Run-DMA

Michael Rushanan
Johns Hopkins University

Stephen Checkoway
University of Illinois at Chicago

Abstract

Copying data from devices into main memory is a computationally-trivial, yet time-intensive, task. In order to free the CPU to perform more interesting work, computers use direct memory access (DMA) engines — a special-purpose piece of hardware — to transfer data into and out of main memory. We show that the ability to chain together such memory transfers, as provided by commodity hardware, is sufficient to perform arbitrary computation. Further, when hardware peripherals can be accessed via memory-mapped I/O, they are accessible to "DMA programs." To demonstrate malicious behavior, we build a proof-of-concept DMA rootkit that modifies kernel objects in memory to perform privilege escalation for target processes.

1 Introduction

Modern computers contain a variety of special purpose, "auxiliary" processors designed to offload specific tasks from the CPU, freeing the CPU to perform other work. Conceptually, the CPU copies data from main memory to the auxiliary processor and requests that it perform its function. When the auxiliary processor has completed its task, it signals the CPU that it is finished. In reality, if the CPU were responsible for copying the data, it would spend most of its time performing data transfers, for example, copying memory to the GPU or network controller. Instead, computers have specialized hardware called *direct memory access* (DMA) engines that perform the copying to and from the auxiliary processors. The DMA engines perform the data transfers in parallel with the computation performed by the various processors by utilizing otherwise-free memory-bus cycles. In this paper, we show that DMA engines, despite their limited functionality, are nevertheless capable of performing Turing-complete computation.

At the same time that computer systems have been gaining additional processors, computer security researchers and practitioners have begun to recognize that the once bright-line separation of code and data is perhaps not so bright. For example, the threat of software exploitation has undergone a paradigm shift from a *malicious code* model (i.e., attacker-delivered payloads), to a *malicious computation* model where the attacker crafts data inputs to induce arbitrary computation on a target system [38]. This style of data-only attack goes by various names including return-oriented programming (ROP) [6, 13–15, 20, 23, 27, 30, 35, 37] and weird machines [4, 10, 38, 47].

The ability to induce arbitrary computation from nothing more than copying bytes from one address to another may be surprising to those who are not steeped in the arcana of weird machines.[1] And indeed, it is a surprisingly strong statement: Any function that can be computed by a Turing machine can be computed using DMA.[2] The induced computation of ROP or weird machines generally takes the form of a sequence of "gadgets" which the attacker strings together to perform the desired computation. Each gadget typically performs some discrete action such as "add two numbers together" or "store a value to memory." Once a Turing-complete set of gadgets has been constructed, any desired behavior can be "programmed" in terms of the gadgets.

Turing-complete behavior in unexpected places is not sufficient to write programs that are interesting from a security (as opposed to a computability) perspective. To be useful, a programming language needs to be what Atkinson et al. [3] call "resource complete." That is, the language needs to "be[] able to access all resources of the system [...] from within the language" [3]. By design, DMA has direct access to (some) hardware peripherals and RAM, including kernel memory and memory-mapped I/O registers.[3] Thus, a Turing-complete set of DMA gadgets should also be resource-complete.

In order to build DMA gadgets, we require several capabilities of the DMA engine. In particular, the DMA engine (1) must be capable of performing memory-to-memory copies; (2) can be programmed by loading the address of DMA *control blocks* or *descriptors* into memory-mapped registers; and (3) supports a *scatter/gather mode* where DMA transfers can be chained together, typically by providing the address of the next control block or descriptor.

Some DMA engines lack capability 1; for example, the Intel Platform Controller Hub EG20T DMA controller only supports transferring data between main memory and PCI memory [24, Chapter 12]. For DMA engines with similar restrictions, capability 1 can be relaxed as long as the restricted source/target memory contains

[1] For example, the x86 `mov` instruction is Turing-complete [16].

[2] As we show in Section 5, DMA transfers can perform sequential interactive computation à la Persistent Turing Machines [22].

[3] In some systems an IOMMU unit may restrict DMA access to certain regions of memory.

a byte that could be used as a staging area enabling memory-to-memory copies by transferring data first to the restricted space and then back to memory.

For ease of implementation and testing, our work targets the Raspberry Pi 2's DMA engine (see Section 2) and thus we make no claim that our results hold for other systems. That said, we believe that the three required capabilities listed above are satisfied by modern DMA engines. For example, the following appear to meet our requirements: Intel 8237 (e.g., legacy IBM PC/ATs), CoreLink 330 [2] (i.e., ARM Advanced Bus Architecture compliant SoCs), Cell multi-core microprocessor [40] (e.g., Sony Playstation 3), and Intel's I/O Acceleration Technology [25] (e.g., Intel Xeon Server).

Our work differs from traditional DMA malware — that is, malware that runs on an auxiliary processor such as a GPU and leverages that processor's DMA access — in that it runs entirely in the DMA engine. An attacker need only access hardware registers to exhibit control. This can be achieved in user space with administrator permissions on the Raspberry Pi 2 by mapping the appropriate region of physical memory [11, Chapter 4].

In this paper, we are concerned with the art of crafting Turing- and resource-complete gadget sets using a DMA engine. In particular, we do not discuss how an attacker would gain permission to reprogram a DMA engine, which typically requires administrator access, nor do we discuss the full power of so-called DMA malware as both topics are well described in prior work (see Section 7). Concretely, we

- describe the theory behind the construction of DMA gadgets (Section 3);
- build an interpreter for a known Turing-complete language and demonstrate resource-completeness (Section 4); and
- build a proof-of-concept DMA rootkit (Section 5).

2 Background

Direct memory access (DMA), is a memory bus architectural feature that enables peripheral devices, such as GPUs, drive controller or network controllers, to access physical memory independently of the CPU. In particular, DMA frees the CPU from I/O data transfer by offloading memory operations (i.e., memory-to-memory copying or moving) to the DMA engine.

In general, each DMA engine has several control registers that specify the operation of DMA transfer, including the direction of data transfer, unit size in which to transfer (e.g., a word or a byte), and the total number of bytes to transfer. DMA transfers are typically configured by the operating system but may be initiated by hardware signals.

Our work targets the Raspberry Pi 2 for implementation and testing. Specifically, the Pi is equipped with the

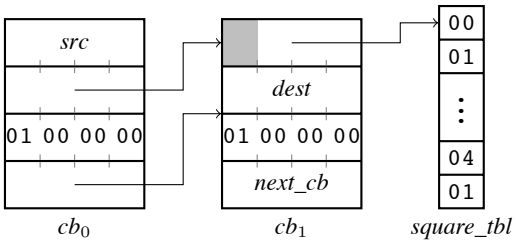


Figure 1: Square gadget. This simple gadget loads a byte x from address *src*, computes x^2 mod 256 by using x as an index into the *square_tbl*, and stores the result at address *dest*. The next control block to be loaded into the DMA engine is at address *next_cb*.

BCM2836 ARM processor which contains a 16-channel Broadcom DMA controller [11, Chapter 4]. DMA transfers are initiated by loading the address of a control block data structure into one of the channel's memory-mapped control registers. This causes the DMA engine to load the rest of its control registers from the control block.

The control block is composed of eight 32-bit words that specify not only which operation to perform, but also the address of the control block to be loaded next. The control block forms the basis of our DMA gadget construction.

3 Constructing DMA gadgets

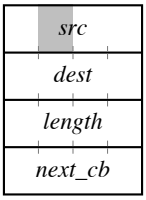
A single DMA transfer is little more than a glorified, hardware-assisted `memcpy(dest, src, size)`. As described in Section 2, on the Raspberry Pi 2, DMA transfers are initiated by loading the address of a *control block* into a memory-mapped register. Each control block contains a source address, a target address, a transfer length, and the address of the next control block to load into the engine. In addition, there are fields that control aspects of DMA transfers that are relevant to reading from/writing to DMA-supported peripherals as well as a variety of options such as 2D transfers. However, to make our results more general, we do not make use of any of these features.

Unlike traditional computer programming, constructing a DMA "program" fundamentally requires using self-modifying constructs. Each of our DMA gadgets consists of a collection of control blocks, chained together using the next control block fields, and zero or more tables of constant data. Most of the control blocks in each gadget modify one of the source, destination, or next control block fields in a subsequently-executed control block. For gadgets that perform basic operations such as increment values in memory, the final control block will copy the result to memory and then transition to the next gadget. For gadgets that perform control flow, the initial control blocks compute the address of the next control block

to "execute" and store it in the next control block field of the final control block — a trampoline — which performs no memory transfer.

In order to compute simple functions $f : \{0,1\}^8 \to \{0,1\}^8$, we use 256-byte tables where the nth entry in the table corresponds to $f(n)$. These tables are stored 256-byte aligned in memory. By putting the address of the table in the source field of a control block with a transfer length of 1, a preceding control block can select the index n by copying a byte to the least significant byte of the source address pointing to the table. Figure 1 demonstrates this by giving the control blocks and table for computing the function $n \mapsto n^2 \bmod 256$.

In Figure 1 and subsequent figures, the source, destination, transfer length, and next control block fields of the control blocks are drawn as follows.



Arrows represent pointers and shaded fields or partial fields are modified by previous DMA transfers.

In the next section, we describe how to build a Turing-complete set of DMA gadgets which we use to build an interpreter for a simple programming language.

4 A Turing-complete gadget set

In 1964, Böhm described the simple programming language $\mathcal{P}''$ and showed that it is Turing-complete. That is, it can compute every Turing-computable function [7, 8]. It holds that a program written in the language can simulate any other computational device or language. In fact, such a program can be written using only six distinct expressions in $\mathcal{P}''$.

The toy programming language Brainfuck (hereafter referred to as BF) consists of six instructions semantically equivalent to the six $\mathcal{P}''$ expressions and two additional instructions used for input and output. To show that we can compute any arbitrary, Turing-computable function, we build an interpreter for BF out of DMA gadgets. In order to implement the I/O instructions, we use DMA gadgets which interact directly with memory-mapped registers for a UART, thus demonstrating that DMA gadgets are resource-complete as well.

4.1 BF details

In this section, we give a brief overview of the BF programming language. Readers familiar with BF are encouraged to skip to the following section.

BF is a minimalistic programming language consisting of eight one-character instructions `+-><[],.`. All other characters act as a no-op. BF instructions operate on a tape divided into cells, much like the tape of a Turing machine. Each cell holds one of 256 values `00`, `01`, ..., `ff` and is initially empty. There is an implicit tape head, *head*, which points to the current cell on the tape. The eight instructions have the follow semantics.

- `+` increment the cell pointed to by *head*
- `-` decrement the cell pointed to by *head*
- `>` increment *head* to point to the next cell
- `<` decrement *head* to point to the previous cell
- `[` if the cell pointed to by *head* is *nonzero*, execute the next instruction; otherwise, jump to the instruction following the matching `]`
- `]` if the cell pointed to by *head* is *zero*, execute the next instruction; otherwise, jump to the instruction following the matching `[`
- `,` store input to the cell pointed to by *head*
- `.` output the cell pointed to by *head*

The increment and decrement instructions `+`/`-` operate modulo 256 and the loop instructions `[]` nest as expected.

Except for the loop instructions which behave as described above, BF instructions are executed sequentially. A program counter, *pc*, keeps track of the currently executing instruction. The program terminates when the *pc* moves past the last instruction. Figure 2 illustrates an example program that outputs the ASCII character D.

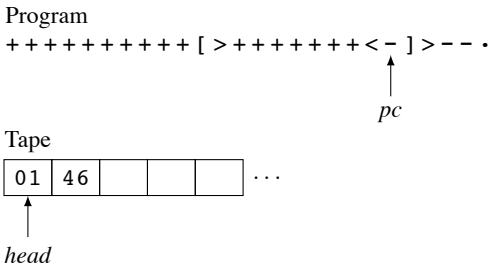


Figure 2: BF example. The program is in mid-execution with *head* currently pointing to cell 0 on the tape. The current instruction is a `-`, which decrements the byte pointed to by *head*, setting it to zero. Next, the right condition checks if the byte pointed to by *head* is zero; it is, so the program executes the next instruction which moves *head* one cell to the right. Cell 1 is then decremented twice, setting its value to `0x44`. Finally, the program outputs the ASCII character 'D' and halts.

4.2 Basic building blocks

We construct our BF interpreter (Section 4.3) using the basic building blocks described in this section. These building blocks can be used to implement a wide variety of gadgets beyond those needed for the BF interpreter. Some of these are described in Section 4.4.

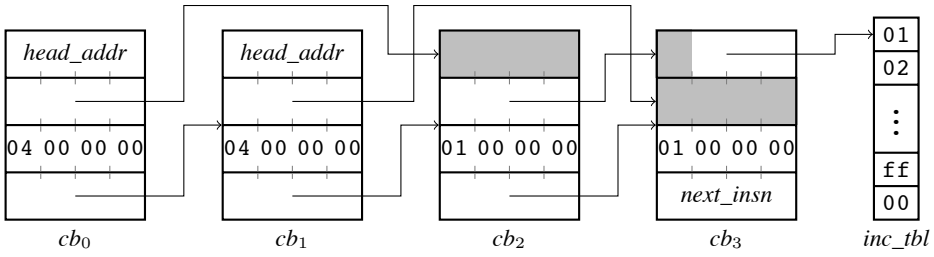


Figure 3: Increment gadget. The tape *head* is stored in a fixed location, *head_addr*. The first two control blocks copy *head* to cb_2's source and cb_3's destination, respectively. Then, cb_2 copies the cell pointed to by *head* into the least significant byte of cb_3's source which acts as an offset into the increment table. Finally, cb_3 stores the selected value back into the tape.

Unary functions. The basic operation of most gadgets involves mapping some input to output. Section 3 and Figure 1 illustrate the construction of 8-bit, unary functions. It is frequently useful to compute a function $g : \{0,1\}^8 \rightarrow \{0,1\}^{32}$. We can do this by constructing a table of the 32-bit output values and using a function $f : \{0,1\}^8 \rightarrow \{0, 4, \ldots, 252\}$ as an offset into the table. I.e., $g(n) = \mathit{table}[f(n)]$.

Variable dereferencing. In order to operate on data stored at a location pointed to by a pointer, we can use a control block to copy the value pointed to by the pointer into the source or destination fields of a subsequent control block. Figure 3 performs the operation

$$*\mathit{head} \leftarrow *\mathit{head} + 1$$

by first copying the 32-bit address pointed to by *head* into the source field of cb_2 and the destination field of cb_3.

Conditional goto. Conditional computation is achieved by writing the address of a control block to the next control block field of a trampoline control block. Which address is written is data-dependent. These conditional gotos can be used to implement if-then-else statements as well as while and do-while loops.

As a minor space-optimization, we implement conditionals using a 512-byte aligned, 512-byte address table consisting of 128 addresses paired with a 256-byte condition table. The mth conditional goto in the program is associated with a pair of addresses: the addresses of the control blocks corresponding to the false condition, $cb_{m,F}$, and the true condition, $cb_{m,T}$. The two addresses are stored 256-bytes apart in the address table. For example, if the address table is stored in memory at address `0x2000`, then $cb_{m,F}$ is stored at address `0x2000` $+ 4m$ and $cb_{m,T}$ is stored at address `0x2100` $+ 4m$. Each entry in the condition table stores either the second least significant byte of the address of the table or that value plus 256. Continuing the example, for each value n for which the condition is false, the nth entry in the condition table would be `0x20` and for each n for which the condition is true, the nth entry would be `0x21`.

By overwriting the second least significant byte of the source field of a control block — whose source is the address table — with the value from the conditional table, that control block can copy the address of either cb_F or cb_T into the next control block field of the trampoline. This is illustrated in Figure 4.

Switch. The switch building block branches to different control blocks depending on a data value. The value is used as an index into a 256-byte offset table. The entries in the offset table are the offsets into an address table which holds the addresses of the various control blocks associated with the switch cases.

Control blocks cb_1 through cb_3 in Figure 5 along with the dispatch and instruction tables are an example of a simple switch statement. ASCII values are mapped to their corresponding BF gadgets by using the dispatch table as the lookup table and the instruction table as the address table.

Memory-mapped I/O registers. Memory-mapped I/O registers are used to control hardware peripherals such as general purpose I/O (GPIO) pins, UARTs, I^2C or SPI buses, and yes, DMA engines. Interacting with such peripherals typically consists of looping, where we read a memory-mapped flag or status register over and over until a particular status is indicated (e.g., transmit buffer not full or receive buffer not empty), and then read or write a value to a memory-mapped data register. This building block is straight-forward to construct using conditionals for the loop and unary functions for the condition test.

4.3 BF interpreter gadgets

In this section, we use the basic building blocks defined in Section 4.2 to construct BF instruction and interpreter-specific gadgets. In addition to the gadgets described below, the BF interpreter requires a BF program to interpret, a region of memory to act as a tape, and three words at known addresses: a program counter, *pc*, a tape head *head*, and a loop counter, *lc*. The program counter and tape head behave as described in Section 4.1. The loop

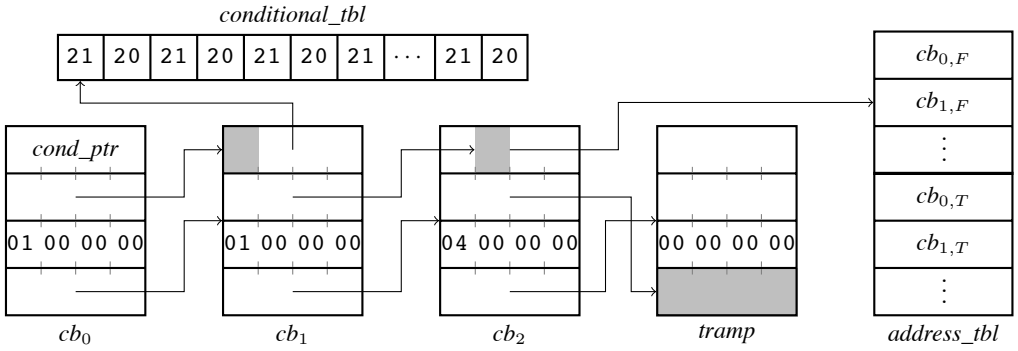


Figure 4: Conditional goto gadget. First, cb_0 copies the byte pointed to by *cond_ptr* into the least significant byte of cb_1's source to use as an index into the conditional table. Then, cb_1 copies the selected byte into the second least significant byte of cb_2's source. This byte selects which of cb_T or cb_F are copied into *tramp*'s next control block field. If the address table is at address `0x2000`, then if the byte pointed to by *cond_ptr* is even, then $cb_{1,T}$ will be the next control block executed. Otherwise, $cb_{1,F}$ will be.

counter is used to find matching brackets in the implementation of the loop instructions.

Dispatch gadget. This specific gadget dispatches a BF instruction. We use the switch building block with the *dispatch table* as the offset table and the *instruction table* as the address table. The dispatch gadget is shown in Figure 5.

Increment/decrement word gadgets. We implement generic 4-byte increment and decrement gadgets which take as input the address of the value to increment (resp. decrement) and the address of the next control block to execute when the operation is complete. These work by operating on a byte at a time. First, we increment (resp. decrement) the least significant byte of the 4-byte word. If the result is `00` (resp. `ff`), then we repeat with the second least significant byte, and so on. This is a straightforward application of unary functions, variable dereferencing, and conditionals.

Next instruction gadget. The next instruction gadget increments the *pc* by one using the increment word gadget and then jumps to the dispatch gadget.

Increment/decrement instruction gadgets. These gadgets increment or decrement the cell pointed to by *pc* using the generic increment and decrement word gadgets and then jump to the next instruction gadget.

Move right/left instruction gadgets. These gadgets move the head right or left by incrementing or decrementing *head* using the generic increment and decrement word gadgets and then jump to the next instruction gadget.

Loop instruction gadgets. The left and right loop instruction gadgets use the increment/decrement byte and word, conditional, and switch gadgets in its implementation. We use the switch gadget and define our lookup table, or *bracket table*, to contain an offset into two distinct address tables, or *scan right table* and *scan left table*, at the nth index, where n equals 0 or the ASCII byte representation of '`[`', or '`]`'. The scan right table assigns its indexes with the following control block addresses in order: scan right, increment loop counter, decrement loop counter, and quit. The scan left table inverts all operations.

We implement the left condition to first check whether the cell pointed to by `head` is zero. If it is, the gadget jumps to the next instruction gadget. Otherwise, it increments *lc* using the increment word gadget and scans right, incrementing and decrementing *lc* as brackets are encountered until $lc = 0$ at which point it jumps to the next instruction gadget.

The right condition is similar with a few exceptions. First, we jump to the next instruction if the cell pointed to by *head* is zero. At the start of scan left we decrement the *pc* using the decrement word gadget. The scan left table, as stated above, simply inverts all operations of the scan right table. This has the effect of scanning left until the matching bracket is found at which point it jumps to the next instruction gadget.

Input/output instruction gadgets. Using the memory-mapped I/O building block, the input and output instruction gadgets use the Pi's UART to receive a byte and store it in the cell pointed to by *head* or to transmit the byte in the cell.

4.4 Other gadgets

In previous sections, we demonstrated that DMA transfers are Turing- and resource-complete by building gadgets to interpret the BF programming language and interact with memory-mapped I/O registers. In this section we sketch the construction of a handful of building blocks that could be used to implement more efficient programs than those built using BF.

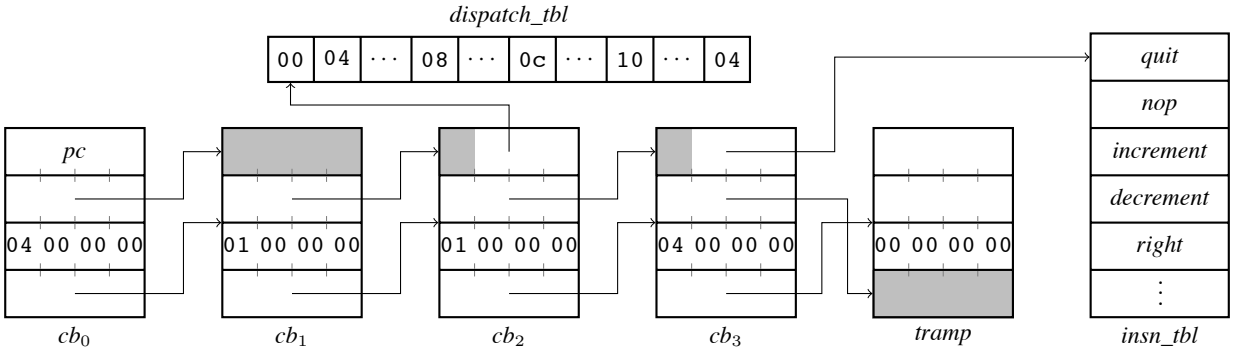


Figure 5: Dispatch gadget. The byte pointed to by the program counter is used as an offset into the dispatch table. The dispatch table contains the offset into the instruction table for the corresponding instruction. For example, the byte '+' has ASCII value 43; the 43rd entry of the dispatch table is 8; and the address of the increment gadget (see Figure 3) is stored at offset 8 in the instruction table. The control block cb_3 copies the corresponding entry from the instruction table into the next control block field of a trampoline control block.

Similar to the unary function building block, we can construct arbitrary binary functions $f : \{0,1\}^8 \times \{0,1\}^8 \rightarrow \{0,1\}^8$ by using a 64-kilobyte table, appropriately aligned such that concatenation of the left and right operands forms an offset into the table. Larger binary operations can be constructed by operating 8-bits at a time. For arithmetic operations such as addition, an additional table containing a carryout bit could be used to implement carries.

Relational operators can be implemented in much the same way or they can leverage a subtraction.

Finally, DMA-specific features can be used to easily implement functionality which would otherwise be more difficult to implement or be less performant. For one example, the DMA engine on the Raspberry Pi 2 is capable of zeroing regions of memory. Another example is the Pi is capable of performing moderately complex copying modes including nonconsecutive 2D copies. Lastly, as mentioned above, the DMA is usually responsible for communicating directly with hardware peripherals and DMA engines typically support gating the transfers between devices and memory using a variety of hardware signals. This would significantly simplify access to supported peripherals.

5 A DMA rootkit

The most common operating system used on the Raspberry Pi is a Debian-derived distribution called Raspbian which has a Linux kernel. Linux maintains a circular linked list of `task_structs` each of which holds information about a process. The head of the list, `init_task`, is an exported kernel symbol which is exposed using the *ksymtab* mechanism. Each `task_struct` contains a pointer to `cred` structure which contains various credentials, including the user ID (UID) of the process.

We implemented a DMA rootkit that first finds the address of `init_task` and then continually walks the linked list. For each process, the rootkit examines the process's UID. If the UID matches the target UID, then the UID is changed to 0, effectively giving the process super user privileges. Any processes with the target UID that are running are modified shortly after the rootkit is started. Similarly, any processes with the target UID that are started after the rootkit are quickly modified.

Unlike the DMA gadgets described in Section 4, for the rootkit we utilize the DMA engine's ability to perform a 2D transfer. This enables the rootkit to copy the `task_struct`'s next struct pointer and its `cred` pointer to a known location in memory given only the address of the next `task_struct` pointer.[4]

In more detail, starting with a four-byte kernel virtual address, *va* for a `task_struct`'s next struct pointer which is stored in a fixed location p, the rootkit first converts *va* to a bus address *ba*. Next, it loads the two words at *ba* and $ba + \Delta$ — where $ba + \Delta$ is the bus address of the `cred` pointer — to p and $p + 4$ using a 2D transfer with an appropriate stride constant Δ. After this transfer, location p contains a kernel virtual address for a `task_struct`'s next struct pointer and $p + 4$ contains a kernel virtual address for the current `task_struct`'s `cred` struct. The latter address is converted to a bus address, the UID is loaded, compared to the target UID, and on a match, 0 is written. In either case, the loop repeats.

Since the list is circular, the rootkit's logic is particularly simple. It consists of two DMA control blocks to get the address of `init_task` and an additional 18 to implement the loop, UID test, and UID setting.

[4]Lists in the Linux kernel contain pointers to the next element's *next element pointer* rather than to the beginning of the structure. In normal kernel code, this leads to an additional arithmetic instruction to recover a pointer to the structure.

6 Implementation

We implemented the BF interpreter described in Section 4.3 and the rootkit described in Section 5 on a Raspberry Pi 2. We were running the common Debian-based operating system, Raspbian. By default, Raspbian exposes the physical address space — including both the SDRAM main memory and the memory-mapped I/O registers — through the psuedo device file `/dev/mem`.

Our code is setuid root. It opens `/dev/mem`, maps pages of physical memory and I/O memory into the process's virtual address space, then closes the file and drops privileges. Next, it crafts DMA control blocks and tables as described above in an unused region of physical memory. Finally, a `run_dma()` function loads the address of the first control block in the DMA engine's memory-mapped I/O control block register which begins execution of the DMA program. All of our code is available at `https://github.com/stevecheckoway/rundma`.

For input and output, we connected an FTDI UART to USB cable to the UART pins on the Pi.

7 Related work

There are two, mostly disjoint, lines of research related to our work: the security of auxiliary processors inside computers, and unintended, Turing-complete computation.

Auxiliary processors. Security researchers have only recently begun examining the security of auxiliary processors and the firmware that runs on them. The most obvious example of an auxiliary processor is the GPU which uses DMA to transfer graphics data between the graphics card and main memory. Vasiliadis et al. [48] use the GPU to implement malware unpacking and runtime polymorphism in order to harden malware against detection. Ladakis et al. [29] use the GPU to build a key logger that monitors the system's keyboard buffer.

Duflot and Perez [17] examine the processor that runs on network interface cards (NICs). They exploit a vulnerability in the NIC's firmware to achieve arbitrary code execution and mount a DMA attack to add a backdoor in the kernel. Triulzi [44, 45] uses both the NIC and video card in concert to recover sensitive data in memory such as cryptographic keys. In follow-up work, Duflot et al. [18] construct an anomaly detection system that uses an IOMMU mechanism to limit access to main memory.

The IEEE 1394 FireWire specification allows the FireWire bus to communicate via DMA to minimize interrupts and buffer copies. Numerous researchers exploit this feature to access main memory directly [5, 9, 21, 34]. Kalenderidis and Collinson [26] exploit Intel Thunderbolt in a similar fashion.

The Intel Management Engine (ME) is a microcontroller embedded in the Intel chip set with a separate NIC, DMA access to main memory, and remote out-of-band management technology called Intel Active Management Technology (AMT). Stewin and Bystrov [42] use the ME to build a DMA key logger, and Tereshkin and Wojtczuk [43] use AMT to construct a "Ring −3" rootkit. Similarly, Farmer [19] and Moore [32] examine vulnerabilities in the Intelligent Platform Management Interface (IPMI).

Other exploitable auxiliary processors include laptop batteries [31] and webcams [12].

Unintended computation. The ability to craft input data to drive programs in the target system has been discussed by the hacker community as far back as Aleph One's seminal article on buffer overflows [1]. Return-to-libc [39], Krahmer's borrowed code chunks technique [28], and return-oriented programming (ROP) [37] represent an evolution of exploitation techniques leading to Turing-complete computation built by borrowing existing program code.

ROP was first introduced by Shacham [37] as a technique to perform arbitrary, Turing-complete computation by executing a string of gadgets: short sequences of instructions, linked together by an "update-load-branch" mechanism [15], that exist within the program or linked library. ROP has since been extended to various architectures [13, 14, 20, 27, 30]. More recent work has focused on the automation of each step in the technique [6, 23, 35, 36]. For example, Bittau et al. [6] explores the limits of ROP by crafting an exploit without possessing the target's binary.

Turing-complete gadget sets need not be comprised of misappropriated CPU instructions. Indeed, parsers for complex file and record formats can be abused to provide Turing-complete computation. Oakley and Bratus [33] uses the Debugging With Attribute Records Format (DWARF) to perform arbitrary computation with the DWARF bytecode. Shapiro et al. [38] use the ELF loader mechanism to effect computation.

The prior work most similar to ours combines specialized hardware and unintended computation. Bangert et al. [4] demonstrate a Turing-complete execution environment using the IA32 arcitecture's page fault handling mechanism. Neither the page fault handling hardware nor the DMA hardware was designed with computation in mind; however, computation emerges from the hardware's complexity.

8 Conclusions

In this work, we have shown that DMA engines can be used to perform Turing-complete computation even though it is not their intended function. In particular, we have crafted DMA Turing- and resource-complete gadget sets that we used to build an interpreter for BF. In

addition, we built a DMA rootkit to performs privilege escalation for targeted programs.

Although we are the first to build malware entirely out of DMA transfers, we are not the first to consider the capabilities DMA provides to auxiliary processors running in the system (see Section 2). Indeed, researchers have considered various countermeasures to such DMA malware. These countermeasures are applicable to our work as well. Example countermeasures include using the input/out memory management unit (IOMMU) [41], peripheral firmware load-time integrity [42, 46], anomaly detection systems [18], and bus agent runtime monitors (BARMs) [41].

Several of these defenses have been found lacking. Researchers have noted that peripheral firmware load-time integrity is inadequate because it does not provide runtime integrity [18, 42]. Stewin and Bystrov [42] further describes the IOMMU as lacking because it can be configured improperly, and it cannot be applied if there are memory access policy conflicts.

Given the current lack of strong defenses against DMA abuse and the ability of DMA to do both Turing-complete and resource-complete computation, it is clear that more work on secure defenses is needed. (ノ °□°) ノ ⌒ ┻━┻

Acknowledgments

We thank our shepherd Travis Goodspeed, Cynthia Taylor and the anonymous reviewers for their time and invaluable comments.

References

[1] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), August 1996. Online: `http://www.phrack.org/issues.html?issue=49&id=14`.

[2] *CoreLinkTM DMA Controller DMA-330*. ARM, July 2010. Online: `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0424c/DDI0424C_dma330_r1p1_trm.pdf`.

[3] Malcolm Atkinson, Françis Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The object-oriented database system manifesto. In Won Kim, Jean-Marie Nicolas, and Shojiro Nishio, editors, *Proceedings of DOOD 1989*, pages 223–40. Elsevier, December 1989. Online: `https://www.cs.cmu.edu/~clamen/OODBMS/Manifesto/Manifesto.PS.gz`.

[4] Julian Bangert, Sergey Bratus, Rebecca Shapiro, and Sean W. Smith. Page-fault weird machine: Lessons in instruction-less computation. In *Proceedings of WOOT 2013*. USENIX, August 2013. Online: `https://www.usenix.org/conference/woot13/workshop-program/presentation/Bangert`.

[5] Michael Becher, Maximillian Dornseif, and Christian N. Klein. FireWire: all your memory are belong to us. Presented at CanSecWest 2005, May 2005. Online: `https://cansecwest.com/core05/2005-firewire-cansecwest.pdf`.

[6] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. Hacking blind. In *Proceedings of IEEE Security and Privacy ("Oakland") 2014*, pages 227–42, May 2014.

[7] Corrado Böhm. On a family of Turing machines and the related programming language. *International Computation Centre Bulletin*, 3:187–94, July 1964.

[8] Corrado Böhm and Giuseppe Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–71, May 1966.

[9] Adam Boileau. Hit by a bus: Physical access attacks with firewire, September 2006. Online: `http://www.security-assessment.com/files/presentations/ab_firewire_rux2k6-final.pdf`.

[10] Sergey Bratus, Trey Darley, Michael Locasto, Meredith L. Patterson, Rebecca Shapiro, and Anna Shubina. Beyond planted bugs in "trusting trust": The input-processing frontier. *Security Privacy, IEEE*, 12(1):83–87, January 2014.

[11] *BCM2835 ARM Peripherals*. Broadcom Corporation, February 2012. Online: `https://www.raspberrypi.org/wp-content/uploads/2012/02/BCM2835-ARM-Peripherals.pdf`.

[12] Matthew Brocker and Stephen Checkoway. iSeeYou: Disabling the MacBook webcam indicator LED. In *Proceedings of USENIX Security 2014)*, pages 337–52. USENIX Association, August 2014. Online: `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/brocker`.

[13] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In Paul Syverson and Somesh Jha, editors, *Proceedings of CCS 2008*, pages 27–38. ACM Press, October 2008.

[14] Stephen Checkoway, Ariel J. Feldman, Brian Kantor, J. Alex Halderman, Edward W. Felten, and Hovav Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the AVC Advantage. In David Jefferson, Joseph Lorenzo Hall, and Tal Moran, editors, *Proceedings of EVT/WOTE 2009*. USENIX/ACCURATE/IAVoSS, August 2009.

[15] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In Angelos Keromytis and Vitaly Shmatikov, editors, *Proceedings of CCS 2010*, pages 559–72. ACM Press, October 2010. Online: `https://www.cs.jhu.edu/~s/papers/noret_ccs2010.html`.

[16] Stephen Dolan. `mov` is Turing-complete. July 2013. Online: `http://www.cl.cam.ac.uk/~sd601/papers/mov.pdf`.

[17] Loïc Duflot and Yves-Alexis Perez. Can you still trust your network card? Presented at CanSecWest 2010, March 2010. Online: `http://www.ssi.gouv.fr/IMG/pdf/csw-trustnetworkcard.pdf`.

[18] Loïc Duflot, Yves-Alexis Perez, and Benjamin Morin. What if you can't trust your network card? In Robin Sommer, Davide Balzarotti, and Gregor Maier, editors, *Proceedings of RAID 2011*, pages 378–397. Springer, September 2011. Online: `http://www.ssi.gouv.fr/IMG/pdf/paper.pdf`.

[19] Dan Farmer. IPMI: Freight train to hell, January 2013. Online: `http://fish2.com/ipmi/itrain.pdf`.

[20] Aurélien Francillon and Claude Castelluccia. Code injection attacks on Harvard-architecture devices. In Paul Syverson and Somesh Jha, editors, *Proceedings of CCS 2008*, pages 15–26. ACM Press, October 2008.

[21] Todd Garrison. Firewire attacks against Mac OS Lion FileVault 2 encryption. September 2011. Online: `http://www.frameloss.org/2011/09/18/firewire-attacks-against-mac-os-lion-filevault-2-encryption/`.

[22] Dina Q. Goldin, Scott A. Smolka, Paul C. Attie, and Elaine L. Sonderegger. Turing machines, transition systems, and interaction. *Information and Computation*, 194(2): 101–28, November 2004. Online: `http://www.sciencedirect.com/science/article/pii/S0890540104001257`.

[23] Ralf Hund, Thorsten Holz, and Felix Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In Fabian Monrose, editor, *Proceedings of USENIX Security 2009*, pages 383–98. USENIX, August 2009.

[24] *Intel Platform Controller Hub EG20T: Datasheet*. Intel, July 2012. Online: `http://www.intel.com/content/www/us/en/intelligent-systems/queens-bay/platform-controller-hub-eg20t-datasheet.html`.

[25] Intel Server Platform Group. Intel QuickData technology software guide for Linux, May 2008. Online: `http://www.intel.com/content/dam/doc/white-paper/quickdata-technology-software-guide-for-linux-paper.pdf`.

[26] Loukas Kalenderidis and Sam Collinson. Thunderbolts and lightning, very very frightening. Presented at SyScan 2014, May 2014. Online: `https://www.youtube.com/watch?v=0FoVmBOdbhg`.

[27] Tim Kornau. Return oriented programming for the ARM architecture. `http://zynamics.com/downloads/kornau-tim--diplomarbeit--rop.pdf`, 2009. Master thesis, Ruhr-University Bochum, Germany.

[28] Sebastian Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique, September 2005. `http://www.suse.de/~krahmer/no-nx.pdf`.

[29] Evangelos Ladakis, Lazaros Koromilas, Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. You can type, but you can't hide: A stealthy GPU-based keylogger. In Thorsten Holz and Sotiris Ioannidis, editors, *Proceedings of EuroSec 2013*. ACM, April 2013. Online: `http://www.cs.columbia.edu/~mikepo/papers/gpukeylogger.eurosec13.pdf`.

[30] Felix Lidner. Developments in Cisco IOS forensics. CONFidence 2.0. `http://www.recurity-labs.com/content/pub/FX_Router_Exploitation.pdf`, November 2009.

[31] Charlie Miller. Battery firmware hacking: Inside the innards of a smart battery. Presented at Black Hat Briefings, August 2011. Online:

http://media.blackhat.com/bh-us-11/Miller/BH_US_11_Miller_Battery_Firmware_Public_WP.pdf.

[32] HD Moore. A penetration tester's guide to IPMI and BMCs. July 2013. Online: https://community.rapid7.com/community/metasploit/blog/2013/07/02/a-penetration-testers-guide-to-ipmi.

[33] James Oakley and Sergey Bratus. Exploiting the hard-working DWARF: Trojan and exploit techniques with no native executable code. In *Proceedings WOOT 2011*. USENIX Association, August 2011. Online: http://dl.acm.org/citation.cfm?id=2028052.2028063.

[34] Peter Panholzer. Physical security attacks on Windows Vista, March 2008. Online: https://www.sec-consult.com/fxdata/seccons/prod/downloads/vista_physical_attacks.pdf.

[35] Ryan Roemer. Finding the bad in good code: Automated return-oriented programming exploit discovery. Master's thesis, UC San Diego, March 2009. Online: https://cseweb.ucsd.edu/~rroemer/doc/thesis.pdf.

[36] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In David Wagner, editor, *Proceedings of USENIX Security 2011*, August 2011. Online: http://users.ece.cmu.edu/~ejschwar/papers/usenix11.pdf.

[37] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Sabrina De Capitani di Vimercati and Paul Syverson, editors, *Proceedings of CCS 2007*, pages 552–61. ACM Press, October 2007.

[38] Rebecca Shapiro, Sergey Bratus, and Sean W. Smith. "Weird machines" in ELF: A spotlight on the underappreciated metadata. In *Proceedings of WOOT 2013*. USENIX, August 2013. Online: https://www.usenix.org/conference/woot13/workshop-program/presentation/Shapiro.

[39] Solar Designer. Getting around non-executable stack (and fix). Bugtraq, August 1997. Online: http://seclists.org/bugtraq/1997/Aug/0063.html.

[40] Vaidyanathan Srinivasan, Anand K. Santhanam, and Madhavan Srinivasan. Cell broadband engine processor dma engines, part 1: The little engines that move data. December 2005. Online: http://www.ibm.com/developerworks/library/pa-celldmas/.

[41] Patrick Stewin. A primitive for revealing stealthy peripheral-based attacks on the computing platform's main memory. In *Proceedings of RAID 2013*, pages 1–20, October 2013. Online: http://link.springer.com/chapter/10.1007%2F978-3-642-41284-4_1.

[42] Patrick Stewin and Iurii Bystrov. Understanding dma malware. In *Proceedings of DIMVA 2012*, pages 21–41. Springer-Verlag, July 2012. Online: http://dx.doi.org/10.1007/978-3-642-37300-8_2.

[43] Alexander Tereshkin and Rafal Wojtczuk. Introducing ring -3 rootkits. Presented at Black Hat Briefings, July 2009. Online: http://www.blackhat.com/presentations/bh-usa-09/TERESHKIN/BHUSA09-Tereshkin-Ring3Rootkit-SLIDES.pdf.

[44] Arrigo Triulzi. Project Maux Mk.II, November 2008. Online: http://www.alchemistowl.org/arrigo/Papers/Arrigo-Triulzi-PACSEC08-Project-Maux-II.pdf.

[45] Arrigo Triulzi. The Jedi packet trick takes over the Deathstar, 2010. Online: http://www.alchemistowl.org/arrigo/Papers/Arrigo-Triulzi-CANSEC10-Project-Maux-III.pdf.

[46] Trusted Computing Group. TCG PC client specific implementation specification for conventional BIOS, July 2005. Online: http://www.trustedcomputinggroup.org/files/temp/64505409-1D09-3519-AD5C611FAD3F799B/PCClientImplementationforBIOS.pdf.

[47] Julien Vanegue. The weird machines in proof-carrying code. In *Proceedings of SPW 2014*, pages 209–13. IEEE Computer Society, May 2014. Online: http://www.ieee-security.org/TC/SPW2014/papers/5103a209.PDF.

[48] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. GPU-assisted malware. In Jean-Yves Marion, Noam Rathaus, and Cliff Zhou, editors, *Proceedings of MALWARE 2010*, pages 1–6. IEEE Computer Society, October 2010. Online: http://dcs.ics.forth.gr/Activities/papers/gpumalware.malware10.pdf.