Automotive Vulnerability Detection System

David K. Wittenberg, PhD
BAE Systems
Burlington, MA

Jeffrey Smith, PhD
BAE Systems
Burlington, MA

Robert Gray, PhD
BAE Systems
Arlington, VA

Gregory Eakman, PhD
BAE Systems
Burlington, MA

***Abstract*—In [1] we presented a Vulnerability Detection System (VDS) that can detect emergent vulnerabilities in complex Cyber Physical Systems (CPSs). It used the attacker's point of view by collecting a target system's vulnerability information from varied sources and populating an Attack Point (AP) database. From these APs, a Hierarchical Task Network (HTN) generated the set of composite device-level attack scenarios. The VDS used Alloy [2], a Satisfiability (SAT) planner to reduce the cardinality of the generated space by evaluating the feasibility of each attack. In [3], we specialized the VDS for the automobile domain. This paper further 1) specializes our prior research by submitting the generated prioritized list to an Automotive-specific Attack Evaluation Process (AAEP) and 2) enhances our prior research with a method to discover and test vulnerabilities by reverse engineering the actual binary code. With a combination of simulation and vehicle instrumented real-time execution, the AAEP confirms each candidate attack. The AAEP's output is used as feedback to refine the SAT constraint model model. A novel part of AAEP is our Automated Reverse Engineering (ARE) system, which greatly reduces the search space for software bugs. The VDS is designed to support short product release cycles.**

A. *Knowledge Acquisition*

I. Introduction

Automobiles are highly complex with a large set of sensors and information sources assisting the driver to safely drive the vehicle. Modern automotive electrical systems are modularized into many increasingly complex, frequently upgraded, Electronic Control Units (ECUs). ECUs are composed of components, with vulnerabilities, some that are known publically, some that are known by the manufacturer, and some that are yet to be discovered. Just as a composite system can display emergent behaviors, i.e. behaviors that are not predicted solely by modeling the assembly of the constituent parts, it can also have **emergent vulnerabilities** leading to exploits that are unique to the aggregate system. These exploits could be catastrophic even though they result from a combination of relatively benign subsystem vulnerabilities. The composition of many components in an automobile leads to a combinatorial explosion of the behaviors to check. Given the short upgrade cycle and the rapidly changing list of vulnerabilities for each part, reaching a minimal level of due diligence when designing and building a automotive electrical system is a daunting task.

With automobiles becoming self-sufficient, even to the point of controlling the driving, the human becomes a passenger. This represents a ubiquitous, massively deployed, safety critical system in the center of our critical transportation infrastructure. Though this autonomy promises great improvements to public safety, it also poses potential threats to it. These threats include the effect of a random car malfunction, and a coordinated malfunction causing havoc throughout the transportation infrastructure. We have seen such attacks demonstrated by [4].

The challenge is to prove the proper functioning of the automobile's electronic systems under all possible conditions in the presence of attackers. This task is extremely difficult based on the vast set of possible scenarios and the failure modes of the components involved. The VDS offers a disciplined approach that identifies vulnerabilities in the system, selects scenarios that would expose those vulnerabilities, and generates controlled tests based on those scenarios. Though this approach still produces a very large set of tests, and has issues with how to compute robustness of the solutions, it is tractable with a massively parallel simulation and storage system. The ARE further reduces the number of required tests by eliminating the requirement to check some paths through the software.

The VDS accelerates the detection of emergent exploits by identifying a manageable prioritized checklist of device-specific, feasible attack scenarios. These guide the discovery of automotive component vulnerabilities at a tempo supporting short ECU update cycles by:

- Automating ingestion of public and private vulnerabilities into a structured, semantically consistent format,
- Organizing vulnerabilities based on potential attack scenarios and transforming vulnerability information into standardized APs,
- Using an HTN planner to explore the APs and generating a large set of composite, device level sub-plans and attack scenarios to segment feasible and infeasible attacks against a known device,
- Using a satisfaction constraint model of the CPS, based on system constraints, to exhaustively evaluate device characteristics for potential exploit techniques, prune unlikely or infeasible attacks, and validate sub-plans to reduce the size of the set of composite vulnerabilities,
- Using the ARE to further characterize which software flows that must be checked and,
- Handing off a reduced, prioritized list of weaknesses, based on the potential for damage, for execution in simulated and actual automotive devices, to validate weakness checklist guidance or provide constraint/success feedback to our planner and constraint satisfaction steps.

The possible permutations of vulnerabilities, physical system configurations and attack scenarios are numerous. This

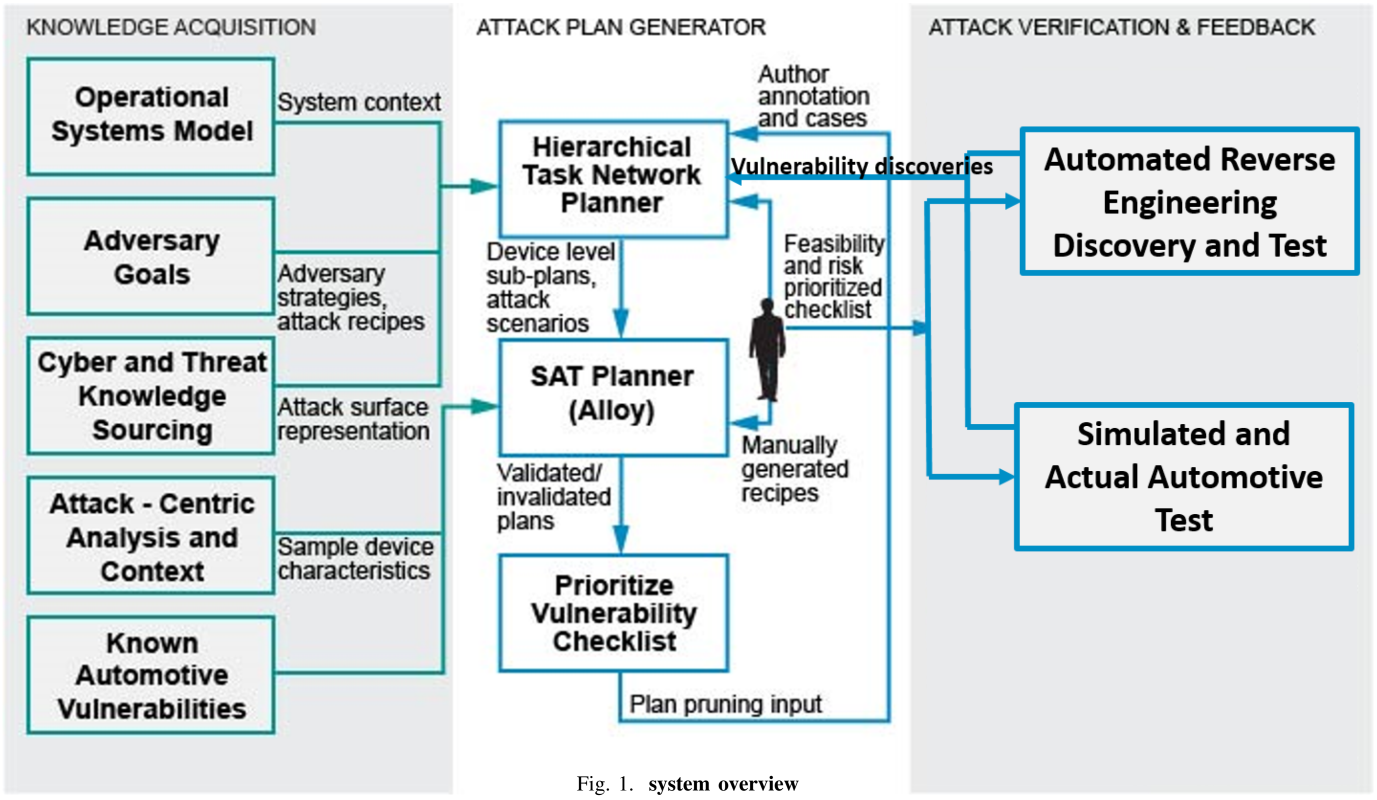

KNOWLEDGE ACQUISITION
ATTACK PLAN GENERATOR
ATTACK VERIFICATION & FEEDBACK
Operational Systems Model
Adversary Goals
Cyber and Threat Knowledge Sourcing
Attack - Centric Analysis and Context
Known Automotive Vulnerabilities
System context
Adversary strategies, attack recipes
Attack surface representation
Sample device characteristics
Hierarchical Task Network Planner
Device level sub-plans, attack scenarios
SAT Planner (Alloy)
Validated/ invalidated plans
Prioritize Vulnerability Checklist
Plan pruning input
Author annotation and cases
Vulnerability discoveries
Feasibility and risk prioritized checklist
Manually generated recipes
Automated Reverse Engineering Discovery and Test
Simulated and Actual Automotive Test

Fig. 1. **system overview**

VDS method of automatically maintaining these ontological relationships is scalable, saves time, and reduces errors. Reducing the number of these permutations, using planning and feedback, yields a manageable set of vulnerabilities and attacks to validate and check. The VDS results in earlier detection of potential exploits than is possible with manual methods, reducing cost and increasing safety.

The VDS analyzes emergent vulnerabilities of automotive component systems, including curve speed warning, traffic sign recognition, distance alert, and laser cruise control, on both an existing instrumented automobile and a simulation environment.

Setup of the Attack Verification process is time consuming and limited in its effective application to rapidly evolving attack scenarios. The VDS solves this problem by introducing extensive automation (including the ARE) for attack scenario generation. We use the Attack Verification results to generate feedback that enables the system to incrementally refine the generated attack plans.

II. System Architecture

Figure 1 depicts the Knowledge Acquisition, Attack Plan Generator, and Attack Feedback and Generation architecture parts. The Attack Plan Generator generates a prioritized (in order of risk) checklist of software and firmware that must be checked. The ARE and automotive test parts of the Attack Verification and Feedback consider the items on that checklist to determine which, if any, of them constitues a real danger.

To detect CPS threats, we consolidate data from several external data sources and restructure it into a format suitable for planning. Based on this team's study of MAISSI (Mediation, Alignment and Information Systems for Semantic Interoperability) [5], BAE Systems developed Dynamic Composition of Enterprise Services (DCES), an innovative approach to semantic alignment that handles the intricacies of real-world data. DCES effectively addresses the problem of service composition using a novel combination of existing open-source products and advanced semantic technologies.

In the computer domain, we have several databases of known attacks such as MITRE's CVE (Common Vulnerabilities and Exposures) [6], and NIST's (National Institute of Standards and Technologies) NVD (National Vulnerability Database) [7]. We need such databases for the automotive domain, and a few are starting to appear. In 2015, the Alliance of Automobile Manufacturers announced the start of such a database, Auto-ISAC [8], and Ring *et al.* [9] propose to strengthen it.

Translation from source data into a semantically equivalent representation in a target ontology faces a number of problems. These include structural dissimilarities (non-isomorphism) in the source and target data models, varied representations of data types, and disparities in the way properties and attributes are packaged into objects. MAISSI transforms each external data source record into a standardized AP record based on an Attack Surface Ontology (ASO) to normalize the data. We develop the ASO to accomplish two objectives: 1) translate vulnerabilities into potential APs and 2) capture attack cen-

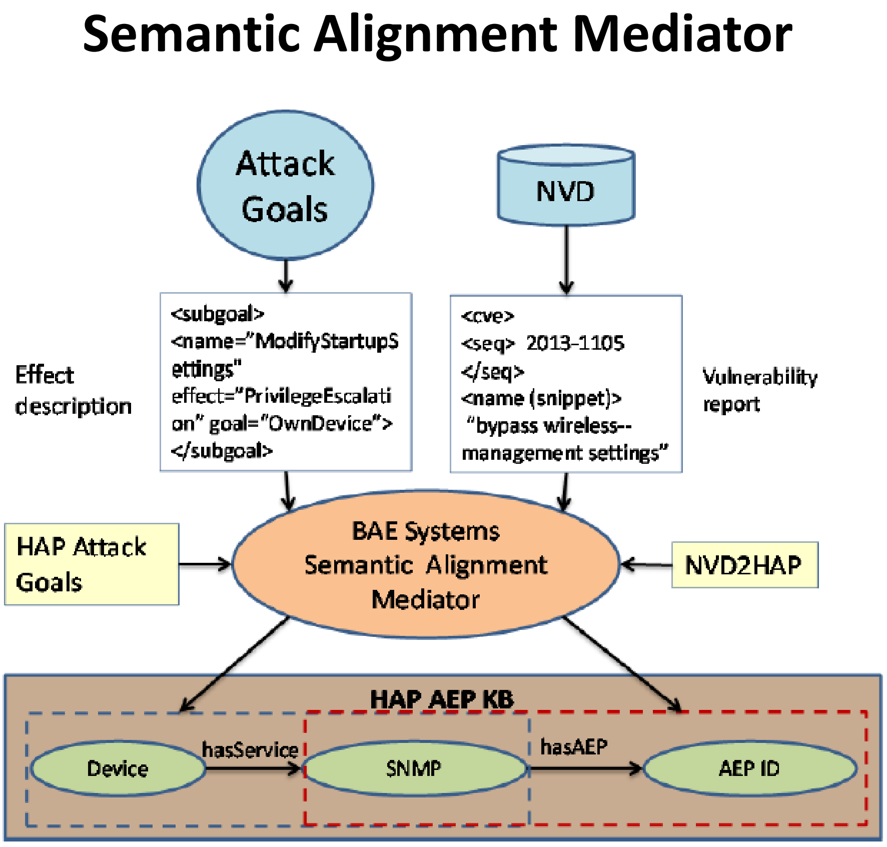

Semantic Alignment Mediator
Attack Goals
NVD
<subgoal> <name="ModifyStartupSettings" effect="PrivilegeEscalation" goal="OwnDevice"> </subgoal>
Effect description
<cve> <seq> 2013-1105 </seq> <name (snippet)> "bypass wireless--management settings"
Vulnerability report
HAP Attack Goals
BAE Systems Semantic Alignment Mediator
NVD2HAP
HAP AEP KB
Device
hasService
SNMP
hasAEP
AEP ID

Fig. 2. Semantic Alignment Mediator (MAISSI)

tered data that may include exploits, configurations, and ways to leverage legitimate device functionality.

A comprehensive AP repository is built through ASO development detailing how hackers may conduct malicious activities. One key aspect will be to apply the advanced reasoning and data mapping methods provided through our mediation solution to attach new semantic Weakness and Effect descriptions to each record. These are critical for developing a clear attack model that includes vulnerabilities and legitimate features that an attacker could misuse. This way, we enhance our attack surface perspective of a given device.

A. Attack-Centric Analysis and Context

The common bottom-up technology-centric perspective promoted by security researchers comes at the expense of understanding how a hacker discovers and exploits vulnerabilities. Hackers champion context, seeking to understand how a system functions in a top-down manner to reach a target. Each weakness grants the hacker leverage to gain greater control. Vulnerability repositories only organize and categorize system vulnerabilities rather than express how an individual vulnerability or feature may relate to, and even anticipate, an attack objective.

This VDS approach combines AP systems to conduct an attack-centric analysis that examines how vulnerabilities and legitimate device services relate to attack objectives. Sheyner and Wing [10] point out that a system represents a combination of services, configurations, functions and vulnerabilities that, when viewed from an attack perspective, combine into attack graphs depicting multistage cyber-attacks against systems. Whereas Sheyner and Wing's research focuses on the layers within a complex network environment, we extend the attack graph definition to represent system services and functions as discrete attack objects, focusing on layers within a complex network environment. Each of those architectural objects represents potential APs that are accessible only from the next layer, with externally available services residing on the outermost architecture layer.

The process of identifying and prioritizing system vulnerabilities involves reasoning over many series of exploits that can rapidly lead to an undesirably huge set of states. The VDS approach uses a predefined list of common attack patterns as an organizing and filtering tool, providing two methods to reduce the search space: probability of attack pattern applicability, and user interaction with the lists of attack categories and patterns. If patterns or categories exist in a hierarchical tree, reduction or inclusion occurs from the selected node and down the tree. Facilitating search with the possibility of human modification of any of the ASO structures used to generate attack goals and subgoals is important to

- Improve and direct search and test,
- Improve the goal generation process with spiked goals and plan fragments, and
- Refine structures with learned behavior.

This also facilitates parallel development and test. Figure 2 illustrates how the Semantic Alignment Mediator supports the VDS solution. In this example, to transform vulnerability data into attack-centric AP records, we combine the output from the Attack Goals analysis and an applicable Auto-ISAC [8] entry. For the information to be useful, our solution needs to know that the subgoal is the same as the Effect assigned to the Auto-ISAC entry. This unification is achieved with declarative mappings from the Attack Goals repository and Auto-ISAC schemas to the common ASO ontology. The result is a set of Resource Description Framework (RDF) statements in the AP knowledge base that links a device with a particular vulnerability that can be analyzed to confirm malice. Common Attack Pattern Enumeration and Classification (CAPEC) [11] attack patterns can also be used to seed the high-level planner portion of the Attack Plan Generator with attack plans or fragments of plans. Attack patterns have their own structure, with prerequisites. These will be translated using the semantic alignment techniques discussed above into forms that the Attack Plan Generator (next section) can reason over.

B. Attack Plan Generation

The VDS uses a hybrid planning mechanism and multiple sources of information to discover feasible attack plans. An SAT planner's (in our case, Alloy [2]) capabilities for exhaustively evaluating hierarchically structured spaces of possible exploit techniques, combined with heuristic search to prioritize choices of device characteristics assumptions, allows a high-level planner to generate high-risk attack plan options. This approach uses a broad range of available information, including known characteristics of the device being evaluated, uncertainties in device properties, repositories that support attack surface operators (e.g. Vulnerabilities [8]), the device class architecture, and adversarial goals and strategies.

The planning engine supports checking for invalid preconditions of tactics. When constructing a plan, invalid preconditions can lead to backplanning to satisfy the preconditions, or to detect a conflict between the results in one subplan and the preconditions for another. If there is not enough data to determine whether a precondition is met, the appropriate analysis must be invoked. The tactics being considered shape the network analysis to be performed, and the results observed are folded into the cost calculations to direct the search. The VDS approach is a mixed initiative, using Alloy for enumerated elements, and mapping the Alloy representation to appropriate base classes.

Alloy takes the plan fragments and attack scenarios from the HTN planner and creates a minimally complete plan. An Activity Estimator uses the HTN's leaf nodes to create an initial Activity Graph, which embodies at least one chain of causally ordered qualitative states that connect the start state and a goal state.

Alloy generates every possible outcome for each individual activity, either validating the feasibility of the attack plan relative to its device model by fully instantiating the plan, or refuting the plan by demonstrating no instantiation is possible. Combining a heuristic search mechanism with Alloy enables reasoning from a diverse set of information and compensates for weaknesses of the individual planners when used in isolation. Search prioritizes high-risk candidates based on a combination of vulnerability related data drawn from general attack/adversary goals, and known vulnerabilities of classes of devices and their services. Alloy uses a model of the automotive component being assessed, made specific through assumptions in the heuristic search to exhaustively check for existence of a sequence of AP-level actions that can achieve adversary goals. This serves to answer the question "is there a way to perform the given actions even if the preconditions are false?". (Formally, if the System Model is *Sys* and Properties are P, Alloy tries to satisfy $Sys \wedge \neg P$).

Given a set of feasible, concrete plans generated by Alloy, combined with the calculated likelihoods of different configurations, the VDS produces a checklist that is ordered by risk and likelihood. For the achievable attacker goals that pose the greatest risk, the VDS groups together attack plans achieving similar goals, and identifies required configuration settings.

The primary value of the Knowledge Acquisition and Attack Plan Steps is to generate a checklist as an input to focus more detailed analysis. This checklist is prioritized and actionable to make best use of limited analyst resources. Approaches that perform a post-processing prioritization step would be hopelessly inefficient. The checklist that the VDS generates is in the order of likelihood of attack success.

III. Attack Verification and Feedback Generation

This Attack Verification and Feedback Generation section is made up of two parts. The first part describes a vulnerability verification method using simulation, for large scale test, and actual automotive test to verify the simulation on actual software running on the real platform. The second part performs independent verification operating solely on the real software binary code and vulnerability checklist clues.

A. Scenario Generation and Automotive Test Attack Verification and Feedback Generation

As a prerequisite, a large set of scenarios (based on real-world driving data) are abstracted into a virtual world model. This large data set is then tagged with descriptors for each scenario.

As depicted in Figure 3, Attack Verification and Feedback (AVF) generation uses the vulnerabilities identified through the feasibility and risk prioritized attack plan checklist, then selects scenarios that would expose those vulnerabilities, and generate controlled tests based on those scenarios.

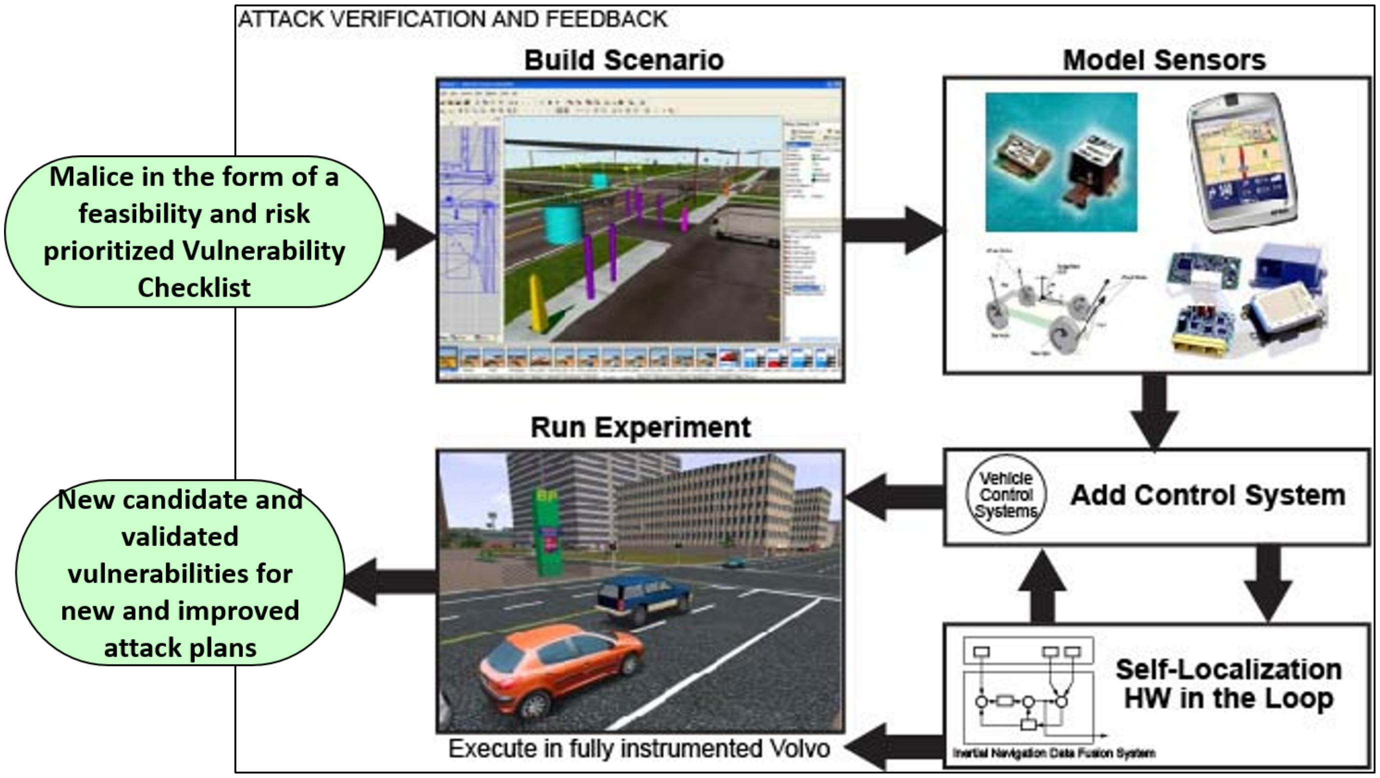

ATTACK VERIFICATION AND FEEDBACK
Build Scenario
Model Sensors
Malice in the form of a feasibility and risk prioritized Vulnerability Checklist
Run Experiment
Vehicle Control Systems
Add Control System
New candidate and validated vulnerabilities for new and improved attack plans
Self-Localization HW in the Loop
Inertial Navigation Data Fusion System
Execute in fully instrumented Volvo

Fig. 3. Generate, verify and prune to manage and prioritize possible attack point vulnerabilities

Testing all possible conditions of a CPS in a non-cooperative environment is difficult due to the vast set of possible scenarios and failure modes. The attack plan checklist selects scenarios that would expose those vulnerabilities and generate controlled tests. By combining this technology with massively parallel computation, storage system, and simulation technology, the VDS generates parameterized scenarios based on real-world data, and runs them in parallel.

Driving tests, and simulation systems that run a multiple scenario closed-loop simulation with a System Under Test (SUT) are executed. The test generator selects test cases out of the collection that likely expose predicted vulnerabilities. These cases are then run using a driving and sensor simulator (PreScan by TASS [12]), a model of the vehicle (CPS) under test (Matlab/Simulink) and a vehicle dynamics model in a closed loop. Simulation results (pass, fail, near miss, emergency, etc.) are fed back to the test generator where modified test scenarios are generated creating the outer loop. Well-defined interfaces of separately pluggable independently developed modules are used to enable scaling and modifying of applications independent of real or simulated sensors. Figure 4 shows Simulink correspondence to existing automotive software that runs the same on computer and test-car test benches. Both positive and negative test bench results are returned to the Attack Plan Generator, with explanation, to improve the HTN and SAT planner process.

In our second use case the SUT is exposed to an external attack. In this case the proposed system generates attacks scenarios on the systems.

In these cases the "real-world" scenarios are taken from the recorded sample set, but then the "test" scenarios are modified by the attacker's approach in the simulator (for example a target appears in different locations for the radar sensor and camera sensor, or a GPS drift is imposed). The test generator then samples the parameter space of this scenario based on results, and randomly varies the scenario (thousands of different variations) to try to expose vulnerabilities in the system. Using the ARE in this step greatly reduces the search space. The found vulnerabilities are then fed back to the system and/or user for further analysis.

The parameter space for these systems is so vast that a brute force method for securing and proving correctness is not feasible. With this approach the search space is reduced to manageable dimensions while maintaining "full" test coverage.

B. *Automated Reverse Engineering (ARE) for Attack Verification and Feedback Generation*

The Automated Reverse Engineering (ARE) system automatically searches software binaries for candidate software vulnerabilities and determines which candidate vulnerabilities can be exercised through external system input to realize a cyber effect such as a crash or exploit [13][14]. The ARE both provides candidate vulnerabilities to Alloy for attack plan generation and verifies whether a candidate attack plan can be realized against the real System under Test (SUT). The ARE is a whitebox system that uses modern constraint solvers to explicitly compute a set of external inputs that

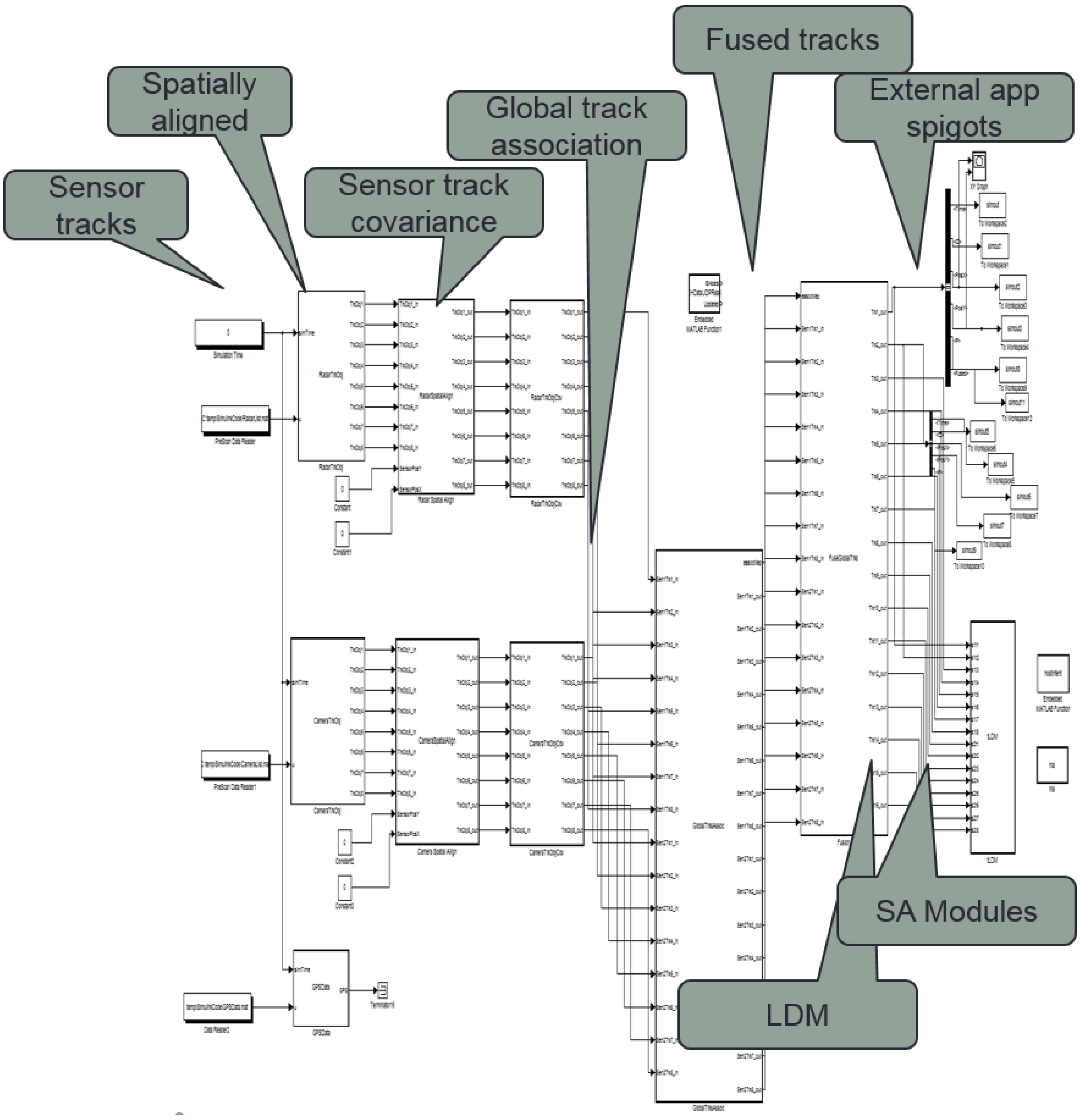


Fig. 4. Executable Simulink Specification is System Under Test that runs on simulation and automotive testbeds

realize each unique execution or control-flow path through the system software. The ARE thus is much more efficient at searching the control-flow space than blackbox systems, such as traditional blackbox fuzzers [15], which simply probe the SUT with inputs drawn from the input space according to some random or other process. More importantly, since the ARE methodically explores each unique control-flow path, it can demonstrate conclusively that a vulnerability of a given type is either not present or cannot be exercised through the external inputs of interest. The ARE *is particularly well suited to the automotive domain where well-defined interfaces and software processing lead to simpler constraint problems than are seen with applications such as Web servers or clients.*

In contrast to other whitebox systems such as Microsoft SAGE [16], [17] the ARE uses statically identified Points of Interest (POIs) to quickly prune paths that ultimately will not be of interest. For example, an attacker-controllable buffer overflow cannot exist without an indexed memory operation, *i.e., base address + index*, where the feasible values of the index are in some way determined or *tainted* by external input. For buffer-overflow analyses, the ARE statically identifies indexed memory operations where the index is at least tainted by something outside the local function as the POIs. During the subsequent dynamic analysis, the ARE uses the POIs and the software's overall call graph to prune paths as soon as it becomes clear that they cannot possibly lead to one of the POIs. This POI-driven approach, in early testing, has shown order-of-magnitude better performance than simply exploring every path without reference to POIs.

Figure 5 shows the architecture and components of the ARE system. The execution environment is a real or virtualized environment in which the software binaries of interest can be executed under real-world conditions.

The ARE uses the execution environment to obtain a state image for the SUT at the point where the SUT has reached an internal state of interest. When searching for candidate vulnerabilities, the ARE would provide a normal input to the SUT and capture a state image when the software component of current interest is about to perform its processing. When validating that an attack plan can be realized, the ARE would provide inputs according to the attack plan and capture the state image when the software component containing the suspected vulnerability is about to perform its processing. In both cases, the ARE then performs its path exploration and analysis starting from the point at which it captured the state image.

The ARE does not perform its analysis against the live system but instead against two intermediate representations of the software binaries. The Power Reverse Engineering Intermediate Language (PREIL) is a platform-independent assembly language in which instructions have no implicit operands or side effects. Every operand and effect is explicitly represented and thus the PREIL translation of a single native instruction typically requires multiple PREIL instructions. The Taint Modeling Functions (TMF) are an algebraic representation of the PREIL where the variables are registers and memory locations. Algebraic simplification produces TMF expressions that can reveal POIs. For example, candidate buffer overflows have a particular signature within simplified TMF expressions.

Since the ARE performs its analysis against these platform-independent representations, the only platform-dependent components in the ARE system are the execution environment and front-end translation modules that convert native machine code to PREIL and TMF. The ARE currently has translation front ends for 16- and 32-bit x86 machine code, 32-bit MIPS machine code, and 32-bit ARM machine code. Adding a new translation module is straightforward but does involve many individual translations for a Complex Instruction Set Computer (CISC) instruction set such as x86. Many of the processors in embedded automotive systems, however, are Reduced Instruction Set Computers (RISC) and have much simpler instruction sets. The translation module for the basic MIPS instruction, for example, took only a few man weeks with each additional MIPS coprocessor adding only a few additional man weeks. Further, since instruction sets change very slowly and are typically backwards compatible with older processor versions, a translation module, once written, requires only sporadic updates and maintenance.

The Static Behavior Sensors (SBS) identify different types of POI for different types of vulnerabilities. The set of POIs is the set of Basic Blocks (BBs) of which a control-flow path must reach at least one for a vulnerability of a certain type to even potentially exist. An exploitable buffer overflow, as noted above, cannot exist without a potentially tainted memory-address calculation. Alternatively, unintended information leakage in a network-transmitted response cannot exist without an OS system call that actually places the response onto the network. In this case, the POIs are BBs that contain a transmission-related system call. Given a set of POIs, the ARE uses the software call- and control-flow graphs to define a set of the paths that the codes structure allows. This set is a superset of realizable paths since, although the code structure may allow a particular path, data-driven decisions may make the particular path impossible. The superset, however, allows the ARE to define a super-subset of paths that reach one of the POIs. If a path under exploration deviates from the super-subset, the ARE can immediately prune that path during its dynamic-analysis steps. In this way, the ARE can consider up to orders of magnitude fewer paths than actually exist in the software while still comprehensively evaluating the software for the given vulnerabilities. This efficiency is particularly important in the safety-critical automotive domain that demands an extremely large number of tests and analyses and must provide six-nines likelihood that certain failures and attacks cannot occur.

The ARE's dynamic analysis, which is to the right of the dashed line in Figure 5, actually explores the control-flow paths. A planner guides the overall process. Starting with a random or other bootstrap input, the planner obtains a first control-flow trace with a PREIL-based emulator. A taint-analysis engine identifies the input-tainted conditional

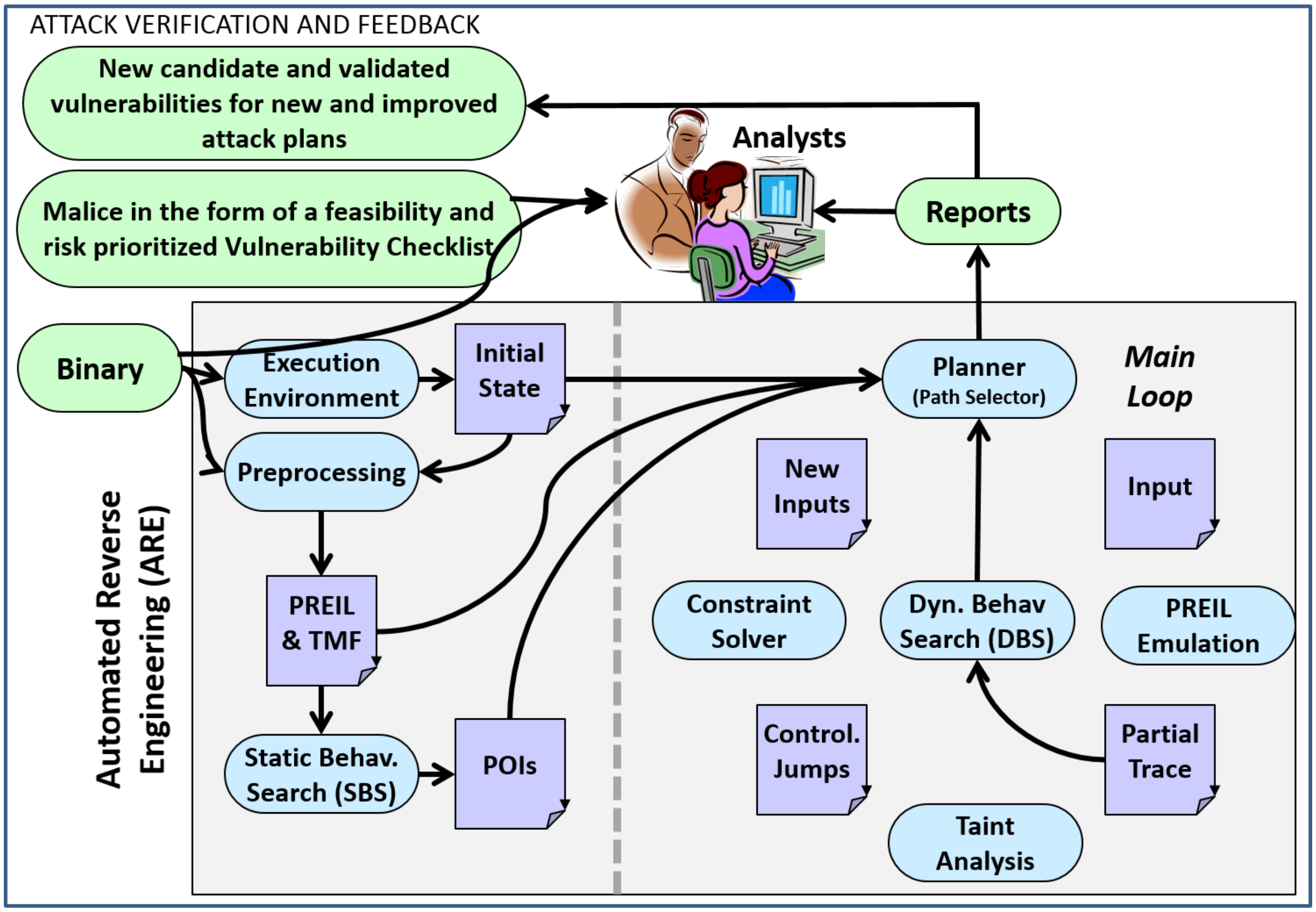
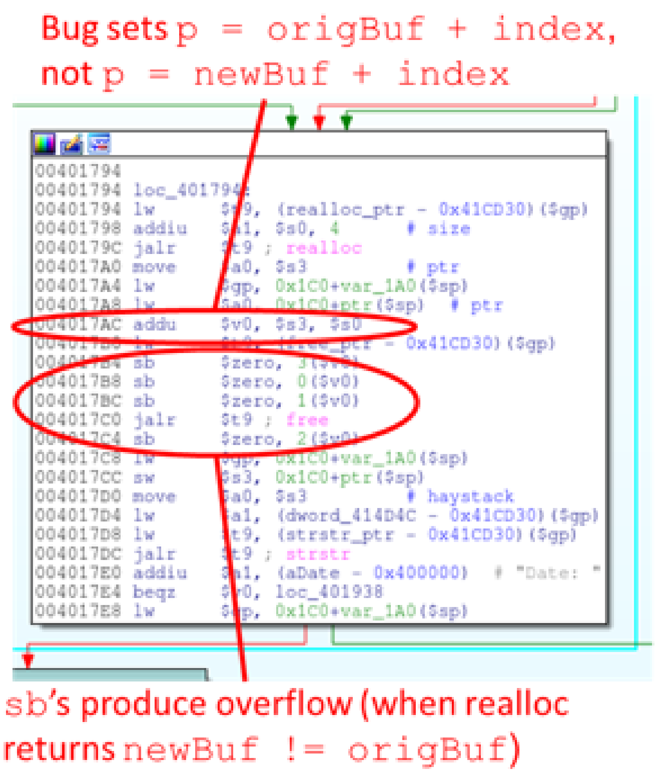

ATTACK VERIFICATION AND FEEDBACK
New candidate and validated vulnerabilities for new and improved attack plans
Malice in the form of a feasibility and risk prioritized Vulnerability Checklist
Analysts
Reports
Binary
Execution Environment
Initial State
Preprocessing
PREIL & TMF
Static Behav. Search (SBS)
POIs
Automated Reverse Engineering (ARE)
Planner (Path Selector)
Main Loop
New Inputs
Input
Constraint Solver
Dyn. Behav Search (DBS)
PREIL Emulation
Control. Jumps
Partial Trace
Taint Analysis

Fig. 5. The Automated Reverse Engineering (ARE) system performs both static analysis (left side of dash lines) and dynamic analysis (right side of dashed line) to identify candidate vulnerabilities and determine which candidate vulnerabilities can be exercised through external system input.

branches *i.e.*, conditional branches where the condition depends in some way on external input within the trace. These input-tainted branches, which we call controllable jumps, represent the potential for deriving new control-flow paths from the existing trace. The constraint solver, for each controllable jump, formulates and solves a constraint problem to compute a new input, if it exists, that will reach the controllable jump along the same path prefix but then take the conditional branch in the other direction. The planner thus ends up with a set of new inputs that can be used to generate new traces that, in their turn, will leads to additional inputs. The planner, of course, does significant work to prune the emerging paths that cannot possibly reach one of the POIs; avoid consideration of duplicate paths; and avoids repeated solution of constraint subproblems when possible.

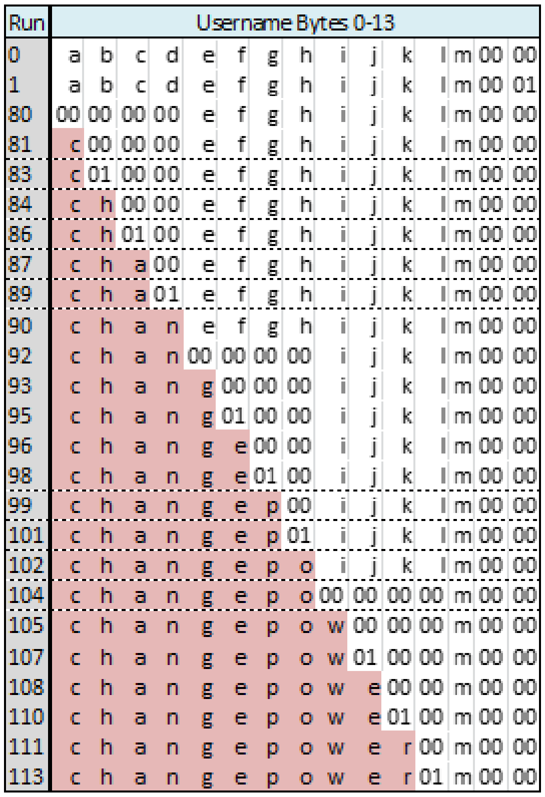
The Dynamic Behavior Sensors (DBS), which are in the middle of the planner loop in Figure 5, consider each path or trace as the planner discovers it and determines whether that path actually contains a vulnerability relative to the systems external input. This determination occurs in one of two ways. First, the path exercises the vulnerability directly. For example, in the case of a potential buffer overflow, the path being considered might overflow the buffer, something that can be directly observed. Second, the path does not exercise the vulnerability directly under the planner-computed input, but a different input exists that would follow the same path and exercise the vulnerability. For example, if the path in question indexes into valid element k of an n-element array where $k < n$, a different input might induce the same control-flow path yet index into invalid element k where $k > n$. The ARE, as it performs the taint analysis, performs full interval analysis to assess the realizable ranges of registers, pointer values, and other data. If interval analysis indicates that the vulnerability might be exercisable with a different input value, the ARE extends the constraint problem associated with the control-flow path to include additional assertions. In this example, the ARE would assert that it wants the same path but with $k > n$ at a certain point. If the expanded constraint problem has a solution, the candidate vulnerability is a real vulnerability that can be exercised through external input.

The ARE's approach of independently assessing each path has two advantages. First, it is typically extremely quick to


Bug sets p = origBuf + index,
not p = newBuf + index
sb's produce overflow (when realloc
returns newBuf != origBuf)

```
00401794
00401794 loc_401794:
00401794 lw      $t9, (realloc_ptr - 0x41CD30)($gp)
00401798 addiu   $a1, $s0, 4          # size
0040179C jalr    $t9 ; realloc
004017A0 move    $a0, $s3             # ptr
004017A4 lw      $gp, 0x1C0+var_1A0($sp)
004017A8 lw      $a0, 0x1C0+ptr($sp)  # ptr
004017AC addu    $v0, $s3, $s0
004017B0 lw      $t9, (free_ptr - 0x41CD30)($gp)
004017B4 sb      $zero, 3($v0)
004017B8 sb      $zero, 0($v0)
004017BC sb      $zero, 1($v0)
004017C0 jalr    $t9 ; free
004017C4 sb      $zero, 2($v0)
004017C8 lw      $gp, 0x1C0+var_1A0($sp)
004017CC sw      $s3, 0x1C0+ptr($sp)
004017D0 move    $a0, $s3             # haystack
004017D4 lw      $a1, (dword_414D4C - 0x41CD30)($gp)
004017D8 lw      $t9, (strstr_ptr - 0x41CD30)($gp)
004017DC jalr    $t9 ; strstr
004017E0 addiu   $a1, (aDate - 0x400000)  # "Date: "
004017E4 beqz    $v0, loc_401938
004017E8 lw      $gp, 0x1C0+var_1A0($sp)
```

Fig. 6. A real ARE-discovered buffer-overflow flaw where the programmer uses the old pointer value rather than the potentially new pointer value from *realloc*



Run	Username Bytes 0-13														
0	a	b	c	d	e	f	g	h	i	j	k	l	m	00	00
1	a	b	c	d	e	f	g	h	i	j	k	l	m	00	01
80	00	00	00	00	e	f	g	h	i	j	k	l	m	00	00
81	c	00	00	00	e	f	g	h	i	j	k	l	m	00	00
83	c	01	00	00	e	f	g	h	i	j	k	l	m	00	00
84	c	h	00	00	e	f	g	h	i	j	k	l	m	00	00
86	c	h	01	00	e	f	g	h	i	j	k	l	m	00	00
87	c	h	a	00	e	f	g	h	i	j	k	l	m	00	00
89	c	h	a	01	e	f	g	h	i	j	k	l	m	00	00
90	c	h	a	n	e	f	g	h	i	j	k	l	m	00	00
92	c	h	a	n	00	00	00	00	i	j	k	l	m	00	00
93	c	h	a	n	g	00	00	00	i	j	k	l	m	00	00
95	c	h	a	n	g	01	00	00	i	j	k	l	m	00	00
96	c	h	a	n	g	e	00	00	i	j	k	l	m	00	00
98	c	h	a	n	g	e	01	00	i	j	k	l	m	00	00
99	c	h	a	n	g	e	p	00	i	j	k	l	m	00	00
101	c	h	a	n	g	e	p	01	i	j	k	l	m	00	00
102	c	h	a	n	g	e	p	o	i	j	k	l	m	00	00
104	c	h	a	n	g	e	p	o	00	00	00	00	m	00	00
105	c	h	a	n	g	e	p	o	w	00	00	00	m	00	00
107	c	h	a	n	g	e	p	o	w	01	00	00	m	00	00
108	c	h	a	n	g	e	p	o	w	e	00	00	m	00	00
110	c	h	a	n	g	e	p	o	w	e	01	00	m	00	00
111	c	h	a	n	g	e	p	o	w	e	r	00	m	00	00
113	c	h	a	n	g	e	p	o	w	e	r	01	m	00	00

Fig. 7. The ARE quickly identifies a hard-coded backdoor username and associated Trojan code that a test team inserted into the real *dropbear* executable from a software router.

solve the extended constraint problem starting from the current input solution. Second, since a vulnerability must exist along a realizable path for it to be a threat, considering the realizable paths independently does not hinder our ability to conclude that a vulnerability is not present in the software.

The ARE, through its overall approach, addresses many different types of vulnerabilities and malicious code extremely well:

- Memory. Improper access to memory, including buffer and stack overflows. Figure 6 shows an ARE-discovered software flaw where the programmer has used *realloc* to obtain a larger buffer but then inadvertently used the old buffer pointer when writing data. In cases where *realloc* could not expand the existing buffer in place, this software flaw produces a buffer overflow as the code writes past the end of the old (smaller) buffer.
- Information leakage. Sensitive information, as defined by an analyst, reaches or taints external output.
- Parameter manipulation. Parameters to dangerous functions, such as `fprintf`, or system calls, such as `execve`, are input tainted. An analyst would evaluate ARE-identified parameters further to determine if the input-tainted parameters have been appropriately sanitized.
- Safety violations. There exists a control-flow path along which the software performs an unexpected action, such as communicating with a component that should not be a communication partner. There exists a control-flow path that reaches function F without going through function G. For example, F might be a function that changes a braking configuration parameter, and G might be an authentication routine that verifies that the parameter-change command can come only from the primary processor. The analyst identifies functions F and G and can impose additional conditions. In this example, reaching function F even though function G returns ACCESS_DENIED is still a safety violation.
- Liveness violations. There exists a control-flow path in which the software is no longer exercising (critical) function F in response to external input. The analyst identifies function F.
- Backdoors and other Trojan code. There exists a control-flow path that makes a decision based on an unexpected characteristic/feature of the external input. For example, if the software makes a decision based on a *specific hard-coded value* of a username or password field, that decision is intrinsically suspicious and worthy of immediate attention, Figure 7 shows an example of exactly this case where the ARE discovered a backdoor username that a test team had inserted into a real router executable. The ARE reveals the username, one character at a time, as it solves for inputs that satisfy the conditional branches

associated with recognizing the username. Although the backdoor username happens to be human-readable in this case, the only thing that matters to the ARE is the existence of a control-flow path that expects a specific username (as opposed to normal control-flow paths that compare a username against a set loaded from a configuration file or other source). The domain analyst can define the unexpected input-driven decisions for the ARE to evaluate.

- Integer overflow/underflow and divide by zero. The ARE, through its taint analysis, interval analysis, and constraint solvers, is uniquely suited to identify potential problems with integer calculations. For example, if external input can produce a denominator of 0, the software may crash or freeze when the division is performed. More interestingly, the ARE is able to identify the multiplication-produced integer-overflow vulnerabilities associated with the recent Android STAGE FRIGHT vulnerabilities. The ARE does not yet have the code to address floating-point issues but ultimately can address floating-point problems in analogous ways.

The ARE has performed extremely well against many software executables but is still under active development across multiple projects. Currently the team is focused on the system engineering needed to (1) reduce developer effort when implementing new front-end translation modules and providing syscall models for the PREIL emulator and (2) perform complete analyses overnight rather than over several days. These enhancements will make it much easier to apply the ARE to new platforms, e.g., proprietary embedded systems in the automotive domain, and to fit the ARE into existing Test and Evaluation (T&E) procedures and timeframes.

As is and as it evolves further, the ARE can play exactly the intended role in our larger approach. An attack plans provide bootstrap inputs and define POIs, e.g., this section of particularly critical code. The ARE determines if the attack plan can realize a real attack. Conversely, the ARE's existing analyses identify vulnerabilities that can be used to generate additional attack plans. The resulting iterative process can analyze system safety, security, and liveness across as comprehensive a set of attack plans as desired.

IV. Conclusion

We have described a method to digest online and manufactured attack-related technical data to proactively secure automotive electrical systems using an Attack Plan Generator that combines a hierarchical task network planner to efficiently hypothesize likely attack scenarios with planning, formal model checking to prune infeasible attacks, and ARE to prune the software search space. While we are in the process of developing each of the related components, they are at varying levels of maturity (e.g. the Semantic Alignment Mediator [5] is the most mature), and we are identifying new development opportunities to connect the research "dots" to bring the Knowledge Acquisition and Attack Plan Generator vision into practice.

V. Acknowledgements

The development of the Automated Reverse Engineering (ARE) system is supported by Defense Advanced Research Projects Agency (DARPA) and SPAWAR Systems Center Pacific (SSC Pacific) under Contract No. N66001-13-C-4047.

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions, and/or findings contained in this article/presentation are those of the author(s)/presenter(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

The authors extend their deep gratitude to the ARE R&D team, Ms. Quy Messiter, Dr. Vu Le, Dr. Rob Ross, Mr. Michael Weber, Mr. Gregory Sadosuk, Mr. Terry Ware, Mr. Haris Stewart, Mr. Christopher Cash, Mr. Carl Washington, and Mr. Kris Ambrose, and to Mr. John (JH) Berry from Cromulence LLC, who has provided feedback on ARE usability from an expert analysts perspective.

References

[1] J. Smith and M. Figueroa, "Reduced realistic attack plan surface for identification of prioritized attack goals," in *2013 IEEE Homeland Security Conference*, Nov. 2013.

[2] D. Jackson, *Software Abstractions – Logic, Language and Analysis*. MIT Press, 2011.

[3] J. Smith, B. Krikeles, D. K. Wittenberg, and M. Taveniku, "Applied vulnerability detection system," in *Proceedings of the IEEE International Conference on Technologies for Homeland Security*, 2015.

[4] C. Miller and C. Valasek, "Adventures in automotive networks and control units," *DEF CON*, vol. 21, pp. 260–264, 2013. [Online]. Available: http://illmatics.com/car_hacking.pdf

[5] BAE Systems, "Mediation, alignment, and information services for semantic interoperability (MAISSI): A trade study," Air Force Research Laboratories, Tech. Rep. AFRL-IF-RS-TR-2007-147, June 2007.

[6] The MITRE Corporation, "Common vulnerabilities and exposures (CVE)." [Online]. Available: http://cve.mitre.org

[7] NIST, "National vulnerability database (NVD)." [Online]. Available: http://nvd.nist.gov

[8] "Automakers announce initiative to further enhance cyber-security in autos," 2015. [Online]. Available: http://www.autoalliance.org/index.cfm?objectid=8D04F310-2A45-11E5-9002000C296BA163

[9] M. Ring, J. Dürrwang, and R. Kriesten, "Building an automotive vulnerability database survey and tools," in *ESCAR*, 2015.

[10] O. Sheyner and J. Wing, "Tools for generating and analyzing attack graphs," in *2nd Intl. Symposium on Formal Methods for Components and Objects*, ser. LNCS, no. 3188. Springer Verlag, 2004.

[11] The MITRE Corporation, "Common attack pattern enumeration and configuration (CAPEC)." [Online]. Available: http://capec.mitre.org

[12] "PreSCAN product page," 2014. [Online]. Available: https://www.tassinternational.com/prescan

[13] B. Ross, "Are: A system for automated reverse engineering," Poster Session, High Confidence Systems and Software Conference, 2013.

[14] V. Lee, Q. Messiter, R. Ross, and G. Sadosuk, "ARE: Automated reverse engineering of machine," BAE Systems CCTR (Cyber and Communications Research) White Paper, 2014.

[15] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.

[16] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166. [Online]. Available: http://research.microsoft.com/en-us/um/people/pg/public_psfiles/ndss2008.pdf

[17] ——, "SAGE: White box fuzzing for security testing," *CACM*, vol. 55, no. 3, March 2012.