Bouncer: Securing Software by Blocking Bad Input

Manuel Costa
Microsoft Research
Cambridge, UK
manuelc@microsoft.com

Miguel Castro
Microsoft Research
Cambridge, UK
mcastro@microsoft.com

Lidong Zhou
Microsoft Research
Mountain View, USA
lidongz@microsoft.com

Lintao Zhang
Microsoft Research
Mountain View, USA
lintaoz@microsoft.com

Marcus Peinado
Microsoft
Redmond, USA
marcuspe@microsoft.com

Abstract

Attackers exploit software vulnerabilities to control or crash programs. Bouncer uses existing software instrumentation techniques to detect attacks and it generates filters automatically to block exploits of the target vulnerabilities. The filters are deployed automatically by instrumenting system calls to drop exploit messages. These filters introduce low overhead and they allow programs to keep running correctly under attack. Previous work computes filters using symbolic execution along the path taken by a sample exploit, but attackers can bypass these filters by generating exploits that follow a different execution path. Bouncer introduces three techniques to generalize filters so that they are harder to bypass: a new form of program slicing that uses a combination of static and dynamic analysis to remove unnecessary conditions from the filter; symbolic summaries for common library functions that characterize their behavior succinctly as a set of conditions on the input; and generation of alternative exploits guided by symbolic execution. Bouncer filters have low overhead, they do not have false positives by design, and our results show that Bouncer can generate filters that block all exploits of some real-world vulnerabilities.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; D.4.5 [**Operating Systems**]: Reliability; D.4.8 [**Operating Systems**]: Performance

General Terms

Security, Reliability, Availability, Performance, Algorithms, Design, Measurement

Keywords

Precondition slicing, Symbolic execution

1. INTRODUCTION

Attackers exploit software vulnerabilities to crash programs or to gain control over their execution. This is a serious problem because there are many vulnerabilities and attacks are frequent. We describe Bouncer, a system that prevents attacks by dropping exploit messages before they are processed by a vulnerable program. Bouncer introduces low overhead and it allows programs to keep running correctly even when under attack.

Several techniques detect (potentially unknown) attacks by adding checks to programs: safe languages include checks to ensure type safety and they throw exceptions when the checks fail (e.g., Java and C#), and checks can be added transparently to programs written in unsafe languages (e.g., [4, 12, 13, 16, 17, 28, 40]). The problem is that these techniques detect attacks too late when the only way to recover may be to restart the program. For example, CRED [40] adds bounds checks to prevent buffer overflows in C programs. These prevent the attacker from gaining control over the execution, but how do we recover when an attack causes a bounds check to fail? In the absence of additional mechanism, restarting the program is the only option because the program does not include any code to recover from the failure. This problem is not exclusive to unsafe languages. Even though out-of-bounds exceptions are part of type safe languages, programs frequently lack correct code to handle run time exceptions [48]. Therefore, these techniques are not sufficient. They leave services vulnerable to loss of data and denial of service.

Bouncer improves the reliability and availability of programs under attack. It uses previous techniques to detect attempts to exploit a vulnerability and it generates filters automatically to match messages that can exploit the vulnerability. The filters are deployed automatically by instrumenting system calls to run the filter on incoming messages and to drop exploit messages before they are delivered to the vulnerable program. We designed the system to ensure that these filters have low overhead and no false positives, that is, they only drop messages that can exploit the vulnerability. Since most programs can cope with message losses and filters have no false positives, Bouncer allows programs to keep working correctly and efficiently even when they are attacked repeatedly.

Bouncer builds on Vigilante's [16] technique to generate filters automatically. It computes an initial set of filter conditions using symbolic execution along the path followed by the program when processing a sample exploit. It assigns symbolic values b_0, b_1, b_2, b_3... to the bytes in the exploit messages, and keeps track of symbolic state for the processor and memory. For example, if `input` points to a buffer with the exploit bytes, register `eax` has symbolic value $b_0 + 1$ after executing `movzx eax, input; add eax, 1`. Whenever a conditional branch is executed, we add a condition to the filter to ensure that inputs that satisfy the filter conditions follow the same execution path. Continuing with the previous example, if `cmp eax, 2; jg target` is executed and the branch is taken, we add the condition $b_0 + 1 > 2$ to the filter. This technique guarantees no false positives: any input that satisfies the filter conditions can make the program follow the same execution path as the sample exploit. These filters block many variants of the sample exploit, but attackers can bypass them by generating exploits that follow a different execution path.

Bouncer introduces three practical techniques to generalize the initial filter to block additional exploits of the same vulnerability:

- *Precondition slicing* is a new form of program slicing [49] that uses a combination of static and dynamic analysis to remove unnecessary filter conditions.
- *Symbolic summaries* generalize the conditions captured by the symbolic execution inside common library functions. They replace these conditions by a succinct set of conditions that characterize the behavior of these functions for a broader set of inputs. These summaries are generated automatically from a template that is written once for each library function.
- *Generation of alternative exploits* guided by symbolic execution. Bouncer uses the initial exploit message and the conditions obtained from symbolic execution to derive new input messages that are likely to exploit the same vulnerability. It uses existing techniques to check if the new input messages are valid exploits, and it computes a new set of filter conditions for each new exploit. The final filter is a combination of the filters obtained for each exploit.

We implemented Bouncer and evaluated it using four vulnerabilities in four real programs: `Microsoft SQL server`, `ghttpd`, `nullhttpd`, and `stunnel`. The results show that Bouncer significantly improves the coverage of Vigilante filters, and that filters introduce low overhead.

Computing a filter that blocks exactly the set of messages that can exploit a vulnerability is similar to computing weakest preconditions [15, 20], which is hard for programs with loops or recursion [50]. Since we guarantee zero false positives, we do not guarantee zero false negatives, that is, Bouncer filters may fail to block some exploits of the target vulnerability. But our initial results are promising: a detailed analysis of the vulnerable code shows that Bouncer can generate filters with no false negatives for the vulnerabilities in `SQL server` and `stunnel`.

The rest of the paper is organized as follows. Section 2 provides an overview of Bouncer. Section 3 describes how Bouncer computes an initial set of filter conditions using symbolic execution. Section 4 describes filter refinement with improved detector accuracy. Section 5 presents precondition slicing. Section 6 describes symbolic summaries. Section 7 describes the procedure to search for alternative attacks. Section 8 presents the results of our evaluation. Section 9 discusses related work and Section 10 concludes.

```
ProcessMessage(char* msg) {
  char buffer[1024];
  char p0 = 'A';
  char p1 = 0;

  if (msg[0] > 0)
    p0 = msg[0];

  if (msg[1] > 0)
    p1 = msg[1];

  if (msg[2] == 0x1) {
    sprintf(buffer, "\\servers\\%s\\%c", msg+3, p0);
    StartServer(buffer, p1);
} }
```

Figure 1: Example vulnerable code: `sprintf` can overflow `buffer`.

2. OVERVIEW AND BACKGROUND

Vulnerabilities in the context of this paper are program errors that an attacker can exploit to make the vulnerable program deviate from its specification. An attacker exploits a vulnerability by crafting input that causes the program to behave incorrectly. We call such an input an exploit.

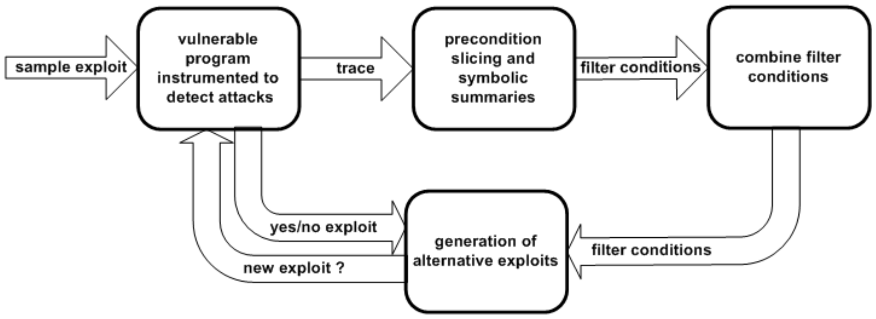
Figure 1 shows a vulnerable code fragment that we will use as an example throughout the paper. This is in C for clarity but Bouncer works with binary code. The function `ProcessMessage` is called immediately after the message `msg` is received from the network. This function has a vulnerability: exploit messages can cause it to overflow `buffer` in the call to `sprintf`. The attacker can exploit this vulnerability to overwrite the return address on the stack, which can cause the program to crash or execute arbitrary code. There are usually many exploits for a vulnerability, for example, any message with the third byte equal to 0x1 followed by at least 1013 non-zero bytes is a valid exploit for this vulnerability.

Bouncer protects programs by generating filters that drop exploits before they are processed. Figure 2 provides an overview of Bouncer's filter generation architecture.

Filter generation starts with a sample exploit that identifies a vulnerability. We obtain a sample exploit by running a version of the vulnerable program instrumented to log inputs and to detect attacks. When an attack is detected, the exploit messages are retrieved from the log [16] and sent to Bouncer. The current prototype uses DFI [12] to detect attacks on C and C++ programs but it would be possible to use other detectors (e.g., [28, 4, 17, 40, 37, 16, 13]) or to apply our techniques to programs written in safe languages.

DFI detects memory safety violations, for example, format string vulnerabilities, buffer overflows, accesses through dangling pointers, reads of uninitialized data, and double frees. For each value read by an instruction in the program text, DFI uses static analysis to compute the set of instructions that may write the value. At runtime, it maintains a table with the identifier of the last instruction to write to each memory location. The program is instrumented to update this table before writes, and reads are instrumented to check if the identifier of the instruction that wrote the value being read is an element of the set computed by the static analysis. If it is not, DFI raises an exception. DFI has low overhead because most instrumentation can be optimized away with static analysis, and it has no false positives: it only raises exceptions when memory safety is violated.

For our example, we will use a sample exploit message that



sample exploit
vulnerable program instrumented to detect attacks
trace
precondition slicing and symbolic summaries
filter conditions
combine filter conditions
yes/no exploit
new exploit ?
generation of alternative exploits
filter conditions

Figure 2: Bouncer architecture.

starts with three bytes equal to 0x1 followed by 1500 non-zero bytes and byte zero. Processing this message causes DFI to throw an exception when `p1` is accessed to set up the call stack for `StartServer` because `p1` has been overwritten.

The messages in the sample exploit are sent to a version of the vulnerable program that is instrumented both to detect attacks and to generate an execution trace (see Figure 2). The current prototype uses DFI to detect attacks and Nirvana [8] to generate an execution trace. If the sample exploit is valid, the execution trace is sent to the module that runs the precondition slicing algorithm. This trace contains the sequence of x86 instructions executed from the moment the first message is received to the point where the attack is detected. We call the instruction where the attack is detected the *vulnerability point*. In our example, the trace contains the instructions up to the call to `sprintf`, the instructions inside `sprintf`, and the remaining instructions up to the vulnerability point, which is the push of `p1` onto the stack.

The module that runs the precondition slicing algorithm (see Figure 2) uses the same technique as Vigilante [16] to generate an initial set of conditions for the filter. This technique replaces the concrete value of each byte in the sample exploit by a symbolic value b_i and performs forward symbolic execution along the trace of x86 instructions. It adds a condition to the filter for each branch that depends on the input. The initial set of conditions for our example is:

$b_0 > 0 \wedge b_1 > 0 \wedge b_2 = 1 \wedge b_{1503} = 0 \wedge \forall_{2<i<1503} b_i \neq 0$

The vulnerable program is guaranteed to follow the execution path in the trace when processing any message that satisfies the initial filter conditions. Therefore, this filter can be used to drop exploit messages without introducing false positives. However, the attacker can craft exploits that are not dropped by this filter because there are some conditions that are not necessary to exploit the vulnerability. For example, the conditions on b_0 and b_1 are not necessary and exploits with both shorter and longer sequences of non-zero bytes starting at index three can exploit the vulnerability.

Bouncer replaces the conditions generated for some library functions, like `sprintf` in our example, by *symbolic summaries* that contain exactly the conditions on the function arguments that cause it to violate memory safety. These summaries are generated automatically from a template that is written once per library function. In our example, Bouncer performs static analysis and determines that `buffer` has size 1024 bytes, and that any sequence with at least 1013 non-zero bytes pointed to by `msg+3` will lead to a memory safety violation independent of the value of `p0`. Therefore, the filter conditions after this step are:

$b_0 > 0 \wedge b_1 > 0 \wedge b_2 = 1 \wedge \forall_{2<i<1016} b_i \neq 0$

After adding symbolic summaries, precondition slicing uses a combination of static and dynamic analysis to remove unnecessary conditions from the filter. In our example, it is able to remove the conditions on bytes b_0 and b_1 producing the optimal filter:

$b_2 = 1 \wedge \forall_{2<i<1016} b_i \neq 0$

In general, the filters produced after the first iteration are not optimal. Bouncer generalizes these filters by repeating the process with alternative exploits of the same vulnerability that cause the program to follow different execution paths. The filter conditions are sent to the module that generates alternative exploits. This module uses the sample exploit and the conditions to generate new input messages that are likely to exploit the same vulnerability. Then, it sends these messages to the instrumented vulnerable program to check if they are valid exploits. If they are, the process is repeated with the new exploit messages. Otherwise, the module generates new input.

The set of filter conditions obtained with each exploit is combined into an efficient final filter by one of Bouncer's modules (see Figure 2).

Bouncer filters can be deployed automatically a few tens of seconds after a new vulnerability is identified and they can be updated as our analysis generalizes the filters. We can also run the filters with vulnerable programs that are instrumented to detect attacks with DFI and to log inputs. This scenario allows Bouncer to refine the filter when an attack that bypasses the filter is detected by DFI. We expect Bouncer to improve availability and reliability significantly until the software vendor issues a patch for the vulnerability, which can take many days.

3. SYMBOLIC EXECUTION

Bouncer computes the initial set of filter conditions by performing forward symbolic execution along the trace obtained by processing a sample exploit. Any input that satisfies these conditions can make the program follow the execution path in the trace until the vulnerability is exploited.

The trace is generated by Nirvana [8] and it contains the sequence of x86 instructions executed by each thread and the concrete values of source and destination operands for each instruction.

The symbolic execution starts by replacing the concrete values of the bytes in the sample exploit by symbolic values: the byte at index i gets symbolic value b_i. Then, it executes the instructions in the trace keeping track of the symbolic value of storage locations that are data dependent on the input. The symbolic values are expressions whose value depends on some of the b_i. They are represented as trees whose interior nodes are x86 instruction opcodes and whose leaves are constants or one of the b_i. We chose this representation because it is trivial to convert into executable code and it avoids the problem of modeling x86 instructions using another language.

The symbolic execution defines a total order on the instructions in the trace that is a legal uniprocessor schedule. The instructions are processed one at a time in this total order. If the next instruction to be processed has at least one source operand that references a storage location with a symbolic value, the instruction is executed symbolically. Otherwise, any storage locations modified by the instruction are marked as concrete, that is, we delete any symbolic value these locations may have had because they are no longer data dependent on the input. For example, consider the trace in Figure 3 that corresponds to the test in the first `if` in Figure 1. Since the source operand of the first instruction is concrete, the value in register `eax` is marked concrete. The source operand of the second instruction references the first byte in the `msg` array that has symbolic value b_0. Therefore, `eax` gets symbolic value (`movsx` b_0) after the instruction is executed. Since the value of register `eax` is now symbolic, the flags register (`eflags`) has symbolic value (`cmp` (`movsx` b_0) 0) after the `cmp` instruction.

```
mov         eax,dword ptr [msg]
movsx       eax,byte ptr [eax]
cmp         eax,0
jg          ProcessMessage+25h (401045h)
```

Figure 3: Assembly code for the first `if` in the example from Figure 1.

Whenever the symbolic execution encounters a branch that depends on the input, it adds a condition to the filter to ensure that inputs that satisfy the filter conditions can follow the execution path in the trace. A branch depends on the input if the value of `eflags` is symbolic. Conditions are represented as a tree of the form: (`Jcc` f), where f is the symbolic value of `eflags`. If the branch is taken in the trace, `Jcc` is the opcode of the branch instruction. Otherwise, `Jcc` is the opcode of the branch instruction that tests the negation of the condition tested in the trace. For example when the last instruction in Figure 3 is executed, symbolic execution generates the condition (`jg` (`cmp` (`movsx` b_0) 0)). If the branch had not been taken in the trace, the condition would be (`jle` (`cmp` (`movsx` b_0) 0)). No conditions are added for branches that do not depend on the input.

Symbolic execution also generates conditions when an indirect call or jump is executed and the value of the target operand is symbolic. The condition in this case asserts that $t_s = t_c$ where t_s is the symbolic value of the target and t_c is the concrete value of the target retrieved from the trace. We represent the condition as (`je` (`cmp` t_s t_c)).

Similar conditions are generated when a load or store is executed and the address operand has a symbolic value. These conditions assert that $a_s = a_c$ where a_s is the symbolic value of the address operand and a_c is its concrete value retrieved from the trace. We represent the condition as (`je` (`cmp` a_s a_c)). EXE [11] describes a technique to generate weaker conditions in this case. We could use this technique to obtain a more general initial filter but our current prototype only applies EXE's technique to common library functions like `strtok` and `sscanf`.

The initial filter is a conjunction of these conditions. Any input that satisfies the filter conditions *can* make the program follow the execution path in the trace until the vulnerability is exploited. We say *can* because the program may only follow the same execution path if the input is processed *in the same setting* as the sample exploit, that is, if the input is received in the same state where the trace started and the runtime environment makes the same non-deterministic choices it made during the trace (for example, the same scheduling decisions). Since this state is reachable and clients do not control the non-deterministic choices, the filter has no false positives.

Additionally, the symbolic or concrete values of instruction operands are *equivalent* across the traces obtained when processing any of the inputs that satisfy the conditions in the initial filter (in the same setting as the sample exploit). Equivalent means identical modulo different locations for the same logical objects, for example, the bases of stacks can differ and locations of objects on the heap can be different but the heaps will be isomorphic.

4. IMPROVING DETECTOR ACCURACY

Detector inaccuracy can lead to filters with unnecessary conditions because it increases the length of the traces used during symbolic execution. Many techniques to detect attacks are inaccurate (e.g., [28, 4, 17, 37, 16, 13, 12]): they detect an attack only when some instruction observes the effect of the exploit rather than identifying the vulnerability. For example, DFI detects a memory safety violation only when it reads data produced by an unsafe write. This write may occur much earlier in the execution.

We analyze the trace to improve DFI's accuracy. When DFI detects a memory safety violation, we traverse the trace backwards to find the unsafe write. We make this write instruction the vulnerability point and any conditions added by instructions that appear later in the trace are removed from the initial filter.

This analysis may be insufficient to identify the vulnerability for attacks that corrupt internal data structures in libraries. For example, a class of attacks corrupts the heap management data structures in the C runtime libraries to write anywhere in memory. Since DFI does not check reads inside libraries, it detects the attack only when an instruction reads data produced by this write. We implemented an analysis to find the instruction that first corrupts the heap management data structures. We first traverse the trace backwards to find the unsafe write (as described above). If this write was executed by one of the heap management functions (e.g., `malloc`), we traverse the trace forward from the beginning to find the first read inside `malloc`, `calloc` or `free` of a value written by an instruction outside these functions. We make the instruction that wrote this value the vulnerability point, and remove any conditions added by later instructions. Our current implementation only deals with heap management data structures but the same idea could be applied to other library functions.

5. PRECONDITION SLICING

The initial filter generated by symbolic execution blocks many exploit variants, but it can be bypassed by attacks that exploit the vulnerability through a different execution path. This section provides an overview of slicing techniques and describes the precondition slicing algorithm that generalizes the initial filter without introducing false positives.

5.1 Static or dynamic?

Program slicing [49] performs static analysis to identify the set of instructions that are relevant to the value of a set of variables when a chosen instruction is reached. This set of instructions is called the slice. We could run an existing program slicing algorithm to remove unnecessary conditions from the initial filter. This algorithm could compute the set of instructions that are relevant to the value of the operands of the instruction at the vulnerability point. Then we could remove from the filter conditions added by branch instructions not in the slice.

The problem with slicing techniques that rely only on static analysis is that they are very imprecise when applied to real C and C++ programs with pointers [29, 24]. They tend to classify most instructions as relevant and, therefore, are not effective at removing conditions from the filter.

Dynamic slicing techniques [29, 51] use dynamic information to improve precision. They take an input, generate an execution trace, and use the dynamic dependencies observed during the trace to classify instructions as relevant. These techniques are not suitable to remove conditions from the filter because they may introduce false positives.

Dynamic slicing can lead to the removal of necessary conditions from the filter because it does not capture dependencies on instructions that were not executed in the trace. Figure 4 shows an example where this can happen. If we apply dynamic slicing to the trace obtained with the sample exploit `msg = "ab"`, the branch corresponding to the second `if` is marked irrelevant. However, removing the condition added by this branch from the initial filter results in a filter that blocks all messages starting with `'a'`. This filter has false positives: it can block messages starting with `"aa"` that can never reach the vulnerability point.

```
int a = 0, b = 0;
int *c = &b;
if (msg[0] == 'a')
  a = 1;
if (msg[1] == 'a')
  c = &a;
*c = 0;
if (a)
  Vulnerability();
```

Figure 4: Example where removing conditions using dynamic slicing can lead to false positives.

We developed a new slicing algorithm to remove unnecessary conditions without adding false positives. It combines ideas from a static slicing algorithm called path slicing [24] with ideas from dynamic slicing. Path slicing is well suited to our problem because it computes the set of statements in a path through a program that are relevant to reach a target location. We improve its accuracy by using not only the path in the execution trace for the sample exploit but also dynamic information from the trace, and we perform slicing of assembly code rather than source code.

5.2 Basic structure

Precondition slicing traverses the execution trace backwards from the vulnerability point to compute a *path slice*, that is, a subsequence of the instructions in the trace whose execution is sufficient to ensure that the vulnerability can be exploited. Intuitively, the path slice contains branches whose outcome matters to exploit the vulnerability and mutations that affect the outcome of those branches [24]. We generalize the initial filter by removing any conditions that were added by instructions that are not in the slice.

The current implementation of precondition slicing is limited to the case where all instructions that are relevant to reach the vulnerability point are executed by the same thread. This does not mean that our algorithm only works with single-threaded programs. In fact, all the programs we used to evaluate Bouncer are multi-threaded. We are working on an extension to handle the case where the interaction between several threads is required to exploit a vulnerability.

We start by describing the algorithm at a high level and explain how we combine static and dynamic analysis to improve precision in the next section.

The algorithm receives as inputs a trace, a representation of the program code, and alias analysis information. The trace has a sequence of entries for each instruction in the execution with the sample exploit. Each entry in the trace has a pointer to the corresponding instruction in the code, the memory addresses or register names read and written by the instruction in the execution trace, and the symbolic or concrete values read and written by the instruction in the symbolic execution. We obtain a representation of the program code by using Phoenix [33] to raise the program binary to an intermediate representation very similar to the x86 instruction set.

We use the alias analysis implemented in DFI [12]. The analysis is performed during the compilation of the program from source code. It generates two relations on operands of instructions in the program code: $\mathit{MayAlias}(o_1, o_2)$ iff the operands o_1 and o_2 may refer to overlapping storage locations in some execution, and $\mathit{MustAlias}(o_1, o_2)$ iff the operands o_1 and o_2 always refer to the same storage location in all executions. These relations are conservative approximations. *MayAlias* may include pairs that never overlap and *MustAlias* may not include pairs that always overlap. The alias relations are written to disk during compilation and later read by our algorithm together with the binary.

The algorithm maintains the following data structures:

- `cur` is the trace entry being processed
- `slice` is a list of trace entries that were added to the path slice. Initially, it contains the entry for the vulnerability point instruction.
- `live` keeps track of dependencies for instructions in `slice`. It contains entries for operands read by these instructions that have not been completely overwritten by instructions that appear earlier in the trace. Entries in `live` contain a pointer to the corresponding operand in the code, the register or memory address from which the instruction read the operand in the execution trace, and the symbolic or concrete value of the operand read by the instruction in the symbolic execution. Entries also keep track of portions of the operand that have been overwritten by instructions that appear earlier in the trace. Initially, `live` contains the operands read by the instruction at the vulnerability point.

We show pseudo code for the algorithm in Figure 5. The algorithm iterates through the trace backwards deciding what instructions to take into the slice. Return, call, and branch

```
ComputeSlice() {
  while (!trace.IsEmpty) {
    cur = trace.RemoveTail();
    if (cur.IsRet) {
      call = trace.FindCall(cur);
      if (MayWriteF(CalledFunc(call), live))
        Take(cur);
      else
        trace.RemoveRange(cur,call);
    } else if (cur.IsCall) {
      Take(cur);
      foreach (e in trace.CallArgSetup(cur)) {
        Take(e);
        trace.Remove(e);
      }
    } else if (cur.IsBranch) {
      if (!Postdominates(slice.head,cur)
          || WrittenBetween(cur, slice.head))
        Take(cur);
    } else {
        if (MayWrite(cur, live))
          Take(cur);
} } }

void Take(cur) {
  slice.AddHead(cur);
  live.UpdateWritten(cur);
  live.AddRead(cur);
}
```

Figure 5: Pseudo-code for the slicing algorithm.

instructions are treated in a special way but other instructions are taken if they may overwrite the operands in `live`.

When `cur` is a return instruction, the algorithm finds the corresponding call in the trace and takes the return if the called function can overwrite operands in `live`; otherwise, none of the instructions in the called function is taken and all the entries between the return and the call are removed from the trace. When the return is taken, the algorithm iterates through the instructions in the called function.

Call instructions are always taken unless they were already removed when processing the corresponding return. We also take the instructions that set up the arguments for the call.

Branches are taken if the direction of the branch is relevant to the value of the operands in `live`, that is, if there is some path originating at the branch instruction that does not lead to the last instruction added to the slice, or if one of the operands in `live` may be overwritten in a path between the branch and the last instruction added to the slice.

The procedure `Take` adds the trace entry of each instruction that is taken to `slice`. In addition, it updates `live` to reflect the writes and reads performed by the instruction in the trace. The method `UpdateWritten` records what locations were written by the instruction in `cur` and `AddRead` adds the operands read by `cur` to `live` recording the location they were read from and their value.

5.3 Combining static and dynamic analysis

Precondition slicing improves the accuracy of path slicing [24] by taking advantage of information from the symbolic execution. It ensures the following invariant. Let F be the intermediate filter that contains all the conditions in the initial filter that were added by instructions up to `cur` and the conditions added by instructions in `slice`. Then all the execution paths obtained by processing inputs that match F (in the same setting as the sample exploit) execute the sequence of instructions in `slice` and the source operands of each of these instructions have equivalent concrete or symbolic values across these paths.

```
ProcessMessage(char* msg, char *p0, char* p1) {
  char buffer[1024];

  if (msg[0] > 0)
    *p0 = msg[0];

  if (msg[1] > 0)
    *p1 = msg[1];

  if (msg[2] == 0x1 && *p0 != 0) {
    sprintf(buffer, "\\servers\\%s\\%c", msg+3, *p0);
    StartServer(buffer, p1);
} }
```

Figure 6: Example to illustrate benefit of using dynamic information to remove operands from `live`.

We use dynamic information to remove entries from `live` sooner than possible using static analysis. The method `UpdateWritten` removes an entry from `live` when the storage location that the operand was read from in the execution trace is completely overwritten by earlier instructions in the trace. Since `live` already captures the dependencies of the instructions that overwrote the removed entry, the entry no longer affects the reachability of the vulnerability at this point in any path obtained with inputs that match F. In contrast, path slicing removes an operand from `live` if *MustAlias* holds for the operand and any of the operands written by the current instruction.

We can illustrate the benefits of our approach using the modified example in Figure 6 and the same sample exploit that we used earlier. Assume that `p0` and `p1` point to the same storage location and that this fact cannot be determined by the static analysis. Path slicing would not be able to remove any condition from the initial filter. Precondition slicing can remove the condition $b_0 > 0$ from the initial filter. When `*p1=msg[1]` is processed, the operand for `*p0` is removed from live because its storage location is overwritten. Therefore, the branch that checks `msg[0]>0` is not added to the slice.

The function `MayWrite` checks if an instruction may overwrite an operand in `live`. We also use a combination of static and dynamic analysis to implement this function. `MayWrite` starts by computing the set L with all operands in the code that may alias at least one operand with an entry in `live`. According to the static analysis, `MayWrite` should return true if any of the operands written by `cur` is in L and false otherwise. We perform an additional check to improve accuracy with dynamic information. We do not take `cur` if its execution did not write over the storage locations of any of the operands in `live` and its target address is determined by concrete values of operands in `live`. This preserves the invariant because the dependencies captured in `live` ensure that `cur` cannot affect the value of the operands in `live` in any path obtained with inputs that match F. So it is not relevant to reach the vulnerability.

To check if the target address of `cur` is determined by concrete values of operands in `live`, we iterate over the instructions in the basic block of `cur`. If all operands read by an instruction must alias an operand with a concrete value in `live` or the result operand of a previous instruction in the basic block, we execute the instruction with the con-

```
ProcessMessage(char* msg, char *p0, char* p1) {
  char buffer[1024];

  if (msg[0] > 0)
    *p0 = msg[0];

  if (msg[1] > 0)
    *p1 = msg[1];

  if (msg[2] == 0x1 && *p0 != 0 && p1 != p0) {
    sprintf(buffer, "\\servers\\%s\\%c", msg+3, *p0);
    StartServer(buffer, p1);
} }
```

Figure 7: Example to illustrate benefit of using dynamic information to compute `MayWrite`.

crete values and record the concrete value of the destination operand. We do not take `cur` if we can compute a concrete value for its target address.

Figure 7 shows a modified version of our example code that illustrates the behavior of `MayWrite`. Assume that `p0` and `p1` point to different locations but static analysis cannot determine this fact. In this case, path slicing cannot remove any conditions from the original filter. Precondition slicing can remove the condition $b_1 > 0$. `*p1=msg[1]` is not taken because it does not overwrite any operand in live and `p1` is in `live`. So the branch that checks `msg[1]>0` is not taken.

`MayWriteF` checks whether a function may write over any operand in `live`. It computes the intersection between the set of all operands the function may modify and L. If the intersection is empty, we do not take the function. Otherwise, we perform an additional check for library functions whose semantics we know. We do not take a library function if the locations it writes are determined by the concrete values of operands in `live` and it did not write over any operand in `live` in the trace. For example, we do not take the call `memcpy(dst, src, n)` if the values of `dst` and `n` are constants or are determined by the concrete values of operands in `live`, and it did not overwrite any operand in `live`.

There are two checks to determine whether to add a branch to the slice. The first one checks if the last instruction added to the slice is a postdominator of the branch [5], i.e., whether all paths from the branch to the function's return instructions pass by `slice.head`. If not, we add the branch to the slice to capture in `live` the dependencies necessary to ensure the branch outcome in the trace. Otherwise, the execution paths might not visit the instructions in `slice`.

We use a standard static analysis to determine postdominance but first we check if the outcome of the branch is already decided given the concrete and symbolic values of operands in *live*. In this case, we do not add the branch to the slice. This is similar to the techniques described to improve the accuracy of `MayWrite` but we make use of symbolic operand values and the conditions added by instructions already in the slice. If the branch flag is symbolic, we check if the conditions already in the slice imply the branch condition or its negation. The current implementation only deals with simple expressions. This preserves the invariant because, when the branch is not added to `slice`, the dependencies captured in `live` already ensure the appropriate branch outcome to reach the vulnerability in any path obtained with an input that matches F.

`WrittenBetween` implements the second check to determine whether or not to take a branch. It returns true if there is some path in the code between the branch and `slice.head` where some operands in `live` may be overwritten. We perform this check by traversing the control flow graph between the branch and `slice.head` in depth-first order. We iterate over the instructions in each basic block visited. We use `MayWrite` (or `MayWriteF` for function calls) to determine if the instructions in the basic block can modify operands in `live`. We also make use of concrete values of operands in `live` to improve the accuracy of the analysis. This is very similar to what was described above.

6. SYMBOLIC SUMMARIES

Precondition slicing is not effective at removing conditions added by instructions inside library functions. Without alias information, it must be conservative and add all the instructions in these functions to the slice. We took a pragmatic approach to address this limitation: we use knowledge about the semantics of common library functions to generate symbolic summaries that characterize the behavior of a function as a set of conditions on its inputs. We use these summaries to replace the conditions extracted from the trace.

We generate symbolic summaries automatically from a template that is written once per library function. There are two cases depending on whether the vulnerability point is inside a library function or the library function is called in the path towards the vulnerability. In the first case, we do not need to characterize the full behavior of the function because what happens after the vulnerability point is not important. Therefore, the symbolic summary is simply a condition on the arguments of the function that is true exactly when the vulnerability can be exploited.

The conditions in a symbolic summary are generated from a template (which depends on the library function) using a combination of static and dynamic analysis. The analysis must determine the symbolic or concrete values of function arguments and potentially the sizes of the objects pointed to by these arguments. For example if the vulnerability is a buffer overflow in the call `memcpy(dst, src, n)`, the summary will state that the size of the object pointed to by `dst` must be greater than or equal to `n`. To generate this condition, the analysis must determine the concrete or symbolic values for `n` and for the size of the object pointed to by `dst`. The value for arguments like `n` is readily available from the trace entry for the corresponding push instruction.

To determine the size of the object pointed to by an argument, the analysis traverses the trace backwards from the function call to the point where the object is allocated. For objects that are allocated dynamically using `calloc`, `malloc`, or `realloc`, the analysis obtains the concrete or symbolic values of the arguments passed to these allocators to compute an expression for the object size. For objects whose size is known statically, the analysis obtains the object size from our representation of the code. During this trace traversal, the analysis builds an expression for the offset between the argument pointer and the start address of the object. The expression for the size used in the condition is equal to the object size minus this offset.

It is harder to compute symbolic summaries for functions in the `printf` family because they have a variable number of arguments with variable types, but it is important because these functions are involved in many vulnerabilities. We distinguish two cases: when the format string depends on the input and when it is known statically.

In the first case, we only deal with calls that receive no arguments beyond the format string, which is the common case with format string vulnerabilities. The analysis generates a summary with a condition on the symbolic values of the bytes in the format string. This condition is true when the format string contains valid format specifiers or when its size (after consuming escape characters) exceeds the size of the destination buffer for functions in the `sprintf` family.

When the format string does not depend on the input, the most common vulnerability is for a function in the `sprintf` family to format an attacker-supplied string into a destination buffer that is too small (as in our example in Figure 1). The summary for this case is a condition on the sizes of the argument strings. The analysis computes the bound on these sizes by parsing the static format string using the same algorithm as `printf`, processing any arguments that do not depend on the input, and determining the size of the destination buffer (as described above).

It is interesting to contrast symbolic summaries with a technique that patches the code by adding a check before the library call. Symbolic summaries allow Bouncer to detect and discard bad input before it is processed. Therefore, services can keep running correctly under attack. Whereas recovering when the check fails is hard. Furthermore, adding the check may require keeping a runtime structure mapping objects to their sizes. This is not needed by symbolic summaries because they are specific to a particular execution path (the one defined by the other conditions in the filter).

We also compute a second type of symbolic summary for library functions that are called in the path towards the vulnerability. We motivate this with the following example:

```
if (stricmp(s,"A string") == 0)
  Vulnerability();
```

the vulnerability in this example is reachable if the attacker supplied string equals “A string” after both are converted to lowercase. The conditions that we extract automatically from a sample execution of `stricmp` will only capture a particular value of s that satisfies the comparison. The techniques described in the next section generate executions with alternative inputs to generalize the filters. But they would require at least 2^8 inputs (where 8 is the size of “A string”) to generate a filter that can block all the attacks that can exploit the vulnerability. If we replace the conditions for the execution of `stricmp` in the sample trace by the summary

$$(s[0] = A \vee s[0] = a) \wedge ... \wedge (s[8] = G \vee s[8] = g) \wedge s[9] = 0$$

we can capture succinctly all values of s that can be used to exploit the vulnerability. Since the vulnerability is not inside these functions, we can alternatively choose to call them directly in the filter. Currently, we only generate summaries for functions in the middle of the path if they have no side effects. There is some recent work that proposes computing similar summaries automatically for arbitrary functions [22].

7. SEARCH FOR OTHER ATTACKS

The techniques described in the previous sections generate filters that block many attacks that exploit the same vulnerability, but they are not sufficient to generate optimal filters that block all exploits.

To generalize our filters further, we search for alternative exploits of the same vulnerability, obtain new execution traces using these exploits, and apply the algorithms described in the previous sections to compute a new filter. The disjunction of the filters obtained from the different execution traces catches more exploits.

The search for all possible exploits may take a significant amount of time. Therefore, the initial filter may be deployed as soon as it is available (within tens of seconds), while the search procedure incrementally improves its accuracy.

We experimented with two techniques to search for alternative exploits. First, we implemented the test generation approach of DART [23] (which was also proposed in [41, 11]). This approach uses conditions obtained from symbolic execution (as in Section 3) to guide new test input generation. It takes a prefix of the conditions negates the last one and feeds the resulting conditions to a constraint solver to obtain new test inputs. This approach has the nice property that given enough time it can find all exploits of the same vulnerability. The problem is that it requires too much time in practice. We are working on search heuristics that might make this approach practical.

We currently use a second search strategy that generates alternative exploits by removing or duplicating bytes in the original exploit messages. This strategy is not guaranteed to find all exploits of the same vulnerability but it is simple, fast and easy to parallelize.

We pick bytes to remove or duplicate using an heuristic based on the filter conditions. We give a score to each condition equal to the total number of bytes in conditions divided by the number of bytes that have an identical condition. Each byte has a score equal to the sum of the scores of the conditions it appears in. We pick the bytes with the lowest scores because they are likely to be filler bytes in buffer overflow exploits.

After generating a potential new exploit, we send it to the version of the vulnerable program that is instrumented to detect attacks (as shown in Figure 2). If the detector signals that the exploit is valid, we repeat the filter generation process for the new exploit. When using symbolic summaries for the library function with the vulnerability, we instrument the vulnerable program to signal success when the call site with the vulnerability is reached. If the exploit is not valid, the detector does not raise an exception. We detect this case using a watchdog that checks if all threads in the vulnerable program are idle. This works well in practice and avoids having to wait for a large timeout.

We use our heuristic to select bytes to remove. If after removing a byte the resulting message is not a valid exploit, we retain that byte and pick another one to remove. We repeat this process until we have tried to remove all bytes or the message size is lower than a bound from a symbolic summary. Then, we start generating new exploits by duplicating bytes in the original exploit message. We pick another byte to duplicate if we did not obtain an exploit or if there are bytes in the resulting exploit message that are not read by the vulnerable program. We stop after we have tried to duplicate all bytes.

We combine the filters generated for each alternative exploit to obtain the final filter. Simply taking the disjunction of all filters can result in a final filter with high overhead. Instead, we compare the conditions applied to each byte index by each filter. A common structure is a set of byte indices in the beginning of a message that have the same condition in all filters. These are typically followed by sequences of byte indices that have different lengths in different filters but have the same conditions applied to each byte in the sequence in each filter. There may be several of these sequences. Typically, they are followed by terminator bytes with the same conditions in each filter.

If we recognize this structure, we take advantage of it to generate an efficient final filter. In this case, the final filter has the conditions for the initial bytes followed by loops that check the conditions on the variable length byte sequences, and conditions that check the terminator bytes.

The final filter is an x86 executable. It is straightforward to convert the conditions generated during symbolic execution into executable code. We use a simple stack-based strategy to evaluate each condition and a short-circuit evaluation of the conjunction of the conditions. The size of the stack is bounded by the depth of the trees in the conditions and filters only access this stack and the input messages. Therefore, filters are guaranteed to run in bounded time and to use a bounded amount of memory.

8. EVALUATION

We implemented a prototype of Bouncer and we evaluated it using real vulnerabilities in real programs: `Microsoft SQL server`, `ghttpd`, `nullhttpd`, and `stunnel`. We started by analyzing each vulnerability carefully to characterize the set of attack messages that can exploit it. Then, we used Bouncer to generate a filter for each vulnerability and evaluated the fraction of attack variants blocked by the filter. The results show that our filters can block all attacks that exploit two of the vulnerabilities.

We also ran experiments to study filter generation. We measured filter generation time, the number of iterations in the search for alternative exploits, and the contribution of each of our techniques to generalize the initial filter.

The final set of experiments measured the overhead introduced by the filters when used to protect running services. We measured both the running time of the filters and the degradation in service throughput with and without attacks. Our results show that the deployed filters have negligible overhead and that attackers must consume a large amount of bandwidth to reduce service throughput significantly.

8.1 Services and vulnerabilities

We start by describing the services and vulnerabilities that we studied, and all the attacks that can exploit each vulnerability. Determining the set of all attacks required detailed analysis of the vulnerable programs aided by our tools. We also describe the sample exploit that we used to bootstrap the filter generation process for each vulnerability.

`SQL server` is a relational database from Microsoft that was infected by the infamous Slammer [34] worm. We studied the buffer overflow vulnerability exploited by Slammer. An attacker can overflow a stack buffer by sending a UDP message with the first byte equal to 0x4 followed by more than 60 bytes different from zero. The stack overflow occurs inside a call to `sprintf`. We use the same exploit as Slammer to start the filter generation process. This exploit has the first byte set to 0x4 followed by a 375-byte string with the worm code, and it overwrites the return address of the function that calls `sprintf`.

`Ghttpd` is an HTTP server with several vulnerabilities [1]. The vulnerability that we chose is a stack buffer overflow when processing the target URL for `GET` requests. The overflow occurs when logging the request inside a call to `vsprintf`. There are many exploits that can overflow the buffer. Successful exploits must have less than 4 Kbytes and have a sequence of non-zero bytes terminated by `"\n\n"` or `"\r\n\r\n"`. They must start with zero or more space characters followed by the string `"GET"` and by one or more space characters. The sequence of remaining characters until the first `'\n'` or `'\r'` cannot contain the string `"\\.."` and must have more than 150 bytes.

The sample exploit to start filter generation for `ghttpd` begins with the string `"GET "` followed by a sequence of 203 non-zero bytes terminated by `"\n\n"`. This exploit overflows the return address of the function that calls `vsprintf`.

`Nullhttpd` is another HTTP server. This server has a heap overflow vulnerability that an attacker can exploit by sending HTTP `POST` requests with a negative value for the content length field [2]. These requests cause the server to allocate a heap buffer that is too small to hold the data in the `POST` request. While calling `recv` to read the `POST` data into the buffer, the server overwrites the heap management data structures maintained by the C library. This vulnerability can be exploited to overwrite arbitrary words in memory.

There is a very large number of messages that can cause the buffer to overflow in `nullhttpd`. Each of these messages is a sequence of lines with up to 2046 non-zero bytes different from `'\n'` terminated by `'\n'`. The first line must start with `"POST"` (case insensitive) followed by two other fields separated by spaces. Then, there can be any number of arbitrary non-empty lines until a line that starts with `"Content-Length: "` (case insensitive) followed by a string that is interpreted as a negative number $-N$ by `atoi`. This line can then be followed by any number of non-empty lines that do not start with `"Content-Length: "`. The message must end with an empty line followed by at least $1024 - N$ bytes of POST data.

We used the exploit described in [14] to start the filter generation process for `nullhttpd`. This is a two message exploit. The first message exploits the vulnerability to modify the CGI-BIN configuration string to allow the attacker to start an arbitrary program. The second message starts a shell. The first message has a line with a cookie that is not necessary for the attack.

`Stunnel` is a generic tunneling service that encrypts TCP connections using SSL. We studied a format string vulnerability in the code that establishes a tunnel for SMTP [3]. An attacker can overflow a stack buffer by sending a message that is passed as a format string to the `vsprintf` function. The buffer overflows if the attacker sends any message with up to 1024 bytes terminated by `'\n'` with a sequence of initial bytes different from `'\n'` that expands to more than 1024 bytes when interpreted as a format string. There are many messages that satisfy these conditions and they can be small, for example, `"%1025.x\n"` overflows the buffer.

The sample exploit that we used to bootstrap the filter generation process for `stunnel` was a message starting with `"%.512x"` followed by 602 `'A'` characters and a `'\n'`. This message overwrites the return address of the function that calls `vsprintf`.

8.2 Filter accuracy

This section evaluates the accuracy of Bouncer filters. Table 1 summarizes our results. A filter has false negatives if it fails to block input that can exploit the vulnerability under study and false positives if it blocks input that cannot exploit the vulnerability.

service	false positives	false negatives
SQL server	no	no
ghttpd	no	yes
nullhttpd	no	yes
stunnel	no	no

Table 1: Accuracy of Bouncer filters.

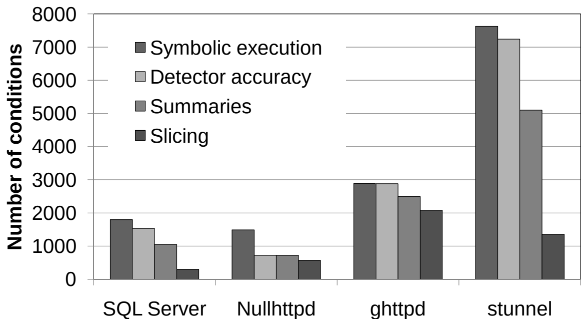


Figure 8: Number of conditions in Bouncer filters after applying each technique. The number of conditions after symbolic execution is the number of conditions in Vigilante filters.

Filters generated by Bouncer have no false positives by design: all the messages they block can exploit the vulnerability. Although we cannot provide strong guarantees on false negatives, we found empirically that Bouncer generates filters with no false negatives for the vulnerabilities in `SQL server` and `stunnel`: our filters block all the attacks that can exploit these vulnerabilities.

The filters for the other vulnerabilities fail to block some exploits but they block many exploits different from the sample exploit. It is harder for Bouncer to generate filters with no false negatives when protocols allow semantically equivalent messages to be encoded in many different ways, or when there are several variable length fields that are processed before reaching the vulnerability. For example, `HTTP GET` messages can have zero or more spaces before `"GET"` and one of more spaces after `"GET"`. Bouncer's filter for `ghttpd` fails to block exploit messages that have spaces before `"GET"` but captures exploits that add spaces after `"GET"`.

Bouncer filters block significantly more attacks than Vigilante filters [16]. Bouncer removes a large fraction of the conditions in Vigilante filters (which are obtained using symbolic execution alone as described in Section 3).

Figure 8 shows the number of conditions in Bouncer filters after symbolic execution (same as Vigilante), after improving detector accuracy, after replacing the conditions in the library function where the vulnerability occurs by a symbolic summary, and after precondition slicing. These numbers were obtained in the first iteration (which processes the sample exploit) and all the conditions depend on the input. Additional iterations would improve the accuracy of Bouncer filters relative to Vigilante filters even further. The results show that all the techniques improve the accuracy of Bouncer filters. Precondition slicing has the largest impact for `SQL server`, `ghttpd`, and `stunnel`. For `nullhttpd`, improving detector accuracy is the largest contributor.

We believe that it would be possible to reduce false negatives by combining Bouncer's techniques with other techniques to compute weakest preconditions (e.g.,[6]). However, there are problems with loops and recursion that have to be solved for these techniques to be useful. Additionally, we could exploit protocol knowledge (when available) to improve the search for alternative exploits as in ShieldGen [19].

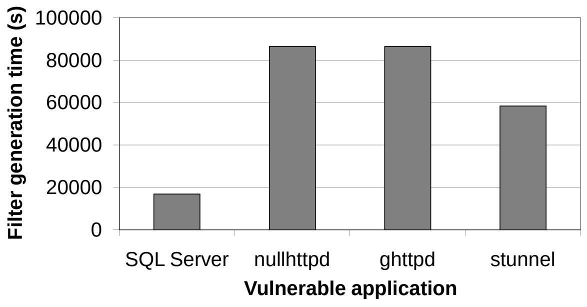


Figure 9: Total time to generate filters.

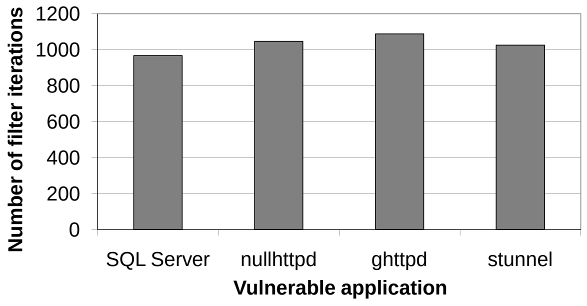


Figure 10: Number of iterations to generate filters.

8.3 Filter generation

We also evaluated the cost of filter generation by measuring the number of iterations and the time to generate the filters. We ran these experiments on a Dell Precision Workstation 350 with a 3GHz Intel Pentium 4 processor and 1GB of memory. The operating system was Windows XP professional with service pack 2.

Figures 9 and 10 show the total time and the number of iterations to generate the filters, respectively. We ran the experiments with a 24-hour time limit. The filter generation process for `nullhttpd` and `ghttpd` did not terminate before this limit. It took Bouncer 4.7 hours to generate the filter for `SQL server` and 16.2 for `stunnel`.

The filter generation process ran for roughly 1000 iterations in all cases. It stopped after 967 iterations for `SQL server` and after 1025 iterations for `stunnel`. The minimum exploit size for `SQL server` is 61 bytes and the maximum is 1024. For `stunnel` the minimum exploit size is 2 bytes and the maximum is also 1024.

There are two reasons for the relatively large filter generation times. First, the time per iteration is large due to inefficiencies in our prototype: 17s for `SQL Server`, 83s for `nullhttpd`, 79s for `ghttpd`, and 57s for `stunnel`. For example, generation and removal of conditions is performed by separate processes that communicate by reading and writing large files. Better integration would significantly reduce the time per iteration. Second, the number of iterations is also large to ensure the final filter has no false positives. We are studying techniques to analyze loops statically that should reduce the number of iterations necessary. Our prototype is useful even with these limitations.

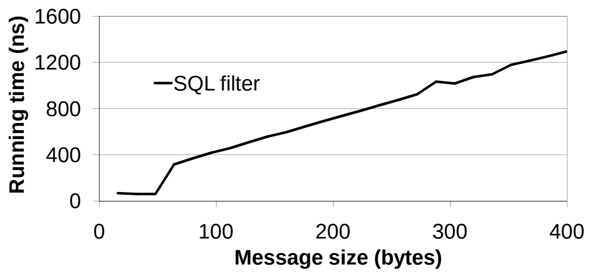


Figure 11: Filter overhead for the Microsoft `SQL server` vulnerability as a function of message size.

In some deployment scenarios, it is easy to reduce filter generation times by exploiting parallelism. Since iterations in our filter generation process are independent, it can be parallelized by assigning each iteration to a different processor. For example, a large software vendor like Microsoft could run the filter generation process in a cluster with 1000 machines and then disseminate the filters to users of vulnerable software. This could speed up filter generation times by up to three orders of magnitude, for example, generating the filters for the `SQL Server` and `stunnel` vulnerabilities would take less than one minute.

In other scenarios, we can deploy a filter after the first iteration, which takes tens of seconds. Then we can deploy an improved filter after each iteration. Additionally, if we run the vulnerable program instrumented to detect attacks with DFI and to log inputs, Bouncer can refine the filter when an attack that bypasses the filter is detected by DFI.

8.4 Filter overhead

We also ran experiments to measure the overhead introduced by deployed filters. The results show that the overhead to process both exploit and non-exploit messages is low for all the filters generated. Therefore, filters allow services to work correctly and efficiently even under attack.

8.4.1 Running time

To measure the filter running time, we varied message sizes from 16 to 400 bytes in increments of 16 bytes. For each message size, we measured the time to process 1000 messages and repeated this experiment 1000 times. We present the average time across the 1000 experiments. We ran these experiments in the machine described in the previous section.

We chose messages to obtain a worst-case overhead for the filters. The messages are picked randomly but with constraints designed to force the filter to check conditions on the maximum number of message bytes possible. For example, the messages used to measure the overhead of the `SQL server` filter have the first byte equal to `0x4` to force the filter to check if the remaining bytes in the message are different from zero. On the other hand, the messages sent to the `stunnel` filter have no `%` characters because the filter stops processing the message bytes when it finds a valid format specifier.

Figure 11 shows the `SQL server` filter overhead. The overhead curve is flat for small message sizes because the first condition in the filter checks if the message has at least 61 bytes. This condition is obtained from the symbolic summary for the `sprintf` function where the vulnerability occurs. The overhead grows linearly with the size for longer

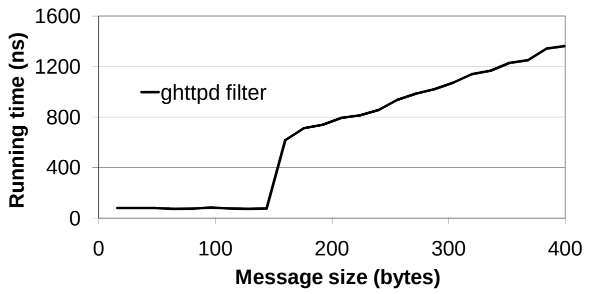


Figure 12: Filter overhead for the `ghttpd` vulnerability as a function of message size.

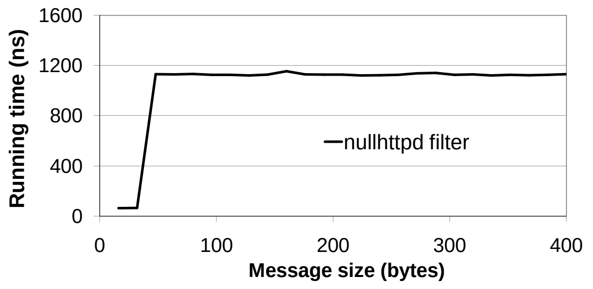


Figure 13: Filter overhead for the `nullhttpd` vulnerability as a function of message size.

messages (which are all exploits) because the number of bytes processed by the filter increases linearly.

Figure 12 shows the filter overhead for the `ghttpd` vulnerability. The curve is similar to the one for `SQL server`: it is flat for messages up to 156 bytes because of the condition obtained from the symbolic summary of the `vsprintf` function where the vulnerability occurs. The overhead grows linearly for exploit messages. As in `SQL server` the overhead for processing non-exploit messages is negligible and the overhead for processing exploit messages is low.

The overhead curve for `nullhttpd` is different from the previous two as shown in Figure 13. This happens because we use `POST` requests to test this filter and increase message size by adding bytes to the `POST` data, which is not processed by the filter. The results show that the overhead to process `POST` requests is low. Since one of the first conditions in the filter checks if the message starts with `POST`, the overhead to process messages that do not start with `POST` is very low.

Figure 14 shows the filter overhead for the `stunnel` vulnerability. This curve is different from all the others because this vulnerability can be exploited with very small messages and the filter must check every byte in the message looking for valid format specifiers. The overhead is higher than that observed for the other filters but this filter is only applied to greeting messages when establishing SSL tunnels for the SMTP protocol. Therefore, its overall impact on performance is negligible.

In all cases shown, the time to run the filter on non-exploit messages is between three and five orders of magnitude smaller than typical wide-area network latencies and between two and three orders of magnitude smaller than typical local-area network latencies. Therefore, our filters have a negligible impact on overall service latency.

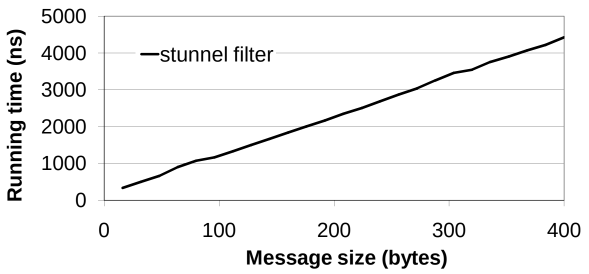


Figure 14: Filter overhead for the `stunnel` vulnerability as a function of message size.

8.4.2 *Effect on throughput*

We also measured throughput reduction due to filters on `SQL server` and `nullhttpd`. The vulnerable services ran on a Dell Precision workstation 350 with a 3GHz Intel Pentium 4 processor and 2GB of RAM, running Windows 2000. We ran clients on a Dell Latitude D600 laptop with a 2GHz Intel Pentium processor and 1GB of RAM, running Windows XP professional with service pack 2, and on a Dell Latitude D620 laptop with a 2.16GHz Core2 Duo processor and 2 GB of RAM, running Windows Vista. Server and clients were connected by a 100Mbps D-Link Ethernet switch.

For `SQL server`, we used the TPC-C benchmark [45] to generate load. To measure the worst case scenario for the filter overhead, clients were configured with zero think time and we used empty implementations for the TPC-C stored procedures. For `nullhttpd`, we generated load using a request from the SPEC Web 1999 benchmark [43]. The request fetches a static file with 102 bytes. We chose this request because the file is cached by the server, which ensures the filter overhead is not masked by I/O.

For both `SQL server` and `nullhttpd`, we measured the maximum throughput in the absence of attacks for a base version without the filter and for a version with the filter deployed. For both versions, we increased the request rate until the server reached 100% CPU usage. We report the average of three runs. The overhead is very low: it was below 1% for both services.

On a second set of experiments, we measured the throughput of the two services under attack. We sent attack probes to servers that were fully loaded and measured the reduction in throughput as we increased the rate of attack probes. The attacks probes carried the same exploits that were used to start the filter generation process. We ran this experiment with services protected by Bouncer filters and with services that restart when they detect an attack. We restart the service immediately after an attack probe is received to make the comparison independent of the performance of any particular detection mechanism. Figures 15 and 16 show the normalized throughput under attack of `SQL server` and `nullhttpd`, respectively.

Detecting the attacks is not enough. If `SQL server` is restarted whenever an attack is detected, the attacker can make the service unavailable with very little effort. The results show that an attacker can reduce the throughput of `SQL server` by more than 90% with an attack rate of only 12 probes per minute. This happens because `SQL server` has a complex start up procedure that takes approximately five seconds. With Bouncer filters, the reduction in throughput with this attack rate is negligible.

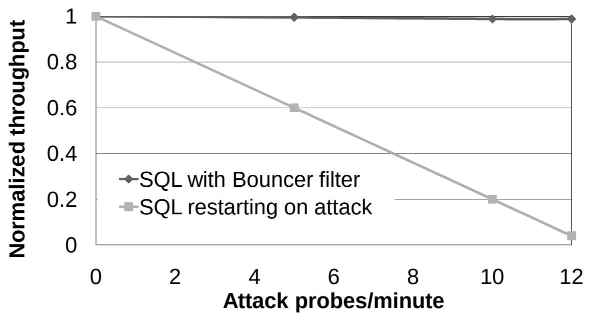


Figure 15: Normalized throughput for Bouncer and Restart on a `SQL server` under attack.

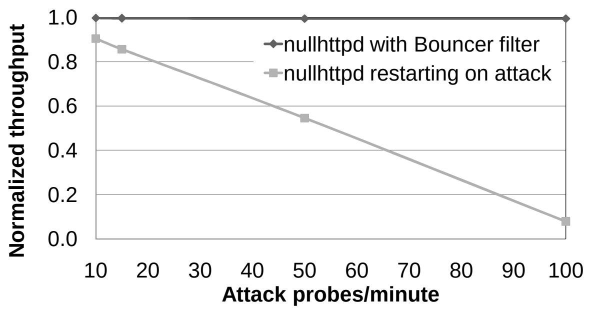


Figure 16: Normalized throughput for Bouncer and Restart on a `nullhttpd` Web server under attack.

The results when `nullhttpd` is restarted on attack are similar: the attacker can reduce the throughput of `nullhttpd` by more than 90% with a rate of only 100 probes per minute. The attack rate required to make the service unavailable is larger for `nullhttpd` than for `SQL server` because the start up time for `nullhttpd` is more than an order of magnitude smaller. The version of `nullhttpd` protected by Bouncer is essentially unaffected by attacks with this rate.

The attacker needs to expend orders of magnitude more bandwidth to affect the throughput of services protected by Bouncer. Figure 17 shows that when the attacker sends almost 18000 probes per second, `SQL server` protected by Bouncer can still deliver 80% of the throughput achievable without attacks. Figure 18 shows that `nullhttpd` protected by Bouncer can deliver 65% of the normal throughput at an attack rate of 1000 probes per second. The throughput degrades faster for `nullhttpd` because it creates a new thread for each request (including attack probes), while `SQL server` uses an efficient thread pooling mechanism.

9. RELATED WORK

There has been previous work on automatic generation of filters to block exploit messages. Most proposals [25, 26, 42, 30, 36, 44, 47, 31, 32, 19] provide no guarantees on the rate of false positives. Therefore, they can make the program stop working even when it is not under attack. From these techniques, ShieldGen [19] is the most closely related to Bouncer. It uses a protocol specification to generate different potential exploits from an initial sample, and it instruments the program to check if potential exploits are valid exploits. We could improve Bouncer's alternative exploit generation by leveraging a protocol specification, but these specifications do not exist for most programs.

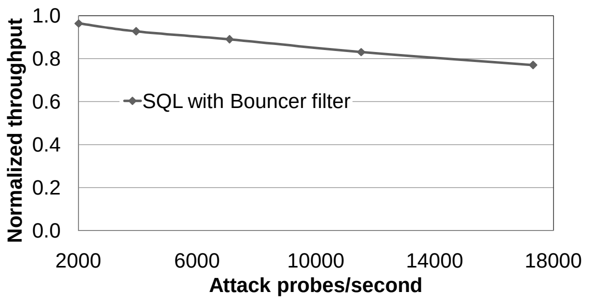


Figure 17: Normalized throughput for `SQL server` under attack.

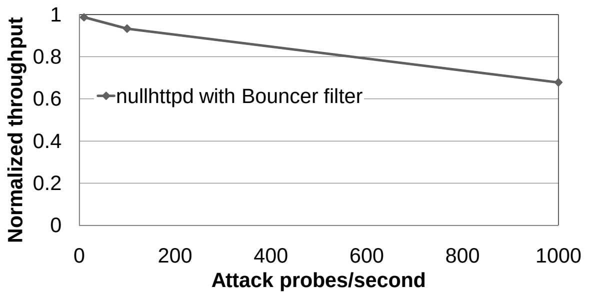


Figure 18: Normalized throughput for `nullhttpd` Web server under attack.

Vigilante [16] computes filters automatically using a form of symbolic execution [27] along the path taken by a sample exploit. Filters are guaranteed to have no false positives and they block all exploits that cause the program to follow the same execution path until the vulnerability point. Crandall et al. [18] have shown that these filters can catch many attack variants. However, attackers can bypass these filters by generating exploits that follow a different execution path.

Recent work has explored techniques to generalize Vigilante filters to block exploits that follow different execution paths. Brumley et al [9] propose three filter representations: Turing machines, symbolic constraints, and regular expressions. Turing machine filters are a chopped version of the vulnerable program that is instrumented to detect the attack. Program chopping removes instructions that cannot be executed from the point where the exploit message is received to the vulnerability point. Turing machine filters can have low false negatives and no false positives, but their overhead is high. The filter can include most of the instructions in the original program (because chopping is imprecise) and it is necessary to initialize the state of the filter before processing each message. The techniques to generate symbolic constraint and regular expression filters do not scale to real programs [10].

Concurrently with our work, Brumley et al. [10] proposed a promising technique to compute symbolic constraint filters, which are similar to Bouncer's filters. They leverage previous work on computing weakest preconditions [6] to create the filter. These filters have no false positives but they may have false negatives because loops are unrolled a constant number of times before computing weakest preconditions. Another concern is that the filters are large (even when loops are unrolled only once) because the addresses in memory accesses are treated symbolically [6]. Bouncer's symbolic execution technique uses concrete addresses to retrieve the symbolic or concrete values of memory cells. This has two advantages: it simplifies the conditions in the filter and it removes unnecessary conditions. Additionally, we use precondition slicing to remove unecessary conditions. It would be interesting to combine Bouncer's techniques with other techniques to compute weakest preconditions [6].

Other techniques prevent attacks by adding checks to programs to detect exploits (e.g., type-safe languages and transparent instrumentation for unsafe programs [4, 12, 13, 16, 17, 28, 40]). These techniques can introduce a significant overhead and they detect attacks too late when the only way to recover may be to restart the program. Vulnerability-specific execution filters [35, 46] can reduce the overhead by instrumenting the program to detect exploits of a single vulnerability, but they cannot solve the second problem.

There are several techniques that allow programs to keep working under attack. Failure-oblivious computing [39] uses CRED [40] to check for out-of-bounds accesses but does not abort the execution when a check fails. Instead, it ignores out-of-bounds writes and it generates values for out-of-bounds reads. This allows programs to keep working but the overhead can be high and programs can behave incorrectly, for example, the authors had to carefully craft values for out-of-bounds reads to prevent infinite loops in their examples. DieHard [7] randomizes the location of objects in a large heap to make it less likely for out-of-bounds writes to overwrite another object. This technique has low overhead but it can be easy for attackers to bypass. Checkpointing and rollback recovery [21] are general techniques to recover from faults. They can be used to recover when an attack is detected [38, 46] but recovery can be relatively expensive and they suffer from the output commit problem [21], that is, they cannot rollback the environment after sending output. Sweeper [46] proposes the use of filters on input messages to reduce the number of times recovery is needed.

10. CONCLUSIONS

This paper described Bouncer, a system that automatically generates filters to block exploit messages before they are processed by a vulnerable program. Bouncer uses DFI to obtain sample exploits for (potentially unknown) vulnerabilities and it generates filters from these samples. Bouncer generates filters using a combination of four techniques: symbolic execution computes an initial set of filter conditions; precondition slicing uses a combination of static and dynamic analysis to remove unnecessary conditions from the filter; symbolic summaries characterize the behavior of common library functions succinctly as a set of conditions on the input; and alternative attack search generates new attack input guided by symbolic execution. Bouncer filters do not have false positives by design and our results show that it can generate filters with no false negatives for real-world vulnerabilities in `SQL server` and `stunnel`. The results also show that these filters introduce low overhead and allow programs to keep running efficiently even when under attack.

Acknowledgments

We thank Angelos Stavrou and Laurent Visconti for help with the code. We thank Byron Cook, Tom Ball and Madan Musuvathi for discussions about this work. We thank our shepherd Yuanyuan Zhou and the anonymous reviewers for comments that helped to improve the paper.

11. REFERENCES

[1] GHttpd Log() Function Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/5960.

[2] Null HTTPd Remote Heap Overflow Vulnerability. http://www.securityfocus.com/bid/5774.

[3] STunnel Client Negotiation Protocol Format String Vulnerability. http://www.securityfocus.com/bid/3748.

[4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow Integrity: Principles, implementations, and applications. In *ACM CCS*, Nov. 2005.

[5] A. Aho, R. Sethi, and J. D. Ullman. Compilers: Principles, techniques, and tools. *Prentice Hall*, 1986.

[6] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *PASTE*, Sept. 2005.

[7] E. D. Berger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *PLDI*, June 2006.

[8] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinic, D. Mihocka, and J. Chau. Framework for instruction-level tracing and analysis of program executuions. In *VEE*, June 2006.

[9] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability signatures. In *IEEE Symposium on Security and Privacy*, May 2006.

[10] D. Brumley, H. Wang, S. Jha, and D. Song. Creating Vulnerability Signatures Using Weakest Pre-conditions. In *Computer Security Foundations Symposium*, July 2007.

[11] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *ACM CCS*, 2006.

[12] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *OSDI*, Nov. 2006.

[13] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *DSN*, July 2005.

[14] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, July 2005.

[15] M. Costa. *End-to-End Containment of Internet Worm Epidemics*. PhD thesis, University of Cambridge, Oct. 2006.

[16] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-End Containment of Internet Worms. In *SOSP*, Oct. 2005.

[17] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Wadpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic detection and prevention of buffer-overrun attacks. In *USENIX Security Symposium*, Jan. 1998.

[18] J. R. Crandall, Z. Su, S. F. Wu, and F. T. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *ACM CCS*, Nov. 2005.

[19] W. Cui, M. Peinado, H. J. Wang, and M. Locasto. ShieldGen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *IEEE Symposium on Security and Privacy*, May 2007.

[20] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, Aug. 1975.

[21] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, Sept. 2002.

[22] P. Godefroid. Compositional Dynamic Test Generation. In *POPL*, Jan. 2007.

[23] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI*, 2005.

[24] R. Jhala and R. Majumdar. Path slicing. In *PLDI*, June 2005.

[25] J. O. Kephart and W. C. Arnold. Automatic extraction of computer virus signatures. In *Virus Bulletin*, Sept. 1994.

[26] H. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *USENIX Security Symposium*, Aug. 2004.

[27] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[28] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *USENIX Security Symposium*, Aug. 2002.

[29] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29, 1988.

[30] C. Kreibich and J. Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *HotNets*, Nov. 2003.

[31] Z. Liang and R. Sekar. Automatic generation of buffer overflow signatures: An approach based on program behavior models. In *ACSAC*, Dec. 2005.

[32] Z. Liang and R. Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *ACM CCS*, Nov. 2005.

[33] Microsoft. Phoenix compiler framework. http://research.microsoft.com/phoenix/phoenixrdk.aspx.

[34] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer worm. *IEEE Security and Privacy*, 1(4), July 2003.

[35] J. Newsome, D. Brumley, and D. Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *NDSS*, Feb. 2006.

[36] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy*, May 2005.

[37] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis and signature generation of exploits on commodity software. In *NDSS*, Feb. 2005.

[38] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies - a safe method to survive software failures. In *SOSP*, Nov. 2005.

[39] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, Dec. 2004.

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

[41] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *ESEC/FSE*, 2005.

[42] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *OSDI*, Dec. 2004.

[43] SPEC. Specweb99 benchmark. http://www.spec.org/osg/web99.

[44] T. Toth and C. Kruegel. Accurate buffer overflow detection via abstract payload execution. In *RAID*, Oct. 2002.

[45] TPC. TPC-C online transaction processing benchmark. 1999. http://www.tpc.org/tpcc.

[46] J. Tucek, J. Newsome, S. Lu, C. Huang, S. Xanthos, D. Brumley, Y. Zhou, and D. Song. Sweeper: A lightweight end-to-end system for defending against fast worms. In *EuroSys*, Mar. 2007.

[47] X. Wang, C.-C. Pan, P. Liu, and S. Zhu. Sigfree: A signature-free buffer overflow attack blocker. In *Usenix Security Symposium*, Aug. 2006.

[48] W. Weimer and G. C. Necula. Finding and preventing runtime error handling mistakes. In *OOPSLA*, Oct. 2004.

[49] M. Weiser. Program slicing. In *Conference on Software Engineering*. IEEE Computer Society Press, 1981.

[50] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

[51] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *PLDI*, June 2004.