This paper appeared at the First International Conference on Sensor Networks and Applications, Nov. 4-Nov. 6, 2009

Architectural Support for Securing Sensor Networks Against Remote Attacks

Mohammed I. Al-Saleh
Computer Science Dept.
Univ. of New Mexico
Albuquerque, NM 87131

Patrick G. Bridges
Computer Science Dept.
Univ. of New Mexico
Albuquerque, NM 87131

Jedidiah R. Crandall
Computer Science Dept.
Univ. of New Mexico
Albuquerque, NM 87131

Abstract

Sensor network devices are no less vulnerable to remote attacks, such as malicious worms, than their general purpose computer counterparts, and are presented with unique threats because of the hostile environments sensors are placed in. It is well known that sensor devices place challenging constraints on any attempt to secure them against these attacks, including small performance and power budgets, infrequent patch updates, and long service lives. However, in this paper we demonstrate that security *can* be built into sensor devices "from the ground up."

In this paper we apply dynamic information flow tracking (DIFT) to sensor devices, where network data is tagged as untrusted and then these tags propagate throughout the system. Our results demonstrate that minor hardware modifications to sensor devices can provide sufficient security guarantees against remote control data attacks. To make these guarantees we address all five dynamic information flow dependency types (copy, computation, load-address, store-address, and control), whereas DIFT schemes for general purpose computers are empirically only able to address the first two. Rigorous testing of eight applications shows that no modifications to existing operating systems, compilers, applications, or binaries is necessary.

1 Introduction

Securing sensor networks before they are deployed is critical. Today, these systems are being deployed in military and medical applications, yet are still built on top of architectures with simple memory models using the C language, or variants of C such as nesC [16]. Memory corruption vulnerabilities can lead to worm attacks [17] and other remote intrusions that allow adversaries to completely take control of the network. Even Harvard architectures, where instructions and data are kept in separate memories, are susceptible to these kinds of attacks [15].

The *dynamic information flow tracking* (DIFT) scheme that we present in this paper marks all data that is read from the network (typically from a radio device) as untrusted using a *tag bit*, sometimes also called a *taint bit* (throughout this paper we use the terms untrusted and tainted interchangeably). Tags are propagated throughout the system by the architecture, with no need for modification to the program binaries. No untrusted data can be used as the target address for a control flow transfer such as a jump, call, or return. This makes attacks that overwrite *control data*, such as buffer overflows and related attacks, impossible. Control data includes return addresses, function pointers, the bases and offsets of linked library functions, and more. Thus the architecture we present stops any attack that overwrites control data and hijacks control flow to take control of a network sensor.

The key insight behind our work is that sensor network applications have very regular and predictable patterns of memory management, when compared to general purpose systems. DIFT [29, 8, 9, 22, 6, 25] has been proposed as an attractive solution to memory corruption vulnerabilities because, if all possible information flow dependencies are addressed, DIFT schemes can secure commodity code with very minor modifications to the hardware. Unfortunately, memory management in general purpose systems is very complex, so that in DIFT schemes for these systems the load- and store-address dependencies, where information can flow based on the address with which a piece of data is loaded or stored, cannot be tainted without rendering the system unusable due to false positives [28]. By not tracking load- and store-address dependencies, current DIFT schemes are open to attacks that use multiple levels of pointer indirection [7, 10, 23]. One study [9] even found that existing attacks not designed to subvert DIFT can evade detection and reduce existing DIFT schemes to the same level of security as NX (non-executable) pages, which has been shown to not stop attacks even when combined with various other limitations on the attacker [27, 4].

What we demonstrate in this paper, however, is that DIFT schemes for sensor networks need not be as limited, and can provide real security guarantees without requiring any modifications to existing application source code, program binaries, compilers, or operating systems and only minor modifications to the hardware. We have implemented a DIFT scheme on the MICA2 platform that is secure against all control data attacks. Through rigorous testing of eight applications, we demonstrate that tracking load/store address dependencies does not break existing applications nor require any modifications to the application source code, program binary, compiler, or operating system. Then through dynamic testing and manual code inspection of these eight applications we determine that all control dependencies in the applications are benign. Thus, whereas existing DIFT schemes have **not** provided satisfactory security guarantees against remote control flow hijacking attacks on general purpose systems, our scheme demonstrates that it is possible to do so for sensor network applications.

The rest of this paper is organized as follows. Section 2 discusses the five types of data dependencies that DIFT schemes must track and gives some basic definitions that are used throughout the rest of the paper. Our implementation is described in Section 3 followed by the results of testing and manual analysis of the applications in Section 4. In Section 5 we discuss current limitations and how they can be addressed in future work, and this is followed by related works and the conclusion.

2 Definitions

In this section we discuss how information can flow, using Suh *et al.*'s [29] categorization of information flow dependencies into five types: copy dependency, computation dependency, load-address dependency, store-address dependency, and control dependency. This section is meant to make it clear what is and is not secure with regards to how a particular DIFT scheme tracks these dependencies.

A *remote control flow hijacking attack* is when an attacker sends data to a remote computer or sensor and causes the control flow of a service to be diverted to the code of the attacker. Without user intervention, virtually all remote attacks such as worms are typically of this form. Also note that the code of the attacker need not be machine instructions, return-into-libc [21] or more advanced attacks [27, 4] are possible.

Control data attacks, such as buffer overflows, are the most common form of remote control flow hijacking attacks, in which control data such as return addresses, function pointers, and jump targets are overwritten by the attacker so that low-level control flow is diverted.

A *copy dependency* occurs when data is copied from one storage object to another, such as register-to-register, memory-to-register, or register-to-memory. When this happens, the tag bit of the destination should be set to the value of the tag of the source. A *computation dependency* occurs when an operation such as an arithmetic operation or bit manipulation is performed on one or more source registers, and the result is written to a destination register. In this case, most DIFT schemes will apply a low-water-mark policy, where the destination is tainted as untrusted if *any* of the source registers are tainted.

Another way for information to flow in a system is through dependencies on the memory addresses used for loads and stores. For example, consider the C code in Figure 1 for converting data read over the network (`InputArray`) from one format into another using a lookup table (`LookupTable`) and storing the result in another buffer (`ConvertedArray`).

Each byte of input from the network is used as the offset in an address for looking up converted values in the lookup table. It is important to taint the value that gets loaded if the calculated address is tainted, because an attacker could send data over the network that would no longer be tainted after conversion but would still be data in the system that had come from the attacker. An example of how this can lead to attacks not being detected by DIFT schemes is the Code Red worm. Most existing DIFT schemes ignore these *load-address dependencies*, so the UNICODE decoding routine that caused the buffer overflow vulnerability exploited by Code Red also removes the taint mark of the network input before overwriting the structured exception handler on the stack. Thus control flow is hijacked, but existing DIFT schemes will not detect the attack. Minos [8, 9] tracks load-address dependencies for 8- and 16-bit loads and is able to stop the Code Red buffer overflow, but does not track 32-bit load-address dependencies. No existing DIFT scheme has successfully addressed load-address dependencies in the general case.

Store-address dependencies are also important. Without tracking these dependencies by tainting the stored value for any store where the address is tainted, indirect attacks are possible. In addition to the possibility that an attacker could use multiple levels of pointer indirection to subvert a DIFT scheme [7, 10, 23, 28], testing has shown that even attacks not designed specifically for DIFT schemes can hijack control flow undetected due to store-address dependencies [9].

No existing DIFT scheme for general purpose computers can sufficiently track information flow through load- and store-address dependencies. The reason is that, even for normal systems that are not vulnerable and not under attack, information flow from the network into heap and stack pointers is common. Using heuristics to determine which address dependencies are acceptable and which constitute avenues of attack is infeasible because of the complexity of dynamic memory allocation in general purpose applications. Fortunately, our results show that, due to the relatively simple and regular memory management of sensor devices, all load- and store-address dependencies can be tracked in sensor network applications.

A *control dependency* occurs when the value of one piece of data affects the execution path of the program, which in-turn affects the value of another piece of data. As an example, consider the following C code:

```
if (x == 1)
     y = 1;
else
     y = 0;
```

```
for (Loop = 0; Loop < Size; Loop++)
     ConvertedArray[Loop] = LookupTable[InputArray[Loop]];
```

Figure 1: **An example of a load-address dependency.**

Clearly, information flows from x to y but there is no explicit assignment and no address dependencies between these two variables. In confidentiality settings, this is referred to as an *implicit flow*. Methods exist for tracking implicit flows dynamically, including temporarily tainting the program counter on conditional control flow transfers [13, 12] and annotating the machine code with label transformations [30], but these methods are not practical without modifying the compiler or performing static analysis and binary rewriting.

Where control dependencies become a security problem for DIFT is when it is possible for the attacker to *launder* the taint bit, so that a value under their control is no longer tainted. An example of this is the following:

```
UntaintedData = 0;
while ((TaintedData--) != 0)
     UntaintedData++;
```

After the above code is executed, the value of `TaintedData` will be copied into `UntaintedData` but the latter will not be tainted in the process, so that data from the attacker is now marked as trusted. An example of a *benign* control dependency that cannot be exploited by the attacker for control data attacks is the following:

```
if (TaintedData == 0)
     HandleRequestTypeA();
else if (TaintedData == 1)
     HandleRequestTypeB();
else
     HandleRequestTypeC();
```

In this case, a conditional control flow transfer is made based on untrusted data from the network, but an attacker can only choose which path to take by sending a different kind of request over the network. Thus the attacker has only as much control over the control flow of the program as a regular client.

Another difference between confidentiality and integrity settings is that typically the former assumes that the code being executed is chosen arbitrarily by the attacker, whereas in most integrity settings like DIFT the source code of the trusted application being executed is assumed not to maliciously attempt to untaint data. Nonetheless, for general purpose systems it has been shown that regular code for operations such as `printf()`-style format string conversion and other format conversions can lead to attacker-controlled data becoming untainted and leading to attacks that evade DIFT schemes [9]. In Section 4 we demonstrate that no similar problems exist in the eight sensor network applications that we tested. Here we define laundering as follows.

A control dependency that causes information to flow from a tainted variable x in state s to an untainted variable y in state t is benign if and only if $I(x_s; y_t) << H(x_s)$, *i.e.*, the mutual information between the two variables after the operation is much less than the entropy of the tainted variable. Thus an attacker cannot cause her inputs to be untainted without a consequent loss of entropy, which takes away her ability to arbitrarily corrupt data. A control dependency that is not benign is said to be laundering.

Another important distinction is that of *first-order* control dependencies *vs.* *higher-order* dependencies. When tainted data is used in a conditional control flow transfer this causes a first-order dependency in which information can flow into untainted values. If a first-order dependency were not benign then it would be necessary to taint additional data that could lead to additional higher-order dependencies, where conditional control flow transfers that would not have depended on tainted data now do. If all first-order dependencies are benign, then there is no possibility for high-entropy, laundering second-order dependencies. Thus in Section 4 we need only to consider first-order dependencies, because all of these are found to be benign.

Existing DIFT schemes [8, 29, 22, 6, 25] do not consider control dependencies. In our scheme for sensor network devices, we have not added any special support for control dependencies because testing and manual analysis revealed that no control dependencies in the applications we tested are laundering, *i.e.*, they are all benign.

3 Implementation

In this section, we describe our emulated DIFT scheme for the MICA2 platform, and the method we used to locate first-order control dependencies.

3.1 Emulated DIFT scheme

We modified ATEMU (ATmel EMUlator) [24] to initiate, propagate, and check tags. ATEMU is an instruction-level emulator that emulates the AVR processor ATmega 128L that is part of the MICA2 platform. ATEMU also implements all of the needed peripheral devices, *e.g.*, CC1000 radio chip, ADC (analog to digital converter), LEDs (light-emitting diodes), and SPI (serial peripheral interface), as plug-in libraries. A bit-per-byte mark extension, or tag, is used for the registers and the SRAM as an indication if that location is tainted or not. All bytes in the SRAM or registers have an associated tag. Our tainting scheme can be divided into three separate functionalities: *taint source*, *taint propagation*, and *taint check*.

Application	Main Functionality	Setup	Pass
AntiTheft	Detects and reports theft	Three nodes, a root binary and two with same binary	Yes
BaseStation	Bridges serial and radio links	Two nodes, different binaries	Yes
RadioSenseToLeds	Samples default sensor and broadcasts readings in an AM packet	Two nodes, same binary	Yes
RadioCountToLed	Maintains a 4Hz counter and broadcasts its value in an AM packet when updated	Two nodes, same binary	Yes
Oscilloscope	Samples the default sensor and broadcasts a message every 10 readings	Two nodes, same binary	Yes
BlinkToRadio	Increments a counter and sends a radio message whenever a timer fires	Two nodes, same binary	Yes
TestAM	Tests radio active messages	Two nodes, same binary	Yes
Blink	Blinks the 3 mote LEDs	One node	Yes

Table 1: **Application functionalities and test results.**

Taint source: When the sensor device boots, all data is considered trusted. All data that is received over the network, which is the taint source in our scheme, is tagged as untrusted.

Taint propagation: Our taint propagation scheme correctly handles copy, computation, load-address, and store-address dependencies. Existing DIFT schemes only consider the first two, although sometimes the latter two are handled in a limited way. For single-operand instructions such as bit rotations (*e.g.*, ROR) the register simply retains its tag. For instructions with multiple operands (*e.g.*, ADD, SUB, AND) we taint the destination register if *any* source register is tainted. This is known as a low-water-mark policy because any data that depends on tainted data is itself tainted. Register-to-register copies of data also copy the tag. PUSH and POP propagate the tag of the register or memory location that is pushed or popped onto or off of the stack. Load (LD) and store (ST) operations apply the low-water-mark policy not only to the source data that is loaded or stored, but also to the address used for the load or store (which is always stored in the X, Y, or Z registers). Thus if the source of a load or store is untainted but the address used is tainted, the destination will always be tainted so that load-address and store-address dependencies are correctly handled. Conditional control flow transfers do not taint the program counter because all sensor network applications that we tested had only benign control dependencies.

Taint check: Our scheme checks the tags of target addresses (whether they are explicit operands or unstacked inputs) for all control flow transfers. This includes RET (subroutine return), RETI (interrupt return), ICALL (indirect call), EICALL (extended indirect call), IJMP (indirect jump), and EIJMP (extended indirect jump). If the target address is tainted, then this indicates an attempted attack and the node is reset.

3.2 Locating first-order control dependencies

The AVR instruction set has conditional control flow transfers that work in a way that is similar to those of the Pentium instruction set architecture. Some instructions, *e.g.*, CMP (compare) and arithmetic expressions, set various flags to indicate properties of their arguments such as equality, sign or less-than, overflow, *etc.* Conditional control flow transfer instructions such as JE (jump if equal) check these flags and either jump to a hardcoded jump target or simply move on to the next instruction depending on the value of the flag.

To locate all first-order control dependencies, we modified the emulator as follows. When an instruction sets a flag, the tag of the flag is set using a low-water-mark policy based on the operands of the instruction. When a conditional control flow occurs based on a flag that is tainted, and therefore based on untrusted data, the program counter for the location of that conditional control flow transfer is recorded. Using a variety of tools it is possible to locate the corresponding nesC code.

4 Results

Our experimental methodology sought to answer three key questions with respect to applying DIFT to sensor network devices and applications:

1. Is it possible to track load- and store-address dependencies, in addition to the copy and calculation dependencies tracked by conventional DIFT systems, without modifying the application, compiler, or operating system in any way? Our results demonstrate that the answer is yes.
2. Is it necessary to add special taint propagation rules to handle control dependencies? Our results indicate that the answer is no, since all first-order control dependencies in the applications we analyzed are benign.
3. Does our scheme detect the attacks that it is designed to detect, including attacks with load- and store-address dependencies? Our results confirm that it does detect control data attacks.

To answer the first question we tested all applications that we were able to obtain that could be executed on the ATEMU emulator. While our hardware DIFT scheme can secure any application for any operating system on

```
//The receiver code
void overflow(uint8_t *rcm)
{
  char buf[3];
  uint16_t *ptr[4];
  strcpy(buf, rcm);
  // copy receive()'s stack pointer into caller()'s stack pointer
  *ptr[0] = *ptr[1];
  // copy receive()'s ret address into caller()'s ret address
  *ptr[2] = *ptr[3];
}
void caller(uint8_t *rcm)
{
  ...
  overflow(rcm);
  ...
}
event message_t* Receive.receive(message_t* bufptr, void* payload, uint8_t len)
{
  uint8_t *rcm = (uint8_t*) payload;
  ...
  caller(rcm);
  ...
}
```

Figure 2: **Receiver code containing a vulnerability with load- and store-address dependencies.**

```
//The sender code
payload = (char *) call Packet.getPayload(&packet, NULL);
strcpy(payload, "\x22\x33\44\xd9\x10\xe0\x10\xd7\x10\xde\x10");
call AMSend.send(AM_BROADCAST_ADDR, &packet, 11);
```

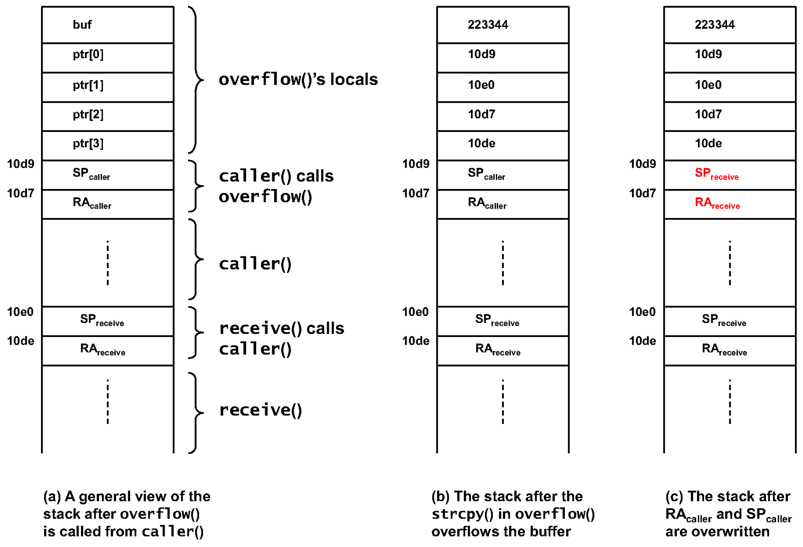
buf
ptr[0]
ptr[1]
ptr[2]
ptr[3]
overflow()'s locals
10d9
SP_caller
10d7
RA_caller
caller() calls overflow()
caller()
10e0
SP_receive
10de
RA_receive
receive() calls caller()
receive()
223344
10d9
10e0
10d7
10de
SP_receive
RA_receive
(a) A general view of the stack after overflow() is called from caller()
(b) The stack after the strcpy() in overflow() overflows the buffer
(c) The stack after RA_caller and SP_caller are overwritten

Figure 4: **Stack trace during the attack.**

the MICA2 platform without any software modifications, we were only able to run TinyOS [20] applications on the ATEMU emulator due to limitations of the emulator (other OSes either are not available for the MICA2 platform or do not work with the ATEMU framework). TinyOS is by far the most commonly used operating system for sensor network applications. The eight applications that we tested use a wide variety of TinyOS' functionalities, as shown in Table 1. Table 1 also shows that rigorous testing of all of the applications' functionalities yielded no false alerts. The third column describes the network setup for the application, and the fourth column indicates that all applications passed rigorous testing without producing any false alerts.

We answered the second question through a combination of testing the applications to locate the first-order control dependencies and then manual analysis of the source code for these control dependencies to determine if they are laundering or benign. We identified thirty first-order control dependencies, which we do not list here due to space limitations. We determined first-order dependencies by printing an alert with the program counter every time a conditional control flow transfer was made using a test of tainted data. All first-order control dependencies in all applications were determined to be benign through manual code inspection. This means that, for all applications tested, no special mechanism is needed to propagate tags for control dependencies and thus no higher-order control dependencies need be considered.

The third question is more of a claim to be confirmed than a question, since tracking all possible data dependencies will stop any attack that overwrites control data with untrusted data from the network. Whereas existing DIFT schemes measure “false positives” and “false negatives,” our scheme for sensor network nodes makes satisfactory security guarantees that ensure that no remote control-flow hijacking attack based on memory corruption of control data can succeed. Nonetheless, we tested various attacks against our scheme to confirm that an alert was indeed raised in all cases. The attacks we tested include all attacks described by Francillon *et al.* [15] and Gu *et al.* [17], as well as an attack we developed to test the tracking of load- and store-address dependencies.

The vulnerability illustrated in Figures 2 and 3 demonstrate how load- and store-address dependencies can be exploited to arbitrarily copy untainted data from one place to another. No previous DIFT schemes can stop an attack of this kind. Basically, four pointers on the stack are overwritten by the attack, then used as the source and destination addresses of two copy operations. Any DIFT scheme that does not consider address dependencies will copy data from an arbitrary source address to an arbitrary destination, both chosen by the attacker, without tainting the result. This can lead to attacks not detected by these DIFT schemes [7, 10, 23]. We tested our DIFT scheme with such an attack based on the vulnerability in Figure 2 and an alert was raised, demonstrating that our scheme is secure with respect to address dependencies.

Figure 4 (a) shows the layout of the stack on the receiver side after receiving the message coming from the sender and then executing the following sequence of function calls: `receive()` calls `caller()`, and then `caller()` calls `overflow()`. The stack shows how the local variables and activation records (including return addresses and stored stack pointers) are laid out on the stack. Figure 4 (b) shows the exact values that will occupy the local variables of `overflow()` right after line 50 is executed. Note that the `strcpy()` in line 50 has overflowed the buffer `buf` with more data than it has room to store. The extra data overflows the four pointers right after the buffer `buf`. Figure 4 (c) shows the stack contents after the execution of lines 70 and 90. The stack pointer of `caller()` has been changed to that of `receive()`, and the return address of `caller()` has been changed to that of `receive()`, Note that the return address has not been overwritten by `strcpy()`, but changed using a tainted pointer. Now when `overflow()` returns it returns to `receive()` directly, bypassing the `caller()` function, and will continue functioning as normal. Existing DIFT schemes will not detect this attack because they do not track load- and store-address dependencies. We tested this attack on our scheme and it was indeed detected. While the vulnerability in Figure 2 is a hypothetical example, vulnerabilities with load- and store-address dependencies are not uncommon in real application code, as discussed in Section 2.

5 Discussion and Future Work

While the focus of this paper has been on sensor network nodes, an interesting question is what other systems can DIFT schemes that check load- and store-address dependencies scale up to. Researchers have explored address dependencies on general purpose systems [28], finding that the complex memory management of these systems makes tracking these dependencies infeasible. It is an open question whether a DIFT scheme that tracks address dependencies would be practical on other embedded devices with more complex memory management than sensor network nodes. Many false alerts could be ameliorated by untainting tainted data under certain conditions. We leave this as the subject of future work.

It is well known that attacks can hijack control flow or otherwise take control of a machine without overwriting control data [5]. After control data and low-level control flow is secured then this can serve as a foundation for securing other types of data. Furthermore, it would be possible to extend our DIFT scheme to enforce various type systems by increasing the number of bits and allowing for more complex taint propagation rules. The Elbrus line of machines as described by Babayan [3] implemented this type of security for C programs, where any memory corruption attack that is a type violation (virtually all are) is prevented, but general purpose applications must be rewritten for reasons related to memory management. Whether sensor networks can be secured against non-control data attacks in this fashion and the impact this would have on power and performance are questions left for future work.

Because all first-order control dependencies in the applications that we tested were benign, we did not implement any special taint propagation rules for control dependencies. For applications that do have laundering control dependencies it would be possible to give developers an automated way of finding these and a secure way to taint them, perhaps by annotating their source code.

DIFT can be implemented simply by adding an extra bit to every byte in the registers, pipeline, and memory, and some simple logic gates where operations occur [29, 8]. The performance and power overheads are negligible. While any pipeline modification requires adoption of a DIFT scheme by chip manufacturers, several reasons make it more likely that sensor network applications will benefit from DIFT than general purpose computers. One reason that hardware DIFT support has not been adopted by chip manufacturers for general purpose computers is that for these systems DIFT provides no real security guarantees. In contrast, we demonstrate in this paper that satisfactory guarantees of security against a major class of attacks can be achieved for sensor network applications. Also, the economics of embedded devices presents less barriers to hardware support for security, testing, reliability, *etc.*, as evidenced by successful extensions such as JTAG [18]. Lastly, the MCU and memory of sensor network nodes are often packaged as one device so only one chip needs to be changed, whereas proper DIFT support for general purpose computers would require new DRAM chips, a motherboard with extra bits for sockets and buses, and operating system support for virtual memory swapping with tag bits.

6 Related Work

Gu *et al.* [17] showed that remote control data attacks can be exploited on sensor network nodes, and Francillon *et al.* [15] demonstrated that after exploiting a buffer overflow an attacker can even reprogram instruction memory with the SPM instruction. Our DIFT scheme for sensor network applications stops any attack that corrupts control data.

Ferguson *et al.* [14] propose a defense against control data attacks on sensor nodes that is similar to Control Flow Integrity [1, 2]. This incurs significant power and performance overhead and requires modifications to the program binary. Furthermore, attacks are not detected after control flow is hijacked until the attacker executes some code that transitions to another function. This is because the control flow sequence check is performed at the end of each function. Furthermore, the function marks are stored on the stack for function calls and are therefore also subject to corruption, and because they are 8 bits in length they can also be guessed with a brute force attack. Lastly, interrupt routines are not considered in the control flow sequence.

Kumar *et al.* [19] use software-based fault isolation to provide a coarse-grained form of memory protection for sensor applications and the operating system. Yang *et al.* [31] present a defense that is based on software diversity, where multiple versions of an application are created. Regehr *et al.* [26] implement efficient type and memory safety on a small, 8-bit embedded device. This requires modifications to the entire compilation chain, and even after optimization some applications can have more than a 35% increase in code size and a 6% performance overhead. Our proposed DIFT scheme would require hardware modifications, but would work with existing compilers and binaries with no need for binary rewriting, and have negligible power and performance overhead.

DIFT was introduced by Suh *et al.* [29]. Crandall and Chong [8] and Crandall *et al.* [9] explored various policy tradeoffs for DIFT schemes and higher-level systems issues such as virtual memory swapping, and Vigilante [6] employed DIFT for automated worm defense. Further research focused on making DIFT more flexible either in software [22] or through more flexible hardware extensions [11]. Argos [25] is a widely-deployed honeypot technology based on DIFT. Researchers have also analyzed the security and applicability of DIFT [7, 10, 23, 28].

Our scheme is the first DIFT scheme for sensor network applications. As discussed earlier in this paper, no existing DIFT schemes for general purpose computers address load-address, store-address, or control dependencies in a way that supports satisfactory security guarantees against control data attacks. Our scheme tracks all load- and store-address dependencies, and through testing and manual analysis we determined that all sensor network applications that we tested had only benign control dependencies. While flexible DIFT schemes [11, 22] could allow any policy to be enforced, nobody has actually specified a DIFT policy that secures general purpose systems. Research suggests that this is impractical for general purpose systems due to address dependencies [28], and even existing attacks not intended to evade DIFT have been shown to not be detected by existing DIFT policies [9].

Control Flow Integrity (CFI) [1, 2] uses binary rewriting and checks to ensure the validity of all control flow transfers. CFI requires no hardware modifications, and gives strong security guarantees even in an attack model where the attacker can read or write any location in memory. However, CFI requires significant changes to the compiler and library linking infrastructure. Furthermore, an implementation of something similar to CFI for sensor network nodes showed significant performance and power overheads, as well as a large increase in code size [14]. Without further research, it is not clear if these overheads were due to the particular implementation or are inherent to CFI.

7 Conclusion

We have presented a DIFT scheme for sensor network applications that is secure against remote control data attacks because all data dependencies are either tracked or shown to be benign. Whereas existing DIFT schemes for general purpose computers consider only two of the five data dependency types (copy, calculation, load-address, store-address, and control), our scheme tracks the first four and testing and manual analysis of applications demonstrates that the fifth is always benign (for the applications we tested, other sensor network applications could easily be tested in a similar manner). Testing of eight applications also showed that no modifications to the compiler, operating system, application source code, or program binaries is necessary. We expect that, whereas DIFT has not yet led to secure general purpose systems, DIFT will lead to secure sensor network devices with security built-in "from the ground up."

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback, and also Wenbo He for reading an early draft of the paper. This work and the publication and presentation of the paper were partially funded by the U.S. Defense Threat Reduction Agency, NSF CAREER #0844880, and NSF Trusted Computing #0905177.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS '05: Proceedings of the 12th ACM conference on Computer and Communications Security*, pages 340–353, New York, NY, USA, 2005. ACM.

[2] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. *ACM Transactions on Information and System Security*. To appear.

[3] B. Babayan. Security, www.elbrus.ru/mcst/eng/ SECURE_INFORMATION_SYSTEM_V5_2e.pdf.

[4] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In P. Syverson and S. Jha, editors, *Proceedings of CCS 2008*. ACM Press, Oct. 2008. To appear.

[5] S. Chen, J. Xu, and E. C. Sezer. Non-control-hijacking attacks are realistic threats. In *USENIX Security Symposium*, 2005.

[6] M. Costa, J. Crowcroft, M. Castro, and A. Rowstron. Can we contain internet worms? In *HotNets III*.

[7] J. R. Crandall and F. T. Chong. A Security Assessment of the Minos Architecture. In *Workshop on Architectural Support for Security and Anti-Virus*, Oct. 2004.

[8] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, December 2004.

[9] J. R. Crandall, S. F. Wu, and F. T. Chong. Minos: Architectural support for protecting control data. *ACM Trans. Archit. Code Optim.*, 3(4):359–389, 2006.

[10] M. Dalton, H. Kannan, and C. Kozyrakis. Deconstructing hardware architectures for security. In *Fifth Annual Workshop on Duplicating, Deconstructing, and Debunking*, June 2006.

[11] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *International Symposium on Computer Architecture (ISCA)*, 2007.

[12] J. S. Fenton. Information protection systems. In *Ph.D. Thesis, University of Cambridge*, 1973.

[13] J. S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(2):143–147, 1974.

[14] C. Ferguson, Q. Gu, and H. Shi. Self-healing control flow protection in sensor applications. In *WiSec '09: Proceedings of the second ACM conference on Wireless network security*, pages 213–224, New York, NY, USA, 2009. ACM.

[15] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 15–26, New York, NY, USA, 2008. ACM.

[16] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. *SIGPLAN Not.*, 38(5):1–11, 2003.

[17] Q. Gu and R. Noorani. Towards self-propagate mal-packets in sensor networks. In *WiSec '08: Proceedings of the first ACM conference on Wireless network security*, pages 172–182, New York, NY, USA, 2008. ACM.

[18] IEEE 1149.1, JTAG: Standard Test Access Port and Boundary-Scan Architecture.

[19] R. Kumar, E. Kohler, and M. Srivastava. Harbor: software-based memory protection for sensor nodes. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 340–349, New York, NY, USA, 2007. ACM.

[20] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. W. E., Brewer, and D. Culler. Tinyos: An operating system for sensor networks. Ambient Intelligence, 2005.

[21] Nergal. The advanced return-into-lib(c) exploits: PaX case study, Phrack 58.

[22] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS 05)*, Feb. 2005.

[23] K. Piromsopa and R. J. Enbody. Defeating buffer-overflow prevention hardware. In *Fifth Annual Workshop on Duplicating, Deconstructing, and Debunking*, June 2006.

[24] J. Polley, D. Blazakis, J. Mcgee, D. Rusk, and J. S. Baras. Atemu: A fine-grained sensor network simulator. In *IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, 2004.

[25] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. In *Proc. ACM SIGOPS EUROSYS'2006*, Leuven, Belgium, April 2006.

[26] J. Regehr, N. Cooprider, W. Archer, and E. Eide. Efficient type and memory safety for tiny embedded systems. In *PLOS '06: Proceedings of the 3rd workshop on Programming languages and operating systems*, page 6, New York, NY, USA, 2006. ACM.

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

[28] A. Slowinska and H. Bos. Pointless tainting? evaluating the practicality of pointer tainting. In *Proceedings of the 4th EuroSys Conference*, Nuremberg, Germany, Apr 2009.

[29] G. E. Suh, J. Lee, , and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of ASPLOS-XI*, Oct. 2004.

[30] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, December 2004.

[31] Y. Yang, S. Zhu, and G. Cao. Improving sensor network immunity under worm attacks: a software diversity approach. In *MobiHoc '08: Proceedings of the 9th ACM international symposium on Mobile ad hoc networking and computing*, pages 149–158, New York, NY, USA, 2008. ACM.