Verified Security and Performance Enhancement of Large Legacy Software (V-SPELLS)

Sergey Bratus
Information Innovation Office (I2O)

Briefing prepared for Proposers Day

29 July 2020

Program objective

Enable piecewise, compatible-by-construction enhancement of software components in legacy DoD systems, by creating methods and tools to recover succinct models of domain data abstractions and logic from the source code, add enhancements to the models, and convert them to performant new component implementations verified to be compatible and secure

*Gain benefits of formal software verification in **incremental** software (re)engineering rather than only in clean-slate introduction*

Who cares?

DoD has a strong need for assured incremental modernization of legacy systems in a manner that reduces rather than raises risk. Lack of this capability blocks the following desirable upgrade scenarios:

- Significant performance gains are possible by shifting operations to modern hardware (packet processing, data stream reassembly, signal processing, etc.), but refactored software must have isomorphic functionality
- A new memory-safe technology to replace legacy vulnerability-prone networking code is available, but legacy under-specified behavior must be replicated precisely
- Spectre vulnerability mitigation requires isolating parts of computation on dedicated CPUs, but distributing execution of software written for a single CPU is risky

Assured incremental modernization, despite availability of source code, is impeded because:

- Module boundaries and dependencies aren't clear
- Available documentation is often not sufficient for reasoning about code, since it may not align with as-built system
- Interface Control Documents may not capture code version differences

Today: Extreme difficulty enhancing legacy code with verified, assured code

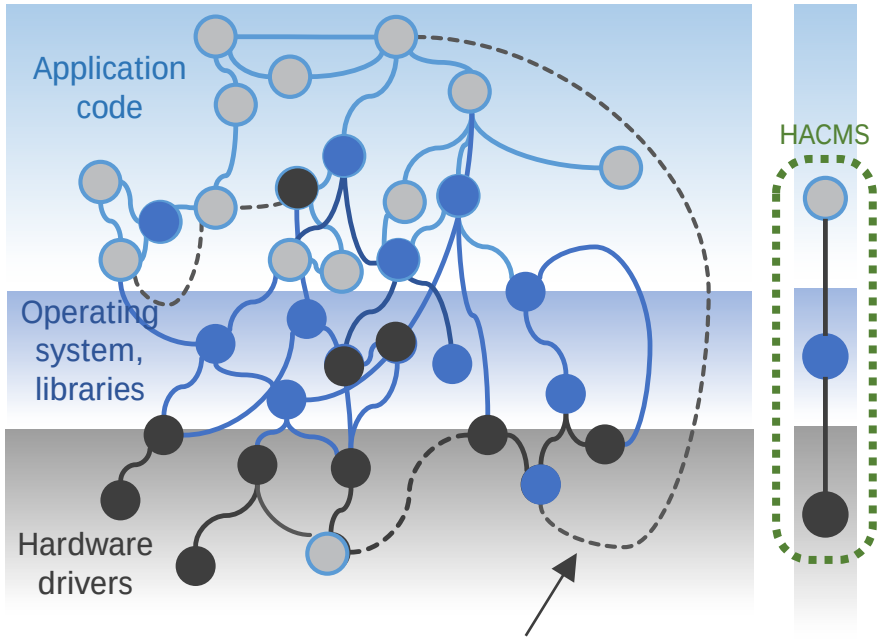
Sustainers struggle to gain effective understanding of legacy code

- Legacy code entangles high-level logic with low-level implementation detail
- Code is difficult to comprehend due to developer shortcuts and deviations from intended design and architecture
- Component specifications aren't available, must be inferred from implementation
- No developer-usable ways to write verified code that integrates with existing code base
- Past optimizations remain although no longer useful

Consequences

- DoD systems are locked into obsolete hardware and software components
- Capability upgrades become increasingly difficult over time
- No capability for high assurance functional enhancement of legacy components

Entangled legacy code base



Non-compliant "shortcut" paths expediently introduced by developers, unintended by architects

Examples: IPv6 stack, Wi-Fi & USB drivers

What is possible today:

- Local repair (ongoing research)
  - No functional enhancement
- Full replacement from scratch (HACMS), but
  - Clean-slate approach has no intake method for existing code
  - No automation for migrating verified code to another platform: proofs don't translate
- Re-engineering components without strong security and compatibility guarantees

Legend:
- Application and domain-specific logic
- Implementation of domain abstractions
- Hardware-specific logic and optimizations

HACMS = High Assurance Cyber Military System

Vision: Piece-by-piece replacement of legacy code with verified code

With V-SPELLS: Incremental repair and enhancement with verified code

1. Decomposition

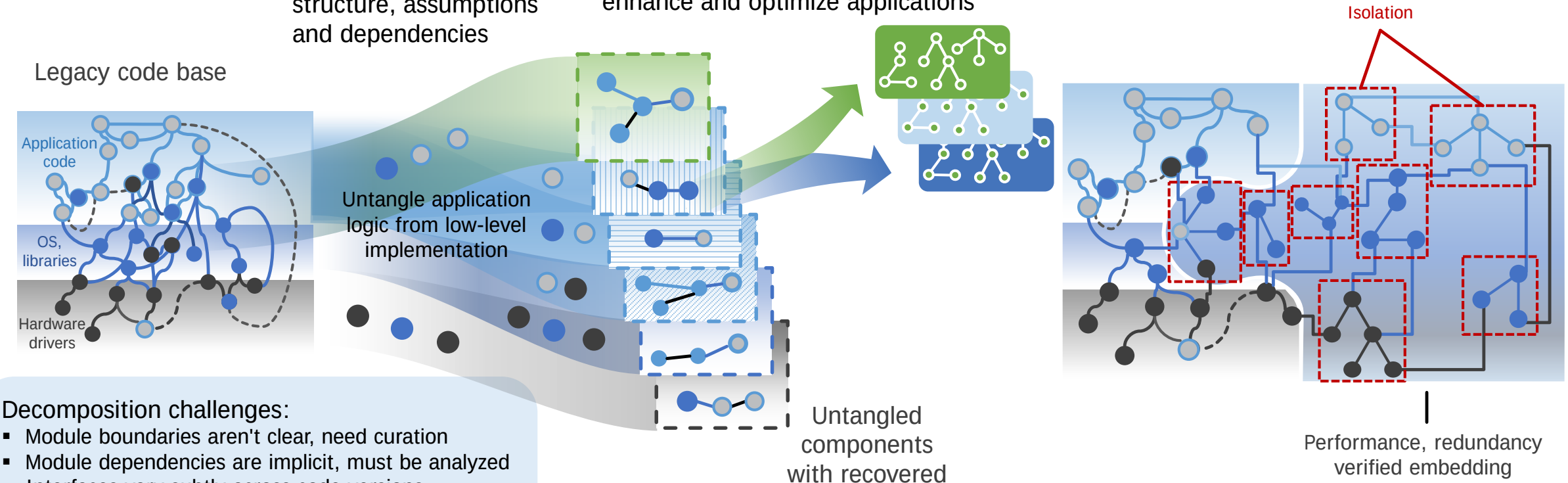
Infer architectural structure, assumptions and dependencies

2. Incremental enhancement

- Interactively iteratively recover domain abstractions into Domain Models and Domain Specific Languages (DSLs)
- Leverage domain models and DSLs to enhance and optimize applications

3. Enhanced verified components

- Develop verified optimization ("flattening") of domain abstractions to enhance performance
- Restructure code among distributed processors for performance and isolation



Decomposition challenges:

- Module boundaries aren't clear, need curation
- Module dependencies are implicit, must be analyzed
- Interfaces vary subtly across code versions
- ICDs aren't granular enough or fully up to date
- Timing constraints are often implicit
- Bug compatibility must occasionally be preserved

Allows restructuring, verification, performance enhancements

Background: Domain Specific Languages (DSLs), a key enabling technology

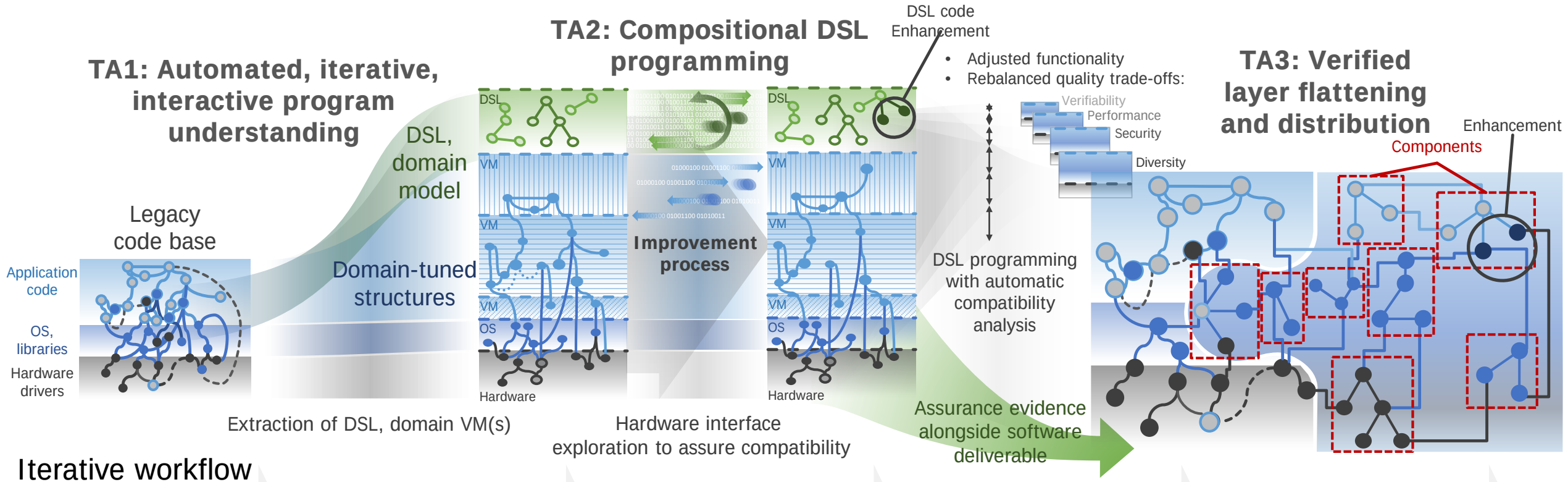
Benefits of DSLs and domain models:

- DSLs capture knowledge about domain data structures and operations
  - Example: Network packets contain IP addresses and data payloads to be reassembled into streams; a packet-processing stream hides how these are represented as bits, automates extraction of data and metadata
- Very concise, allow domain expert to understand functionality at a glance
  - Example: Stream reassembly may take several pages of code in C, several lines in DSL
- DSLs hide and automate low-level code bookkeeping, eliminate programmer error in implementing domain operations
  - Examples: Packet processing DSLs eliminates wrong-offset errors and wrong byte-order errors; event DSLs eliminate concurrency and synchronization errors
- DSL code is easier to validate, adapt, and optimize, due to explicit and succinct semantics
  - Example: Graphics processing DSL beats native code performance by reordering operations a compiler would never attempt to reorder
- DSLs help domain experts re-engineer and reason about the systems
  - Example: Networking DSLs make it easy to prototype new protocol features, which is very hard to code natively in the OS kernel

Challenges to success with DSLs:

- Performance reduction due to added layers of abstraction (novel flattening technology is needed, *challenge for TA3*)
- Matching abstractions with the domain (novel program understanding automation is needed, *challenge for TA1*)
- Multiple DSLs seem hard to link together (novel verified linking technology is needed, *challenge for TA2*)

Approach: Iterative verified enhancement via fast, secure DSLs



Iterative workflow

Untangle
Application logic is untangled from low-level implementations of domain operations, past optimizations

Separate
Legacy component is separated into high-level application logic and low-level domain operations on domain data structures

Recover abstractions
A DSL and domain model are extracted for the application logic and a Virtual Machine (VM) is extracted from low-level operations, domain data structure definitions

Re-implement
Application logic is re-implemented in the DSL, code unit by code unit, with continuous automated compatibility checking against the rest of the legacy code base

Flatten
DSL implementation is optimized and reduced ("flattened") together with the VM stack, distributed among hardware units if needed for the enhancement

Program structure

TA 1: Automated, iterative, interactive program understanding

- Extract domain data structures and operations, domain-specific virtual machine(s)
- Extract domain-specific semantic model and language (DSL) for application logic
- Automate lifting of code to the extracted DSL

TA 2: Compositional DSL programming, component specification inference

- Create usable integrated development environment for compositional DSL programming
- Develop compatibility-focused program analyses with counterexample generation and refinement
- Automate hardware interface exploration

TA 3: Verified layer flattening and distribution

- Develop principled layer-smashing and distribution for DSL/model stacks
- Create universal application binary interface (ABI) descriptions and extensions for multi-target composition
- Develop proof composability and translation approaches, ABI-like extensions for proofs

TA 4: Demonstration and Evaluation

- Curate and provision open source software and hardware case studies suitable for fundamental research
- Set up and manage transition to DoD systems

TA1: Automated, iterative, interactive software understanding

Challenges:

- Achieve the capability to refactor significant code bases in *months*, not years
- Automate code lifting to DSL for majority of code, not isolated code fragments

Approaches:

- Decompose legacy code into functional modules with domain data structure and operation definitions
  - Example: Network stack decomposes into protocol modules; protocol packets have domain definitions and operations, such as particular field extraction and assembly into data streams
- Automate untangling of legacy code into low-level domain operation implementations (e.g., domain VM(s)) and higher-level application logic (e.g., DSL/model)
  - Example: Protocol implementations decompose into handling of bitwise representations of protocol data and metadata vs higher-level protocol logic; DSL hides bitwise representations, protecting developers from errors
- Automate lifting of legacy code into extracted DSL, backed by the extracted VM
  - Example: Code for assembling data streams from packets is refactored into DSL that hides packet's bit-level details, offloads to NIC's hardware accelerator where possible

TA2: Compositional DSL programming, component specification inference

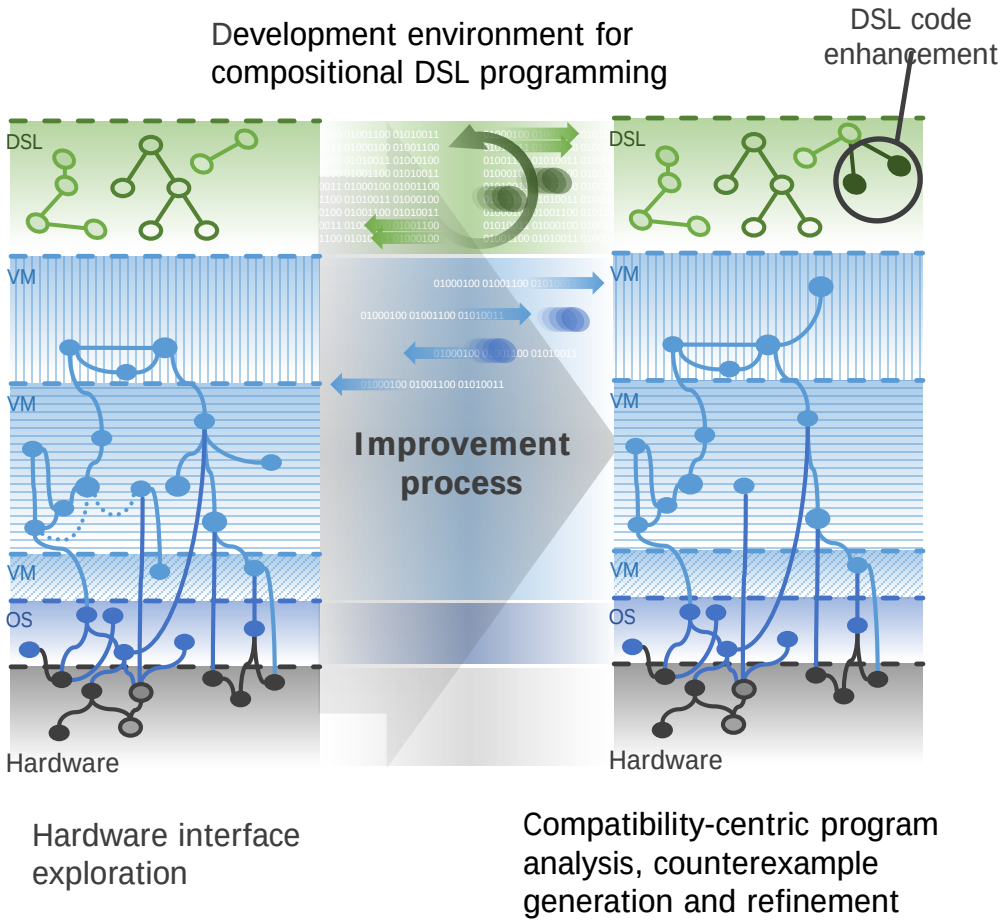
Operating in DSLs enables regular developers to quickly create new verified enhancements that safely compose with the rest of the system

Challenges of classic verification (HACMS, NSF DeepSpec):

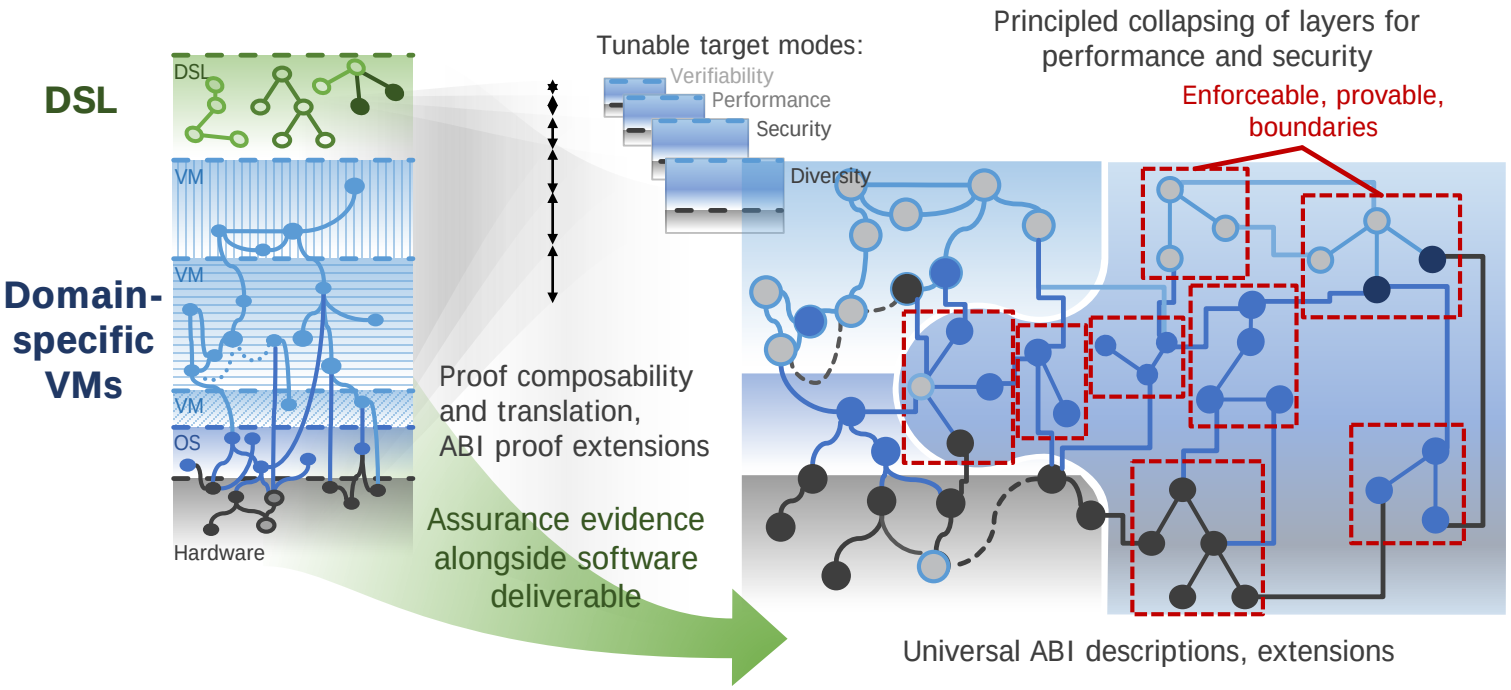
- Specification is created in proof logic languages, not accessible to developers
- Components are tested for compatibility only after fully built
- Incompatibility means costly specification adjustments, laborious flow-down through application logic and proof specialists to developers

V-SPELLS approaches for TA2:

- Create development environment for *compositional DSL programming:* Developers re-implement a component in **DSLs**, receive immediate, interactive feedback on compatibility with old code
- Develop novel program analyses focused on compatibility, providing efficient, intelligible feedback and refined counterexamples to developers
- Automate hardware interface exploration to support verification



TA 3: Verified layer flattening and distribution



Challenges:

- Develop compilation methods and ABI binary representations for platforms consisting of multiple distributed targets
  - Today's compilation and ABIs do not support distribution, assume a single address space
  - Today's ABIs are obscure, under-documented
- Develop ABI-like rules for composable proof representation, packaging, and transformation
  - Today's proofs are brittle and non-granular

Application Binary Interfaces (ABIs) establish the rules of interaction and composition between pre-compiled code objects such as function, libraries, and operating system kernels.

Approaches:

- Develop "layer-smashing" compilation techniques for DSL/VM stacks that are tunable for performance, security, diversity, verifiability
  - Develop comprehensive metrics to assess progression of provable properties relevant to the goals
- Create language for universal description of existing ABIs, design extensions for supporting multi-target composition
- Develop ways of maintaining dependencies and transformations for composable, granular proofs

TA 4: Demonstration and evaluation

Challenge:

- Represent diverse, DoD-relevant use cases of legacy understanding and hardware platform translation

Approach – DoD system transition activity:

- Identify platforms and use cases for DoD transition, apply tools and methodologies developed by TAs 1-3

Approach – Open-source activity:

- Open source components are a critical part of the DoD supply chains
- Curate and provision open source software and hardware case studies suitable for fundamental research
  - Large deployments in industry, known need for modernization despite common use
  - Realistic hardware offload scenarios
  - Software decomposition/distribution for cloud deployments

No Controlled Technical Information (CTI) to facilitate broad research community engagement

Transition

Exemplary targets:

- Phase 2: Software enhancement of an Air Force legacy embedded platform
- Phase 3: UAV hardware platform representative of the DoD incremental enhancement needs

- Proposing teams will be strengthened by combining fundamental research capability with team capability for handling legacy DoD code, to facilitate transition

Evaluation metrics

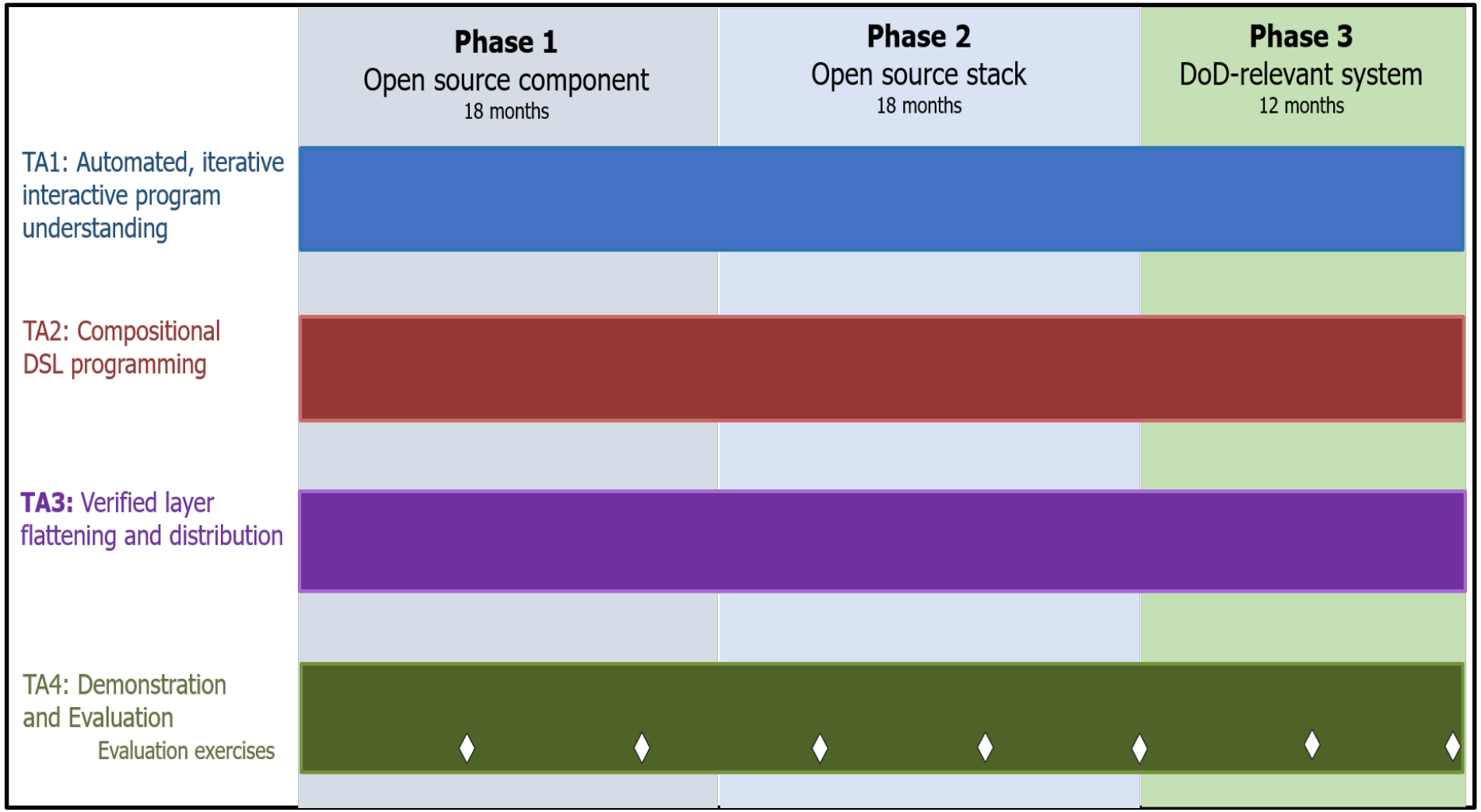
- TA1: Understanding is measured by *capability milestones* achieved for respective code sizes of each phase
  - Considered achieved if the *accuracy* goal is achieved on the part of code larger than the *coverage* goal
- TA2: Usability is measured by *speed and completeness* of feedback (code equivalence assertions or refined counterexamples) provided by the DSL programming environment to the programmer
  - Considered achieved if more that the goal percentage of equivalence analyses complete in less than the given time
- TA3: Flattening efficiency is measured by *performance overhead* over the average optimized compiler results for the platform
  - For hardware offloading, overhead is measured over typical hardware acceleration speed-ups for the platform

Metric	Phase 1 goals (18 months)	Phase 2 goals (18 months)	Phase 3 goal (12 months)
Code scale	10Ks SLoC	100Ks SLoC	1000Ks SLoC
Program understanding capability	Effective hook system (60%/70%) (all domain data types accesses and operations are instrumented)	Effective composition (80%/90%) (all domain operations can be programmed in DSL, compatibly with native code)	Complete replacement (95%/99.9%) (all code is translated to DSL/VM, except a few effectively analyzed enclaves)
Usability of interactive analyses	Hours at 60% completeness	Minutes at 80% completeness	Minutes at 97% completeness
Efficiency of flattened DSL (relative to initial system state)	70-80%	85-95%	>110-200% (>10x for some domains)
Initial memory load reduction	5x	10-20x	100x
DSL code size reduction	10x	100x	>500x for user-facing code

Evaluation details

- Each performer participates in technical challenge events
  - Government reserves the right to engage third parties to independently validate results
  - Government reserves the right to engage third parties to provide challenge problems

Program schedule



Meetings and reporting requirements

- Two annual Principal Investigator (PI) meetings centered around a challenge problem
- PM site visits between PI meetings
- Monthly financial reporting
- Quarterly technical progress reporting
  - Technical report describing progress, resources expended and issues requiring Government attention, provided 10 days after the end of each quarter
- System Development Plan provided one month after the kick-off meeting for each phase
  - Describe the scope/design and hardware and software architecture
- Real-time reporting of publications, prototypes, significant results via website/wiki or similar mechanism
- Financial/technical progress reporting to DARPA Technology Financial Information Management System (TFIMS)
- Final Technical Report

Funding and programmatic details

- Proposals due: Thursday, September 9, 2020 at 12:00 noon (ET)

- Government anticipates multiple awards
  - Procurement contracts, cooperative agreements or other transactions

- Organizations can submit separate proposals to all Technical Areas (TAs)
  - Each proposal may address only one TA, or any combination of TA1, TA2, and TA3
    - If proposing to a combination of TA1, TA2, or TA3, proposals must make efforts addressing different TAs **clearly distinct and separable** by costs and Statement of Work
  - Multiple proposals may be submitted, however:
    - Selected **TA4** performer(s) **may not** perform on other TAs,
    - Which to consider for award (if any) is at the discretion of the Government
  - Proposals will be strengthened by combining a fundamental research capability with a team capability for handling legacy DoD code

- To expedite award contracting, proposers are encouraged to have sub-award agreements in place ahead of award notification

www.darpa.mil