Term Paper One

Hamza Mughal
Department of Computer Science
George Mason University
Fairfax, Virginia
hmughal2@gmu.edu

***Abstract*—Four papers have been selected to form the basis of this term paper. The selected papers deal with two areas of Autonomic Computing: self-healing and self-protecting. This term paper covers the following topic for each selected paper: general issue addressed, summary, and critique.**

Keywords—self-healing, self-protecting, term paper

I. Introduction

Autonomic Computing involves computer systems that do not involve any sort of human interaction asides from defining any input to the system. Autonomous systems have a self-* property where the * can represent different properties of Autonomic Computing. Four areas of Autonomic Computing have been defined. The four areas are self-configuring, self-healing, self-protecting, and self-optimizing.

Four term papers have been selected of which two involve the self-healing property and the remaining two involve the self-protecting property. The two papers that deal with self-healing are ShieldFS: A Self-healing, Ransomware-aware Filesystem [1] and Antilizer: Run Time Self-Healing Security for Wireless Sensor Networks [2]. The two papers that deal with self-protection are VirtSC: Combining Virtualization Obfuscation with Self-Checksumming [3] and Tightly-coupled self-debugging software protection [4].

This term paper reviews each paper and answers the following topics: issue addressed within Autonomic Computing, summary of the paper, and critique of the paper.

II. Issue addressed within autonomic computing

A. ShieldFS

Ransomware encrypts data on a user's machine which can only be decrypted after the user pays money for decryption. Typically, ransomware payments are done in cryptocurrency because they are untraceable. The success of such attacks is entirely dependent on whether a victim pays to have their data decrypted. Families of recent ransomware attacks are now much more technologically advanced compared to their first-generation counterparts as these attacks now encrypt every file with unique keys which in turn reduces the chance of recovering files without paying a ransom fee.

Current security measures do not fully protect against ransomware attacks that have been ever so increasing year after year. Some approaches, such as a pure detection approach, are too late to efficiently protect against an attack. The paper proposes a self-healing add on driver that protects against ransomware by toggling on a protective layer from which a set of models are monitored. If a model is violated, then the running process is considered malicious and damage is mitigated.

B. Antilizer

Wireless security networks (WSNs) consist of many sensor nodes that are resource constrained. WSNs have been applied successfully to IoT applications by providing connectivity and information gathering abilities. Inclusion of WSNs to larger systems runs the risk of them being open to the public, and thus being hard to secure. Research efforts have been made to harden defense of WSNs however they do not allow a WSN to recover, prevent disruption, or recover data. A few security approaches address intrusion detection and autonomous intrusion prevention however it provides a single point of failure for an attacker.

This paper proposes a run time solution for WSNs called Antilizer that detect network attacks and subsequently adapts decisions to avoid effects of such attacks using a trust model.

C. VirtSC

Self-checksumming (SC) ensures that code in memory hashes to known values at runtime for tamper proofing. This requires computation of hashes which in turn requires hash values be adjusted in the binary executable. This causes SC to be limited with other protections. SC by itself can be easily disabled which is why obfuscation is necessary as well. This paper proposes layered protection using virtual obfuscation that produces SC protection that requires no adjustment.

D. Self-Checksumming and Self-Debugging Scheme

Currently existing self-debuggers are extremely simple in that they implement an obfuscated inter process control flow transfer mechanism using predetermined tools along with simply binary code patching. This implies that it is straightforward for hackers to inject a debugger to reset every breakpoint to the original instruction causing iterative reconstruction/de-obfuscation of the original program. This paper proposes migrating whole chunks of code fragments to the self-debugger making it more difficult to re-engineer the original unprotected application.

III. Summaries

A. ShieldFS

Ransomware encrypts data on a user's machine which can only be decrypted after the user pays money for decryption. Typically ransomware payments are done in cryptocurrency because they are untraceable. The success of such attacks are entirely dependent on whether or not a victim pays to have their data decrypted. 50 percent of ransomware victims succumb to paying such fees according to a November 2015 survey. In the first quarter of 2016 in the USA, more than 209 million dollars' worth of ransomware payments were done.

Kharraz et al. found that filesystems were a focal point for monitoring of ransomware. This paper takes the work done by Kharraz et al. and expands on it by developing a filesystem that prevents ransomware attacks from occurring. Understanding how ransomware compares to benign software, at the filesystem level, is crucial to understand by looking at I/O request packets (IRP). IRPs are basic data units that originate from high level operations such as file reading, and file opening. IRPLogger, a tool developed by the authors, is used to collect IRPs from ransomware free machines. It was

installed on 11 machines whose users use the machines for typical day activities. This was done for a month and approximately 1.7 billion IRPs were collected by 2,245 unique applications.

A set of analysis machines were then populated with the data collected to resemble typical filesystems such as the ones in the 11 real world machines in order to create a realistic environment to trigger ransomware attacks. IRPLogger was used once again to monitor the filesystems on the analysis machines that were infected with ransomware.

The initial assessment on the analysis machines guided the approach in the design of a detection system that combines analysis of entropy of multiple read and write operations of files and folders. The approach creates models that are able to distinguish ransomware from benign processes at runtime. ShieldFS willingly adapts such models to the filesystem by looking for indications of cryptographic primitives. ShieldFS also scans memory of any potentially malicious process for any traces of block cipher key schedules. Modern ransomware typically may use code injection to have a process execute malicious code. ShieldFS detects code injection by considering the long and short term history of each process and of the system as a whole by considering the run time effects. ShieldFS is applied in a virtual filesystem that shadows write operations, so if a file gets modified by a malicious process the filesystem reverts to the mirrored copy dynamically depending on the output of the detection logic. An example of how ShieldFS operates when reverting to a mirror copy can be seen in figure one below.

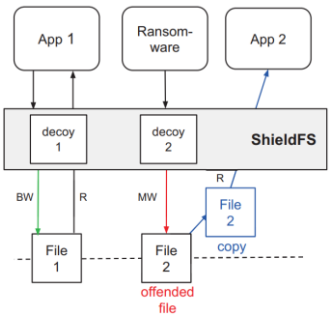

Fig. 1 [1]

Their approach is divided into two parts: ransomware activity detection and file recovery. Files are considered as decoys meaning that malware eventually will reveal itself because it cannot avoid access to files that it needs to encrypt. Some I/O level activities recorded by decoy files include files read and files written. ShieldFS is designed as a supervised classification task based off of the preliminary data analysis done. The classifier is trained on filesystem activity features. The list of activity features selected in available in table three in the original paper. After training, the classifier decides whether or not the extracted features at run time are malicious of not.

Multiple models have been adapted. One set of models is the process centric model which trains on each process. A second model called the system centric model trains on all IRP logs from a single large process. The system centric model may lead to more false positives which is why it is used solely in junction with the process centric model. Two approaches are adapted to frequently and efficiently monitor all processes. The first approach involves splitting data into ticks where each tick is the fraction of files accessed by the monitored process. This results on an incremental classifier which trains on larger data intervals. The second approach monitors long and short term history of a process by organizing the incremental classifiers into hierarchical structures where each observes larger spans of data. An example is shown in Figure two below.

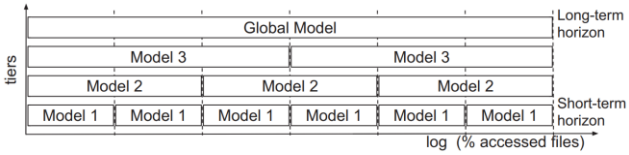

Fig. 2 [1]

Detecting traces of a cipher is an indication of malicious activity which is done through a per file random key encryption with a symmetric cipher. The most popular symmetric encryption algorithm is fast block cipher which combines plaintext with a secret key through a series of iterations. A side effect to the fast block cipher is that the key schedule is put into memory during the encryption process. Because of this, a scan can be done on memory of the process to determine the content of the memory however it is unlikely to result in a match due to fragmentation.

Newly created processes enter an unknown state in ShieldFS. If the process modifies a file, the file contents are copied to a read only storage. A process only moves out of the unknown state into either the benign, suspicious, or malicious state after enough IRP data is collected. Files that belong to benign processes are deleted immediately or deleted later through asynchronous deletion. Processes that are in the malicious state are suspended and files are restored to their original contents based off whether there is any presence of ciphers. If no ciphers are found, then ShieldFS waits until K positive ticks are reached before suspending the process regardless of whether any ciphers are found. Processes are solely in a suspicious state when an exact decision cannot be made. As a result, the system centric models get queried to determine whether the process is malicious or benign.

High level architecture details are provided in figure three and the detection loop algorithm is defined in algorithm one. ShieldFS is focused with Windows system in mind primarily because Windows systems are the primary targets of ransomware attacks. IRPs are intercepted by the filter manager APIs where each feature value is updated for each IRP. Normalization of features is done in two ways. The first mechanism uses a kernel thread to update normalization factors in real time. The second mechanism updates normalization factors periodically.

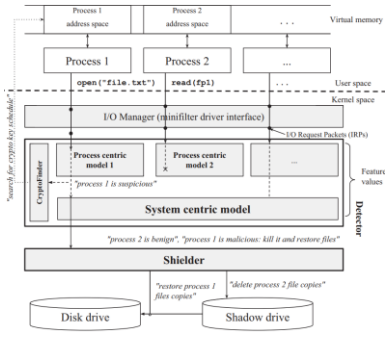

Fig. 3 [1]

Each classifier is a random forest of 100 trees where each classifier outputs -1 indicating benign or +1 indicating malicious. The sum of the random forest is used to classify a process, and in the event of a tie the process is classified as suspicious. 28 tiers of ticks are used where each corresponds to a distinct model tier. ShieldFS checks the memory of processes that have been classified as suspicious or malicious for the presence of cryptographic primitives. ShieldFS maintains a log of IRPs. Whenever a process is classified as malicious, the logs of the process are inspected and files are also restored. Files, such as temporary files, that have no value to ransomware attacks are whitelisted to reduce performance overhead.

Experimental testing is done on an environment with virtual machines so that file systems of potential machines can be mimicked. Cross validation is done to assess the capabilities of the classifier and the impact of parameter choice on the quality of the classifiers. Physical machines are also infected with three samples of ransomware. ShieldFS on those machines successfully detected and recovered compromised files. Lastly the overhead of ShieldFS is considered.

Three cross validation experiments were done to evaluate the quality of the detector on the ransomware samples and benign applications of the 11 user machines. 10 fold cross validation was done on 10 random train test splits which reveals that using the system centric model to break ties is good and that the incremental approach is good for early detection. Parameter setting reveals that setting K, the number of consecutive positive detection to make a decision about a process, to three maximizes the true positive rate (TPR) with few false positives (FPR) as seen in table five..

K	FPR	TPR	IRPs
1	0.208%	100%	35664
2	0.076%	100%	43822
3	**0.038%**	**100%**	**67394**
4	0.019%	99.74%	80782
5	0.019%	99.74%	104340
6	0.000%	99.74%	135324

Table 5 [1]

ShieldFS is evaluated on three production systems. The three machines were infected with three families of ransomware. ShieldFS was able to successfully detect and restore files. Dummy files were produced to mimic a real user machine where then ShieldFS was trained on the dataset of IRP logs to see whether ShieldFS could detect and recover. 305 unseen samples were used to assist with cross validation. ShieldFS prevented encryption of all cases and detected 298 of the unseen samples without false positives. The seven files that were false positives were found to not be using AES encryption.

The performance of system overhead is evaluated. A new version of IRPLogger was developed and distributed to six uses to determine the overhead. The estimated overhead was .26. Storage space requirements of ShieldFS was also tracked. When T (how often ShieldFS should create copies of files) is set to 3h in the worst case, ShieldFS requires an extra 14.73 GB of storage space. Setting T to a high value to is recommended as it does not significantly influence performance overhead.

ShieldFS is susceptible to targeted evasion. At each tick a decision is made. Ticks are not time based but rather based on the percentage f files accessed. It does not make sense for a ransomware attack to encrypt as few files as possible thus it may attempt to find a benign application that has accessed the number of files it wants to encrypt. If such a benign application can be found, the processes features will change as soon as encryption of the processes files start to occur. If the thresholds of the classifiers and the value of T is known, then a mimicry attack can be done by encrypting as few files to remain below the threshold until T hours. Setting T to large values can make the attack wait for longer, whereas setting T to a low value guarantees data freshness.

Ransomware that injects code into multiple benign processes can evade the detector if the feature values are known. This would allow encryption to be done slowly however this may not be an ideal approach for an attack. False negatives in ShieldFS may be caused by the use of Intel AES-N1 to do encryption off memory. ShieldFS only supports AES currently.

Because ShieldFS runs in a privileged kernel mode, it is non unloadable at run time however an admin process can try to prevent ShieldFS from booting at run time by medication of the Windows registry. A malware can compromise ShieldFS by filling the shadow drive however ShieldFS can monitor said drive and alert the user when it is about to fill up.

In conclusion ShieldFS makes modern systems more resilient to ransomware attacks by detecting them and reverting any negative effects by keeping a fresh back up of files modified

B. Antilizer

Wireless Security Networks (WSNs) consist of many sensor nodes that are resource constrained. Inclusion of WSNs to larger systems runs the risk of them being open to the public, and thus being hard to secure. WSNs rely on radio networks which are easily disrupt able through cyber-attacks. Research efforts have been made to harden defense of WSNs however they do not allow a WSN to recover, prevent disruption, or recover data.

Antilizer is a run time security solution for WSNs that detects network attacks and subsequently adapts decisions to avoid effects of such attacks. It uses a trust model at every node to evaluate behavior of neighbors. Neighbor communication info is self-collected which is hard to falsify thus allowing eollection of network metrics without using the content of communication. Network metrics are mapped to a trust value. If a malicious neighbor is found, the position of that node is sent to the base which in turn tells nodes on the way to the base that there is a malicious node thus constraining the damage of an attack in a neighborhood.

Multiple models have been developed which are used in Antilizer. The network model consists of multiple devices communicating in a multi hop fashion with S being the set of sensor nodes and R being the set of root and base stations. The network is modelled as a time varying weighted graph. The security model considers the base station as a trusted source with a secure mechanism of deploying updates to the network with it having the final decision whether to initiate a request for revoking malicious nodes. The Threat model assumes the OSI network, as applied, attacks are specific to the network

layer. The network layer is responsible for providing routing. Attacks here can delay the flow of data to the base station.

At the threat model, multiple avenue of attacks are possible to disrupt data. Attackers can falsify information to other nodes thus affecting their routing decision, fail to transit to neighbors thus degrading data reception, and inject false information into the network wasting energy due to increased number of messages which reduces the ability of the network to carry data to the base station.

Every node in Antilizer collects information to generate the trust model of nearby neighbors. Whenever a node detects a malicious intent from a neighbor, its trust of the neighbor is changed, and communication decisions change based off the new trust value. The base station is notified when a malicious node gets detected with the ID of the malicious node. The base station also authenticates the network security information. All these steps are categorized into tasks into five different modules.

Malicious behavior gets detected when there are large changes in the collected information of a neighbor which results in a lowered trust value for that neighbor. Neighbors with lower trust values have data routed around it. Trust models are built in a distributed fashion with each node has a unique view. Nodes inform the base station of malicious nodes. The node that detects a malicious neighbor changes its parent node to one that has a higher trust value, and then an Antilizer Notification Ticket (ANT) gets sent to the base station of the ID of the malicious node. Neighbors of the new parent also get informed. The ANT that is sent to the base station determines what action to take for a malicious node Authenticity of information sent to the base station is verified through a filtering mechanism. Actions to a malicious node may be to revoke or reboot the node.

Because default WSN nodes communicate with other nodes that may be malicious, a trust factor is introduced to combat this. Information is collected regarding neighbors within a one hop distance as seen in figure two. Information is collected through listening in on radio packet transmissions. A limitation is that B cannot hear from node G however this is still enough to build trust values. A node only trusts itself and any information it can overhear. The information collected is stored as a set of metrics. Information collected is packets transmitted to a neighbor, ratio of packets received/sent, and rank of the node.

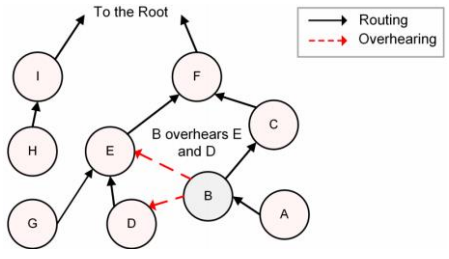

Fig. 2 [2]

Expected Similarity Estimation method (EXPoSE) is used so that no particular distribution is used. Network metrics are put into a vector for each time slot where the vectors are mapped into a Hilbert space. Incremental updates are applied to the inner product using a KEA vector. The trust metric is described in algorithm one.

Antilizer can be applied to any algorithm that uses distance measured routing. Routing Protocol for Lower-Power and Lossy Networks is used in the paper to evaluate Antilizer because it uses an objective function where trust values can be input as an added metric. Because Antilizer only affects the objective function, it can apply to a multitude of routing algorithms. The objective function is defined as how a node translate network metrics into a rank value where rank is used to determine the best neighbor to forward value to the base station.

ANTs work as follows in Antilizer: ANTs are used to address the issue that disruption of the network due to reconfiguration of the malicious node. ANTs are created whenever a malicious node is detected and ANTs do the following: ID of the malicious node is sent to the new parent which is then sent to the base station notifying all a malicious neighbor,

The base station then decides any appropriate action. Overhead of broadcasting due to ANTs is very low and is at most 2z + 1 additional messages. Because ANTs are only made when a malicious node is detected, there is no chance of no attack scenarios. When a new parent is selected due to a malicious node, it may be possible that the new parent is flagged as malicious due to increasing values which can cascade. This is counteracted through a grace period that allows the network to adapt to the new network behavior.

Antilizer has some limitations that must be noted. Attackers can flood the network with ANTs by falsifying ANTs. The base station will identify this as malicious behavior as ANTs are arriving at a high frequency. Flooding of ANTs also prevents routing away from a malicious neighbor. The attack model of Antilizer does not assume the modification of packet contents, thus encryption of ANT messages is needed however this is not enough as an attacker can retrieve the packets and replay them. When ANTs are sent to the base station, they can potentially be lost thus allowing a potentially affected neighbor to never be routed through again.

Antilizer is developed in Contiki (an open source OS for WSNs and IoTs). It provides an IPv6 stack and RPL routing. A CSMA/CA driver is used for the MAC layer with a neighbor table size of 50. Each compromised node initiates one or more of following activities: sinkhole attack, blackhole, hello flood. The RPL routing algorithm Is used with overhearing implemented with the extension of the Contiki network link stats module. This data gets stored in a neighbor table. Expected similarity values get calculated every 20 seconds for every node and are used as a weight to calculate the nodes rank. ANTs are implemented using RPL ICMP6 and RPL broadcast.

Evaluation of Antilizer is done through Contiki using Cooja. Networks with 25, 50 and 100 nodes are randomly distributed over 100mx100m, 200mx200m, and 400mx400m areas. Each node has a range of 50m and sends out data every 4 seconds. The number of Monte Carlo samples is set to 200 with a standard deviation of .35. Two attack scenarios are considered: single attacker, and multi attacker. Simulations were ran for over four hours with attacks starting at 20 minutes for 25 node simulations or at 40 minutes for 50 and 100 node simulations.

The performance metric of Antilizer is evaluated off the following: End to End Data Loss, Average End to End delay, Overhead, and Detection Reliability. Antilizer is first

evaluated with a no attack scenario. There is no performance penalty in this case with no attacks. When ran with a sinkhole attack, Antilizer detects changes in rank and subsequently reroutes node communications. There is no impact on End to End data loss however data sent is being delayed. Overhead is also very low.

For blackhole attacks, Antilizer notices drops in transmissions and reroutes data. Data loss is down to 1% and transmission delays are kept to normal latency. Overhead is less than 0.5%. For Hello Flood attacks, all nodes detect and send ANTs to the base station. The average time to send ANTs is up to 40 seconds. Antilizer is able to identify about 98.6% of anomalies with 3.6% false positive rate.

In conclusion, Antilizer enables WSNs to detect and recover from anomalous behavior using a trust model to detect any malicious nodes.

C. VirtSC

Man-at-the-end attacks threaten software execution and intellectual property of entities. Dynamic tampering of software modifies runtime memory. Static tampering of software modifies bytes of the executable. An example of static tampering is bypassing license key validation which can lead to pirated software. This encourages developers to introduce code tampering to reduce illegal distribution.

Self checksumming allows programs to detect any change in the binary using guards. Upon detection of a change, responses may include aborting or self-repair. Self checksuming post processing is a tedious error prone process that requires knowledge of the underlying system which also limits use of other obfuscation. To reduce the need for post processing, LLVM is used to implement self checksumming on top of virtualization.

Virtualization Obfuscations (VO) goal is to transform the control flow of a program to its semantic equivalent. This transforms all instructions into a new random Instruction Set Architecture (ISA) equivalent where the interpreter is created specific to the ISA. The interpreter fetches, decodes, and dispatches execution to the original instructions handler which emulates original behavior. Program memory allocation is done via virtual memory and a virtual program counter keeps track of the last executed instruction. VO can prevent automated attacks from occurring

Self checksumming (SC) uses guards to calculate hashes over code segments to detect code manipulations. SC requires extensive knowledge of the underlying architecture which imposes additional cost in development life cycle and maintenance of such protections. SF guards are prone to pattern matching which can make it easily identifiable and defeatable. Using obfuscation with SC can break SC guards thus SC with other protections is limited.

Protection schemes have been researched by previous works. This paper focuses on the composability of protections and the removal of any architectural dependencies.

The high level architecture, shown in figure one, of VirtSC involves a random Instruction set Architecture (RISA). It is a volatile data structure that capture random instructions. The Virtual Program Array (VPA) contains a translation of the original instructions so whenever an instruction does not have a RISA equivalent it is added into RISA. Memory accesses now use virtual memory. SC guards are injected into the VPA which calculate hashes and do verification. The interpreter fetches instructions from the VPA and it is decoded into RISA semantics to be able to execute.

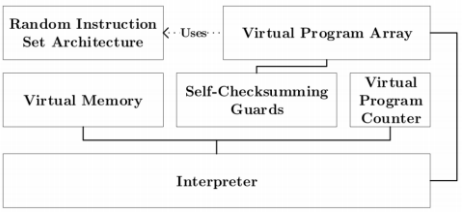

Fig. 1 [3]

The RISA is composed of Opcodes, operands, and semantics Opcodes capture mnemonics, operands refer to indexes in the VM. The VM holds all operands required. Values are stored contiguously. The VPA is a static array which contains lifted instructions which allows any function to calculate the hash of any other function. The interpreter fetches instructions, decodes it and dispatches execution to the handler.

Self checksumming guards protection utilizes guards to verify values of different virtual functions. It hashes a functions VPA to make sure it matches the expected values at different time intervals. A random network of checkers are generated in a directed acyclic graph which results in sensitive functions being checked by other non-sensitive functions. Response mechanism in VIRTSC results in process termination

The machine code output is likely to be extremely unoptimized such as in the instance of switch statements so the interpreter uses indirect threading to optimize handlers with each other. This optimization may impact the security of an application as it makes control flow linear and machine code easier to read. As a result, this optimization has been turned off by default however is toggleable.

Evaluation is done on a dataset consisting of a subset of 22 programs from MiBench along with three open source CLI games. A way to determine effectiveness is to measure protection coverage which refers to the number of instructions that are virtualized and the number of instructions that are guarded. 100% coverage indicates that all program instructions are sensitive and all are protected. Overhead is also measured for both secure and optimized modes. Benchexe is used to measure performance of programs before and after protections are set. Each program is ran 100 times with same inputs and overheads are measured.

For the secure mode, the average overhead with just using VO is lower than when using VO+SC except in the 25 protection level where VO+SC overhead was smaller due to the potential randomness of function selection. For the optimized mode, the VO+SC overheads were larger than VO as expected.

Security threats are analyzed from three perspectives: SC protection threats, VO treats, and combined protection threats. Attacks on SC are presented in figure four. Disable checkers is where an attacker disables checkers for a function. An attacker in this instance has to analyze and reverse engineer the interpreter of all checkers thus increasing complexity and time of the attack. Disable response mechanism disable the response function. If an attacker removes the function call that

checks hash mismatches, it will cause the program to run into a segmentation fault.

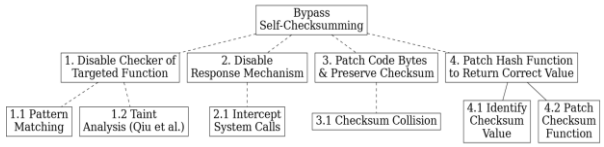

Fig. 4 [3]

Patch Code Bytes & Presum Checksum attacks are stopped by the VO due to changes needing to go through the interpreter. For Patch Hash function a attacker modifies the hash function to return the correct value for the current checker. The VO has multiple threats particularly the optimized approach due to the VPC being replaced with direct jumps to next blocks. Performance gets improved however control flow recovery is possible with static analysis.

Known attacks on the VO include generic attacks such as using taint analysis, symbolic execution and code simplification . This attack was not able to be tested due to a compilation error. Attacks on the interpreter are also possible without having to reverse engineer bytecode. It is possible to change jump addresses of blocks to different memory locations however this can be mitigated by reusing handlers .The interpreter can have more protection added by using other protection software. Because VirtSC has been developed in LLVM, it is possible to use in junction with other protection software.

Some instructions were left unprotected during evaluation most likely because they reside in the root of the network of checkers or they reside in nodes with no edges. 100% coverage can be obtained by not checking guards however this downgrades resilience. Cycles are also not supported so some nodes are unprotected.

There is a tradeoff between security and performance. The security implementation was on average 10x slower than the unprotected implementation. Further research will be done to implement security that does not downgrade performance. The optimized implementation resulted in more efficient binaries. This resulted in a 50% decrease of overhead protection.

Limitations of VirtSC include the following: Few Windows instructions that deal with exception handling are not support. This is due partly because Windows doesn't allow loadable modules. VirtSC is also leaking memory of the VM due to not having accesses to a technique called RAII in LLVM. Global variables cannot also be added

In conclusion, VirtSC enables virtual orfuscation with self checksumming that is independent of any architecture.

D. Self-Checksumming and Self-Debugging Scheme

Recent trends have been to give applications as few permissions as needed to prevent malware from taking advantage of any unneeded permissions. This includes restricting debugging capabilities which in turns prevents any malware from eavesdropping on the application memory and its I/O. It must be noted that these protections entirely depend on the run time environment. An application run time environment can be diverted to a targeted environment through a man-at-the-end (MATE) attack.

Assets such as cryptographic keys are important to prevent malware from accessing on a user device. Often because of the closed environment of user devices, attacks are focused more towards developer devices where attacks have the potential to control the complete running environment. To protect against MATE attacks and other attacks, this paper proposes building protections not into the environment rather the software itself.

Currently existing self debuggers are extremely simple in that they implement an obfuscated interprocess control flow transfer mechanism using predetermined tools along with simply binary code patching. This implies that it is straightforward to hackers to inject a debugger to reset every breakpoint to the original instruction causing iterative reconstruction/deobfuscation of the original program.

This paper proposes migrating whole chunks of code fragments to the self-debugger which in turn has many advantages. The advantages include making the I/O of the self-debugger self-contained. When the self-debugger gets called, what functionality gets executed is no longer pre-determined so automated analysis, deobfuscation, and decompilation can no longer occur. It also makes it harder for an attacker to remove protections/reconstruct the program if they happen to figure out control flow.

Figure one describes the proposed self-debugger at a high level. On the left side is an unprotected application with a control flow fragment. This fragments gets converted into two sections: a debuggee (in the middle) and a debugger (right side). The control flow fragment has been migrated into the debugger. Note that onlysingle entry multiple exit code ragments that don't have function calls are supported. At run time the debuggee gets launched then the initializer forks a new process for the debugger where the debuggers initializer attaches to the debuggee process. Red arrows indicate the control flow of when an exception occurred. When an application exits it performs detaching operations.

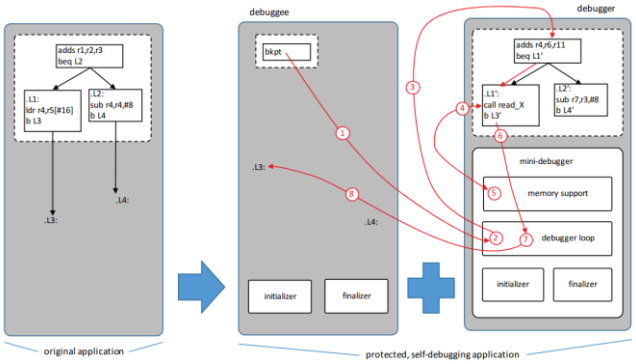

Fig. 1 [4]

Code fragments that are determined for code migration are annotated with pragmas, comments, and other annotations that denote the start and end of a fragment such as grep. Another option may be to list functions and files to protect or to selectively choose fragments semi automatically. Hot code fragments should not necessarily be favored over other fragments as it is necessary to choose fragments that throw exceptions frequently for there to be a relationship between the debuggee and debugger so this does not mean to necessarily pick the hot code fragments.

An important decision to choose what fragments to migrate determines in stronger or weaker protection. Selected

fragments should be before and inside code regions where attackers are interested.

Any compiler can be used to compile the program however one requirement is that the compiler and the assembler and linker must provide a link time rewriter with accurate symbol and relocation information to be able to reliably conserve link time code analysis and transformations that are required for implementing self-debugging. Debug information is also necessary to be able to find annotated source code.

The link time rewriter has two options when outputting a protected application. The first option produces two binaries where one is for the application/debuggee and the other for the debugger. This may be more secure however provides a higher overhead. The second option, which is used in the implementation, embeds debuggee code and debugger code into a single binary which allows forking to launch the debugger. Further research needs to be done to know if this provides any security.

An extra initialization routine is added to the protected binary which calls fork to create a parent and child process where either the parent or the child becomes the debugger or debuggee preferably with the parent becoming the debugger as it keeps the process ID the same. Multiple threads may exist when the application is started where one executes the initialization and the others perform debuggee processes. Exceptions are raised in the application through a software breakpoint using the BKPT instruction.

The thread that raises the exception is noted as the requesting thread where it asks the debugger to execute a code fragment. When the debugger gets notified, the debuggee passes an ID of the fragment. Multiple options exist for assigning a value to the ID. One option is the pass the address of the exception, another is to pass the ID in a fixed register before raising of the exception. The debugger loop fetches the program state of the debuggee before a code fragment gets executed, and for control flow transfer back to the original program.

Load and store operations require an access to the debuggees memory. Multiple options exist for this. One is to use ptrace. Another is to open the /proc/[pid]/mem file of the debuggee and then read and write into this file. Another option is to determine which memory pages will be accessed in the migrated code fragment and then copy the pages into the debugger address space. Another option is to provide a custom heap for the debuggee where all allocated memory is shared memory between the debuggee and debugger which allows data access. It is not known which option works better security wise.

The binary rewriter that implements the self-debugging scheme also implements additional protections, control flow obfuscation, code layout randomization, code mobility, code guards, control flow integrity, instruction set virtualization. A limitation of the current design and tool support is that it can only be deployed once in a running process so only the main application or a shared library can be protected.

The self-debugger implementation is strictly targeting ARMv7 platforms. Testing hardware consists of several developer boards. The self-debugging scheme is deployed on three use cases in three market leader companies. The first use case involves communicating with media server and DRM Android frameworks. This use case is good because the mediaserver is multi threader. The libraries are loaded/unloaded frequently. When the process crashes, it makes a new instance which results in the debugging to be more complex. Last the mediaserver and drmserver are involved in inter process communication.

The second use case involves a software license manager to store passwords. This library is loaded into Androids execution environment which has multi-threading, JIT compiler, garbage collection which there is no control over. The third use case involves a onetime password generator. All three use cases were tested extensively. The self-debugging scheme did not introduce much overhead for all three use cases.

For categories of attacks are possible against the self-debugging scheme. The first is circumvention and avoidance. This prevents the mini debugger from being invoked which requires finding a path between an entry point of the protected binary and the area they are interested in that has no control flow fragments. Here they can detach the mini debugger and attach their own debugger. It may be easier to find such areas in shared libraries due to their being multiple entry points.

A second attack is reverting transformations. All transformations done to migrated code are reverted which in turn disables the mini debugger. This can be done easily with static analysis by determining the address to where the control flow is transferred to. The third attack is attacking the mini debugger. The mini debugger acts as an attack surface. The child process is not protected so an attacker can attach a debugger to it and observe/manipulate the application. The fourth attack is full syste emulation although this may be harder than just debugging a single program.

For each use case, professional penetration testers were used to attack assets in the code protected. All tried tracing, and live debugging however none were able to successfully collect full traces or able to attach debuggers on code fragments. Some were able to succeed on code out of context without first executing a migrated fragment.

Some limitations of the scheme presented include: because the debugger process is single threaded and the debuggee process is multi threaded, a bottleneck may occur when handling all requests to execute migrated fragments. The self debugging scheme may not work on certain Linux distributions due to them choosing higher restriction levels for ptrace_scope which places restrictions on attaching to another process.

In conclusion, a tightly coupled self-debugging scheme has been proposed which makes it more difficult to reverse engineer the original program due to full code fragments being migrated.

IV. CRITIQUES

A. *ShieldFS*

This paper did a good job at explaining the problem formulation and details of the proposed scheme. Evaluation of ShieldFS was thorough and addressed multiple issues of concern such as overhead.

Some negative critique were found when reading the paper. Algorithm one was presented which is the detection routine for processes. This algorithm was not explained at all. High level abstractions may have been covered the algorithm

implicitly however the algorithm contains new notations that would be good to explain to the reader..

When it was discussed that they installed SheildFS on 11 production machines, it was stated that ShieldFS simply worked in protecting. However, there was no real evidence given for their blanket statement. It would have been better if they had explained why that was the case.

The conclusion did not necessarily feel like a conclusion of all the arguments and ideas presented in the paper. Instead they talked about how ShieldFS was good for short term files and then gave an argument for such a reasoning. This argument may have been better off in the limitation section.

B. Antilizer

This paper did an excellent job at explaining what the motivation of the proposed scheme was. Mathematical definitions provided were explained with enough detail to make sense of making sure to not go overboard. The algorithm, that computes the trust values, was self-documenting and did not need further explanation in any additional section.

Some negative critiques of the paper include the following: Evaluation of Antilizer was done on simulated systems that the authors had full control of. Despite the result being initially promising, Antilizer should have additionally been tested on a real world system. Perhaps the results they get in a real-world system may be drastically different than the simulated results.

Antilizer should have been compared to other state of the art techniques that are currently in place for WSNs to see how it performs against them. It is mentioned in the paper that current WSNs defense mechanisms do not allow the recovery of data, so it would have been a good idea to compare to current defense mechanisms.

The performance metrics, such as End to End Data, that were chosen were not explained as to why those specific metrics were picked over others. It would have been helpful to know why they choose these certain performance metrics over other metrics.

C. VirtSC

The design and implementation of VirtSC was explained in great depth which fully explained the inner workings. The full dataset that was being used in evaluation, and the subsequent data was provided which made it clear exactly how complicated or not complicated each dataset item was.

Some negative critiques of the paper include: In a few instances there were statements made that stated that they believed x was done because of y. An example is " We believe this is due to the randomness of function selections " [1]. These blanket statements may have been better off if they definitively knew what caused x to occur rather than them assuming because of y.

VirtSC should have been compared to other state of the art mechanisms that enforce self-checksumming and obfuscation to see how well it compared against them. Additional testing could have also been done using professional penetration testers to see exactly how well it faired off. It is mentioned that a combination of tools is needed to break the virtual obfuscation and check-summing. Using these tools to see if their scheme could be broken would have been a good test to see if what is proposed is indeed resilient.

D. Self-Checksumming and Self-Debugging Scheme

It was clear to understand how the whole proposed scheme operated through the inclusion of a figure which walked through the control flow of an application under the proposed scheme at a high level.

Negative critiques of this paper are the following:: The use cases developed for evaluation of the proposed scheme should have been explained more. Use cases one and two had an explanation as to why the use case was good for testing, however no such explanation was provided for use case three which was a little bit odd.

Professional penetration testers were used to test the implementation proposed. It would have been useful to additionally have the penetration testers test other state of the art programs and see how well they faired off against those versus the implementation proposed.

No quantitative data was provided at all. It would have been useful if such data were provided. Such data could have been gathered if comparisons to other programs were done.

CONCLUSION

This paper has reviewed four paper that deal with self-protection and self-healing. Each paper has had the following question answered: issue addressed, summary, and critique.

REFERENCES

[1] Andrea Continella, Alessandro Guagnelli, Giovanni Zingaro, Giulio De Pasquale, Alessandro Barenghi, Stefano Zanero, and Federico Maggi. 2016. ShieldFS: a self-healing, ransomware-aware filesystem. In Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC '16). Association for Computing Machinery, New York, NY, USA, 336–347. DOI:https://doi.org/10.1145/2991079.2991110

[2] Ivana Tomić, Po-Yu Chen, Michael J. Breza, and Julie A. McCann. 2018. Antilizer: Run Time Self-Healing Security for Wireless Sensor Networks. In Proceedings of the 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous '18). Association for Computing Machinery, New York, NY, USA, 107–116. DOI:https://doi.org/10.1145/3286978.3287029

[3] Mohsen Ahmadvand, Daniel Below, Sebastian Banescu, and Alexander Pretschner. 2019. VirtSC: Combining Virtualization Obfuscation with Self-Checksumming. In Proceedings of the 3rd ACM Workshop on Software Protection (SPRO'19). Association for Computing Machinery, New York, NY, USA, 53–63. DOI:https://doi.org/10.1145/3338503.3357723

[4] Bert Abrath, Bart Coppens, Stijn Volckaert, Joris Wijnant, and Bjorn De Sutter. 2016. Tightly-coupled self-debugging software protection. In Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering (SSPREW '16). Association for Computing Machinery, New York, NY, USA, Article 7, 1–10. DOI:https://doi.org/10.1145/3015135.30151