Security Benchmarking using Partial Verification

Thomas E. Hart, Marsha Chechik
Department of Computer Science
University of Toronto

David Lie
Department of Electrical and Computer Engineering
University of Toronto

Abstract

Implementation-level vulnerabilities are a persistent threat to the security of computing systems. We propose using the results of partially-successful verification attempts to place a numerical upper bound on the insecurity of systems, in order to motivate improvement.

1 Introduction

Despite recent attention to secure computing, programs continue to be plagued with vulnerabilities. While some of these vulnerabilities are due to deep logical errors, most are due to implementation bugs such as buffer overflows and other input validation errors. Improving software security requires eliminating these bugs, but doing so requires investing in programmer time to audit and fix a program, or sacrificing performance by automatically adding runtime safety checks; hence, severe vulnerabilities persist [23]. We find this unacceptable, and agree with Bill Joy's view of software vulnerabilities:

> *Speaking with one voice, we should insist that software is not complete unless it is secure. The alternative is unacceptable — we can't tolerate identity theft, financial loss, organizational downtime, and national security threats from untested and therefore inadequately secure software. We deserve, and can demand, better.* [16]

To demand secure software, we must be able to quantify software (in)security. While vendors have the means and motivation to measure programmer time and application performance, we lack good methods of measuring insecurity. The state of the art seems to be the "construction site" method of measuring the time since the last reported vulnerability in a program, or the number of reported vulnerabilities in a given time period. This method assumes that software is safe until proven otherwise, despite repeated experience showing that this assumption is unreasonable. Vendors thus have little incentive to actively improve their software until a third party demonstrates the presence of a vulnerability, through either responsible disclosure or malicious attacks.

We propose a "guilty until proven innocent" approach to quantifying insecurity using the partial results of verification attempts. We limit our scope to *property checking*, which has experienced much industrial success [1, 10, 27]. Automated tools instrument the checked program with assertions which halt execution when a property, such as memory safety [21] or correct use of security APIs [4], is about to be violated. Analysis tools then check the safety of these assertions individually (ie. whether they can fail). Since software verification is undecidable in general, a tool will typically be able to prove some subset of the assertions safe. We refer to this partial success as *partial verification*. We claim that it is feasible to gradually rewrite programs so that more assertions can be proven safe, meaning that there are fewer places in which the property may be violated, and thus fewer potential vulnerabilities. We give metrics to measure this progress.

Using verification to measure potential insecurity has several advantages. The metrics' conservative nature gives vendors an incentive to pro-actively fix parts of a program which *may* be vulnerable, in order to increase their score. Furthermore, it is possible for a vendor to get a perfect score by proving the program obeys the checked properties. A perfect score is an attractive goal, as it means that a program is free of an entire class of vulnerabilities, thus provably cutting off an avenue for attack.

Unfortunately, measuring properties of code introduces the possibility of software vendors gaming the metric to make their code appear more secure. We discuss how a naïve metric could be gamed by a *code path redirection* attack, and present candidate metrics which are resistant to gaming.

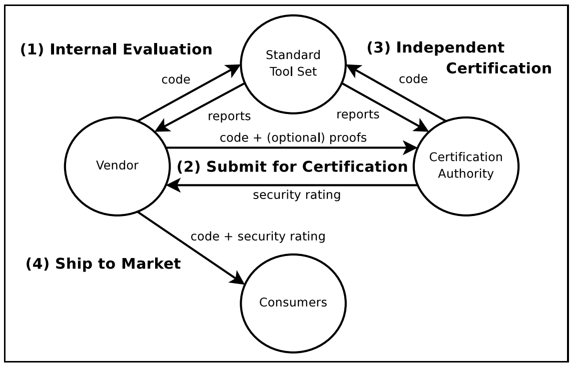

Figure 1: Security benchmarking framework.

Our benchmarking process measures provable assurance rather than estimating risk, and has the limitations associated with any such approach. If a program is written in such a way that proving safety is difficult, it could get a lower score than a competitor's product, even if the competitor's product contains more vulnerabilities. Furthermore, we do not address ease of exploitation, nor do we address the potential consequences of an exploit. Despite these limits, our scheme allows a customer to see whether successive revisions of a program are being hardened, thus increasing security assurance over time.

The next section of this paper discusses our benchmarking framework. In Section 3, we argue that property checking should be considered feasible, given modest coding guidelines. Section 4 proposes metrics which are resistant to gaming by malicious vendors. We conclude with a discussion in Section 5.

2 Benchmarking Framework

In this section, we discuss (1) the principals in our scheme, (2) the scope of the properties being checked, and (3) the techniques for checking these properties.

Principals. Figure 1 shows the principals in our benchmarking scheme — vendors, consumers, and the certification authority — and their interactions. The vendors' goal is to convince their customers that their code is of high quality. Since the customers do not trust the vendors, an independent certification authority attempts to verify the software, and produces a rating which vendors can use when marketing their software. Both the vendors and the certification authority have access to a common set of verification tools, so that the vendors are able to perform internal audits, and harden their code before submitting it to the certification authority.

Note that the certification authority must be able to compile the vendors' source code, in order to ensure that the binary shipped to the customers corresponds to the version of the software which was verified.

Properties. The properties which can be verified are safety properties which can be enforced by *security automata*, a class of Büchi automata which enforce safety properties by terminating a program's execution when a property is violated [24]. Example properties in this class include memory safety and proper API usage, such as correct use of `setuid` [4] and input sanitization functions. For the purposes of static verification, the automata are typically implemented using program instrumentation, in which assertions are automatically added to the program to terminate execution when a property is violated. Several systems, such as CCured [21], automatically instrument for memory safety using "fat" pointers and bounds checking assertions; similarly, numerous automata-based systems provide instrumentation for checking API usage properties [2, 3, 13]. Figure 2 shows an example program with instrumentation to check for double-free errors, using notation similar to [3].

The assertions which encode a property can be checked individually; hence, verification can be partially successful if some assertions are proven safe and others are not. We refer to an assertion which has not been proven safe by any tool as an *unverified assertion* (UVA). Progress towards provably safe code comes from removing UVAs from a program. As the number of UVAs decreases, we can increase our confidence that the software obeys the checked property.

Several security properties are well-studied in the formal verification community, and should be included in the benchmarking process. The high frequency and severity of buffer overflows make array bounds checking important, and many tools, notably those based on abstract interpretation, target this property [10, 26, 27]. API usage properties, such as use-after-free and double-free errors, and correct use of setuid [4], are typically amenable to verification by software model checking [1]. The absence of SQL injection vulnerabilities can be verified using string analysis [28], or by ensuring that SQL statements are only made using a standardized API for parameterized queries, and that tainted strings only enter these queries as parameters [5, pp. 160–166]. Chess and West discuss several security properties in detail [5].

We expect the set of properties for which automated verification is feasible to grow. Currently, the standard solution for checking for cross-site scripting and command injection vulnerabilities is taint checking [5]. This approach checks that tainted data undergoes sanitization checks, but not that these checks are correct. A vendor could thus add useless sanitization checks to their code to make it do well in the benchmarking process, without actually increasing security. Recent work [29] applying string analysis [6] to these problems gives us hope that the situation will improve.

Tools. The instrumentation and verification tools are trusted by all principals. The set of verification tools can include tools using different paradigms, such as abstract interpretation [8], model checking [7], type sys-

```


char *p, *q;
p = malloc (10);

if (p == NULL) exit (0);
q = p;

free (p);

free (q);
```

(a) Original Program

```
struct shadow_char_ptr { char *orig; int state; };
/* ... */
struct shadow_char_ptr *p, *q;
p = malloc (10);
if (p != NULL) p->state = ALLOCATED;
if (p == NULL) exit (0);
q = p;
assert (p->state == ALLOCATED);
free (p);   p->state = UNALLOCATED;
assert (q->state == ALLOCATED);
free (q);   q->state = UNALLOCATED;
```

(b) Instrumented Version

Figure 2: Example instrumentation to check character pointers for double-free errors.

tems, string analysis [6], and deductive verification using Hoare logic [14]. The only requirement on the tools is that they are *sound* — ie. that they never proclaim unsafe assertions safe.

We assume that the tool set includes an automated *proof checker*, so that vendors can supply manually-generated proofs if no automated tool is able to verify an assertion.

3 Feasibility of Property Verification

The feasibility of our benchmarking scheme depends on a vendor's ability to eliminate UVAs. Since Rice's Theorem [22] reduces checking of non-trivial properties to the halting problem, one might think that the situation is hopeless. This level of cynicism is not justified, however, since the theorem only guarantees that given a verification algorithm, there will be programs which it cannot verify. It does not guarantee that there will be a need to write these difficult-to-verify programs. Programmers are already encouraged to write highly structured and understandable programs [9], and when both the programs and the verification tools are of high quality, verification can be very successful. For example, Ball et al. [1] report checking 26 device drivers for conformance with 64 API usage rules, and were able to verify safety for 93% of $\langle$*program*, *property*$\rangle$ pairs, and Venet et al. [27] report verifying the safety of 80% of array bounds checks in a piece of NASA flight control software.

We claim that vendors have significant control over the ease of verifying their code. The key is to make correctness proofs *obvious* to automatic tools, and there are many ways to do this.

Redundant checks. For the properties within the scope of our benchmarking scheme, there is a trivial way to get rid of UVAs — one can use the instrumenter to see where to add runtime checks to the program, and leave these checks in the shipped product. This guarantees security with a minimal investment in programmer time, but sacrifices application performance. For some properties and programs, this solution may be practical — for example, CCured reports low overhead for I/O-bound applications [21] — but in other cases this overhead may be unacceptable. In addition, runtime checks which halt a program leave systems vulnerable to denial-of-service attacks. In general, humans are able to add needed runtime checks with less performance overhead and more graceful recovery than automated tools.

Coding practices. Verification is easiest when a proof of correctness depends only on simple computations which are predictable by a tool. For example, many tools which check for memory safety contain optimizations for analyzing C strings [10, 26], including optimized stubs for standard library functions. Tools can then track string length and allocated array sizes, and verify the absence of buffer overflows using systems of linear inequalities. Similarly, verifying that API usage rules are obeyed often involves only reasoning about flags representing typestate [25] information, which are easily modeled using boolean programs [1].

Verification is harder when proving safety requires proving the correctness of more complex or unusual computations. Dor et al. [10] give a good example of a function in which the absence of buffer overflows is difficult to verify. The function processes an input string, and may exceed the string's bounds if it does not contain an equal number of '(' and ')' characters. Even though this requirement is satisfied by the callers, verifying the program is difficult, because a tool must (1) represent the property that an array contains an equal number of '(' and ')' characters, (2) verify that a function computes this fact correctly, and (3) use this fact to prove safety. This proof is much more difficult than solving a system of linear inequalities. An alternative implementation which does *not* depend on a complex property of array contents would be much easier to verify.

Annotations. Adding annotations to the source code can significantly ease the job of a verification tool, as they can guide a tool towards the correct proof. Annotations which tell a tool *how* to prove a property holds, rather than to *assume* that some property holds, do not compromise soundness. These annotations reduce the extent to which tools must *conjecture* proofs, bringing their job closer to the easier task of proof-checking. A typical use of annotations is to annotate functions with conditions under which they are safe, prove that these conditions are satisfied by all callers, and analyze the function under these assumptions. The annotation approach has worked well for Microsoft, which has gotten rid of many buffer overflows using SAL annotations [12].

Manual proofs. In some cases, one may need to write part of a program in a way that is not amenable automated verification. A typical example would be writing a computationally-intensive task using hand-optimized inline assembly. In such cases, it may make sense for the programmer to supply a proof manually; for example, using techniques from proof-carrying code [20].

4 Metrics

For any benchmarking scheme to be useful, the results must be expressed as a meaningful number which is difficult to game. The standard measure of quality in software engineering is *defect density* — the number of defects divided by the size of a program [11]. A straightforward adaptation of this metric for our scheme would be the number of UVAs divided by the lines of code in a project. However, this metric is unsuitable when vendors are untrusted, as a malicious vendor could improve their score simply by adding more lines to a program.

Another straightforward adaptation would be to count the absolute number of UVAs in a program, but even this is easy to game, by performing a *code path redirection attack* which changes the control flow of a program so that multiple assertions are collapsed into one. Figure 3 shows an example. Instrumenting array writes with bounds checks would result in two assertions in the program in Figure 3(a) (lines 8 and 9); however, a malicious vendor could transform the program into the one in Figure 3(b), which adds a layer of indirection and would contain only a single assertion (line 2). Chess and West [5, pp. 63–64] discuss how levels of indirection can stymie attempts to count possible vulnerabilities in a program, even if the vendor is not acting maliciously.

An ideal metric would (1) be resistant to gaming, (2) consistently reward vendors for fixing UVAs, and (3) allow comparability between products. While code obfuscation makes it difficult to devise a metric which is perfect in all three respects, we believe a practical metric that does well in all three is possible. We give four proposals, leaving refinements as future work.

The legal option. One simple option is to ban gaming, charge the certification authority with inspecting programs for attempts to game the system, and simply report the number of UVAs in a program. This solution would be problematic, as it would require significant human judgment. Furthermore, it would not address the effects of benign indirection.

Scaling by paths. To perform a code path redirection attack, a malicious vendor must increase the number of paths through the program's control flow graph (CFG) which end at a given UVA. We can thus defend against redirection attacks by weighting each UVA based on the number of paths leading to it.

We can formalize this using the program's interprocedural CFG, which includes edges from each function call to the head of the called function, and from each **return** statement to all corresponding return points, treating function pointers conservatively. Let the UVAs in a program P be $U = \{u_1, u_2, \ldots, u_N\}$, and let l_u be the line of a UVA u. For any line l, let $\mathcal{P}(l)$ be the number of paths to l in P's CFG, conflating paths that differ only in the number of loop iterations. We can then weight each UVA u as:

$$\sum_{u \in U} \mathcal{P}(l_u) \tag{1}$$

For a large program, calculating $\mathcal{P}(l_u)$ is likely to be infeasible; however, an upper bound on $\mathcal{P}(l_u)$ is tractably computable. Let $R(l)$ be the CFG nodes from which line l can be reached. We can then approximate $\mathcal{P}(l_u)$ as:

$$\prod_{j=0}^{\text{max in-degree}} j^{|\{x \in R(l_u) \text{ such that } \textit{in-degree}(x)=j\}|} \tag{2}$$

This metric limits a malicious vendor's options for code path redirection attacks to those which can be hidden in loops — for example, by putting the arrays and indices in Figure 3(a) in a pair of arrays and looping over them. Although not perfect, this limits the redirection attacks a malicious vendor may easily perform.

The advantages of this metric are that it compensates for benign indirection, and that removing a UVA will always improve a program's score. A disadvantage is that the number of paths to a UVA will increase exponentially with the size of a program, making the scores of different programs vastly incomparable. The primary use of this metric would therefore be to ensure that successive revisions of a program are being hardened over time.

Blocks reaching UVAs. A somewhat simpler scheme is to count the number of basic blocks in the CFG from which *any* UVA can be reached. This metric grows linearly in the size of the program, rather than exponentially, so the values are more comparable across different programs. Since a redirection attack will only change *which* UVAs a basic block reaches, this metric is also resistant to redirection attacks. Vendors can only game the metric to the extent that they can remove or merge basic blocks, which is limited by the need to have decision points in programs.

The downside of this metric is that there is no guarantee that removing a UVA will improve a program's score, since many UVAs may be reachable from a given block.

```



void foo (int i1, int i2) {
  char A[100], B[50];
  A[i1] = 'H'; /* assert (i1 < 100); */
  B[i2] = 'i'; /* assert (i2 < 50); */  }
```

(a)

```
void write_char (char *p, char c) {
  *p = c; /* assert (offset(p) < end(p)) */  }

void foo (int i1, int i2) {
  char A[100], B[50];
  write_char (A + i1, 'H');
  write_char (B + i2, 'i');  }
```

(b)

Figure 3: A code path redirection attack in a contrived program. (a) Original program. (b) Malicious vendor has made all writes to character arrays happen on the same line.

This metric thus may make it too difficult for a vendor to improve a program's score, making them less likely to try to improve a program.

Dynamic analysis. A fundamentally different approach is to rely on the facts that (1) the benchmarking framework uses runtime instrumentation to specify properties, and (2) vendors want their programs to perform well. Let P be the original program, and let P' be the instrumented program, with all safe assertions removed. The metric is:

$$\frac{\text{performance of } P' \text{ (in seconds)}}{\text{performance of } P \text{ (in seconds)}} \qquad (3)$$

where the performance is measured on some industry-standard benchmark, such as WebStone. Removing assertions will bring this ratio closer to 1, and the goal is to get this ratio as close to 1 as possible. Code path redirection attacks will not, in general, improve this ratio, and vendors have a disincentive to improve their score by sabotaging the performance of P. Furthermore, as the ratio approaches 1, the customers have an incentive to ask the vendor to ship P', rather than P.

A disadvantage of this metric is that it requires that standard benchmarks exist for the application class. The other disadvantage is that P' will, in general, contain updates to metadata needed for the assertions (see lines 5, 9, and 11 of Figure 2(b)), and removing UVAs will not remove these metadata updates, unless all UVAs are removed, making the metadata unnecessary. There is thus a limiting factor making it hard for the ratio in Eq. 3 to approach 1 until all UVAs are removed.

5 Discussion

We discuss potential objections to our scheme, and policy issues associated with it.

5.1 Objections

Vendors will not want to have their code benchmarked. Vendors often consider defect data proprietary [17], and thus may find a benchmark based on potential defects unattractive. Uncooperative vendors can be pressured by consumers to submit their products for benchmarking. The academic and open source communities can also exert pressure on vendors by attempting to harden and benchmark open source equivalents.

Vendors will not want to share their code with the certification authority. We view disclosure of code to the certification authority as being similar to current practices of disclosing code to business partners, governments, and academics; for example, Microsoft's Shared Source initiative [19]. Such licensing and non-disclosure agreements can prevent employees of the certification authority from disclosing proprietary information.

UVAs are not necessarily vulnerable, so the benchmark is unfair. While it is true that the presence of UVAs does not imply the presence of vulnerabilities, we do not believe it is reasonable to give vendors the benefit of the doubt. We furthermore argue that if correctness arguments are made obvious to verification tools, they will also be obvious to programmers, making new programmers less likely to introduce bugs into existing code.

Verification will never be practical. Verification is already deployed in commercial tools, such as Microsoft's SDV [18], the PolySpace C Verifier [26], and NEC's F-Soft [15]. Given the advances in program verification in recent years, it is also reasonable to expect continuous improvement. However, a two-way effort is needed: programmers must learn how to write code which is easily verifiable.

The scheme encourages programmers to fix non-existent bugs rather than doing useful work. When any benchmarking process is put in place, there is a risk that people will optimize for the benchmark. As with performance benchmarking, we believe that security benchmarking is worthwhile despite this risk. The alternative of simply trusting vendors, and thus counting on having insecure software, is much less attractive.

5.2 Policy Issues

There are policy issues about what properties should be checked, what tools to include in the standard tool set, and what assumptions, if any, the tools may make. For example, many tools assume that integers behave as theoretical integers rather than 32-bit integers, and such unsound assumptions can make the verification task much easier. The principals must agree on whether such assumption should be allowed.

Unsound assumptions may introduce opportunities for vendors to game the system by making assertions appear to lie within dead code; for example, if a tool assumes that integers are unbounded, a malicious vendor could enclose an assertion in an `if (MAX_INT + 1 <= 0) {...}` statement. Tools will typically say that assertions in dead code are safe, since they cannot fail (or execute). To guard against this form of cheating, the benchmarking scheme could mandate that no assertion may have a dependence on an infeasible branch. A related problem is that since most analysis tools assume memory safety, deliberate memory safety violations could be used to hide UVAs; legal disincentives, with enforcement aided by testing for deliberate violations, are a possible defence.

Agreeing on what properties to verify may be difficult when different vendors' products use different programming languages and libraries, as properties are closely tied to them. In this case, the self-interests of vendors may be at odds. As with reaching agreement on performance benchmarks, we expect bloodshed in these cases, leading to eventual compromise.

6 Conclusions and Future Work

We have laid out a benchmarking scheme for pressuring software vendors to use verification technology to provably secure their code over time. We believe that this scheme, if adopted, would lead to substantially more secure software. Problems for future work are providing metrics which achieve the best balance of resistance to gaming, consistency in rewarding vendors for fixing UVAs, and comparability between products, and in determining the best set of properties to verify.

Acknowledgments

We thank our colleagues at Carleton University, RMC, the University of Toronto, and CMU; in particular Kelvin Ku, Lionel Litty, Greg Wilson, Arie Gurfinkel, Paul Van Oorschot, Anil Somayaji and Scott Knight. Thomas E. Hart is supported by a MITACS project grant.

References

[1] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. "Thorough Static Analysis of Device Drivers". In *Proc. EuroSys'06*, pages 73–85, 2006.

[2] T. Ball and S. Rajamani. SLIC: A specification language for interface checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research, 2002.

[3] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The BLAST query language for software verification. In *Proc. SAS'04*, LNCS 3148, pages 2–18, 2004.

[4] H. Chen, D. Wagner, and D. Dean. Setuid demystified. In *Proc. USENIX Security'02*, pages 171–190, 2002.

[5] B. Chess and J. West. *Secure Programming with Static Analysis*. Addison-Wesley Software Security Series. Addison-Wesley, 2007.

[6] A. S. Christensen and A. Møller and M. I. Schwartzbach. Precise analysis of string expressions. In *Proc. SAS'03*, pages 1–18, 2003.

[7] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[8] P. Cousot and R. Cousot. "Abstract Interpretation: A Unified Lattice Model For Static Analysis of Programs by Construction or Approximation of Fixpoints". In *Proc. POPL'77*, pages 238–252, 1977.

[9] E. W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, 1968.

[10] N. Dor, M. Rodeh, and S. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In *Proc. PLDI'03*, pages 155–167, 2003.

[11] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. International Thompson Computer Press, 1997.

[12] B. Hackett, M. Das, D. Wang, and Z. Yang. "Modular Checking for Buffer Overflows in the Large". In *Proc. ICSE'06*, pages 232–241, 2006.

[13] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proc. PLDI'02*, pages 69–82, 2002.

[14] C. Hoare. "An Axiomatic Basis for Computer Programming". *Communications of the ACM*, 12(10):576–580, October 1969.

[15] F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-Soft: Software verification platform. In *Proc. CAV'05*, pages 301–306, 2005.

[16] B. Joy. Software isn't complete unless it's secure. *BusinessWeek*, September 2006. `http://www.businessweek.com/technology/content/sep2006/tc20060926%5f175%459.htm?chan=top+news%5ftop+news+index`.

[17] L. M. Laird and M. C. Brennan. *Software Measurement and Estimation: A Practical Approach*. Quantitative Software Engineering Series. John Wiley & Sons, Inc., 2006.

[18] Microsoft Research. Static driver verifier, 2004. `http://www.microsoft.com/whdc/devtools/tools/SDV.mspx`.

[19] Microsoft Shared Source Initiative Home Page. `http://www.microsoft.com/resources/sharedsource/default.mspx`.

[20] G. C. Necula. Proof-carrying code. In *Proc. POPL'97*, pages 106–119, 1997.

[21] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. "CCured: Type-Safe Retrofitting of Legacy Software". *ACM TOPLAS*, 27(3):477–526, 2005.

[22] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74(2):358–366, 1953.

[23] SANS Institute. SANS Top-20 2007 Security Risks (2007 Annual Update), 2007. `http://www.sans.org/top20/2007/`.

[24] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[25] R. E. Strom and S. Yemini. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.

[26] The MathWorks. Polyspace products for embedded software verification. `http://www.mathworks.com/products/polyspace/`.

[27] A. Venet and G. Brat. Precise and efficient static array bound checking for large embedded C programs. In *Proc. PLDI'04*, pages 231–242, 2004.

[28] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proc. PLDI'07*, pages 32–41, 2007.

[29] G. Wassermann and Z. Su. Static Detection of Cross-Site Scripting Vulnerabilities. In *Proc. ICSE'08*, pages 171-180, 2008.