Enhancing Security Through Hardware-assisted Run-time Validation of Program Data Properties [*]

Divya Arora
Dept. of Electrical Engineering
Princeton University,
Princeton, NJ 08544
divya@princeton.edu

Anand Raghunathan,
Srivaths Ravi
NEC Laboratories America
Princeton, NJ 08540
{anand, sravi}@nec-labs.com

Niraj K.Jha
Dept. of Electrical Engineering
Princeton University,
Princeton, NJ 08544
jha@princeton.edu

ABSTRACT

The growing number of information security breaches in electronic and computing systems calls for new design paradigms that consider security as a primary design objective. This is particularly relevant in the embedded domain, where the security solution should be customized to the needs of the target system, while considering other design objectives such as cost, performance, and power. Due to the increasing complexity and shrinking design cycles of embedded software, most embedded systems present a host of software vulnerabilities that can be exploited by security attacks. Many attacks are initiated by causing a violation in the properties of data (*e.g.,* integrity, privacy, access control rules, *etc.*) associated with a "trusted" program that is executing on the system, leading to a range of undesirable effects.

In this work, we develop a general framework that provides security assurance against a wide class of security attacks. Our work is based on the observation that a program's permissible behavior with respect to data accesses can be characterized by certain properties. We present a hardware/software approach wherein such properties can be encoded as data attributes and enforced as security policies during program execution. These policies may be application-specific (*e.g.,* access control for certain data structures), compiler-generated (*e.g.,* enforcing that variables are accessed only within their scope), or universally applicable to all programs (*e.g.,* disallowing writes to unallocated memory). We show how an embedded system architecture can support such policies by (i) enhancing the memory hierarchy to represent the attributes of each datum as security tags that are linked to it through its lifetime, and (ii) adding a configurable hardware checker that interprets the semantics of the tags and enforces the desired security policies. We evaluated the effectiveness of the proposed architecture in enforcing various security policies for several embedded benchmarks. Our experiments in the context of the *Simplescalar* framework demonstrate that the proposed solution ensures run-time validation of program data properties with minimal execution time overheads.

Categories and Subject Descriptors: K.6.5 [**Management of Computing and Information Systems**]: Security and Protection
General Terms: Security
Keywords: Data tagging, run-time checks, secure architectures

1. INTRODUCTION

The large number of successful information security attacks in recent years, and the extent of damage that they have caused, have led to the emergence of security as an important concern in the design of electronic and computing systems. While most security attacks have hitherto manifested themselves in the context of general-purpose computers (PCs) and the Internet, embedded systems, which are used to perform a wide range of critical functions, represent an equally if not more serious target. Embedded systems are becoming increasingly complex, networked, and functionally extensible through software, exposing them to a host of security problems that have plagued general-purpose systems [7].

[*]This work was supported by NSF under Grant No. CCR-0326372.

The increasing complexity of software, together with shorter design cycles, implies that several vulnerabilities go undetected during the software design process, presenting easy targets for attackers. It should be noted that designing and implementing a program that is vulnerability-free even in a malicious environment is much harder than just ensuring functional correctness under well-behaved inputs. According to NIST's ICAT, the number of reported vulnerabilities in software has increased fivefold from 246 in 1998 to 1307 in 2002 [16].

Many attacks are initiated by causing a violation in the properties of data (*e.g.*, integrity, privacy, access control rules, *etc.*) associated with a trusted program that is executing on the system, leading to a range of undesirable effects. For example, the well-known heap and stack overflow attacks are based on causing a program to write excessive amounts of data into a variable or object in memory, resulting in other memory locations being corrupted [11]. Sometimes, this may even happen without the explicit execution of "malicious code," making detection very difficult. In this paper, we focus on such attacks, and propose a general hardware-assisted framework to prevent them.

1.1 Paper Overview and Contributions

We present an efficient and scalable framework that allows programs to specify security policies for their data and validate them during program execution. These security policies are specified by associating program data with *data attributes*, which can be defined either statically or during execution. The memory hierarchy is modified to provide easy access and efficient storage for these attributes, and to maintain their consistency during application execution. We also add a configurable hardware *checker* that uses the above attributes to determine the validity of data accesses. We describe the checks that our framework permits and discuss how they can be used to support a wide range of security requirements. Several experiments on embedded software benchmarks demonstrate that the proposed framework can be used to implement practical security policies with minimal overheads.

Ensuring security properties for program data has been previously studied in the context of software checking engines or interpreters. *We believe our work is the first attempt to provide a unified hardware-assisted platform to enforce a broad range of data properties.* The use of hardware permits us to address shortcomings of previous methods in the following areas:

- Coverage: Many software techniques sacrifice program coverage for performance, *i.e.*, they only prevent very specific attacks or only check a small set of known vulnerabilities (*e.g.*, a few C library functions). This doesn't provide any protection for programs that use custom versions of these functions or have bugs elsewhere.
- Flexibility: The type of security checks required may vary not only from one application to another, but also within a single application. Our framework allows the specification of security policies at different levels (application source, compiler, library, operating system (OS)), and is general enough to support this flexibility.
- Low overhead: Any security extension entails overheads in terms of time (development time, compilation time, execution time) and space (code bloat, run-time memory requirement). Software-based schemes impose high performance penalties making them inapplicable in practical scenarios.

The rest of the paper is organized as follows. Section 2 surveys relevant past work. Section 3 motivates the intuition behind the proposed technique, detailed in Section 4. Section 5 describes several widely applicable security policies that can be supported using the proposed framework. Section 6 presents experimental results and Section 7, the conclusions.

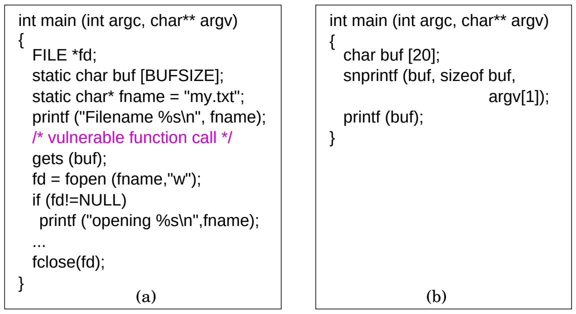
2. RELATED WORK

Various techniques have been proposed to address software security by ensuring that the code comes from a trusted origin and has not been corrupted during network transmission or storage (*e.g.*, hash checking) [19]. However, these techniques offer no protection against vulnerabilities that are unintentionally "built-into" trusted programs and exploited during execution. Recent techniques have extended the above approaches to dynamically ensure code integrity through run-time control-flow monitoring and hash checking [1, 13]. While they offer better protection than static techniques, they still do not directly address data protection. In contrast with the above works, a key objective of our approach is to detect the violations of data properties, irrespective of their eventual consequences. *Data protection is significantly harder than just protecting code, since unlike code (which is mostly known after compilation), data are highly dynamic, and may be allocated, initialized, and changed at run-time.*

Buffer Overflow Prevention: Various techniques have been proposed to protect against stack and heap-based buffer overflows. Function wrappers are employed in [9] to intercept calls to memory allocation/ deallocation (`malloc, free`) functions and maintain a record of the address and size of allocated memory chunks in an internal table. Likewise, calls to a few vulnerable C functions are intercepted and bounds checks performed before the actual function is called. This method works only for programs that are dynamically linked with the standard C library. TIED/LibsafePlus [3] are tools that operate on similar principles, except that they include stack protection too. The authors in [18] protect against a specific type of heap overflow that results in corruption of memory management information (stored in the beginning of the allocated chunk). Most of these techniques offer protection against very limited types of attacks. Moreover, the performance penalty for such software-based techniques is very high, especially if protection needs to be extended beyond a small set of functions and data buffers.

Memory Access Checking: The objective of ensuring validity of memory accesses extends beyond security, and was first explored in the context of software debugging/validation. Purify [10] is a software debugging tool that detects memory access errors and leaks. The technique uses object code instrumentation and imposes an execution time overhead of about 15X and code increase of 2X, hence it is used only during program debugging/development. Approaches, such as Cyclone [8] and CCured [15], have been developed to address the limitations inherent in C. However, they suffer from compatibility issues as they use an extended pointer format and rely on garbage collection. In [12], a compiler extension is proposed that maintains a table containing intended referents, size, location, *etc.*, for all pointers and checks pointer dereferences against this table. Austin *et al.* [2] introduced the *safe pointer* representation, to incorporate attributes such as size, storage class, *etc.*, within the pointer data type. In [21], the authors perform run-time checking through the use of memory tags. They maintain 1-bit tags to denote allocated/ unallocated memory but reduce the amount of run-time checking by static analysis. Despite the limited scope of checking, this technique results in large overheads in terms of compilation time (maximum 9X), execution time (8X) and code size (4X). A broader version of the above technique [14] uses tags to perform run-time type checking, but suffers from even higher overheads.

In summary, we conclude that any technique to safeguard program data has to be employed at run-time. It should be extremely efficient - hence our proposal for a hardware-assisted framework. Finally, it should be scalable and provide flexibility to check for different properties during different phases of execution.



```
int main (int argc, char** argv)
{
   FILE *fd;
   static char buf [BUFSIZE];
   static char* fname = "my.txt";
   printf ("Filename %s\n", fname);
   /* vulnerable function call */
   gets (buf);
   fd = fopen (fname,"w");
   if (fd!=NULL)
     printf ("opening %s\n",fname);
   ...
   fclose(fd);
}
```
(a)

```
int main (int argc, char** argv)
{
   char buf [20];
   snprintf (buf, sizeof buf,
                        argv[1]);
   printf (buf);
}
```
(b)

Fig. 1: (a) Program data corruption via heap overflow, and (b) unauthorized stack access by format string

3. MOTIVATION

Fig. 1 illustrates program vulnerabilities that can be exploited to violate data properties, leading to security attacks. Fig. 1(a) shows a program that is vulnerable to heap overflow attacks. The code contains two static variables, `buf[ ]` and `fname`, which are allocated on the program heap. The buffer pointed to by `fname` stores the name of a file into which the program writes its output. The program uses a vulnerable C function `gets()` to get user input and store it in `buf[ ]`. A malicious user can provide a long input that overflows `buf[ ]` and overwrites `*fname` with, say, "/etc/hosts," tricking the program into writing to a completely different (sensitive) file.

Another kind of security violation is shown in Fig. 1(b). The given program naively assumes that the user input (`argv[1]`) is a character string and copies it into `buf[ ]`. However, a user can supply "`%x`" as the command line argument, in which case the succeeding function call looks like `printf("%x")`. This pops a four-byte integer value off the stack and sends it to `stdout`, permitting the adversary to view contents of the caller's (`main()`) stack-frame. Sensitive information stored on the caller's stack may be revealed in this way.

The above examples highlight three key points – (a) attacks that read/corrupt program data *without redirecting control flow* are possible and can have grave consequences, (b) many such attacks are facilitated by vulnerabilities present in the application code itself, and (c) neither the OS nor processor architecture provides any mechanism to prevent transgressions by one part of an application against another (*e.g.*, the programmer may want to restrict access of `gets()` to `buf[ ]` in Fig. 1(a)). Our framework aims at providing security at both coarse and fine granularities. For example, it enables an application to be self-checking and dynamically specify which pieces of data can be read/modified by each section of code.

4. PROPOSED FRAMEWORK

In this section, we provide an overview of the proposed framework for validation of program data properties. We describe the architectural enhancements that it requires, and the software interface that allows applications to use it.

4.1 Overview

Fig. 2 provides a high-level overview of the proposed technique, and identifies the parts of the hardware architecture and software design flow that are enhanced in it. At any point in execution, the state of a program variable can be represented as a tuple *s(VA, Size, Value)*, where *VA* is the virtual address of the variable and *Size* is its size in bytes. We add a new field – *SECTAG* (*SECurity TAG*) to the abstract data state, which encodes security-specific attributes associated with each piece of data. These attributes may be derived from application-specific policies, *e.g.,* allowing only certain functions to access a particular global data structure, or they may be universally applicable to all programs, *e.g.,* disallowing writes to unallocated memory. Depending on the policy, data SECTAGs may be set explicitly by the application or implicitly with support from run-time libraries. The former usually requires knowledge of application semantics accompanied by source code modification. The latter may be imple-

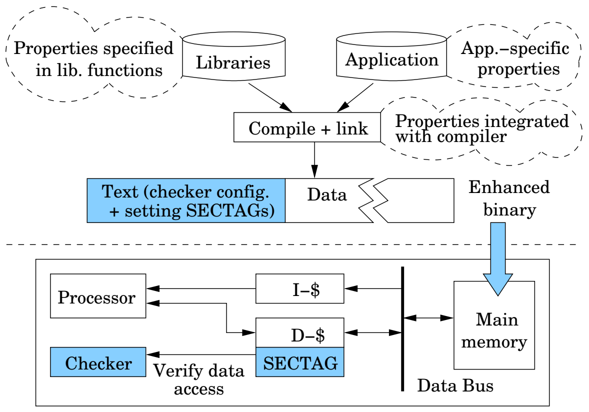


Fig. 2: Proposed framework for run-time data validation

mented by intercepting function calls to dynamically-linked libraries, adding user-defined hook functions, *etc.* Additionally, some attributes may be set automatically by the compiler, *e.g.*, making the return address of a function unwritable by the callee on each function call.

The modified compilation flow produces an enhanced binary which contains instructions to set SECTAGs at appropriate locations and to communicate configuration parameters (explained later). Once SECTAGs are set, all data memory accesses are monitored and verified in parallel by a dedicated hardware checker.

4.2 Architectural Enhancements

The simplest hardware implementation of the above technique would involve expanding the capacity of the register file, on- and off-chip caches, TLBs and main memory with extra space per data byte, which is reserved exclusively for storing SECTAGs. There are two problems with this approach. First, it implies a constant space overhead – even for applications that are not using any security checks. Some applications may require a multi-bit representation for their data attributes. A naive implementation will suffer from unacceptable area overhead in that case. Secondly, most contemporary processors have separate first-level (L1) instruction and data caches but a unified level-two (L2) cache. Having extra tagging space for all L2 cache lines, many of which may actually be storing code, is a waste.

On the other extreme is a purely software scheme where there are no special hardware structures and SECTAGs are treated as any other virtual memory space and fetched into cache only on application request. Such schemes suffer from high execution time overhead even for a single bit tag [21].

We propose a compromise between the two extremes, *i.e.*, we maintain a separate SECTAG memory space only for the L1 D-cache and register file. In the L2 cache, SECTAGs are managed by the modified cache controller and are "packed" in the cache for maximum storage efficiency. L1 SECTAG bytes

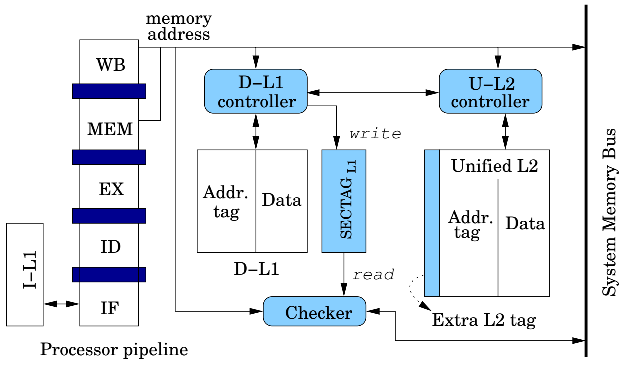


Fig. 3: Architecture for run-time data validation

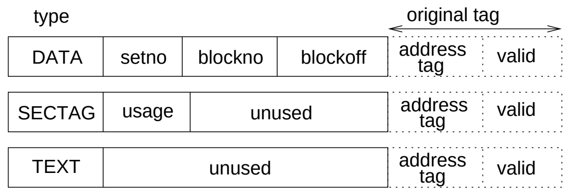


Fig. 4: Modified L2 tag structure

are checked in parallel with L1 accesses ensuring that there is no slowdown on an L1 hit.

4.3 Architectural Details

Fig. 3 shows the block-level processor architecture with the new/ modified parts highlighted. The design choices made in this architecture are driven by the need to balance extra on-chip space with checker efficiency. In the following subsections, we analyze the modified components one by one.

4.3.1 L2 Cache Controller

A major chunk of our modifications involves the operation of the L2 cache controller and tags. Under normal operation, on an L2 hit, the controller transfers data to L1 (the size of an L1 cache block). On a miss, it fetches an L2 cache block sized data from main memory. The fetched data occupy an empty cache line or replace another valid cache line. The replacement policy, which decides which cache line is to be evicted, is implemented in hardware and embedded in the controller.

For reasons mentioned earlier, we do not want to add SECTAG bytes to each cache line. Therefore, we permit any cache line to store the SECTAG field for any other L2 cache block and modify the cache controller to update the bookkeeping information. New fields are added to the regular L2 cache tag, as shown in Fig. 4, for bookkeeping. The two-bit field `type` signifies the type of cache line which can be any of the following:

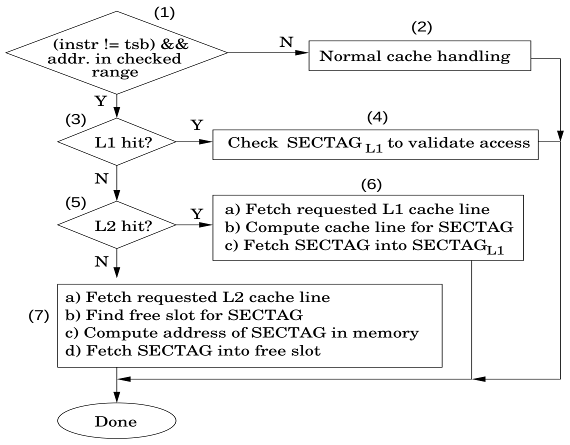
- `DATA`: This includes cache lines which store application data that are subject to checking. Each byte in this area can have some attributes assigned to it via the SECTAG field and checked at run-time.
- `SECTAG`: This field is set for cache lines that store the SECTAG field of some other L2 cache line.
- `TEXT`: Cache lines having program code or data for which attributes are not checked belong to this category.

These fields are set by the cache controller based on the address of the memory accessed. The application loader communicates two parameters after loading, to configure the controller – lower address bound of checked data and bit-width of SECTAG.

The semantics of the remaining fields differ depending on the `type` of the cache line. If the `type` is `DATA`, the remaining fields encode the location of its SECTAG with set index (`setno`), block number (`blockno`) and offset within the block (`blockoff`). The `blockoff` field is required because a single cache line may store multiple SECTAGs in series (as explained below). For `SECTAG type` cache lines, the `usage` field is a bit-mask that signifies which slots in the line are filled. In the beginning, a `DATA` line, say l_1, is fetched from memory. Subsequently, its SECTAG is also fetched and stored in a cache line, say l_2, whose `type` is marked off as `SECTAG`. However, depending on the bit-width of SECTAG, a single cache line may have space for multiple SECTAG slots. As and when more `DATA` lines are fetched, their SECTAGs are stored in l_2 and its `usage` field updated. Now, if l_1 is evicted and replaced by a `TEXT` line, its SECTAG slot in l_2 becomes vacant and available for subsequent use.

A simple back-of-the-envelope calculation shows that for a one-bit SECTAG, an 8-way associative, 64K L2 cache, with 128 byte lines, requires ($block\ size \times width(SECTAG)$) or a 128-bit tag per cache line, if the tags are added to each line. However, with our scheme, the number of additional bits per cache line is $2+log_2(\#sets)+log_2(associativity)+log_2(8/width(SECTAG))$, or 14. This difference becomes even more significant when higher bit-width SECTAGs are employed.

SECTAGs of data bytes are efficiently packed into cache lines by the cache controller to allow a maximum number of



(1)
(instr != tsb) && addr. in checked range
N
(2)
Normal cache handling
Y
(3)
L1 hit?
Y
(4)
Check SECTAG L1 to validate access
N
(5)
L2 hit?
Y
(6)
a) Fetch requested L1 cache line
b) Compute cache line for SECTAG
c) Fetch SECTAG into SECTAG L1
N
(7)
a) Fetch requested L2 cache line
b) Find free slot for SECTAG
c) Compute address of SECTAG in memory
d) Fetch SECTAG into free slot
Done

Fig. 5: Modified cache hit/miss handling

DATA lines to be resident in the cache. The above organization along with a modified L2 miss handler maintains the following invariant: *For all valid cache lines marked* DATA *in L2, their respective SECTAGs are also present in L2 at all times.* The modified cache hit/miss handling procedures are summarized in Fig. 5. L2 miss handling logic (Step 7) fetches the requested cache line from memory as before. In parallel, it computes the address of the corresponding SECTAG location. Currently, a portion of the memory is reserved for storing SECTAGs and their address is determined from the address of the data they belong to, using a fixed mapping. After the requested data fetch completes, the controller fetches its associated SECTAG too. The miss replacement policy is altered to not permit a cache line of type SECTAG to be independently evicted. This is because all the associated DATA cache lines will still point to the evicted line for their SECTAGs, leading to inconsistency.

4.3.2 L1 Cache Controller

In L2 cache design, our emphasis was on minimizing the extra memory required and providing scalability to accommodate programs with varied security requirements (in terms of amount of data to protect, SECTAG bit-widths, *etc.*). However, in L1 design, we focus on minimizing the execution time overhead of the common case - L1 hit.

We add a SECTAG$_{L1}$ section to the data L1 cache which is accessed in parallel with every D-L1 (see Fig. 3) access. This means additional memory per cache line, the size of the maximum SECTAG width that we expect an application to require. This may seem like a lot of overhead even for a single-bit wide SECTAG. However, a D-L1 cache is usually much smaller than a unified L2 cache. We propose a separate on-chip SRAM to store L1 SECTAGs instead of increasing the tag length of D-L1 (cache tags are implemented using content addressable memories whose cell is about 8 times the size of a standard SRAM cell).

The D-L1 cache controller maintains the following invariant: *For all valid cache lines in D-L1, their corresponding SECTAGs are stored in SECTAG$_{L1}$ at all times.* On an L1 hit, while the data are being accessed by the controller, the corresponding SECTAG is accessed in parallel by the checker. Thus, there is no increase in the hit latency of D-L1. Note that we do not need to limit the checking to data writes. When data and SECTAGs are in place, checking of both reads and writes comes for free. On an L1 miss, however, both the data and its SECTAG have to be fetched from L2 (or main memory). In case of an L2 hit (Step 6), the L1 miss penalty is approximately double that of execution with a regular cache.

4.3.3 Checker

The checker is a separate hardware unit that performs SECTAG verification for the executing program. The simplest way to allow/disallow access to a particular region of memory is to set its SECTAGs to a particular value before executing code that accesses the region, and resetting SECTAGs after exit from the code fragment. The checker verifies, for each data read and write, if its SECTAG is set to the specified value. More complicated policies can be expressed as a comparison of SECTAG values and enforced by the checker. The checker has several registers that can be used to configure it, either statically (during application loading) or dynamically (during execution). The application specifies the address range of checked data in $R_{High\ addr}$, $R_{Low\ addr}$ registers – *e.g.*, $R_{High\ addr}$ can be made equal to the stack pointer register, so that only the program heap is subject to checking. Register $R_{High\ value}$ is used to restrict a section of code from accessing data whose SECTAG is higher that the value specified in this register. $R_{Command}$ enables/disables various options associated with checking - exact value checking (which uses $R_{Read\ value}$ or $R_{Write\ value}$ as reference), range checking (against values in $R_{High\ value}$, $R_{Low\ value}$), *etc.*

Apart from denying access to certain parts of code, access can also be restricted based on the attributes of source data. A small addition is required to our base architecture to enforce such policies - the register file is augmented with SECTAG memory in which the SECTAG of data is copied whenever data are loaded into a register from memory. This SECTAG is propagated across move/arithmetic instructions and used for comparison when data in one location are copied (source) to another (destination) to control write access. Despite all the functionality described above, the logic of the checker is fairly simple (mostly comparators), thus it can easily run concurrently with L1 data cache accesses.

4.4 Application Interface

The responsibility for determining which data need checking, and for setting their SECTAGs to appropriate values, lies with the program. The hardware described above is only involved in maintaining its consistency and providing easy access. On a data access, the translation of data address to SECTAG address is done in hardware and they are read from the memory consecutively.

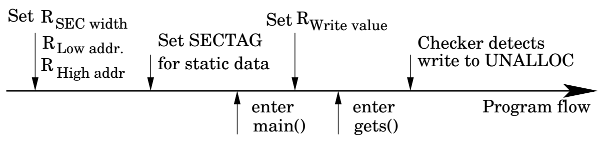
We introduce a new instruction `tsb` $\$R_s$,$\$R_t$ (*tag store byte*) to enable setting SECTAGs. The address in $\$R_s$ is the virtual address of the data byte whose SECTAG is to be set to the value in $\$R_t$. This is a modified `sb` (*store byte*) instruction used to store SECTAGs. Regular virtual memory protection ensures that the application is allowed to access the address specified in the instruction. However, the actual address to which the value is written is determined by the hardware via a pre-defined address translation. Also, on a cache miss, a regular store instruction would result in fetching and checking of corresponding SECTAGs (as explained above). However, a `tsb` instruction just fetches the translated address from memory and writes to it (thus saving us from a recursive situation!). Registers of the checker are memory-mapped and accessed like any other I/O device by the application.

5. SECURITY POLICIES

We envision a number of scenarios where our framework can be employed by applications to impose specific policies to secure their data during execution.

Ensuring memory writes are to allocated memory: A large number of security exploits can be avoided if every data write is verified to be within bounds. To enforce this, on memory allocation, the application sets SECTAGs of data to, say ALLOC, and resets it to UNALLOC on deallocation. All writes and reads between these two tag setting operations are verified by the hardware transparently. A linear buffer overflow is caught easily by ensuring that each data chunk in memory is surrounded by at least one byte of unallocated memory. Hence, when a function attempts to access the byte beyond the buffer end, it will find the respective SECTAG to be UNALLOC and immediately flag an error.

Example: Fig. 6 shows the sequence of operations that lead to detection of the security violation presented in Fig. 1(a). Recall that the call to function `gets` could be exploited to overwrite the contents of `fname` by providing an input string of excessive length. Under the proposed framework, the compiler adds instructions to write SECTAG width (1 in this case) to $R_{SEC\ width}$, the value ALLOC to $R_{Write\ value}$, and the start/end of the `data` section to $R_{Low\ addr}/R_{High\ addr}$, respectively. It also instruments the program to mark static array `buf` and location pointed to by `fname` as ALLOC in the beginning of `main()`. It keeps 4 bytes of padding between the two buffers which is tagged `UNALLOC`. Now, if `gets()` is supplied with a long user input that overflows `buf`, the processor attempts to write to a data location marked `UNALLOC`, which is flagged by the checker.



Set $R_{\text{SEC width}}$
$R_{\text{Low addr.}}$
$R_{\text{High addr}}$
Set SECTAG for static data
Set $R_{\text{Write value}}$
Checker detects write to UNALLOC
enter main()
enter gets()
Program flow

Fig. 6: Sequence of steps leading to detection

Ensuring memory reads are to initialized memory: The implementation for this case is similar to above, *i.e.*, SECTAGs are set to `ALLOC` after allocation. The checker is configured to allow only write operations on data that are tagged `ALLOC`. After explicit initialization, SECTAGs are set to `INIT`. The program is allowed to issue reads only for data that are tagged `INIT`.
Controlling access to sensitive data on stack: Before calling another function, the caller can choose to make its stack unreadable/unwritable. By setting the SECTAGs corresponding to different local variables differently, the caller can grant access privileges at a fine-grained level. Also, different privileges can be given to different callees for different call instances. After compilation, the layout of local variables on the function stack is known. Therefore, the compiler can be instrumented to include instructions in the binary to set SECTAGs of local variables as required.
Partitioning application data: A program can be partitioned into parts that require different security levels. Then the program can enforce policies which disallow code segments to read/write data above (or below) a specified security level. This level can be changed dynamically to allow different privileges for different subroutines. Software techniques such as Privtrans [4] automatically partition an application into a master and a slave process with the former running with greater privileges. Our framework allows fine-grained separation of privileges into multiple levels without requiring the application to be split into two processes.
Preventing data overwrite by user input: Schemes, such as dynamic information flow tracking [20], can be extended to the data space, wherein all user input is tagged as untrusted and not permitted to be copied to sensitive data locations. This is actually a special case of partitioning application data based on its source, *i.e.*, treating user input as the lowest level and not allowing it to be copied onto function pointers, *etc.*

6. EXPERIMENTAL RESULTS

We experimented with two application properties with the goal of evaluating the effectiveness of our architecture and the execution time overheads involved in using it. As mentioned above, our architecture supports enforcement of application-specific security policies. However, to avoid any bias originating from the choice of application, we selected properties that are quite generic and applicable to most programs. We used benchmarks from the MediaBench and MiBench benchmark suites, as many of them are data-dominated applications. Performance estimation was performed using a cycle-accurate simulator based on the SimpleScalar 3.0/PISA simulation toolset [5]. All simulations were run till the termination of the application under test. The default configuration of the simulated processor is shown in Table 1.

Table 1: Architectural parameters used in our experiments

Parameter	Config.	Parameter	Config.
I-L1/D-L1	32kB/32kB	Unified L2	64kB
I-L1 line size	32B	D-L1 line size	32B
I-L1 assoc.	32	D-L1 assoc.	32
L1 hit latency	1 cycle	L2 hit latency	6 cycles
L2 line size	64B	L2 assoc.	8
out-of-order	yes	Mem. width	8B
Mem. latency: 1st chunk (8B)	18 cycles	inter-chunk	2 cycles
Fetch Q size	8	Issue width	2

6.1 Policy I: Eliminating Reads/Writes to Unallocated Memory

We instrumented the memory management routines, `malloc` and `free`, to set SECTAGs of the requested buffer to `ALLOC` and `UNALLOC`, respectively. These functions are part of the standard C library and allow addition of one's own hooks. Our hook function calls regular `malloc` and then sets SECTAGs for the allocated memory. If the library is dynamically linked, the function call can be intercepted at run-time and does not require change in source code or recompilation. Once these functions are modified, they can keep track of memory allocated throughout the application, including other library routines. Most current versions of `malloc` store memory management information associated with each memory chunk "in band" in a four-byte header at the beginning of the chunk itself. Our modified `malloc` sets the SECTAGs to `ALLOC` only for the `size` specified in its argument. The header is marked `UNALLOC`. This delineates dynamic buffers from one another and catches overflow.
Performance metrics: Fig. 7(a) shows the performance impact of checking a 1-bit tag for every heap access, for different L2 cache sizes. The figure shows the processor cycle count normalized over cycle count for the unmodified processor. The average and maximum overhead for our default cache configuration (64K) are 7.6% and 32.7%, respectively. Except for one benchmark (`pegwit`) performance does not deteriorate significantly with decreasing cache size. Fig. 7(b) depicts the breakdown of the overhead. `set_latency` is the time taken to set the tags using `tsb` instructions, `fetch _latency` is the time the processor is stalled because of extra cache miss penalty and `other` represents the loss due to the side-effects of maintaining SECTAGs in cache. The last two factors account for a major part of the overhead in most benchmarks with `set_latency` contributing to only 13.7% of the overhead on average. Fig. 8 shows the strength of our approach in supporting SECTAGs of higher bit-widths. The experiments were conducted with the same set-up as above, *i.e.*, checking every heap access, but setting tags for dynamically allocated memory to multi-bit values. The incremental performance loss for increasing SECTAG size is small for most benchmarks (average increase of 2.6%, 11.2% and 43.5% when going from width 1 to 2, 4, and 8, respectively).

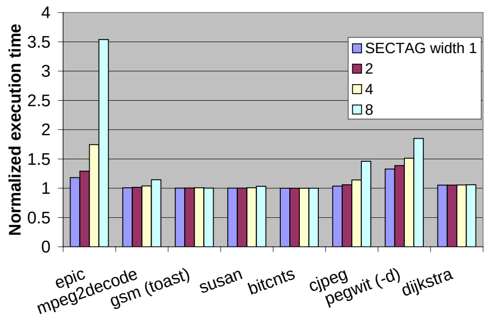
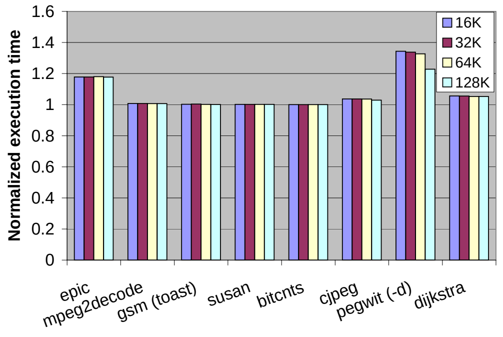


Fig. 8: Execution time with varying SECTAG bitwidth

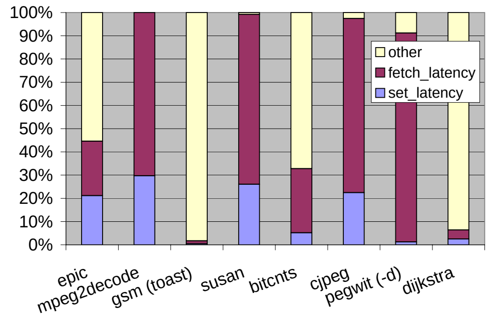
Security implications: This policy catches all reads and writes to unallocated memory including overflow onto an adjacent buffer and accessing an array after it has been freed. In addition, it also prevents exploits that operate by corrupting the `malloc` chunk header where management information is stored [17]. However, each call to `malloc()` and `free()` involves manipulation of header information, such as pointers to other memory chunks on the heap. Hence, the checker has to be disabled while these functions are being executed, otherwise it signals a large number of false positives. In our implementation, this is done by disabling the checker in the hook functions before the actual library call is made, and reenabling it on return.

6.2 Policy II: Restricting Access to the Program Stack

The size and location of local variables within each function's stack frame are fixed by the compiler. We use 2-bit



(a)



(b)

Fig. 7: (a) Execution time overheads for checking heap accesses, and (b) breakdown of overheads

SECTAGs to mark data as `READ`, `WRITE`, `R/W` and `NONE`. The calling function sets these SECTAGs for its local variables before calling another function. *E.g.*, a function that passes all parameters by value, sets its entire stack to `NONE` before the call, and reverts SECTAGs back to `R/W` on return. The process of setting tags for the stack is more involved than the previous policy, since a large number of functions use pointers to their local variables as arguments in function calls. This information is readily available at compile-time, and was used to perform the appropriate SECTAG setting for each function call.

Performance metrics: Fig. 9 shows the performance impact of checking each memory access to the program stack, with varying L2 sizes. The average and maximum overhead for the default configuration (64K) are 8.4% and 23.5%, respectively. The performance of our scheme is fairly robust to decreasing cache size.

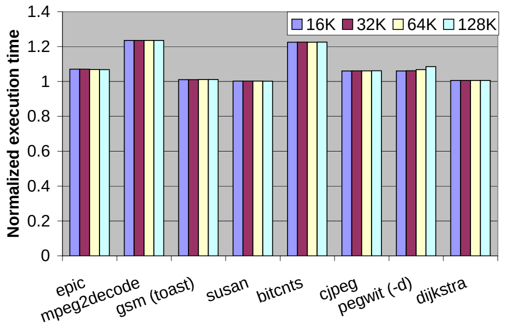


Fig. 9: Execution time overheads for checking stack accesses

Security implications: This scheme prevents unwarranted disclosure or corruption of sensitive information on the caller's stack. As a special case, it prevents invalid control flow behavior resulting from modification of the return address (RA). In a typical "stack smashing" exploit, a local buffer on the stack is overflown resulting in corruption of the RA, which is also stored on the stack. With this scheme, the caller marks the RA and frame pointer unwritable before making the call, thus preventing their corruption. Some techniques [6] detect attempts to corrupt the RA at the expense of little performance loss. However, the protection they offer does not extend to other data on stack.

7. CONCLUSIONS

In this paper, we presented an architectural framework to enable programs to specify and validate security policies for their data. Applications can leverage this platform to enforce custom security policies governing access control by parts of the same application. Our design provides a run-time configurable checker that monitors specified data attributes and detects violations with minimal performance loss.

Ensuring secure software execution in the presence of potential vulnerabilities in programs is an important but challenging task. We believe that the proposed framework for hardware-assisted run-time validation of data properties is a promising step in this direction.

8. REFERENCES

[1] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha. Secure embedded processing through hardware-assisted run-time monitoring. In *Proc. Design, Automation and Test in Europe*, pages 178–183, Mar. 2005.

[2] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proc. SIGPLAN Conf. Programming Language Design and Implementation*, pages 290–301, June 1994.

[3] K. Avijit, P. Gupta, and D. Gupta. TIED, libsafeplus: Tools for runtime buffer overflow protection. In *Proc. USENIX Security Symp.*, pages 45–56, Aug. 2004.

[4] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proc. USENIX Security Symposium*, pages 57–72, Aug. 2004.

[5] D. Burger and T. M. Austin. The SimpleScalar tool set, Version 2.0. Technical report, Computer Sciences Department, University of Wisconsin-Madison, June 1997.

[6] C. Cowan *et al.* Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. USENIX Security Symp.*, pages 63–77, Jan. 1998.

[7] P. Kocher et al. Security as a new dimension in embedded system design. In *Proc. ACM/IEEE Design Automation Conf.*, pages 753–760, June 2004.

[8] T. Jim et al. Cyclone: A safe dialect of C. In *Proc. USENIX Annual Technical Conf.*, pages 275–288, June 2002.

[9] C. Fetzer and Z. Xiao. Detecting heap buffer overflow through fault containment wrappers. In *Proc. Symp. Reliable Distributed Systems*, pages 80–89, Oct. 2001.

[10] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proc. Winter USENIX Conf.*, pages 125–136, Jan. 1992.

[11] M. Howard and D. LeBlanc. *Writing Secure Code*. Microsoft Press, 2002.

[12] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proc. Int. Wkshp. Automated and Algorithmic Debugging*, pages 13–26, May 1997.

[13] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Proc. USENIX Security Symp.*, pages 191–206, Aug. 2002.

[14] A. Loginov, S. H. Yong, S. Horwitz, and T. W. Reps. Debugging via run-time type checking. In *Proc. Int. Conf. Fundamental Approaches to Software Engineering*, pages 217–232, Apr. 2001.

[15] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proc. Symp. Principles of Programming Languages*, pages 128–139, Jan. 2002.

[16] ICAT statistics. http://icat.nist.gov/icat.cfm?function=statistics.

[17] Once upon a free(). `http://www.phrack.org/phrack/57/p57 -0x09`.

[18] W. Robertson, C. Kruegel, D. Mutz, and F. Valeur. Run-time detection of heap-based overflows. In *Proc. USENIX Large Installation Systems Administration Conf.*, pages 51–60, Oct. 2003.

[19] W. Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall, 1998.

[20] G. E. Suh, J. Lee, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pages 85–96, July 2004.

[21] S. H. Yong and S. Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *Proc. European Software Engineering Conf. held jointly with 11th ACM SIGSOFT Int. Symp. Foundations of Software Engineering*, pages 307–316, Sept. 2003.