Verifying Linearizability of Intel® Software Guard Extensions

Rebekah Leslie-Hurd[1(✉)], Dror Caspi[1], and Matthew Fernandez[2]

[1] Intel Corporation, Hillsboro, USA
{rebekah.leslie-hurd,dror.caspi}@intel.com
[2] NICTA and UNSW, Sydney, Australia
matthew.fernandez@nicta.com.au

Abstract. Intel® Software Guard Extensions (SGX) is a collection of CPU instructions that enable an application to create secure containers that are inaccessible to untrusted entities, including the operating system and other low-level software. Establishing that the design of these instructions provides security is critical to the success of the feature, however, SGX introduces complex concurrent interactions between the instructions and the shared hardware state used to enforce security, rendering traditional approaches to validation insufficient. In this paper, we introduce Accordion, a domain specific language and compiler for automatically verifying linearizability via model checking. The compiler determines an appropriate linearization point for each instruction, computes the required linearizability assertions, and supports experimentation with a variety of model configurations across multiple model checking tools. We show that this approach to verifying linearizability works well for validating SGX and that the compiler provides improved usability over encoding the problem in a model checker directly.

1 Introduction

When a programmer writes code to manipulate a computer, they have a mental model of the machine, involving a small set of registers with processors executing assembly instructions atomically. The reality of a modern multiprocessor is significantly more complex. The exposed registers are a small component of the internal processor state and execution of a single assembly instruction is not necessarily atomic with respect to other processors. This internal concurrency is particularly complex in the new Intel® Software Guard Extensions (SGX) [17], which introduce security critical internal processor state that is shared between privileged and user-mode instructions.

SGX is a collection of CPU instructions that enable an application to create secure containers within the application address space. These secure containers, called enclaves, provide strong integrity and confidentiality guarantees for the

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

D. Kroening and C.S. Păsăreanu (Eds.): CAV 2015, Part II, LNCS 9207, pp. 144–160, 2015.
DOI: 10.1007/978-3-319-21668-3_9

code and data pages that reside inside the enclave. Once placed in an enclave, memory pages are inaccessible to untrusted entities, including the operating system and other low-level software. SGX allows programmers to isolate the security-critical portion of their application, for example, to harden applications against vulnerabilities [15] or to protect their computation in an untrusted environment like the cloud [1].

To maximize compatibility with the existing software ecosystem, the SGX instructions work in tandem with the operating system (OS), trusting the OS to manage the system resources associated with enclave pages, such as page table mappings, but verifying that the OS never breaks the confidentiality and integrity guarantees of SGX. To this end, SGX tracks metadata for each enclave page to ensure that every access is secure. Accesses to this data structure, which is shared across all logical processors, must be appropriately synchronized to maintain security, while still maximizing parallelism for performance.

Finding the appropriate line between security and performance has been a particularly difficult aspect of the SGX architecture design and was a source of bugs that could have been devastating for the feature had they not been found soon enough. Applying formal verification techniques early in the design process enabled us to find pernicious concurrency bugs and to increase our confidence that we were not overlooking a critical error in the design. Though formal verification is commonly used at Intel® for arithmetic and protocol validation, SGX has more in common with software algorithms where multiple threads access a shared data structure and is not a natural fit for the hardware verification tools and methodologies that are currently in place.

Identifying this similarity to software algorithms led us to linearizability as a correctness condition. Linearizability [14] is a classic approach to reasoning about concurrent accesses to a shared data structure. A system is linearizable if each operation (in our case, instruction) appears to take effect atomically at some moment in time between its invocation and response, called its linearization point. In a linearizable system, we cannot observe the difference between a sequentialized trace where each instruction executes atomically at its linearization point and a real trace that arises in the concurrently executing system.

An important consequence of linearizability is that we can reason about operations on linearizable concurrent data structures as if they were atomic. As such, we can divide our verification challenge into two tasks: first, to prove that the SGX instructions uphold the desired security guarantees in a sequential (or single threaded) setting; and second, to prove that the system is linearizable. We have verified the sequential correctness of the instructions using DVF [13], but in this paper we focus on the second task, proving that SGX is linearizable.

We employ a standard technique for model checking linearizability [9,23] using a domain-specific heuristic for placing linearization points. Scalability presents a major challenge in our setting—there are 22 instructions that share the concurrent data structure, some of which contain as many as 50 interleaving points—and design changes during the early stages of development are frequent. The primary contribution of this paper is a domain specific language and compiler that supports automatic linearizability checking while providing

fine-grained control over the generated model to improve scalability without the overhead of creating multiple models by hand. A secondary contribution of the paper is a demonstration of our approach on the industrial case study of SGX. We first describe how to prove that SGX is linearizable directly using model checking and then show how this process is improved by the use of Accordion. With this approach, we identified previously undiscovered concurrency bugs in a design that had already been intensively reviewed.

The remainder of the paper is organized as follows. In Sect. 2, we give an overview of the internal hardware data structures used by SGX, as well as the SGX instructions, to provide a basis for understanding the examples in the later sections. Section 3 describes our formal model of SGX in the iPave model checker and illustrates an architecture bug that was caught as a violation of linearizability. Section 4 introduces Accordion, our domain specific language and compiler for automatically proving linearizability. The remaining sections discuss related work (Sect. 5) and provide a summary (Sect. 6).

2 SGX Overview

SGX defines new processor internal state, outlined in Sect. 2.1, and a collection of instructions for creating, executing, and manipulating enclaves, covered in Sect. 2.2. In this paper, we focus on the instructions and processor state that directly affect the integrity and confidentiality guarantees of SGX, and thus, are particularly interesting targets of our linearizability analysis. For a complete overview of SGX, see the Programmer's Reference Manual [17].

2.1 Enclave Page Cache

The Enclave Page Cache (EPC) is a protected area of memory used to store enclave code and data as well as some additional management structures introduced by SGX. Each page of EPC memory has an associated entry in the Enclave Page Cache Map (EPCM), which tracks metadata for that page. The SGX instructions use this metadata to ensure that EPC memory pages are always accessed in a secure manner. The EPCM is also used in the address translation lookup algorithm for enclave memory accesses, providing a secure additional layer of access control on top of existing mechanisms such as segmentation, paging tables, and extended paging tables [16].

The EPCM is managed by the processor as part of SGX operation and is never directly accessible to software or to devices. The format of the EPCM is microarchitectural and implementation specific, but conceptually each EPCM entry contains the following fields:

`VALID` Unallocated EPC pages are considered to be invalid. Pages in this state cannot be read or written by enclave threads and can only be operated on by allocation instructions that specifically require an invalid page as an input. If the `VALID` bit is not set, the remaining fields should not be examined.

`OWNER` An enclave instance is identified by its enclave control structure, which is a special kind of EPC page called an SECS. Each EPC page is associated with a single enclave instance. We track membership in an enclave instance through the `OWNER` field in the EPCM, which points to the SECS page of the enclave to which the page belongs.

`PAGETYPE` The `PAGETYPE` field describes the kind of data that is stored in the EPC page. In this paper we discuss four types of enclave page contents: regular enclave code or data (`REG`), thread control structures (`TCS`), enclave control structures (`SECS`), and data that has been deallocated but not yet reclaimed (`TRIM`).

`LINADDR` Enclave pages are accessible through a single linear address that is fixed at allocation time. SGX ensures that all accesses to an EPC page are through the appropriate linear address by comparing the address of the access to the stored `LINADDR` value.

`RWX` EPC page permissions may be set independently from page table and extended page table permissions, resulting in the minimal common access rights. The `RWX` bits of the EPCM track these supplementary permissions.

`PENDING` When an EPC page is dynamically added to a running enclave, the enclave code approves the addition of the new page as a protection mechanism against malicious or buggy systems software (see Sect. 2.2). During this intermediate period when the page has been allocated but not approved, the `PENDING` bit is set to prevent enclave code from accessing the page.

`MODIFIED` When the EPCM attributes of a page are dynamically modified by systems software, such as the `PAGETYPE`, the enclave code acknowledges the change using a process similar to dynamic EPC page allocation. In this case, the `MODIFIED` bit is set to prevent enclave code from accessing the page.

See Sect. 2.2 for a description of how these fields are manipulated by the enclave instructions.

Figure 1 illustrates how the EPCM enforces security on enclave page accesses, even in the presence of incorrect OS behavior. In this example, the OS has incorrectly mapped a page belonging to enclave B to enclave A, but any attempt by A to access the page will be prevented by the SGX hardware due to the mismatch in the EPCM `OWNER` field.

2.2 Instructions

A summary of the SGX instructions is shown in Table 1. The remainder of this section will examine the behavior and usage of each instruction in more detail.

Enclave Creation. The enclave creation process begins with `ECREATE`, a supervisor instruction that allocates an enclave control structure from a free EPC page. As part of invoking the instruction, systems software selects the desired location in the EPC for the enclave control structure and a linear address range that will be associated with the enclave. A successful call to `ECREATE` sets the `VALID` bit for the page, sets the `OWNER` pointer to itself, sets the `PAGETYPE` to `SECS`, and the EPCM `RWX` bits to zero.

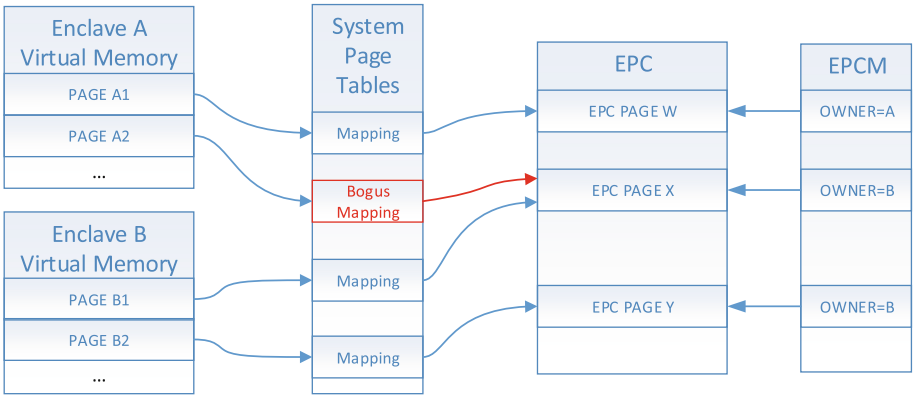

Fig. 1. Security protection in SGX. Systems software controls enclave memory with traditional structures like page tables, but cannot override the SGX security guarantees. Here, the OS maps enclave `A`'s virtual address `A2` to physical page `X`, which belongs to enclave `B`. Before allowing a memory access to `X`, the hardware checks the `OWNER` field, issuing a fault if the access does not come from enclave `B`. Here, this check prevents an unsecure access to `X` through the illegal mapping `A2`.

Enclave Initialization and Teardown. Once the SECS has been created, the enclave is initialized by copying data from normal memory pages into the EPC using `EADD`. A successful call to `EADD` sets the `VALID` bit for the page, associates the page with the specified enclave and sets the `OWNER` pointer to the appropriate SECS, sets the `PAGETYPE` to the specified type, and initializes the `RWX` bits. To destroy an enclave, system software deallocates all of its pages using `EREMOVE`.

Entering and Exiting an Enclave. SGX supports a standard call and return execution pattern through the instructions `EENTER` and `EEXIT`. The `EENTER` instruction puts the processor in a new enclave execution mode whereas `EEXIT` exits enclave mode and clears any enclave register state.

Dynamic Memory Management. Once an enclave is running, dynamic changes to its memory are performed as a collaborative effort between systems software and the enclave. The OS may allocate a new page (`EAUG`), deallocate a page or convert a `REG` page into a `TCS` (`EMODT`), and restrict the EPCM permissions available to the enclave (`EMODPR`). Without checks in place within the enclave, this could provide a vector for systems software to corrupt enclave data. To address this concern, we introduce the enclave-mode instruction `EACCEPT`, which an enclave executes to approve a change that was made by the OS. A successful call to `EACCEPT` finalizes the change by clearing the `PENDING` and `MODIFIED` bits. We also enable the enclave to perform dynamic changes itself where possible to reduce the number of enclave/kernel transitions. SGX currently supports enclave-mode permission extension (`EMODPE`) and a variant of `EACCEPT` that supports initialization of a newly allocated page (`EACCEPTCOPY`).

Table 1. Summary of the SGX instruction set. The table describes the instruction name, the processor mode from which the instruction should be called (supervisor, user, or enclave mode), and the usage of the instruction.

Name	Mode	Description
EACCEPT	enclave	Approve a dynamic memory change
EACCEPTCOPY	enclave	Approve and initialize a dynamically allocated page
EAUG	supervisor	Dynamically allocate a REG page
EADD	supervisor	Allocate a REG or TCS page
ECREATE	supervisor	Allocate SECS page and initialize control structure
EENTER	user	Call an enclave function
EEXIT	enclave	Return from enclave execution
EMODPE	enclave	Extend the EPCM permissions of a page
EMODPR	supervisor	Restrict the EPCM permissions of a page
EMODT	supervisor	Change the type of a page
EREMOVE	supervisor	Deallocate an enclave page

3 Proving SGX Is Linearizable in iPave

We encode linearizability as a model checking problem by inserting an assertion at the linearization point of each instruction. The assertion compares the current state of the EPCM (reached by some concurrent execution) with the known value that the EPCM would hold in a sequential execution. Any mismatch between the expected state and the actual state is caught by the model checker, and indicates that the instruction has observed an update by a concurrently executing instruction (that is, the instruction is not linearizable at that point). The linearization points are easy to identify in SGX because the instruction definitions all follow a similar pattern:

1. Pre-checking of parameters
2. Lock acquisition(s)
3. EPCM and other state checks
4. EPCM and other state updates
5. Lock release(s)

There is occasionally overlap between these steps, but it is always the case that a write is the last access to the EPCM and that this is a location where a correct SGX instruction implementation will have a linearization point.

We determine the appropriate linearization assertion on a per instruction basis by examining the EPCM state on which the instruction depends. We will see examples of this in the coming sections. In general, any EPCM value read by the instruction should not change between the time of the read and linearization point. The value of any EPCM field written by the instruction should not change between the time of the write and the linearization point. In some cases, tracking the value that was written requires the use of logic variables to remember intermediate values of the state.

3.1 Model Overview

We construct our formal model of SGX in iPave [10], a graphical specification language and SMT-based bounded model checker built on top of Boolector [4]. Similar to other modeling languages, an iPave model is specified as a finite state machine with guarded transitions between states. An example instruction specification in iPave is shown in Fig. 2. We can see that the `EMODPE` instruction shown in the figure follows the standard pattern, performing pre-checks on the EPCM to see if the running enclave may access the target page, acquiring a lock to synchronize accesses to the EPCM, checking that the state of the page is appropriate for the operation, and finally updating the `RWX` bits of the page.

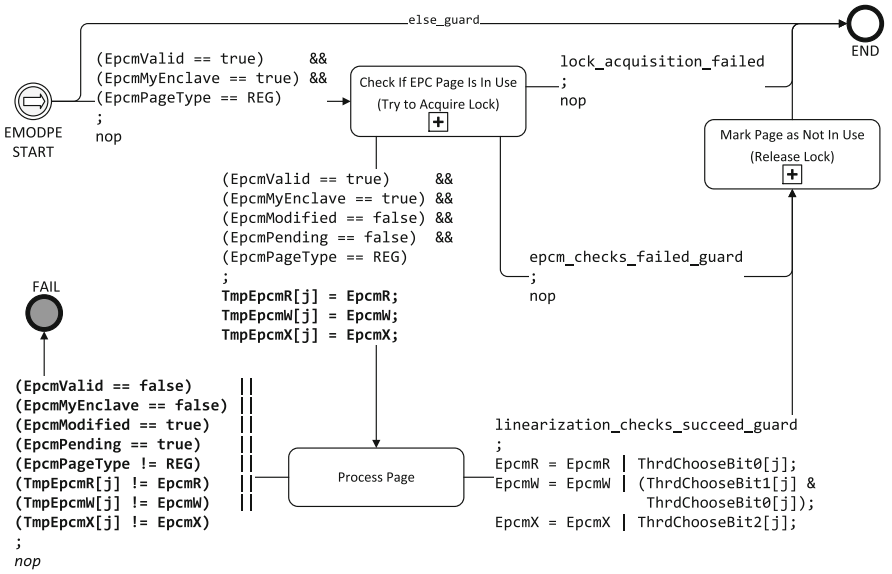

Fig. 2. Simplified `EMODPE` instruction specification in iPave. The model begins execution in the `EMODPE START` state in the upper left-hand corner of the diagram. Circles represent start or end states, rounded boxes represent intermediate states in the model's execution, and arrows represent transitions. Arrows may be labeled with guards and actions of the form `guard ; action`$_1$`; ...; action`$_n$`;`, which will only be executed if the guard is true. Either the guard or the action sequence may be empty; here the empty action sequence is represented by the effectless `nop`. Linearization checks and logic variable assignments are shown in bold.

The primary purpose of the iPave model is to verify that the SGX accesses to the EPCM are linearizable. To keep the problem tractable, the content of the model includes the minimum state necessary to describe the interaction between the instructions and the EPCM. The modeled state includes:

- A single concrete EPCM entry and its fields (see Sect. 2.1).
- An abstract representation of "the other" EPCM entries on the machine. Accesses to EPCM entries by SGX instructions are symmetric, so we employ this abstraction for improved performance.
- A single concrete enclave. Threads may enter this enclave and execute instructions on pages that belong to this enclave.
- An abstract representation of "the other" SGX enclaves.
- An array of logical processor states, which includes per hardware-thread data such as whether the processor is executing in enclave mode (and thus, allowed to access EPC pages) and other microarchitectural state.
- Metadata used to track model parameters and write assertions.

To improve the performance of the model checker, we initialize the state to an arbitrary reachable configuration, rather than a zeroed initial state. Though not all reachable states are known a priori, there is an easily calculable subset of the reachable states that can be used for initialization. In our experience, this significantly reduces the steps required to find interesting bugs. All of the SGX instructions described in Sect. 2.2 are modeled, as well as other relevant events such as memory accesses by enclave and non-enclave code.

3.2 Linearizability Assertions

We add linearizability checks to our iPave model according to the algorithm described at the beginning of the section, but optimize the insertion of the linearization point to reduce the number of possible interleavings. Our optimization is sound, but makes assumptions about the other instructions, making the approach less ideal than the general algorithm that we implement in Accordion. Examining Fig. 2, we see that the linearization checks are performed immediately before the state update to the `RWX` bits, avoiding the need to introduce an additional state after the `RWX` assignment. Immediately before the linearization point, we save the value of the `RWX` bits into logic variables. The intermediate state `Process Page` serves as a preemption point where other instructions could access the EPCM, after which we insert the linearization checks. In the case of `EMODPE`, the page must be valid, not pending, not modified, have the `REG` type, and belong to the currently running enclave. Furthermore, the values contained in the EPCM `RWX` bits should match the values saved in the logic variables. No other aspects of the state are accessed by the instruction, and thus no other fields need to be checked by our assertion.

3.3 Results

Our linearizability model in iPave uncovered architectural concurrency bugs that had not been discovered by manual inspection or testing, despite intensive review. We were also able to confirm a number of previously discovered bugs and increase overall confidence in the architecture design. Formal verification using iPave has been integrated into the SGX development and validation flow, where such methods were not previously common practice.

As an example of the kind of race that this methodology can detect, consider again the EMODPE example from Fig. 2. In that example, the PAGETYPE and OWNER of the page being modified by the instruction are checked twice: once before the EPC page lock is taken and once afterward. In an earlier version of the instruction definition, shown in Fig. 3, the OWNER is only checked before the lock acquisition and the PAGETYPE is only checked afterward. Due to an interaction with the EREMOVE instruction that is possible when a page has the type TRIM, this design allowed EMODPE to change the permissions of a page that did not belong to the running enclave, a clear violation of the SGX security guarantees. Though the bug might seem straightforward, it depends on a particular interleaving that is not easily triggered through testing.

```
(* Check security attributes of the EPC page *)
IF ((EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).OWNER != CR_ACTIVE_SECS))
   Then #PF(DS:RCX); FI;
(* Check the EPC page for concurrency *)
IF (EPC page in use by another SGX2 instruction) Then #GP(0); FI;
(* Re-check security attributes of the EPC page *)
IF ((EPCM(DS:RCX).PENDING != 0) or (EPCM(DS:RCX).MODIFIED != 0) or
    (EPCM(DS:RCX).PAGETYPE != REG) or (EPCM(DS:RCX).LINADDR != DS:RCX))
   Then #PF(DS:RCX); FI;
```

Fig. 3. Original EMODPE specification excerpt (Simplified)

The race condition found using iPave is shown in Table 2. The initial state for the model is that the concrete EPCM page belongs to the running enclave and is valid, not pending, not modified, and of the TRIM type. Examination of the race case revealed the root cause: the EPCM constraints in EMODPE were not sufficient to prevent the page from being removed during the instruction, nor was the removal detected as a failure mode of the instruction. The additional checks in the model shown in Fig. 2 prevent this race.

4 Automating the Process with Accordion

The iPave model was sufficient to demonstrate linearizability for a reasonably concrete model of SGX, but the modeling language and toolchain did not provide us with all of the features that we would like for our architecture explorations. We found that the graphical nature of the input language created a heavy translation burden from the original SGX specification. The translation process was a frequent source of modeling errors, and the disconnect between the specification and modeling language made it difficult for the architects to understand and evaluate the accuracy of the model. A further source of difficulty was the lack of abstraction mechanisms, such as functions, in the language. As a result, it was

Table 2. `EMODPE` race example

Step	Logical processor X	Logical processor Y
1	Start `EMODPE`	
2	Check `VALID` and `OWNER` fields	
3		Start `EREMOVE`
4		Check `PAGETYPE` and `MODIFIED` fields
5		Remove page; set `VALID=0`
6		End `EREMOVE`
7		Allocate page to another enclave with `VALID=1`, `MODIFIED=0`, `PENDING=0`, `PAGETYPE=REG`
8	Acquire exclusive access to page	
9	Perform post-lock EPCM checks	
10	Continue on another enclave's page!	

not easy to experiment with different SGX configurations (number of simultaneously running threads, number of memory pages, instructions to include in the verification run), and modifications required extensive manual effort.

To address the gaps in iPave for our usage, we designed and implemented Accordion, a domain specific language and compiler for proving linearizability. We focused on the following goals in the design of the language:

Mirror Existing Design Specification Language. Instruction set extensions are typically specified in a semi-formal notation that is not machine checkable. The Accordion language should match this syntax as closely as possible so that architects can comprehend the models easily, while also providing a machine checkable format with a defined semantics. The ultimate goal is that Accordion will supplant the informal specification language for SGX.

Support Rapid Prototyping. Design changes are frequent as a feature is extended and optimized. Bugs must be found as early in the design process as possible for verification to be worthwhile. Synchronization behavior is a particular source of experimentation, so the language should support a variety of locking primitives and should make varying the size and location of atomic blocks simple for the designer.

Enable Designers to Leverage a Variety of Analysis Tools. No tool consistently yields superior results. A single source avoids translation errors and provides significant time savings.

Support Experimentation within a Particular Analysis Tool. Model checkers and other verification tools can be sensitive to the size of the input problem. When analyzing a particular design, the validator needs to experiment with the configuration of the model to produce results in a timely fashion.

Automate Linearizability Analysis. Calculate the linearization points and generate linearizability assertions automatically in the compiler.

Facilitate Experimentation with Different Interleaving Semantics. In our experience, full interleaving semantics does not scale to models of the complete SGX architecture. To gain traction in our analysis, we would like to evaluate a collection of models with a variety of interleaving semantics, for example, analyzing atomic versions of enclave initialization instructions with fully interleaved versions of the dynamic memory management instructions.

In the remainder of this section, we will show how Accordion meets these goals by providing an overview of the language syntax, describing the compiler implementation in Haskell [18], and sketching the algorithm for automatically calculating linearization points and linearizability checks.

4.1 Language Syntax

The Accordion language supports a basic set of types including physical addresses, Boolean values, and unsigned integers. SGX data structures like the EPCM are built into the language as well, but ultimately we would like to introduce user-defined data structures to maximize extensibility to future hardware features. The expression language contains constants, variables, standard Boolean and arithmetic operations, structure accesses (for reading SGX data structures), and an address validation operation that performs SGX-specific checks on a physical address, such as membership in the EPC. The statement language includes variable assignments, conditionals, assertions, mutex and reader/writer lock acquire and release operations, abort statements, structure updates (for writing to SGX data structures), and atomic blocks (used to override the default grouping of statements into rules in the compiler; see Sect. 4.2).

Figure 4 shows the code for the `EMODPE` instruction in Accordion syntax and is analogous to the iPave specification shown in Fig. 2. As we will see in the next section, Accordion is implemented as an embedded domain specific language in Haskell, so aspects of the Haskell syntax are mixed with the Accordion syntax in the example. The instruction specification is written as a function with four arguments, only three of which are used by `EMODPE`. The argument `cr_active_secs` is a pointer to the enclave that is currently running, `rcx` contains the physical address of the EPC page to be modified, and `rbx` contains a data structure called a `secinfo` that specifies the desired permissions for the target EPC page. In hardware, and in the model produced by the Accordion compiler, these instruction parameters are provided implicitly as part of the system state.

Those familiar with Haskell will notice that the code is written in a monad using do-notation, but it is not essential to understand this mechanism in order to comprehend the code. The instruction content begins on Line 3 with a check that the target EPC page is accessible (valid, regular, and owned by the currently running enclave). If the check fails, execution will abort with page fault semantics (#PF). If the check succeeds, the instruction will continue with the

```
1  emodpe :: SGXInstruction
2  emodpe cr_active_secs rbx rcx _ = do
3   ift ((!)((epcm rcx).valid) ||
4         (epcm rcx).pagetype != REG ||
5         (epcm rcx).owner != cr_active_secs) (do
6     (#)PF rcx)
7   (epcm rcx).mutex.acquire() (do
8     (#)GP 0)
9   ift ((!)(epcm rcx).valid) || (epcm rcx).pt != REG ||
10        (epcm rcx).pending || (epcm rcx).modified ||
11        (epcm rcx).owner != cr_active_secs) (atomic $ do
12    (epcm rcx).mutex.release()
13    (#)PF rcx)
14  (epcm rcx).r =: (epcm rcx).r || (secinfo rbx).r
15  (epcm rcx).w =: (epcm rcx).w || (secinfo rbx).w
16  (epcm rcx).x =: (epcm rcx).x || (secinfo rbx).x
17  (epcm rcx).mutex.release()
18  end_of_flow
```

Fig. 4. Simplified `EMODPE` specification in accordion.

next statement at the leftmost level of indentation (Line 7). The rest of the code follows a similar pattern of execution. Except for the occurrences of `do`, `$`, and some case mismatches, the syntax shown here is very close to the informal specification language used by the SGX architects.

4.2 Compiler Implementation

We implemented Accordion as an embedded domain specific language (DSL) in Haskell [11]. When writing an embedded DSL, the language designer encodes the abstract syntax tree of the new language directly in the host language, allowing the designer to take advantage of the parser and type system of the host language in their DSL. This allowed us to get a version of Accordion running much faster than would have been possible with a standalone DSL. The disadvantage to using an embedded DSL is that there are certain syntactic aspects of the host language that cannot be overridden. For example, we cannot use the same symbol for assignment in Accordion as in the SGX specification, `:=`, because of the special treatment of colon in Haskell.

Model Generation. Our compiler supports two back-ends: one for generating Murphi syntax—which is compatible with the explicit state model checker CMurphi [7] and its distributed counterpart PReach [3]—and one for generating input to the symbolic bounded model checker SAL [2]. We found that running SAL with a relatively low bound performed well on many of our examples and was useful for finding early modeling bugs. PReach was slow to run but for small enough models (with either an abstract definition of the SGX instructions or a

model that does not include the full instruction set) was able to give us a full proof of linearizability.

The full SGX architecture without any simplifying abstractions is too large to be fully verified, which is why it is so important for the compiler to provide support for generating models with a variety of configurations. We provide control over the number of threads, the number of memory pages, and the set of instructions to include in the generated model with compile time flags. Interleaving semantics are also specified at compile time on a per instruction basis. By default, every statement in an Accordion program becomes a rule in the resulting model. Sometimes, we are interested in evaluating the concurrent interaction between two or three instructions at this level, but are not interested in analyzing the initialization instructions that are necessary to drive the model into an interesting state. In those cases, we would compile the specification with full interleaving semantics for the instructions of interest but with atomic semantics, where the entire instruction becomes a single model transition, for the rest. The DSL also provides an `atomic` primitive for concurrency control between these extremes. Figure 4 shows an example of this primitive on Line 11, which will cause the lock release statement on Line 12 and the abort statement on Line 13 to compile to a single model checker rule.

Linearizability Inference. As discussed in Sect. 3, the SGX instruction definitions follow a common pattern that makes the location of the linearization point clear from a cursory inspection. In fact, this pattern is predictable enough that it can be computed, along with the precise set of checks that must be satisfied in order for the instruction to be linearizable.

Calculating the linearization point is relatively straightforward. We perform a backward walk through the control flow graph of the instruction, skipping past any abort statements (`end_of_flow` or a fault) or lock releases, until an SGX state update is found. We insert the linearization point here, after the final state update that the instruction performs. In our `EMODPE` example from Fig. 4, the linearization point would be placed in between Lines 16 and 17.

Once the linearization point has been identified, the compiler computes the assertion that should be placed there in the generated code. For this analysis, the compiler performs a forward walk over the control flow graph of the instruction, accumulating assumptions based on the portions of the SGX state that the instruction reads or writes. In Line 3 of `EMODPE`, for example, the instruction checks that the `VALID` bit is set, that the `PAGETYPE` is `REG`, and that the `OWNER` of the page matches the running enclave. This will generate an assertion that checks for that scenario at the linearization point. If another thread has modified the `VALID` bit, `PAGETYPE`, or `OWNER` of the page in the mean time, the assertion will fail. The full linearization assertion for `EMODPE` is:

$$
\begin{aligned}
\mathit{assert}(\ & (\text{epcm rcx}).\text{valid} \wedge (\text{epcm rcx}).\text{pagetype} = \text{REG} \wedge \\
& (\text{epcm rcx}).\text{owner} = \text{cr\_active\_secs} \wedge \\
& (!)(\text{epcm rcx}).\text{pending} \wedge (!)(\text{epcm rcx}).\text{modified} \wedge \\
& (\text{epcm rcx}).\text{r} = ((\text{epcm rcx}).\text{r'} \parallel (\text{secinfo rbx}).\text{r}) \wedge \\
& (\text{epcm rcx}).\text{w} = ((\text{epcm rcx}).\text{w'} \parallel (\text{secinfo rbx}).\text{w}) \wedge \\
& (\text{epcm rcx}).\text{x} = ((\text{epcm rcx}).\text{x'} \parallel (\text{secinfo rbx}).\text{x})\ )
\end{aligned}
$$

As outlined in Sect. 3, this assertion checks that any state accessed by the instruction does not change between the time of the access and the linearization point. The values `r'`, `w'`, and `x'` in the example are logic variables used to track the intermediate value of the EPCM.

Recall that the buggy version of `EMODPE` discussed in Sect. 3.3 checked the page owner before acquiring the lock but not afterward and only checked the page type afterward. This allowed the page to be removed and assigned to a different owner, without the ownership change being caught by the instruction. We can see that these automatically generated linearization assertions would catch this error, by identifying the requirement that the page owner remain unchanged from the moment of the first read until the linearization point.

Results. The models produced by the Accordion compiler are able to replicate the bugs that were found both by inspection and by formal verification using iPave. By making use of Accordion's model configuration facilities, we are able to construct experiments that find bugs in a matter of minutes and complete a total verification for a subset of instructions in a matter of hours, as opposed to the many hours or even days that iPave would require. These models are by no means equivalent, but for design-time analysis of new instructions the immediate feedback provided by the Accordion models is more useful than a long-running exhaustive verification. We can of course perform a full verification of SGX using Accordion as well.

No new bugs were found using Accordion because the SGX architecture was largely stable by the time our work on Accordion was complete. However, there are many ongoing projects within the SGX architecture team that would benefit from the sorts of analysis that Accordion provides.

5 Related Work

Model checking linearizability is not itself a new idea. Our algorithm for checking linearizability using a shadow state is equivalent to the algorithm for verifying commit-atomicity in [9]. In our case, we are able to calculate the shadow state within the Accordion compiler and generate the appropriate linearizability checks, avoiding the need to explicitly track the shadow state in the generated model. Much of the other work on linearizability to date has focused on general purpose concurrent data structures, such as lists [6,22,24]. There are also tools for automatically proving linearizability, such as CAVe [23] and Line-Up [5]. With SGX, our domain specific knowledge allows us to prove linearizability with

a simpler approach, using off-the-shelf tools. We focus our efforts on the model generation code, which supports experimentation with a variety of tools and model configurations. Unlike other tools, Accordion provides fine grained control over the interleaving semantics in the generated model, allowing for a great deal of control over the possible schedules.

Similar to our work, CHESS [20] tackles the challenge of finding concurrency bugs in large systems that may not be amenable to full verification. The tool uses a happens-before analysis to analyze the concurrent execution of a system and exhaustively explores all interleavings in the program. This approach is not ideal for SGX because we do not use a standard set of concurrency primitives in our implementation. CHESS achieves scalability by bounding the number of context switches that may occur in a particular run of the system and by scoping the context switches to an area of interest in the program. These techniques for improving scalability would likely work well in our problem domain where races tend to involve a small number of context switches and where we have a strong sense for where races are likely to occur.

The Copilot DSL [21] is similar to our work on Accordion in that it is also embedded in Haskell and provides multiple formal verification back-ends. One back-end that Copilot compiles to is the Haskell SBV library [8], which supports reasoning about Haskell programs using SMT. This approach would be a valuable extension to Accordion. Currently our sequential verification proofs which use SMT and our linearization work using model checking are not connected via the same source. The two projects differ in the domain targeted by the DSLs: Copilot focuses on run-time monitoring for hard real-time programs, an area with very different considerations than SGX design. Other domain specific languages for hardware design exist, such as Kansas Lava [12] and Hawk [19], but these target low-level circuit designs rather than high-level features like SGX.

6 Summary

This paper introduced our approach to verifying SGX, a novel collection of hardware instructions for providing strong integrity and confidentiality guarantees in a highly concurrent setting. We identified linearizability as the relevant correctness condition for analyzing concurrent interactions between SGX instructions and described an algorithm for demonstrating linearizability using off-the-shelf model checking tools. Our work showed that this approach is capable of finding critical security bugs and underscored the importance of performing formal verification early in the design process of complex features like SGX. Building on the success of our verification in iPave, we outlined the development of the Accordion domain specific language and compiler, a tool that automatically proves linearizability for SGX instructions via model checking and supports experimentation with a wide variety of model configurations across multiple model checking tools.

References

1. Baumann, A., Peinado, M., Hunt, G.: Shielding applications from an untrusted cloud with Haven. In: 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2014. USENIX Advanced Computing Systems Association, October 2014
2. Bensalem, S., Ganesh, V., Lakhnech, Y., Muñoz, C., Owre, S., Rueß, H., Rushby, J., Rusu, V., Saïdi, H., Shankar, N., Singerman, E., Tiwari, A.: An overview of SAL. In: LFM, pp. 187–196, Williamsburg, VA, USA (2000)
3. Bingham, B., Bingham, J., Paula, F.M.d., Erickson, J., Singh, G., Reitblatt, M.: Industrial strength distributed explicit state model checking. In: PDMC, pp. 28–36, Washington, DC, USA (2010)
4. Brummayer, R., Biere, A.: Boolector: an efficient SMT solver for bit-vectors and arrays. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 174–177. Springer, Heidelberg (2009)
5. Burckhardt, S., Dern, C., Musuvathi, M., Tan, R.: Line-up: a complete and automatic linearizability checker. In: Programming Language Design and Implementation, PLDI 2010, June 2010
6. Černý, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., Alur, R.: Model checking of linearizability of concurrent list implementations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 465–479. Springer, Heidelberg (2010)
7. Dill, D.L., Drexler, A.J., Hu, A.J., Yang, C.H.: Protocol verification as a hardware design aid. In: ICCD, pp. 522–525, Cambridge, MA, USA (1992)
8. Erkök, L.: SBV: SMT based verification in Haskell. http://leventerkok.github.io/sbv/
9. Flanagan, C.: Verifying commit-atomicity using model-checking. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 252–266. Springer, Heidelberg (2004)
10. Fraer, R., Keren, D., Khasidashvili, Z., Novakovsky, A., Puder, A., Singerman, E., Talmor, E., Vardi, M., Yang, J.: From visual to logic formalisms for SoC validation. In: Formal Methods and Models for System Design, MEMOCODE 2014
11. Gill, A.: Domain-specific languages and code synthesis using Haskell. Queue **12**(4), 3030–3043 (2014)
12. Gill, A., Bull, T., Farmer, A., Kimmell, G., Komp, E.: Types and associated type families for hardware simulation and synthesis: the internals and externals of Kansas lava. In: Higher-Order and Symbolic Computation, pp. 1–20 (2013)
13. Goel, A., Krstić, S., Leslie, R., Tuttle, M.R.: SMT-based system verification with DVF. In: Satisfiability Modulo Theories, SMT 2012, pp. 32–43 (2012)
14. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. (TOPLAS) **12**(3), 463–492 (1990)
15. Hoekstra, M., Lal, R., Pappachan, P., Phegade, V., Del Cuvillo, J.: Using innovative instructions to create trustworthy software solutions. In: Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP 2013, ACM, New York, NY, USA (2013)
16. Intel Corporation, Santa Clara, CA, USA. Intel® 64 and IA-32 Architectures Software Developer's Manual, September 2014
17. Intel Corporation, Santa Clara, CA, USA. Intel® Software Guard Extensions Programming Reference, October 2014
18. Marlow, S.: Haskell 2010 language report 2010

19. Matthews, J., Cook, B., Launchbury, J.: Microprocessor specification in Hawk. In: Proceedings of the 1998 International Conference on Computer Languages, pp. 90–101. IEEE Computer Society Press (1998)
20. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: Eigth Symposium on Operating Systems Design and Implementation, OSDI 2008. USENIX, December 2008
21. Pike, L., Goodloe, A., Morisset, R., Niller, S.: Copilot: a hard real-time runtime monitor. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) RV 2010. LNCS, vol. 6418, pp. 345–359. Springer, Heidelberg (2010)
22. Sethi, D., Talupur, M., Malik, S.: Model checking unbounded concurrent lists. In: Bartocci, E., Ramakrishnan, C.R. (eds.) SPIN 2013. LNCS, vol. 7976, pp. 320–340. Springer, Heidelberg (2013)
23. Vafeiadis, V.: Automatically proving linearizability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 450–464. Springer, Heidelberg (2010)
24. Vechev, M., Yahav, E., Yorsh, G.: Experience with model checking linearizability. In: Păsăreanu, C.S. (ed.) Model Checking Software. LNCS, vol. 5578, pp. 261–278. Springer, Heidelberg (2009)