# CAWDOR: Compiler Assisted Worm Defense

Jun Yuan
Department of Computer Science
Stony Brook University
Stony Brook, NY 11790, USA

Rob Johnson
Department of Computer Science
Stony Brook University
Stony Brook, NY 11790, USA

***Abstract*—This paper explores how much the source code analysis can assist worm defense system. Previously-proposed worm defense systems have used disparate mechanisms to detect worms, analyze exploits, verify alerts, and apply mitigations. Furthermore, previous systems have not offered predictability, i.e. it is not possible to verify, in advance, that the defense system will never generate a mitigation that breaks the program.**

**This paper describes a program transformation technique that makes collaborative worm defense systems easy to build, predictable and fast-responsive. Our transformation provides a single building block that can be used to perform worm detection, exploit analysis, alert verification, and mitigation application. In fact, our transformation makes most of these tasks trivial. Furthermore, software vendors and users can test, in advance, that the defense system will very unlikely apply a mitigation that breaks their software. Mitigations are vulnerability-specific not exploit-specific. Finally, our system can respond extremely quickly to a new worm. The exploit analysis becomes trivial so sentinel hosts can issue an alert the instant they detect a worm.**

**We have implemented a prototype of our system based on the Jones and Kelly program transformation for memory safety. During normal operation, our system incurs only 5% overhead. We take advantage of static analysis to develop several optimizations and make the Jones and Kelly approach to memory safety efficient and practical.**

## I. Introduction

Internet worms have caused billions of dollars in damage, have interfered with critical services, and may have even indirectly caused loss of human life [12]. Worms exploit bugs in host software and spread automatically from computer to computer. Criminal organizations use worms to build botnets from which they launch DDoS attacks, steal personal information via keyloggers, conduct phishing attacks, and send spam [9]. Many worms exploit zero-day vulnerabilities and researchers have demonstrated that a well-engineered worm could infect all vulnerable hosts in a matter of seconds [14], [22].

Previous proposed worm defense systems have not met the desirable needs of low overhead, fast responsive, automatic, accuracy and vulnerability-specific. Much research has focused on automatic signature generation[11], [20]. Input filters are exploit-specific and therefore may have both false positives and false negatives. Even defense systems that generate vulnerability-specific mitigations, such as Vigilante or Sweeper, may still break working systems. Some installations may routinely cause benign safety violations (e.g. they may silently overflow a buffer into unused space, causing no visible ill effects), so applying a mitigation, even a mitigation that has no "false positives", may break such an installation. Furthermore, these systems are large and complex – they use different mechanisms for the detection, analysis, verification, and application tasks.

We show that the worm defense system can be simple and can offer two seemingly contradictory features: complete predictability, and the ability to respond to zero-day worms. We propose a compiler transformation that inserts instrumentation that can be dynamically enabled or disabled at run-time. Hosts can perform worm detection by enabling a random subset of these checks. The system can have low overhead because hosts do not turn on all checks, enabling a large fraction of hosts to participate in worm detection. Upon detecting a worm, analysis is greatly simplified, since the host knows exactly which check failed. Alerts can include the identifier of the relevant check, simplifying alert verification and making mitigation application trivial: the recipient simply needs to activate the indicated check. This simply can respond quickly because a significant amount of the analysis and mitigation generation work is done at compile time.

Since mitigations are checks embedded in the program at compile time, software vendors and users can test mitigations in advance. They can, for example, test the software with all checks turned on. Any check violation discovered during testing will correspond to a real software bug that can be fixed before deployment. Thus vendors and users can have high confidence that the defense system will unlikely break their installations. There is a slight chance that turning on different sets of checks may change the undefined program behavior that a bug depends to trigger, however, it is impractical to prevent all the undefined and unspecified program bugs from the compiler level. In turn, it is more difficult for attackers to exploit some undefined bugs from programs with random distributed checking mechanism.

We have implemented a transformation supporting individually activation based on the Jones and Kelly [10] memory-safety transformation [7], [26], [15], and we have solved several shortcomings of the original design. Our transformation has a sophisticated optimizer for reducing run-time overhead. It supports out-of-bounds pointers without the complexity of OOB structures [18]. It also supports linking transformed and untransformed code, providing good compatibility with libraries that may not have our defense applied.

We evaluate the performance of our system under four scenarios. Some hosts may not be performing detection and

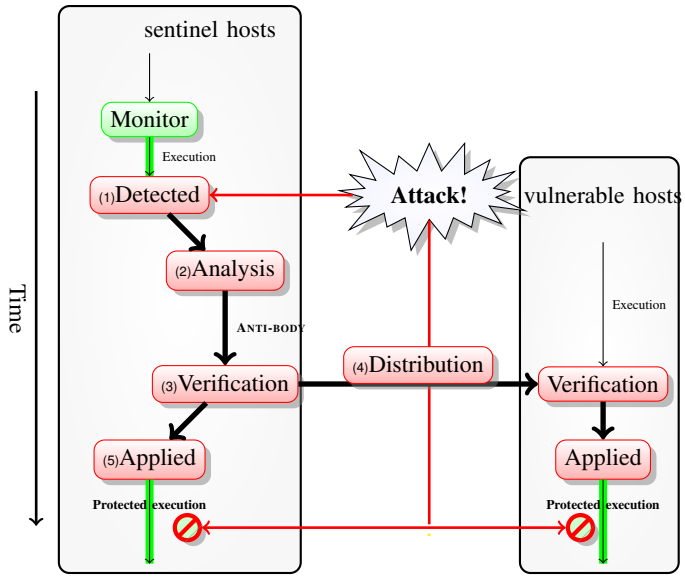


Fig. 1. A general collaborative worm defense system.

hence will have all the checks disabled. In this scenario, the average overhead of our latent run-time checks is 5%. Once a host receives a valid alert, it will activate a single check. The overhead of our system in this scenario is 9%. A host performing worm detection will have a small random subset of checks enabled. For example, with 5% of checks enabled, our system incurred a 29% overhead. Finally, we evaluate the overhead of our system with all checks activated: the average overhead was 108%. This demonstrates that our optimizations provide a significant speedup of the Jones and Kelly defense (10X) , and that dynamically-activated checks can provide a significant additional reduction in overhead.

In summary, this paper makes the following contributions:

- We show that much of the work in detecting, analyzing, verifying and patching the vulnerabilities in a collaborative worm defense system can be made simpler and more efficient with compiler support.
- As a side benefit, we show that an automated worm defense system can be totally predictable.
- We build a prototype compiler and show that our program transformation for collaborative worm defense has low overhead.
- We show how to use static analysis to dramatically improve the performance of the Jones and Kelly memory safety transformation, how to support a larger set of safe programs, and how to solve several compatibility issues between transformed and untransformed code.

## II. Related Work

### A. Internet Worm Defense Systems

Figure 1 shows the overall architecture of a collaborative worm defense system. There are several variations on this basic design. In some defense systems, the sentinel hosts are a specially designated subset that are willing to tolerate the overhead of run-time monitoring [8], [21], [6], [24]. Other defense systems attempt to reduce monitoring overhead so that every host can be a sentinel. Recipients of alerts may or may not trust the sentinel host that generated it. Most defense systems include a copy of the worm in alert messages so that the recipient can replay the bad input to verify that an anti-body is really necessary. They may then generate the anti-body themselves, eliminating the need to trust other hosts in the defense system.

Vigilante presents approaches to detect the worm and generate the self-certifying alerts (SCA) and then distribute the alerts to other hosts. Vigilante shows the dissemination of SCA can take up to seconds and its normal execution overhead is too high for practical widespread deployment. TaintCheck [17] based on Valgrind also suffers from high normal execution overhead.

Designed in response to the expensive instrumentation of Vigilante and TaintCheck, Sweeper picks address space randomization [3] as the lightweight runtime checker of its detection engine. During the normal execution, Sweeper only runs lightweight checkpoints and lightweight monitoring until an attack is detected. After an attack is detected, Sweeper rolls back and replays the attack in combination with dynamic taint analysis and dynamic backward slicing to analyze the vulnerability and generate the anti-body. As compared to Vigilante and Sweeper, CAWDOR does not intend to redesign or re-implement the whole worm system. Instead, it explores how much more the complex collaborative worm system can simply get out of the source code—-CAWDOR can always go with randomization or recovery technology of the Sweeper as an integral part of worm defense system. CAWDOR offers some properties which the above defense systems can not, the *predictability* and *controllability*, without sacrificing their desired properties of performance and accuracy. CAWDOR implements the SCA like concept with even simpler verifier since latent instrumentation can pre-generate the signature and thus lead to a trivial replay verification. Like the detection engine of Sweeper, CAWDOR also speculates the light mode is more usual for normal execution. Unlike Sweeper which combines two different monitoring mechanism and attack replay to make lightweight normal execution possible, CAWDOR keeps the same runtime-checker for both the detection and the analysis. Last but not least, Sweeper employs dynamic taint analysis to track the exploit input and then backward dynamic slicing to analyze the vulnerability. These two technologies may duplicate some track logic with each other. CAWDOR pre-slices the program and generates the run-time checker in advance. The detection and analysis are integrated and will not duplicate the application logic. it is easy to trace back to the root cause of the vulnerability by following the dependency chain.

### B. Runtime Checker

Program instrumentation and transformation have played important roles in enforcing security properties at run-time: RICH [5] is an integer overflow(underflow) checker. Dynamic

taint-tracking verifies the data input from tainted source. Squashy [4] tracks and erases sensitive information from crash reports. Jones and Kelly [10], CRED [18], CCured [7], MSCC(Memory Safety C Compiler) [26] and SoftBounds [15] are runtime checker for C memory access. The full discussion about C run-time checker is deferred to next session.

## III. Implementation

### A. CAWDOR

CAWDOR assigns each instrumentation site a unique index within a global bit-vector. The instrumentation at that site only executes if its corresponding entry in the vector is set. The transformation generates inter-instrumentations dependency and encodes their dependency as a data structure inside the resulting object file. The contents of the global bit-vector are loaded from a file at program startup. The run-time system uses the dependency information to enable all supporting checks after initializing the bit-vector from disk. Our transformation generates fast-path/slow-path code. Each function decides whether to run its fast path or its slow path upon entering the function. The function can also switch between fast and slow paths at the beginning and end of every loop. Thus, for example, if the function is executing along its slow path and reaches a loop that contains no active instrumentation, then the function will temporarily switch to the fast path for the duration of the loop. The optimal placement of switching points depends on the program's run-time behavior, so we chose the above heuristic since it is likely to give a good payoff for relatively few switching points. If a function contains no instrumentation, then we only generate a single path for it. As with dependencies between instrumentations, there are also control-flow dependencies between switch points and instrumentations. CAWDOR computes these dependencies at compile time and embeds them in the same dependency data structure as the inter-instrumentation dependencies. The run-time dependency resolution algorithm thus activates the correct set of switch points for any set of active instrumentations.

In this section we describe CAWDOR, the implementation of our prototype collaborative worm defense transformation. CAWDOR is implemented in the CIL(C intermediate language) [16]. Our prototype inserts latent checks to enforce memory safety, although the basic idea could be applied to transformations for enforcing other safety properties. CAWDOR follows the general approach of the Jones and Kelly[5] bounds checker, although we have made several enhancements described below.

*1) Jones and Kelly:* The Jones and Kelly transformation inserts instrumentation to maintain bounds information for each allocated object in the program. As long as a pointer points to a valid object, the bounds for the pointer can be obtained by looking up the bounds for the pointer's referent. When the program performs pointer arithmetic, the Jones and Kelly transformation inserts code to verify that the new pointer points to the same object as the pointer from which is derived. When the program dereferences a pointer, the Jones and Kelly transformation inserts code to look up its bounds and ensure that the dereference does not violate those bounds. Pointer assignments require no instrumentation. Finally, the transformation adds code to register bounds for new objects when they are allocated, and to remove bounds for objects when they are destroyed.

*2) CAWDOR bounds tracking:* Jones and Kelly bounds tracking strategy has several advantages. It supports linking transformed code with most untransformed code, such as system libraries. We describe an improvement that enables even greater interoperability between transformed and untransformed code. Jones and Kelly's bounds tracking organization also makes dependencies between instrumentations simpler. Since our project enables each instrumentation to be activated individually, it must handle any inter-check dependencies introduced by the transformation. Simple dependencies make this process easier.

Object-based bounds tracking has some disadvantages. The Jones and Kelly system stores object bounds in a splay tree. Lookups in the splay tree are quite slow, causing high overhead – programs can run over 10 times slower after being transformed. Even worse, pointers must always point to valid objects, since otherwise the system may not be able to lookup bounds for a pointer. Many real-world program violate this assumption, making the original Jones and Kelly system impractical.

Other program transformations, such as CCured or SoftBounds, use pointer-based bounds tracking. In these systems, bounds information is associated with the pointer, not the object to which it points. When a pointer is dereferenced, it is checked against its associated bounds. When one pointer is assigned to another, the associated bounds information is also copied. Pointer arithmetic does not require a check. Therefore, these systems support programs that generate pointers that go out of bounds, as long as the program does not dereference the pointer while it is out of bounds. It can also be faster to lookup a pointer's bounds in these systems – for example, SoftBounds uses a hashtable to store bounds. Pointer-based bounds tracking harms compatibility: Pointers passed from untransformed code to transformed code will not have any associated bounds information or lose track of updating the bounds information.

We use a hybrid bounds tracking scheme to achieve the library compatibility and simplicity of the Jones and Kelly approach and the efficiency and program compatibility of pointer-based approaches. We store bounds information for objects in a splay tree, and we cache this bounds information with pointers that point to the object. A pointer can go out of bounds as long as its bounds information is cached with the pointer. We now explain the CAWDOR bounds caching mechanism.

CAWDOR classifies every pointer expression in the program as either *solid* or *non-solid*. Solid pointers can have bounds caching, non-solid pointers cannot.

***Solid pointers***: Solid pointers are local variables (or parameters) that do not have their address taken. Solid pointers have the *strong update* property: all updates to the pointer must be

performed by assigning to the pointer itself. Thus, no weak update through its alias. For example, the pointer cannot be changed by an assignment through some other pointer, e.g.

```
int *p = ...; // Solid pointer
int **q = ...; // Does not point to p
...
*q = ...;  // Definitely does not modify p
```

CAWDOR creates a bounds variable for each solid pointer and inserts code to update the bounds variable whenever the pointer is updated in the original program. By the strong update property, CAWDOR can statically identify all the locations in the program that update the original pointer, so the bounds information will always be correct. Consequently, solid pointers can go out of bounds (but cannot be dereferenced while out of bounds), and bounds information can be obtained without an expensive splay tree lookup. When one solid pointer is assigned to another, the bounds variable of the first is assigned to the bounds variable of the other, as well. CAWDOR modifies functions so that they take bounds information for their solid pointer parameters.

***Non-solid pointers***: Non-solid pointers may not have the strong update property: it may be possible to change the pointer through its aliases without mentioning the pointer by name. CAWDOR uses object-based bounds information for these pointers. Whenever the program needs the bounds for one of these pointers, it looks up the information in a splay tree. Consequently, these pointers must always be in bounds. We conjecture that in most programs, only solid pointers go out of bounds. The benchmark results presented in Section V support this hypothesis.

When a solid pointer is assigned to a non-solid pointer, CAWDOR inserts a check to confirm that the solid pointer is in bounds. When a non-solid pointer is assigned to a solid pointer, the bounds information of the non-solid pointer is looked up and stored in the bounds variable for the solid pointer.

To sum up, CAWDOR bounds-tracking strategy reduces run-time overhead by storing and propagating pointer based bounds information to eliminate many lookups. By storing bounds information, CAWDOR can manipulate temporarily out of bounds pointers as long as the pointers are not dereferenced, which enables CAWDOR support most programs without the complexity of CRED OOB structures [18].

We now describe compatibility improvements and optimizations to this basic approach. We then describe the optimizer of CAWDOR.

### B. Compatibility

CAWDOR must support function calls between transformed and untransformed code. There are two key challenges: (1) untransformed code does not register bounds information for objects it allocates and, (2) untransformed code does not pass or expect bounds information.

Bounds passing with untransformed code is easy to handle. Whenever CAWDOR adds bounds parameters to a function, it changes the name of the function and creates a wrapper version of that function that does not take bounds information. The wrapper function has the same name as the original function. The wrapper uses the splay tree to look up any missing bounds information and calls the real function with the required bounds. Thus callers in other compilation units or in untransformed libraries can always call the original function by its original name without passing any bounds information. This ensures backwards compatibility. Whenever the program takes the address of a transformed function, CAWDOR rewrites this code to take the address of the wrapper, since the wrapper can be called anywhere the original function could. When a function in a transformed compilation unit calls a function outside that compilation unit, CAWDOR cannot determine whether the targeted function will be transformed or not. Therefore, CAWDOR calls the function without passing bounds information. The caller verifies that all pointer arguments are in bounds before making the call.

Programs can allocate memory in three places – on the stack, on the heap, and in their data or bss segments – so CAWDOR uses three strategies to register allocations performed by untransformed code. First, CAWDOR intercepts all calls to `malloc`, `free`, etc., to register the resulting object allocated on the heap. Second, CAWDOR intercepts the dynamic linker to register the data and bss segments of dynamically loaded libraries.

Stack allocations are slightly trickier. Whenever an untransformed function calls a transformed one, CAWDOR needs to register any objects allocated on the stack by the untransformed code. Making matters more complicated, the untransformed function may have been called by a transformed one, so CAWDOR needs to infer the exact range of stack space used by the untransformed code. Figure 2 shows how CAWDOR handles this situation. Whenever a transformed program calls a function in another compilation unit (which may or may not be transformed), it records the current stack pointer in a global variable. Whenever a wrapper function gets called, which will always be the case when untransformed code calls transformed code, it registers the entire stack between the saved stack pointer value and its current value. The wrapper deregisters this range before returning.

### C. Optimizations

CAWDOR includes optimizations to the data structures used to track bounds information and static analyses to remove unnecessary checks.

***Data structure optimizations*** Jones and Kelly use a splay tree to lookup bounds information because a splay tree supports range queries. Range queries are necessary because a pointer may point into the middle of an allocated region. However, this is relatively uncommon – most pointers point to the beginning of an allocated region. Therefore we maintain a hash table of the starts of allocated objects and attempt to lookup pointers in the hash table first. The hash table lookup is fast and likely to succeed. Only if it fails do we fall back to the splay tree.

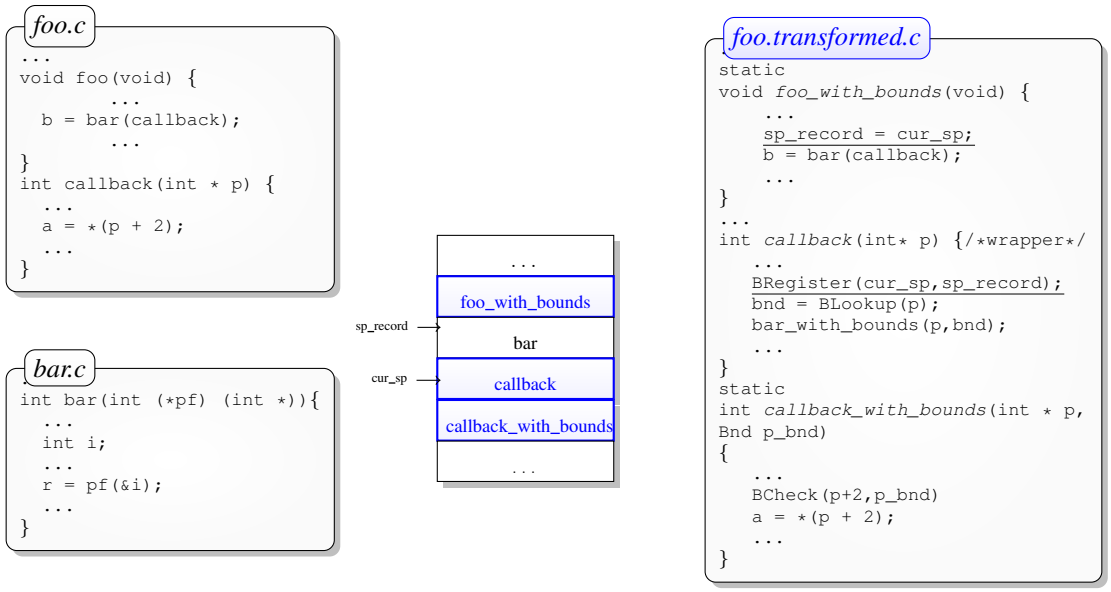


Fig. 2. An example of CAWDOR instrumented code interacting with un-instrumented code and the corresponding stack

Second, stack allocations are guaranteed to occur in decreasing address order, so we can maintain a simple sorted array of bounds information for stack allocated objects. New allocations get pushed onto the end of the array, and popped off when de-allocated. Bounds lookups on the stack can be accomplished with binary searchm, which may be slower than a splay tree lookup. However, most pointers into the stack are solid pointers, so stack bounds lookups are relatively rare. Overall, this trade-off is expected to be a net performance gain.

***Static analysis optimizations*** After inserting checks to track pointer bounds and to verify that all pointer dereferences are in bounds, CAWDOR uses static analyses to merge and eliminate unnecessary checks. All the optimizations are built around a simple theorem prover that can answer queries of the form $e_1 \leq e_2$. Such queries can be used to determine if one check is covered by another.

The optimizer recognizes when two adjacent checks can be converted into a single check. Checks are of the form $(p, s) \subseteq B$, where $p$ is a pointer, $s$ is the size of a dereference, and $B$ is the bounds the pointer must satisfy. If two adjacent checks are of the form $(p_1, s_1) \subseteq B$ and $(p_2, s_2) \subseteq B$, where the theorem prover can verify statically compute $\min(p_1, p_2)$ and $\max(p_1 + s_1, p_2 + s_2)$, then the optimizer will convert these two checks to a single check

$$(\min(p_1, p_2), \max(\max(p_1 + s_1, p_2 + s_2) - \min(p_1, p_2)) \subseteq B$$

CAWDOR also uses a loop analysis to move instrumentations outside of loops. The analysis can hoist both invariant checks and checks on affine loop variables, i.e. variables that change by a static constant on each iteration of the loop.

CAWDOR uses a CCured-like safe pointer analysis to eliminate bounds checks on singleton pointers. Safe pointers do not participate in any casts or pointer arithmetic They are therefore always NULL or safe to dereference. In the example in Figure 4, the `lp` local variable in `foo_with_bounds` is safe.

An escape analysis can help to reduce the unused registrations. If we can statically verify that all accesses to an allocated object are safe, then its bounds information will never be needed, so it does not need to be registered. An *undo-bounds-optimizer* can remove bounds arguments that are never used in the body of the function.

Most optimization for spacial dereference of MemSafe [19], including its *Non-incremental Dereference*, *Monotomically Address Ranges* and *Unused Metadata*, can be found correspondingly in the above optimization of CAWDOR. The *Dominated Dereference* of Memsafe is achieved by a more thorough and more aggressive control flow optimizer of CAWDOR called *HCheck Placement*.

Considering the following example:

```
if(cond1) {*p = 0;} else {*p = 1;}
q = p;
if(cond2) {*q=0;}
```

$check(q)$ is redundant as it is covered by any path which can reach $*q$ from $cond1$, though it is not dominated by any of them, which means the *Dominated Dereference Optimization* does not help with this case–not until the common factor $check(p)$ of both branches is hoisted over the condition and $check(q)$ is hypothetically hoisted over $q = p$.

The location, and logically equivalent replacement, of the inserted checks is very important to enable other optimizers and discover the redundant checks, which is not trivial even for the simple example as above. Considering another example:

```
if(cond2) {*p = 1; p++;
           for(i=0;i<n;i++)sum += a[i];
           *p = sum;}
else {*p=0;}
```

The optimizer has to figure out the better placement of $check(p+1)$ is ahead of the loop so that it can get merged with $check(p)$. However, if it does not merge with $check(p)$, $check(p)$ on both branches may be hoisted ahead of the $condition$. For either of the cases the static count of the checks would be two but the actual executions may be different. A good optimizer should be able to model the placements of both circumstances.

*HCheck Placement* aims to optimize the placement of checks. *HCheck Placement* uses *Hoare's assignment axiom* to hypothetically hoist checks up through the program's control flow graph and attempts to search the optimal solution of hypothetical checks set. Hoare's assignment axiom states that, if $P[e/x]$ is true before executing assignment $x := e$, then $P$ will be true afterwards. Thus, for example, the program

```
p = q;
assert (lo <= p && p <= hi);
```

can be rewritten as

```
assert (lo <= q && q <= hi);
p = q;
```

We define *base-check*s are the checks emitted by CAWDOR instrumentation. *hypo-Loc* for a check is a location at which the check could be hoisted up in conformity to *Hoare's assignment* without introducing any false positive. A *hypo-check* is the representation of a check when hoisted up to its *hypo-loc*. *HCheck Placement* algorithm has 4 steps.

(1). Compute the set of all program locations where each *hypo-check* could be placed using a backward data flow module in figure 3. The convergence is guaranteed because the value domain is of finite height, up to the set of all the checks. The transfer and joint functions are monotonic in terms of hypo-checks. In the implementation the iteration is based on each basic control block and bounds look-up is actually included as part of value domain in addition to bounds checking.

(2). Each program location now has a set of hypothetical checks. Compute redundancies among these checks (i.e. merge all the ones that can be merged). In the above example of loop, *hypo-check* on $p+1$ can merge with *hypo-check* on $p$.

(3). Compute logical implications between the *hypo-check*s. CAWDOR encodes the control flow graph information into the implication graph of hypo-checks. First of all, we map every *hypo-check* at each program location to a node called *hypo-node*. The structure of the node contains its id, the hypo-check and the program location. If there is an implication between two *hypo-check*s, then a path is drawn between two corresponding *hypo-node*s. If a program location $p$ has only one successor $s$ and a *hcheck(i,ptr)* appears in both of their hypo-checks set, then the *hcheck(i,ptr)* in both p and in s are logically implied by each other. The hypo-checks only in $p$ not in $s$ must have their *base-check*s in p. The hypo-checks only in $s$ not in $p$ must have been stuck in $p$ during hoisting. The $base$ and $stuck$ nodes, which indicate where the *base-check*s are initially placed and where the *hypo-checks* are stuck, will then

Initial:

$$Out[s] = \begin{cases} \emptyset, & if \text{ succs[s]=}\emptyset \\ \cup, & \text{otherwise} \end{cases}$$

Iteration:

$$\begin{cases} Out[s] = \cap_{p \in succs[s]} In[p]; \\ In[s] = \begin{cases} Checks(Out[s], s) & \text{if } s \text{ is a check} \\ Hoist^{\alpha}(Out[s], s) & \text{otherwise} \end{cases} \end{cases}$$

- $succs[s]$: the successors of s.
- $\cap(ck_1, ck_2) = \begin{cases} ck_1, & if (bounds, ptr, size)_{ck_1} = (bounds, ptr, size)_{ck_2} \\ \varnothing, & otherwise \end{cases}$
- $Checks(d, s) = \begin{cases} d, & \exists s' \in d, id_{s'} = id_s \wedge ptr_{s'} = ptr_s \\ d - s, & \exists s' \in d, id_{s'} = id_s \wedge ptr_{s'} ! = ptr_s \\ d \cup \{s\}, & otherwise \end{cases}$
- $Hoist^{\alpha}(d, s) = \{ck_i' \mid \forall (x := e) \in Effect(s), ck_i \in d \wedge \alpha(ck_i) \cap Effect(s) = \emptyset, ck_i' = ck_i[e/x]\}$ in which $\alpha$ maps the pointer of a check to its alias set.

Fig. 3. *hypo-check*s computation

be generated and connected. If program location $p$ has multiple successors $s_1, s_2, ...s_i, ...$, and a hypo-check *hcheck(i,p)* shows in the hypo-checks set of all $s_i$ and $p$, then *hcheck(i,p)* in p shall logically imply the corresponding *hcheck(i,p)*s in all $s_i$. Joint program location works the other way around.

```
if(cond1) {*p=0;q=r}
 else {*p=1;q=r+1} *q=-1;
```

$if$ statement has *hcheck(i,p)*, $\{*p = 0; q = r\}$ has *hcheck(i,p)* and *hcheck(j,r)*, $\{*p = 0; q = r + 1\}$ has *hcheck(i,p)* and *hcheck(j,r+1)* and $*q = -1$ has *hcheck(j,q)*. Therefore the *hcheck(i,p)* on the $if$ statement can imply the *hcheck(i,p)* on both branches. *hcheck(j,q)* implies both *hcheck(j,r+1)* and *hcheck(j,r)* but *hcheck(j,r+1)* has to be combined with *hcheck(j,r)* to imply *hcheck(j,q)*.

(4). Brute force search for smallest set of *hypo-check*s that logically imply the *base-check*s. We heuristically set the least static checks count as the optimal criteria, however, the model is open to other heuristic scheme, for instance, to minimize the checks on busiest path for the worst execution case. We sort the hypo-checks in an ascending order of distance to the start of the function and run a brute-force search. Checks closer to the top of the function has some advantages: the closer to definitions of the variables upon which they depend, the higher chance for the theorem prover to statically verify them. This can also bring related more checks together, enabling the optimizer to merge them into a single check. In order to reduce the searching space, CAWDOR splits the implication graph into disjoint sets of connected components. CAWDOR implements some pruning algorithms to compact the implication graph, including reducing the self-contained loop nodes chain into one node and shrinking the straight-line nodes. Empirical experiment on benchmark shows *HCheck Placement* only accounts for a small part of compilation time compared to the whole program pointer analysis.

CAWDOR also includes two inter-procedural optimizations. The inter-function optimizer hoists a check at the top of a function body to all the call sites of this function, including

**Before Optimizations**

```
1  /*Dependencies:(1, 7);(4, 3);(3, 8);(8, 9);(2, 1)
2  (6, 1);(5,11);(10,11);(11,-1);(-1,8);(-1,1)*/
3  int * gp;
4  int foo_with_bnd(int n, int * a, Bound B_a) {
5     Bound B_lp, B_t;
6     int t ;
7     int *lp = &t;
8     BCheck(B[6], B_a, a, Sizeof(*(a+0))*n);
9     B_gp = BLookup(B[11],gp);
10    B_t = BRegister(B[8], &t, B_t, Sizeof(t));
11    B_lp = BAssign(B[3], B_t);
12    BCheck(B[4], B_lp, lp, Sizeof(*lp));
13    *lp  = -1;
14    BCheck(B[5], gp, B_gp, sizeof(*gp));
15    if(*gp==0)  {
16        BCheck(B[10], B_gp, gp, sizeof(*gp));
17        *gp = -1;
18     }
19    BCheck(B[6], B_a, a, Sizeof(*(a+0))*n);
20     for(int i=0;i<n;i++) {
21          BCheck(B[6], B_a, a+i, Sizeof(*(a+i));
22           a[i] = 0;
23     }
24    BDeregister(B[9], &t);
25     return 0;
26 }
27    int bar (void) {
28      Bound B_arr1;
29      int arr1[5];
30      int arr2[6];
31      BRegister(B[1],arr1, B_arr1, 5*int);
32      BCheck(B[2], B_arr1, arr1, 1bit);
33      gp = arr1;
34      BCheck(B[6], B_arr1, arr1, Sizeof(int)*5);
35      foo_with_bnd(5, arr1, B_arr1);
36      BDeregister(B[7],arr1);
37   }
```

**After Optimizations**

```
1     /*Dependencies: (1, 7);(5,11);(11,-1);(-1,1) */
2      int * gp;
3      int foo_with_bounds(int n, int * a) {
4           int t ;
5           int *lp = &t;
6           B_gp = BLookup(B[11],gp);
7           BCheck(B[5], gp, gp, sizeof(*gp));
8           *lp  = -1;
9           if(*gp==0)  {
10           *gp = -1;
11          }
12           for(int i=0;i<n;i++)
13                a[i] = 0;
14           return 0;
15     }
16    int bar (void) {
17      Bound B_arr1;
18      int arr1[5];
19      int arr2[6];
20      BRegister(B[1],arr1,B_arr1, 5*int);
21      gp = arr1;
22      foo_with_bounds(5, arr1);
23      BDeregister(B[7],arr1);
24   }
```

Fig. 4. CAWDOR intermediate output before and after optimizations. In blue are solid, in redare non-solid, underlined are inserted instrumentations. In *italic gray* are hypothetical instrumentations of *BAction 6* along the optimization passes

its wrapper function. The intra-procedural optimizer can then attempt to eliminate the check from the callers. For example, in Figure 4, once the hypothetical check on *a* reaches line 8, it can be hoist ahead of *foo* (line 34), then the intra-procedural optimization will kick in and finally the static checker will get rid of it.

Figure 4 gives an example of the impact of the CAWDOR optimizations. The only check remaining after optimizations, *BCheck 5*, is for non-solid pointer *gp*. *lp* is a solid pointer so CAWDOR could always cache bounds for *lp*. However, *gp* is weak-updated because it is a global variable subject to change anytime. Aiken et al. [1] has pointed out with pointer analysis, we can infer the *restricted* scope of non-solid pointers in which non-solid pointers do not have any local alias. In another word, within their *restricted* scope, non-solid pointers are the only way to access their referents. CAWDOR then generates temporal bounds variables for non-solid pointers and apply similar optimization within the *restricted* scopes. Hence any check on *gp* is on its temporary bounds variable which is looked up at the beginning of its restricted scope and any its value in and out its restricted scope has to be checked in bounds first (Figure 4 (top) line 32). The transformation modifies *foo* to expect bounds as additional parameters though later on the optimizer removes this bounds-passing, since it is unneeded by the optimized callee.

For the example in Figure 4 (top), three instrumentations (line 10-12) on *lp* and line 32 are removed by the peephole and static checker. The check on *a[i]* within loop will be hoisted out as a check on *(a,a+n*sizeof(*a))* at line 19. The check hoister will indicate that it can be moved to the top of function body(line 8). Now the *inter-procedure-optimizer* takes over and pulls it to the call site(line 34), which feeds off other optimizers. The *BAction 10* is optimized out because it is covered by *BAction 5*.

These optimizations are described separately but they are integrated to work together. We split the optimization phase from core transformation and run each optimizer in turn. We repeat the optimization phase until the program reaches a fixed point. The optimizers all operate independently, but one optimization my create a state that will allow another optimization to proceed where previously it could not.

## IV. Application: Collaborative Worm Defense Systems

The key idea of applying CAWDOR to worm defense system is that anti-bodies can be generated in advances. Sentinel hosts can perform worm detection by activating a random subset of the latent checks. When a worm begins spreading through the Internet, it will eventually hit a sentinel host that is monitoring the vulnerability exploited by the worm. At that point, the sentinel will have detected the worm and will know precisely how to mitigate it: activate the failed check. It can then disseminate this information to other hosts, who can verify the alert by replaying the worm input with the indicated check enabled. If the check fails, then they know the alert is legitimate and can leave the check enabled to remain protected against future attacks.

An exploitable underflow vulnerability in ncompress/decompress42.c, reported in CVE-2006-1168 is transformed in Figure 5. The while loop performs zero bounds check, which is subject to a global segment underflow.

When a worm triggers this check in a sentinel host in a worm defense system, the host will instantly know the location of the bug and the mitigation that other hosts should

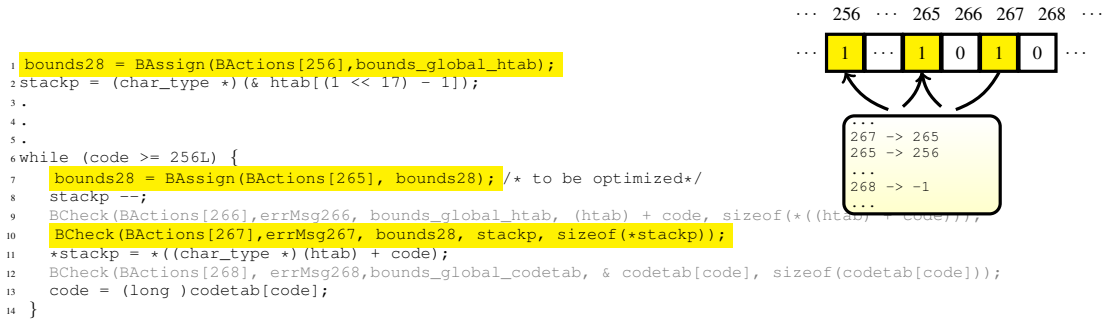


Fig. 5. The example of latent checks and inter-instrumentation dependency. The instrumentations activated to catch the bug are highlighted and the irrelevant checks disabled are in gray.

| Program | Type | Detection | | | |
|---|---|---|---|---|---|
| | | Purify | Valgrind | CCured | CAWDOR |
| gzip-1.2.4 | GO | Y | Y | Y | Y |
| bc-1.06 | HP | Y | Y | Y | Y |
| polymorph-0.4.0 | ST | Y | P | Y | Y |
| ncompress-4.2.4 | GO | N | N | Y | Y |
| tpop3d-1.5.5 | ST | - | - | - | Y |

TABLE I
EFFECTIVENESS OF DETECTING VULNERABILITIES. Y = YES. N = NO. P = PARTIALLY. -= NOT AVAILABLE GO=GLOBAL SEGMENT OVERFLOW; HP = HEAP BUFFER OVERFLOW; ST = STACK BUFFER OVERFLOW

apply: they should also enable check 267. Thus, our compiler infrastructure makes the analysis phase of a worm defense system trivial.

## V. EVALUATION

In this session, we experimentally evaluate CAWDOR's (1) effectiveness, (2) correctness, and (3) overhead. An effective defense system should catch attempts to exploit programs. A correct defense system should not break working programs. We evaluate CAWDOR's overhead under several different scenarios: during detection, defense, and testing.

### A. Effectiveness

We evaluated CAWDOR's effectiveness on BugBench [13] and tpop3d. BugBench provides several programs – *ncompress*, *bc*, *polymorph*, and *gzip* – with memory safety bugs, along with inputs for triggering those bugs. These programs include a variety of buffer overflow against stack, heap and global segments. We tested CAWDOR in two modes: with all checks activated, and with the only the crucial check required to catch the bug enabled. CAWDOR was able to catch all the bugs in our benchmark programs, both in all-checks-on mode and in crucial-check-only mode. We compare the detection ability of CAWDOR with some other C run-time checkers in the Table I. tpop3d is not part of BugBench, but we discovered a previously unknown bug in it while testing CAWDOR, so we include it here.

### B. Correctness

Apart from the effectiveness enabled by all-checks-on and crucial-check-on, we have to verify the correctness of CAWDOR with arbitrary subsets of the checks enabled. Since memory errors taken from real-world programs can be brittle and highly-dependent on architecture and compiler details, we have chosen to evaluate the correctness and security of our transformation using a suite of simple test programs to cover comprehensive overflows on stack, heap and global segments. Each test program accepts a command-line argument indicating whether it should execute code with a memory violation. The test programs are designed so that, when compiled with a normal compiler, the memory access violation are all silent and harmless. We run each of these programs through the CAWDOR transformation and verify that the resulting executables all satisfy the following requirements

- With all checks disabled and with no memory access violation, the program executes normally.
- When all checks are enabled and no memory access violation occurs, the program completes execution normally.
- When all checks are enabled and a memory access violation occurs, the program aborts on a run-time check.
- We then turn off all checks except the failing check from the previous test (and any supporting checks), and re-run the program with the same bad input. It must abort as before.
- Finally, we run the program several times with random subsets of 10%, 20%, 30%, and 40% of the run-time checks enabled, but with no memory access violation, and confirm that the program always executes normally.

Our test-suite includes hand-written tests and tests derived from programs written by the authors for un-related projects. All tests, including buffer overflow tests targeting function pointers, de-allocated pointers, variable arguments, pointer parameters and arguments pass, cover most attacking categories [25]. The benchmarks used for the performance analysis described below serve as further evidence of the correctness of our transformation.

The sentinel host can always test the software with all checks on in advance, which will ensure that there are no

| Benchmark | LOC | Gcc time | CIL ratio | MSCC ratio | CCured ratio | J & K ratio | CAWDOR Ratio All Off | CAWDOR Ratio All On |
|---|---|---|---|---|---|---|---|---|
| bh | 2010 | 1.03 | 1.00 | 2.82 | 1.44 | 32.4 | 1.06 | 2.79 |
| bisort | 351 | 1.28 | 1.01 | 1.76 | 1.45 | 18.3 | 1.00 | 1.02 |
| em3d | 590 | 2.22 | 1.03 | 1.79 | 1.87 | - | 1.02 | 2.61 |
| health | 710 | 4.12 | 1.03 | 2.72 | 1.29 | - | 1.04 | 1.04 |
| perimeter | 399 | 0.8 | 1.00 | 3.37 | 1.09 | 10.4 | 1.01 | 1.01 |
| power | 760 | 0.48 | 1.00 | 1.22 | 1.07 | - | 0.98 | 1.00 |
| treeadd | 377 | 2.32 | 1.09 | 3.23 | 1.10 | 7.5 | 1.05 | 1.06 |
| tsp | 583 | 0.93 | 1.01 | 2.28 | 1.15 | 9.0 | 1.02 | 1.01 |
| mst | 482 | 1.36 | 1.00 | 1.76 | 1.06 | 22.4 | 1.07 | 2.78 |
| *Average* | - | - | **1.01** | **2.33** | **1.30** | **16.7** | **1.03** | **1.59** |
| anagram | 650 | 2.27 | 1.00 | - | 1.43 | - | 1.07 | 1.26 |
| yacr2 | 3971 | 0.3 | 1.17 | - | 1.56 | - | 1.13 | 2.8 |
| bc | 7299 | 0.55 | 0.98 | - | 9.91 | - | 0.98 | 3.5 |
| ft | 2001 | 3.27 | 0.99 | - | 1.03 | - | 1.05 | 1.12 |
| ks | 782 | 2.52 | 1.11 | - | 1.11 | - | 1.13 | 1.66 |
| *Average* | - | - | **1.05** | - | **3.01** | - | **1.07** | **2.07** |
| mcf | 2512 | 12.46 | 1.03 | 2.85 | - | - | 1.08 | 4.5 |
| milc | 15042 | 35.7 | 1.02 | - | - | - | 1.02 | 1.71 |
| sjeng | 13847 | 9.02 | 1.00 | - | - | 25.7 | 1.08 | 4.05 |
| *Average* | - | - | **1.02** | - | - | - | **1.06** | **3.4** |
| ncompress | 1935 | 11.05 | 1.04 | - | - | - | 1.11 | 1.73 |
| tpop3d | 16726 | 0.42 | 1.02 | - | - | - | 1.05 | 3.4 |
| gzip | 8189 | 2.7 | 1.07 | - | - | 9.9 | 1.05 | 1.72 |
| gunzip | | 0.49 | 1.00 | - | - | - | 1.03 | 1.9 |
| *Average* | - | - | **1.04** | - | - | - | **1.06** | **2.19** |
| **Average** | **-** | **-** | **1.03** | **2.38** | **1.91** | **17.01** | **1.05** | **2.08** |

TABLE II
OVERHEAD OF CAWDOR TRANSFORMATION, COMPARED WITH THE REPORTED OVERHEAD FROM OTHER C RUN-TIME CHECKERS [19], [15]

latent buffer overflows uncovered by a later check activation. The correctness of CAWDOR should ensure that the program will work with any subset of checks enabled.

### C. Performance

The overhead of normal execution without any known vulnerabilities, which is the most common scenario for real applications, as Sweeper [23] pointed out, is one of the most important concerns for a worm defense to be deployed widely. CAWDOR can disable all the checks during normal execution. Occasionally, the host will have a single check activated, along with its supporting checks, on hearing of a known vulnerability in the application from sentinel hosts. This overhead should be as low as possible, but it is less important than overhead with all checks off. The heavyweight monitoring enabled by full checks on is useful when the programmer or sentinel hosts want to test prior to deployment. This overhead is not too important unless it is too high for the application to be tested off-line. Finally, the administrator may wish to trade off between run-time overhead and risk. Different proportions of enabled checks might be set for specific scenarios and different security levels. Thus, we measure the overhead of CAWDOR transformation in these five configurations: all checks off, one check enabled, 5% checks enabled, 10% checks enabled and all checks enabled.

Table II shows the overhead of our transformation on the programs from the Olden, Ptrdist and SPEC2006 benchmarks on two important modes: all checks disabled and all checks enabled. The transformation appears to incur slight overhead (about 5%) when all checks are disabled. Relative to the base overhead of CIL, all checks-off CAWDOR incurs negligible performance penalty.

A few well known C memory runtime checkers are not in Table II. Baggy (around 60% overhead) is so far the fastest bounds checker [2]. It changes the OOB scheme to perform efficient bounds checking but, as a result, some benchmarks require a few manual changes to the source code. SoftBounds and its variation MemSafe [19] (67% to 87% overhead), provide good compatibility and separate meta-data maintenance. Many of the optimization and implementation ideas in CAWDOR could also be applied to a SoftBounds implementation. The all checks activated mode of CAWDOR (108% overhead) is not the best among all the bounds checker but comparable to CCured and MSCC. Considering it is based on Jones and Kelly approach, we believe this is a significant progress over Jones and Kelly, which on average, slows down application by a factor of 10.

Figure 6 show the overheads when a single check, 5% of checks, or 10% of checks are activated in our benchmark programs. Since the overhead varies depending on which check is activated, we ran each benchmark in Olden and Ptrdist 100 times and in SPEC 50 times, activating a randomly chosen set of checks on each execution. We report the average execution time ratio, along with error bars indicating the 95% confidence interval. Overheads can vary significantly depending on the activated check, but on average the overhead is around 9%(1 check), 29%(5% checks) and 41%(10% checks). Although 5% checks and 10% checks are moderately high overhead, the benefit is that, for most of the hosts, in the common execution and the occasionally crucial mode, we just need to run all-checks-off and one check enabled, the overhead are much lower, less than 10%.

## VI. CONCLUSION

We show how compiler support can assist the creation of an Internet worm defense system. By using the compiler to pre-generate mitigations for future vulnerabilities, we can make a worm defense system that is faster, simpler, and more predictable manageable. Our compiler support system can help with the detection, analysis, verification, and application tasks in a worm defense system. Without loss of flexibility, performance or detection accuracy, our system uses the same run-time checker for both lightweight monitoring and heavyweight monitoring enabled by a fine instrumentation activation.

We present CAWDOR, a prototype compiler that transforms C program to participate in a collaborative worm defense system. Transformed programs incur low run-time overhead. CAWDOR has lower overhead and better compatibility improves than the Jones and Kelly memory safety transformation. Our prototype also proved effective at detecting memory safety violations – it caught all the violations we tested.

## ACKNOWLEDGEMENTS

The material is based upon work supported by the Department of Energy under Award Number DE-OE0000220. This report was prepared as an account

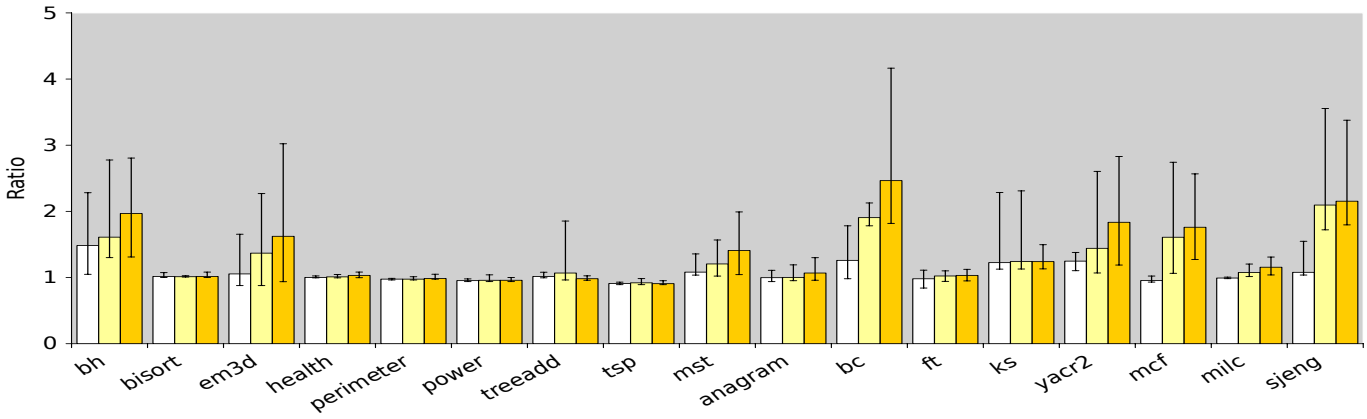


Fig. 6. CAWDOR overhead with respectively one check, 5% checks and 10% checks enabled.

of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacture, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

## REFERENCES

[1] Alex Aiken, John Kodumal, Jeffrey S. Foster, and Tachio Terauchi. Checking and inferring local non-aliasing, 2003.

[2] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven H. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors.

[3] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the* $12^{th}$ *USENIX Security Symposium*, pages 105–120, August 2003.

[4] Pete Broadwell, Matt Harren, and Naveen Sastry. Scrash: a system for generating secure crash information. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 19–19, Berkeley, CA, USA, 2003. USENIX Association.

[5] David Brumley, Tzi cker Chiueh, Robert Johnson, Huijia Lin, and Dawn Song. Rich: Automatically protecting against integer-based vulnerabilities. In *Proceedings of the 14th Annual Network & Distributed System Security Symposium*, 2007.

[6] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[7] J. Condit, M. Harren, S. McPeak, and etc. Ccured in the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, June 2003.

[8] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: end-to-end containment of internet worms. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 133–147, New York, NY, USA, 2005. ACM.

[9] Arik Hesseldahl and Olga Kharif. Cyber crime and information warfare: A 30-year history. http://images.businessweek.com/ss/10/10/1014_cyber_attacks/1.htm.

[10] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In M.K. Kamkar and D. Byers, editors, *Proceedings of the Third International Workshop on Automated Debugging*, volume 2–9 of *Linköping Electronic Articles in Computer and Information Science*, pages 13–26, Linköping, Sweden, May 1997. Linköping University Press.

[11] Hyang-Ah Kim and Brad Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proceedings of the USENIX Security Conference*, 2004.

[12] Robert Lemos. 'slammer' attacks may become way of life for net. http://news.com.com/Damage+control/2009-1001_3-983540.html, February 2003.

[13] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *In Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[14] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the slammer worm. *IEEE Security and Privacy*, 1:33–39, 2003.

[15] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. In *PLDI*.

[16] George C. Necula, Scott McPeak, Westley Weimer, Raymond To, and Aman Bhargava. Cil: Infrastructure for c program analysis and transformation. http://www.cs.berkeley.edu/ necula/cil, 2002.

[17] James Newsome. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[18] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *NDSS*, 2004.

[19] Matthew S. Simpson and Rajeev Barua. Memsafe: Ensuring the spatial and temporal memory safety of c at runtime. In *Tenth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, 2010.

[20] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated Worm Fingerprinting. In *OSDI*, 2004.

[21] Alexey Smirnov and Tzi cker Chiueh. Dira: Automatic detection, identification and repair of control-hijacking attacks. In *In NDSS*, 2005.

[22] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to 0wn the Internet in your spare time. In *USENIX Security*, 2002.

[23] Joseph Tucek, Newso James, Shan Lu, Chengdu Huang, Spiros Xanthos, David Brumley, Yuanyuan Zhou, and Dawn Song. Sweeper: a lightweight end-to-end system for defending against fast worms. EuroSys, 2007.

[24] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of ACM SIGCOMM*, August 2004.

[25] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *IN NDSS*, 2003.

[26] Wei Xu, Daniel C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of c programs. *SIGSOFT Software Engineering Notes*, 29(6):117–126, 2004.