Synthesizing Evidence of Emergent Computation

Extended Abstract

Scott Moore
Galois, Inc.
scott@galois.com

Jennifer Paykin
Galois, Inc.
jpaykin@galois.com

Olivier Savary Bélanger
Galois, Inc.
olivier@galois.com

Proving a compiler to be secure is costly, and even testing its security often requires domain knowledge about potential attacks. In this work, we present a tool to synthesize examples of compiler insecurity from just specifications of the languages and the compiler between them. The tool also provides evidence of how severe discovered vulnerabilities may be by synthesizing gadgets that represent composable exploitation APIs within implementations.

The *SEEC* tool

In previous work [3], we connected the task of reasoning about secure compilation to the search for "emergent computations" and "weird machines" [1, 2] that demonstrate the exploitability of a system. Building on those ideas, we developed the *SEEC* modeling and synthesis framework to automatically generate witnesses of compiler insecurity and reason about their exploitability. The framework is built on top of Rosette [5], a Racket dialect for building solver-integrated programming languages that exposes symbolic execution and synthesis capabilities. The *SEEC* framework consists of two parts: (1) a modeling language for describing both specifications and implementations of software systems; and (2) a set of synthesis tools that produce comprehensible evidence of emergent computation, if it exists.

A *SEEC* model begins with a user-defined *grammar* of terms expressed in BNF notation, from which expressions, contexts and behaviors will be taken. *SEEC* provides a number of built-in data structures including booleans, integers, strings, bitvectors and lists of other nonterminals. For example, `set-api` defines a grammar for sets implemented as list of integers.

```
(define-grammar set-api
  (value  ::= integer boolean)
  (trace  ::= list<value>)
  (set    ::= list<integer>)
  (method ::= (insert integer) (remove integer)
              (member? integer) select)
  (interaction ::= list<method>))
```

The semantics of the language is then defined as a Racket function interpreting *whole programs* into *behaviors*.

The *SEEC* construct `define-language` combines these pieces together to create a *SEEC* language. The `#:size` modifier specifies a default upper-bound on the size of abstract expressions, and the `#:where` clause allows the user to specify a well-formedness predicate. For example, `abstract` is a *SEEC* language using Racket function `abstract-interpret` that takes as input a pair of a `set` and an `interaction` and returns a list of `value`:

```
(define (abstract-insert s v)
  (set-api (,v ,(abstract-remove s v))))
...
(define (abstract-interpret interaction state)
  (match interaction
    [(set (cons (insert v:value) r:interaction))
         (abstract-interpret r (abstract-insert state v))]
...
(define-language abstract
  #:grammar set-api
  #:expression interaction #:size 4
  #:context    set         #:size 2
  #:link cons
  #:evaluate abstract-interpret)
```

Two *SEEC languages* can be connected via a *SEEC compiler*, which consists of a Racket function from programs in the source language to programs in the target language and two relations that characterize an equality relation between the behaviors (the results of the interpretation functions) of the two languages and their contexts. For example, with a second language `concrete` representing an implementation of sets where some operations have a more concrete implementation, we can define a compiler between `abstract` and `concrete`:

```
(define-compiler abstract-to-concrete
  #:source abstract
  #:target concrete
  #:behavior-relation equal?
  #:context-relation equal?
  #:compile id)
```

Discovering Vulnerabilities

An *exploit* of a vulnerable source program V is a target attacker context $A \in \mathcal{A}$, where $\mathcal{A}$ corresponds to the sorts of attacks being considered, such that there is no source context C^S with $B(C^S[V]) \mapsto B(A[V\downarrow])$.

Given a compiler and a source expression V, the query `find-weird-behavior` synthesizes a target context A which is an exploit of V.

`find-weird-behavior` is able to find bugs in `concrete` such as an implementation of `remove` that only removes the first occurrence of an element from the list.

PRISC'22, ,
2022.

This query universally quantifies over all source language contexts using Rosette's counter-example guided inductive synthesis [4] capabilities to efficiently find example emergent computations.

A variation of this query, `find-changed-behavior`, searches instead for source contexts that, when compiled, give rise to different results in the target language than in the source language. While this query may not identify as many vulnerabilities as `find-weird-behavior`, the fact that it does not have any universal quantifiers means that it is computationally easier for the solver. We often have found that `find-changed-behavior` is strong enough to identify many bugs on its own before checking our results with a call to `find-weird-behavior`.

`find-changed-behavior` would be able to discover a change of behavior between an `abstract` implementation of `select` which picks a random element from the list, and a `concrete` one that picks the first element.

When a vulnerability is discovered by *SEEC*, the user has multiple options for how to proceed:

1. refining the source model, which usually corresponds to correcting the specification;
2. changing the target model, which could correspond to fixing a bug, introducing a mitigation, or encoding more details in order to better capture a certain aspects of the underlying implementation;
3. changing the equivalence relation between source and target behaviors.

SEEC can then be used to verify that the mitigation successfully eliminated the weird behavior.

Synthesizing Composable Gadgets

There are times at which it is not cost effective to implement security mitigations, either due to the cost of designing, implementing, or running the mitigations in a real system. In these cases, it is useful to have an understanding of how dangerous a particular vulnerability is before deciding how to deal with it.

`find-gadget` is a built-in query that will find an expression $\mathbf{G}$ that satisfies a specification $\mathcal{S}$ in all contexts:

$$\forall \mathbf{C}.\, \mathcal{S}(\mathbf{C}[\mathbf{G}], B(\mathbf{C}[\mathbf{G}]))$$

In `concrete`, where `insert` is implemented by adding the element as the head of the list, we can synthesize a gadget with a specification where the resulting list is one element longer than the initial one with a single call to `insert`.

The offensive security community has observed that the most dangerous exploits stem are those that enable highly composable or programmable emergent computations [2]. To help assess the severity of compiler insecurity, we find evidence of programmable emergent computations by looking for multiple gadgets that can act on a shared data representation. These gadgets are synthesized at the same time as a *decoder*, mapping information from the program state to a chosen data structure. The technique of synthesizing multiple programs from a specification that describes how they relate to each other is called *relational synthesis* [6].

Synthesizing a *decoder* rather than using a fixed specification allows us to consider any possible encoding that could be used by an attacker. The construct `define-attack` represents the capabilities of an attacker to observe and interact with a system modeled using a *SEEC* language. As the decoder is used to chain multiple gadgets together, it also determines the computational capabilities afforded to the attacker.

```
(define-attack dec-while-int
  #:grammar set-api
  #:gadget interaction #:size 3
  #:evaluate-gadget interpret-interaction
  #:decoder dec-while #:size 4
  #:evaluate-decoder interpret-dec-while)
```

The query `find-related-gadgets` take as input a *SEEC* language, a list of specification functions and an attack model. It returns a decoder $\mathbf{D}$ and gadgets $\mathbf{G}_{f_1}, ..., \mathbf{G}_{f_n}$ such that such that, for each specification function f_i,

$$\forall \mathbf{C}.\, \mathbf{D}(\mathbf{C}[\mathbf{G}_{f_i}]) = f_i(\mathbf{D}(\mathbf{C}))$$

Gadgets satisfying this equation are composable with each other with respect to the decoder, such that they implement the programmable space of operations f_i.

Using `find-related-gadgets` on `dec-while-int` and looking for an encoding of natural numbers with operations $+1$ and -1, *SEEC* synthesizes a conflict-free replicated data type, with a decoder subtracting the number of occurrences of 1 from the occurrences of 0, and gadgets `insert 0` and `insert 1`.

Results and Future Work

The results of our exploration have been highlighted with four key case studies—a heap allocator, an API model, an implementation of `printf` format strings, and a model of C and assembly programs. These case studies highlight the range of expressive synthesis queries whose results can be obtained by synthesis. In addition, they show that having an AI oracle for determining unexpected behaviors is extremely powerful.

In the future, we will investigate combining multiple techniques including synthesis, fuzzing, and other machine learning approaches to scale up analysis of the theoretical frameworks on which *SEEC* is built. We plan to use fuzzing or random testing to quickly search for examples of insecure compilation and, if no such examples are found, use symbolic execution to verify that no such examples exist.

Acknowledgments

This material is based upon work supported by the United States Air Force and DARPA under Contract No. FA8750-15-C-0124. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force and DARPA.

References

[1] Sergey Bratus, Michael Locasto, Meredith Patterson, Len Sassaman, and Anna Shubina. 2011. Exploit Programming: from Buffer Overflows to Weird Machines and Theory of Computation. *USENIX* (Dec. 2011).

[2] Sergey Bratus and Anna Shubina. 2017. Exploitation as code reuse: On the need of formalization. *Information Technology* 50, 2 (2017). https://doi.org/10.1515/itit-2016-0038

[3] Jennifer Paykin, Eric Mertens, Mark Tullsen, Luke Maurer, Benoît Razet, Alexander Bakst, and Scott Moore. 2019. Weird Machines as Insecure Compilation. *arXiv:1911.00157 [CS]* (Oct. 2019).

[4] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. 404–415.

[5] Emina Torlak and Rastislav Bodik. 2013. Growing solver-aided languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. Association for Computing Machinery, Indianapolis, Indiana, USA, 135–152. https://doi.org/10.1145/2509578.2509586

[6] Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2018. Relational Program Synthesis. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 155 (Oct. 2018), 27 pages. https://doi.org/10.1145/3276525