ControlFreak: Signature Chaining to Counter Control Flow Attacks

Sergei Arnautov and Christof Fetzer
Technische Universität Dresden
Dresden, Germany
Email: {firstname.lastname}@tu-dresden.de

***Abstract*—Many modern embedded systems use networks to communicate. This increases the attack surface: the adversary does not need to have physical access to the system and can launch remote attacks. By exploiting software bugs, the attacker might be able to change the behavior of a program. Security violations in safety-critical systems are particularly dangerous since they might lead to catastrophic results. Hence, safety-critical software requires additional protection.**

We present an approach to detect and prevent control flow attacks. Such attacks maliciously modify program's control flow to achieve the desired behavior. We develop ControlFreak, a hardware watchdog to monitor program execution and to prevent illegal control flow transitions. The watchdog employs chained signatures to detect any modification of the instruction stream and any illegal jump in the program even if signatures are maliciously modified.

I. Introduction

Modern embedded systems do not operate in isolation, but become increasingly connected and use networks to exchange data, receive software updates, and communicate with other devices to provide a better user experience. One example is the automotive domain. Modern cars contain up to 70 electronic devices (Electronic Control Units, ECU) communicating over a network in a distributed fashion [12]. While bringing a number of benefits, the network connectivity increases the attack surface raising security requirements, since the system becomes exposed to remote attacks. Malicious input to software received over the network may affect program behavior and disrupt correct functioning of the system.

Increasing software complexity is another threat to security. The software stack of embedded systems grows as they perform a richer set of tasks. The estimated size of the binary code running in luxury cars already reached 100 MB [15]; it is predicted that software components in future cars will contain 100 million lines of code [5]. Such complexity leads to an increased number of bugs that can be exploited by attackers in order to change the behavior of programs.

In safety-critical systems, security violations might affect safety and lead to heavy consequences, putting human lives in danger and causing substantial financial losses. Even though safety-critical code is subject to a strict certification process and is usually developed according to domain-specific standards to minimize the number of bugs, it is still difficult to prove that the implementation is bug-free [23]. Checkoway *et al.* [7] demonstrated multiple remote attacks on a real-world car using bugs in its firmware. The authors exploited vulnerabilities to inject malicious code and gained full access to all car systems, including the ones that control the brakes and the engine. Furthermore, even bug-free software needs protection given the trend for *mixed-criticality* systems. In such systems, safety-critical components run on the same hardware or are connected to the same network as non-critical components in order to meet non-functional requirements, such as cost, space or weight. Hence, ensuring that only safety-critical parts are protected does not suffice, since non-critical software may affect safety by interfering with critical code [22].

In this work, we focus on a ubiquitous attack vector: attacks that exploit bugs in software aiming to hijack the control and execute malicious code. Many solutions were proposed in the past to mitigate such intrusions. Some of them (e.g., stack canaries, non-executable stack, address space layout randomization) became widely adopted on personal computers and are integrated in hardware, operating systems or compilers. However, their acceptance in embedded systems is going at a slower pace [7].

For a control flow attack to succeed, the attacker needs to diverge the original control flow of the program. Both software [1], [9], [11] and hardware [2], [17] mechanisms were investigated to detect unexpected transitions during execution. Software implemented approaches are themselves potential targets of attacks, while tampering with hardware is considerably more challenging. Additionally, hardware implemented mechanisms have the advantage of achieving better performance compared to pure software methods. In fact, guidelines for automotive ECUs development prescribe the use of hardware modules to monitor the execution of safety-critical code [10]. In this paper, we present ControlFreak, a hardware watchdog that monitors program execution and detects any deviation from the predefined control flow graph (CFG). The watchdog uses *signatures* to detect (1) any change in executed instructions and (2) any jump outside the control flow graph. Our signature calculation scheme allows to check that each basic block is executed correctly, i.e., no instruction was modified, and to ensure that for each basic

This is a pre-print version. The definitive version is available at https://doi.org/10.1109/SRDS.2015.35

block only successors that are allowed by the CFG of the program are executed.

We implemented a prototype of the watchdog and evaluated it using simulation. Our experiments show that the performance overhead varies from negligible to moderate (12% on average) for different scenarios.

The contributions of the paper are:

- We present an approach to detect control flow transitions outside of the control flow graph using a hardware watchdog.
- We describe a novel signature calculation scheme that uses *signature chaining*; it allows to check the integrity of instructions comprising a basic block as well as the signatures of allowed successors.
- We present the analysis of the security guarantees provided by our approach.
- We describe the implementation of the watchdog using a simulator and present evaluation.

II. Background and Related work

A. Control flow

Control flow defines the order in which instructions of the program are executed. In the following we assume a generic instruction set architecture (ISA) that can be mapped to almost any modern architecture. We differentiate between the following types of instructions:

Non control flow instructions (i_k) Such instructions (e.g., addition, subtraction, etc.) do not cause jumps in control flow.

Control flow instructions: Such instructions may diverge the program from executing a sequential instruction:

- Direct jump and indirect jump instructions ($jump$ and $jumpx$). These instructions change control flow unconditionally. The target of a direct jump is encoded in the instruction word, while indirect jump uses an address stored in a register.
- Branch instruction ($branch$). A branch instruction has exactly two successors. By asserting the branch condition one of two successors is chosen at run-time.
- Direct call and indirect call instructions ($call$ and $callx$). A call instruction transfers control to a beginning of a subroutine. The target of a direct call is encoded in the instruction word. An indirect call uses an address from a register as the target address.
- Return instruction ($return$). Return instruction transfers control to the next instruction after the corresponding call using previously saved return address from the stack or a register.

A program's control flow graph (CFG) reflects all transitions that are allowed in the program. It is usually based on a notion of a *basic block* (BB) – a sequence of non-control flow instructions terminated by a control flow instruction.

B. Control flow attacks

Control flow attacks change predefined control flow of an application to achieve the attacker's goal. One prominent example of such an attack is a *stack smashing attack*: the attacker overflows a buffer on the stack with malicious code and overwrites the return address with an address pointing to the injected code. This attack can be prevented by making the stack non-executable, preventing the attacker to execute her code located on the stack.

Non-executable stack can be circumvented by a more sophisticated attack known as *return-to-libc* or its generalization *return-oriented programming* (ROP) [20]. These attacks do not require injection of new code and use sequences of instructions already present in the program or in shared libraries. The attacker overwrites several stack frames redirecting control flow to a sequence of *gadgets* - multiple instructions typically terminated with a return. Each gadget performs a certain operation, and by chaining multiple operations together the attacker is able to achieve the desired result. Initially discovered for x86, ROP has been extended to various architectures, including ARM [4] and SPARC [18].

Stack canaries [8] can be used to protect return addresses from being undetectably overwritten by placing special *canary* values before the address that the attacker needs to correctly guess. However, a return address can be overwritten using other vectors of attacks, e.g., format string vulnerabilities. Address space layout randomization is an approach widely adopted in desktop systems. It changes location of the stack, the heap, and shared libraries in order to prevent the attacker from using code sequences at known locations in memory. This approach is less effective on 32-bit architectures [21], thus its applicability in embedded systems is limited.

Even if the attacker is not able to modify return addresses, there exist other possibilities to redirect control flow to a desired location. For example, a buffer overflow can be used to overwrite the target address of a function pointer. Moreover, a modification of ROP [6] exists that does not require execution of return instructions. Bounds checking can be used to protect against buffer overflows, however such approaches (e.g., [3]) usually pose significant performance overhead, since each access requires performing an additional check to detect boundary violation.

The disadvantage of the previously described approaches is that they target only specific vulnerabilities. Control-flow integrity [1] is a more general pure software method that uses binary rewriting to prevent jumps outside of the predefined control flow graph. A label is assigned for each target of an indirect jump or call and special instructions are inserted before each jump to check that a valid successor is targeted at run-time. The approach is based on the assumption that the code segment cannot be modified at run-time, and memory

that belongs to the data segment cannot be executed as code. We do not pose such restrictions on the attacker in our work.

Another direction of the related work are hardware monitors. They typically rely on a hardware extension that uses precalculated information about the program and check that the run-time execution follows the predefined behavior. Hardware monitors were first introduced in the domain of fault tolerance [16], [19]. The fault model used by these approaches assumes single-event upsets and is not suitable to prevent attacks.

Mao *et al.* [17] investigate various information patterns that can be derived from the program to describe its behavior: control flow information (i.e., addresses of basic blocks), opcodes of instructions, load/store patterns, hashes over each instruction and its address. This information is securely loaded to a hardware monitor at run-time. The monitor is connected to the CPU and receives information about program execution and checks if the execution follows the predefined pattern.

The monitor by Arora *et al.* [2] controls the program execution on three levels. On the highest level the monitor checks that each function call is the part of the function call graph. Within a function the sequencing of basic blocks is checked using information about their addresses stored in a table. Additionally, for each basic block hash values of instructions are calculated to check their integrity. The approach uses hardware tables to store the derived data, which complicates software updates. Also, the data is address specific, meaning that the program cannot be relocated.

The shortcoming of the described monitors is that they require expensive on-chip area to store reference information used at run-time. With software sizes growing quickly, the provided area might be not sufficient to hold all required data. By reducing the amount of stored data the capabilities of the monitor to detect errors drop significantly In our work the reference data is stored in RAM, hence the program size is limited only by the amount of available memory, which is typically much larger and cheaper than the on-chip storage.

III. Signature chaining

A. Attacker model and overview

The goal of the attacker is to modify program behavior by executing a sequence of instructions not defined by the program. We assume a powerful remote adversary who is able to exploit bugs in the application, take over the control flow and execute arbitrary code. Hence, memory contents are not trusted as they can be overwritten by the attacker. We need to check that (1) any instruction executed by the CPU is not modified and (2) any jump is performed according to the predefined control flow graph. Even though some embedded systems use read-only memory to store code, there exist several vectors of attacks that justify the adoption of such model. For example, return-oriented programming attacks can be launched, or malicious code can be loaded to a writable memory followed by redirection of control to that location.

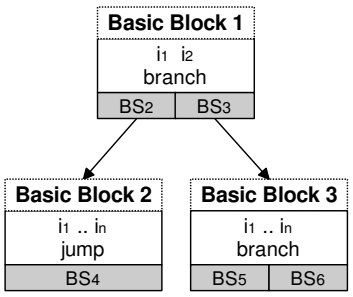

Figure 1: Example of a control flow graph. Gray boxes depict signatures for corresponding successors.

To prevent such attacks, we leverage a hardware watchdog, ControlFreak. For each basic block in a program, we statically calculate a *signature*, which reflects the instructions of the block and information about valid successors. At run-time the signatures are loaded to memory. The watchdog monitors program execution and recalculates signatures based on the actual executed instructions. In the end of each basic block the watchdog compares the precalculated value (read from memory) against the dynamically calculated one. If unexpected instructions were executed, the two values will differ. We assume that ControlFreak is trusted, and the attacker can affect its operation only by modifying the inputs, i.e., instructions or signatures in memory.

In this work we focus only on the detection of attacks, thus different strategies to recover once an attack is detected are out of scope. In our implementation, the CPU is reset as soon as an attack is detected.

To simplify the description, we first introduce the signature calculation procedure for direct control flow transitions and describe program monitoring at run-time. After that we present the techniques for indirect control flow.

B. Static signature calculation

Fig. 1 shows an example of a partial control flow graph. BB_1 consists of two non-control flow instructions followed by a branch instruction. Two basic blocks are valid successors: BB_2 and BB_3. A signature for BB_1 is calculated in the following way:

$$BS_1 := f(i_1, i_2, branch, BS_2, BS_3)$$

where BS_n is block signature, and f is the function used for signature calculation. We will discuss the requirements for the function in Section IV. As the input to the function we use binary representation of instructions comprising the basic block. Additionally, signatures of valid successors are appended. We call this process *signature chaining*.

The intuition behind signature chaining is the following. The program execution starts from a first basic block, for

which the signature is securely transferred to the watchdog and its integrity is checked (see Section IV). Since each signature also contains signatures of successors, once the watchdog performs the check and confirms that the run-time calculated signature is correct, we can conclude that signatures of successors are also correct (i.e., they were not modified in memory). This means that only the basic blocks that correspond to these signatures may be executed next. If another basic block is executed, the signatures will not match and ControlFreak will signal an error. In this way, successors for any basic block in the program are restricted.

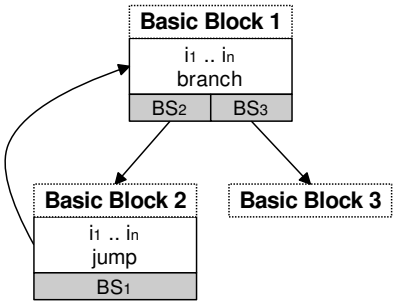

Figure 2: Loops in the control flow graph result in mutual dependencies between signatures: BS_1 and BS_2 contain signatures of each other.

Loops in the control flow graph (Fig. 2) result in a circular dependency between the first and the last block of the loop:

$$\underline{BS_1} := f(i_1, ..., i_n, branch, \underline{BS_2}, BS_3)$$
$$\underline{BS_2} := f(i_1, ..., i_n, jump, \underline{BS_1})$$

To break the dependency we assign a random value BS_{rand} for one of the blocks (BB_2 in our example). This allows us to calculate the signature for BB_1 by using BS_{rand} instead of BS_2:

$$BS_1 := f(i_1, ..., i_n, branch, BS_{rand}, BS_3)$$

Since BS_2 is not equal to the randomly chosen value BS_{rand}, we introduce another factor, a *correction value*, into BS_2, such that BS_2, updated with this value, results in BS_{rand}:

$$f(BS_2, Correction) := BS_{rand}.$$

Note that to calculate the correction value we require the function f to have an inverse. A practical example of calculating a correction value is presented in Section V-C.

C. Signature table

Calculated signatures and information about successors are organized in a *signature table*. Each row in the table corresponds to a basic block in the control flow graph. Depending on the type of the control flow instruction that terminates the block, different number of successors may be provided. If the basic block has a correction value, it is also

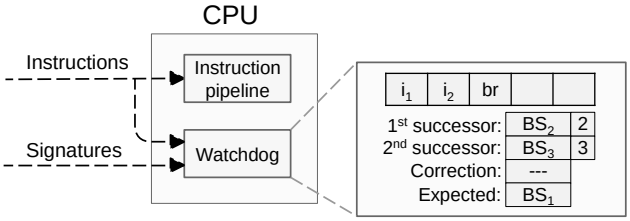

Figure 3: State of the watchdog.

reflected in the table. Each successor is represented with a pair of values: the signature of the block and its row id.

An example of a signature table is presented in Table I. In this table BB_1 has two successors, BB_2 and BB_3, which have signatures BB_2 and BB_3 respectively. BB_2 has only one successor that has the signature BB_4, and BB_3 also has two successors - BB_5 and BB_6. The table is loaded to memory at run-time along with the program binary and is used by the watchdog to obtain pre-calculated values.

Table I: Signature table example.

Row	First successor		Second successor		Correction
1	BS_2	2	BS_3	3	—
2	BS_4	4	—	—	—
3	BS_5	5	BS_6	6	—

D. Checking at run-time

Fig. 3 shows the state of the watchdog in the end of BB_1 from our example in Fig. 1. The state consists of an *expected* register, an instruction buffer, and the signature table row for the current block (i.e., signatures of successors, their row ids and a correction value if required).

The watchdog monitors the instruction stream, executed by the CPU. It accumulates instructions in the buffer until a control flow instruction is detected. Then it fetches the signatures of successors of the current basic block, recalculates the signature and compares the resulting value against the pre-calculated one, stored in the *expected* register. If the values do not match, an error is signalled and the execution is stopped. If the two values match, the watchdog chooses one of the signatures of successors as the next expected value depending on the path taken by the program.

E. Indirect control flow

Several types of control flow instructions do not have successors directly specified in the instruction word. We treat such instructions differently, since indirect control flow instructions might have a large number of successors, and we adapt the calculation procedure for such instructions.

1) Indirect jumps and calls: Indirect instructions can have more than two possible successors since the target of a jump or a call is stored in a register. We assume that all

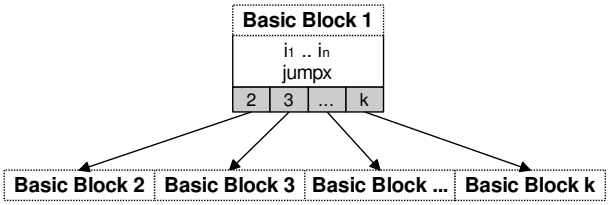

Figure 4: Indirect jump example. BB_1 can have a large number of possible targets.

possible targets of such instructions are known. The concrete technique to extract the targets is discussed in Section V.

An example of a basic block (BB_1) containing an indirect jump is presented in Fig. 4. The straightforward way to calculate the signature for BB_1 is to follow the scheme presented in Section III-B, including all signatures of all possible successors in the signature of such block:

$$BS_1 := f(i_1, ..., i_n, jumpx, BS_2, BS_3, ..., BS_k)$$

Alternatively, only the signature of the actual target of the jump chosen at run-time can be used to reduce the number of signatures read from memory, and the number of operations to calculate the signature for BB_1. However, using just the target signature will result in a different run-time signature for BB_1 depending on the chosen target. Therefore, we use the same approach introduced for loops: We choose a random value for BB_1 (BS_{1rand}), and for each allowed successor t we calculate a correction value $Corr_t$:

$$BS_{1rand} := f(i_1, ..., i_n, jumpx, BS_t, Corr_t)$$

Signatures of all successors of such jumps and the corresponding correction values are stored in the signature table (see Section V for details). At run-time, the watchdog needs to fetch only the signature and the correction value for the selected target to calculate the signature for BB_1.

Indirect calls are treated in a similar fashion. The only difference is that the signature for the basic block after return is additionally checked.

2) Function calls and returns: Whenever a function is called, it must return to the instruction following the corresponding call. These locations (and the corresponding basic blocks) differ depending on the call site. An example of a function call is presented in Fig. 5. Function A (consisting of one basic block BB_A for simplicity) is called from two places: from BB_1 and BB_2.

We use the following approach to check that the return address was not modified and the function returns to the correct location. For each basic block that contains a call instruction (BB_1 and BB_2 in our example), a basic block starting with the next instruction after the call is considered to be a successor (BB_3 and BB_4 respectively); signatures of basic blocks with return instructions (BB_A) do not contain

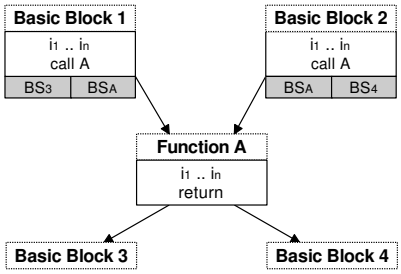

Figure 5: Function call example. Depending on the call site either BB_3 or BB_4 will be executed after returning from A.

any signatures of successors. Hence, we calculate signatures for BB_1 and BB_A as follows:

$$BS_1 := f(i_1, ..., i_n, call, BS_A, BS_3)$$
$$BS_A := f(i_1, ..., i_n, ret)$$

At run-time, once BB_1 passes the check, BB_3 is stored inside the watchdog on a designated stack. When a return instruction is executed, the signature from the top of the stack is popped and used as the expected value. Using the signature stack allows us to restrict the set of basic blocks following a return instruction to a single block, and not to any possible target as in the case of indirect jumps and calls.

IV. Security Analysis

In this section we analyse the assumptions of our approach and discuss why ControlFreak can detect any control flow based attack.

A. Assumptions

Our attacker model assumes that the adversary is able to modify both the code and the contents of the signature table (Fig. 6). To be able to detect attacks, we rely on the two following assumptions which restrict the attacker's capabilities.

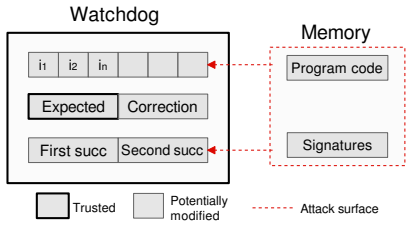

Figure 6: Attack surface. Instructions and signatures stored in memory are potentially modified and are not trusted.

Assumption 1: The result of applying a signature function f to two different inputs is different.

This assumption states that there are no collisions, i.e., it is not feasible for the attacker to find two different instruction

sequences and signatures of successors that have the same signature.

Assumption 2: The attacker cannot control the output of the signature function even having the control over the input.

Since the attacker control the memory contents, she could modify the signature table and provide correct signatures for injected malicious code. This assumption excludes such possibility.

Cryptographically secure functions, such as MD5 or SHA, can be used to maximize the probability that these two assumptions hold. However, they cannot be easily inversed due to their nature, thus the program cannot contain loops. This limitation can be eliminated by using an encryption algorithm, since it can be easily inversed (decrypted) if the secret key is known. Without knowing the key, the attacker is not able to calculate valid signatures. The same holds for correction values, since it would require decrypting the randomly chosen value.

Assumption 3: The watchdog contains a key that cannot be obtained by the attacker.

To perform signature calculation the watchdog needs to obtain the key used to create signature table. This assumption implies that this key can be securely transferred into the watchdog. This could be achieved by having a private key installed inside the CPU, and using the corresponding public key to encrypt the application key.

Assumption 4: The first signature BS_1 is securely transferred into the watchdog and its integrity is checked.

We require that the signature of the first basic block in a program is correct. If the attacker is able to undetectably modify the first signature, she can start program execution from any basic block. A message authentication code for the signature can be calculated and checked by ControlFreak upon start.

Assumption 5: The attacker needs to execute a control flow instruction in order to succeed.

Since the watchdog checks only on a basic block boundary, the attacker could inject code that does not contain any control flow statements and thus the check will be never performed. This assumption could be relaxed by including the number of instructions of each basic block into the signature. The watchdog then will perform the check as soon as the specified number of instructions is executed regardless of the type of the last instruction.

B. Discussion

Consider the execution of the first basic block (BB_1) of the program. According to Assumption 4 the signature for this block is securely stored inside the watchdog and is guaranteed to be correct.

The minimal possible goal of the attacker is to execute a single instruction not in the program order. This can be seen as executing BB_x instead of BB_1. Both blocks are comprised of the same instructions except for one (Fig. 7).

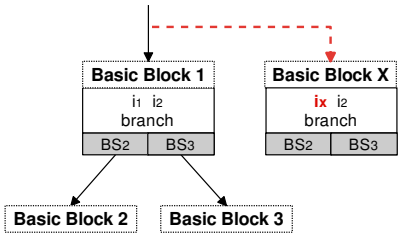

Figure 7: Attacks are carried out by modifying the instruction stream.

Once the instructions of BB_1 are modified, the attacker needs to either change the rest of instructions of the basic block, or signatures of successors such that the new input will result in the expected signature BS_1. However, according to Assumption 2 the attacker cannot provide the input such that the outcome of the signature function results in the chosen by the attacker value. Hence, if the first basic block was modified, it will be detected by ControlFreak.

Since a signature also encompasses the signatures of successors, a successful check additionally guarantees that not only instructions of the current block were not modified, but also that the signatures are correct and belong to valid successors. This means that the expected value can only be updated with a valid signature. Hence, for any succeeding basic block the same arguments can be used to show that any modification of instructions or signatures is detectable by the watchdog. Thus, the expected register (Fig. 6) is trusted: it is always either updated with a correct signature, or the execution is stopped.

C. Limitations

The focus of this work is on control flow restriction. An obvious limitation are data flow based attacks. If the attacker is able to overwrite, for example, a critical variable that provides her with more privileges without modifying the control flow, such an attack will not be detected by the watchdog. Also, ControlFreak can only detect jumps outside the control flow graph. If the attacker manages to perform a jump to a valid successor, but not a correct one, such an attack will stay undetected, since the target is a part of the CFG. For instance, the attacker could modify a branch condition and make the program take a different path, or change the number of loop iterations. To detect such attacks additional data flow protection is required.

V. Implementation

A. Watchdog architecture

We described the watchdog using SystemC and a cycle-accurate simulator of a 32-bit RISC CPU. The CPU im-

plements an instruction set architecture called *Xtensa*. Our implementation closely follows the description provided in Section III. ControlFreak receives the stream of instructions from the commit stage of the CPU pipeline and performs signature calculation. Memory controller component fetches signatures from memory which are then compared against dynamically calculated counterparts.

The watchdog processes multiple basic blocks simultaneously. Additional metadata is maintained to distinguish between different blocks, thus basic blocks can be checked out-of-order, synchronizing with the program only on externalization points, i.e., before executing a system call to prevent possibly malicious state from being outputted.

If an interrupt occurs during the execution, the control is transferred to one of the interrupt handlers. Such transition would be considered as an attack by the watchdog due to unexpected executed instructions. To detect such cases, the watchdog monitors the `PS` (processor status) register. This register contains a flag, which is set when an interrupt is being handled. Execution of all handlers is checked as well. Once the CPU starts handling an interrupt, ControlFreak initiates a new basic block for the handler and fetches the corresponding signature. When the execution returns to the application, the watchdog continues with the BB that was being executed before the interrupt.

B. Control-flow graph extraction

To obtain the control flow graph of a program and produce a signature table we implemented a signature calculation tool. As input it uses a disassembly file of the program binary produced by *objdump* and does not require the source code of the application or any modification of the program binary, hence legacy software can be supported. The tool parses the file, extracts all instructions, and groups them into basic blocks. For each basic block successors are identified. Produced signature table follows the same format as described in Section III.

The tool is able to resolve targets of indirect jumps, which are typically used to implement *switch* statements. Consider the following example of the instruction sequence performing an indirect jump:

```
(1)  l32r    a8, 6007a7a4
(2)  addx4   a8, a14, a8
(3)  l32i.n  a8, a8, 0
(4)  jx      a8
```

Line 1 loads the base address of the jump table into `a8` register. Line 2 adds the offset of the target in the table (stored in register `a14`), to the base address and stores the result in `a8`. Line 3 loads the calculated address from memory to `a8`. Finally, the address is used by *jx* (indirect jump) instruction on line 4. The loaded base address points to the read-only section of the binary. The targets are extracted by sequentially traversing memory starting at the pointed address. To check that the discovered values are indeed possible targets of the jump, we use a simple heuristic: the address has to point to the beginning of an instruction within the same function. Clearly, this solution is not perfect. For example, if tables for two indirect jumps in the same function are stored sequentially in memory too many possible targets will be identified. To restrict the target set further, a more sophisticated data flow analysis is required to identify the possible offset values. The produced signature table is adapted to allow for more than two targets. The field used to store the row id of the first successor contains the number of successors, and for each successor a row follows that contains the address offset of the target instruction, the corresponding signature and a correction value.

Our current implementation does not perform any analysis of indirect calls, hence we conservatively add all functions present in the program to the set of possible targets for any such call. Similar to indirect jumps, for each function we add an additional row containing its offset and the signature of the first basic block.

Once all signatures are calculated, the signature table is inserted into the program binary. We use *objcopy* to create an additional section at a predefined address where the signatures are stored. Besides using the tool, we do not require any further changes in the existing tool chain to produce the binary. Nevertheless, signature calculation can be also done at compile-time, leveraging existing compiler infrastructures and techniques to obtain more precise CFG.

Upon boot, the CPU is initialized by a simple operating system - a single-threaded runtime, which includes routines for initialization, *libc* and a system call library. OS code and libraries are statically linked with the program binary (a common case in embedded systems), and are available for analysis by the signature calculation tool. They are checked by ControlFreak at run-time as well.

C. Signature calculation

We use Advanced Encryption Standard (AES) as the signature calculation function. AES is a symmetric encryption algorithm that supports multiple key-sizes (128, 256 and 512 bits) and operates on 16 B blocks, which defines the size of a signature. All instructions comprising a basic block are split into 16 B chunks and encrypted using cipher-block chaining (CBC) mode of operation. In this mode, the input to the AES engine is XORed with the result of the previous encryption or an initialization vector (IV) for the first operation. We use the same initialization vector for all basic blocks. Once the

http://ip.cadence.com

Randomized vectors are necessary for the same message to have different ciphertexts when encrypted with the same key, which is not a requirement in our setting.

instructions are encrypted, signatures of successors are used to obtain a complete signature for a basic block. Free bytes that might appear when instructions do not complete the buffer are filled with zeroes, and not with a signature of the first successor. This is due to the performance reasons, since at run-time it allows performing encryption of instructions and fetching of signatures from memory in parallel.

Correction values are calculated in the following way. First, we obtain $BB_{partial}$ by encrypting the instruction sequence and the signatures of successors. As an example we use BB_2 from Fig. 2:

$$BB_{partial} := AES(AES(i_1, ..., i_n, jump) \oplus BS_1)$$

Then, we select a random value BB_{rand}. The correction value should satisfy the following equation:

$$BB_{rand} = AES(BS_{partial} \oplus AES(Corr))$$

Hence, to calculate $Corr$ we decrypt BB_{rand}, XOR the result with $BB_{partial}$ and decrypt the result again.

VI. Evaluation

A. Performance

1) Methodology: We evaluated ControlFreak using MiBench [13] - an embedded benchmark suite that simulates a typical workload in embedded systems. We made no changes either to the source code, or to the binary, except for adapting makefiles to use the compiler for the targeted architecture, and adding signatures using the tool described in Section V. All programs were compiled with -O3 flag. In our experiments we used small inputs provided by the suite due to substantial execution time in the simulator. To demonstrate the ability to protect more complex software, we chose sqlite, a widely used database engine, as an additional benchmark. As the workload, we populate an in-memory database with multiple rows and perform a select query on the data. We disabled multi-threading as it is not supported by the execution environment and our implementation. We execute each application with and without the watchdog, and measure the number of consumed cycles. Since all programs are deterministic, there is no difference in the number of cycles for the same application between different runs.

We examine three scenarios:

- General-purpose (GP). In this scenario we use the simulation parameters specified in Table II. The AES engine was modeled after [14]. The values used for the number of pipeline stages and the encryption latency correspond to the variant with the lowest area overhead. We use instruction cache to additionally store signatures.
- Real-time systems (RT). Using cache in real-time embedded systems complicates the analysis of execution time and hence caches are often not used. To assess performance of ControlFreak in such setups, we disable cache forcing every access to go directly to memory.
- Perfect cache (PC). To investigate how the AES engine affects performance of the watchdog, we eliminate the impact of the memory subsystem by performing all memory accesses within 1 cycle thus simulating ideal cache behavior.

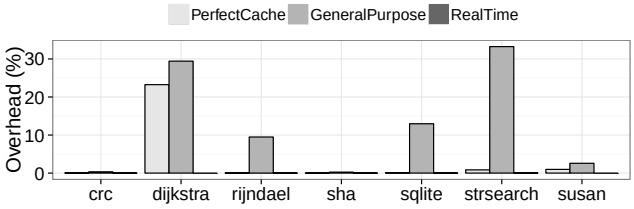

Figure 8: Performance overhead of the watchdog.

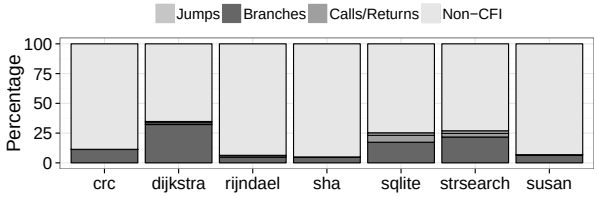

Figure 9: Distribution of executed instructions by type.

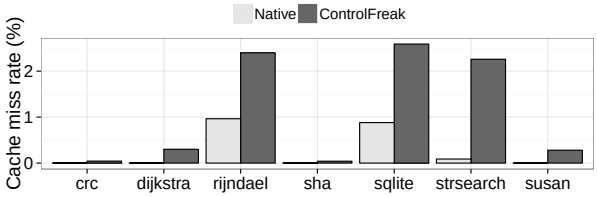

Figure 10: Cache miss rate.

Table II: Simulation parameters

Parameter	Value
CPU pipeline stages	5 stages
CPU clock frequency	200 MHz
Memory access time	20 cycles
Cache access time	1 cycle
Cache hierarchy	L1-I and L1-D caches
Cache associativity	8-way set associative
Cache line size	32 B
Cache size	4 KiB
Cache eviction policy	LRU
Encryption latency	31 cycles
AES pipeline stages	4 stages
AES key size	128 bit

2) Experimental results: The results are presented in Fig. 8. RT and PC variants pose negligible overhead with one

http://www.sqlite.org

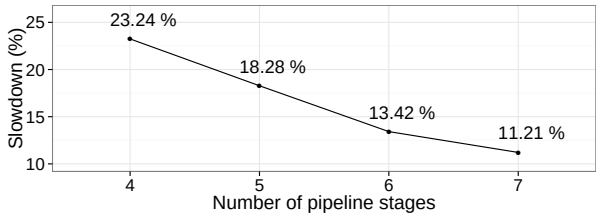

Figure 11: Effect of the number of pipeline stages in the encryption engine on performance.

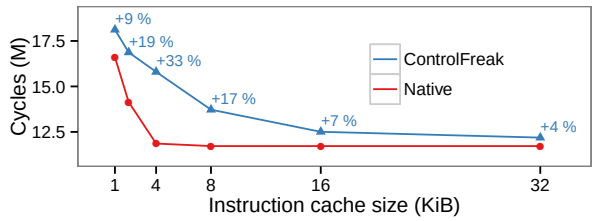

Figure 12: Performance of strsearch with varying cache sizes.

exception - dijkstra (23%). Introducing cache changes the picture: several applications have overhead of 10% or more (12% on average). To understand the cause, we further investigate two applications (dijkstra-PC and strsearch-GP) in more detail, as they they exhibit the highest slowdowns.

Examining execution traces of dijkstra showed a very high number of branch instructions, often performed one after another in a sequence. Such control flow intensive behavior renders the worst-case scenario for our approach, since a single branch instruction constitutes a basic block and requires signature calculation, i.e., performing at least 3 encryptions, quickly saturating the AES engine. To verify our observations, we used execution traces and extracted the type information for all executed instructions. The results are summarized in Fig. 9. Approximately 35% of all instructions in dijkstra are control flow instructions - the highest number among all programs.

To demonstrate the influence of the AES engine on performance, we increase the number of pipeline stages allowing for higher throughput - allowing for more basic blocks to be checked in parallel. With each additional stage we also conservatively increase the encryption latency by 5 cycles. Fig. 11 presents the results for dijkstra. Increasing the number of pipeline stages reduces the overhead from 23% to 11%. Hence, using a better encryption engine can reduce the overhead for programs with a large amount of control flow instructions.

In GP scenario, three applications (rijndael, sqlite and strsearch), besides dijkstra discussed above, pose non-negligible overhead. Since this is not the case in PC experiments, we demonstrate how the signatures influence the cache behavior. Fig. 10 shows miss rates of the instruction cache during GP runs. 4 KiB cache is large enough for most applications, thus miss rates are low. Rijndael and sqlite show higher miss rates than other benchmarks when executed without the watchdog. This is reflected in the overhead once ControlFreak is enabled, since the number of misses increases further due to additional conflicts caused by fetching signatures. In case of strsearch, miss rate increases significantly when executed with the watchdog compared to other applications, resulting in a high performance overhead. Fig. 12 demonstrates the impact of the cache size on performance of strsearch. The native application observes an increase in performance with cache sizes up to 4 KiB, however larger caches do not have further effect on the execution time. With ControlFreak, performance of the application also increases, but at a slower pace, resulting in growing overheads (with sizes up to 4 KiB). Once the native application reaches its limit and does not perform better, the overhead quickly decreases. Since we do not observe a high overhead for strsearch in the PerfectCache scenario, another way to improve performance (besides increasing the cache size to accommodate signatures) is by structuring the signature table in a cache-friendly way. It can be tailored to the particular cache parameters such that often used signatures do not conflict with each other and with instructions, causing an increase in miss rate. We leave such study as a direction of future work.

B. *Signature table size*

Table III: Additional memory required for signatures.

	Binary size (KiB)	Signatures (KiB)	Overhead (%)
crc	333	110	33
dijkstra	459	200	43
rijndael	353	112	31
sha	310	108	34
sqlite	1407	1457	103
strsearch	300	106	35
susan	429	182	42

Another direction of our evaluation concerns the additional storage required for signatures. Table III shows the binary size for each application and the size of the corresponding signature table. MiBench applications require 36% of additional storage on average. The larger size of the signature table for sqlite is due to the complexity of the program: it contains a large number of functions, indirect jumps and calls, resulting in much higher overhead.

VII. Conclusion

Control flow attacks are often used by the adversaries to obtain the control over the system. We presented a novel signature calculation scheme that detects such attacks without limiting the size of the program by allowing to store signatures in main memory. Code injection attacks are

excluded by using an encryption algorithm to calculate signatures, and modification of existing instruction sequences is not feasible due to signature chaining. We implemented a prototype of a watchdog that uses signatures to monitor program execution and to detect violations of control flow. Experimental results show small performance overhead due to the use of additional hardware which can be further reduced by employing a faster encryption engine and better utilization of caches.

While ControlFreak can effectively detect deviations from the predefined control flow of an application, another types of attacks (e.g., data flow attacks) remain feasible and require orthogonal protection measures.

VIII. Acknowledgment

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement 645011 (SERECA) and from the state of Saxony under grant of ESF 100111037 (SREX).

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communication Security (CCS)*, 2005.

[2] D. Arora, S. Ravi, A. Raghunathan, and N. Jha. Secure embedded processing through hardware-assisted run-time monitoring. In *Design, Automation and Test in Europe (DATE)*, 2005.

[3] E. D. Berger and B. G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[4] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[5] R. N. Charette. This car runs on code. *IEEE Spectrum*, 2009.

[6] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *ACM Conference on Computer and Communication Security (CCS)*, 2010.

[7] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Conference on Security*, 2011.

[8] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[9] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi. Mocfi: A framework to mitigate control-flow attacks on smartphones. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[10] EGAS Workgroup. *Standardized E-Gas Monitoring Concept for Gasoline and Diesel Engine Control Units*. Version 5.5 edition, July 2013.

[11] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. Xfi: Software guards for system address spaces. In *Symposium on Operating System Design and Implementation (OSDI)*, 2006.

[12] S. Furst. Challenges in the design of automotive software. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2010.

[13] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE International Workshop on Workload Characterization*, 2001.

[14] A. Hodjat and I. Verbauwhede. A 21.54 gbits/s fully pipelined aes processor on fpga. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2004.

[15] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental security analysis of a modern automobile. In *IEEE Symposium on Security and Privacy (SP)*, 2010.

[16] A. Mahmood and E. McCluskey. Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers*, 1988.

[17] S. Mao and T. Wolf. Hardware support for secure processing in embedded systems. *IEEE Transactions on Computers*, June 2010.

[18] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 2012.

[19] N. Saxena and E. McCluskey. Control-flow checking using watchdog assists and extended-precision checksums. *IEEE Transactions on Computers*, Apr 1990.

[20] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[21] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security (CCS)*, 2004.

[22] M. Wolf, A. Weimerskirch, and T. Wollinger. State of the art: Embedding security in vehicles. *EURASIP Journal of Embedded Systems*, 2007.

[23] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, Oct. 2009.