Mitigating Emergent Computation: the need for new approaches in systems engineering

Sergey Bratus
DARPA
Information Innovation Office (I2O)

HCSS 2021

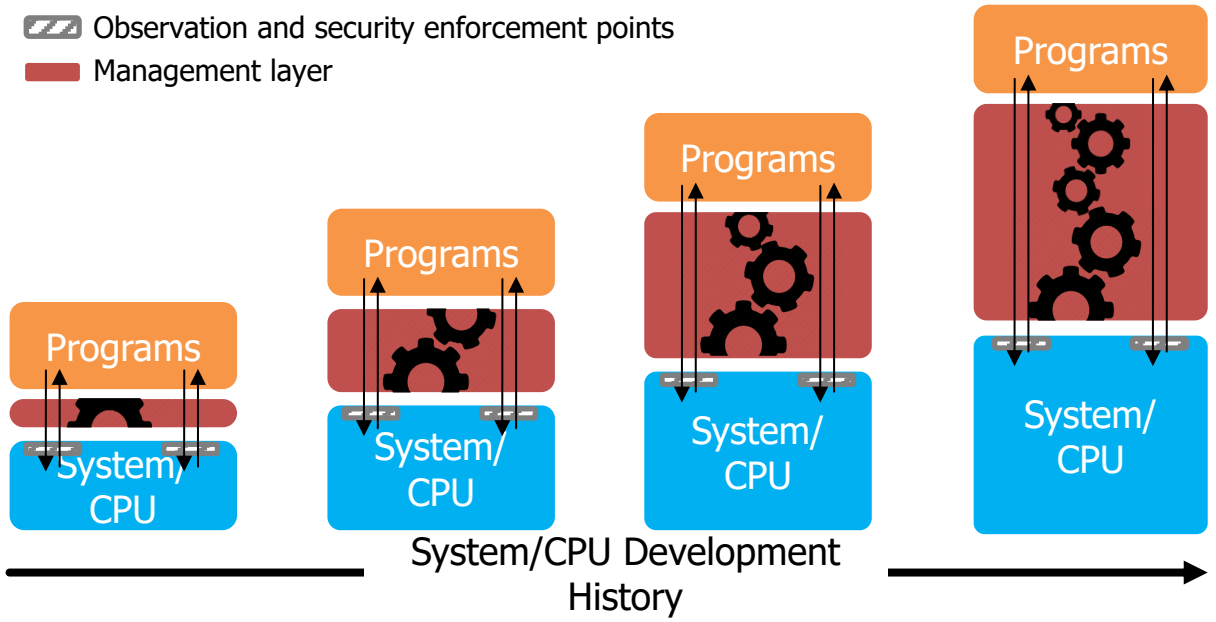
May 2021

What is Emergent Computation and why we care?

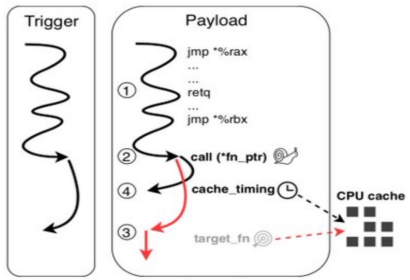
Emergent Computation

- Modern computing systems demonstrate strong propensity for unintended, emergent computations and the related unintended, emergent programming models that **enable or amplify cyber-attacks**

- Computing mechanisms built for a particular purpose and with particular intended models of execution prove to be capable of executing **unintended computing tasks** outside of their original specification and their designers and programmers' mental models



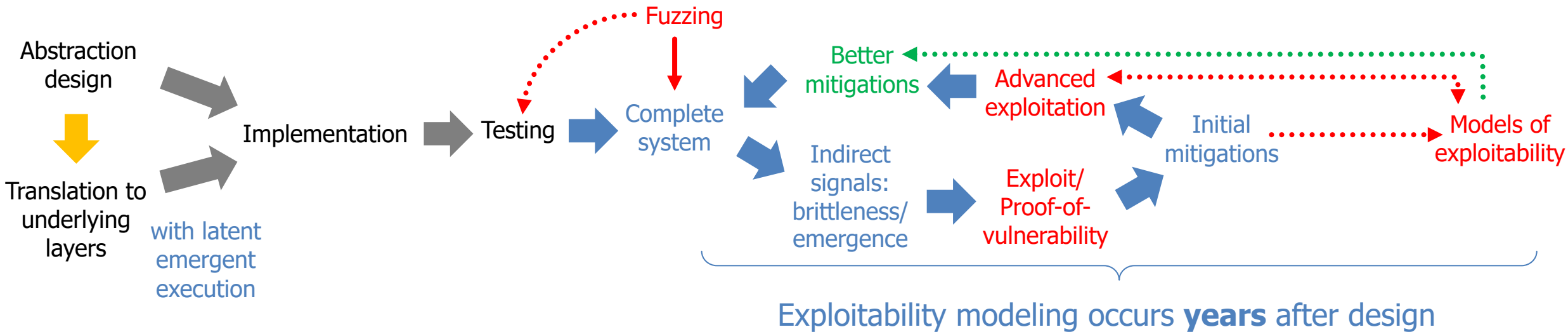
Spectre, Meltdown, Zombieload, MDS leaks, Foreshadow, …



ExSpectre: Hiding Malware in Speculative Execution, Wampler et al., NDSS '19

Computing with time: microarchitectural weird machines, Evtyushkin et al., ASPLOS '21

Emergent computation, abstractions, and the SDLC



- We start examining systems for signs of emergent behavior—with methods such as fuzz-testing—only after they are fully built
- However, a system's exploitability models and propensity for emergent execution arise—and can also therefore be mitigated—already at the **design stage**
  - *Spectre is here to stay: An analysis of side-channels and speculative execution*, Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L. Titzer, Toon Verwaest, https://arxiv.org/abs/1902.05178, 2019
  - *Exploit Programming: from Buffer Overflows to Weird Machines and Theory of Computation*, Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, Anna Shubina, USENIX ;login:, 2011

Examples of unintended emergent programmability ("weird machines")

- *Weird Machines in ELF: A Spotlight on the Underappreciated Metadata*, Shapiro et al., USENIX WOOT '13
  - GNU/Linux runtime dynamic linker-loader can be generically programmed with ELF relocation and symbol metadata
- *Exploiting the Hard-Working DWARF: Trojan and Exploit Techniques with No Native Executable Code*, Oakley et al., USENIX WOOT '11
  - GNU C++ DWARF exception handling mechanism can be generically programmed with contents of *eh_frame*
- *The Page-Fault Weird Machine: Lessons in Instruction-less Computation*, Bangert et al., USENIX WOOT '13
  - x86 MMU can be generically programmed with the contents of CPU's descriptor tables (GDT, LDT, IDT, and TSS)
- *Framing Signals - A Return to Portable Shellcode*, Bosman et al., IEEE S&P '14
  - Unix signal handling can be generically programmed with fake signal frames
- *Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector,* Bosman et al., IEEE S&P '16
  - Windows 8.1-10 built-in memory deduplication feature combined with RowHammer yields a powerful weird machine
- *Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications,* Schuster et al., IEEE S&P 2015
  - Contents of OOP objects' v-tables allow generic programming similar to return-oriented programming

Beginnings of formalism: making sense of "weird machines"

- *The Weird Machines in Proof-Carrying Code*, Julien Vanegue, 1st IEEE S&P Language-theoretic Security (LangSec) Workshop, 2014
  - Non-foundational PCC still admits emergent behaviors when called upon outside of the proof's preconditions
- *Weird machines, exploitability, and provable unexploitability*, Thomas Dullien, IEEE Transactions on Emerging Topics in Computing, December 2017
  - Intended Finite State Machine vs an implementation admitting extra "weird" states and transitions between them
- *Exploitation as Code Reuse: On the Need of Formalization*, Sergey Bratus et al, Information Technology, vol. 59, no. 2, 2017
  - Exploitation programming always violates one abstraction but fully obeys another, a lower one
- *Weird Machines as Insecure Compilation*, Jennifer Paykin et al., 2019, https://arxiv.org/abs/1911.00157
  - Emergent execution is modeled as violations of the 'full abstraction' property of compilation

More to come!

Beyond specification: Countering emergent execution with system and data design

DARPA High Assurance Cyber Military Systems (HACMS)

militaryaerospace.com/computers/article/16711582/cyberde...

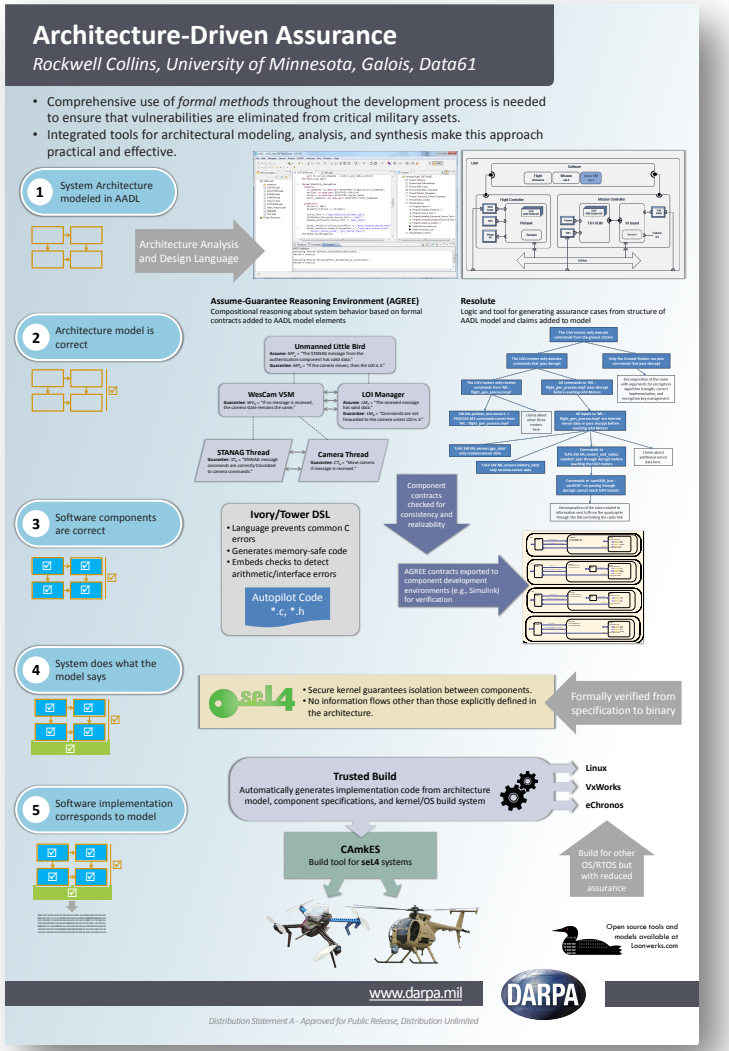
Military&Aerospace Electronics LOG IN REGISTER

Cyber-defense software in the real world: the High-Assurance Cyber Military Systems (HACMS) program

MESA, Ariz. – In February 2017, a helicopter took off from a Boeing facility in Mesa, Ariz., on a routine mission around nearby hills. The aircraft and its software flew its course fully autonomously, and the safety pilot, required by the Federal Aviation Administration, did not touch any controls during the flight.

The central mission computer was attacked by rogue camera software, as well as by a virus delivered through a compromised USB stick that had been inserted during maintenance. The attack compromised some subsystems but could not affect the safe operation of the aircraft.

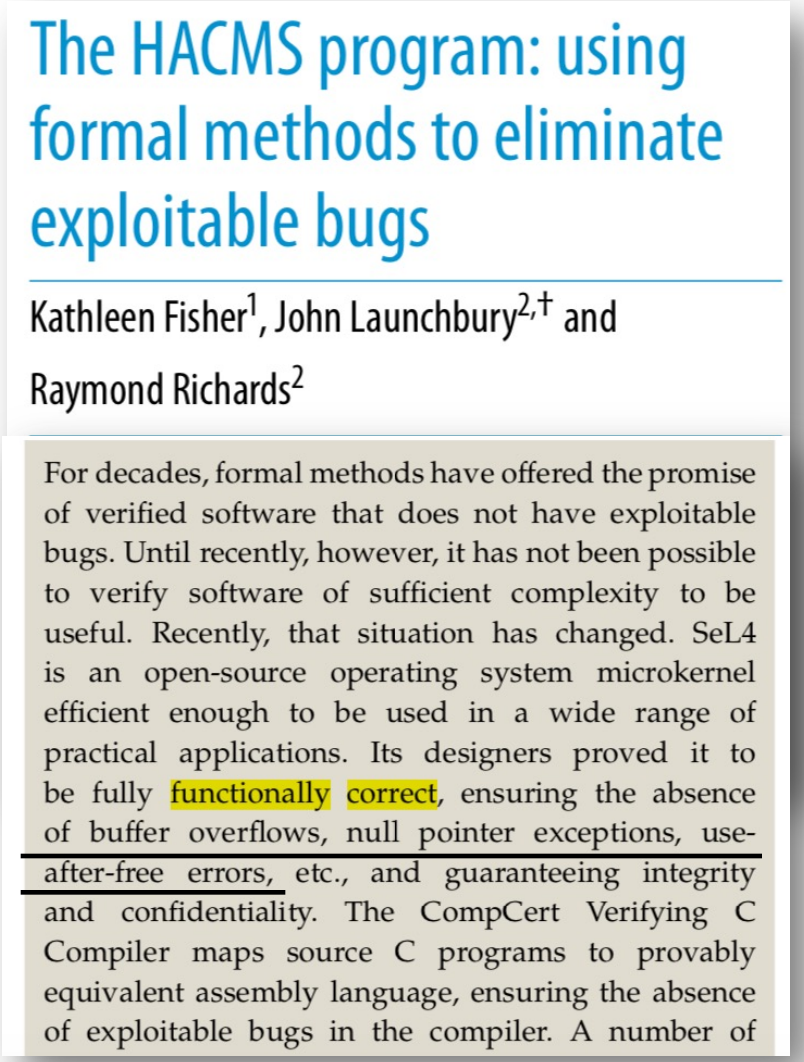
Source: https://www.militaryaerospace.com/computers/article/16711582/cyberdefense-software-in-the-real-world-the-highassurance-cyber-military-systems-hacms-program



Source: http://loonwerks.com/projects/hacms.html

Clean-slate, functionally correct secure software

- DARPA HACMS demonstrated that formal methods scaled to real systems of DoD relevance
  - Boeing Unmanned Little Bird (AH-6) with HACMS flight firmware proved resilient to in-flight cyber-attacks
- SeL4 microkernel is a triumph of proving functional correctness in a real system
  - So is CompCert, the only compiler to withstand CSmith's fuzzed C without crashing
- NSF DeepSpec extends functional correctness from a Coq/Gallina application spec to hardware
  - Edging out the primitives of emergent execution throughout the computing stack down to hardware
- **IF** software's intent is expressible in the spec, and the chain of proofs can be completed



The HACMS program: using formal methods to eliminate exploitable bugs

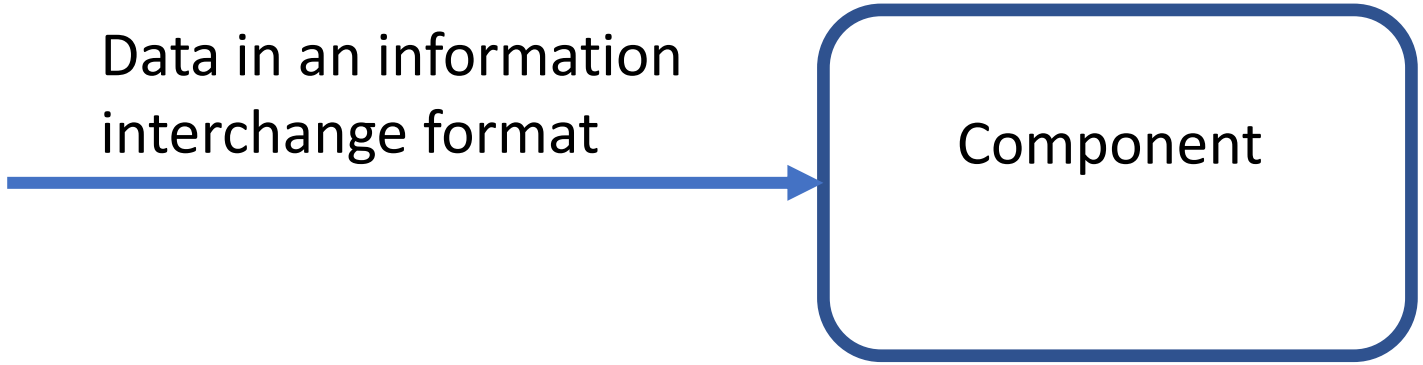
Kathleen Fisher[1], John Launchbury[2,†] and Raymond Richards[2]

For decades, formal methods have offered the promise of verified software that does not have exploitable bugs. Until recently, however, it has not been possible to verify software of sufficient complexity to be useful. Recently, that situation has changed. SeL4 is an open-source operating system microkernel efficient enough to be used in a wide range of practical applications. Its designers proved it to be fully functionally correct, ensuring the absence of buffer overflows, null pointer exceptions, use-after-free errors, etc., and guaranteeing integrity and confidentiality. The CompCert Verifying C Compiler maps source C programs to provably equivalent assembly language, ensuring the absence of exploitable bugs in the compiler. A number of

Source: https://royalsocietypublishing.org/doi/10.1098/rsta.2015.0401

Bringing verification insights to systems-of-systems design

Challenge: Give systems designers the tools that would prevent them from designing unverifiable or hard-to-verify systems



Designer's leap of faith: Intent is expressible

- *Designers:* Correct implementation will tell maliciously crafted inputs from valid inputs.
- *Also designers:* Inputs are really programs in rich bytecode or scripts.
- *Attackers:* Your security game is lost at the point of input format RFC!

This seems like a weird case, but it's ubiquitous:

- Inputs mean actions
  - Commands, memory allocations when constructing object representation, state changes, …
- Actions must abide by policy
- Policy must be computable (*) – what if implied policy isn't?

[*] Cf. K.Hamlen, G.Morrisett, F.Schneider, "Computability Classes for Enforcement Mechanisms", 2003

Designer's leap of faith: Intent is expressible cont.

- *Designers:* Correct implementation will tell maliciously crafted inputs from valid inputs.
- *Also designers:* Inputs are really programs in rich bytecode or scripts.
- *Attackers:* Your security game is lost at the point of input format RFC!

Systems engineers must design data interchange formats so that validity checking of inputs is *tractable*

Otherwise functional correctness properties relating inputs and outputs cannot be specified

Programmer's dilemma: To DWIM* or not to DWIM?

- *Customers:* Implement standard ISO xxxxx-v:2020 for input data.
- *Also customers:* Fix trivial errors and pre-standard variants in inputs
- *Attackers:* Let's see in how many ways your input validator and your input interpreter can be made to disagree**

This seems like another weird case, but it's ubiquitous***:

- Real data has dialects
  - Even when there's only the standard, reasonable implementors will disagree on corner cases
- There's enormous pressure to interoperate
- Input validation (and its specification!) now must include a **rewriting system**

(*) "Do What I Mean", correct trivial errors automatically
(**) On what actions your validator sees as allowed as per policy, but the executor interprets differently
(***) J.Chen, V.Paxson, J.Jiang, "Composition Kills: A Case Study of Email Sender Authentication", USENIX Security '20

Programmer's dilemma: To DWIM* or not to DWIM?

- *Customers:* Implement standard ISO xxxxx-v:2020 for input data.
- *Also customers:* Fix trivial errors and pre-standard variants in inputs
- *Attackers:* Let's see in how many ways your input validator and your input interpreter can be made to disagree**

Systems engineers need not only *unambiguous* and *tractable* definitions of data interchange formats, but also "what-if" tools for their changes

Inevitable spike of changes to proofs during system integration calls for "differentiable" proofs and proof tool chains

Protecting the system-of-systems designers

Challenge: Give systems designers the tools that would prevent them from designing an unverifiable or hard-to-verify system

SafeDocs

Data in an information exchange format

Component (Interpreter)

"Everything is an interpreter"
--Greg Morrisett

"Every input is a program"
--Language-theoretic security principle

Help designers avoid creating at-boundary validation problems that aren't specifiable, tractable, or provable.

Describe information interchange formats in systems-of-systems with suitable Data Definition Languages
that capture validity:

- Concepts
- Relationships
- Constraints

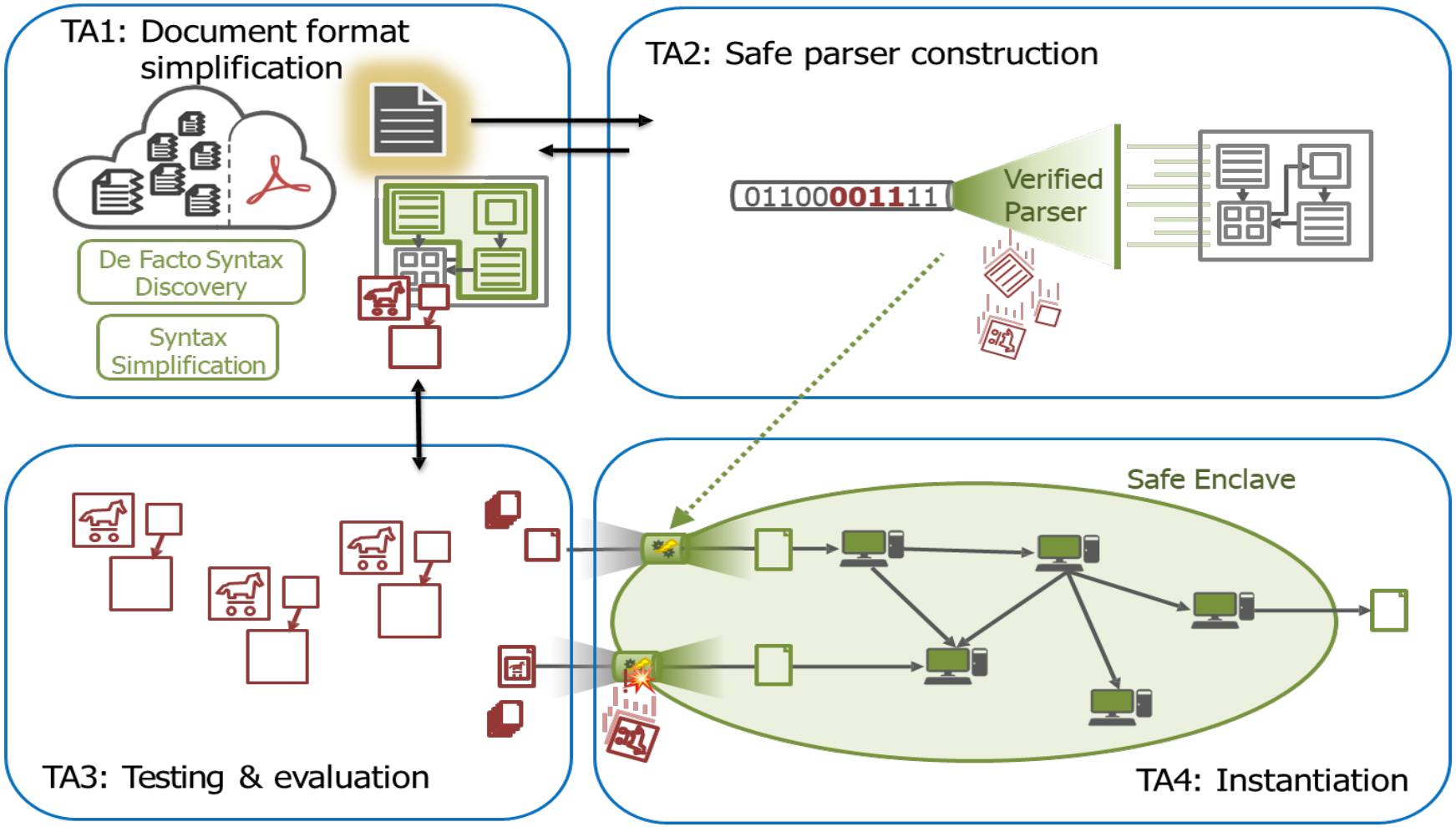
AIMEE

Anticipate and mitigate emergent execution models at system design time

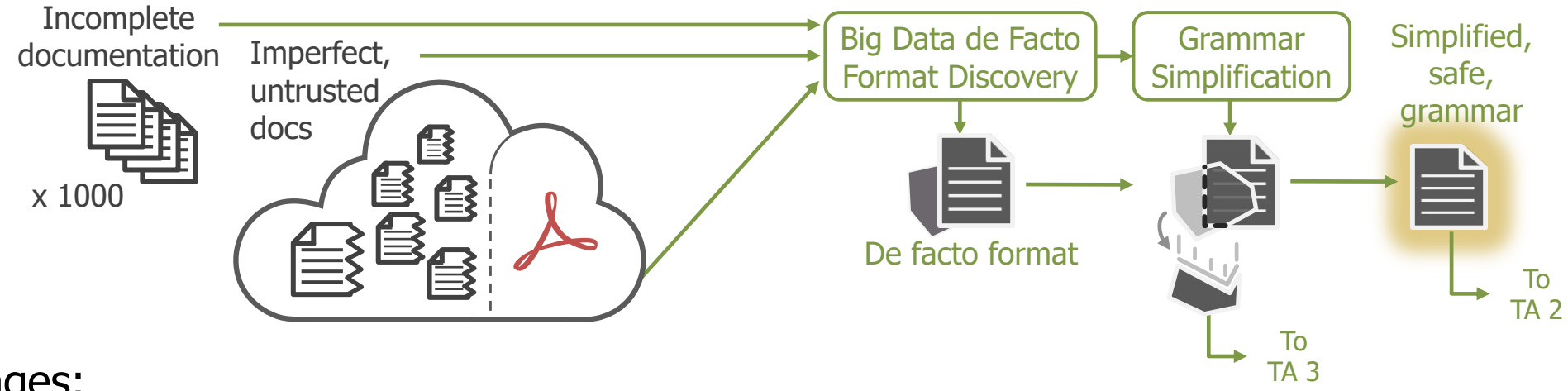
- If we consider inputs as bytecode, what kind of a VM would the component be?
- Do implementation models of component's abstractions allow emergent execution by design?

Safe Documents (SafeDocs)

Objective: Reduce electronic document complexity and build verified parsers to radically improve software's ability to reject invalid and malicious data



Unambiguously describe de facto data formats



Challenges:

- Lack of effective theory for describing actual complex electronic data formats
- Actual syntax includes many **ad hoc extensions** of recorded standards (on top of standards' own ambiguities)
  - We must discover 'benign' malformations in the wild, describe them intelligibly, and analyze them for assurance
- Current parsing theory is biased towards programming languages, not data formats (either binary or PDF-like)

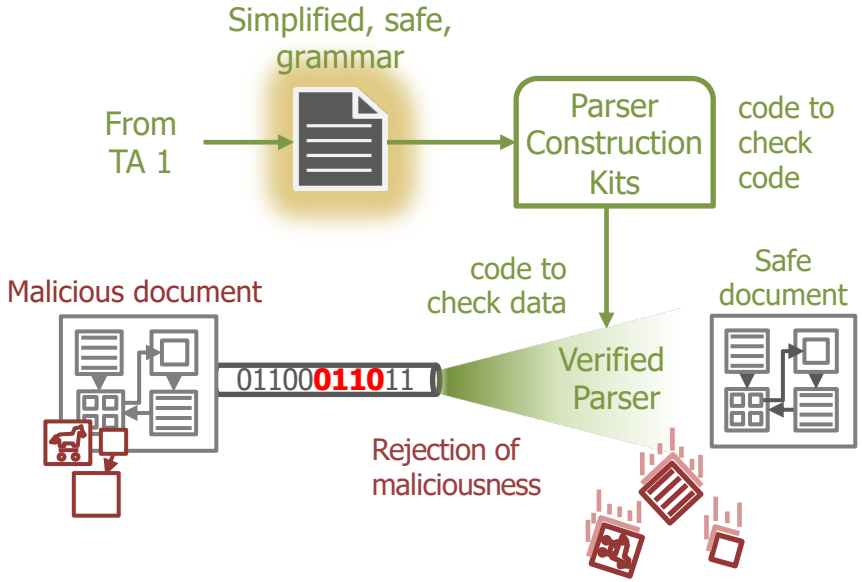
Approaches:

- Develop novel computer science theory to effectively describe **de facto** data formats
- Create formal ways to represent and reason about complex **logical dependencies** between format elements
- Create **unambiguous** ways to describe allowed variations and dialects of data formats in the wild
- Survey **large corpora** of openly posted documents to summarize use of features and malformations
- Create **machine-readable, human-intelligible** descriptions of data formats, deduce **safe format subsets**

Make safe parser construction a convenient default

Challenges:

- Lack of verification theory for parsers:
  - No program logics and type systems target parser development specifically
  - Data parsing algorithms and abstractions are not designed with verification in mind
- Verification-friendly parsing is beyond the common developer's reach
  - Demands unrealistic levels of mathematical expertise
  - Understanding of the date format is not convertible to declarative, verification-friendly programming idioms
- Verification is at odds with performance



Approaches:

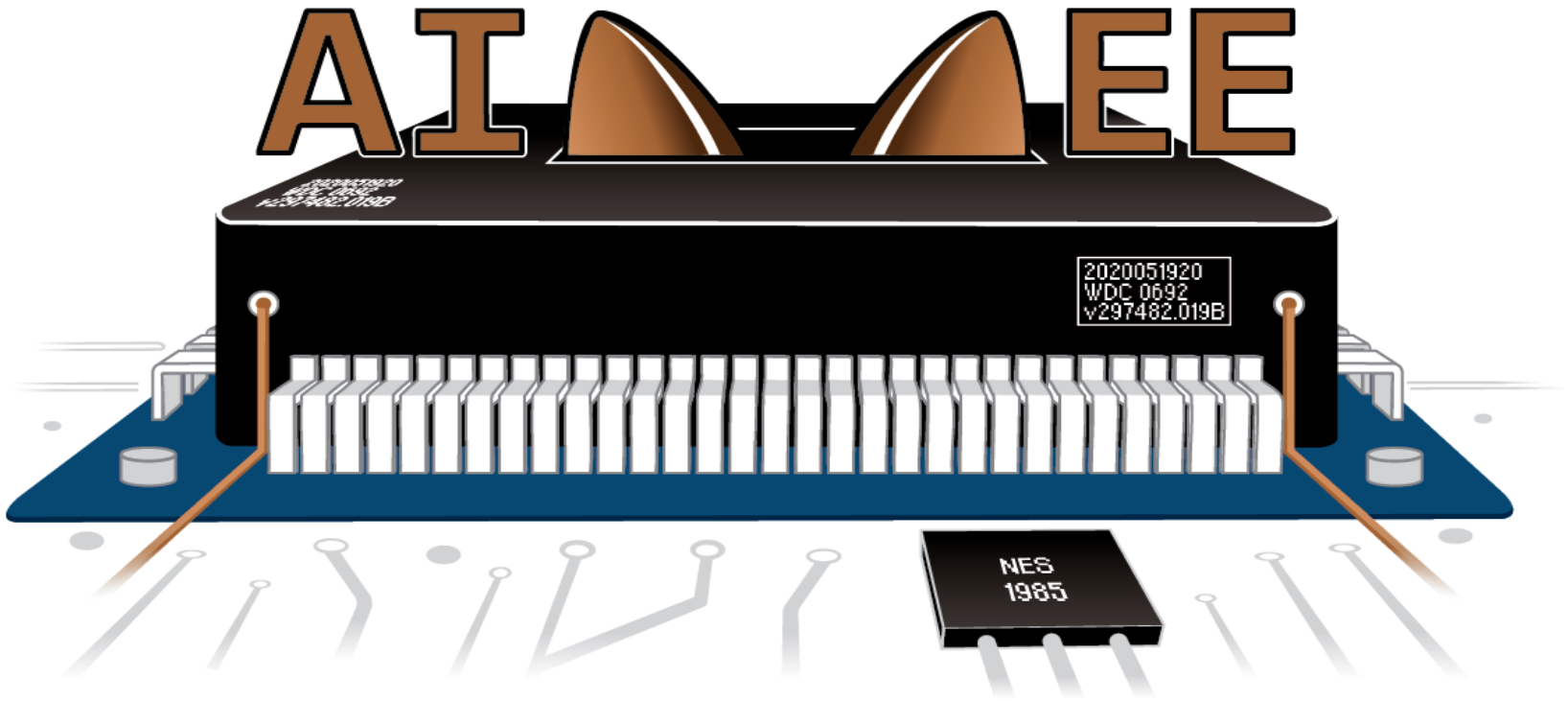
- New theories of parser functional correctness, logics for input data validation, type systems for documents and messages
  - Relational refinement **type systems** for data languages, parsers
  - **Program logics** for parsers in PVS, Coq, ACL2
  - Verification-oriented DSLs for parsers, with multi-language code extraction
- Usable **parser construction kits** and development tools for intuitive, verification-friendly development
  - Declarative programming styles that expose ambiguities of format specifications, enable format exploration

SafeDocs' contributions to document standards

- Submitted over 60 Candidate Edits to the ISO/FDIS 32000-2 (**PDF 2.0**) International Standard, removing vulnerability-producing ambiguities in the PDF format description
  - 50 Candidate Edits accepted into the standard, others under consideration
- Developed PDF Object Model grammar sets for every PDF version (1.0 through 2.0)
  - Every PDF version now has its own **machine-readable** Document Object Model specification
  - To be released at the 2021 IEEE S&P Language-theoretic Security (LangSec) workshop, May 27—28 (http://langsec.org/spw21/)
- Developing open-source tools for experts to explore the format in the wild and make value judgments on specific features and malformations
  - *Building a Wide Reach Corpus for Secure Parser Development,* Timothy Allison et al, 2020 IEEE S&P LangSec workshop, http://spw20.langsec.org/papers.html#corpus

AIMEE: Artificial Intelligence Mitigations of Emergent Execution

Objective: Examine systems for signs of emergent behavior: unintended computing tasks outside of their original specification and their designers and programmers' mental models



https://www.darpa.mil/program/artificial-intelligence-mitigations-of-emergent-execution

AIMEE use cases and models

Applying design-stage modeling to discover and mitigate emergent execution engines in:

- Program flow control abstractions
  - E.g., anticipating variations of *<control flow primitive>*-oriented programming
- Heap memory management logic
  - E.g., countering heap massaging, use-after-free, double-free, etc., and other manipulations of memory locality and adjacency
- Package management logic
  - E.g., countering manipulation of package managers via crafted packages (cf. Android Master Key bugs)
- Container management logic
  - E.g., countering manipulation of cloud orchestrators via crafted container images
- More to come!

Parthian shot: emergent execution in neural networks?

ADVERSARIAL REPROGRAMMING OF NEURAL NETWORKS

Gamaleldin F. Elsayed*
Google Brain
gamaleldin.elsayed@gmail.com

Ian Goodfellow
Google Brain
goodfellow@google.com

Jascha Sohl-Dickstein
Google Brain
jaschasd@google.com

https://arxiv.org/abs/1806.11146

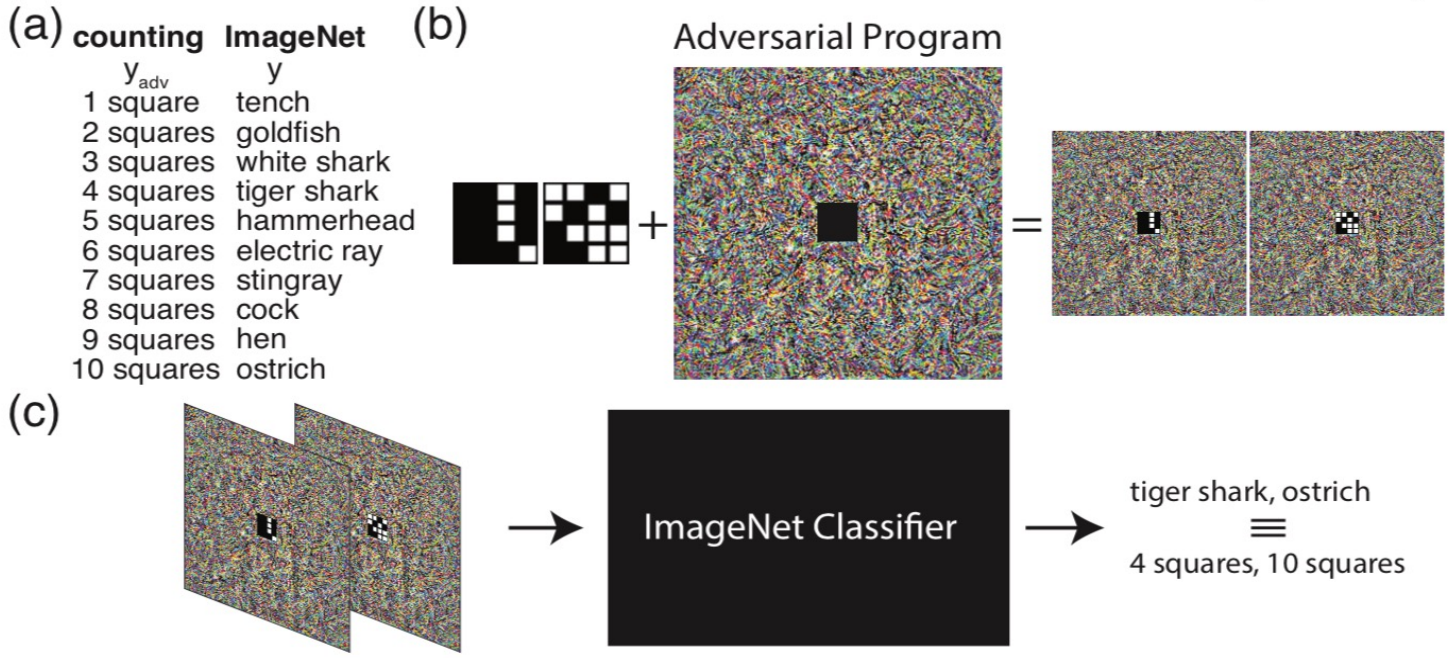
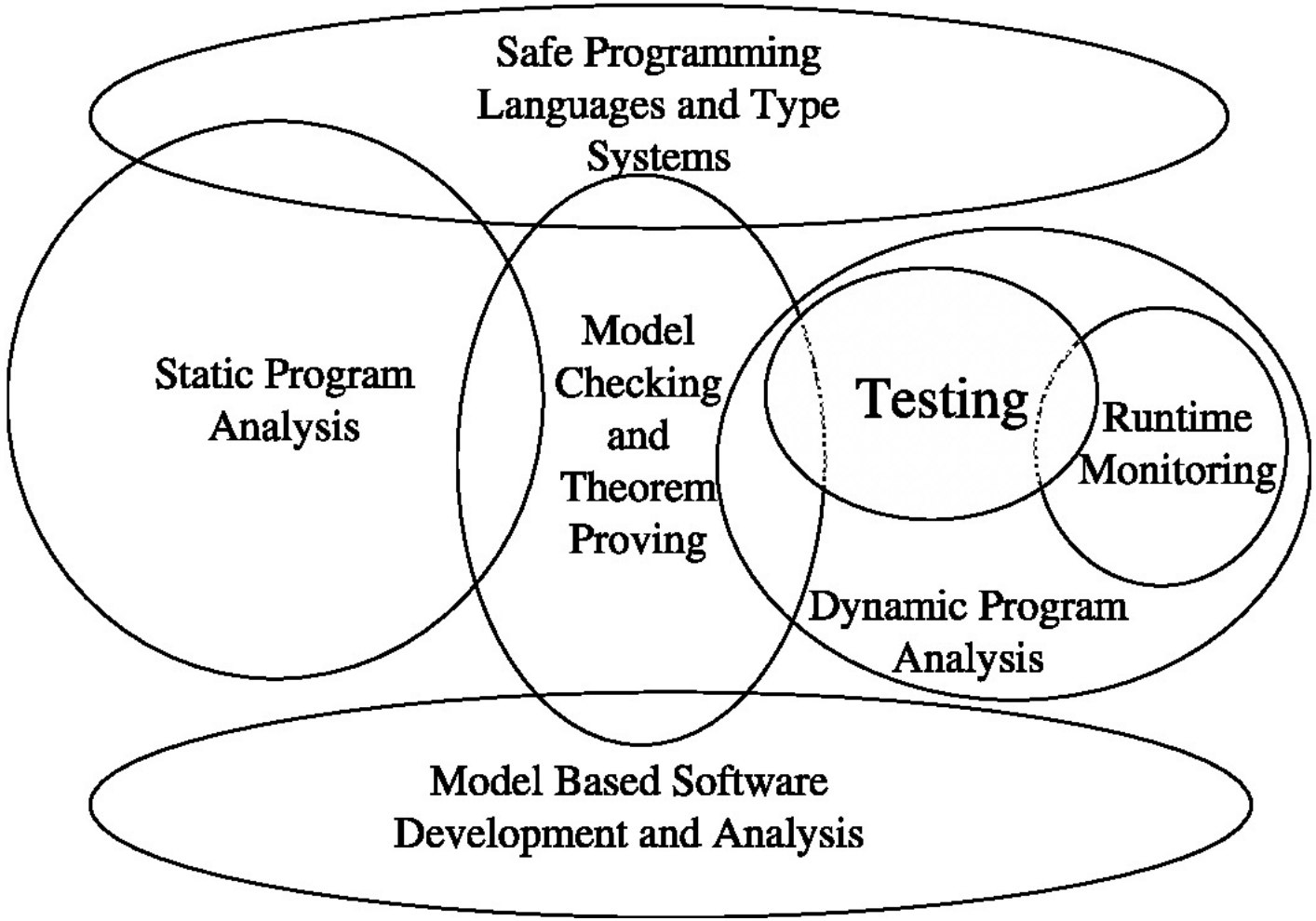


Figure 1: **Illustration of adversarial reprogramming.** (a) Mapping of ImageNet labels to adversarial task labels (squares count in an image). (b) Two examples of images from the adversarial task (left) are embedded at the center of an adversarial program (middle), yielding adversarial images (right). The adversarial program shown repurposes an Inception V3 network to count squares in images. (c) Illustration of inference with adversarial images. The network when presented with adversarial images will predict ImageNet labels that map to the adversarial task labels.

www.darpa.mil

Towards a Unified Program Analysis

Program Analysis



Generated from Wikipedia page on Program analysis with Wolfram Alpha online

Is anything missing?

K. Sen, "Scalable automated methods for dynamic program analysis", 2006

http://osl.cs.illinois.edu/media/papers/sen-2006-scalable_automated_methods_for_dynamic_program_analysis.pdf

The Program Analysis Paradox

All the interesting general problems in program analysis are either algorithmically intractable or undecidable when the desired solutions must be either complete or sound

All the things I really like to do are either illegal, immoral, or fattening
-- *Alexander Woollcott*
[in *Reader's Digest,* 1933]

source: https://en.wikiquote.org/wiki/Alexander_Woollcott /

Fuzzing?

A Survey of Symbolic Execution Techniques

ROBERTO BALDONI, EMILIO COPPA, DANIELE CONO D'ELIA, CAMIL DEMETRESCU, and IRENE FINOCCHI, Sapienza University of Rome

Symbolic execution techniques have been brought to the attention of a heterogeneous audience since DARPA announced in 2013 the Cyber Grand Challenge, a two-year competition seeking to create automatic systems for vulnerability detection, exploitation, and patching in near real-time [95]. More remarkably, symbolic execution tools have been running 24/7 in the testing process of many Microsoft applications since 2008, revealing for instance nearly 30% of all the bugs discovered by file fuzzing during the development of Windows 7, which other program analyses and blackbox testing techniques missed [53].

https://arxiv.org/abs/1610.00502

A long vocabulary of methods

119 papers as of 2017

[..] **dozens** of tools developed over the **last four decades**, leading to major practical breakthroughs in a number of prominent software reliability applications [..]

A Survey of Symbolic Execution Techniques

ROBERTO BALDONI, EMILIO COPPA, DANIELE CONO D'ELIA, CAMIL DEMETRESCU, and IRENE FINOCCHI, Sapienza University of Rome

Many security and software testing applications require checking whether certain properties of a program hold for any possible usage scenario. For instance, a tool for identifying software vulnerabilities may need to rule out the existence of any backdoor to bypass a program's authentication. One approach would be to test the program using different, possibly random inputs. As the backdoor may only be hit for very specific program workloads, automated exploration of the space of possible inputs is of the essence. Symbolic execution provides an elegant solution to the problem, by systematically exploring many possible execution paths at the same time without necessarily requiring concrete inputs. Rather than taking on fully specified input values, the technique abstractly represents them as symbols, resorting to constraint solvers to construct actual instances that would cause property violations. Symbolic execution has been incubated in dozens of tools developed over the last four decades, leading to major practical breakthroughs in a number of prominent software reliability applications. The goal of this survey is to provide an overview of the main ideas, challenges, and solutions developed in the area, distilling them for a broad audience.

that would cause property violations. Symbolic execution has been incubated in dozens of tools developed over the last four decades, leading to major practical breakthroughs in a number of prominent software reliability applications. The goal of this survey is to provide an overview of the main ideas, challenges, and solutions

https://arxiv.org/abs/1610.00502

Fuzzing == Randomized Program Analysis

Program analysis

From Wikipedia, the free encyclopedia

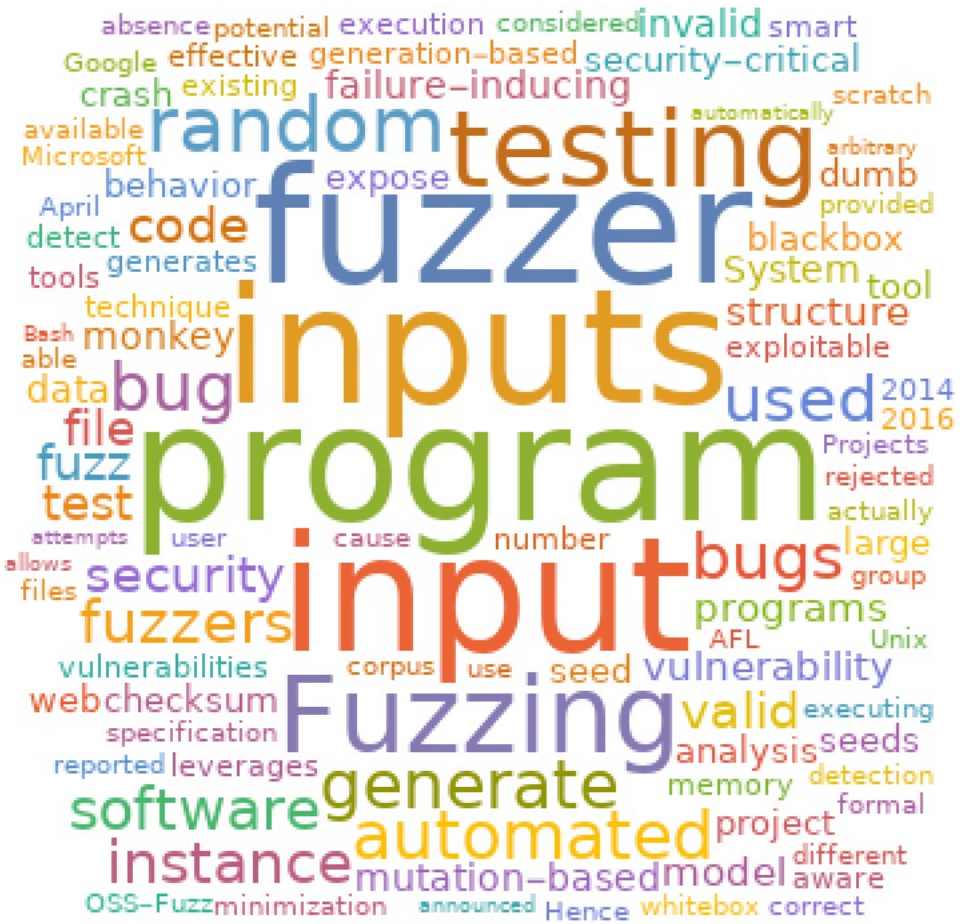
Contents [hide]

1 Static program analysis
- 1.1 Control-flow
- 1.2 Data-flow analysis
- 1.3 Abstract interpretation
- 1.4 Type systems
- 1.5 Effect systems
- 1.6 Model checking

2 Dynamic program analysis
- 2.1 Testing
- 2.2 Monitoring
- 2.3 Program slicing

3 See also

Fuzzing



Generated from Wikipedia page on Fuzzing with Wolfram Alpha online

"Fuzzing" is Randomized Program Analysis

Program analysis

From Wikipedia, the free encyclopedia

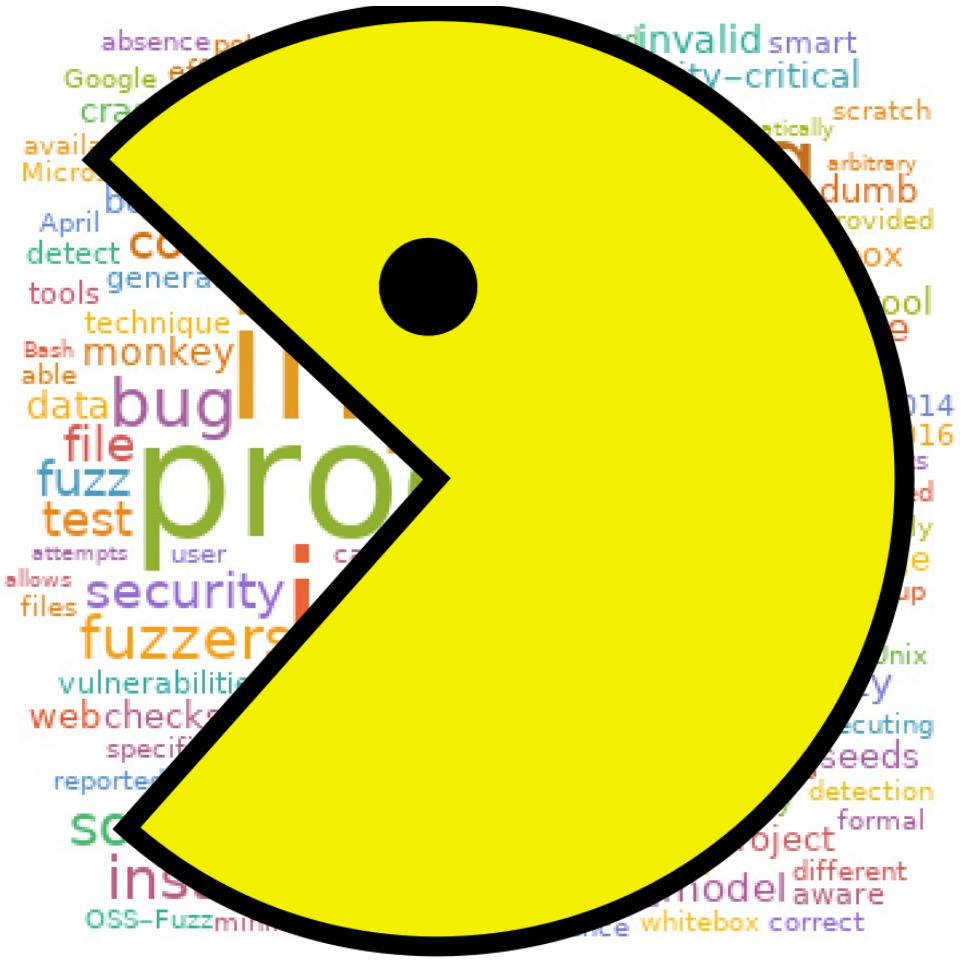
Contents [hide]

1 Static program analysis
- 1.1 Control-flow
- 1.2 Data-flow analysis
- 1.3 Abstract interpretation
- 1.4 Type systems
- 1.5 Effect systems
- 1.6 Model checking

2 Dynamic program analysis
- 2.1 Testing
- 2.2 Monitoring
- 2.3 Program slicing

3 See also

Fuzzing



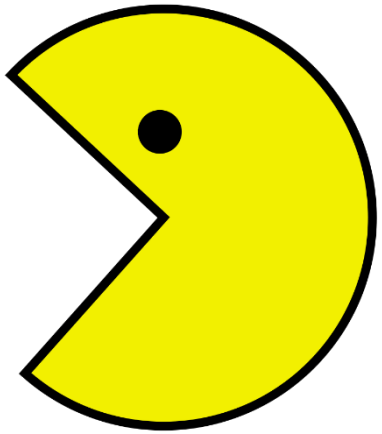
Source: https://simple.wikipedia.org/wiki/File:Pac_Man.svg (public domain)

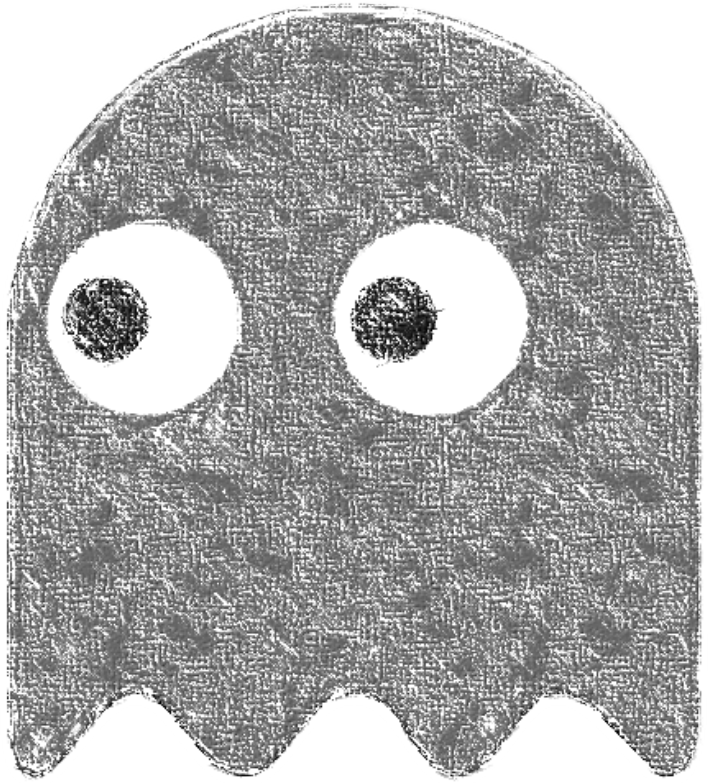
Fuzzing is Randomized Program Analysis

Path explosion

State explosion

Fuzzing





Source: https://www.downloadclipart.net/download/87877/pac-man-ghost-png-clipart-svg (Free for personal or commercial use, attribution not required)

Vocabularies of methods

A Vocabulary of Program Slicing-Based Techniques

JOSEP SILVA
Universidad Politécnica de Valencia

This article surveys previous work on program slicing-based techniques. For each technique we describe its features, its main applications and a common example of slicing using such a technique. After discussing each technique separately, all of them are compared in order to clarify and establish the relations between them. This comparison gives rise to a classification of techniques which can help to guide future research directions in this field.

Categories and Subject Descriptors: F.3.1 [**Theory of Computation**]: Logics and meaning of programs—*specifying and verifying and reasoning about programs*; D.3.1 [**Software**]: Programming Languages—*formal definitions and theory*

General Terms: Languages, Theory

Additional Key Words and Phrases: program slicing, software engineering

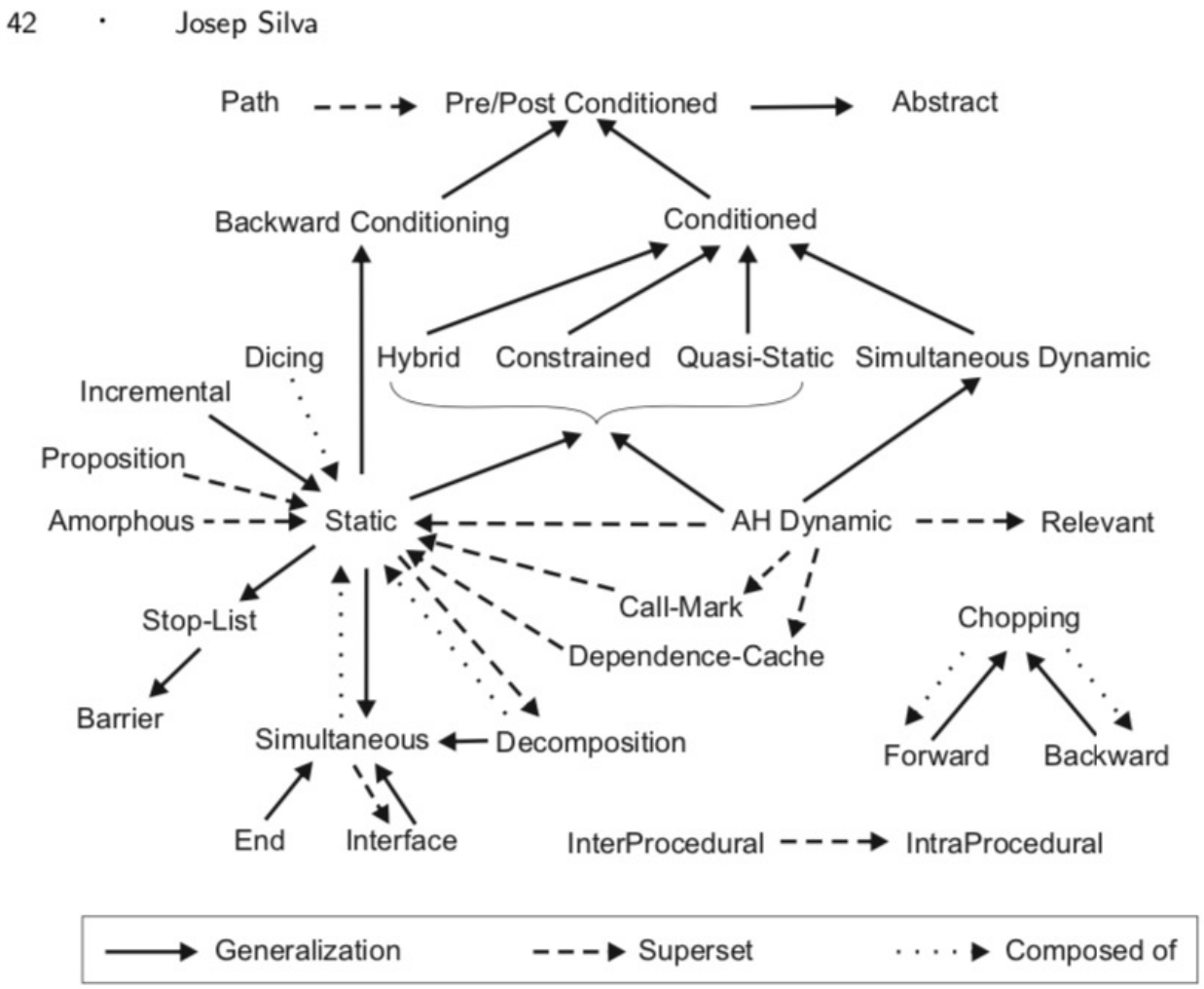
1. INTRODUCTION

Program slicing was originally introduced in 1984 by Mark Weiser. Since then, many researchers have extended it in many directions and for all programming paradigms. The huge amount of program slicing-based techniques has lead to the publication of different surveys [Tip 1995; Binkley and Gallagher 1996; Harman et al. 1996; Harman and Gallagher 1998; De Lucia 2001; Harman and Hierons 2001; Binkley and Harman 2004; Xu et al. 2005] trying to clarify the differences between them. However, each survey presents the techniques from a different perspective.

http://personales.upv.es/josilga/papers/Vocabulary.pdf

A. VOCABULARY INDEX

Do vocabularies have structures we can search?



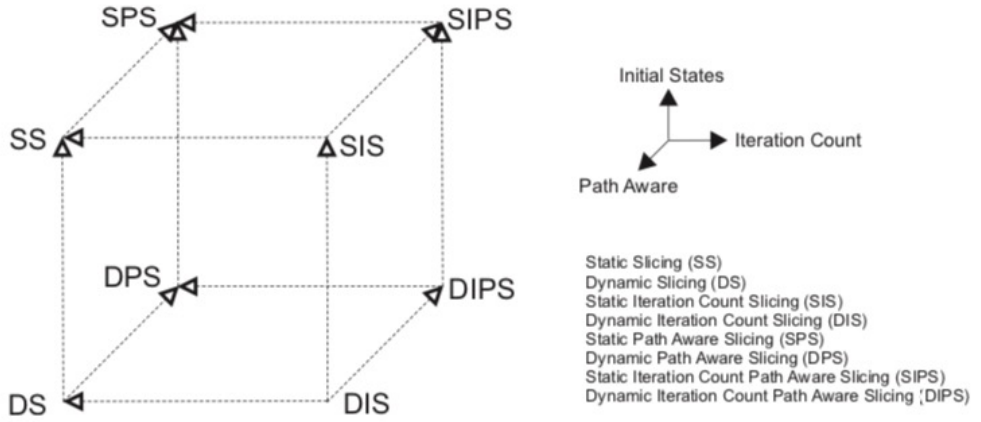


Fig. 32. Superset relationships between the program slicing techniques of Table II

http://personales.upv.es/josilga/papers/Vocabulary.pdf

Hypothesis

Machine learning in program analysis may produce qualitative improvements by providing means for exploring the space of over-approximations and soundness tradeoffs, by unifying static and dynamic analysis with tracing as self-supervision

Fighting the monsters of space and path explosion

so faster, instruction-level symbolic emulation. Additionally, QSYM loosens the strict soundness requirements of conventional concolic executors for better performance, yet takes advantage of a faster fuzzer for validation, providing unprecedented opportunities for performance optimizations, e.g., optimistically solving constraints and pruning uninteresting basic blocks.

QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing

Insu Yun† Sangho Lee† Meng Xu† Yeongjin Jang* Taesoo Kim†

† *Georgia Institute of Technology*
* *Oregon State University*

Abstract

Recently, hybrid fuzzing has been proposed to address the limitations of fuzzing and concolic execution by combining both approaches. The hybrid approach has shown its effectiveness in various synthetic benchmarks such as DARPA Cyber Grand Challenge (CGC) binaries, but it still suffers from scaling to find bugs in complex, real-world software. We observed that the performance bottleneck of the existing concolic executor is the main limiting factor for its adoption beyond a small-scale study.

To overcome this problem, we design a fast concolic execution engine, called QSYM, to support hybrid fuzzing. The key idea is to tightly integrate the symbolic emulation with the native execution using dynamic binary translation, making it possible to implement more fine-grained, so faster, instruction-level symbolic emulation. Additionally, QSYM loosens the strict soundness requirements of conventional concolic executors for better performance, yet takes advantage of a faster fuzzer for validation, providing unprecedented opportunities for performance op-

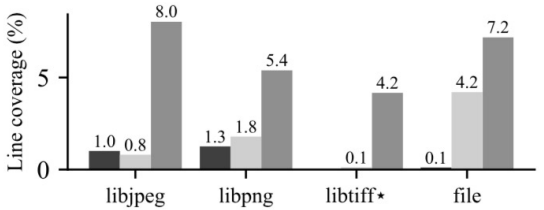


Figure 1: Newly found line coverage of popular open-source software by state-of-the-art concolic executors, Driller and S2E, and our system, QSYM, until they saturated. We used five test cases in each project that have the largest code coverage. Test cases generated by QSYM cover significantly more lines than both concolic executors. In libtiff, Driller could not generate any test case due to incomplete modeling for `mmap()`.

at discovering inputs that lead to an execution path with loose branch conditions, such as `x > 0`. On the contrary, concolic execution is good at finding inputs that drive the program into tight and complex branch conditions, such

https://www.usenix.org/conference/usenixsecurity18/presentation/yun

Redefining the problem to avoid the paradox

- Program under analysis "as is" -> programs custom-built for analysis
- Sound non-stochastic analyses -> randomized analyses (better-defined fuzzing)
- Static or dynamic analyses -> hybrid with all three/four (static, dynamic, custom-built randomized)
- Behaviors of PuA -> Changes in behaviors based on code & input changes
- Path and state explosion heuristic trade-offs -> ML to explore patterns

From program "as is" to programs custom-built for analysis

implementations by orders of magnitude. We present SYMCC, an LLVM-based C and C++ compiler that builds concolic execution right into the binary. It can be used by software developers as a drop-in replacement for `clang` and `clang++`,

https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

Symbolic execution with SYMCC: Don't interpret, compile!

Sebastian Poeplau
EURECOM

Aurélien Francillon
EURECOM

Abstract

A major impediment to practical symbolic execution is speed, especially when compared to near-native speed solutions like fuzz testing. We propose a compilation-based approach to symbolic execution that performs better than state-of-the-art implementations by orders of magnitude. We present SYMCC, an LLVM-based C and C++ compiler that builds concolic execution right into the binary. It can be used by software developers as a drop-in replacement for `clang` and `clang++`, and we show how to add support for other languages with little effort. In comparison with KLEE, SYMCC is faster by up to three orders of magnitude and an average factor of 12. It also outperforms QSYM, a system that recently showed great performance improvements over other implementations, by up to two orders of magnitude and an average factor of 10. Using it on real-world software, we found that our approach consistently achieves higher coverage, and we discovered two vulnerabilities in the heavily tested OpenJPEG project, which have been confirmed by the project maintainers and assigned CVE identifiers.

Large spaces to search with Machine Learning

- Analyses form algebraic structures (e.g., w.r.t. to over-approximations or under-approximations)
  - Abstract interpretation is mathematically general, but requires human involvement in creation of abstractions
  - Dynamic analysis should be included in these structures

- Machine learning to navigate between the algebraic structures composed of analyses
  - ML to learn appropriate abstractions and navigate their granularity
  - ML should interact with Compilation to enable effective learning

www.darpa.mil