ORNL/SPR-2024/3222

Final Report: Energy Delivery Systems with Verifiable Trustworthiness

Stacy Prowell
Ryan Shivers
Raymond Borges
Bryan Lyles
January 2024

OAK RIDGE National Laboratory

ORNL IS MANAGED BY UT-BATTELLE LLC FOR THE US DEPARTMENT OF ENERGY

ORNL/SPR-2024/3222

Cyber and Resilience and Intelligence Division

FINAL REPORT: ENERGY DELIVERY SYSTEMS WITH VERIFIABLE TRUSTWORTHINESS

Stacy Prowell
Ryan Shivers
Raymond Borges
Bryan Lyles

January 2024

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831
managed by
UT-BATTELLE LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

ABSTRACT

Energy Delivery Systems (EDS) must be verified to be free from intrusive and malicious software. One way of verifying this software is to perform device scans to detect malicious code. Because it is possible to have "fileless" malware that exists only in device (volatile) memory, offline scanning and even many forms of online scanning is insufficient for detection. This project ("Verify") addresses this need by performing direct sampling of memory during device operation to detect unexpected or modified software while not interfering with device operation. The Verify project provides a proof-of-concept of detection by random sampling combined with remote software- and timing-based attestation methods for robust detection of in-memory threats. An external review of Verify was performed by our partner, General Electric (GE), and a summary of their findings is provided.

ACKNOWLEDGEMENT

This work was performed in partnership and with advice from General Electric Research, ConEd, the Electric Power Board of Chattanooga (EPB), the Electric Power Research Institute (EPRI), FoxGuard, the National Rural Electric Cooperative Association (NRECA), Schneider Electric, TDI Technologies, and Tennessee Technological University. All funding for this project was provided by the US Department of Energy's Cybersecurity for Energy Delivery Systems (CEDS) program. This effort is CEDS Project M617000252.

1. MOTIVATION

Supervisory Control and Data Acquisition (SCADA) networks play a vital role in *all* 16 critical infrastructure sectors and are vulnerable to remote attacks. Verifying the integrity of a device's stored instructions, data, and setpoints, or determining the "what, who, how, and why" of any successful attack on a device requires memory forensics. For example, a device may be taken out of service and its persistent storage (flash memory, hard drives, solid-state drives) scanned for signatures of malicious content, or even for non-whitelisted content. This scanning is highly effective against known threats.

Fileless malware [1], [2], [3] exploits vulnerabilities in systems in order to modify an existing running process or install itself *in memory*, but leaves no trace in persistent storage. This prevents offline and many online scanning methods from detecting it, and further leaves no "forensic residue" for discovery after a breach or attack is uncovered, as volatile memory is corrupted when the device is powered off.

The critical need is to discover that (1) a memory-resident process has been modified, or (2) an unexpected process is in memory. Doing this requires direct scanning of the device memory at runtime while the device is in operation. We must also make sure that the probability of detection is sufficiently high, verify the results, and be able to attest that the results we see truly come from the device and are not manufactured by the malware to fool detection. This project represents a proof-of-concept approach that addresses these needs.

2. APPROACH

This project, "Energy Delivery Systems with Verifiable Trustworthiness" (Verify), in partnership with General Electric (GE), Schneider Electric, and the Electric Power Research Institute (EPRI), addresses the needs described in the prior section by performing memory sampling and comparison. This operates in two distinct phases: an initialization phase where a "golden" memory image is collected for each device,

and a verification phase, where the memory is periodically sampled and checked against the golden image. This is illustrated in Figure 1.

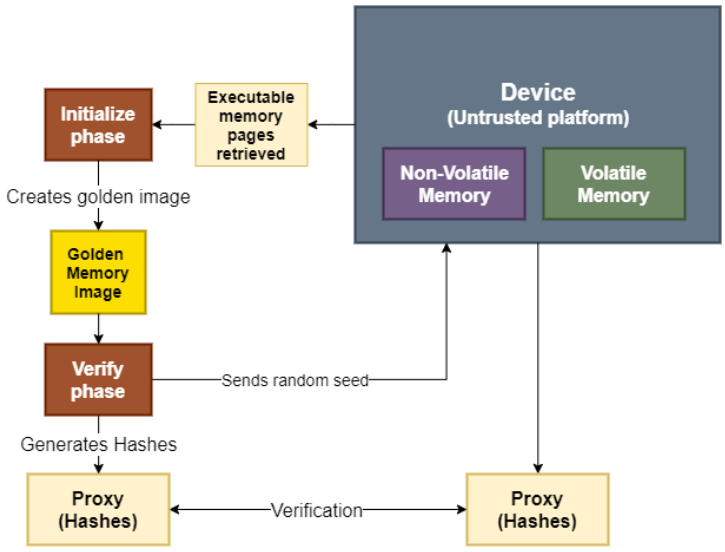

Figure 1: Verify Concept

1. On-device software is remotely triggered to perform memory sampling, and a "seed" value is provided.
2. A kernel module identifies all memory-mapped pages that are marked as "executable," meaning that the contain code that will be run natively by the processor.
3. Each executable page of memory is then randomly (based on the provided seed value) sampled, where the number of samples is chosen to allow sampling to complete with the allotted time frame.
4. The final sample is transmitted to the server, where it is checked (using the same seed value) against a "golden" image of the device constructed when the device is provisioned or updated.
5. Elapsed time both on-device and on-server is checked against expectations to attest that the result was constructed on the device and that there was insufficient time for an attacker to compute the correct result from the initial seed.
6. Failure to match flags a process as modified or unexpected, and this can be reported to a Managed Detection and Response (MDR) or Security Information and Event Management (SIEM) system, as appropriate.

To accomplish the above the Verify system relies on demonstrating the feasibility of two key items.

- Fast random sampling of executable memory in an externally reproducible manner
- Timing-based attestation of the results

Verify uses a simple client-server architecture with asymmetric encryption[1]. The high-level concept of operations of the Verify solution is as follows, also illustrated in Figure 2.

---

[1] *Asymmetric encryption* is discussed here, but *symmetric encryption using asymmetric endpoint verification*, as with secure shell (SSH) is a simple, reliable, and likely already-implemented approach and can be used. In this case the recommendation would be to start a second SSH server to isolate connections to *just* the specific use cases described here for better security.

- At provisioning or after an update, a *"golden" memory image* is taken for each monitored device. These images are stored for later use. Due to commonality of images, this information is highly compressible using existing approaches. For the Verify proof-of-concept effort we did not implement any compression.
- A *server* periodically or on demand generates a random seed and transmits this as a sampling request to connected clients. The sampling request is asymmetrically encrypted using the client's public key. The server records the precise time that each request is sent.
- The server then computes the *correct result* for each client using that client's stored golden memory image.
- Each *client* decrypts the message using their private key and notifies the Verify kernel module that immediately begins sampling memory, including any anonymous executable pages if found and the kernel itself. The sampling method is described later in this document. The result is "signed" by hashing it and including the hash.
- Upon successful completion of sampling, each client encrypts the result with the server's public key and transmits the result.
- The server records the precise time of receipt of each reply and computes the total time for the reply. This is compared to the expected time to compute the result and, if it is outside the allowed window, it flags the result as suspicious.
- The server decrypts each reply using its private key and compares it to the correct result. Any mismatch is flagged and reported.

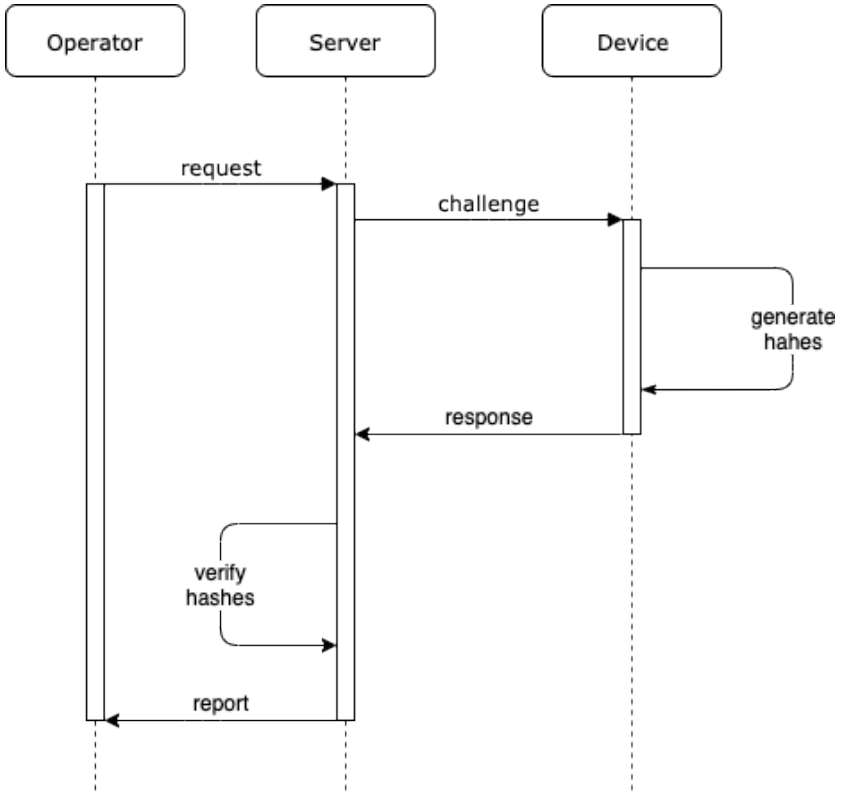

Figure 2: Client Challenge/Response (Verify Phase)

The golden image, and the client's reply, contains information about each process so that a suspicious process or a process that has been modified can be directly reported for follow-up.

2.1 FAST RANDOM SAMPLING

Random sampling prevents malware from knowing *what* will be sampled in advance, and thus increases the difficulty in "spoofing" the results. Sampling also allows for "tuning" to fit the memory scan within a given time budget by controlling the number of random samples produced. This latter item is especially important as these devices have critical control functions that should not be suspended or interrupted for scanning. The target for Verify was to fit a reasonable memory sampling event into 1/60th of a second (one "cycle" on the U.S. electric grid).

The sampling is performed randomly for two reasons.

- Scanning all of memory is time prohibitive.
- The random seed is an a priori unknown to any malware or attacker.

Code motion in memory cannot protect against detection of executable code because the sampling is random. If *all* executable code is swapped out of memory, it will either cease to run (and thus no longer be a problem) or it will have to write itself to persistent storage so it is started again in the future. In the latter case it is subject to normal detection strategies, forensics, etc.

2.1.1 Probability of Detection for a Single Sample

We can investigate the detection power of random sampling as follows. Suppose we have an n element array of bytes (memory), of which k are altered from the "golden" image. How likely are we to discover the altered content if we sample s times? If we consider sampling with replacement (the simplest case to implement), then we can model this with the Binomial distribution. For sampling without replacement, we would use the hypergeometric distribution. Because speed of sampling was paramount, and since n will be much larger than s, sampling with replacement was chosen for Verify.

Each memory sample is a Bernoulli trial; either we sample a byte that is different from the "golden" image, or we do not. We will call sampling one of the k altered bytes a *failure* and sampling any of the other n-k bytes a *success*. For a single randomly sampled byte, the probability of a failure is k/n, and the probability of a success is $(n\text{-}k)/n$.

We will see f failures among s trials with the following probability.

$$P = \binom{s}{f}\left(\frac{k}{n}\right)^{f}\left(\frac{n-k}{n}\right)^{s-f}$$

The probability of observing *no* failures, and thus not sampling any of the k altered bytes, is thus the following.

$$\begin{aligned} P &= \binom{s}{0}\left(\frac{k}{n}\right)^{0}\left(\frac{n-k}{n}\right)^{s-0} \\ &= \left(\frac{n-k}{n}\right)^{s} \end{aligned}$$

The probability that we detect at least one altered byte during our s samples is thus 1-P.

To make this more concrete, suppose a single executable page (4,096 bytes) contains 100 malicious bytes. Table 1 shows the probability of detection *for a single sampling event* for different numbers of bytes randomly sampled from the page. Because we are using sampling with replacement, this probability will approach one as the number of samples increases. It is perhaps surprising that with a modest ten samples and a single sampling event our probability of detecting the malicious bytes is already greater than one in five.

Table 1: Probability of Detection vs. Number of Samples for 100 Bytes in a Page

Number of Samples	Probability of Detection
10	21.9%
50	70.9%
100	91.6%

2.1.2 Sampling Over Time

Suppose the probability of detecting tampering during a single sampling event is p. Then we can treat this as the "probability of failure" just as we did in the prior section and consider the probability of detection over many samples.

In this case we are not as concerned with the probability of detection as we are the *mean time to detection* (MTTD). This is the number of sampling events until the first detection occurs and has expectation $1/p$ with variance $(1-p)/p^2$. Suppose we are again trying to detect just 100 modified bytes in a single page, and that we restrict ourselves to sampling just ten bytes from each page. The probability of detection on a single sampling event is 21.9%, giving a MTTD of 4.57 events with a variance of 16.3 events. Using the rule of thumb that 99.7% of a normally distributed outcome can be found within three standard deviations of the mean, we conclude that roughly 53.5 events will give us a cumulative 99.7% confidence of detection.

We can reverse this to obtain the number of trials until we detect with a given probability. Let the probability of detection (for a single sampling event) be d. Then the probability that we sample s times without a detection is $(1-d)^s$. The probability that we *do* detect at least once during that period is thus $P = 1-(1-d)^s$. Solving this for s gives the following expression.

$$s = \frac{\ln(1-P)}{\ln(1-d)}$$

Suppose we wish to detect with at least 90% probability. Assume that we are again trying to detect just 100 bytes in a page, and that we restrict ourselves to sampling just 10 times during an event. This gives a single-event detection probability of 21.9%. Plugging in $P = 0.90$ and $d = 0.219$ we obtain $s = 9.32$ events. That is, even if we just sample 10 bytes per page and there are only 100 modified bytes, we would still have a 90.0% chance of detection by the 10th sampling event. Sampling once every five minutes means this will take 50 minutes.

The point of this isn't that these computations are useful in practice, but that even with *small sample sizes* and *relatively infrequent sampling*, we still have a high probability of detection of tampering. As the number of samples grows over time, the probability of detection approaches one.

2.1.3 Alternative Sampling Strategies

Our implementation of Verify relied on sampling with replacement over *all* pages marked as executable. Unfortunately, in testing by our partner (GE) on their devices, this strategy did not complete within the desired 1/60th of a second. Two ways of reducing the sampling time are explored here. The Verify project took place during the COVID-19 epidemic, and staff were not able to meet in the laboratory to conduct this testing, so both are proposed strategies that have not been tested, but that show excellent promise in improving the time to sample.

Multibyte Sampling

The initial, naïve implementation of Verify sampled *bytes*. The entire page is available in-cache during the sampling event, and a more efficient approach would be to use the pointer size of the platform. For a 32-bit processor that would be four bytes, while for a 64-bit processor that would be eight bytes. An operation that uses several bytes equal to the platform memory bandwidth is potentially *faster* than one that uses single bytes.

A potentially surprising result is that the number of trials to detect a specific number of bytes in a page *does not change* with the multibyte sampling *for the worst case*. Suppose we sample four contiguous bytes each time (a "double word" in X86 parlance). Then a 4,096-byte page consists of 1,024 double words. Suppose our 100 altered bytes are clustered together in 25 double words (the worst case as they occupy the fewest number of double words). The ratios 4096:100 and 1024:25 are the same, and the probability after a number of trials is thus the same.

However, if the altered bytes are spread out, then the probability of detection may increase significantly. For instance, if the 100 altered bytes are spread among 100 double worlds (the best case) then the ratio improves to 1024:100 and detection probabilities are much higher. This improvement cannot be realized with byte-oriented sampling. Since neither the worst- or best-case scenario is likely in practice, we should expect an improvement in probability, and thus the need for fewer samples. An amortized cost computation for this approach was not done as part of this study.

More modern processors may allow the use of *vector processing extensions*. This requires investigation to determine whether additional savings can be realized, but clearly a larger number of bytes being sampled at a time can improve the best-case probability and the amortized probability of detection.

Random Page Selection

The initial, naïve implementation of Verify sampled from *every executable page* in memory at every sampling event. This means the time to complete a single sampling event depends on the number of pages present at runtime and is likely why the tool performed worse in testing at GE than it did at ORNL.

We can significantly reduce the time to perform the sampling by applying random selection to the pages themselves. While this reduces the probability of detection for a single sampling event, it allows us to make the sampling time much more *deterministic*. If we know the time to sample a single page (which can be determined empirically and adjusted as necessary) then we can compute the number of pages X we can sample within a given time window to guarantee that we meet the desired schedule. It was the intent of the Verify team to implement this for further testing with GE, but this proved impractical given the unforeseen project constraints.

While this would seem to dramatically reduce the probability of detection, we can instead use a strategy of partitioning all pages of memory into N blocks such that each block has the required size X. We then choose a single block of the partition for sampling. We can then choose a permutation of the blocks and iterate over the blocks in that order during sampling, guaranteeing that every block has been sampled after N sampling events. This minimizes the impact on the probability of detection. To increase flexibility we can provide, in addition to a random seed, the partition strategy and block number for each sampling request, so this is not known to the adversary prior to sampling.

2.2 TIMING-BASED ATTESTATION

Given sufficient time, an adversary can compute and return the correct result for each sampling event, preventing detection. Avoiding this possibility requires that each result produced and returned by the device to the server be *attested* in some manner so that its authenticity can be verified.

The Verify project early on made the decision *not* to rely on specialized hardware (such as trusted platform modules (TPMs), encrypted storage, or solutions like the Intel "Trust Zone") that may not be available, especially on legacy devices. A survey of attestation methods was performed [4], and the decision was made for Verify to use *timing-based* attestation.

Several favorable properties of the Verify approach make timing-based attestation suitable. First, the sampling process runs in the kernel, is very simple, and runs *as fast as possible*. An adversary would need to intercept the encrypted sampling request, obtain the random seed value, perform the same sampling on a clean memory image, and then return the encrypted result. (Since encryption is asymmetric, an adversary that has compromised a client only has access to a single key pair *for that client*.) This effort must be completed in much the same time. The adversary has two possibilities.

1. Perform the work using a stored (unmodified) copy of the golden image on the device.
2. Perform the work externally on a significantly faster machine and transmit the result.

Performing the work locally is complicated in two ways. First, the golden image must be stored. Storing it persistently uses space in the file system and creates additional possibility for detection, *while also slowing sampling*. Sampling from a stored image on disk will necessarily be slower than sampling directly from memory via the kernel. Alternately, storing the image in memory requires twice the memory of a non-compromised device and may create memory pressure at runtime. Though our proof-of-concept did not implement it, because it runs in the kernel additional runtime memory checking (such as a simple gauge of memory use) can be done and returned to the server to detect this approach. In either case the attack must do significant work on the client device without being detected during sampling or by other conventional means.

If the work is done externally and transmitted, this adds network overhead to the reply. Because the sampling is normally done over the utility's own network, this additional latency may be high enough for detection, and is also exposes the intrusion to detection using normal network monitoring as the work must be done *for every sampling event for every compromised device*, generating a lot of extra traffic.

3. PROJECT ACTIVITIES

The Verify proof-of-concept was implemented as a simple server and a client-based Linux kernel module resulting in an end-to-end demonstration of our approach for a single device: a Compact RIO running NI Linux. The server display for a device is shown in Figure 3.

Several real-time operating systems were investigated, including NI Linux Real-Time, Riot OS, Contiki OS, and VxWorks. Staff were trained on VxWorks, and initially a sbRIO-9636 running VxWorks was used. A shared library was compiled for VxWorks using the GNU toolchain and tested using the VxWorks kernel shell. Ultimately, after discussions with industry partners, the team decided to focus on real-time Linux distributions, instead, as this was regarded as a better fit for the Verify approach.

The Rekall memory forensic tool was installed on a cRIO-9039 single board controller and on a virtual machine running ContikiOS. Rekall was used to collect full volatile memory images in AFF4 format as part of our initial investigations.

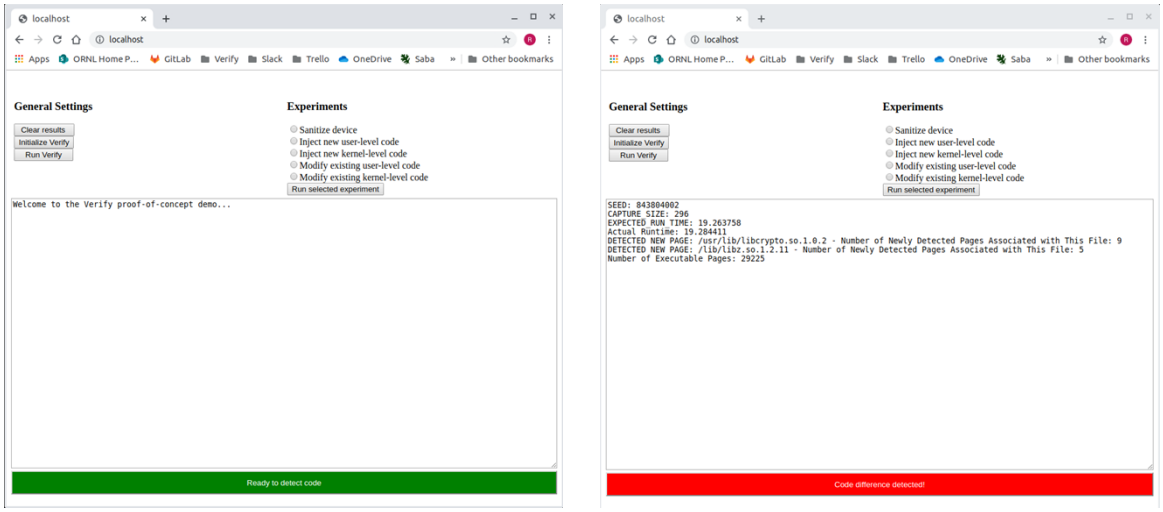

Figure 3: Successful Verification (Left) and Tamper Detection (Right)

Based on these preliminary investigations, a kernel module was developed that identified the outermost page table for each process on the system (the Page Global Directory) as well as the kernel space itself (Page Map Level 4). The kernel module walked the levels of these page tables to acquire each individual page assigned on the system as well as its execution rights. Process pages were tagged with a unique identifier and their bytes dumped in hexadecimal format for comparison. This kernel module was compiled and tested on the cRIO-9039, cRIO-9056, and sbRIO-9037 with various configurations.

Investigation identified the storage location for firmware was on an NI Linux Real-Time installation. An NI Linux firmware configuration file was downloaded and analyzed, determining the compression method and contents, allowing investigation of the generation of "golden" image data from that file.

Through EPRI we identified a supplier and a device for end-to-end testing and began work with the supplier. The team required kernel-level access to the device configuration but was unable to reach an agreement with the supplier. Instead, we decided to target the GE G500 Remote Terminal Unit (RTU) running Linux.

Different hashing strategies were investigated, with the team settling on SHA1 for the proof of concept. SHA1 was chosen over stronger hashes because it can be computed quickly on the device, but even so an adversary operating on the device will not have sufficient time to break and "fake" the hash. Any final solution should optimize the hash used based on the hardware available on the device.

The kernel module was then extended to handle huge page middle directory (PMD) and page directory pointer table (PDPT) entries during the page table walk, resulting in a much larger number of kernel pages to sample.

Initially the golden image was collected from memory after the device reached a stable state as part of Verify's device initialization phase. The team then began work on golden image generation *without* copying over all the current pages. Work was done on analyzing file relocations to record offsets relative to the virtual addresses of pages at runtime. This should allow Verify to correctly initialize by scanning ELF files and to correctly handle address space layout randomization (ASLR). Subsequently, at verification time, for each page you would open the corresponding ELF binary on a stored "golden" disk image and recompute relocations to obtain the appropriate image. The client would send the virtual address for at least one page of each binary in memory, allowing the server to compute all other relocations. Considerable code was developed to implement this relocation handling, but it was not completed.

While improvements in the sampling approach may lead to overall improved performance, additional strategies for improving the performance of the kernel module itself were explored. The current module uses the seq_file interface. Alternative approaches to this should be considered. Likewise, the page discovery algorithm can likely be improved. Right now, for each page table entry that is *not* marked as executable in the kernel page table the system walks all process page tables to resolve the entry to an executable page. This computation is costly due to the number of processes. A method to identify these pages faster by either a full process walk with caching at the beginning or avoiding the lookups altogether by identifying static parts of the page table has the potential to dramatically increase the performance of the kernel module.

4. EXTERNAL EVALUATION

Following successful end-to-end demonstration of the system at ORNL, the software was sent to GE Research for external evaluation. While those full findings are part of a separate confidential report, the following summarizes and quotes from those findings only as it relates to Verify's performance.

GE successfully integrated Verify with their G500 RTU and made necessary modifications to allow the remote attestation server to communicate with the client. Performance metrics were collected from the G500 during Verify execution, including the following.

- CPU utilization: the amount of time that the CPU spends executing the kernel module as a percentage of total available CPU execution time.
- Cache misses: the number of times the CPU must fetch data from RAM rather than the cache.
- CPU instructions per cycle: number of CPU instructions executed per clock cycle. We expect this metric to decrease if the kernel module execution is bound by memory fetching.
- Memory utilization: amount of memory (physical and virtual) used by kernel module throughout initialization and verification execution.
- Memory latency: delay between CPU fetching data from RAM and receiving the data.
- Execution time: time needed to execute initialization and verification.

GE confirmed correct functioning of the G500 during kernel module execution by observing that the G500 was still performing its normal operations, consuming DNP3 and R-GOOSE messages with metered data and displaying this data along with internal sensor values to the human machine interface (HMI). No impact on the normal operation of the G500 was observed, due to the kernel module's relatively low memory usage and execution on a single core of the device's four core CPU.

GE tested four attack scenarios provided by ORNL. While these were correctly detected by the Verify server, it also reported "runtime outside predefined bounds," as the page sampling simply took too long to complete.

In their findings, GE also proposed changes to Verify to improve performance, including replacing SSH with TLS and providing a detailed proposal for implementing that. More significantly, GE showed that "the performance bottleneck for the kernel module is the functions associated with the conversion of memory page virtual addresses to physical addresses." They recommended using a single-instruction multiple-data (SIMD) architecture to break virtual address conversion into multiple parallel instructions and noted that the G500's processor supports AVX instructions for this purpose.

5. FURTHER WORK

The Verify approach shows considerable promise for detecting in-memory threats and catching fileless malware without disrupting the normal operation of critical infrastructure devices. While the 1/60th second time constraint was not reached during this effort, suggestions from GE Research in the External Evaluation section and improved sampling methods discussed in the Approach section should allow attaining the necessary performance. The primary bottleneck is the page table walk. The team was beginning to investigate this with the GE personnel when the project ended.

6. REFERENCES

[1] S. Mansfield-Devine, "Fileless attacks: compromising targets without malware," *Netw. Secur.*, vol. 2017, no. 4, pp. 7–11, Apr. 2017, doi: 10.1016/S1353-4858(17)30037-5.

[2] J. Smelcer, "Rise of Fileless Malware," M.S., Utica College, United States -- New York, 2017. Accessed: Sep. 23, 2020. [Online]. Available: https://search.proquest.com/docview/1990643022/abstract/253F6C2D036A4295PQ/1

[3] Sudhakar and S. Kumar, "An emerging threat Fileless malware: a survey and research challenges," *Cybersecurity*, vol. 3, no. 1, p. 1, Jan. 2020, doi: 10.1186/s42400-019-0043-x.

[4] W. A. Johnson, S. Ghafoor, and S. Prowell, "A Taxonomy and Review of Remote Attestation Schemes in Embedded Systems," *IEEE Access*, vol. 9, pp. 142390–142410, 2021, doi: 10.1109/ACCESS.2021.3119220.