Ensuring Safety and Reliability of IP-based System Design – A Container Approach

Arun Chandrasekharan, Kenneth Schmitz, Ulrich Kühne and Rolf Drechsler
Institute of Computer Architecture
University of Bremen, Germany
{arun,kenneth,ulrichk,drechsle}@cs.uni-bremen.de

***Abstract*—The application of built-to-order embedded hardware designs in safety critical systems requires a high design quality and robustness during operation. Flawless execution of the involved software can be compromised by malfunctioning hardware components or by software-induced errors. Furthermore, *intellectual property* (IP) tends to become unavoidable in modern hardware designs. Any unexpected behavior of IP components may cause unrecoverable system errors. In order to construct correct and safe systems from unverified and potentially malicious components, we propose a system integration approach which encapsulates IP blocks in verifiable container modules. The synthesis of these container modules is driven by a *domain specific language* (DSL) augmented with *sequential extended regular expressions* (SEREs). The approach is demonstrated by showing the synthesis of an effective countermeasure against software-induced memory disturbance errors.**

***Keywords*—Container-Verification, Safe IP Integration, Model-to-HDL Synthesis, Safety**

I. Introduction

In the last decades, the continuous improvements in semiconductor fabrication have made it possible to produce integrated circuits with billions of transistors on a single chip. This has led to the development of tremendously complex *systems on chip* (SoCs). In the wake of these technological advancements, smaller embedded devices have become cheaper and more energy efficient. At the same time, new and also more complex features are finding their way into these embedded devices. Driven by rapidly emerging markets, mobile and network connectivity, multimedia, and sensory capabilities are becoming common functionality in such devices.

With this burst of new features, embedded systems have become a part of our everyday life, but they are also commonly used in safety critical domains like transportation systems and medical equipment. For such applications, correctness and reliability are crucial to prevent system failure, which could have catastrophic consequences. Testing and verification already account for more than half of the development costs of embedded systems. With a growing number of components in an SoC system, there are also more potential sources of design bugs. Since the reuse of IP blocks is crucial to keep time-to-market constraints, it is often impossible to ensure the correctness of all components in a newly designed system.

Even worse, with increasing connectivity over different interfaces, there is also a growing potential of external attacks against a system. The miniaturization of semiconductor fabrication processes has lead to various positive effects such as reduced power consumption, smaller chip-area, lower fabrication costs and higher operational clock frequencies. However, these advances have introduced side effects, which can be exploited by an attacker to compromise the integrity of a system. As an example, a *Rowhammer* attack [1] can alter the content of a memory cell – for example in order to manipulate access rights or priorities – by a series of seemingly harmless read operations. By injecting malicious software, attacks can not only come from the outside, but also from processor components *within* a system [2].

Summarizing these developments, we are faced with the challenge to build correct systems with potentially incorrect, malicious or vulnerable components. As a consequence, several works have tried to address these issues by incorporating security and protection mechanisms in the architecture of an SoC, see e.g. [3], [4], [5], [6]. Most of the existing work requires a complete redesign of the target system, which contradicts the assumption that we are often dealing with IP blocks that cannot be changed.

In this paper, we propose an alternative view on the problem of safe IP integration: By encapsulating the suspected or vulnerable blocks within a container module, the overall system is protected against erroneous or malicious behavior. The container modules are generated automatically based on a DSL that can be used to describe the expected behavior and specific protection measures against known attacks. In this way, a provably correct behavior of the container can be achieved without touching the contained IP block. This approach is based on the concept presented in [7], where container modules have been used in a different context, in order to create a robust bus interface.

A central component in the proposed approach are *reactive monitors*. By decoupling the interface of a contained IP from the rest of the system, a reactive monitor can take over the control of the interface signals whenever it detects erroneous or malicious behavior. The method has been implemented within the *Eclipse* integrated development environment, in which system integration engineer can conveniently select an IP in a complex design to be protected, specify the protection scheme, and automatically generate a container. To further automate and facilitate the design flow, the framework will generate a simulation model along with the design files and will create the default templates for the well known and standard threats and error-prone use cases. We demonstrate the tool flow in a case study, protecting a memory block against the Rowhammer attacks.

The paper is structured as follows: After discussing related work in Section II, the methodology is introduced in Section III. In Section IV we briefly discuss the implementation of the proposed flow. The case study is presented in Section V, before concluding the paper in Section VI.

II. Related Work

In [3] and [4], Porquet et al. propose an enhanced memory management in a SoC in order to create secure *compartments* and allow the co-hosting of multiple applications without interference. While the idea of compartments is similar to our containers, their approach requires a specific SoC architecture. Furthermore, like the security measures presented in [5], [6], it assumes the functional correctness of the involved modules.

On a lower level of abstraction, *robustness* or *resilience* is the ability of a system to tolerate faults, either permanent defects or soft errors caused by radiation events [8]. Techniques have been proposed to asses [9] and improve [10], [11] robustness. However, these methods target only a very specific and very low level fault model, while we aim for a more general framework to create correct systems at the design level.

Methods to create circuits which are correct by construction are presented e.g. in [12], [13], [14]. Starting with a specification in a temporal logic like LTL, an automaton is synthesized that fulfills the spec. However, the involved algorithms have a high complexity. While we do not aim to create full systems from scratch, we plan to investigate the integration of these techniques with our approach, e.g. to automatically generate glue logic between different IP blocks. The construction of *monitors* or *checkers* from assertions – as presented e.g. in [15] – forms a basis of our automatic synthesis flow, that is further enhanced by allowing the specification of *actions* in order to counter attacks.

Recently, there has been a growing interest in techniques to reverse engineer large gate-level netlists, without any structural knowledge, see e.g. [16], [17], [18], [19]. This is motivated by the concern that register transfer level designs could be tampered with before being manufactured in order to introduce hardware Trojans. By matching the netlist to the register transfer level design, changes can be detected. Although the methods have improved recently, they still cannot be applied to larger designs. Using our container approach, we circumvent the complex reverse-engineering task by providing an external protection mechanism that detects threats at run-time.

III. Methodology

A. Safe IP Integration

As IP components are widely used for the implementation of complex systems, the used IP cores need to be correct and reliable, as these components can influence the entire system adversely. Especially in the security domain and in safety critical applications this circumstance is a primary concern [2]. However, the complete verification is near to impractical when respecting reasonable time-to-market constraints. Since these modules are usually provided as black boxes for which no inspectable source files are available, the system designer has to rely on the proper realization of the desired functionality. Although a lot of effort is spent on the careful verification of hardware designs, glitches and flaws may remain undiscovered. The full verification or simulation in the system design phase of the project is infeasible. As an alternative paradigm, reused IP cores may be encapsulated inside a wrapping container, which provides means to protect the overall system from erroneous or intentionally malicious behavior. This makes it possible to integrate components that are a potential threat to the system stability without compromising the overall reliability and with a high level of confidence.

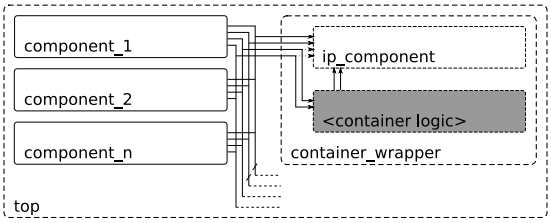

Fig. 1. Integration

Figure 1 shows an example application of the basic methodology. Whenever an IP component is inserted into an existing system, this component is placed inside a container, which implements the required protection mechanisms. The internals of the container are transparent to the remaining system modules. Thus, the surrounding system needs no modification and the integration overhead is kept at the lowest possible level. Depending on the complexity of protective countermeasures of the wrapping container module, the size of the *container logic* may vary significantly. It needs to contain the necessary logic to implement the desired protection. In the proposed design flow, the logic is synthesized automatically from a dedicated specification language, allowing a rapid integration of new components.

The complexity of the container logic – and of its automatic synthesis – depends on the threat model and the properties that one wants to enforce on the contained IP block. As introduced in [7], we can distinguish between three general use cases with increasing complexity:

1) *Monitoring* the incoming or outgoing communication in order to recognize erroneous or malicious behavior.
2) *Filtering & altering* incoming or outgoing data in order to block attacks or to fix minor bugs and glitches.
3) *Actively manipulating* the behavior of the contained module in order to bring it back to a safe state. This increases the difficulty of exploiting the module.

While the application in [7] was restricted to the monitoring and fixing of bus protocols glitches, in this work we are aiming to extend the container approach to more complex and more general use cases. The central concept for more generic and more powerful applications is the use of *reactive monitors*, which will be described in the following.

B. Reactive Monitors

The detection of erroneous or malicious behavior requires observation of the involved interface signals. Large and highly

integrated systems strongly rely on high-speed and high-bandwidth bus or point-to-point communication mechanisms. Whenever a single component carries out illegal commands to a shared bus, the entire system may be compromised. Thus, the container approach keeps track of the communication of the contained instance in order to guarantee error-free operation.

Run-time monitoring is a common technique applied in hardware-design. A monitor (or checker) is a component that is able to detect the violation of some kind of property. The synthesis of such monitors is well-understood (see e.g. [15]). Starting from a temporal logic specification such as *linear temporal logic* (LTL) or *property specification language* (PSL), the formula of interest is translated to an automaton that recognizes violating prefixes of the formula. By synthesizing the automaton and wiring it up with the interface signals of the target design, it is possible to create a monitor circuit in hardware that observes the design at run-time. In case of a property violation, the checker circuit may rise a signal to indicate the faulty behavior. This technique is very useful for the analysis and diagnosis of failure scenarios.

However, since our goal is to guarantee error-free operation and a seamless integration of the contained modules into their system environment, passive monitoring is not enough. Rather, we empower the monitors to intervene in case of a property violation and fix the occurring error if possible, making them *reactive monitors*. Instead of relying on a separate trusted master component that takes over control in case of an error, the actions to be taken can be described in a single specification. For this purpose, we make use of *sequential extended regular expressions* (SEREs), augmented with assignment and storage statements. The synthesized reactive monitors are placed at the interface of the contained module. In this way, in case of an error, they can block the communication, isolate the module or change the value of specific signals to prevent system failure.

In the following, the specification language will be introduced, before we discuss the synthesis of the reactive monitors.

C. Domain Specific Specification Language

The extension adds the capability to implement active countermeasures when a given sequence is observed. We introduce the SERE syntax to our methodology. In safe systems, many attacks (e.g. [1]) rely on penetration attacks [20]. Without the notion of sequences, it is impractical to anticipate an upcoming attack to the system.

As a result we created an operation-skeleton which contains the necessary keywords to express even complex attack or protection sequences. As a subset of the PSL, SERE is well-suited in this field of application. However, at the current state of development, the supported subset is limited. Store **$** and assignment **=** operations have been added. Negations, comparisons and arithmetic operators are implemented. Logic operators can also be used. Consider the given sequence $\{sig\_a; sig\_a; sig\_a\}$. It represents the logical value 1 of signal sig_a for three consecutive cycles. SERE allows to simplify this sequence as follows: $sig\_a;[*3]$. From our given attack-scenario, we extract a snippet from the application scenario of a detection sequence to point out the advantages.

$$\{(\text{cmd\_en} \;\&\&\; \text{cmd\_instr}); \text{rd\_en}; (\text{cmd\_en} \;\&\&\; \text{cmd\_instr}); (\text{cmd\_byte\_addr} == (\text{address} + 2)) \;\&\&\; \text{rd\_en};\}[*10]; \tag{1}$$

Here, a sequence of three consecutive cycles is shown, where signals from the design are observed. If this sequence is repeated ten times, the detection is triggered. To cope with more complex problems, we added a SERE related syntax, which also allows the system integration engineer to store and assign values. This way, it is possible to store a signal's current value to a temporary buffer to reuse it later or assign a static value to a given signal output.

$$\{\$\text{address} = \text{cmd\_byte\_addr}; \text{cmd\_byte\_addr} = \text{address} + 1; \text{wr\_en} = 1;\}[*10]; \tag{2}$$

The above sequence contains an assignment statement to a temporary register for the later use. This way it is possible to compare data or address values at different cycles of operation. The following operation stub can be used to express complex sequences to observe erroneous behavior and engage appropriate countermeasures.

```
operation NAME      // operation name
    detect:         // detection sequence
    correct:        // correction sequence
    reset:          // reset trigger
    sequence:       // generic subsequences

    system_clock:   // clock event
    system_reset:   // reset event
end
```

The semantic of the keywords is explained in the following:

operation NAME indicates the start of an operation. The operation represents a single cycle of at least *detect* and *reset*. It can contain the following keywords as shown above. A single operation is finalized by the keyword **end**.

detect: This keyword is followed by a SERE sequence. This sequence contains all of the previously mentioned operations except the assignment. For example, address or data buses can be observed. Equation (2) provides an impression how this part of the operation-skeleton can be used to store temporary values for later use.

correct: The correct statement contains the countermeasures. In this part of the operation signals can be modified and the system can be protected. Here it is possible to reassign a temporarily stored value to a certain signal line.

reset: As a state machine will be the result after synthesizing these operations (properties), it requires a condition under which the automaton will be reset to its initial state. Thus, reset indicates the SERE sequence to trigger this transition. In case of penetration attacks, it is possible to reset the automaton after a cool-down period has expired.

sequence: The definition of subsequences can be considered as syntactical sugar. It provides the option to reduce the complexity of large verification sequences by encapsulating inherently complete subsequences by a substituting variable. An example for this is given below.

```
sequence : op_READ = {
  !ack_i,
  !we_o;
};
```

PSL is a very rich language, and SERE are an essential part of it. The full standard includes linear and branching time properties, allowing the specification of complex safety and liveness properties. However, in the context of dynamic verification and run-time monitoring, it is necessary that properties can be invalidated and sequences matched by a finite prefix, which poses a natural restriction on the kind of properties that can be treated. As an example, it is not useful to state unbounded liveness properties or infinite sequences, since they will never fire in a finite simulation run. While the PSL standard [21] defines a simple subset by syntactic restrictions, [22] gives a semantic characterization. Obviously. these restrictions also apply for possible extensions of our specification language.

D. Synthesis

We designed the synthesized code as multiple interacting state machines from the statements which represent the given sequences functionally. Each statement (*detect, correct, etc.*) inside a single operation will yield an additional state machine.

Each SERE statement yields a new state within the corresponding state machine. The previously given sequence $\{sig\_a; sig\_a; sig\_a\}$ requires three discrete states for the detection of the given sequence. The occurrence of sig_a will trigger the connected transition. Consequently it is possible to create state machines which can detect complex signal sequences. To demonstrate the overall synthesis, consider the following example. The Wishbone bus [23] was developed to interconnect a huge variety of IP-components within a SoC. The underlying bus protocol must be implemented by each participant carefully. It is essential that the address lines remains constant during an entire read operation, since otherwise an unpredictable result may occur. Therefore we need to synthesize a stability property which maintains the stability of the `adr_o` signals during the entire read operation.

The utilization of SEREs allows a fine grained definition of sequences. In the following example, we define a subsequence `op_READ` which represents the read operation on the bus. Consequently all read operations will be detected. Initially the address is latched internally for later assignment in the occurrence of the mentioned bug. Combinatorial logic then is synthesized, which applies the latched value if needed.

```
operation READ
    detect:  det = {op_READ, $addr = adr_o;
                    !(adr_o == addr);};
    correct: stabilize = {adr_o = addr};

    sequence : op_READ = {!ack_i, !we_o;};
end
```

The given property is synthesized as shown in Figure 2.

As a first step in synthesis, each sequence is parsed and an *abstract syntax tree* (AST) is built with the parsed information. For each state and input only one transition is allowed in the DSL. Hence this is identical to the state diagram shown in Figure 2 and the required *finite state machines* (FSMs) are

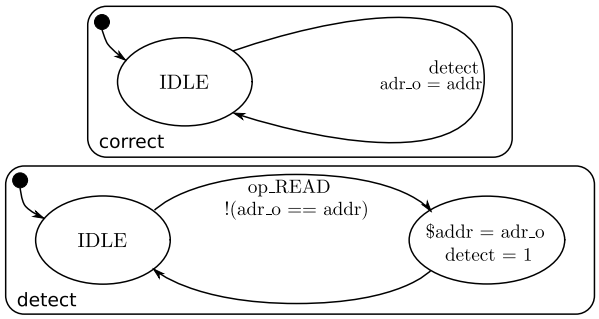

Fig. 2. Resulting synthesized state machine

synthesized directly from the state transition information. Each FSM provides an output once all the states are traversed and it is internally used to synchronize the various FSMs it is related to. The *detect* state machine is merely an observer whose output is one only, when all the states are traversed. The *correct* state machine is triggered when the *detect* is fired. It provides an implicit output to signal when the correction action is completed. Other sequences are also translated in a similar way. A *bypass* FSM is also synthesized as part of the container logic. The *bypass* FSM is triggered once the *detect* FSM is fired and maintains its output high until the *correct* FSM is fired. This bypass signal is used to effectively multiplex the various output signals when the correction action is in place. Every storage element used is made globally visible to all the FSMs. Apart from the sequence grammar, the design information is also needed to effectively synthesize the FSMs. We use ZamiaCAD [24] - a static HDL analysis plug-in for Eclipse to get this information. The exact implementation details and how each plug-in is integrated is explained in the next section.

IV. Implementation

We implemented the container module generator as a plug-in for Eclipse. Eclipse provides an easy to use plug-in development mechanism for creating customized add-ons which can utilize the existing framework from Eclipse itself. Additionally, the ZamiaCAD plug-in provides the capability of reading HDL code into its AST, where interface-descriptions in terms of ports and their direction can be extracted easily. This design information forms the input to the plug-in for the generation of the container module.

In [7], an operation skeleton which was able to keep track of the Wishbone bus protocol was modeled. For the proposed approach the underlying grammar needs several modifications. The parser for this extended grammar is created within Xtext [25]. Xtext is a framework within Eclipse which allows the rapid development of domain specific languages. This combination of add-ons provides the basic infrastructure.

The proposed architecture for the tool flow is shown in Figure 3. The initial HDL design is fed to the ZamiaCAD framework. The internal parser and data structures create the representation for the design. Interface methods provide a convenient way to access the necessary information. From the DSL-grammar file, a parser is derived. This derived parser accesses the HDL-port information from its attached methods for the container-generation methodology. In the next step,

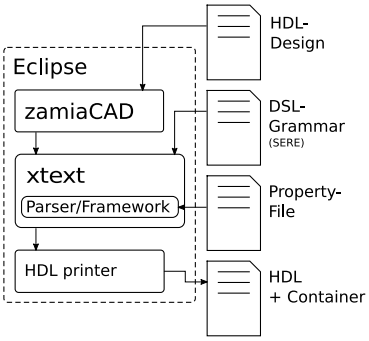

Fig. 3. Design flow

a property file is fed to the DSL-parser. Within the HDL-printer, the new code is generated. As a first step the container interfaces are created to maintain the communication with the surrounding system. Within the container the original IP module is instantiated and the additional container logic is generated. The resulting design will finally be written to the HDL output file.

V. Application

A. Rowhammer Attack

In this section we present the application example for our new approach. Recently a lot of effort has been put into the investigation on the so called *Rowhammer* effect. This effect can cause memory disturbance errors induced by interleaving read or write access to certain memory cells. This effect was fostered due to decreasing fabrication size of memory cells. In [1], the authors demonstrated that this parasitic effect within modern state of the art DDR3 memory technology can cause bit errors which then can be maliciously exploited. It has been recently demonstrated that an escalation of privileges or the escape from a sandbox environment can be caused as described in [26]. Given this circumstance, the system security can be compromised on the one hand, on the other hand mission-critical data could also be manipulated due to this known issue.

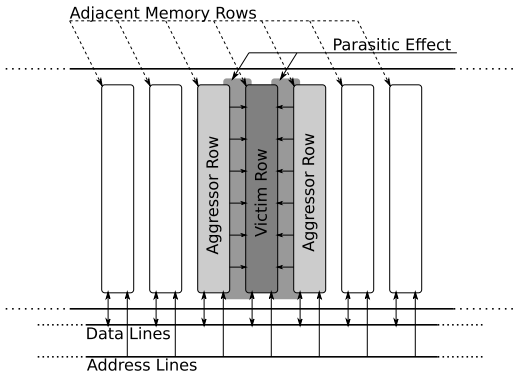

Fig. 4. Parasitic Effects

Different parasitic effects within the DRAM cell [27], [28] are known to facilitate this disturbance effect. Figure 4 shows a simplified version of the setup which can be found inside a DRAM chip. Typically, a large amount of consecutive read operations from a specific memory location would easily be caught by the computers memory hierarchy. However, the `clflush()` operation will cause the *central processing unit* (CPU) to directly access the main memory, regarding the cache content as invalid (simplified).

> *Intel 64 and IA-32 Architectures Software Developers Manual*: Invalidates the cache line that contains the linear address specified with the source operand from all levels of the processor cache hierarchy (data and instruction). The invalidation is broadcast throughout the cache coherence domain [29].

Due to the internal architecture, read operations may induce crosstalk on signal lines within the DRAM cell grid or leak charge from the capacitors which contain the data value. The effect could be observed with an augmented HDL simulation model. The following wave-trace illustrates the basic idea of the attack.

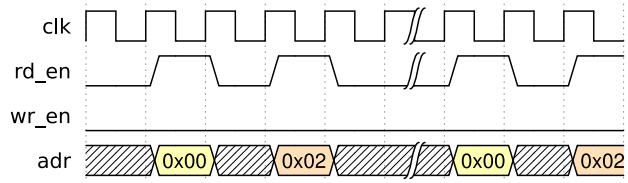

Fig. 5. Attack scheme

Figure 5 shows the interleaving access to the two aggressor rows (at address `0x00` and `0x02`), which then can introduce the disturbance errors in the victim row (at address `0x01`). This wave-trace correlates with Figure 4 as two adjacent memory rows induce parasitic effects on the row in between. Hence it is possible that the alternating read operation on the aggressor rows will yield a bit-flip in the victim row's data.

B. Countermeasures

We propose a wrapping container module which effectively can prevent the attack. As a result of the `clflush()` command, the cache-hierarchy is bypassed and the retention time of the memory cells can actively be reduced. The following wave-trace will show a simple and yet effective way to prevent the error from occurring.

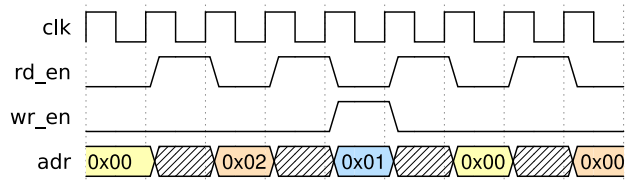

Fig. 6. Countermeasures

In Figure 6 a single write operation on the victim-row within the consecutive read commands on the aggressor-rows is dispatched. It would also suffice to read from the cell once, as it will also refreshes the cell's content, too. Consequently, the charge of the retention capacitor will be refreshed. Depending on several ambient parameters, the retention time can strongly differ. In the data sheet for the MT41J256M4 DDR3 SDRAM module by Micron Technology, Inc. [30] a refresh cycle is required as follows:

64ms, 8192 cycle refresh at 0°C to 85°C
32ms, 8192 cycle refresh at 85°C to 95°C

Thus the observation period and the interval of countermeasures needs to be evaluated, which finally can be expressed at the end of a given sequence in terms of the repetition.

Every counteraction will require a short stall of the connected system, as it is necessary to execute the access to the victim-row. The results are two conflicting objectives: The execution speed of the system might be compromised when the attack is detected. As a result, the validity of the stored data will not be compromised. This drawback has to be taken into account. After the generation process of the encapsulating container, the overall setup is shown in Figure 7.

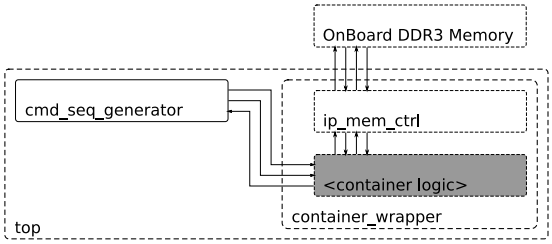

Fig. 7. Application

C. Evaluation

We demonstrate the effectiveness of our approach by simulation, using the memory controller IP from Xilinx [31]. The memory controller IP is interfaced to a Micron DDR3 SDRAM memory [30] and the framework is used for *Rowhammer* protection. Since the original Micron DDR3 simulation memory models is ignorant of *Rowhammer* attacks, possible attack scenarios are augmented as a SystemC model to the DDR3 memory models. The SystemC *Rowhammering* module will modify the data to and from the memory such that it will introduce arbitrary bit-flips when the two aggressor-rows adjacent to a victim-row are accessed, as explained previously. Figure 7 shows the complete system after container generation. The memory controller IP is instantiated inside the container wrapper module alongside with the container logic. The original signal lines are partially intersected by this logic to observe the command sequences. Finally, when a suspicious read/write sequence is detected, the container logic will stall the IP for the duration of a single read or write command to refresh the cells' data retention time.

A simplified DSL snippet for the Rowhammer attack is given below.

```
operation scenario_1

    detect:attack = {
        attack_seq; //attack case-1
        // complete atomic-operations
        rd_finish || wr_finish;
    };

    correct:correct = {
        // read & write data at the victim addr
        (read_sequence, addr_line = addr1 + 1);
        $victim_data1 = data_line;
        addr_line = addr1 + 1;
        (write_sequence,
         data_line = victim_data1);
    };

    // repeat 200 times to detect as an attack
    sequence: attack_seq = {
        (read_sequence, $addr1 = addr_line);
        read_sequence, addr_line = addr1 + 2;
    } [*200];
end
```

Simulation is carried out using Mentor QuestaSim software with mixed language support to incorporate the SystemC model. The final verification flow is generated by the framework automatically.

We synthesized the system with prevention schemes for two possible Rowhammer attack scenarios. Xilinx ISE software was used targeting the Spartan-6 FPGA [32]. The overhead for the container logic is found to be an extra 100 flip flops and 304 extra *look-up tables* (LUTs). It is important to note that this container overhead is unrelated to the complexity and size of the underlying IP, rather it depends only on the interface signals and the protection scheme. Like for any known protection mechanism, there is a trade-off between the overhead in terms of area and delay on the one hand side, and the objectives of the protection scheme, such as safety, security, and reliability. By reusing the encapsulated IP as is, we keep the design overhead small.

VI. Conclusion

In modern SoC design, IP reuse is inevitable in order to match time-to-market constraints. Often, IP components cannot be verified, since there is no source code available. In order to create correct and reliable systems from untrusted components, this paper introduces an approach to encapsulate IP blocks in safe container modules. The method relies on reactive monitors, which are located at the interfaces of the contained module. Based on a convenient specification language, the container logic is synthesized automatically, enabling a rapid system integration. The method has been demonstrated in a case study, where a memory block is encapsulated in order to counter a memory disturbance attack.

In the future, we would like to explore approaches for property-based synthesis in order to raise the level of automation, reduce the design effort even more and generalize our approach for application in wider fields. In terms of concrete scenarios, we will investigate in the area of vulnerable and errata instruction replacement using the container approach for modern microprocessor architectures.

Acknowledgment

This work was supported by the Graduate School SyDe (funded by the German Excellence Initiative within the University of Bremen's institutional strategy), by the German Research Foundation (DFG) within the Reinhart Koselleck project under grant no. DR 287/23-1, and by the German Academic Exchange Service (DAAD).

REFERENCES

[1] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, June 2014, pp. 361–372.

[2] S. Bhasin, J.-L. Danger, S. Guilley, X. Ngo, and L. Sauvage, "Hardware trojan horses in cryptographic IP cores," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, Aug 2013, pp. 15–29.

[3] J. Porquet, C. Schwarz, and A. Greiner, "Multi-compartment: a new architecture for secure co-hosting on SoC," in *International Symposium on System-on-Chip (SOC)*, 2009, pp. 124–127.

[4] J. Porquet, A. Greiner, and C. Schwarz, "NoC-MPU: a secure architecture for flexible co-hosting on shared memory MPSoCs," in *Design, Automation & Test in Europe (DATE)*, 2011, pp. 1–4.

[5] J. Sepúlveda, G. Gogniat, R. Pires, W. J. Chau, and M. J. Strum, "Dynamic NoC-based architecture for MPSoC security implementation," in *Symposium on Integrated Circuits and Systems Design (SBCCI)*, 2011, pp. 197–202.

[6] J. Sepúlveda, G. Gogniat, R. Pires, W. Chau, and M. Strum, "Security-enhanced 3D communication structure for dynamic 3D-MPSoCs protection," in *Symposium on Integrated Circuits and Systems Design (SBCCI)*, 2013, pp. 1–6.

[7] R. Drechsler and U. Kühne, "Safe IP integration using container modules," in *International Symposium on Electronic System Design (ISED)*, Dec 2014, pp. 1–4.

[8] S. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *Micro, IEEE*, vol. 25, no. 6, pp. 10–16, Nov 2005.

[9] S. Frehse, G. Fey, E. Arbel, K. Yorav, and R. Drechsler, "Complete and effective robustness checking by means of interpolation," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2012, pp. 82–90.

[10] S. A. Seshia, W. Li, and S. Mitra, "Verification-guided soft error resilience," in *Design, Automation & Test in Europe (DATE)*, 2007, pp. 1442–1447.

[11] S. Krishnaswamy, S. Plaza, I. L. Markov, and J. P. Hayes, "Enhancing design robustness with reliability-aware resynthesis and logic simulation," in *International Conference on Computer-Aided Design (ICCAD)*, 2007, pp. 149–154.

[12] N. Piterman, A. Pnueli, and Y. Saar, "Synthesis of reactive(1) designs," in *Verification, Model Checking, and Abstract Interpretation*, ser. LNCS. Springer, 2006, vol. 3855, pp. 364–380.

[13] R. Ehlers, R. Könighofer, and G. Hofferek, "Symbolically synthesizing small circuits," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2012, pp. 91–100.

[14] K. Morin-Allory, F. N. Javaheri, and D. Borrione, "Fast prototyping from assertions: A pragmatic approach," in *Formal Methods and Models for Codesign (MEMOCODE)*, 2013, pp. 23–32.

[15] R. Drechsler, "Synthesizing checkers for on-line verification of system-on-chip designs," in *Circuits and Systems, 2003. ISCAS '03. Proceedings of the 2003 International Symposium on*, vol. 4, May 2003, pp. IV–748–IV–751 vol.4.

[16] W. Li, Z. Wasson, and S. A. Seshia, "Reverse engineering circuits using behavioral pattern mining," in *International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2012, pp. 83–88.

[17] W. Li, A. Gascon, P. Subramanyan, W. Y. Tan, A. Tiwari, S. Malik, N. Shankar, and S. A. Seshia, "WordRev: Finding word-level structures in a sea of bit-level gates," in *International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2013, pp. 67–74.

[18] P. Subramanyan, N. Tsiskaridze, W. Li, A. Gascon, W. Y. Tan, A. Tiwari, N. Shankar, S. Seshia, and S. Malik, "Reverse engineering digital circuits using structural and functional analyses," *IEEE Transactions on Emerging Topics in Computing*, 2014.

[19] B. Sterin, M. Soeken, R. Drechsler, and R. K. Brayton, "Simulation graphs for reverse engineering," in *Formal Methods in Computer Aided Design (FMCAD)*, 2015.

[20] D. Geer and J. Harthorne, "Penetration testing: A duet," in *Computer Security Applications Conference*, 2002, pp. 185–195.

[21] Accellera, "Property specification language reference manual version 1.1," 2004. [Online]. Available: http://www.eda.org/vfv/docs/PSL-v1.1.pdf

[22] S. Ben-David, D. Fisman, and S. Ruah, "The safety simple subset," in *Hardware and Software, Verification and Testing*. Springer, 2006, pp. 14–29.

[23] OpenCores, "WISHBONE system-on-chip (SoC) interconnection architecture for portable IP cores (rev. B4)," 2010.

[24] A. Tsepurov, G. Bartsch, R. Dorsch, M. Jenihhin, J. Raik, and V. Tihhomirov, "A scalable model based RTL framework ZamiaCAD for static analysis," in *International Conference on Very Large Scale Integration (VLSI-SoC)*, 2012, pp. 171–176.

[25] M. Eysholdt and H. Behrens, "Xtext: Implement your language faster than the quick and dirty way," in *International Conference on Object Oriented Programming Systems Languages and Applications*, 2010, pp. 307–309.

[26] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in javascript," *CoRR*, vol. abs/1507.06955, 2015. [Online]. Available: http://arxiv.org/abs/1507.06955

[27] M. Redeker, B. Cockburn, and D. Elliott, "An investigation into crosstalk noise in dram structures," in *International Workshop on Memory Technology, Design and Testing*, 2002, pp. 123–129.

[28] D.-S. Min, D.-I. Seo, J. You, S. Cho, D. Chin, and Y. Park, "Wordline coupling noise reduction techniques for scaled DRAMs," in *Symposium on VLSI Circuits*, 1990, pp. 81–82.

[29] Intel Corporation, "Intel 64 and ia-32 architectures software developers manual," 2011. [Online]. Available: http://www.intel.com/Assets/en_US/PDF/manual/253666.pdf

[30] Micron Technology, Inc, "Micron 1Gb: x16 DDR3 SDRAM datasheet."

[31] Xilinx Inc, "Spartan-6 FPGA memory controller user guide," 2009.

[32] ——, "Spartan SP695 FPGA product brief," 2009.