2015 IEEE 35th International Conference on Distributed Computing Systems

MAVR: Code Reuse Stealthy Attacks and Mitigation on Unmanned Aerial Vehicles

Javid Habibi *, Aditi Gupta*, Stephen Carlson†, Ajay Panicker*, and Elisa Bertino*
*Department of Computer Science
†Department of Electrical and Computer Engineering,
Purdue University, West Lafayette, Indiana 47907
Email: {jhabibi,aditi,carlson8,apanicke,bertino}@purdue.edu

***Abstract*—As embedded systems have increased in performance and reliability, their applications have expanded into new domains such as automated drone-based delivery mechanisms. Security of these drones, also referred to as unmanned aerial vehicles (UAVs), is crucial due to their use in many different domains. In this paper, we present a stealthy attack strategy that allows the attacker to change sensor values and modify the UAV navigation path. As the attack is stealthy, the system will continue to execute normally and thus the ground station or other monitoring entities and systems will not be able to detect that an attack is undergoing. With respect to defense, we propose a strategy that combines software and hardware techniques. At software level, we propose a fine grained randomization based approach that modifies the layout of the executable code and hinders code-reuse attack. To strengthen the security of our defense, we leverage a custom hardware platform designed and built by us. The platform isolates the code binary and randomized binary in such a way that the actual code being executed is never exposed for an attacker to analyze. We have implemented a prototype of this defense technique and present results to demonstrate the effectiveness and efficiency of this defense strategy.**

I. Introduction

As demand for automation has increased, embedded systems have evolved to satisfy this requirement. Consequently, improvements in processing speed, storage capability, and power consumption have increased the breadth of their applications. Examples of applications include but are not limited to modern cars, Internet of Things (IoT) devices and most notably unmanned aerial vehicle (UAV) control systems. Companies have recently unveiled plans to utilize these UAVs to automate different services that they provide to customers. These companies include Amazon's PrimeAir service, Dominoes drone pizza delivery, and DHL automated package delivery.

All the platforms that will be deployed will use an autopilot system to automate specific functionalities within the UAVs. Security of autopilot systems is thus crucial. Several attacks have already been reported on multiple embedded systems such as the ones used in cars [10], [39], [42] and we can expect similar attacks to be carried out against UAVs.

In this paper, we focus on return oriented programming (ROP) [31] attacks on UAVs. These are an advanced form of buffer overflow attacks that utilize existing code for malicious purposes. They can bypass traditional defenses against code injection attacks such as write or execute protection [30]. In a simple ROP attack, the attacker identifies small sequences of binary instructions, called gadgets, that end in a `ret` instruction. By placing a sequence of carefully crafted return addresses on the stack, the attacker can use these gadgets to perform arbitrary computation. However, there is a major drawback to these types of attacks. Although chaining together the gadgets allows attackers to execute an arbitrary set of operations, the attack itself destroys the stack frame. This often results in the vulnerable program to perform an illegal operation which prevents further execution. A halt in execution might be acceptable by attackers under most circumstances, provided that they gain access to their target. However in a UAV, a system crash usually results in defensive actions such as a secondary system taking over the primary failed system. Such a recovery may thus render the attack ineffective or make further attacks impossible.

In this work, we introduce a new form of ROP attack called "stealthy ROP attack." A stealthy attack first executes the attack payload completely and then reconstructs the 'smashed' stack frame before the final return. This allows the victim application to continue executing unlike in the case of a traditional ROP attack which destroys the stack frame. A stealthy attack is invaluable when the attacker is trying to gain control of a UAV without the ground station ever realizing that an attack has occurred. We also introduce a *trampoline* technique in this stealth attack that allows the attacker to inject arbitrarily large payload into the application's stack.

In addition to the stealthy attacks, we propose a defensive technique, MAVR, to mitigate code-reuse attacks on UAV systems. The mitigation technique aims at breaking critical factors that are required for code-reuse attacks. Our mitigation technique involves a new approach to code randomization which leverages specialized hardware design. The specialized hardware extends the autopilot platform with an external flash chip for storing the code binary and a secondary processor responsible for the code randomization. The randomized binary is executed by the primary autopilot processor. This hardware design is critical to ensure that the binary executed by the primary processor is completely different and isolated from the binary stored on the external flash chip. This prevents the attacker from identifying gadgets and their offsets. At software level, we utilize a high precision randomization technique which allows shuffling of the binary code at functional level granularity obfuscating the offsets so that the attacker is unable to identify gadgets and calculate necessary offsets to perform an ROP attack.

1063-6927/15 $31.00 © 2015 IEEE
DOI 10.1109/ICDCS.2015.71

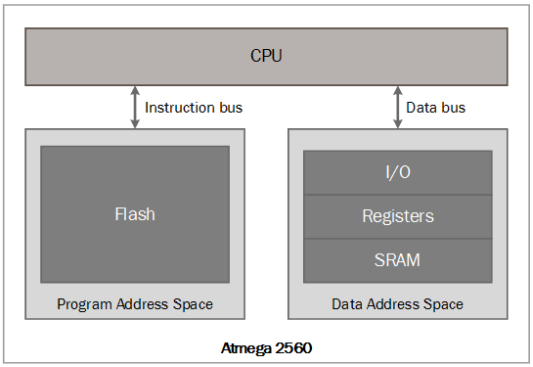

Fig. 1. Memory for Atmega 2560 micro controller

II. Background

In this section, we start with a brief summary of the platform that we are targeting for attack and mitigation.

A. The AVR architecture

The platform targeted in this attack is the Ardupilot Mega 2.5 (APM) [1]. This is a popular example of an open source UAV control system. The APM 2.5 is based on an Atmel AVR ATmega2560 8-bit microcontroller clocked at a frequency of 16 MHz. Additionally, this board also contains a 3-axis gyroscope, accelerometer, and magnetometer, along with a high-performance barometer for telemetry. The Atmel Atmega 2560 is a Harvard Architecture microcontroller. In this architecture, a microcontroller has physically separate signal and storage pathways for instructions and data. The control unit can only load instructions from the instruction memory and write values into the data memory. Thus, the data memory is not executable. At its core, the Harvard architecture prevents remote modifications of the program space, as modifications require physical access to the APM board.

B. Memory in AVR

The ATmega2560 microcontroller [2] used on the ArduPilot Mega board has three internal memories and no external memory (see Figure 1).

- The internal 256 KB flash is used to store program instructions. All AVR instructions have a size which is a multiple of two bytes, so the memory is addressed as two-byte words; 128 Kwords of instructions can be stored in this internal flash memory. This memory is the only area from which programs can be executed, as the Harvard architecture does not allow placement of the program counter in data memory.
- AVR processors have a single linear data space, containing memory mapped I/O registers and physical SRAM. All addresses in this range are accessible to the program, but no code can be executed from this region. Typically, the stack, global variables, and heap all reside in the SRAM address space.
- A small EEPROM memory is also included for persistent storage of configuration settings. This memory is not included in the data or program address space and cannot be used for random access or program code.

State magic number (1 byte)
Length (1 byte)
ID of message sender (1 byte)
Packet Sequence # (1 byte)
ID of message sender component (1 byte)
ID of message in payload (1 byte)
Message (<255 bytes)
Checksum (2 bytes)

Fig. 2. Mavlink packet structure

C. MAVLink Protocol

MAVLink or Micro Air Vehicle Link [27] is a protocol for communication between a small UAV and a ground station. At its core, a MAVLink message is a stream of bytes that has been encoded by the ground station and is sent to the APM via USB serial or Telemetry. The structure of a MAVLink packet is comprised of a 6 byte header, payload of up to 255 bytes but a minimum size of 9 bytes and a two byte checksum for integrity error detection (see Figure 2). This adds up for a minimum packet length of 17 bytes.

The APM receives the stream of bytes via one of its hardware interfaces and decodes the message in software. This supports seamless communication with the ground station.

III. Challenges in Exploiting and Securing UAVs

Traditional buffer overflow attacks usually rely on the fact that the attacker is able to inject a piece of code into the stack and execute it. In the APM platform which is based on Harvard architecture, the code and data memories are physically separated. Additionally, the program counter cannot point to an address in data memory. Therefore it is not possible to perform traditional code injection attacks on the APM.

Furthermore, the APM has other characteristics that limit the capabilities of an attacker. Since the APM controls the flight characteristics of the UAV, it has very strict deadlines that must be met. These are not software defined deadlines, instead they are physical deadlines, such as updating the control surfaces of the UAV to maintain its current course. This is due to the fact that the executing software must maintain control of the UAV. This constraint requires the attack to perform a "clean return", that is, once the stack is overwritten with the attack payload, the attack should not leave the stack in a corrupted state. In order to be stealthy and avoid detection, the attack must reconstruct the stack frame before returning such that the target program is able to continue executing. Due to all of these constraints, remote exploitation is very challenging.

In addition to these unique constraints for attacking an embedded system such as the APM, there are challenges in securing this platform as well. In most x86 based defenses it is acceptable to incur some overhead for the defense implementation. This however is not the case with the APM board where strict deadlines must be met. As with all embedded systems there are very limited resources available for defenses. These limited resources, such as CPU cycles, memory and code size, were factors that had to be considered in the design of MAVR.

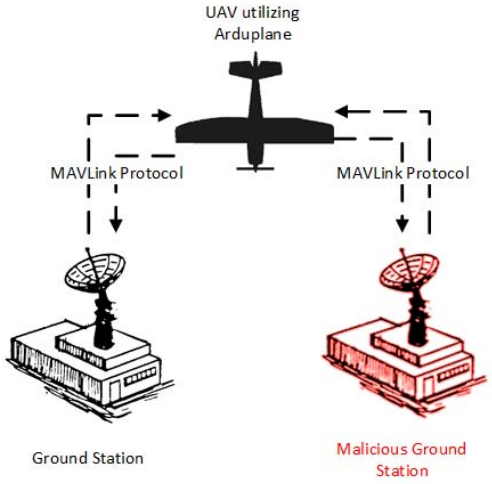

Fig. 3. Attack Vector

Due to the limited cycles available, the modifications done to the target binary cannot require too much computation. The reason is that an APM board running Arduplane 2.7 is already at about 96% CPU usage. Thus any defense implementation cannot have a large performance impact due to the numerous interrupts with strict timetables that must be met in order for the control unit to maintain the UAV flight characteristics. In addition the APM board has a limited code size. As stated in section II-B, the Atmega 2560 has only 256KB available to store program instructions.

IV. Stealthy Attack Details

In this section we introduce the stealthy attack, assumptions behind it, and the gadgets required to perform the attack.

A. Attack Assumptions

Throughout this paper, we make the following assumptions:

- UAV under attack uses an APM 2.5 as its control unit.
- The running code has at least one exploitable buffer overflow vulnerability.
- The attacker has access to the application binary that is uploaded on the board.
- The attacker has access a malicious ground station or has compromised a legitimate ground station.

B. Attack Setup

To demonstrate the strength of these attacks, we injected an artificial vulnerability that would allow a buffer overflow. This was a traditional buffer overflow vulnerability that is used by various attacks. Our intention in this work is to show the extent of damage that can be caused if a vulnerable buffer is present in the code which is not uncommon in huge codebases such as those of UAV systems. Once an attacker has compromised or attained a malicious ground station through any number of scenarios they would then have the ability to communicate with the UAV (See figure 3). This would allow them to take advantage of the specific vulnerability that we injected. In order to create a buffer overflow vulnerability within the Arduplane software we disabled the length check within the MAVLink buffer. This allowed us to send packets of any arbitrary length within our experiments as discussed in the following sections.

C. ROP Attack V1

In a traditional ROP attack, the attacker calculates the necessary address offsets for gadgets in the victim binary and chains them together to execute attack logic. Our basic ROP attack (version 1) for Ardupilot Mega follows a similar approach. In this basic attack, we construct an attack payload that will modify the value of the gyroscope sensor on the APM.

Our attack used a single gadget, called `write_mem_gadget`, that allowed us to copy arbitrary data on to a specified location in memory, for example location where sensor data was stored. Normally, pushing values into a register and writing them to the stack requires two gadgets. However, we found a combination gadget that stores the values of `r5`, `r6`, and `r7` into locations `Y+1`, `Y+2`, `Y+3`, where `Y` is a two byte register which consists of `r28` for the lower byte and `r29` for the upper byte. The second half of this combination gadget then pops registers `r5`, `r6`, `r7`. This means that we use the second half of the program section as our first gadget, and then we use the first half of the gadget to store the values onto the stack. We then used this to set the gyroscope sensor value which is currently recorded in the data address space. This is possible due to the fact that in the AVR architecture all registers are memory mapped. While setting the gyroscope sensor value allowed us to easily see the effect of the attack we would expect attackers to instead target the gyroscope configuration registers stored in memory that would have a continuous effect on the system.

To execute the attack we then filled the buffer with garbage and inserted our attack string utilizing the `write_mem_gadget` gadget to change the sensor value to anything we wanted. However, there were two major drawbacks to this attack strategy. The first is that the stack frames around our attack payload were completely destroyed. Second, because of the destroyed stack frames the board continued executing random garbage that would be easily detectable from the ground station. In the following section we propose a solution to these problems.

D. ROP Attack V2 – Stealthy Attack with Small Payload

Although a buffer overflow makes an ROP attack possible, the problem with this type of attack is that it smashes the stack which causes the system to crash.

In order to prevent this, we show that a clean return is possible. A clean return is a way to allow the system to resume normal operation after the ROP attack has completed. The reason why an ROP attack causes a system crash is the damage done to the run time stack. In order to get an ROP attack to return cleanly, we have to both minimize and repair the damage to the stack. To minimize the damage, we utilize two gadgets. The first gadget, called `stk_move`, moves the stack pointer to any address we specify (see Figure 4). The second gadget, called `write_mem_gadget`, copies arbitrary data to a specified location (see Figure 5).

More specifically, we first utilize the `stk_move` gadget to move the stack pointer, `SP`, to the beginning of the buffer,

Gadget 1: stk_move		
Instr. Address	Instr	Comments
5d64	out 0x3e, r29	New stack high byte
5d66	out 0x3f, r0	
5d68	out 0x3d, r28	New stack low byte
5d6a	pop r28	
5d6c	pop r29	
5d6e	pop r16	
5d70	ret	

Fig. 4. stk_move gadget

Gadget 2: write_mem_gadget		
1b284	std Y+1, r5 ; 0x01	Store byte to Y+1
1b286	std Y+2, r6 ; 0x02	Store byte to Y+2
1b288	std Y+3, r7 ; 0x03	Store byte to Y+3
1b28a	pop r29	stack address repair high byte
1b28c	pop r28	stack address repair low byte
1b28e	pop r17	
1b290	pop r16	
1b292	pop r15	
1b294	pop r14	
1b296	pop r13	
1b298	pop r12	
1b29a	pop r11	
1b29c	pop r10	
1b29e	pop r9	
1b2a0	pop r8	
1b2a2	pop r7	stack byte repair
1b2a4	pop r6	stack byte repair
1b2a6	pop r5	stack byte repair
1b2a8	pop r4	
1b2aa	ret	

Fig. 5. write_mem_gadget gadget

ADDR. We do this in an effort to minimize the damage to the current stack frame by utilizing the buffer space to store the attack payload unlike a traditional ROP attack that fills the buffer with random padding bytes. The stk_move gadget pops from the stack into registers r28 and r29 and writes the values into the location where the stack registers are stored. This is possible due to how AVR uses memory mapped registers. It is crucial to move the SP in order to perform the "clean-return" that allows the target program to keep executing. Now that SP is at ADDR, we can utilize the vulnerable buffer for the ROP payload. Once the payload has executed and returned, we are faced with another issue, the damage done to the stack. Particularly the damage done to registers r28 and r29, in addition to the original return address of the exploited gadget which are damaged during the execution of stk_move.

We thus utilize the write_mem_gadget to repair the stack frame after the execution of the payload. After the execution of the write_mem_gadget, the runtime stack as well as the current program context is restored to the state they

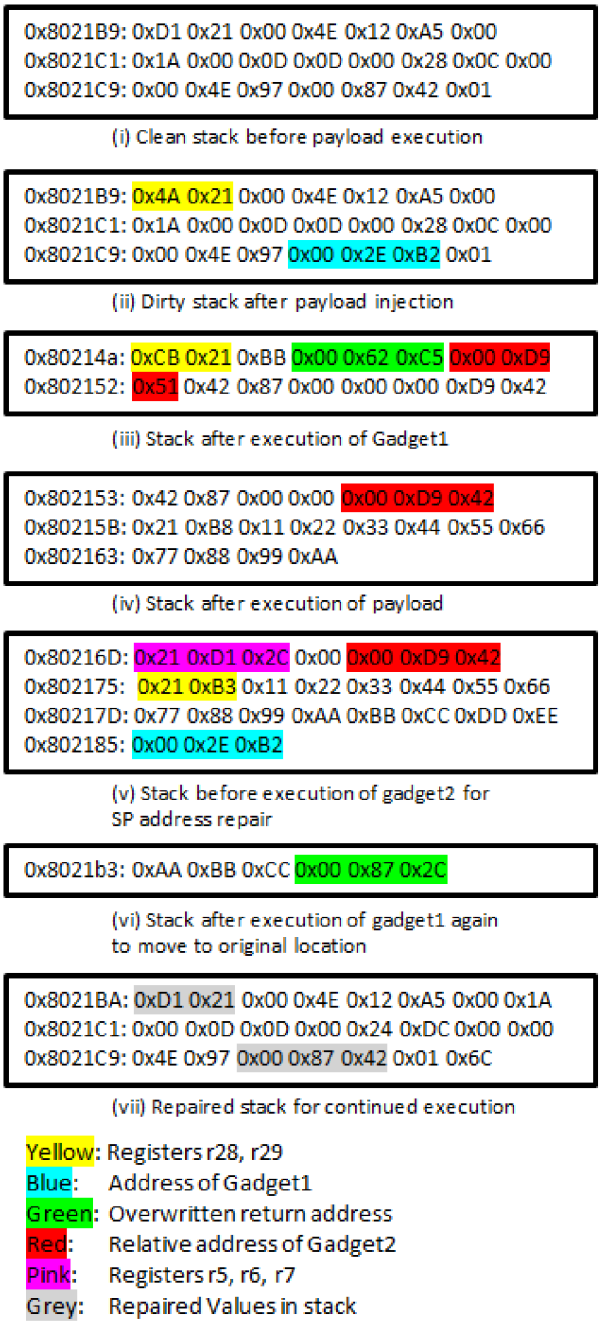

Fig. 6. Stack progression during attack

had before the execution of the ROP attack. This allows the normal execution of the target program to continue. The stack progression during this attack execution is shown in Figure 6.

E. ROP Attack V3 – Stealthy Attack with Large Payload

In the above attack, the length of the vulnerable buffer limits the size of attack payload. With the clean return attack strategy described above, we can make an additional step forward to create an arbitrarily large payload attack. For this improved attack, we use a trampoline technique that utilizes the same two gadgets used in the previous attack. In the initial stage we still utilize the stk_move gadget to move SP to the

beginning of the buffer and in the final stage we utilize the `write_mem_gadget` to repair the stack.

However, since we are able to load arbitrary values into memory by utilizing the `write_mem_gadget`, we easily construct a larger attack that is not bound by the buffer. In order to do so, we first utilize the clean return to initially load a payload into the buffer. Once this is done we utilize the `stk_move` gadget again to move `SP` to an unused portion of memory. Then by utilizing the `write_mem_gadget` we load and store the contents of the payload into this new location. Upon completion we utilize the `stk_move` gadget again to move `SP` back to its original position and repair the current frame. By utilizing this strategy we can load arbitrarily large payloads into memory. The size of the payload is bounded only by the amount of free memory. Once the large attack payload has been loaded into memory we utilize the `stk_move` gadget to move `SP` to the top of the attack string. Upon completion of the attack we again utilize the `write_mem_gadget` to repair the original stack frame and return. This allows the board to resume normal execution as though the attack never happened.

V. MAVR Defense Technique

We now describe our MAVR defense technique to mitigate ROP attacks on the APM platform. MAVR uses a fine grained randomization technique to randomize an application (such as the autopilot code) so that an attacker is unable to determine the gadget addresses and hence unable to construct a meaningful ROP attack payload. In this section, we first describe the hardware design of MAVR and then elaborate on the software aspect of MAVR that deals with the application randomization. Our solution operates directly on AVR 8-bit ELF binaries without requiring access to their source code. However, our implementation does require the ELF binary to be compiled with certain flags as discussed later in section VI-B1.

A. MAVR Hardware Design

The hardware design of MAVR extends the basic APM hardware by introducing an external flash memory chip and an additional processor, referred to as the *master processor* (as seen in figure 7). The original processor on the APM hardware that is responsible for executing the autopilot code is referred to as the *application processor*. The details of these major hardware components are discussed below.

1) External Flash Memory: We use an external flash memory to store the original unrandomized application binary and symbol information. This information is uploaded onto this memory chip at the time of flashing. This flash chip serves as the only entry point to introduce new code onto the MAVR system. The randomized binary is never stored on this external flash memory and the application processor never reads from this flash memory. This ensures complete isolation between the original application binary and the randomized binary.

The storage capacity of the flash memory chip used in MAVR is limited to the same size as the target application processor. This is because we cannot have a larger external storage for the generated binary than the processor can handle.

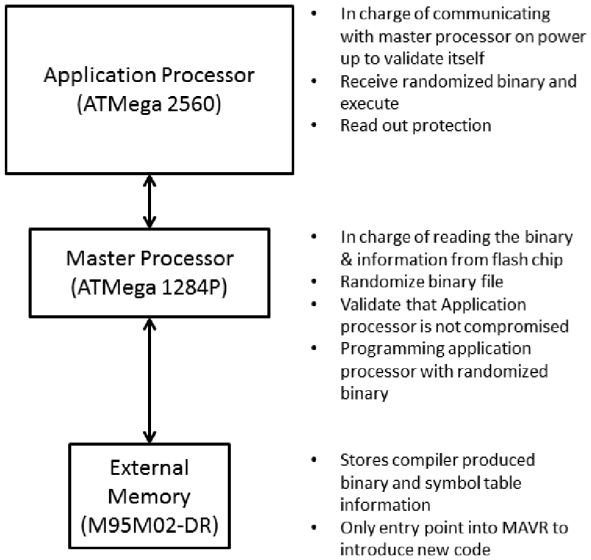

Fig. 7. MAVR System diagram

2) Master Processor: Since a microcontroller cannot read it's own binary and flash itself, we utilize a second processor as the master processor to perform the application randomization. The master processor is responsible for reading the application binary along with the symbol information from the external flash memory and randomizing it. It programs the application processor on the APM board with this newly generated randomized binary. At this stage, a randomized binary is uploaded to the application processor. Upon completion of such upload, the master processor listens to the application processor and performs simple timing analysis to determine whether a failed attack has occurred.

3) Application Processor: The application processor is the core component of the entire MAVR system as it is responsible for the control of the UAV. It needs to communicate with all the sensors and make control decisions based on various sensor inputs. Thus, it is crucial to prevent an attacker from obtaining a copy of the current binary (that is, randomized binary) stored in the application processor's flash memory. This is achieved in MAVR by enabling the read out protection fuse on the application processor. This prevents the attacker from obtaining a copy of the randomized binary and guarantees complete isolation from the original unrandomized binary.

4) Cost: With modification to the architecture and introduction of new chips on the board there will be an increase in cost. For our prototype the price of ATMega 1284P was $7.74 per chip and the M95M02-DR flash memory chip was $3.94, when purchased in a batch of ten for prototyping. This is a total of a $11.68 increase in cost of the hardware materials. These numbers are based on batches of ten, during manufacturing the chips will be purchased at higher quantities and the increase in cost will be less. This may seem like a significant cost; however in comparison to the base price of the APM board of $159.99 it is only a 7.3% increase in materials cost. This is a small increase in cost for the added benefits it provides when combined with the software solution.

B. MAVR Software design

Code-reuse attacks assume a predictable address layout of an application's executable code that enables them to identify gadget addresses and construct attack string. MAVR's randomization technique aims at breaking these assumptions by shuffling the function blocks in the binary's `.text` section. The frequency of randomization depends on certain factors as described in section V-C. Frequent fine-grained randomization significantly increases the difficulty of these attacks since the attacker would need to guess the exact randomizing permutation being used in the application's current execution.

1) Preprocessing phase: The first step in MAVR processing is the preprocessing phase that is responsible for identifying the function blocks within the application binary to be randomized. In the preprocessing phase, the ELF binary is parsed to extract the function symbols and related information such as the start address of the function and the length of the individual function block. Since the flash utility strips all symbol information from the binary before uploading it onto the board, we modified it by constructing our own symbol table (using information parsed earlier) and prepending it to the application's hex file. This phase occurs on the host development machine. Upon completion of the preprocessing phase, the hex file consisting of the target binary and the symbol table is uploaded to the external flash memory, ready for the next phase.

2) Randomization Algorithm: Once the preprocessing is completed, the function blocks have been identified and the modified binary uploaded to the external memory. The MAVR's master processor then generates a random permutation of this set of symbols. The function blocks are then moved within the binary according to this random permutation. By doing this, the relative offsets between instructions are changed and this may affect different jump and call instructions. The reason is that the original destination address for the jump/call instructions can be specified either as a relative or as an absolute address. Thus, when the function blocks are shuffled, the targets of jump/call instructions are no longer valid and must be corrected to point to the desired location. This is achieved by performing jump and call patching as discussed next.

3) Patching: Upon completion of the randomization phase, several jump/call instructions will no longer work. Each of these now needs to be 'fixed' to point to the proper locations. This is achieved by performing jump patching as the binary is written to the application processor by the master processor. During this process, we utilize the information stored in the symbol table that was constructed earlier and uploaded on the external flash memory. With this information and the new address of each function, we can calculate the new relative offset and absolute addresses for each jump/call instruction to properly patch the binary on the application processor. This is discussed in more detail later in section VI-B3.

C. Frequency of Randomization

One important parameter in MAVR defense is the frequency of randomization. Randomizing frequently, such as at every application restart, will result in a stronger defense. However, since every randomization will require the application processor to be reprogrammed, this will significantly reduce the lifetime of the processor. Thus, a tradeoff needs to be achieved so that the application is randomized after a certain number of restarts rather than every restart. MAVR can be configured so that it can randomize every time the master processor boots or with any desired frequency. However, upon detection of any failed ROP attack, the binary is immediately randomized again. Thus, to defeat MAVR an attacker would need to dynamically construct a new exploit for not only every instance of every application but also for every attack.

D. Security Evaluation

We now evaluate MAVR randomization technique and show that it significantly increases the brute force effort required to attack the system. An attacker utilizing a brute force strategy will usually assume a memory layout and craft an exploit payload based on that permutation of the address layout. A failed attempt will result in the program counter to be incremented incorrectly thus executing garbage bytes. This will result in a system crash and cause the master processor to restart the application processor. A successful attempt is assumed to be equivalent to guessing the correct permutation of functions used for randomization.

Given a single permutation of the randomization every failed attempt will eliminate one permutation. The probability that success is achieved at the j^{th} attempt is

$$P(j) = \left(\prod_{i=1}^{j-1} \frac{N-i}{N-i+1}\right) * \frac{1}{N-j+1} = \frac{1}{N}$$

The average number of attempts before success can be computed as

$$\mathrm{E}[X] = \sum_{x=1}^{N} x * P(x) = \sum_{x=1}^{N} x * \frac{1}{N} = \frac{N+1}{2}$$
$$\Rightarrow \mathrm{E}[X] = \frac{n!+1}{2}$$

So, the attacker will need an average $\frac{n!}{2}$ number of brute attempts to correctly guess the randomization and launch successful ROP attack as originally calculated in shown in [22]. In MAVR, upon detection of a failed ROP attack, the master processor resets the board and randomizes the target binary again. Thus, the average case for a brute force attack on MAVR is $\frac{n!+n!}{2} = n!$. By utilizing this strategy in combination with the readout protection fuse, MAVR is able to provide multiple layers of defense increasing the complexity of attacking MAVR. By design, there is no way for an attacker to gain access to the randomized code that is executing. The reason is that the target binary is physically isolated on the application processor. This gives a huge advantage in that the executing randomized binary is never exposed to the attacker for any static analysis.

VI. Implementation Details

The implementation was done for a 8-bit AVR architecture and consists of three major steps. The first step flashes the users' binary to an external flash chip. The second step randomizes the executable code and generates the randomized

binary. Finally, the third step flashes the randomized binary to the primary processor for execution. We discuss the details of the MAVR implementation below.

A. Hardware Design

Large applications like ArduPlane can use over 90% of the available memory on the AVR platform, leaving little to no space for code to perform address space layout randomization. Therefore, a second AVR processor, the Atmel ATmega1284P [2], was incorporated as a master processor to perform the randomization. This processor is available in a smaller package with fewer pins and at a lower cost than the current ATmega2560 application processor; in addition, it does not need to scale up for larger devices, keeping the system cost low. An external flash memory chip sized to match the program memory of the application processor [37] is also connected to the master to store the original program.

At runtime, this processor is in charge of reading the preprocessed application binary. Using this information, a new randomly generated application is uploaded to the slave ATmega2560 processor at application start time. Readout protection fuses are set on this processor to prevent an attacker from discovering the content of the executing application with a debugger. As performing randomization on every board startup would quickly exhaust the 10,000 programming cycles of the embedded program memory [2], a periodic schedule must be implemented to randomize based on the security level of the device. Post randomization the master processor then assumes a role similar to a watchdog timer listening to the application processor. By doing so the master processor can easily detect when a failed attack has occurred since the application processor will not feed the master by signaling high for a period of time.

As seen in figure 8 the prototype constructed in the development of MAVR contains an Arduino Mega as the application processor since it utilizes the same Atmega 2560 processor as the APM 2.5 board. In addition a Debug LCD is attached to the master processor for development purposes to verify that different components were communicating properly with each other.

B. Code Randomization

Randomizing an application's executable code segment on AVR consists of three stages: preprocessing, randomization, and programming. The preprocessing stage is conducted on a host computer before flashing the target binary to the MAVR system, while randomization and programming tasks are carried out by a secondary onboard processor at run time.

1) Compilation: Applications for AVR architectures such as the ATmega2560 are usually compiled with the GNU Compiler Collection [21], or with the manufacturer-provided Atmel Studio which uses GCC under the hood. However, the need for compact code and fast execution on embedded systems often perpetuates optimizations that are unfriendly to address space layout randomization.

For example, the GCC linker by default will replace jump and call instructions with the short-ranged relative versions where possible. When working with a static binary, this behavior is acceptable, but the assumptions of fixed function locations are not valid for the MAVR platform. Using the command-line linker option `--no-relax` will prevent this behavior, but this flag exposes bugs in the GNU Binutils tools [20] underlying GCC which cause wrong code generation. Replacing the stable Binutils version with a recent development snapshot version 2.24.51 resolves the problem.

Another optimization technique used by GCC is to group commonly used instructions in function prologues and epilogues used to push and pop registers and allocate buffers on the stack. The AVR calling convention [2] specifies a number of registers as callee save, so any modifications in the function require these registers to be saved in the prologue and restored in the epilogue. To reduce the number of redundant pushes and pops, many developers use the C compiler option `-mcall-prologues` to generate a jump instruction into the midst of a block of predefined push and pop instructions.

While this option essentially consolidates most gadgets into one area, the resulting very useful gadget has hundreds of references scattered throughout the program which are prone to leaking information about its new location. Disabling call prologues with `-mno-call-prologues` in the application configuration is not sufficient; the C standard library (AVR-Libc) [4] and the GCC compiler support library `libgcc` itself also had to be recompiled with these flags disabled to completely eliminate call prologues. GCC 4.5.4 and AVR-LibC 1.8.0 were used for the resulting custom toolchain, as this version of GCC had fewer outstanding incorrect code generation bugs than recent versions and could compile the ArduPlane software without crashing or generating errors.

2) Preprocessing: ELF (Executable and Linkable Format) files generated by the C compiler have useful symbol table information which aids in the randomization process. However, this information is typically removed from the binary during its conversion to an Intel HEX file before being uploaded to the MAVR system. The preprocessing stage modifies the HEX file to prepend the most important symbol information required to perform the randomization. As no modifications to the application binary code are performed at this time, the preprocessing stage is independent of subsequent executions of the target binary.

The standard ELF symbol table contains information about the functions and data sections present in the application. A list of all functions is compiled from this function table, and a list of function start addresses in ascending order is added to the HEX file to allow the MAVR system to move functions as blocks. In addition, references in the data section are scanned for function pointers. Any pointers found, particularly C++ class vtables and global arrays of functions used in some applications for call routing, are also added to the HEX file to allow MAVR to update these locations at runtime. It is possible to encode function pointers in the instruction stream with load immediate (LDI) instructions, but the C compiler will not generate a sequence like this and will encode an absolute or relative call instead.

Once the modified HEX file is generated, standard tools such as `avrdude` [3] are used to upload the file to the MAVR system. An application on the master processor receives the HEX file and stores it verbatim in the external flash memory

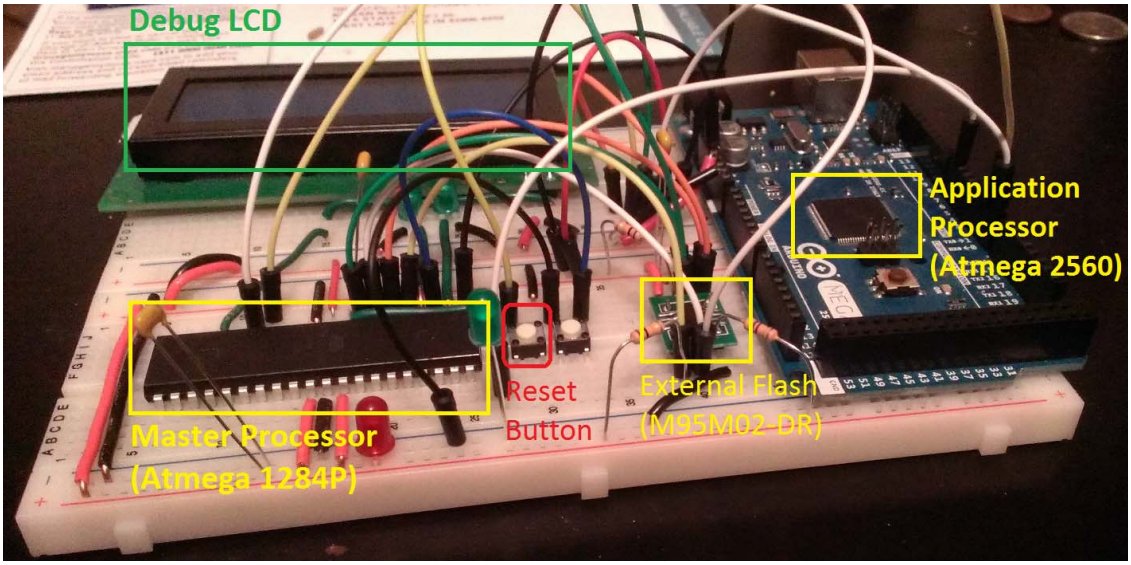

Fig. 8. Hardware prototype for MAVR

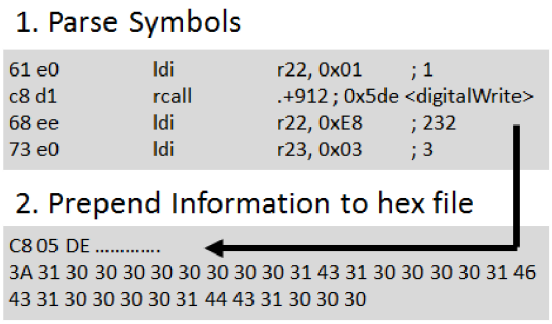

Fig. 9. Prepend symbol information to HEX file

chip. At this time, the external chip is the same size as the memory on board the ATmega2560 slave processor, leading to the possibility of memory exhaustion if a large number of symbols need to be encoded in addition to a binary that is perilously close to the maximum allowable size. Even with the extremely complicated ArduPlane software, this event did not occur in this analysis, but production systems should allocate a larger external storage chip to preclude this failure mode.

3) Randomization: When the software determines that application address randomization is required, perhaps by a program crash detected or a time interval since the last update, the master processor triggers the reset pin on the application processor forcing it into the bootloader state and then reads the preprocessed binary from the flash memory chip. First, the list of functions is read in ascending order, and a copy of the addresses is shuffled to create a map of old function addresses to new addresses. A list of memory addresses containing function pointers is also read from the flash memory at this time.

The entire binary is then read in a few bytes at a time and incrementally sent to the application processor. Since the external flash memory permits random access, each function can be processed in a streaming fashion, eliminating the need to fit the entire application into volatile memory at runtime. If the current address is inside a constant or data section, the list of function pointer addresses is used to determine if the current location needs to be updated. As the application is executing, these pointers can end up in volatile or non-volatile memory, but the flat memory application binary will always contain the original values in some form which can be detected by preprocessing.

If the current address is inside an executable section, call and jump instructions are updated to reflect the new addresses of each function. Removal of call prologues and relaxation forces all call and jump instructions into their long form, allowing the entire memory space to be addressed from any point by any function call. Some call and jump instructions used as trampolines for switch case statements do not point exactly to an address in the table, so a binary search is used to determine the largest old symbol address that is less than or equal to the targeted address; the offset is calculated and used to adjust the new address to ensure correct code execution.

4) Programming: The ATmega2560 processor is commonly fitted with a boot loading functionality that works over its primary asynchronous serial port for in-application programming [2]. This functionality is invoked by briefly asserting the `RESET` line and sending a specific byte sequence within a few milliseconds after boot. The randomized binary is then incrementally transferred to the application processor; the bootloader performs the work of writing the data to the non-volatile program memory. After the entire binary is processed, a reset command is issued by the master processor to exit the bootloader and begin normal program operation. As the software bootloader must sit at a fixed location, it provides targets for an ROP attack; in a production system, the hardware implemented In-System Programming functionality [2] of the application processor would be used instead of a serial boot loader to remove any static code from program memory.

TABLE I. NUMBER OF FUNCTIONS

Application	Number of Functions
Arduplane	917
Arducopter	1030
Ardurover	800

VII. EVALUATION

We now describe various experiments that were performed to evaluate the MAVR technique. These experiments test the effectiveness of MAVR and also the performance overhead incurred due to the randomization. These experiments were performed on common autopilot applications for UAVs.

A. Effectiveness

First, we tested the effectiveness of MAVR using the same test application that we utilized in section IV to develop the stealthy attack. We found 953 gadgets in this target application and were successful in crafting a successful ROP attack using these gadgets. Next, we randomized this application using MAVR and tried to attack it using stealthy ROP attacks (as described in section IV). None of the previously described stealthy attacks were successful on the randomized application. Instead, the board started executing garbage. MAVR detected this and reflashed the board.

This highlights the sensitivity of these attacks to slight modifications in the address layout. ROP attacks require precise knowledge of the address layout of the target executable code. In our threat model, the attacker only has access to the unprotected binary and is not aware of the exact permutation that has been used for randomization. Thus, they can only analyze the unprotected binary and not the randomized version.

1) Brute force effort: In section V-D, we discussed the average number of brute force attempts required to successfully attack a randomized binary. This brute force effort is approximately $n!$ attempts where n is the number of symbols in a binary. To measure this effort, we evaluated three different applications for the application processor. Table I shows the number of function symbols in these applications. We observed that the autopilot applications had at least 800 symbols present, thus requiring approximately 800! attempts for brute force. In addition, we observed an average of 915 symbols and a median of 917 symbols for these applications.

It is interesting to note that the effectiveness of the MAVR defense increases as the modularity of the code increases. That is, good code design that utilizes more modules also increases the number of symbols that can be shuffled around by MAVR, hence increasing brute force effort.

B. Efficiency

1) Startup overhead: MAVR defense technique introduces a startup overhead due to the additional processing cost incurred by MAVR for randomizing the application and programming the application processor upon boot. This overhead is incurred only when the application needs to be randomized (see section V-C), otherwise the application starts up normally. We evaluated the efficiency of MAVR by measuring this startup overhead as shown in Table II. We utilized the same autopilot applications and found that it took an average of 18609 ms with a median of 19209 ms for the application to be randomized and transferred to the application processor.

TABLE II. MAVR STARTUP OVERHEAD

Application	Time (ms)
Arduplane	19209
Arducopter	21206
Ardurover	15412

TABLE III. CHANGE IN CODE SIZE

Application	Stock Code Size (bytes)	MAVR Code Size (bytes)
Arduplane	221608	221294
Arducopter	244532	244292
Ardurover	177870	177556

Two things to note here are that this time directly correlates to the size of the target application and we are limited by how fast we can transmit the bytes via serial port to the application processor. For our prototype design, we are limited to 115200 baud rate which allows for a maximum of 11 bytes per millisecond transfer rate. In a full production PCB, this factor would be eliminated as impedance control could be applied allowing the master and application processor to communicate at mega-baud rates. A conservative estimate on a production PCB of the startup overhead that MAVR introduces would be 4 seconds as the bottleneck becomes how fast we can write the randomized binary to the application processor's internal flash.

2) Code Size: In addition to the startup overhead, we measured the increase in code size of these applications. This is due to the heavy modifications that we had to make to the libraries and compiler flags, and we wanted to test and see if there was a significant change in code size. However, as we discovered there was actually a decrease in code size with our randomized binaries. For example, in Arduplane 2.7.4 the stock code size is 221608 bytes but when compiled with our custom toolchain the code size was found to be only 221294 resulting in a 0.001% reduction. Table III shows the result of this analysis on the three test applications. This is due to the fact that we used compilation flags `-mcall-prologues` and `--no-relax`. As stated in section VI-B1, both these flags respectively increase and decrease the code size.

VIII. DISCUSSION

A. Software Solution

When considering the best approach to mitigate issues discussed in section IV, initially a software only solution was contemplated. The solution was to randomize the application binary at flash time as it was being written into the board. However multiple issues were found with this approach. One such issue is the fact that when the hardware is deployed, it contains only a single permutation of the randomization.Successive failed ROP attempts could then be utilized to leak information to the attacker as the randomization will not change. Thus, using techniques such as those demonstrated in [34], an attacker can guess the permutation in a much shorter time. In addition a software only solution is not fault tolerant. As described in section V-D a failed attempt will result in the application processor executing garbage bytes and becoming inoperable. The only way to recover from this problem is to reset the application processor by cycling its power source which is extremely difficult when a UAV is in flight. The small

increase in hardware cost mitigates this software problem. Thus, the combination of software and hardware solution was selected for MAVR.

B. Entropy

One approach considered to increase MAVR's entropy was to introduce random padding between each function. This is possible due to the fact that there is a small finite amount of memory on the APM board which would give the attacker an advantage. However after implementing MAVR and evaluating it we discovered this was not necessary. For the smallest autopilot software, Ardurover, contains 800 symbols that we could randomize. This generates 6567 bits of entropy which is computationally secure against a brute force attack.

IX. Related Work

In this section, we discuss the relevant research work for code reuse attacks and defense techniques.

A. Code reuse attacks

ROP attacks, introduced by Shacham et al [33], reuse existing executable code in an application towards malicious purposes. These attacks have been shown to work on both word-aligned architectures like RISC [9] and unaligned CISC architectures [31]. Variations of ROP attacks, such as jump oriented programming attacks [7], [13], have been proposed that do not rely on `ret` instructions but also utilize gadgets that end in jump instructions. Unlike our work, these attack techniques do not consider the stealthy property of the attack and corrupt the stack frames in the process of executing the attack.

Our work focuses on attacks on embedded systems, in particular UAV control systems based on APM platform. Some attacks on other embedded systems have also been proposed. Francillon et al. [17] used ROP techniques to inject code permanently into the memory of Harvard architecture based sensor nodes. Teso [38] demonstrated attacks on commercial flight management systems (FMS) by leveraging the lack of security in ADS-B and ACARS communication that is used by aircrafts to upload malicious data to FMS. Verdult et al. [39] recovered the secret key used by cars by remotely exploiting the weak cipher used by Hitag2 (used in car immobilizer). Checkoway et al. [10] performed a comprehensive analysis of attack surface in automobiles and demonstrated the feasibility of remote exploitation by uncovering various attack vectors such as Bluetooth and cellular radio. Our attack has a stealthy property to avoid detection and is targeted towards AVR architecture. In addition, we also propose a randomization based defense technique to mitigate such attacks.

B. Defenses

Several defense techniques for mitigating ROP attacks have been proposed. Address obfuscation [5] and ASLR [30] are two well-known defense techniques against such attacks. However, they suffer from small amounts of randomization available and have been shown to be vulnerable on 32-bit architectures [34], [32]. This immediately dismisses them as a valid defense on the AVR 16-bit address space due to the reduction in entropy. Instruction set randomization (ISR) [25], another well known defense technique, has also been shown to have similar limitations [36]. Several fine grained diversification techniques have been proposed as a defense against code reuse attacks such as ILR [23], In-place randomization [29], STIR [40], Marlin [22], XIFER [16], Librando [24], Code Shredding [35], ASR [19], Genesis [41], nop-insertion [18] and Bhatkar et al. [6]. The software design of MAVR is inspired by these techniques but it addresses several challenges that are unique to AVR 8-bit architecture.

Approaches such as checking stack will lead to runtime overhead that is unacceptable in resource-constrained embedded systems. APM 2.5 board running Arduplane software is already at 96% CPU usage (III) and would not have enough spare cycles to perform additional runtime checks. MAVR does not use any runtime data structures or monitoring, thus making it very efficient with minimal overhead. Further, techniques such as ASLR, canaries and stack-splitting do not recover from a failed attack and lead to undefined program state. This is unacceptable for systems such as drone control units as this would result in UAV performing erratically. MAVR addresses this (reference sections V-C, V-D), where upon detection of a failed ROP-attack, the master processor resets and re-randomizes the binary on the application processor. This allows for safe recovery of UAV in-flight.

Compiler based solutions that create code without return instructions have also been proposed [28], [26], but these are unable to handle ROP variants such as jump oriented programming [7] attacks. Another mitigation tactic for code reuse attacks is to detect and terminate the attack as it occurs. Examples of these include DROP [11], DynIMA [14], CCFIR [43], CFL [8], ROPdefender [15], [12] and [44]. The dynamic monitoring approach used by these techniques make them unsuitable for our target platform where the processor is already at 96% usage. Our target platform for UAV systems calls for efficient defense techniques that do not incur high runtime overhead. All of the above techniques are mainly software based solutions, unlike MAVR which incorporates both software and hardware into the defense technique.

X. Conclusion and Future Work

In this work, we proposed a new stealthy approach to ROP attacks on an embedded system that allows the attacker to craft multiple attacks and execute them continuously without the ground station ever detecting that the control of UAV has been compromised. In addition, we demonstrated a defensive technique, MAVR, that mitigates these ROP attacks by significantly increasing the brute force effort required to successfully attack MAVR. Based on our implementation and analysis of the results, we assert that randomization at a function level on AVR is both feasible and practical as a defense against these code-reuse based attack techniques.

It is important to point out that MAVR is not limited to only UAV applications. We focused on UAVs for this paper as it is one of the growing applications for MAVR. Other applications where MAVR framework could be utilized are any networked embedded systems utilizing a real time operating system (RTOS). MAVR has been designed as a security solution that entails end to end protection. Future work will evaluate the effectiveness of the stealthy attack and

mitigation technique on other embedded applications. Some examples include home security systems like Scout Alarm, home automation systems such as Neurio, sensor networks, and the ARM based PX4 alternative to the APM board. Each of these systems utilize networked devices that depend on an RTOS meeting strict deadlines while presenting attack surfaces to be leveraged in malicious ways that MAVR could mitigate.

REFERENCES

[1] Ardupilot Community. APM Multiplatform Autopilot. http://ardupilot.com/.

[2] Atmel Corporation. 8-bit Atmel Microcontroller. http://www.atmel.com.

[3] AVR Dude Team. AVRDUDE - AVR Downloader/Uploader. http://www.nongnu.org/avrdude/.

[4] AVR LibC Team. AVR Libc Home Page. http://www.nongnu.org/avr-libc/.

[5] E. Bhatkar, D. C. Duvarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *2003 USENIX*, pages 105–120, 2003.

[6] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. SSYM'05, pages 17–17, 2005.

[7] T. Bletsch, X. Jiang, and V. Freeh. Jump-oriented programming: A new class of code-reuse attack. Technical Report TR-2010-8, North Carolina State University, 2010.

[8] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. ACSAC '11, pages 353–362, New York, NY, USA, 2011. ACM.

[9] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to risc. In *ACM CCS 2008*, pages 27–38, 2008.

[10] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno. Comprehensive experimental analyses of automotive attack surfaces. SEC'11, pages 6–6, Berkeley, CA, USA, 2011. USENIX Association.

[11] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. DROP: Detecting return-oriented programming malicious code. In *ICISS 2009*, pages 163–177, 2009.

[12] P. Chen, X. Xing, H. Han, B. Mao, and L. Xie. Efficient detection of the return-oriented programming malicious code. In *ICISS 2010*, pages 140–155, 2010.

[13] P. Chen, X. Xing, B. Mao, and L. Xie. Return-oriented rootkit without returns (on the x86). In *ICISS 2010*, pages 340–354, 2010.

[14] L. Davi, A.-R. Sadeghi, and M. Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *ACM SDC 2009*, pages 49–54, 2009.

[15] L. Davi, A.-R. Sadeghi, and M. Winandy. ROPdefender: a detection tool to defend against return-oriented programming attacks. In *ACM CCS 2011*, pages 40–51, 2011.

[16] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi. Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and arm. ASIA CCS '13, pages 299–310, New York, NY, USA, 2013. ACM.

[17] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *ACM CCS 2008*, pages 15–26, 2008.

[18] M. Franz, S. Brunthaler, P. Larsen, A. Homescu, and S. Neisius. Profile-guided automated software diversity. CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

[19] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX 2012*, Security'12, pages 40–40, Berkeley, CA, USA, 2012. USENIX Association.

[20] GNU Bin Utils Team. GNU Binutils. http://www.gnu.org/software/binutils/.

[21] GNU Compiler Collection Team. GCC, the GNU Compiler Collection. http://gcc.gnu.org/.

[22] A. Gupta, J. Habibi, M. Kirkpatrick, and E. Bertino. Marlin: Mitigating code reuse attacks using code randomization. *IEEE TDSC*, PP(99):1–1, 2014.

[23] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. Ilr: Where'd my gadgets go? In *2012 IEEE S&P*, pages 571–585, 2012.

[24] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Librando: transparent code randomization for just-in-time compilers. CCS '13, pages 993–1004, New York, NY, USA, 2013. ACM.

[25] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. CCS '03, pages 272–280, New York, NY, USA, 2003. ACM.

[26] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *Eurosys 2010*, pages 195–208, 2010.

[27] MAVLink Team. MAVLink: Micro Air Vehicle Communication Protocol. http://qgroundcontrol.org/mavlink/start.

[28] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *ACSAC 2010*, pages 49–58, 2010.

[29] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. SP '12, pages 601–615, Washington, DC, USA, 2012. IEEE Computer Society.

[30] PaX Team. PaX. http://pax.grsecurity.net/.

[31] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, Mar. 2012.

[32] G. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *ACSAC '09*, pages 60 –69, dec. 2009.

[33] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM CCS 2007*, pages 552–561. ACM, 2007.

[34] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM CCS 2004*, pages 298–307, 2004.

[35] E. Shioji, Y. Kawakoya, M. Iwamura, and T. Hariu. Code shredding: Byte-granular randomization of program layout for detecting code-reuse attacks. ACSAC '12, pages 309–318, New York, NY, USA, 2012. ACM.

[36] A. N. Sovarel, D. Evans, and N. Paul. Where's the feeb? the effectiveness of instruction set randomization. In *Usenix 2005*, pages 10–10, 2005.

[37] STMicroelectronics. 2-Mbit serial SPI bus EEPROM.

[38] H. Teso. Aircraft hacking: Practical aero series. In *4th Annual Hack In The Box Security Conference (HITBSECCONF2013)*, 2013.

[39] R. Verdult, F. D. Garcia, and J. Balasch. Gone in 360 seconds: Hijacking with hitag2. In *2012 USENIX*, Security'12, pages 37–37, Berkeley, CA, USA, 2012. USENIX Association.

[40] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. CCS '12, pages 157–168, New York, NY, USA, 2012. ACM.

[41] D. Williams, W. Hu, J. Davidson, J. Hiser, J. Knight, and A. Nguyen-Tuong. Security through diversity: Leveraging virtual machine technology. *Security Privacy, IEEE*, 7(1):26–33, Jan 2009.

[42] A. Wright. Hacking cars. *Commun. ACM*, 54(11):18–19, Nov. 2011.

[43] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *IEEE S&P 2013*, pages 559–573. IEEE Computer Society, 2013.

[44] M. Zhang and R. Sekar. Control flow integrity for cots binaries. SEC'13, pages 337–352, Berkeley, CA, USA, 2013. USENIX Association.