Dynamic Information Flow Tracking for Embedded Binaries using SystemC-based Virtual Prototypes*

Pascal Pieper[1] Vladimir Herdt[1] Daniel Große[1,2] Rolf Drechsler[1,2]
[1]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
[2]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
{Pascal.Pieper,Vladimir.Herdt}@dfki.de {grosse,drechsle}@informatik.uni-bremen.de

***Abstract*—Avoiding security vulnerabilities is very important for embedded systems. *Dynamic Information Flow Tracking* (DIFT) is a powerful technique to analyze SW with respect to security policies in order to protect the system against a broad range of security related exploits. However, existing DIFT approaches either do not exist for *Virtual Prototypes* (VPs) or fail to model complex hardware/software interactions.**

In this paper, we present a novel approach that enables early and accurate DIFT of binaries targeting embedded systems with custom peripherals. Leveraging the SystemC framework, our DIFT engine tracks accurate data flow information alongside the program execution to detect violations of security policies at run-time. We demonstrate the effectiveness and applicability of our approach by extensive experiments.

I. Introduction

Embedded systems are small application specific devices with a broad range of applications, such as the *Internet-of-Things* (IoT) or automotive. They integrate several peripherals alongside the CPU core and extensively rely on embedded SW for configuration as well as complex functionality and communication. Avoiding security vulnerabilities in the embedded SW is crucial to prevent leaking sensitive information or compromising safety.

Dynamic Information Flow Tracking (DIFT) [1], [2] is a powerful technique to analyze and protect software against a broad range of security related exploits by tracking and checking the information flow between inputs and outputs alongside the SW execution. Therefore, the DIFT engine is configured according to a security policy that essentially specifies the *classification* of input data, the rules of propagation (*Information Flow Policy – IFP*) and what kind of information is allowed to leave the system at which output interfaces (*clearance*) [3]. A *security policy* enables the specification of several fine grained *Access Control Models* (ACMs) including *confidentiality* (secret data must not leak to untrusted places) as well as *integrity* (untrusted data must not influence sensitive registers/data).

While several SW- and HW-based approaches for DIFT have been proposed, they suffer from deficiencies if SW targeting embedded systems is considered: i) SW-based approaches do not consider the HW in sufficient details and thus are susceptible to miss complex HW/SW interactions, e.g. due to interrupts, memory mapped peripheral access as well as *Direct Memory Access* (DMA) controllers, and ii) HW-based approaches can only be used once the HW is available, hence the development and validation of security policies has to wait until then. At the same time, the security policy has influence on the SW development and HW design, hence it is important to consider security policies *early* in the design flow to avoid costly iterations.

*This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project SATiSFy under contract no. 16KIS0821K and within the project VerSys under contract no. 01IW19001.

Contribution: In this paper we present a novel approach that enables early and accurate DIFT of SW binaries targeting embedded systems. Our approach works by integrating the DIFT engine in combination with the security policy into a *Virtual Prototype* (VP) of the embedded system. VPs are essentially executable SW models of the entire HW platform, and they are pre-dominantely implemented in IEEE-1666 SystemC [4] employing *Transaction Level Modeling* (TLM) [5] for abstract communication, and hence very fast simulation. Therefore, VPs are heavily used for early SW development and design space exploration [6], [7]. Our **approach extends the VP use-cases to early development and validation of security policies**. Leveraging the VP, the proposed DIFT engine can track information flow on the embedded binary taking fine-grained HW/SW interactions into account. As SystemC is a C++ class library, we can benefit from the C++ features of templates and operator overloading to enable a transparent and virtually non-intrusive integration into the VP. We demonstrate the effectiveness and applicability of our approach in several RISC-V experiments. This includes the development of a security policy for a car engine immobilizer, the detection of code injections as well as the evaluation of the performance overhead.[1]

Summarizing, the major contributions of this paper are:

- VP-based DIFT on embedded binary taking fine-grained HW/SW interactions into account
- Early development and validation of security policies, before the HW is available
- Transparent and virtually non-intrusive integration in VP
- Moderate performance overhead using VP-based DIFT

Paper Structure: Section II discusses related work. Afterwards, the basics of SystemC are reviewed in Section III. The definition of a security policy and the threat model is given in Section IV. Then, in Section V we present our proposed VP-based DIFT approach. Finally, Section VI describes our experimental results and Section VII concludes the paper.

II. Related Work

Several HW-based DIFT approaches have been proposed. For example [8]–[11] focus on integration of DIFT into processor cores. There are also some approaches for extending DIFT support to the whole SoC [12]–[14]. Finally, several approaches consider DIFT at RTL and gate-level in general [15], [16]. HW-based DIFT is complementary to our VP-based DIFT, since our approach enables early development and validation of security policies before the HW is available. In addition, requirements

[1]Visit http://www.systemc-verification.org/risc-v to find the open source implementation of our VP-based DIFT engine for RISC-V and also our most recent RISC-V related approaches.

for the HW mechanisms can be derived. There also exist various SW-based DIFT approaches, e.g. [17]–[19], and methods based on static analysis and symbolic execution focusing on security validation, e.g. [20]–[22]. However, due to the source-level abstraction it is very challenging to provide accurate models for peripherals and to consider complex HW/SW interactions such as interrupts and DMA accurately. [1] integrated a DIFT engine into the Bochs x86 emulator to enable DIFT of SW binaries with full platform support. [23] is conceptually similar but uses QEMU. However, these approaches only target very specific security aspects (integrity-based validation [1] and malware detection [23]) instead of generic security policies, and only offer limited support for data flows outside of the CPU which are necessary to track fine grained HW/SW interactions. In addition, they do not support SystemC-based VPs, which is an industry-proven modeling standard (IEEE-1666).

Finally, an approach for SoC security validation using VPs has been proposed in [24]. However, the approach targets to find security vulnerabilities in the VP model, i.e. the HW. In [25] a dynamic VP-based IFT method for security validation has been introduced. However, the approach only supports a much simpler security policy and threat model compared to this work. Overall, a VP-based generic binary-level DIFT approach specifically tailored for embedded SW binaries is not yet available to the best of our knowledge.

III. SystemC and TLM

SystemC TLM is an industry-proven modeling standard to create VPs [6]. SystemC is not a new language, rather a C++ class library which includes an event-driven simulation kernel [4], [26]. The structure of a SystemC design is described with ports and modules, whereas the behavior is modeled in processes which are triggered by events. Communication between SystemC modules is abstracted using TLM transactions at the cost of timing accuracy, but significant improvements in simulation speed, i.e. up to a factor of 1,000 in comparison to RTL simulation. A transaction object essentially consists of a command (e.g. read/write) and the data (payload) to be transmitted. Transactions are routed based on their address from an initiator to a target socket which is all defined in the SystemC TLM-2.0 standard.

IV. Security Policy and Threat Model

A. Security Policy

A *security policy* consists of three parts: (i) the **classification** which assigns security classes to data that enters the system, (ii) the **Information Flow Policy (IFP)** which is a lattice of security classes that describes the allowed information flow in the system and how the combination of differently labeled data is computed when the data propagates through the system, and (iii) the **clearance** which assigns allowed security classes to system outputs and execution units. Recall that output/execution to/of data labeled with a certain security class is allowed iff the flow of the given security class X to the output/execution security class Y is allowed according to the IFP, i.e. there is a (transitive) connection from X to Y (denoted as *allowedFlow(X,Y)*).

Security policies enable the specification of several *Access Control Models* (ACMs) including *confidentiality* (secret data must not leak to untrusted places) as well as *integrity* (untrusted data must not influence sensitive registers/data).

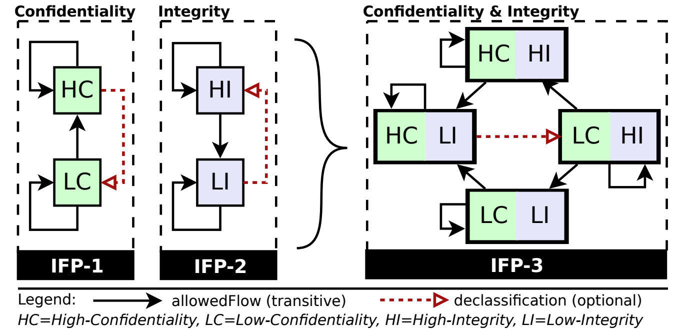

Fig. 1. Three example IFPs. IFP-1 and IFP-2 show a simple policy that models confidentiality and integrity, respectively. IFP-3 is a natural combination of IFP-1 and IFP-2, thus modeling confidentiality and integrity together.

In the following we provide an example to demonstrate the principles of IFPs.

Example 1. *Fig. 1 shows three IFPs. IFP-1 (see left side of Fig. 1) has two security classes:* ***High-Confidentiality*** *(HC) and* ***Low-Confidentiality*** *(LC). Data flow is allowed from LC to HC but not the opposite way, i.e. confidential information is not allowed to leave the system through an output interface without appropriate clearance. IFP-2 (see middle of Fig. 1) only allows data flow from a* ***High-Integrity*** *(**HI**) to a* ***Low-Integrity*** *(**LI**) security class, i.e. untrusted data (LI security class) is not allowed to influence sensitive data (HI security class).*

It is possible to consider confidentiality and integrity together as shown in IFP-3 (right side of Fig. 1). IFP-3 is a natural combination of IFP-1 and IFP-2 by combining the individual security classes (hence, IFP-3 has 4 security classes) and allow a flow iff the individual flows are allowed in IFP-1 and IFP-2.

An important operation on an IFP (lattice) is the *Least Upper Bound* (LUB). Essentially, the LUB of two security classes A and B denotes the next security class C that has equal or more restrictive clearance than both A and B. LUB is used to compute the resulting security class when applying operations (like addition, shift, etc) on data with different security classes. For example, in IFP-3 the LUB of *A=(LC,LI)* and *B=(HC,HI)* is *C=(HC,LI)* which essentially means that the resulting data becomes untrusted (as specified in A) but stays confidential (as specified in B).

Declassification: Another important concept is *declassification* [3], [27]. It allows introducing fine-grained exceptions to the IFP by selectively changing the security class of specific data at run-time (cf. red dashed arrows in Fig. 1). Typically, only trusted HW peripherals are allowed to declassify data to reduce the risk that an attacker exploits the declassification mechanism.

The main use case for declassification is to ensure that a system operating with confidential information can interact with the environment. A concrete example is changing the data classification to non-confidential after it has been encrypted (otherwise no encrypted information could be send out on a public output interface because it depends on a secret key, even though in practice the secret key is sufficiently protected with getting only access to the encrypted data). Another example is a login prompt that leaks internal (secret) information about the password with every attempted login and thus would be blocked by a strict security policy without declassification.

B. Threat Model

We assume a threat model where an attacker can write arbitrary (malicious) data at every input port of the embedded system. The goal of the attacker is for example to obtain confidential information or destroy the integrity of the system. The primary attack vector is to exploit functional SW bugs as well as accidentally included information flows, for example indirect/implicit information flow or an unsecured debug port.

In our approach we assume that the HW is trusted and only the HW can perform declassification. We further assume that the initial SW binary, to be executed on the system, cannot be changed by an attacker.

The security policy of the system is specified by the (security) engineer. How our VP-based DIFT approach works and can be used to validate the security policy is presented in the next section.

V. DIFT FOR EMBEDDED BINARIES USING VPS

Our VP-based DIFT approach tracks information flow on the binaries for embedded systems with peripherals. This is performed taking *fine-grained HW/SW interactions* into account, i.e. the flow is also tracked within the peripherals and the way back to the SW. Our DIFT engine benefits from the SystemC/C++ features of templates and operator overloading to enable a transparent and virtually non-intrusive integration into the VP.

In the following, we start with an overview of our approach (Section V-A) and then present more details on our DIFT engine and VP integration (Section V-B).

A. Approach Overview

Our approach is centered around a VP that represents the target SoC. An overview of our approach is shown in Fig. 2. We integrate a DIFT engine into the VP that enables DIFT at the VP level (see center of Fig. 2) and we specify security policies that are encoded into the VP and checked alongside the SW execution. Please recall from Section IV-A that a security policy consists of three components that reason about security classes: i) classification, ii) IFP, and iii) clearance.

We represent security classes in the DIFT engine as (integer) *tags* by simply mapping each security class of the IFP to a unique tag (see first red box on the right side of Fig. 2 below IFP implementation). Tags are assigned to input data (for example a secret key stored in memory or the data generated by a sensor peripheral) and output interfaces (e.g. the output port of a UART) according to the classification and clearance mappings, respectively (see left side below Virtual Prototype in Fig. 2). In addition, we specify the *execution clearance* by assigning tags to specific execution units in the CPU. We discuss the concept of execution clearance later in Section V-B2 in more detail. To implement the specified IFP, we provide the LUB and *allowedFlow* functions that operate on tags according to the IFP semantics (bottom red boxes on the right side of Fig. 2). Based on these two functions, the DIFT engine propagates and checks the tags, triggering a runtime error upon violation.

B. DIFT Engine

*1) **Implementation Sketch:***

The main ingredient of our DIFT approach is a custom *Taint* data type with a template parameter *T* for the to be tainted data.

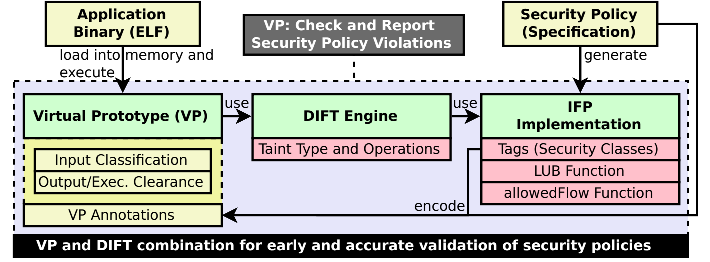

Fig. 2. Overview of our VP-based approach

```
1  typedef uint8_t Tag;
2  template <typename T>
3  class Taint {
4    T value;  // data
5    Tag tag; // security class
6
7    Taint(const T value, const Tag tag) {
8      this->value = value;
9      this->tag = tag;
10   }
11   // convert instance to and from a Taint byte array
12   void to_bytes(Taint<uint8_t> ar[sizeof(T)]) const {
13     for (uint8_t i=0; i<sizeof(T); i++) {
14       ar[i].value = ((uint8_t*)(&value))[i]; // copy each byte
15       ar[i].tag = tag; // use the same tag for each byte
16     }
17   }
18   void from_bytes(Taint<uint8_t> ar[sizeof(T)]) {
19     tag = ar[0].tag;
20     for (uint8_t i=0; i<sizeof(T); i++) {
21       tag = LUB(tag, ar[i].tag); // combine all tags
22       ((uint8_t*)(&value))[i] = ar[i].value; // copy each byte
23     }
24   }
25
26   void check_clearance(uint8_t required_tag) const {
27     if (!allowedFlow(tag, required_tag))
28       throw ClearanceException();
29   }
30
31   // Operator overloading to perform regular operation according
           to data of type T _and_ tainting
32   Taint<T> operator+(const Taint<T>& other) {
33     // apply operation and merge tags according to IFP
34     Taint<T> ans(value + other.value);
35     ans.setTag(LUB(tag, other.tag));
36     return ans;
37   }
38   //...other operators implemented similarly...
39 }
```

Fig. 3. Code excerpts of custom *Taint* data type

Fig. 3 shows the main code excerpts of the *Taint* struct: *value* stores the data (Line 4) and *tag* captures the assigned security class (Line 5).

We now use this data type to represent CPU and peripheral registers as well as memory bytes. More precisely, we selected the open-source RISC-V VP from [28], [29] as a representative example and performed the following three modifications in the SystemC VP model:

1) Replace the register types to use our *Taint*<int32_t> data type instead of the native int32_t. With the *Taint* operator overloading (see Line 32 and following), the RISC-V instruction execution, e.g. an addition *regs[RD] = regs[RS1] + regs[RS2]*, works without any further modification, but now also performs the tainting wrt. the given security policy (Line 34 shows the addition and Line 35 shows the taint update based on the least upper bound of both arguments, respectively).
2) Integrate execution clearance checks at appropriate locations (primarily to handle implicit information flows, more details follow in the next section).
3) Adapt the memory interface, which is responsible to translate load/store instructions into TLM transactions, to support tainted values. To ensure compatibility with TLM transactions, our *Taint* data type provides the *to_bytes* (Line 12) and *from_bytes* (Line 18) functions that convert any *Taint* (e.g. *Taint*<uint32_t>) to and from an array of

```
 1 struct SimpleSensor : public sc_core::sc_module {
 2   tlm_utils::simple_target_socket<SimpleSensor> tsock;
 3   // memory mapped data frame
 4   std::array<Taint<uint8_t>, 64> data_frame;
 5
 6   // security tag for the generated data
 7   uint8_t data_tag = Taint::LowConf;
 8
 9   // register SystemC thread and TLM transport function
10   SC_HAS_PROCESS(SimpleSensor);
11   SimpleSensor(sc_core::sc_module_name) {
12     tsock.register_b_transport(this, &SimpleSensor::transport);
13     SC_THREAD(run);
14   }
15   void run() {
16     while (true) {
17       sc_core::wait(25, sc_core::SC_MS); // 40 times per second
18       // fill with random printable data
19       for (auto &n : data_frame) {
20         // generate data of the specified security class
21         n = Taint<uint8_t>(rand() % 96 + 128, data_tag);
22       }
23       // notify interrupt controller (IC) about new sensor data
24       IC->trigger_interrupt(2 /*IRQ NUMBER*/);
25     }
26   }
27
28   // the VP bus routes transactions to this function
29   void transport(tlm::tlm_generic_payload &trans,
           sc_core::sc_time &delay) {
30     auto addr = trans.get_address();
31     auto cmd = trans.get_command();
32     auto len = trans.get_data_length();
33     auto ptr =
34         reinterpret_cast<Taint<uint8_t>*>(trans.get_data_ptr());
35     if (addr <= 63) {
36       // access data frame
37       assert(cmd == tlm::TLM_READ_COMMAND);
38       assert((addr + len) <= data_frame.size());
39       // return last generated random data at requested address
40       memcpy((void *)ptr, &data_frame[addr],
41           sizeof(Taint<uint8_t>) * len);
42     } else {
43     if (cmd == tlm::TLM_READ_COMMAND) {
44     // the configured security class is not confidential
45       *ptr = Taint<uint8_t>(data_tag, Taint::LowConf);
46     } else if (cmd == tlm::TLM_WRITE_COMMAND) {
47       data_tag = *ptr;
48     } else {
49       assert(false && "invalid access);
50     }
51   }
52 };
```

Fig. 4. Implementation of a sensor peripheral using SystemC TLM

tainted bytes (i.e. *Taint*<uint8_t>), respectively. Casting the *Taint*<uint8_t> array into a char pointer allows to transparently embed the *Taint* data array into a TLM transaction and route it as usual through the bus. The receiving HW peripheral obtains the Taint<uint8_t> (array) pointer by casting the char data pointer of the TLM transaction back.

Besides the CPU of the VP, also some adaptations in the HW peripherals were done. Fig. 4 shows a sensor peripheral implementation (other peripherals are implemented similarly). The sensor contains a memory mapped 64 byte *data frame* (Line 4) using our custom *Taint* data type to store a tag alongside the value. To allow the sensor to send confidential or unconfidential data we add 8 bit *data_tag* register (Line 7). The sensor periodically generates new data in the SystemC *run* thread using the configuration as given by the *data_tag* (Line 19-22). By this, depending on the concrete application differently classified sensor sources can me modeled. SW read/write accesses are routed by the VP's bus via TLM transactions to the *transport* function. The TLM *generic_payload* provides the transactions data and size. Based on the transaction type, either a read or a write access is handled in the sensor peripheral.

To extend the original version of the sensor, we only had to change 6 lines of code (see highlighted lines in Fig. 4). In Lines 4 and 41 the modifications were straight forward from `uint8_t` to `Taint<uint8_t>`. Line 34 casts the transport data pointer to an array of tainted bytes instead of the original char buffer. This convention needs to be adapted in every peripheral that uses TLM transactions. In Line 21, tagged random data (the sensor's source) is generated using the *Taint* constructor with the tag as the second argument. Note, that Line 47 does not have to be changed; this is due to the overloaded conversion routine of the *Taint* class. This implicit cast to its underlying type (here *uint8_t*) requires by default a low confidentiality (*LC*) tag, throwing an error otherwise.

In summary, the integration of the DIFT engine into the VP (including peripherals) only affected 6.81% of lines of code of the original VP, of which 58.7% are type-conversions (as seen e.g. in Line 4).

*2) **Execution Clearance:*** Beside direct information flow from computational instructions and clearance checks at output interfaces, the DIFT engine has also to consider implicit information flow (confidentiality specific aspect) and protection of internal resources (integrity specific aspect). We have identified three operations in the CPU core that are relevant in this context: a) branch execution, b) instruction fetching, and c) memory access. These operations are handled by assigning each of them an execution clearance (i.e. a security class represented as tag). For example, the instruction fetch unit performs a clearance check based on its own security class *A* and the security class *B* of the fetched instruction, i.e. it requires *allowedFlow(B, A)*. For branch instructions the clearance check is performed on the branch condition and for memory access operations on the address. The execution clearance is configurable to let the engineer select the most suitable configuration (it is specified in the security policy). Furthermore, fine grained exceptions to the execution clearance are possible by using declassification (recall Section IV-A) to selectively change the security class of specific data (e.g. one specific branch condition) at runtime. Only trusted HW peripherals are allowed to do declassification to reduce the risk that an attacker exploits the declassification mechanism. We discuss the rational behind the execution clearance in the following for the three operations in the CPU core in more detail:

a) Branch Execution: Observing the control flow can implicitly reveal confidential information. Consider for example a branch *if(secret == 1) then public = 1* with a confidential condition. The control flow dependence of *public* with *secret* allows to infer information about the value of *secret* by outputting *public*. Therefore, control flow dependencies need to be considered alongside data flow dependencies by the DIFT engine. However, in the presence of an attacker that may be able to inject code (by exploiting SW bugs), their computation is challenging. Requiring an *LC* clearance on the branch condition is a safe approximation to avoid leaking sensitive information. Please note, the same clearance is used to check the interrupt/trap handler address.

b) Instruction Fetch: Similar to branches, instruction fetching/decoding can also leak sensitive information. For example consider a confidential memory word fetched by the CPU. In case the word is an illegal instruction, a jump to the (SW error) trap handler is performed. The trap handler may write to public variables, hence posing a risk of leakage. Also, the behavior of the system changes based on the fetched instruction which may provide an additional attack surface. Again, requiring an *LC* clearance on the fetched instruction is a safe approximation to avoid leaking sensitive information.

In addition, to reduce the risk of code injection by exploiting SW bugs, it makes sense to also use a *HI* clearance for instruction fetching. This prevents execution of data from external untrusted sources. However, it still cannot fully prevent code injection, since an attacker might be able to exploit bugs in the embedded SW to inject malicious code by re-using trusted code from memory.

c) Memory Access: A memory access with confidential address can also leak information. For example consider *Mem[secret] = public*. Then, the value of *secret* may be inferred by querying the memory, e.g. *public2 = Mem[i]* and check *public == public2* for $i = 0$, $i = 1$, etc. Even if the value of *Mem[secret]* is confidential too, an inference of the *secret* address is still possible by writing *Mem[0]*, *Mem[1]*, etc., to a public output interface and observe if an error is raised (due to insufficient clearance in case Mem[i] is confidential). Using an *LC* clearance on the memory address, prevents these attacks.

VI. Experimental Evaluation

We have implemented our proposed VP-based DIFT approach for early development and validation of security policies by integrating our DIFT engine into the open-source SystemC TLM RISC-V VP [29]. We evaluate our approach in three steps. First, in Section VI-A we present a case-study on developing and validating the security policy for an *Electronic Control Unit* (ECU) of a car engine immobilizer. Then, we show the effectiveness of our approach in detecting code injection (Section VI-B). Finally, we evaluate the performance overhead of our DIFT engine (Section VI-C).

A. Security Policy Evaluation: Car Engine Immobilizer

In the first experiment, we consider as case-study an ECU of a car engine immobilizer. The immobilizer holds a secret key (PIN) in memory which is used for a challenge-response protocol together with the engine's ECU for authentication. Therefore, the engine sends a challenge (random number) and the immobilizer returns a response (challenge encrypted by PIN using an AES peripheral). The engine holds the same PIN as the immobilizer and checks the response by performing the same encryption. The communication channel between the ECUs is established by reading and writing to a CAN peripheral. Please note, that in this authentication process the PIN is never exchanged on the CAN bus in plain-text.

Our goal is that the PIN is neither leaked (to prevent unauthorized access to the car) nor modified (to keep the car operational). Thus, our security policy uses IFP-3 (see Section IV-A) and classifies the key as *(HC,HI)* and use *(LC,LI)* clearance on all input and output devices (including the CAN peripheral). In addition, the AES peripheral has *(HC,HI)* clearance and performs declassification, i.e. all encrypted data has *(LC,LI)* classification so it can be sent out on the CAN bus.

By running a manually written test-suite we observed that the security policy is violated because the immobilizer can be instructed to perform a complete memory dump (including the secret key) on the UART (which exists for debugging purposes). We fixed this security vulnerability by correcting the debug function to exclude the memory region of the key.

For further evaluation purposes we modified the immobilizer SW to include common attack scenarios: 1) directly or indirectly (through an intermediate buffer or buffer overflow) write the PIN to an output interface, 2) use control flow instructions that depend on the PIN, and 3) override the PIN in memory with external data. All attack scenarios have been detected successfully.

However, further testing revealed another attack scenario that is still not covered by the security policy yet. While the current security policy prevents overwriting the PIN with external data (i.e. *LI*), it does not protect against overwriting with trusted data (i.e. *HI*). Thus, according to the security policy it is still possible to e.g. overwrite byte 2, byte 3, etc. of the PIN with byte 1. This significantly reduces the encryption entropy (all bytes in the PIN are equal) and hence enables a brute-force attack (by trying 256 possibilities) to obtain the PIN byte by byte from the encrypted response on the CAN bus. We fixed this issue by modifying the security policy to use a separate security class for each byte of the PIN, hence further reducing the risk of a security vulnerability.

B. Code Injection Protection

In the second experiment we evaluate the effectiveness of our approach in detecting code injection. Therefore, we use the Wilander-Kamkar buffer overflow attack suite [30] which has been ported for RISC-V by [9], though some attacks are not applicable (N/A) in the RISC-V environment, primarily due to differences in the calling convention [9]. Table I shows an overview of the attacks. The suite features several attack patterns that exploit buffer overflows on the stack or the Heap/BSS/Data segment (column: *Location*) to target e.g. the return address, base pointer, function pointer or longjmp buffer (column: *Target*). The buffer is either accessed directly or indirectly through a pointer (column: *Technique*). All attacks try to inject and execute a pre-defined malicious code payload which is a serious security breach and may gain the attacker complete access to the system.

TABLE I
Buffer-overflow test-suite results

Atk #	Location	Target	Technique	Result
1	Stack	Function Pointer (param)	Direct	N/A
2	Stack	Longjmp Buffer (param)	Direct	N/A
3	Stack	Return Address	Direct	Detected
4	Stack	Base Pointer	Direct	N/A
5	Stack	Function Pointer (local)	Direct	Detected
6	Stack	Longjmp Buffer	Direct	Detected
7	Heap/BSS/Data	Function Pointer	Direct	Detected
8	Heap/BSS/Data	Longjmp Buffer	Direct	N/A
9	Stack	Function Pointer (param)	Indirect	Detected
10	Stack	Longjump Buffer (param)	Indirect	Detected
11	Stack	Return Address	Indirect	Detected
12	Stack	Base Pointer	Indirect	N/A
13	Stack	Function Pointer (local)	Indirect	Detected
14	Stack	Longjmp Buffer	Indirect	Detected
15	Heap/BSS/Data	Return Address	Indirect	N/A
16	Heap/BSS/Data	Base Pointer	Indirect	N/A
17	Heap/BSS/Data	Function Pointer (local)	Indirect	Detected
18	Heap/BSS/Data	Longjmp Buffer	Indirect	N/A

To protect against code injection, we use a security policy based on IFP-2. The memory holding the program is classified as *HI* during program loading, and the instruction fetch unit in the CPU is also set to *HI* clearance, i.e. it will raise an error when fetching instructions with *LI* classification. All other information in the system (including data coming from the serial console) is classified as *LI*. Because the test-suite features a well-defined function as a representation for malicious code, we specifically classify this function as *LI* before conducting the tests. In a real world scenario, this code would be inserted

TABLE II
RESULTS ON THE PERFORMANCE OVERHEAD OF OUR APPROACH

Benchmark	#instr. exec.	LoC ASM	Sim. Time		MIPS		Ov.
			VP	VP+	VP	VP+	
qsort	430,719,182	17,052	11.6	18.3	37.1	23.5	1.6x
dhrystone	1,370,010,911	17,158	39.1	60.1	35.1	21.1	1.6x
primes	7,114,988,890	16,793	186.3	390.0	38.1	18.2	2.1x
sha512	7,578,047,617	17,862	251.6	441.5	30.1	17.1	1.8x
simple-sensor	1,393,000,060	2,970	67.6	83.0	20.6	16.7	1.2x
freertos-tasks	5,937,843,750	11,146	141.6	411.5	41.9	14.4	2.9x
immo-fixed	931,083,025	17,188	26.1	46.9	35.6	19.8	1.8x
– average –	3,536,527,633	14,309	103.4	207.3	33.2	17.0	2.0x

by external components (e.g. the terminal) and thus also have an *LI* security class. With this security policy all applicable attacks were detected which demonstrates the effectiveness of our approach in detecting code injection attacks.

C. Performance Overhead Evaluation

To evaluate the performance overhead of the DIFT engine we compare the execution times of our approach (denoted *VP+*) against the original RISC-V VP (denoted *VP*). All benchmarks are executed on a Linux machine with Fedora 29 and an Intel™ i5-8250U processor.

Table II shows the results. The first three columns report the benchmark name, the number of executed instructions (column: *#instr. exec.*) and number of assembler opcodes (column: *LoC ASM*) in the final binary (which includes linked libraries). The remaining columns compare the simulation time (in seconds) and MIPS (*Million Instructions Per Second*) for VP and VP+, and the resulting performance overhead of VP+ (column: *Ov.*). The last row summarizes the results by providing average values for all benchmarks. As benchmarks, we use *qsort* from the *newlib* C library, a standard *dhrystone* implementation, a prime number generator, the hash sum function *sha512*, a simple-sensor application that copies randomly generated data from a sensor to a UART peripheral, a FreeRTOS application SW scheduling two interleaved tasks, and the fixed car immobilizer SW (the example from the previous section).

It can be observed that VP+ is in average a factor of 2x slower (worst and best case at 2.9x and 1.2x, respectively) than the original VP on the benchmark set, which is a very reasonable performance overhead.

VII. CONCLUSION AND FUTURE WORK

In this paper, we have introduced a VP-based DIFT approach for embedded binaries taking fine-grained HW/SW interactions into account. Our approach supports a wide range of security policies which can be fully configured by the user. Moreover, since our approach leverages SystemC-based VPs security policies can be developed and validated early, i.e. before the HW is available. In addition, we utilized the benefits offered by SystemC/C++, in particular templates and operator overloading, to design a taint data type that enables a straightforward integration of our DIFT engine into the VP platform. Extensive RISC-V experiments demonstrated the effectiveness of our approach.

For future work we plan to investigate automatic test-case generation methods that consider the SW as well as the VP level (e.g. [31], [32]) and are tailored for stress-testing security policies.

REFERENCES

[1] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *ASPLOS*, 2004.

[2] D. Hedin and A. Sabelfeld, "A perspective on information-flow control," in *Software Safety and Security - Tools for Analysis and Verification*, 2012, pp. 319–347.

[3] D. E. Robling Denning, *Cryptography and Data Security*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1982.

[4] *IEEE Standard SystemC Language Reference Manual*, Std. 1666, 2011.

[5] *OSCI TLM-2.0 Language Reference Manual*, OSCI, 2009.

[6] T. De Schutter, *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press, March 2014.

[7] Automotive Working Group, *Automotive Virtual Prototyping Platform (White Paper)*, edacentrum, 2019.

[8] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI: hardware-assisted data-flow isolation," in *Security and Privacy*, 2016.

[9] C. Palmiero, G. Di Guglielmo, L. Lavagno, and L. P. Carloni, "Design and implementation of a dynamic information flow tracking architecture to secure a RISC-V core for IoT applications," in *IEEE HPEC*, 2018.

[10] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A flexible information flow architecture for software security," in *ISCA*, 2007, pp. 482–493.

[11] H. Kannan, M. Dalton, and C. Kozyrakis, "Decoupling dynamic information flow tracking with a dedicated coprocessor," in *DSN*, 2009.

[12] L. Piccolboni, G. Di Guglielmo, and L. P. Carloni, "Pagurus: Low-overhead dynamic information flow tracking on loosely coupled accelerators," *IEEE TCSDI*, 2018.

[13] J. Porquet and S. Sethumadhavan, "Whisk: An uncore architecture for dynamic information flow tracking in heterogeneous embedded socs," in *ISSS*, 2013.

[14] C. Pilato, K. Wu, S. Garg, R. Karri, and F. Regazzoni, "Tainthls: High-level synthesis for dynamic information flow tracking," *TCAD*, 2019.

[15] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner, "Register transfer level information flow tracking for provably secure hardware design," in *DATE*, 2017.

[16] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *ASPLOS*, 2009.

[17] L. C. Lam and T. Chiueh, "A general dynamic information flow tracking framework for security applications," in *ACSAC*, 2006, pp. 463–472.

[18] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu, "Lift: A low-overhead practical information flow tracking system for detecting security attacks," in *MICRO*, 2006.

[19] J. Clause, W. Li, and A. Orso, "Dytan: A generic dynamic taint analysis framework," in *ISSTA*, 2007, pp. 196–206.

[20] P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. M. Fung, "Verifying information flow properties of firmware using symbolic execution," in *DATE*, 2016.

[21] W. Yang, Y. Vizel, P. Subramanyan, A. Gupta, and S. Malik, "Lazy self-composition for security verification," in *CAV*, 2018.

[22] A. Danese, V. Bertacco, and G. Pravadelli, "Symbolic assertion mining for security validation," in *DATE*, 2018, pp. 1550–1555.

[23] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *CCS*, 2007.

[24] M. Hassan, V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Early SoC security validation by VP-based static information flow analysis," in *ICCAD*, 2017, pp. 400–407.

[25] M. Goli, M. Hassan, D. Große, and R. Drechsler, "Security validation of VP-based SoCs using dynamic information flow tracking," *it-Information Technology*, vol. 61, no. 1, pp. 45–58, 2019.

[26] D. Große and R. Drechsler, *Quality-Driven SystemC Design*. Springer, 2010.

[27] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *Journal of Computer Security*, vol. 17, no. 5, pp. 517–548, 2009.

[28] V. Herdt, D. Große, P. Pieper, and R. Drechsler, "RISC-V based virtual prototype: An extensible and configurable platform for the system-level," *JSA*, vol. 109, p. 101756, 2020.

[29] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Extensible and configurable RISC-V based virtual prototype," in *FDL*, 2018, pp. 5–16.

[30] J. Wilander and M. Kamkar, "A comparison of publicly available tools for dynamic buffer overflow prevention," in *NDSS*, 2003.

[31] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Early concolic testing of embedded binaries with virtual prototypes: A RISC-V case study," in *DAC*, 2019, pp. 188:1–188:6.

[32] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Verifying instruction set simulators using coverage-guided fuzzing," in *DATE*, 2019, pp. 360–365.