ORNL/TM-2022/2524

Automated Vulnerability Detection (AVUD) for Compiled Smart Grid Software

Stacy Prowell
Kirk Sayre
Mark Pleszkoch
Richard Linger
Richard Willems
David Heise
Joel Reed
Kelly Huffer
Ada Lindberg

July 2022

ORNL/TM-2022/2524

Cyber Resilience and Intelligence Division

FINAL REPORT: AUTOMATED VULNERABILITY (AVUD) DISCOVERY FOR COMPILED SMART GRID SOFTWARE

Stacy Prowell
Kirk Sayre
Mark Pleszkoch
Richard Linger
Richard Willems
David Heise
Joel Reed
Kelly Huffer
Ada Lindberg

July 2022

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831
managed by
UT-BATTELLE LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR2272

CONTENTS

ABSTRACT

This project developed and implemented a system for conducting cybersecurity vulnerability detection of smart grid components and systems by performing static analysis of compiled software ("firmware"). The resulting system for automated vulnerability detection (AVUD) was implemented as part of Oak Ridge National Laboratory's existing test bed for smart meters, the Sustainable Campus Initiative. The work consisted of two phases: the first phase implemented the necessary software and computational models to perform the analysis, and the second phase demonstrated the system on example firmware in partnership with smart meter manufacturer Sensus USA, Inc. The resulting system won an R&D 100 award and has been successfully commercialized, winning a National Laboratory Consortium Commercialization Award.

1. SUMMARY

This project developed and implemented a system for conducting cybersecurity vulnerability detection of smart grid components and systems by performing static analysis of compiled software ("firmware"). The resulting system for automated vulnerability detection (AVUD) was implemented as part of Oak Ridge National Laboratory's existing test bed for smart meters, the Sustainable Campus Initiative. The work consisted of two phases: the first phase implemented the necessary software and computational models to perform the analysis, and the second phase demonstrated the system on example firmware in partnership with smart meter manufacturer Sensus USA, Inc. The resulting system won an R&D 100 award (2015) and has been successfully commercialized, winning a Federal Laboratory Consortium (FLC) Commercialization Award (2016).

2. MOTIVATION

Current- and next-generation energy delivery systems (EDS) are *software defined* devices. That is, the capabilities of these devices are largely determined by their software and firmware. For example, a meter may contain electromechanical systems that are used to detect electricity usage, but the *recording and reporting* of the usage is entirely implemented by software. Software in these devices requires remote connections, and, because software vulnerabilities are inevitably detected after deployment, also require remote updates. Deployed vulnerabilities in a meter are a source of potential compromise, but ability to update a meter's software remotely (to address the deployed vulnerabilities) is a source of potential compromise. Attacks on the power grid exploiting vulnerabilities in components are not hypothetical.[1]

When this effort began, Supervisory Control and Data Acquisition (SCADA) systems represented a subset of the space considered. A 2012 report by Symantec report identifies 85 publicly disclosed SCADA vulnerabilities.[2] These are vulnerabilities that cover a range of severities but include the ability to compromise control system software by bypassing access controls. This number has since grown significantly, and Symantec's latest (2021) threat report no longer calls them out explicitly, but instead notes trends such as ransomware and supply chain attacks.[3]

[1] See NERC Lesson Learned "Risks Posed by Firewall Firmware Vulnerabilities," September 2019, online: https://legacy-assets.eenews.net/open_files/assets/2019/09/06/document_ew_02.pdf, accessed June 20, 2022.
[2] "Internet Security Threat Report," Symantec Corporation, 2013, online: https://www.insight.com/content/dam/insight/en_US/pdfs/symantec/symantec-corp-internet-security-threat-report-volume-18.pdf, accessed June 20, 2022.
[3] "The Threat Landscape in 2021," Symantec, Inc. (division of Broadcom, Inc.), online: https://symantec.drift.click/Threat_Landscape_2021_Whitepaper, accessed June 20, 2022.

Once deployed, vulnerabilities can persist in deployed systems for months *even after they are discovered.* This issue can be partially addressed by rigorous testing, conducted under proper experimental control, such as the testing performed by the NSTB. Unfortunately testing alone is insufficient to catch all vulnerabilities because testing cannot be exhaustive of the combinatorial input space of software. There is thus a need for rigorous static analysis methods that consider the entire input space of software.

Even rigorous design and coding methods cannot eliminate vulnerabilities. Otherwise, non-malicious compilers themselves can insert vulnerabilities into otherwise secure code through optimization.[4] There is thus a need for analysis methods that work with the compiled software.

3. OVERVIEW

The AVUD system directly consumes the compiled software that is part of the device and performs "behavior computation." This is a process whereby all behaviors of the system (externally observable events) are computed, along with the conditions under which those behaviors manifest. This resulting behavior catalog is challenging to read and understand, but it can be rewritten in terms of external application program interface (API) invocation sequences. While this does not make it easier for humans to understand, it does enable the specification of *behavior patterns* that can be matched no matter how they occur in the software. What constitutes a vulnerability depends, in part, on the system, the context in which it is used, etc. Behavior patterns are then created that allow the detection of inappropriate behavior, or the confirmation of correct behavior.[5]

Because the AVUD system analyzes compiled software, it largely *does not matter* how the vulnerabilities were introduced. This analysis of the compiled software is as close to "ground truth" as one can (reasonably) get. While it does not address processor vulnerabilities or misconfiguration, behavior patterns can be created that detect the related software exploits.

The AVUD system was demonstrated on Sensus USA, Inc., smart meter's firmware, and the code subsequently reviewed by Sandia National Laboratory. Following this, the software system was renamed "Hyperion" and won an R&D 100 Award in 2016.[6] The Hyperion software was subsequently licensed exclusively by R&K Cyber Solutions, LLC[7], with the promise to apply this in multiple sectors, including specifically the energy delivery sector. The technology was spun out of R&K into its own company, Lenvio, Inc., subsequently renamed to CodeHunter, Inc.[8] The commercialization efforts of the AVUD team and ORNL won a Federal Laboratory Consortium (FLC) Commercialization Award.[9]

[4] X. Wang, *et. al.*, "Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior," to be presented at 24th Symposium on Operating Systems Principles (SOSP'13), Farmington, Pennsylvania, November 2013, online: http://pdos.csail.mit.edu/~xi/, accessed June 20, 2022.
[5] See S. J. Prowell, M. Pleszkoch, K. D. Sayre and R. C. Linger, "Automated vulnerability detection for compiled smart grid software," *2012 IEEE PES Innovative Smart Grid Technologies (ISGT)*, 2012, pp. 1-5, doi: 10.1109/ISGT.2012.6175814.
[6] See "ORNL Wins Six R&D 100 Awards," online: https://www.ornl.gov/news/ornl-wins-six-rd-100-awards, accessed June 20, 2022.
[7] https://rkcybersolutions.com/
[8] https://www.codehunter.io/
[9] See "Commercial Licensing of the Hyperion Cyber Security Computer Code," FLC Award, 2016, online: https://federallabs.org/successes/awards/awards-gallery/2016/commercial-licensing-of-the-hyperion-cyber-security-computer, accessed June 20, 2022.

4. APPROACH

Directly analyzing compiled software and computing the complete behavioral catalog is a significant challenge. The AVUD team relied on three key characteristics to reduce the technical risk of the effort.

- *Feasibility*. The AVUD work was based on existing, successful work ("function extraction" or FX) performed at the SEI for analysis of malware on a Windows / Intel platform.[10] Further, AVUD's vulnerability detection capability is also based on published research on computational modeling of cybersecurity properties.

- *Soundness*. The AVUD approach was not just based on prior success with the same technology ("function extraction"), but on the inclusion of several of the key personnel from this prior technology.[11,12,13]

- *Reasonableness*. The AVUD effort provided a significant advantage over vulnerability testing. Tests only provide information about the specific scenarios executed during testing, whereas static analysis of the software can provide information about *any* scenario of use. Further, AVUD does not require hard-to-get source code but can operate directly from the compiled firmware present on the device.

The AVUD system operates by executing the following sequence of operations.

1. Identify the code to analyze. This may require extraction of firmware from a device, or the acquisition of the "binary blob" to be loaded on a device.
2. Identify the APIs that are needed to understand the software operation. These include operating system APIs as well as specialized library APIs and also interfaces to physical devices.
3. Perform a "threaded disassembly" of the compiled software.[14]
4. Structure the disassembled code. This produces a static control flow graph of the program.
5. Annotate each instruction with its complete functional effect.
6. Compute the overall functional effect for each higher-level control structure.
7. Rewrite the resulting behavior catalog to expose the call sequence of external APIs.
8. Perform pattern matching on the resulting call sequences to detect behavior of interest.

Each of these operations is discussed in more detail below.

[10] The Java source code for the FX system was subsequently made open source by the SEI and served as a part of the basis for the AVUD effort. The open source codebase does not appear to be available at the time of writing.
[11] See S. Prowell, M. Pleszkoch, L. Burns, T. Daly, R. Linger, K. Sayre, "Function Extraction Technology for Software Assurance," *CERT Research Report for 2008,* SEI CERT, online: https://resources.sei.cmu.edu/asset_files/CERTResearchReport/2008_013_001_54241.pdf, accessed June 20, 2022.
[12] See M. Pleszkoch, S. Prowell, C. Cohen, and J. Havrilla, "Applying Function Extraction (FX) Techniques to Reverse Engineer Virtual Machines," *CERT Research Annual Report for 2009*, SEI CERT, online: https://resources.sei.cmu.edu/asset_files/CERTResearchReport/2009_013_001_51315.pdf, accessed June 20, 2022.
[13] See K. Sayre, M. Pleszkoch, T. Daly, R. Linger, and S. Prowell, "Function Extraction for Malcious Code Analysis," *CERT Research Annual Report for 2009,* SEI CERT, online: https://resources.sei.cmu.edu/asset_files/CERTResearchReport/2009_013_001_51315.pdf, accessed June 20, 2022.
[14] Threaded disassembly is a technique to overcome certain anti-disassembly techniques. By branching within a program it is possible to re-interpret program bytes as either an *opcode* or an *operand*, confusing disassemblers and producing a disassembly that is *not what the processor will execute*. Threaded disassembly follows the control flow and incorporates this information into disassembly to avoid this issue.

4.1 IDENTIFY THE CODE TO ANALYZE

It may seem trivial to identify the code to analyze, but this is not the case. All software systems live in a computer *ecosystem* that includes the following.

- Other software systems such as operating systems and external libraries may be *considered* out of scope for analysis.
- There may be "hidden" firmware (for instance, in baseband processors) that is *necessarily* out of scope for analysis because it is, for practical purposes, inaccessible. For example, a microcontroller may contain internal ROM or FLASH memory that is protected by both legal and physical constraints, and that cannot be accessed by conventional means.
- Finally, modern devices live in an environment of *connectivity* with other systems. The analysis of the code will usually stop at the network boundary.

Here we limit our scope as follows. We consider *programs* to be analyzed to be a single deployed binary (a single file), stopping at the interface to the network, to the operating system (if present), and the interface to the hardware. This means we have three key interfaces that we must capture and understand, in some way, to understand a complete program.

- External library APIs
- Operating system APIs, including system calls
- Hardware interfaces, primarily in terms of direct memory access (DMA) and interrupt handlers.

4.2 IDENTIFY APIS

While the prior step allows us to draw a boundary around the code to analyze, it does not mean we can simply ignore external code. The AVUD system specifically developed software to import API definitions from C header files and even from Microsoft library definition files.[15] For some environments, interrupts and memory maps must also be included. For example, the Sensus smart meter discussed previously required establishing these definitions.

4.3 PERFORM THREADED DISASSEMBLY

The AVUD system initially relied on the disassembler created as part of the open-source FX project from the SEI mentioned previously, but this was discarded in favor of the diStorm open-source library for X86 binaries,[16] and a customized disassembler targeting the MSP-430 processor. The latter was necessary because the Sensus smart meter used the MSP-430 processor.[17]

Threaded disassembly requires that the disassembler be able to be called from within an exploration loop, targeting specific addresses in the code. This is complicated by the need to translate between *virtual* addresses and *offsets* in the file, but this is typically taken care of by the disassembler itself. The basic exploration loop for AVUD works as follows. The starting address (the "entry point") is pushed onto the stack and then, while the stack is not empty, the next address is popped and the system explores that address, accumulating program function as it goes. When a conditional branch, a jump, or another

[15] A "header" file specifies the portion of the external API necessary for program linkage, allowing a compiler to understand how to organize inputs and outputs for an external code library.
[16] https://github.com/gdabah/distorm
[17] At the time of the AVUD project, the NSA Ghidra tool was not publicly available, but does provide disassembly for both the X86 and MSP-430 instruction set architectures.

modification to the instruction pointer is encountered, the accumulated program function is "sliced"[18] on the program counter and any resulting value(s) are pushed onto the stack. This process continues until the stack is empty. While actual static program flow is, in general, undecidable,[19] in practice program control flow is computable by direct means.

4.4 STRUCTURE THE DISASSEMBLED CODE

Analysis of a computer program is complicated by unseen, improperly understood, and complicated control flow. This is the well-known "spaghetti code" problem. AVUD addresses this by computing a *structured program* from the disassembly. A structured program is one that is composed of a finite set of specific, a priori structures. For AVUD, these are the following.

- *Sequences*. These are simple sequences of instructions or other structures that are executed sequentially.
- *Alternation*. These are "if-then-else" structures that check a predicate on program state and then choose one of two possible paths based on that predicate.
- *Looping*. These execute a body of code *while* or *until* a condition is met.
- *Cases*. This is another form of alternation where there are many possible paths (more than two).

AVUD converts the program into a structured program for two primary benefits.

- The program control flow is made *explicit*. This means it is possible to study a small portion of the code, like a single structure, without reference to other parts of the code, enabling a "divide and conquer" approach to analysis.
- Subsequent analysis can be *structure specific*. That is, the problem of analyzing *any* control flow has been reduced to analyzing a few specific types of control flow.

4.5 ANNOTATE EACH INSTRUCTION WITH ITS COMPLETE FUNCTIONAL EFFECT

AVUD uses *function-theoretic* analysis. Each instruction in the program is treated as a mathematical function mapping one machine state to a new machine state. We can think of the machine state as consisting of all processor state (typically registers), memory, pending input and generated output. A single instruction modifies that state in a predictable way. For example, the x86 instruction "add eax, 1" adds one to the 32-bit value stored in the "eax" register. Other registers are left unchanged except for the status register, where bits are modified based on the outcome, and the instruction pointer, which is modified to point to the next instruction's address.

As this is a labor-intensive process, the AVUD project created a higher-level language that could be used to tersely describe the complete functional effect of an instruction. This language could then be "compiled" to generate the behavior catalog for a processor. Catalogs for the X86 and MSP-430 instruction set architectures (ISAs) were created for the project.[20]

[18] Program slicing is a well-known tactic to reduce program complexity by focusing on a specific aspect of behavior. In this case the program's behavior itself is sliced to reveal only the portion that relates to the program counter.
[19] Computing the static program flow can require computing the destination of a computed jump. While this is often easy and usually even trivial, it can require determining the final value of an arbitrary computation. In pure static analysis this can be reduced to the halting problem, and it therefore undecidable. For cases where static analysis does not reveal a clear answer, AVUD falls back on simulating the computation with techniques like loop unrolling.
[20] The approach used here is surprisingly similar to the approach used by the NSA Ghidra tool, which was not publicly available at the time. The Ghidra tool uses PCode for essentially the same purpose.

4.6 COMPUTE THE OVERALL FUNCTION OF EACH STRUCTURE

The structured program can be thought of as a tree, with the individual instructions as leaves and internal nodes (places where there are branches) as the structures. For example, an if-then-else node has three children: the predicate, the "then" part, and the "else" part. Simple rules are used to combine the computed behavior catalogs of the parts into a behavior catalog for the structure itself. This effort proceeds up the tree to, theoretically, the root and a behavior catalog for the entire program.

4.7 REWRITE THE CATALOG TO EXPOSE EXTERNAL APIS

AVUD truncates this analysis approach by rewriting the behavior expressions as it goes and identifying external API invocations. Since the API invocations represent an invocation and computation outside the scope of the program being analyzed, their behavior cannot be folded into the overall program behavior. Instead, the API arguments are decoded and the condition that leads to that API invocation is captured. The result of this is a behavior catalog represented as a set of disjoint (mutually exclusive) conditions leading to a sequence of API invocations. These API-based behavior catalogs are then populated up through the rest of the program.

4.8 PERFORM PATTERN MATCHING ON EXTERNAL BEHAVIOR SPECIFICATIONS

AVUD provides for the development of *Behavior Specification Units* (BSUs) that detect and label of high-level software behavior. BSUs define higher-level behavior as compositions of lower-level behavior and other BSUs. A simplified language was developed to allow practitioners to write BSUs in terms of external function calls, and this approach proved to be quite successful.

For example, a keylogger needs to accomplish two things.

- "Hook" or intercept keypresses by some means.
- "Log" or write the intercepted keypresses to storage.

We can capture this as a hierarchical BSU that does these two actions in series: hook and then log, where the result of the first action is used in the second action. We then represent each action as another BSU, continuing in this fashion until we can finally represent the behavior of a BSU entirely in terms of a *pattern* of API calls. These patterns inherently ignore actions that occur between events of interest and that do not modify an important result and can include placeholders and symbolic variables. Many malware behaviors were codified into BSUs for the AVUD project and subsequently reviewed by external parties.

4.9 REWRITING

A key insight that led to the success of AVUD was the identification of *term rewriting* as a critical part of the analysis. Whenever possible, the AVUD team attempted to avoid cases where full *theorem proving* would be needed but saw the advantage of being able to rewrite expressions as they were generated, initially for mathematical simplification, but later for API discovery, function composition, and a host of other needs. Several term rewriting libraries were explored, but, for each, cases were discovered that led to incorrect results for programs under study. For example, term rewriters treated "$x < x+1$" as simply true, while that is *not the case* for twos-complement arithmetic. Especially for malware, it was important to avoid polluting the environment with incorrect conclusions.

An open-source term rewriter, Elision, was written specifically for this case and was adopted by the AVUD team.[21]

4.10 SENSUS SMART METER

The final AVUD system was applied to the Sensus smart meter's firmware, with vulnerabilities noted reported to Sensus for remediation.

5. CONCLUSION

The AVUD project focused on direct computation of the behavior of compiled software, focusing on the firmware in a specific smart meter as a proof of concept. Software implementing this analysis was developed and applied the firmware of a Sensus USA, Inc., smart meter. This effort resulted in licensable intellectual property (IP) including two patents and copyrighted source code. This was subsequently licensed exclusively for commercialization.[22]

[21] https://elision.github.io/
[22] https://www.energy.gov/oe/articles/lenvio-inc-exclusively-licenses-ornl-malware-behavior-detection