit similar security performance in terms of design. However, *InvisiGuard* incurs lower overhead compared to OAT (Section VI-B and VI-C).

False Positives A false positive happens when *InvisiGuard* does not construct an allowlist for a critical store instruction or maintains an incomplete allowlist. If this happens, a legal store would falsely trigger a security alert. Based on our empirical study, we never observed any such false alerts. This indicates that our points-to analysis performs well in the evaluated firmware. In contrast, OAT failed to execute six out of the seven samples. As shown in Table II, OAT found less critical store instructions on the same set of critical variables (24.86 vs. 39.29 on average). As a result, it misses instrumentation to correctly track the value flow. The root cause is that OAT does not resolve indirect calls nor trace pointers across functions, while *InvisiGuard* does. We have communicated this issue with OAT authors but it seems that there is no easy fix to their implementation.

Despite the encouraging results, there is no guarantee that our implementation will scale well to unforeseen firmware, due to the fundamental problems with static analysis. For example, our implementation may fail indirect call resolutions if a pointer is a formal argument in a callee function without any caller [74]. To mitigate this problem, the secure runtime should collect as much information about the alert context as possible. Such information can be invaluable for developers to diagnose the problem and manually suppress false alerts.

B. Performance Overhead

MCU firmware runs in an infinite loop, with each loop completing an operation. To evaluate the performance overhead, we therefore measured the time to complete one operation cycle. For example, in the Syringe Pump firmware, we measured the time from receiving user input to finishing the dispensation. To measure the execution time cost in a firmware-independent way, we implemented a high-resolution timer based on DWT. Recall that DWT also supports program profiling. In our implementation, we leveraged `DWT_CYCCNT` that can record the cycle number for a period of execution.

To mitigate performance bias caused by different implementations, the define-site and use-site instrumentation in OAT was corrected by the points-to analysis of *InvisiGuard*. Moreover, we applied the same optimization to OAT so that memory intrinsic functions do not need to trap in the secure world multiple times.

For OAT, we recorded the number of triggered instrumentation at define-sites and use-sites of critical variables. Each instrumentation traps the execution to the secure world. For *InvisiGuard*, we recorded the number of triggered legitimacy checks, including debug monitor exceptions and `memcpy` invocations. Invocations to `trusted_malloc` and `trusted_free` were also recorded since they are implemented in the secure world. We executed each firmware sample ten times, and we took the average over the runs. Table IV shows the results. In calculating the overall overhead, we used geometric means over all the firmware samples to mitigate issues with different baselines [79].

The average runtime overhead of *InvisiGuard* is 2.74%, compared with 6.37% in OAT. *InvisiGuard* costs less than OAT in all samples, reducing runtime overhead by 56.99% on average. However, it is more expensive for *InvisiGuard* to complete a check in the secure world than OAT (7.11 μs in *InvisiGuard* vs. 4.72 μs in OAT on average). In particular, recovering the address of the write target involves complex instruction decoding. We further provide a breakdown analysis of the overhead. The overhead mainly comes from three aspects: 1) recovery of addresses of write instructions and targets from the context of the exception, 2) legitimacy checks against the allowlist, and 3) dynamic memory management in the secure world. Table V shows the percentage of each part. It can be seen that the cost of *legitimacy check* is the largest, followed by *address recovery* and *memory management*. Meanwhile, *legitimacy check* and *address recovery* account for majority of the overhead (91.23%). The address recovery overhead is about 31.35%, which means that with hardware support for recovering addresses, our overhead could be further reduced by ˜30%.

The main reason why *InvisiGuard* outperforms OAT is that OAT triggers significantly more traps to the secure world than *InvisiGuard* did (2.5x - 39.2x, 8.8x on average). This huge difference outweighs the higher overhead of *InvisiGuard* in individual secure-world handling. Indeed, *InvisiGuard* only needs to perform the legitimacy check when necessary. More specifically, the hardware only traps the execution to the secure world when a critical variable is really being written. Furthermore, OAT needs to trap both the define-sites (to record the value) and use-sites (to check the value), while *InvisiGuard* only needs to trap the define-sites on demand. Reading the table, the number of use-sites is much larger than that of def-sites. This means OAT traps in the secure world more frequently. These statistics also agree with the results reported in related work [51], [17].

Finally, to demonstrate *InvisiGuard*'s robustness and scalability, we conducted two experiments with intensive loads –

TABLE IV: Performance overhead.

Firmware	Baseline	OAT			*InvisiGuard*		
	Time (cycles)	# Use	# Def	Overhead	# Legitimacy Check	# trusted_malloc/free	Overhead
Syringe Pump	102,752,850	14,408	377	6.81%	377	258	0.17%
Light Controller	220,904,123	6,900	1,130	1.74%	1,130	150	0.52%
Alarm	5,441,524	574	362	8.32%	362	8	3.76%
PinLock	23,675,299	3,588	1,597	9.94%	1,597	56	5.86%
RF Door Lock	11,748,938	407	226	2.64%	226	4	1.27%
Steering Control	13,866,606	77	24	0.28%	24	6	0.21%
PID Controller	312,992	52	35	15.70%	35	14	7.64%
Geometric Mean	-	-	-	6.37%	-	-	2.74%

TABLE V: Overhead breakdown of *InvisiGuard* (numbers are percentage).

Firmware	Legitimacy Check	Address Recovery	Memory Management
Syringe Pump	63.54	26.82	9.64
Light Controller	62.24	26.27	11.49
Alarm	53.91	40.79	5.30
PinLock	61.96	31.64	6.40
RF Door Lock	50.69	44.39	4.92
Steering Control	60.72	21.10	18.18
PID Controller	59.16	30.74	10.37
Geometric Mean	59.88	31.35	8.77

monitoring **ALL** the variables in the PID algorithm from the PID controller firmware and **ALL** the variables in the SHA256 algorithm from the PinLock firmware. The PID algorithm involves 19 variables and 69 critical stores, while the SHA256 algorithm involves 19 variables and 206 critical stores. With manual checks, we confirm that all the write operations to the annotated variables were precisely caught and checked. Compared to OAT, the performance overhead was 40.52% vs. 172.67% in the SHA256 algorithm and 11.19% vs. 17.25% in the PID algorithm. We can see that *InvisiGuard* still performs better in both tested samples, and this also demonstrates the robustness of the implementation of *InvisiGuard*.

C. Memory Overhead

We evaluated memory overhead in terms of RAM consumption and flash consumption. The results are shown in Table VI.

RAM Overhead For both solutions, the RAM overhead mainly comes from the runtime data maintained in the secure world. Notably, *InvisiGuard* compiles the allowable write target table into the secure runtime as a hashmap. Moreover, the variable bounds table is also dynamically maintained. In contrast, OAT needs to store the recorded values for each occurrence of all critical variables. As shown in the table, the RAM overhead of *InvisiGuard* is 27.48% on average, compared to 121.41% in OAT. The key factor leading to the difference is that the hashmap in *InvisiGuard* is indexed by variable IDs, whereas the hashmap is indexed by the variable address in OAT. A variable ID can correspond to multiple instances at runtime. In addition, we note that while the ratios appear to be high due to the small size of the firmware, the absolute overhead is not (4.68KB for OAT vs. 1.11KB for *InvisiGuard*).

Flash Overhead The flash memory is mainly used to store code and read-only global data. We separate the secure runtime from the target firmware in evaluating flash overhead. For the target firmware itself (non-secure part), both *InvisiGuard* and OAT exhibit negligible overhead, although *InvisiGuard* consumes slightly less flash memory than OAT. *InvisiGuard* introduces additional code mainly due to the inserted `NOP`s and store instruction restriction. OAT introduces additional code due to instrumentation on define-, and use-sites. Regarding the secure runtime implementation, *InvisiGuard* uses more space. The reason is that it needs to duplicate a heap manager to the secure world for relocating critical variables. For both systems, the overhead on the secure flash is constant. Overall, the average flash overhead is 15.56% for *InvisiGuard*, and 12.55% for OAT.

D. Power Consumption

Low power consumption is a fundamental requirement for low-end embedded devices, which are often powered by limited energy sources, such as batteries [77]. Power consumption would be influenced by various factors, including CPU clock frequency, workload, and connected peripheral devices, among others [63]. Before measurement, we ensure that other factors remain unchanged, such as keeping the same default CPU clock frequency of 110 MHz. The only variable is the running program. Our development board, the STM32L562E-DK, embeds energy meter tools. Notably, it incorporates an "on-board energy meter" interface, which facilitates convenient power consumption measurement through a dedicated USB interface (see Figure 3). The results are presented in Figure 4, demonstrating that both OAT and InvisiGuard introduce negligible overhead in terms of power consumption, with an average of 1.35% for OAT and 1.12% for *InvisiGuard*.

E. Attack Detection

By enforcing integrity constraints on critical variables, *InvisiGuard* greatly raises the bar for data attacks against MCU firmware. In particular, attackers cannot take advantage of a large number of non-critical writes directly. To validate this,

TABLE VI: Memory overhead.

Firmware	RAM Consumption (kB)			Flash Consumption (kB)						
	Baseline	OAT (%)	*InvisiGuard* (%)	Baseline	OAT -secure	OAT -nonsecure	Overhead	*InvisiGuard* -secure	*InvisiGuard* -nonsecure	Overhead
Syringe Pump	4.19	116.47%	45.35%	83.78	12.52	85.95	17.53%	14.92	84.58	18.76%
Light Controller	3.77	147.48%	25.99%	79.99	12.52	81.14	17.08%	14.92	81.09	20.02%
Alarm	4.09	96.09%	5.87%	79.27	12.52	79.56	16.17%	14.92	79.66	19.31%
PinLock	3.96	128.03%	47.47%	242.55	12.52	244.41	5.93%	14.92	243.92	6.72%
RF Door Lock	3.46	132.08%	16.18%	78.89	12.52	79.57	16.73%	14.92	79.28	19.42%
Steering Control	3.77	116.89%	16.88%	86.99	12.52	87.68	15.18%	14.92	87.19	17.38%
PID Controller	3.86	116.32%	40.93%	189.22	12.52	191.20	7.67%	14.92	189.81	8.20%
Geometric Mean	-	121.41%	27.48%	-	-	-	12.55%	-	-	15.56%

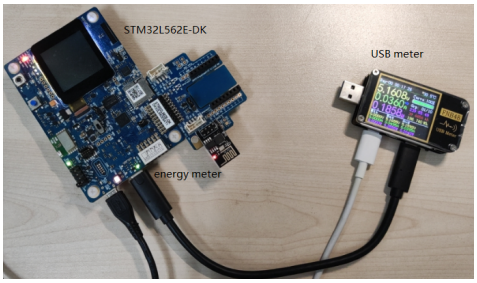


Fig. 3: Power Consumption Measurement Setup.

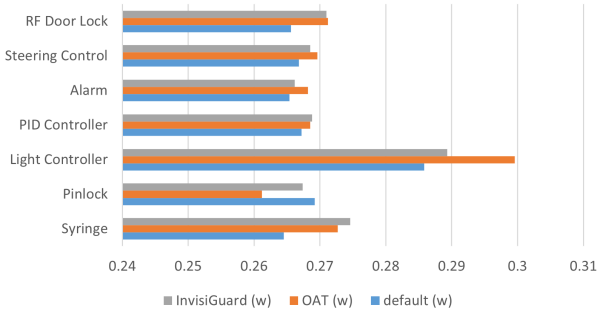


Fig. 4: Power Consumption.

we conducted an attack detection experiment. Specifically, we used `PinLock` as the attack target to demonstrate how *InvisiGuard* can defeat exploitation attempts. The attack goal is to bypass the authentication process to unlock the door without credentials, or to disable *InvisiGuard* protection.

We intentionally introduced three vulnerabilities and developed corresponding exploits. They allow attackers to 1) overwrite a stack variable named `lock_status`, which is used in the condition check before executing the `unlock()` function, 2) overwrite a global buffer named `key`, which holds the hash value to check user PINs, and 3) write to the DWT control register to disable DWT. These exploits were successful on the unprotected firmware. After deploying *InvisiGuard*, the first two exploits immediately triggered an exception and the secure runtime raised an alert. Since *InvisiGuard* uses TrustZone-M to present the normal world from manipulating DWT registers (i.e., memory range `0xE0001020-0xE0001050`), the third exploit triggered a security exception and our runtime also raised an alarm. As a result, we successfully defeated all these attacks.

F. Security Analysis

Although we have successfully defeated attacks in section VI-E, this does not mean that our solution is perfect. We further examine the potential attack surface of *InvisiGuard*.

Our threat model anticipates that attackers may find and exploit unknown vulnerabilities in the programs running in the normal world. However, we assume that control flow hijacking or modification cannot occur, as significant orthogonal work on CFI for MCU devices exists [85], [7], [59], [43], and we assume that such protection is already in place. Additionally, we assume that the trusted software within the TEE is bug-free. To maliciously tamper with critical variables (CVs), attackers would need to achieve any of the following: ① write CVs using non-critical stores, ② write CVs using critical stores, or ③ disable the protection mechanism of *InvisiGuard*.

① is eliminated. Since *InvisiGuard* is effectively deployed, then any non-critical store on a CV will be easily detected, because there is no non-critical store in the allowable write target table. For ②, *InvisiGuard* raises the bar for successful exploitation, but there remains some attack surface. On the one hand, each critical store is limited in using its allowlists, so it can not write other CVs, and thus we improve the security of these CVs. On the other hand, our current static analysis is neither context-sensitive nor field-sensitive. Therefore, a critical store could be maliciously leveraged by abusing the execution context or structure field in some circumstances. However, as evaluated at Section VI-A, even in the worst-case scenario, *InvisiGuard* can reduce the attack surface of critical variables by 99.42%. Therefore, it is very hard for attackers to launch an effective exploitation against this weakness.

③ is also ruled out. To achieve ③, attackers would need to: i) Disable the Data Watchpoint and Trace (DWT) mechanism or ii) Tamper with the metadata in the secure world. Considering that the configuration of DWT is fully protected by the TEE, option i) can be dismissed. Regarding option ii), attackers would need to access metadata from the normal world and they would: 1) Modify the metadata directly. However, since the metadata is protected by the TEE, attackers cannot perform this from the normal world. 2) Exploit debug monitor interruptions. The only input used by the handler is stacked exception context, which is saved by the hardware immediately after writing

a CV, and thus can prevent modification from any attackers. Although there may be a delay of up to 2 instructions, as discussed in section V-D, this delay is extremely challenging to exploit. 3) Abuse the interfaces exposed to the normal world, such as secure API calls (e.g., trusted_malloc/trusted_free and trusted_memcpy-style libraries). However, it is important to note that these interfaces are hardcoded in the program, and the attacker cannot access them through indirect calls, jumps, or returns due to the existing CFI enforcement deployed. Furthermore, these interfaces are free of bugs and include additional security checks. For example, the trusted_memcpy function verifies the memory boundaries before performing a copy. Therefore, this scenario is also improbable. As a result, attackers can not achieve ③.

VII. Discussion

Remaining Attack Surface Apart from the false negatives discussed in Section VI-A, our current prototype does not propagate implicit critical variables, which for example can indirectly influence the value of an annotated variable by changing the execution path. While deciding which variables are critical is orthogonal to our work, it can be addressed by using more advanced static analysis methods. Meanwhile, *InvisiGuard* does not prevent attackers from reading credential information from the system. Solutions like full data flow integrity [5] and address sanitizers [51], [67] can potentially provide more comprehensive data protection but imposes prohibitive overhead. We leave this for future work.

Protecting an Excessive Number of Variables Protecting an excessive number of variables may result in a greater overhead. Nevertheless, we found no evidence of this issue in any of our benchmarks. Furthermore, our experiments demonstrated that even when all variables are protected, in the worst possible case, the resulting overhead is not extreme (Section VI-B). Critical variables are typically associated with very specific features, such as authentication or authorization, making it unlikely for a large number of genuine dependencies to exist. Consequently, in the unlikely case – this issue were to occur, it can likely be solved by improving the static analysis techniques.

Evaluating CFI Protection Since CFI is not the focus of this paper and its overhead only relates to instrumenting checks at indirect jump sites (i.e., forward indirect jump instructions and backward indirect jump instructions), it does not influence overhead introduced by the CVI solution and *InvisiGuard*, which only perform checks at specific store or read instructions. Therefore, our evaluation does not measure the overhead brought by integrating existing CFI solutions, which is enough and fair to compare the overhead brought by CVI solution and *InvisiGuard*.

Peripheral Protection Our prototype currently only protects critical variables in memory, not memory-mapped peripheral registers, which can also be critical. However, developers can easily use the provided annotation interface to mark these addresses. Then, *InvisiGuard* can use additional DWT registers to monitor the corresponding MMIO addresses. Unfortunately, unlike memory corruption, the damage caused by an illegal MMIO write is immediate and unrecoverable. Nevertheless, *InvisiGuard* can generate an alert if this happens. It is worth noting that the value-based define-use check proposed in OAT fundamentally cannot support this feature since MMIO registers are volatile and thus no data-flow can be tracked.

Real-Time Constraints Some MCU-based devices need to react to the environment under certain time constraints, which may be broken by the overhead imposed by *InvisiGuard*. However, as shown in our evaluation, such overhead is tiny and predictable. As such, it can be accounted for before deployment when real-time constraints have to be considered.

Performance Overhead *InvisiGuard* traps every writes to critical variables. When a critical variable needs to be written excessively (e.g., in a loop), the overhead grows. A straightforward solution is to complete the legitimacy check for a piece of code all at once, instead of checking individual write instructions, as we did for the `memcpy` function. This can improve performance but requires case-by-case manual effort or sophisticated static analysis to assist the process.

Production Consideration Currently, the meta-data is hard-coded and should be updated with each firmware update. However, we will design secure APIs so that the metadata can be updated without updating the secure code. In this design, we can wrap metadata with a cryptographic signature, and then transmit the signed metadata to the secure world via a secure API. In the secure world, the metadata signature first needs to be cryptographically verified. If the signature is valid, the metadata can then be used to update the allowlist. Otherwise, the metadata should be rejected. This helps ensure that only authenticated and authorized metadata can make security state changes.

Leveraging Hardware Features The applicability of our approach may be limited by the reliance on specific hardware features – **DWT and TEE**. Specifically, i) the DWTs are primarily designed for debugging purposes and are often limited in number within microcontrollers. If a microcontroller does not support this feature or undergoes future hardware updates that render it incompatible with *InvisiGuard*, the system may become unusable. However, *InvisiGuard* only monitors a single range of memory and therefore requires only a single pair of DWTs. By automatically remapping critical variables to this monitored memory, *InvisiGuard* enables simultaneous monitoring of these variables. The watchpoint-based monitoring mechanism, is the only convenient way to debug applications where hardware, such as a motor, must remain operational [66]. This feature is generally supported by current mainstream MCUs, including ARM [11], RISC-V [71], and MIPS [47]. Consequently, it is reasonable to expect that future hardware updates will continue to support this feature. Therefore, our approach holds potential benefits across different platforms and remains compatible with future hardware updates. Additionally, ii) our approach utilizes TEE as a *trust anchor* (Root of Trust) to ensure the integrity and

confidentiality of important metadata. Although not every embedded system can support a TEE, there are numerous existing works, such as APEX [55], LIRA-V [69], GAROTA [6], and ARM TrustZone [12] that offer similar trust anchors on low-end embedded system. This sheds light on the feasibility and applicability of our design in the future.

Secure Runtime Protection We assume that the secure runtime, including the metadata maintained by it, is not vulnerable to attacks. Although there have been real-world attacks to the TrustZone on smartphones, this assumption is well-accepted in academia [76], [64], [14], [33]. In practice, we can either adopt a safer programming language such as RUST to implement the secure-world software, or formally verify the interface between the normal world and the secure world.

VIII. Related Work

Data Protection DFI [18] and WIT [5] rely on static analysis to extract a data flow graph. Then, instrumentation makes sure that the memory access instructions follow the data flow. Sanitizers [51], [67], [17], [53] track and check bounds information of data objects at runtime. However, their overhead is substantial (e.g., 44% to 103% in DFI, 73% slowdown in AddressSanitizer [67]) and they are rarely used in production environments except for in-house fuzzing. Furthermore, there are numerous studies aiming at providing selective protection [61], [17], [60], [76]. However, it is hard for them to prevent widespread instrumentation and fully gain performance benefits. *InvisiGuard* leverages a commonly available hardware feature in a novel way by "freely" intercepting interesting memory accesses, and significantly reduces overhead.

Security Solutions for MCU Devices The increasing prevalence of MCU devices also stimulates attackers' interests. As a result, many lightweight designs have been proposed to protect MCU devices from exploitation. There is significant research (e.g., APEX [55], LIRA-V [69], GAROTA [6], RealSWATT [77]) proposing the construction of a *trust anchor* suitable for low-end MCUs. The *trust anchors*—also commonly referred to as Roots of Trust (RoTs)— are categorized as software-based, hardware-based, or hybrid (i.e., based on hardware/software co-designs) [56]. Among them, APEX [55] and LIRA-V [69] aim to guarantee the integrity of recorded traces to prevent tampering, which is crucial for remote attestation. GAROTA [6] goes even further by ensuring the availability of real-time critical tasks, allowing desired actions to be performed even if the software is fully compromised. Similarly, RT-TEE [80] provides real-time system availability guarantees for critical tasks, ensuring they are executed in real-time as expected before their deadlines. However, RT-TEE [80] achieves this by leveraging existing TrustZone hardware support (ARMv8-A and ARMv8-M) instead of building a trust anchor from scratch. While these approaches protect MCUs, they do not specifically address protection against certain attacks, such as control flow hijacking or data-only attacks.

Based on a trust anchor, remote attestation has emerged as a prominent research direction. It involves transmitting encoded execution traces of firmware to a remote server, which then verifies the accuracy of the execution. This includes control-flow attestation (e.g., [4], [58], [54]), data-flow attestation (e.g., [57]), and critical task/mission attestation (e.g., [81]), etc. However, these forms of attestation are essentially offline detection solutions, meaning that by the time an attack is detected, damage may have already occurred to some extent. Recently, Jakkamsetti [37] highlighted this limitation and suggested a future research direction of low-overhead online detection and prevention. *InvisiGuard* represents an effort in this direction concerning data-flow integrity. As for online protection, there has been more research on control flow integrity then on data flow integrity for MCU devices, as also pointed out in a recent survey by Mishra et al. [49]. CFI CaRE [59] and TZmCFI [43] enforce CFI by leveraging TrustZone to implement a shadow stack mechanism. μRAI [7] enforces backward CFI (return address integrity) by removing the need to spill return addresses to the stack. Silhouette [85] and Kage [26] introduce an incorruptible shadow stack using store hardening. OAT [76] proposes CVI to enforce data-flow integrity online. In contrast, *InvisiGuard* significantly reduces this overhead with a novel design, combining TrustZone and the data watchpoint mechanism.

Code diversity solutions [70], [22] mitigate attacks by bringing non-determinism to the system. uXOM [45] implements a software-based eXecution-Only Memory (XOM) for ARM MCUs. ACES [21] and OPEC [86] support arbitrary compartmentalization policies for bare-metal systems. These solutions re-enable many mechanisms that were only available on traditional platforms for MCU devices. *InvisiGuard* improves state-of-the-art online data integrity solutions of MCU devices by leveraging hardware features.

Hardware-Assisted Security DWT has been investigated before, but not for data integrity. PicoXOM [68] implements XOM by leveraging DWT to prevent illegal reads from code regions. While not directly designed for MCU devices, Jinsoo et al. [38] propose using DWT to implement an efficient in-process memory isolation mechanism and to harden the OS kernel for mobile devices [39]. We are the first to use DWT to enforce data integrity for MCU devices. TrustZone has also been extensively used in recent work [59], [43], [70], [76], [80]. We combine the two features to build a lightweight data integrity solution for MCU devices.

IX. Conclusions

This work proposes a hardware-assisted approach for data integrity. The key innovation lies in the fact that our approach avoids expensive instrumentation-based data flow tracking with the help of hardware, which passively monitors access to critical variables. As a result, 1) security checking is triggered only when necessary, and 2) all recognized critical variables are comprehensively monitored. Given that many cyber-physical systems and industrial control systems are powered by resource-constrained devices, it has become essential to protect critical variables on these systems with sufficient efficiency. Our prototype, named InvisiGuard, runs on ARM Cortex-M based MCUs with TrustZone and DWT support. Our evaluation

results show that InvisiGuard can successfully detect and thwart data-only attacks with low overhead.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their valuable feedback. This work is supported by Beijing Tianjin Hebei basic research cooperation special project of Natural Science Foundation of China (Grant No.V1640354653903), Beijing Municipal Natural Science Foundation (Grant No.L234033), and the US National Science Foundation under grant CNS-2238264.

REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[2] A. Abbasi and M. Hashemi, "Ghost in the plc designing an undetectable programmable logic controller rootkit via pin control attack," *Black Hat Europe*, vol. 2016, pp. 1–35, 2016.

[3] A. Abbasi, J. Wetzels, T. Holz, and S. Etalle, "Challenges in Designing Exploit Mitigations for Deeply Embedded Systems," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 31–46.

[4] 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 *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 743–754.

[5] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *2008 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2008, pp. 263–277.

[6] E. Aliaj, I. D. O. Nunes, and G. Tsudik, "GAROTA: Generalized Active Root-Of-Trust Architecture (for Tiny Embedded Devices)," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, 2022, pp. 2243–2260.

[7] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "μRAI: Securing Embedded Systems with Return Address Integrity," in *Network and Distributed Systems Security Symposium (NDSS)*, San Diego, CA, 2020.

[8] Amazon Web Services, Inc., "FreeRTOS Memory Management," https://www.freertos.org/a00111.html#heap_2, 2022, (Retrieved: 04/09/2023).

[9] ARM Ltd., "ARM Cortex-M Programming Guide to Memory Barrier Instructions," https://documentation-service.arm.com/static/5efefb97dbdee951c1cd5aaf, 09 2012.

[10] ——, "Armv8-M Architecture Reference Manual," https://developer.arm.com/documentation/ddi0553/latest, 2022, (Retrieved: 04/09/2023).

[11] ——, "Data Watch Trace," https://developer.arm.com/documentation/ddi0439/b/Data-Watchpoint-and-Trace-Unit/DWT-functional-description?lang=en, 2022, (Retrieved: 04/09/2023).

[12] ——, "Introduction to the armv8-m architecture," https://documentation-service.arm.com/static/5f86dc74f86e16515cdb6be6?token=, 2022, (Retrieved: 04/09/2023).

[13] A. Ayub, H. Yoo, and I. Ahmed, "Empirical study of PLC authentication protocols in industrial control systems," in *2021 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2021, pp. 383–397.

[14] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 90–102.

[15] G. Beniamini, "Over The Air: Exploiting Broadcom's Wi-Fi Stack," Google Project Zero Blog, April 2017.

[16] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 161–176.

[17] S. A. Carr and M. Payer, "Datashield: Configurable data confidentiality and integrity," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017, pp. 193–204.

[18] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 147–160.

[19] S. Chen, J. Xu, and E. C. Sezer, "Non-Control-Data Attacks Are Realistic Threats," in *14th USENIX Security Symposium (USENIX Security 05)*. Baltimore, MD: USENIX Association, 2005.

[20] A. Cherepanov, "WIN32/INDUSTROYER: A new threat for industrial control systems," *White paper, ESET*, June 2017.

[21] A. A. Clements, N. S. Almakhdhub, S. Bagchi, and M. Payer, "ACES: Automatic compartments for embedded systems," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 65–82.

[22] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer, "Protecting bare-metal embedded systems with privilege overlays," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 289–303.

[23] R. R. C. DDrazir, "Alarm system for home based on Raspberry PI 3," https://github.com/ddrazir/alarm4pi, 2016, (Retrieved: 04/09/2023).

[24] M. Derhambakhsh, "PID Controller library for ARM Cortex M (STM32)," https://github.com/Majid-Derhambakhsh/PID-Library, 2022.

[25] Drifton A/S, Inc, "Digital Laboratory Syringe Pump dLSP500," https://www.drifton.eu/shop/24-syringe-pumps/2460-digital-laboratory-syringe-pump-dlsp500/, 2023.

[26] Y. Du, Z. Shen, K. Dharsee, J. Zhou, R. J. Walls, and J. Criswell, "Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, 2022, pp. 2281–2298.

[27] L. Feng, J. Huang, J. Huang, and J. Hu, "Toward taming the overhead monster for data-flow integrity," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 27, no. 3, pp. 1–24, 2021.

[28] M. E. Garbelini, C. Wang, S. Chattopadhyay, S. Sumei, and E. Kurniawan, "SweynTooth: Unleashing Mayhem over Bluetooth Low Energy," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 911–925.

[29] L. Garcia, F. Brasser, M. H. Cintuglu, A.-R. Sadeghi, O. A. Mohammed, and S. A. Zonouz, "Hey, My Malware Knows Physics! Attacking PLCs with Physical Model Aware Rootkit." in *Network and Distributed Systems Security Symposium (NDSS)*, San Diego, CA, 2017.

[30] X. Ge, W. Cui, and T. Jaeger, "Griffin: Guarding control flows using intel processor trace," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 585–598, 2017.

[31] A. Ginter, "The top 20 cyber attacks against industrial control systems," *White Paper, Waterfall Security Solutions*, 2017.

[32] B. Green, R. Derbyshire, M. Krotofil, W. Knowles, D. Prince, and N. Suri, "PCaaD: Towards automated determination and exploitation of industrial systems," *Computers & Security*, vol. 110, p. 102424, 2021.

[33] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, "Trustshadow: Secure execution of unmodified applications with arm trustzone," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, 2017, pp. 488–501.

[34] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel *et al.*, "Toward the analysis of embedded firmware through automated re-hosting," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, 2019, pp. 135–150.

[35] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic Generation of Data-Oriented Exploits," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 177–192.

[36] 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 *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 969–986.

[37] S. Jakkamsetti, "Root-of-trust architectures for low-end embedded systems," Ph.D. dissertation, University of California, Irvine, 2023.

[38] J. Jang and B. B. Kang, "In-process Memory Isolation Using Hardware Watchpoint," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[39] ——, "Revisiting the arm debug facility for os kernel security," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[40] J. Judin, "Light controller for controlling remote controllable switches," https://github.com/Barro/light-controller, 2016, (Retrieved: 04/09/2023).

[41] S. Kalle, N. Ameen, H. Yoo, and I. Ahmed, "CLIK on PLCs! attacking control logic with decompilation and virtual PLC," in *Binary Analysis Research (BAR) Workshop, Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2019.

[42] O. Karliner, "FreeRTOS TCP/IP Stack Vulnerabilities – The Details," https://blog.zimperium.com/freertos-tcpip-stack-vulnerabilities-details/, December 2018.

[43] T. Kawada, S. Honda, Y. Matsubara, and H. Takada, "TZmCFI: RTOS-Aware Control-Flow Integrity Using TrustZone for Armv8-M," *International Journal of Parallel Programming*, pp. 1–21, 2020.

[44] M. Kol and S. Oberman, "19 Zero-Day Vulnerabilities Amplified by the Supply Chain," *JSOF, White Paper*, 2020.

[45] D. Kwon, J. Shin, G. Kim, B. Lee, Y. Cho, and Y. Paek, "uXOM: Efficient eXecute-Only Memory on ARM Cortex-M," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, 2019, pp. 231–247.

[46] W. Landi, "Undecidability of static analysis," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 4, pp. 323–337, dec 1992.

[47] Lauterbach GmbH, "MIPS Debugger and Trace," https://www2.lauterbach.com/pdf/debugger_mips.pdf, 2022, (Retrieved: 04/09/2023).

[48] J. Mayer, "An arduino firmware that outputs a PWM signal for steering motor control via serial," https://github.com/jabelone/car-controller, 2016, (Retrieved: 04/09/2023).

[49] T. Mishra, T. Chantem, and R. Gerdes, "Survey of Control-Flow Integrity Techniques for Embedded and Real-Time Embedded Systems," *arXiv preprint arXiv:2111.11390*, 2021.

[50] E. Montalbano, "Hacker Tries to Poison Water Supply of Florida Town," https://threatpost.com/hacker-tries-to-poison-water-supply-of-florida-town/163761/, 2021, (Retrieved: 04/09/2023).

[51] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for C," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 245–258.

[52] ——, "CETS: compiler enforced temporal safety for C," in *Proceedings of the 2010 international symposium on Memory management*, 2010, pp. 31–40.

[53] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 3, pp. 477–526, 2005.

[54] A. J. Neto and I. D. O. Nunes, "Isc-flat: On the conflict between control flow attestation and real-time operations," in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2023, pp. 133–146.

[55] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "APEX: A verified architecture for proofs of execution on remote devices under full software compromise," in *29th USENIX Security Symposium (USENIX Security 20)*. Boston, MA: USENIX Association, 2020, pp. 771–788.

[56] I. D. O. Nunes, S. Hwang, S. Jakkamsetti, and G. Tsudik, "Privacy-from-birth: Protecting sensed data from malicious sensors with versa," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 2413–2429.

[57] I. D. O. Nunes, S. Jakkamsetti, and G. Tsudik, "Dialed: Data integrity attestation for low-end embedded devices," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 313–318.

[58] ——, "Tiny-cfa: Minimalistic control-flow attestation using verified proofs of execution," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 641–646.

[59] T. Nyman, J.-E. Ekberg, L. Davi, and N. Asokan, "CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2017, pp. 259–284.

[60] T. Palit, J. F. Moon, F. Monrose, and M. Polychronakis, "Dynpta: Combining static and dynamic analysis for practical selective data protection," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1919–1937.

[61] K. Pattabiraman, V. Grover, and B. G. Zorn, "Samurai: protecting critical data in unsafe languages," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 4, pp. 219–232, 2008.

[62] J. Pearce, "Open-source syringe pump," https://www.appropedia.org/Open-source_syringe_pump, 2016, (Retrieved: 04/09/2023).

[63] M. Pedram, "Power optimization and management in embedded systems," in *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, 2001, pp. 239–244.

[64] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, pp. 1–36, 2019.

[65] Ryan, "Industrial Control Systems Killed Once and Will Again, Experts Warn," https://www.wired.com/2008/04/industrial-cont/, 2008, (Retrieved: 04/09/2023).

[66] Segger Ltd., "Monitor mode debugging," https://www.segger.com/products/debug-probes/j-link/technology/monitor-mode-debugging/, 2023, (Retrieved: 04/09/2023).

[67] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-Sanitizer: A Fast Address Sanity Checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 309–318.

[68] Z. Shen, K. Dharsee, and J. Criswell, "Fast Execute-Only Memory for Embedded Systems," in *2020 IEEE Secure Development (SecDev)*. IEEE, 2020, pp. 7–14.

[69] C. Shepherd, K. Markantonakis, and G.-A. Jaloyan, "LIRA-V: lightweight remote attestation for constrained RISC-V devices," in *2021 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2021, pp. 221–227.

[70] J. Shi, L. Guan, W. Li, D. Zhang, P. Chen, and N. Zhang, "HARM: Hardware-Assisted Continuous Re-randomization for Microcontrollers," in *2022 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2022, pp. 520–536.

[71] SiFiv, Inc, "RISC-V Debug Specification," https://github.com/riscv/riscv-debug-spec/blob/master/riscv-debug-stable.pdf, 2022, (Retrieved: 04/09/2023).

[72] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity." in *Network and Distributed Systems Security Symposium (NDSS)*, San Diego, CA, 2016.

[73] K. Stouffer, J. Falco, K. Scarfone *et al.*, "Guide to industrial control systems (ICS) security," *NIST special publication*, vol. 800, no. 82, pp. 16–16, 2011.

[74] Y. Sui, "Question regarding the callgraph," https://github.com/SVF-tools/SVF/issues/732, 2022, (Retrieved: 04/09/2023).

[75] Y. Sui and J. Xue, "SVF: interprocedural static value-flow analysis in LLVM," in *Proceedings of the 25th international conference on compiler construction*. ACM, 2016, pp. 265–266.

[76] Z. Sun, B. Feng, L. Lu, and S. Jha, "OAT: Attesting Operation Integrity of Embedded Devices," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1433–1449.

[77] S. Surminski, C. Niesler, F. Brasser, L. Davi, and A.-R. Sadeghi, "RealSWATT: Remote Software-based Attestation for Embedded Devices under Realtime Constraints," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2890–2905.

[78] M. Team, "BadAlloc – Memory allocation vulnerabilities could affect wide range of IoT and OT devices in industrial, medical, and enterprise networks," https://msrc-blog.microsoft.com/2021/04/29/badalloc-memory-allocation-vulnerabilities-could-affect-wide-range-of-iot-and-ot-devices-in-industrial-medical-and-enterprise-networks/, April 2021.

[79] E. van der Kouwe, G. Heiser, D. Andriesse, H. Bos, and C. Giuffrida, "SoK: Benchmarking flaws in systems security," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 310–325.

[80] J. Wang, A. Li, H. Li, C. Lu, and N. Zhang, "RT-TEE: Real-time System Availability for Cyber-physical Systems using ARM TrustZone," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 352–369.

[81] J. Wang, Y. Wang, A. Li, Y. Xiao, R. Zhang, W. Lou, Y. T. Hou, and N. Zhang, "ARI: Attestation of Real-time Mission Execution Integrity," in *32nd USENIX Security Symposium (USENIX Security 23)*. IEEE, 2023, pp. 2761–2778.

[82] Z. Wang, H. Wang, H. Hu, and P. Liu, "Identifying Non-Control Security-Critical Data in Program Binaries with a Deep Neural Model," *arXiv preprint arXiv:2108.12071*, 2021.

[83] H. Wen, Z. Lin, and Y. Zhang, "Firmxray: Detecting Bluetooth link layer vulnerabilities from bare-metal firmware," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 167–180.

[84] C. Yadin, "There's More to Vulnerability Management than CVSS score," https://devicetotal.com/elementor-1180/, 2022, (Retrieved: 04/09/2023).

[85] J. Zhou, Y. Du, Z. Shen, L. Ma, J. Criswell, and R. J. Walls, "Silhouette: Efficient protected shadow stacks for embedded systems," in *29th USENIX Security Symposium (USENIX Security 20)*. Boston, MA: USENIX Association, 2020, pp. 1219–1236.

[86] X. Zhou, J. Li, W. Zhang, Y. Zhou, W. Shen, and K. Ren, "OPEC: operation-based security isolation for bare-metal embedded systems," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 317–333.

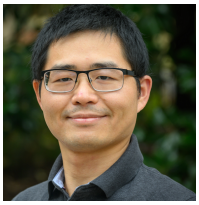
Dongliang Fang received the B.E. degree from WuHan University, China, in 2017. He is currently pursuing the Ph.D. degree with the Institute of Information Engineering, Chinese Academy of Sciences, China. His research interests include the security of the Internet of Things and Industrial Control Systems.

Anni Peng received the B.E. degree from Huazhong University of Science and Technology, China, in 2017. She is currently pursuing the Ph.D. degree with the National Computer Network Intrusion Protection Center, Chinese Academy of Sciences, China. Her research interest is network and system security.

Le Guan is an Assistant Professor of Cybersecurity in the Department of Computer Science at the University of Georgia, and a member of the UGA Institute for Cybersecurity and Privacy. His research interests cover a wide range of systems security, including mobile security and IoT systems security.

Erik van der Kouwe is an Assistant Professor co-leading the VUSec systems security group at the Vrije Universiteit Amsterdam. He received his PhD in software reliability from prof. Andy Tanenbaum, and afterwards broadened hist scope to include systems security. His recent work is on practical compiler-assisted defenses against zero-day vulnerabilities and on properly benchmarking such defenses.

Klaus von Gleissenthall is an Assistant Professor in Computer Science at the Vrije Universiteit Amsterdam, affiliated with the theory group and VUSec. He works on methods that help practitioners write correct, secure and reliable systems, where he focuses on keeping user effort low. His research interests span programming languages, security and systems.



Wenwen Wang is an Assistant Professor in the Department of Computer Science at the University of Georgia. His research interests have spanned a wide spectrum of criticaissues in computer systems that cut across computer architectures, operating systems,compilers, runtimes, and security.

Yuqing Zhang is a Professor of Computer Sciences and the Director of the National Computer Network Intrusion Protection Center at University of CAS. His research interests include network and system security, cryptography, and networking.

Limin Sun is a Professor with the Institute of Information Engineering, Chinese Academy of Sciences, China. He is also the Secretary General of the Select Committee of CWSN and the Director of the Beijing Key Laboratory of IoT Information Security Technology. His main research interests include the Internet-of-Things (IoT) security and Industrial Control System security. He is an Editor of the Journal of Computer Science and Journal of Computer Applications.