Work In Progress: Discovering Emergent Computation Across Abstraction Boundaries

Mark Boddy[*], **Jim Carciofini, Todd Carpenter, Alex Mahrer, Ryan Peroutka, Kyle Nelson**[†]
Adventium Labs
{*firstname.lastname*}@adventiumlabs.com

Abstract

In this Work in Progress report, we describe ongoing research on our *DECIMAL* project, addressing the problem of modeling computational mechanisms at sufficient fidelity to reason about the execution semantics of programs across abstraction boundaries. The automata-based formalism that we have developed is specifically constructed to support reasoning about timed behavior over compositions of multiple component automata, modeling different parts of the system under study. We show how we use composition to model a wide variety of constructs, including synchronization across abstraction boundaries, communicating asynchronous processes, and specifying programs that can be generalized across different architectures and over localized variations in the program specification.

Introduction

The research community that has coalesced around the LangSec workshops has generated significant results addressing the difficulties posed by *ad hoc* treatment of program inputs, including messages passed between parts of a distributed system. While this is clearly crucially important work and of the first priority to be addressed, verifiably safe input parsing only solves part of the problem. As argued by (Dullien 2017), any sequence of inputs provided to an executing process can be viewed as a *program*. The execution semantics for that program is then determined by the process itself, along with the environment in which that process runs, potentially including details down to the underlying hardware.

As extensively demonstrated in the very large and growing body of recent work on speculative exploits in modern CPU architectures (e.g., Meltdown (Lipp et al. 2018), Spectre (Kiriansky and Waldspurger 2018), ExSpectre (Wampler, Martiny, and Wustrow 2019), RIDL (Van Schaik et al. 2019), among a host of others), it is not necessary for there to be *any* errors or inconsistencies in input parsing for profoundly dangerous exploits to be feasible. There need not be any *weird states* as defined in (Dullien 2017). The problem that we highlight here is what happens in the "runtime" that executes the program. One common example (but by no means the only instance of such a runtime) is in CPU micro-architectures that execute ISA-level instructions. As argued in (Mcilroy et al. 2019), such runtime interpreters are *emulators*. By direct implication, unless very carefully constrained, these interpreters are permissively specified. Given a set of instructions, they will (if correctly implemented) perform computations and report results according to the specified execution semantics of the language being interpreted. It is in what *else* they may do along the way that the problems arise.

In this Work in Progress report, we describe ongoing research on our *DECIMAL* project, addressing the problem of modeling computational mechanisms at sufficient fidelity to reason about the execution semantics of programs across abstraction boundaries. The automata-based formalism that we have developed is specifically constructed to support reasoning about timed behavior over compositions of multiple component automata, modeling different parts of the system under study. Composition is used to model *abstraction*, such as between different layers in a hierarchy, as well as to represent the possibly-asynchronous execution of multiple processes within a given abstraction level.

Our main achievements on this project to date include:

- Rigorous syntax and semantics for the *DECIMAL* formalism, available on request to the research community.
- Implemented tools for automaton composition and translation to SMV.
- Execution of a small set of case studies, focused on emergent computation across the ISA/CPU abstraction boundary.
- Explicit representation for asynchronous processes, pipelining, and race conditions in the micro-architecture.
- Conventions for using automaton composition to generalize system models across different kinds of speculative exploits, and across different CPU architectures.
- Formulation and automated checking of a general property indicating the potential for speculative information disclosure.

[*]Corresponding author

[†]This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112090029. Approved for public release; distribution is unlimited. Copyright 2021 Adventium Labs.

In the rest of this paper, we summarize our objectives and technical approach, provide and motivate a specialized definition of the term *abstraction*, and discuss how we can use automata composition to model a wide variety of constructs, including synchronization across abstraction boundaries, communicating asynchronous processes, and specifying programs that can be generalized across different architectures, and over localized variations in the program specification. We then wrap up with a summary, observations, and a discussion of future work.

Summary of Approach

The objective of this research is to identify *emergent computation*, meaning unexpected or unauthorized computation using existing constructs, enabled by interactions across abstraction boundaries. Such boundaries occur, for example, between protocol layers in a layered communication stack, or between computational abstractions, as for example between the ISA and the micro-architecture in a CPU. We are specifically interested in the use of time, both as a communication channel and as a computational mechanism. In our research to date, the abstraction boundary that we have spent the most time investigating is the interface between the Instruction Set Architecture (ISA) and the μ-architecture of the Central Processing Unit (CPU) in a general-purpose computer.

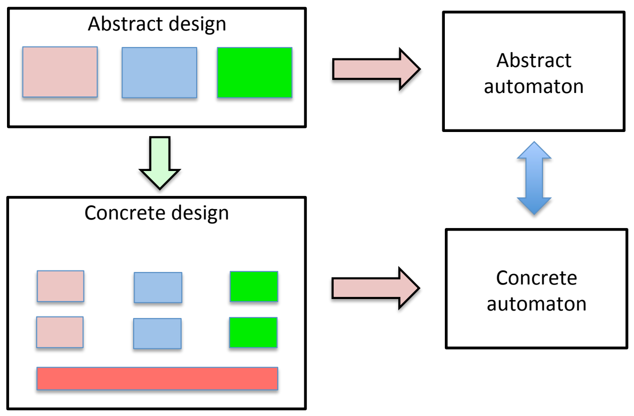


Figure 1: Abstract design components are refined into concrete components. An abstract automaton is constructed by composing models of a selection of abstract components. The corresponding concrete automaton is similarly constructed by composing models of concrete components.

As shown schematically in Figure 1, the methodology we are implementing includes:

1. Choosing a subset of abstract system components to include (e.g., one might choose a subset of the ISA, or focus on a single CPU core). Composing the corresponding automata models defines the abstract automaton.
2. Choosing a set of concrete components implementing the concrete design. For example, one might or might not choose to model various levels of cache, or the memory management unit (MMU), or both. Composing the corresponding automata models defines the concrete automaton.
3. Defining the interaction of the abstract and concrete automata at the abstraction boundary.
4. Adding any desired assumptions regarding special capabilities (e.g., Rowhammer-style bit flipping as an "additional instruction." (Mutlu and Kim 2019)).
5. Evaluating the resulting model for behavior indicating potential emergent computation.

In order to efficiently represent compositions of computational elements, we borrow from and build upon recent work of several researchers, including (Aarts and Vaandrager 2010), (Cledou, Proença, and Soares 2017), (Jonsson and Vaandrager 2017), and (D'Antoni et al. 2019). The resulting *DECIMAL* formalism includes the following features:

Interface automata as defined in (de Alfaro and Henzinger 2001) provide a clean separation between system *inputs* controlled by the environment, and *outputs* controlled by the system. For system components, the environment may consist in whole or in part of other parts of the system, and so this separation also simplifies the problem of composing automata representing different system components, either within or across levels of abstraction.

Timed automata model time explicitly. Timing side channels are prevalent, may have high bandwidth, and readily cross security boundaries (it is not even required to have a shared clock). Further, variable timing can itself be used as a form of computation, for example by determining which of two asynchronous events wins a timing race.

Registers provide a means of modeling the storage and comparison of information, supporting an explicit memory model.

Symbolic Predicates over transition labels provide a means of defining *classes of information*, for example differentiating between legal and illegal memory accesses without having to enumerate the corresponding sets of addresses.

To keep the resulting models finite, we require registers to be of finite size, and bound the length of symbolic predicates and restrict them to finite domains. Similarly, time is represented by a finite set of bounded-integer-valued countdown timers, rather than clocks.

The three automata shown in Figure 2 demonstrate how these syntactic extensions are used in practice. The automaton above the blue line represents a simple ISA-level `READ` instruction. Each transition is labeled with a symbol, consisting of a *functor* and some number of *parameters*. The automaton immediately below the blue line specifies the corresponding operations in the micro-architecture, in particular showing a branch in execution depending on whether the memory location to be read is in cache. In the first transition in that automaton, `eip := param` is a register operation, "storing" the value denoted by `param` in a register `eip`.[1] The symbol `tm_ur` denotes a timer. When it appears on a transition with an arrow and a number, that is an assignment that starts the timer. When it appears on a line by itself, that indicates a transition that is initiated by the timer timing out.

The composed automaton at the bottom of the picture *synchronizes* the two initial and two final transitions between the ISA-level and CPU-level automata. There are two such transitions because in that model we represented ISA-level instruction sequencing separately from parameter information, here the address to be read, and the information returned. The `EXEC()` transitions are used to define instruction sequence and to track the value of the instruction pointer `(eip)`. Note that by virtue of the way this model is constructed, there is no flexibility for out-of-order, or even pipelined, execution in the CPU.

As defined above, the final step in the analysis process supported by *DECIMAL* is to search for evidence of emergent execution. The *DECIMAL* formalism is sufficiently expressive for this purpose. However, performance considerations, in particular the availability of existing, highly optimized tools, have led us to implement a translator from that formalism to the model-checking language SMV. The resulting SMV model is then submitted to the nuSMV model-checker to be checked against properties expressed in an appropriate logical formalism, in this case Linear Temporal Logic (LTL) or Computational Tree Logic (CTL).

The properties to be checked encode the potential for specific kinds of emergent execution. For example, here is the CTL expression of a property encoding a disabling condition, prohibiting a Meltdown-style exploit:

$$\neg\ (\neg\ \text{legal}[\mathit{protectedAddr}] \wedge \neg\ \text{inCache}[S] \wedge \\ \mathbf{E}\ [\neg\ \text{inCache}[S]\ \mathbf{U}\ \text{inCache}[S]\,])$$

In words:

> It is false that the data S residing at a prohibited-access location $\mathit{protectedAddr}$, initially not in the cache, will on some execution path(s) eventually appear in the cache.

While time does not appear in the CTL statement above, timing plays a crucial role in whether this property can be verified for a given model. For a model of the Meltdown exploit we generated as a case study, nuSMV generates a counter-example trace in about 10 seconds, using a laptop-class machine.

Defining Abstraction

As commonly used in automata theory, *abstraction* defines a relation with specific properties between two different transition systems. This mathematical view does not explicitly account for the flow of control and information through the system in operation, which in practice is of central importance in modeling the use of that system to accomplish some objective. Nor does the conventional view explicitly support the representation of autonomous, asynchronous processes within a given layer in the abstraction hierarchy.

To address these issues, we offer the following operational definition of abstraction, using the extensions in the *DECIMAL* formalism. An *abstraction level* is a Timed Interface Automaton. An *abstraction interface* is a mapping between two abstraction levels. The mathematical definition described above defines an abstraction as a relation between sets of states. Here, we are more interested in transitions. An abstraction interface specifies:

- Output labels in the upper-level (abstract) automaton, which are synchronized with a corresponding set of input labels in the lower-level (concrete) automaton. By convention, user input is restricted to the top-level automaton, and so we say that the interface is *controlled by* the abstract automaton, and that it *controls* the concrete automaton
- Output labels in the concrete automaton, which are synchronized with a corresponding set of input labels in the abstract automaton. These synchronized labels are used to model the flow of information from concrete to abstract automata.

As with the more conventional mathematical definition, we have said nothing about information hiding, for example whether the internal state of a given abstraction layer is externally visible. Certainly it can be made visible from above, by inputs requesting some report regarding that state.

Figure 1 shows these relationships schematically. Note in particular that there are states and transitions in the concrete layer that are not visible in and do not correspond to any specific states or transitions in the abstract layer. Mapping this definition of abstraction into the set-based mathematical one is straightforward. In order to map transition paths that are of length greater than 1 in the concrete automaton, abstract transitions that in the set-based model would be of length 1 are now of length 2. These can be thought of as "dispatch" and "commitment" transitions. The intervening added state can be interpreted as the interval over which some activity is occurring.

In this view of abstraction, the top-level, most abstract automaton in the system model defines the space of possible programs (or messages, or other possible inputs to a system), and thus is effectively determined by that space. If this automaton encodes, say, programs over the entire x86 instruction set, then the automaton may become immense. One possible solution is to make the automaton very strongly Markov. Suppose, for example, that all transitions are self-loops back to the initial state. Then the automaton consists of a single state and a number of transitions equal to the size of the instruction set, rather than growing exponentially with program length. For *DECIMAL*, this is a more practical approach than it may seem at first. In *DECIMAL*, changes to and tests of registers, symbolic predicates, and timer operations are all defined on transitions, not as properties of states. Synchronization of transitions across abstraction layers is defined in terms of transition *labels*. Finally, as shown in the simple example in Figure 2, execution effects are largely driven by lower levels in the abstraction hierarchy.

[1]As used here, the term "register" refers to a syntactic extension to finite automata, not to storage locations commonly found in CPUs.

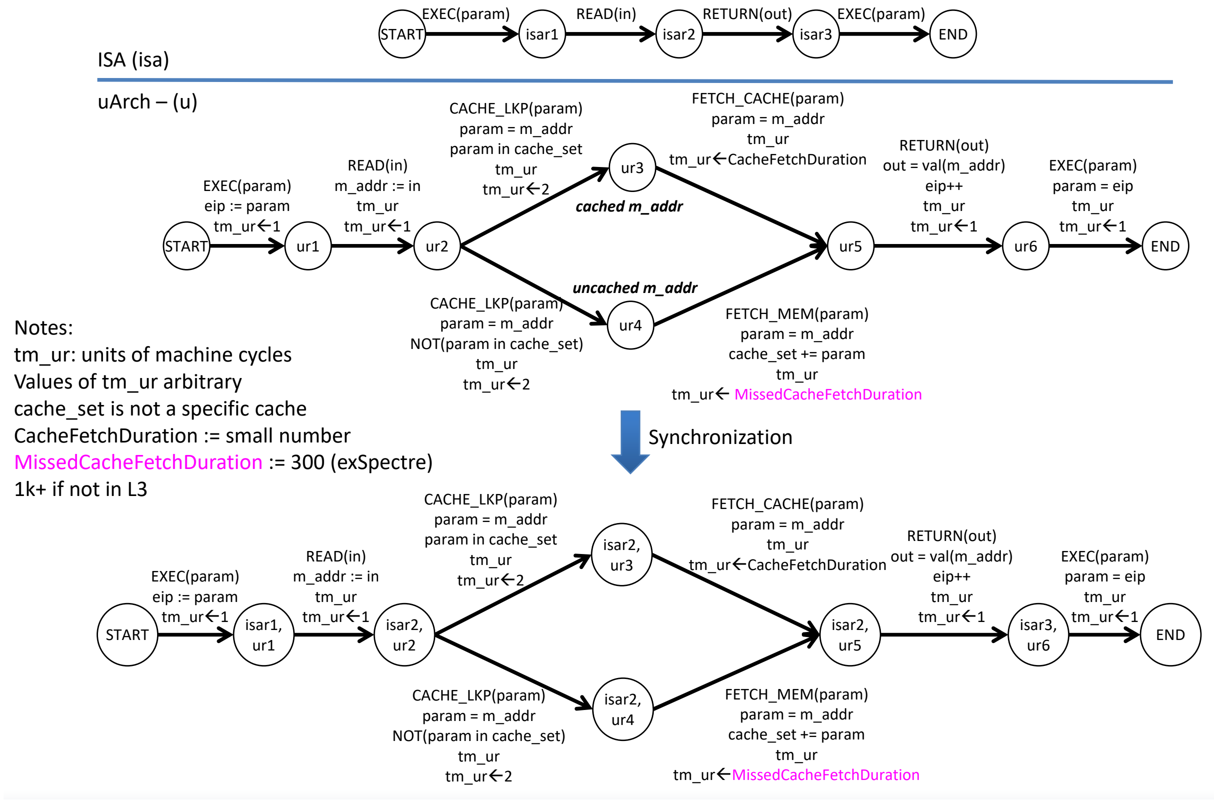


Figure 2: Composing `READ` across the ISA/CPU boundary reveals the deterministic timing variation that enables covert timing channels using the cache.

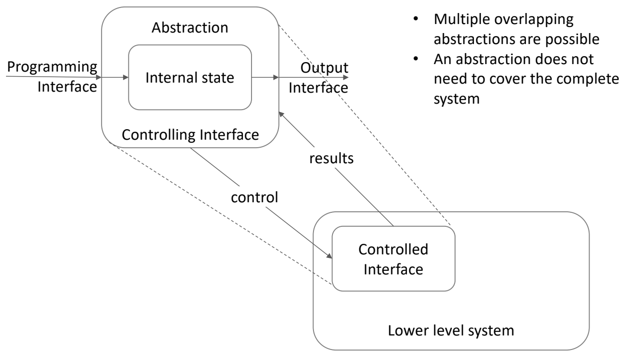


Figure 3: Abstraction and emulation

As a result of these properties, we can encode all possible programs very efficiently. Of course, this does not alter the number of possible programs of less than a given length, which is still very large. It is worth noting as well that there is a large middle ground. For example, we can encode a program of length N using an ordering of N states, with transitions only between adjacent pairs of states. This might be a good choice for, say encoding an ordering requirement over the space of possible programs.

Composition

The most important function in the modeling process for *DECIMAL* is the *composition* of two or more automata, usually with the specified *synchronization* of two or more labels between those automata. Composition has several uses, including mapping across abstraction boundaries (for example as shown in Figure 2), specifying programs in the top-level abstraction, and integrating multiple computational processes within a given abstraction layer. In this section, we provide a brief sketch of some of these applications.

As previously stated, one of the first things we did on this project was to automate composition. Figure 4 shows the composition of three micro-architectural components of a model of Meltdown from a case study we conducted. That model is discussed in more detail, though not fully described, in the rest of this section. Figure 4 makes it clear why automating the process of composition was a high priority for us.

Figure 5 shows an example of the uses of composition and synchronization for asynchronous processes within a given abstraction layer, in this case the micro-architecture of a Load/Store CPU. In Figure 5, we see a model of the processes involved in a memory load to a register, based on an indirect reference. The two automata shown model separate processes, which are composed and synchronized on the transition labels `post(ISR,v)`, `allow`, and `deny`. Un-

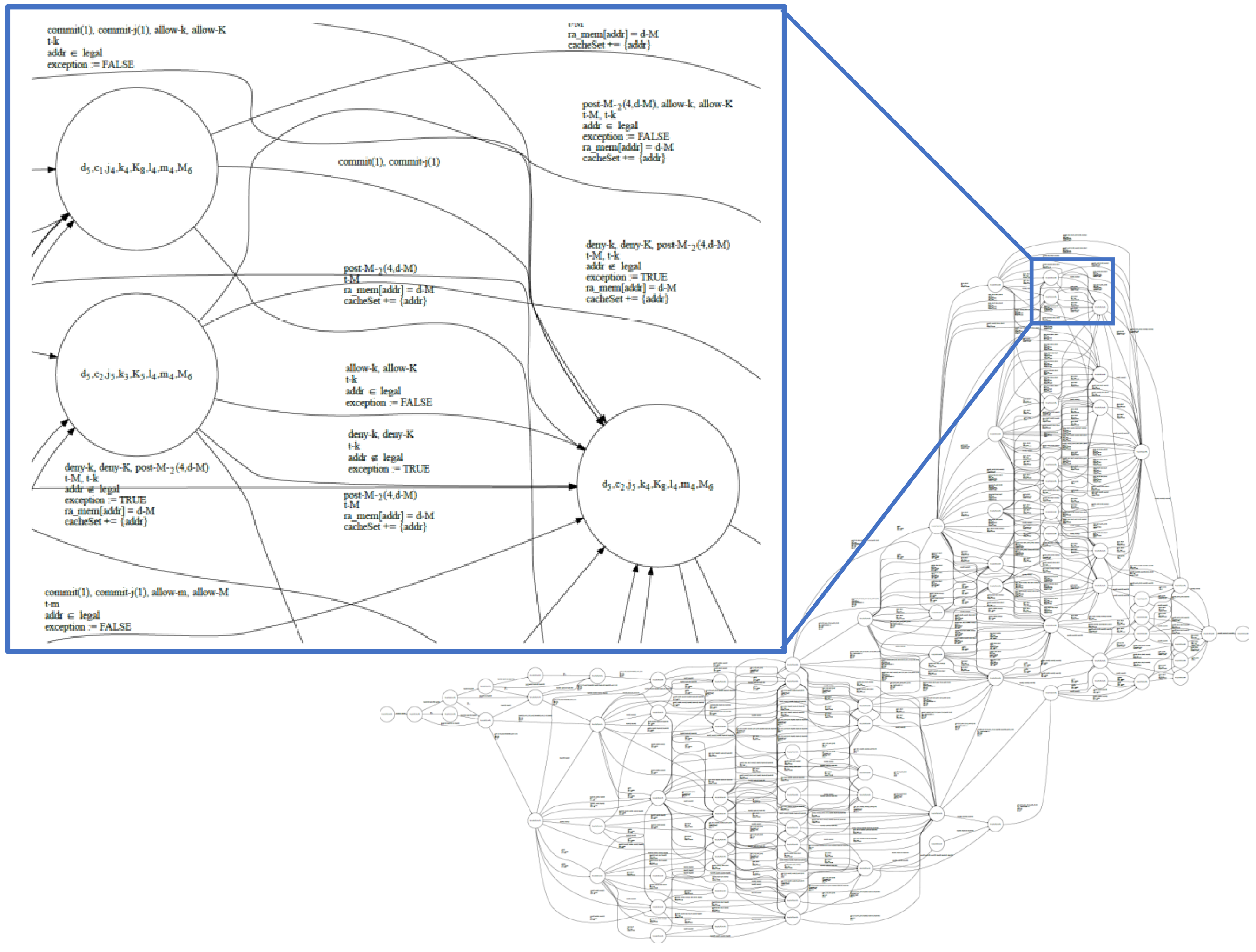


Figure 4: Composition of the component automata for the detailed Meltdown model from our case study, with a small part shown in detail.

derstanding this model requires keeping in mind a specific distinction in our formalism: synchronization is specified in terms of transition *labels*, rather than individual transitions. For example, in Figure 5, synchronizing on the label `deny` means that the transition labelled `deny` in the top automaton must occur at the same time as *one or the other* of the two transitions labeled `deny` in the lower automation. Conversely, neither of the the two `deny` transitions in the lower automaton can occur at all, if it does not co-occur with the `deny` transition in the top automaton.

This choice encodes the race condition that potentially enables a specific speculative execution vulnerability, resulting in access to unauthorized information. If the delay `CT` required to do the authorization check happens fast enough to permit the upper automaton's `deny` transition to co-occur with the earlier `deny` transition in the lower automaton, then execution will be rolled back before the required side-effect on the cache can occur. The three transitions shown in red, all with the label `post(DR,d)`, appear on three different paths through the automaton. In the top path, authorized *or unauthorized* data is retrieved and posted, with any exception occurring only subsequently. In the middle path, authorized data is retrieved and posted, with no pending exception. In the bottom path, the exception occurs before the unauthorized data can be retrieved, and what is posted is a default value of 0.

A second important use for composition in *DECIMAL* is defining the flow of information and control across abstraction layers. In Figure 2, we showed a simple approach to mapping across the ISA/CPU boundary, which precludes any out-order, or even pipelined, execution in the CPU. In contrast, Figure 6 shows an abstracted model of CPU operations related to Meltdown in a Load/Store architecture. The transitions labeled `post` represent information being exchanged between instructions, and are synchronized as part of composing the model. The light blue arrows show synchronized transitions between automata representing the three different instructions. Those synchronized transitions

LOAD-REGISTER-INDRECT(DR,ISR) - template to load a value from memory using indirect register addressing. Uses timing and concurrent "process" to check address legality.

- DR - Destination register. The register to store the value loaded from memory
- ISR - Indirect source register. The register contains the memory address to be loaded.

Constants

- CT - Memory address legality check time
- CRT - Cache (L1) read time
- MRT - Memory read time

When CRT < CT < MRT this models Intel behavior:
cacheHit -> post(Rx, mem[addr]) (regardless of legality)
cacheMiss -> post(Rx, if legal(addr) mem[addr] else 0)

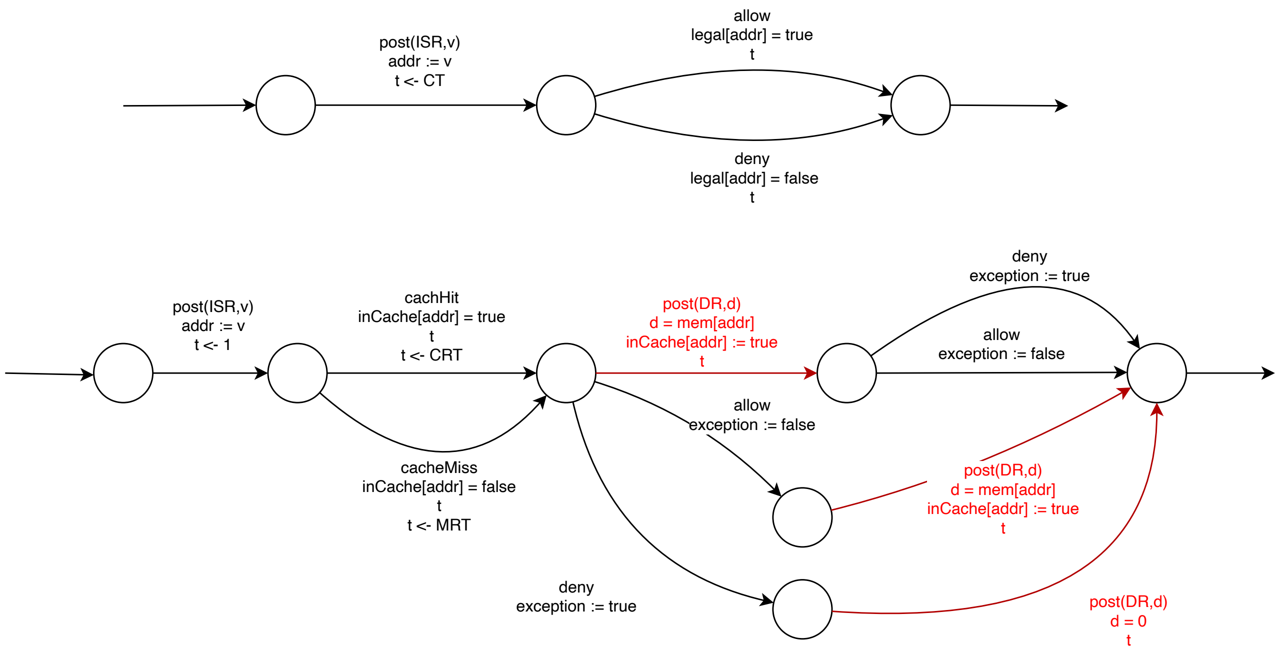


Figure 5: Load register with indirect reference

represent information passing from the result of one action, stored in a register, to a subsequent action. The branching paths in the first and third automata model different execution paths taken depending on whether the data being fetched from memory is currently in cache or not. Finally, the `commit` transitions are the events that make the results of each instruction visible at the ISA level. The hollow red arrow shows the temporal relationship that determines whether Meltdown can succeed. Can the illegally-accessed information be encoded in the cache before the illegal access is signalled, rolling back the entire speculative branch?

Now, suppose that these instructions are being executed in a CPU that enforces in-order dispatch and commitment, such as the Ariane RISCV-64 architecture (Zaruba and Benini 2018). Figure 7 shows a refinement of the model in Figure 6, adding explicit dispatch and commit operations in the micro-architecture. The program instruction stream shown at the upper left in Figure 7 is now four instructions, rather than three, making explicit the load/store operations required. In the same figure, the four automata shown are instances of those four instructions.

The two automata in Figure 8 will, when composed with the automata in Figure 7, enforce constraints on in-order instruction dispatch and in-order commitment of instruction results. The result is shown in Figure 9. There are no other ordering restrictions on instruction execution, other than those imposed by data dependencies.

Finally, the top abstraction layer in the system model may as previously discussed encode one or more programs, represented as paths through the automaton. We can use composition as a means to reduce the size of the space of possible programs, as well as to organize that space in ways that a human user may find more congenial. The basic technique is essentially abstraction as defined in the Abstraction section, but applied in a limited and specialized way. The approach borrows from research in *hierarchical planning*, which typically involves decomposing abstract actions into sequences of more detailed ones. The motivating insight: how one such abstract action is decomposed may be largely or entirely independent of how other actions at the same or lower levels of abstraction are decomposed. This composition becomes difficult only when interactions among decomposition choices are manifold and complex, such that choices made in one part of the plan constrain many other choices, directly or in-

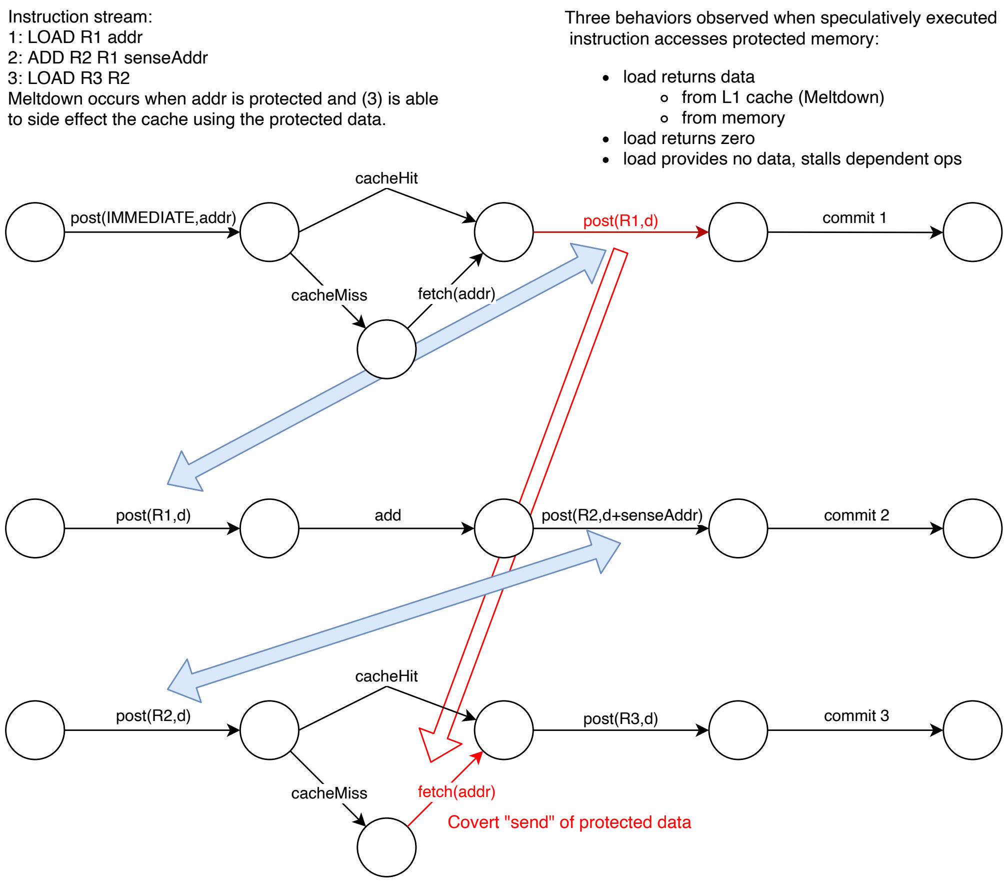


Figure 6: Abstract Meltdown model

directly.[2]

For example, reading information out of the micro-state can be done in many different ways. Two of the more common are `Flush+Reload`, and `Prime+Probe`. Either one of these techniques can be combined with a wide variety of speculative execution race conditions in order to form an attack. The overall structure of any of the resulting attacks is

1. Set up the timing channel (e.g., flush cache).
2. Perform the speculative exploit, resulting in unauthorized information present in the micro-architecture.
3. Encode that information, e.g., by referencing one of an array of 256 page-sized memory blocks.
4. Time memory accesses to those same addresses, revealing the secret information.

This sequence is independent of details of how the steps are accomplished, though there are some inter-step dependencies. For example, the details of steps 1, 3, and 4 must be consistent. `Flush` plus the appropriate encoding in step 3 sets up the conditions for `Reload`. `Prime` plus a different encoding in step 3 sets up the conditions for `Probe`.

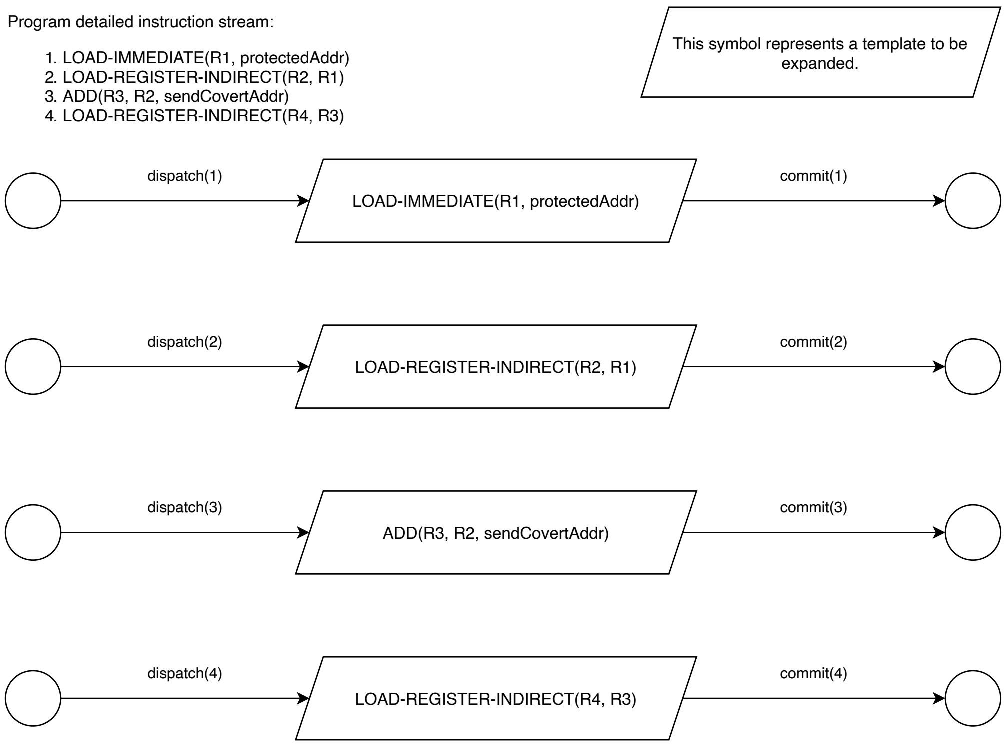
[2]An (inexact) analogy might be made to Bellman's principle of optimality and dynamic programming.

Figure 10 shows a simple hierarchical structure for this example. `Read_Secret` can be accomplished in one of two ways, using either `Flush+Reload` or `Prime+Probe` wrapped around a speculative memory access. That speculative access can be accomplished in one of three ways, here simplified to the names of the attacks in which those methods are employed. We have thus encoded six different possible exploits. We can compose across the different levels in this hierarchical representation, tying abstract task to component subtasks in composed models by synchronizing transitions in essentially the same patterns as across the abstraction boundary in Figure 2.

In addition to supporting a more generic and flexible representation of classes of emergent execution as shown above, this mechanism can be used for other forms of generalization as well. For example, in our earlier case studies we demonstrated that with only small, localized differences, the same basic exploit can be modeled across multiple, significantly different architectures.

Summary and Conclusion

In this Work in Progress report, we have described ongoing research on our *DECIMAL* project, addressing the problem of modeling computational mechanisms at sufficient fidelity



Program detailed instruction stream:
1. LOAD-IMMEDIATE(R1, protectedAddr)
2. LOAD-REGISTER-INDIRECT(R2, R1)
3. ADD(R3, R2, sendCovertAddr)
4. LOAD-REGISTER-INDIRECT(R4, R3)
This symbol represents a template to be expanded.
dispatch(1)
LOAD-IMMEDIATE(R1, protectedAddr)
commit(1)
dispatch(2)
LOAD-REGISTER-INDIRECT(R2, R1)
commit(2)
dispatch(3)
ADD(R3, R2, sendCovertAddr)
commit(3)
dispatch(4)
LOAD-REGISTER-INDIRECT(R4, R3)
commit(4)

Figure 7: Automata corresponding to individual instructions

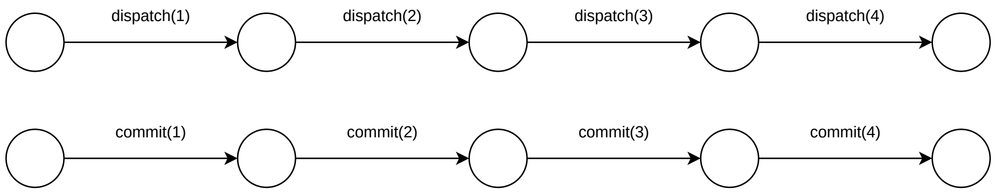


Figure 8: Automata enforcing in-order dispatch and commit

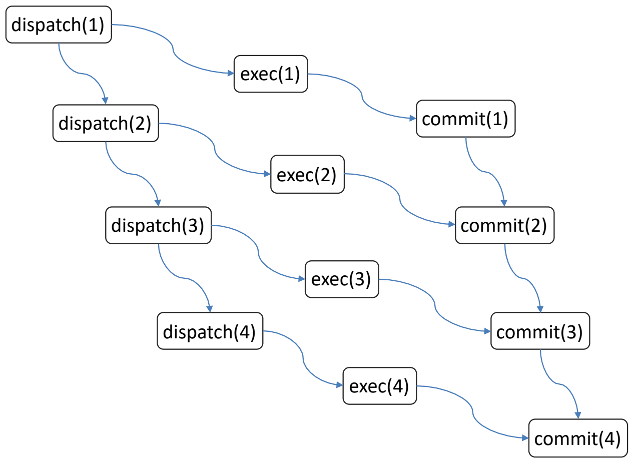


Figure 9: Composition of automata in Figures 7 and 8

to reason about the execution semantics of programs across abstraction boundaries. The models and tools we have developed are somewhere between proof-of-concept and prototype stage. Our focus to date has been more on understanding the issues and tradeoffs involved, than on developing a fully-functional set of tools.

The automata-based formalism that we have developed is specifically constructed to support reasoning about timed behavior over compositions of multiple component automata, modeling different parts of the system under study. Composition is used to model *abstraction*, such as between different layers in a hierarchy, as well as to represent the possibly-asynchronous execution of multiple processes within a given abstraction level.

Our main achievements on this project to date include:

- Defining a rigorous syntax and semantics for the *DECIMAL* formalism, available on request to the research community.
- Implementing tools for automaton composition and trans-

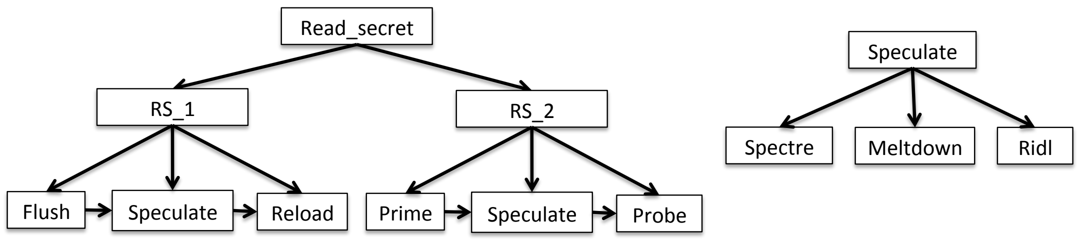


Figure 10: Hierarchical representation of a range of possible attacks.

lation to SMV.

- Conducting a set of case studies, focused on emergent computation across the ISA/CPU abstraction boundary.
- Demonstrating explicit representation for asynchronous processes, pipelining, and race conditions in the micro-architecture.
- Defining conventions for using automaton composition to generalize system models across different kinds of speculative exploits, and across different CPU architectures.
- Formulating and demonstrating automated checking of a general property indicating the potential for speculative information disclosure.

The case studies referred to above provide evidence that *DECIMAL* generalizes nicely across both exploits and architectures. We can represent ISA-level instructions and compose them with different automata, reflecting differing CPU implementations. We can model an exploit such as Meltdown at an abstract level, for example leaving unspecified the precise timing channel to be used to extract information from the micro-architecture. We can further extract, abstract, and model common elements across functionally different exploits, such as Spectre and ExSpectre, or Spectre and Meltdown.

One of the important questions remaining is how to determine the appropriate level of the system implementation at which to ground out our models. It almost certainly does not make sense to model details down to the atomic level, reasoning about the quantum properties of semiconductors. At the same time, a model that is just detailed enough to permit one to represent a known phenomenon is no more useful than a scientific experiment that is designed (and sometimes claimed) to "predict" such a phenomenon.

In future work, we plan to extend these results in several directions. One such direction is further investigation of how details of system architecture determine execution semantics. For example, Ariane and x86 CPUs both support some form of out-of-order execution, but the specific details of exactly what they do and how are quite different. How should those differences be reflected in our models? What differences result, in the inferences that the respective models support?

Longer-term, we are also interested in extending this work to qualitatively different applications. For example, multi-layer communication protocols involve a quite different notion of abstraction from that found at the ISA/CPU boundary in modern computer systems. In some cases, those abstractions nonetheless appear to be similarly imperfectly isolated, such that communication streams that are correct according to the individual requirements for different protocol levels may still be collectively problematic.

References

[Aarts and Vaandrager 2010] Aarts, F., and Vaandrager, F. 2010. Learning i/o automata. In *International Conference on Concurrency Theory*, 71–85. Springer.

[Cledou, Proença, and Soares 2017] Cledou, G.; Proença, J.; and Soares, B. L. 2017. Composing families of timed automata. In Dastani, M., and Sirjani, M., eds., *Fundamentals of Software Engineering*, Lecture Notes in Computer Science. Springer.

[de Alfaro and Henzinger 2001] de Alfaro, L., and Henzinger, T. 2001. Interface automata. *SIGSOFT Software Engineering Notes* 26:109—120.

[Dullien 2017] Dullien, T. 2017. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing*.

[D'Antoni et al. 2019] D'Antoni, L.; Ferreira, T.; Sammartino, M.; and Silva, A. 2019. Symbolic register automata. *CoRR* abs/1811.06968.

[Jonsson and Vaandrager 2017] Jonsson, B., and Vaandrager, F. 2017. Learning mealy machines with timers. In *IPA Fall Days, Nunspeet*.

[Kiriansky and Waldspurger 2018] Kiriansky, V., and Waldspurger, C. 2018. Speculative buffer overflows: Attacks and defenses. arXiv. https://arxiv.org/pdf/1807.03757.pdf.

[Lipp et al. 2018] Lipp, M.; Schwarz, M.; Gruss, D.; Prescher, T.; Haas, W.; Fogh, A.; Horn, J.; Mangard, S.; Kocher, P.; Genkin, D.; Yarom, Y.; and Hamburg, M. 2018. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 973–990. Baltimore, MD: USENIX Association.

[Mcilroy et al. 2019] Mcilroy, R.; Sevcik, J.; Tebbi, T.; Titzer, B. L.; and Verwaest, T. 2019. Spectre is here to stay: An analysis of side-channels and speculative execution. *ArXiv* abs/1902.05178.

[Mutlu and Kim 2019] Mutlu, O., and Kim, J. S. 2019. Rowhammer: A retrospective. *ArXiv*.

[Van Schaik et al. 2019] Van Schaik, S.; Milburn, A.; Osterlund, S.; Frigo, P.; Maisuradze, G.; Razavi, K.; Bos, H.; and Giuffrida, C. 2019. Ridl: Rogue in-flight data load. In *Proceedings - 2019 IEEE Symposium on Security and Privacy, SP 2019*, Proceedings - IEEE Symposium on Security and Privacy, 88–105. United States: Institute of Electrical and Electronics Engineers Inc.

[Wampler, Martiny, and Wustrow 2019] Wampler, J.; Martiny, I.; and Wustrow, E. 2019. Exspectre: Hiding malware in speculative execution. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society.

[Zaruba and Benini 2018] Zaruba, F., and Benini, L. 2018. Ariane: An open-source 64-bit risc-v application-class processor and latest improvements. In *8th RISC-V Workshop*. Integrated Systems Laboratory ETH Zurich. https://riscv.org//wp-content/uploads/2018/05/14.15-14.40-FlorianZaruba_riscv_workshop-1.pdf.