Repeating the Past
Experimental and Empirical Methods in System and Software Security

Stephan Neuhaus

Dissertation zur Erlangung des Grades des
Doktors der Ingenieurswissenschaften
der Naturwissenschaftlich-Technischen Fakultäten der Universität des Saarlandes

Saarbrücken, 2007

Main Typeface URW Garamond No. 8 10/12
Additional Typefaces Adobe Times, Adobe Courier,
Computer Modern Math Symbols
Typesetting System PDF-LaTeX

Printed in Germany
4 3 2 2 3 4 5

To Stefanie, with love.

It's like déjà vu all over again.
— Yogi Berra

Contents

Abstract

I propose a new method of analyzing intrusions: instead of analyzing evidence and deducing what must have happened, I find the intrusion-causing circumstances by a series of automatic experiments. I first capture process's system calls, and when an intrusion has been detected, I use these system calls to replay some of the captured processes in order to find the intrusion-causing processes—the cause-effect chain that led to the intrusion. I extend this approach to find also the inputs to those processes that cause the intrusion—the attack signature.

Intrusion analysis is a minimization problem—how to find a minimal set of circumstances that makes the intrusion happen. I develop several efficient minimization algorithms and show their theoretical properties, such as worst-case running times, as well as empirical evidence for a comparison of average running times.

Our evaluations show that the approach is correct and practical; it finds the 3 processes out of 32 that are responsible for a proof-of-concept attack in about 5 minutes, and it finds the 72 out of 168 processes in a large, complicated, and difficult to detect multi-stage attack involving *Apache* and *suidperl* in about 2.5 hours. I also extract attack signatures in proof-of-concept attacks in reasonable time.

I have also considered the problem of predicting before deployment which components in a software system are most likely to contain vulnerabilities. I present empirical evidence that vulnerabilities are connected to a component's imports. In a case study on *Mozilla*, I correctly predicted one half of all vulnerable components, while more than two thirds of our predictions were correct.

Zusammenfassung

Ich stelle eine neue Methode der Einbruchsanalyse vor: Anstatt Spuren zu analysieren und daraus den Ereignisverlauf zu erschließen, finde ich die einbruchsverursachenden Umstände durch automatische Experimente. Zunächst zeichne ich die Systemaufrufe von Prozessen auf. Nachdem ein Einbruch entdeckt wird, benutze ich diese Systemaufrufe, um Prozesse teilweise wieder einzuspielen, so dass ich herausfinden kann, welche Prozesse den Einbruch verursacht haben—die Ursache-Wirkungs-Kette. Ich erweitere diesen Ansatz, um auch die einbruchsverursachenden Eingaben dieser Prozesse zu finden—die Angriffs-Signatur.

Einbruchsanalyse ist ein Minimierungsproblem—wie findet man eine minimale Menge von Umständen, die den Einbruch passieren lassen? Ich entwickle einige effiziente Algorithmen und gebe sowohl theroretische Eigenschaften an, wie z.B. die Laufzeit im ungünstigsten Fall, als auch empirische Ergebnisse, die das mittlere Laufzeitverhalen beleuchten.

Meine Evaluierung zeigt, dass unser Ansatz korrekt und praktikabel ist; er findet die 3 aus 32 Prozessen, die für einen konstruierten Angriff verantwortlich sind, in etwa 5 Minuten, und er findet die 72 von 168 Prozessen, die für einen echten, komplizierten, mehrstufigen und schwer zu analysierenden Angriff auf *Apache* und *suidperl* verantwortlich sind, in 2,5 Stunden. Ich kann ebenfalls Angriffs-Signaturen eines konstruierten Angriffs in vernünftiger Zeit erstellen.

Ich habe mich auch mit dem Problem beschäftigt, vor der Auslieferung von Software diejenigen Komponenten vorherzusagen, die besonders anfällig für Schwachstellen sind. Ich bringe empirische Anhaltspunkte, dass Schwachstellen mit Importen korrelieren. In einer Fallstudie über *Mozilla* konnte ich die Hälfte aller fehlerhaften Komponenten korrekt vorhersagen, wobei etwa zwei Drittel aller Vorhersagen richtig war.

Ausführliche Zusammenfassung

Üblicherweise werden Einbrüche analysiert, indem Spuren gesucht und ausgewertet werden. Daraus wird dann zeitlich rückwirkend auf die Ursache-Wirkungs-Kette geschlossen, die den Einbruch verursacht haben muss. Eine solche deduktive Vorgehensweise hat jedoch einige Nachteile:

Vollständigkeit. Spuren sind oft nicht in ausreichender Menge vorhanden, um die Ursache oder die Ursachen verläßlich bestimmen zu können. Spuren werden von Angreifern gelöscht oder manipuliert, fallen aber auch manchmal routinemäßigen Wartungsarbeiten zum Opfer.

Minimalität. Die relevanten Spuren sind oft in einer großen Menge anderer nicht relevanter Daten verborgen und sind deshalb schwer zu finden. Manche Spuren sind verschlüsselt oder kodiert und können daher mit automatischen Verfahren nicht gefunden werden.

Greifbarkeit. Deduktive Ergebnisse erfordern immer eine Modellierung der Verhältnisse. Wenn die Analyse mittels deduktiver Verfahren gelingen soll, müssen alle relevanten Eigenschaften des Systems korrekt modelliert werden. Darauf muss man vertrauen, wenn man einer deduktiven Analyse Glauben schenkt.

In dieser Arbeit stelle ich eine neue Methode der Einbruchsanalyse vor: Anstatt Spuren zu analysieren und daraus den Ereignisverlauf zu erschließen, finde ich die einbruchsverursachenden Umstände durch automatische Experimente. Wenn das gelingt, kann man den Nachweis einer Einbruchsursache erbringen, indem man einen Testfall ausführt, der nur die als relevant vermuteten Umstände des Angriffs enthält. Sind diese Umstände nicht für den Einbruch verantwortlich, wird der Testfall den Einbruch nicht hervorbringen. Ein solcher Testfall ist unmittelbar greifbar und kann beliebig wiederholt werden. Es ist keine korrekte Modellierung eines Betriebssystems oder anderer komplizierter Laufzeitkomponenten nötig und das Vorhandensein von Spuren ist auch irrelevant.

In dieser Sichtweise ist Einbruchsanalyse ein Minimierungsproblem—wie kann man eine minimale Menge von Umständen finden, die den Einbruch geschehen lassen? Ich entwickle einige effiziente Algorithmen zur Minimierung und gebe sowohl theroretische Eigenschaften an, wie z.B. die Laufzeit im ungünstigsten Fall, als auch empirische Ergebnisse, die das mittlere Laufzeitverhalen beleuchten.

Um Einbrüche mit Experimenten analysieren zu können, benötige ich zunächst eine *Infrastruktur*, die das Aufzeichnen und Wiederabspielen relevanter Systemereignisse ermöglicht. Dazu speichere ich zunächst die Systemaufrufe von Prozessen in einer Datenbank. Nachdem ein Einbruch entdeckt wird, benutze ich diese Systemaufrufe, um Prozesse teilweise wieder einzuspielen, so dass ich herausfinden kann, welche Prozesse den Einbruch verursacht haben—die Ursache-Wirkungs-Kette.

Meine Evaluierung zeigt, dass der Ansatz korrekt und praktikabel ist; er findet die 72 von 168 Prozessen, die für einen echten, komplizierten, mehrstufigen und schwer zu analysierenden Angriff auf *Apache* und *suidperl* verantwortlich sind, in 2,5 Stunden. Dieser Angriff kann mit keinem anderen heute bekannten Werkzeug automatisch aufgedeckt werden.

Ich habe diesen Ansatz erweitert, um auch die einbruchsverursachenden Eingaben dieser Prozesse zu finden—die *Angriffs-Signatur*. Studenten haben unter meiner Leitung für zwei Angriffstypen (Puffer-Überlauf und blockweises Auftreten von

Schadcode) Algorithmen entwickelt, die die Angriffs-Signatur besonders effizient erstellen, wobei sie immer noch korrekt funktionieren, wenn dieser spezielle Angriffstyp nicht vorliegen sollte.

Im Gegensatz zu anderen Signatur-Extraktionsalgorithmen hat dieser Ansatz einige Vorteile. So benötigt er nicht hunderte oder tausende Angriffsinstanzen wie z.B. typische Wurm-Analyseprogramme, sondern funktioniert auch mit einer einzigen Angriffsinstanz. Ausserdem ist er nicht auf bestimmte Angriffsarten festgelegt, sondern analysiert z.B. auch Angriffe, die durch Fehlkonfiguration möglich wurden.

Ich habe mich auch mit dem Problem beschäftigt, vor der Auslieferung von Software diejenigen Komponenten vorherzusagen, die besonders anfällig für Schwachstellen sind. Dazu stelle ich zunächst eine Methode vor, wie Berichte von Schwachstellen („Security Advisories“) denjenigen Komponenten zugeordnet werden können, die verändert werden mussten, um diese Schwachstellen zu beseitigen. Das ermöglicht eine neue Sicht auf Komponenten, welche beispielsweise die Entscheidung zum verstärkten Test oder zum Neuentwurf einer Komponenten unterstützen können.

Ein Indikator für Schwachstellen ist die Domäne, in der sich eine Komponente befindet. Darunter verstehe ich den Bereich, in dem sie Dienste anbietet oder nutzt, also z.B. „grafische Benutzerschnittstelle“, „Datenbankanbindung“ usw. Die Domäne einer Komponente drückt sich in den meisten Programmiersprachen durch Importe aus, mit der Definitionen aus der angebotenen oder verwendeten Domäne dem Übersetzer bekannt gemacht werden, bzw. durch Funktionsaufrufe aus diesen Domänen.

Ich bringe empirische Anhaltspunkte, dass Schwachstellen mit *Importen* und *Funktionsaufrufen* korrelieren. In einer Fallstudie über *Mozilla* zeige ich, dass es Importe gibt, die in 96% aller Fälle mit Schwachstellen korrelieren. In anderen Fällen korrelieren Importe sogar zu 100% mit Schwachstellen.

Mit diesen Informationen erstelle ich einen Prediktor, der unbekannte Komponenten als „schwachstellenbehaftet“ oder als „nicht schwachstellenbehaftet“ klassifiziert. In unserer Evaluierung mit Mozilla konnte ich die Hälfte aller fehlerhaften Komponenten korrekt vorhersagen, wobei über zwei Drittel aller meiner Vorhersagen richtig war. Ausgehend vom Stand Januar 2007 habe ich eine Liste von 10 Komponenten erstellt, die nach unserer Methode besonders anfällig für Schwachstellen waren. Von diesen 10 Komponenten wurden in den folgenden 6 Monaten fünf wegen Schwachstellen geändert.

Veröffentlichungen

Stephan Neuhaus und Andreas Zeller, *Isolating intrusions by automatic experiments*. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium*, Reston, VA, USA, Februar 2006, S. 71–80. Internet Society.

Stephan Neuhaus und Andreas Zeller, *Isolating cause-effect chains in computer systems*. In Wolf-Gideon Bleek, Jörg Rasch, and Heinz Züllighoven, editors, *Software Engineering 2007*, number P-105 in Lecture Notes in Informatics, Bonn, March 2007, S. 169–180. Köllen Druck + Verlag GmbH.

Stephan Neuhaus, Thomas Zimmermann, Christian Holler und Andreas Zeller. *Predicting vulnerable software components*. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, New York, NY, USA, October 2007. ACM Press.

Preface

Working in computer security is working at the limits of computer science. Blackhats, unlike ordinary users, are constantly and actively seeking to undermine the assumptions that we make when we design or deploy systems, and are aggressively trying to find ways to make the system behave in an unexpected way.

This thesis is the result of studying ways to mitigate the damage attackers can do to computer systems. The central assumptions are that attacks happen because of faulty software and that they will remain a fact of life for the forseeable future. The rationale behind these beliefs is explained in Chapter 1.

One promising way to mitigate damage *after* we deploy software is to use experimental methods to find out which interactions of the system with the outside world are relevant for an attack, and which are not. In order to be able to use experimental methods, we need the ability to repeat past interactions at will, and to change what is repeated: we need capture and replay capabilities.

While this work was very interesting, it also had its low points. The reason was that replay was much more difficult than I had anticipated. At more than one time during the implementation of capture and replay, I believed that I had made assumptions that would make it impossible to analyze many interesting classes of attacks. Fortunately, this usually turned out to be wrong, and where it didn't, it turned out to be fixable, and where it wasn't fixable, it could be argued that it was irrelevant or uninteresting.

If we look at the part of the software lifecycle *before* deployment, we can use machine learning methods to find those components of a software system that are most likely to contain vulnerabilities. This was the most rewarding part of my work, since I was presented with an unexpected result: of all the features I looked at, the one best suited was the *import structure* of a component. There is some theory that explains why this might be so in Section 6.6.

Along for the ride was my advisor Andreas Zeller, who was very patient and supportive. Thanks, Andreas! Michael Backes was kind enough to be my second reviewer. Thank you, too! Other passengers included at various times some past and present members of the Software Engineering Group at the Saarland University, notably Martin Burger, Silvia Breu, Holger Cleve, Valentin Dallmeier, Christian Lindig, Rahul Premraj, and Thomas Zimmermann.

Above all, I dedicate this work with gratitude and love to Stefanie.

Saarbrücken, February 2008
Stephan Neuhaus

Chapter 1

Introduction

According to an estimate in the FBI's 2005 Computer Crime Survey, computer crime has cost the US economy a staggering $67 billion in 2005, or about 0.5% of the entire US gross domestic product [59, 74]. The 2006 survey [39] no longer has a national estimate, but the figures do not suggest that the situation has significantly improved. Where do these huge losses come from? According to the report, they are mostly due to "viruses"[1] (65% of all respondents), unauthorized access to network resources (42%), and unauthorized access to information (32%). This means that most of the losses come not from human nature, bad communications, wrong policies and so on, but instead from *faulty software*.

One possibility to reduce or completely eliminate these losses would be to deploy only software that has been rigorously proved to be defect-free. Even if this could be enforced—and there is no evidence that it could—, there are several drawbacks with this approach:

- Specifying software and proving it correct slows down the software's time to market, a penalty that most software companies would rather not incur, given today's market pressures. In the words of one specialist, "If they don't have to do it—for example in order to get a certificate—, they won't do it." [153]
- It would work only or future software, but not for most legacy software because most legacy software is generally without a formal specification. It is certainly possible to create specifications retroactively. But software that is already deployed without prior specification will have specifications that are much more complicated than those that are created before the software is written. Such specifications will therefore tend to take even longer than pre-deployment specifications. Whether such post-deployment specifications are at all amenable to automatic proofs is unclear. Experience within the Verisoft project [167] with AMD's Hypervisor—a hardware project—indicate that it may be possible, but only at the expense of significant manpower by exceptionally well-trained personnel [98]. Hardware is in general much better specified than software because the deployment of fixes—in this case, defect-free processors—is much more difficult than for software: one cannot download a new processor over the Internet.

[1]One suspects that the CSI and FBI lump together all kinds of malware under the descriptive title of "viruses" here.

- Rewriting such software is also out of the question most of the time because of the huge investment in time, money and training that went into the creation and deployment of the original software.
- Even software that has been rigorously proved to be defect-free and secure may not be immune from practical attacks. For example, an encryption algorithm may have perfect secrecy, yet be vulnerable to timing attacks as done by Dan Bernstein on AES [21], or by David Brumley and Dan Boneh on SSH [25]. As David Basin put it: "You cannot neglect best practices just because you have proved something to be correct." [19]

Therefore, if we want to reduce the damage done by security incidents due to faulty software, we can do that on two fronts:

After deployment. If we were able to *analyze* attacks automatically, we could enable vulnerable computers to analyze attacks and then prevent them automatically (Chapters 2, 4, and 5).

Before deployment. If we were able to *predict* which components of a program were particularly prone to vulnerabilities, we could direct our quality assurance efforts at these components. For example, we might expose the components in question to source code review and security assessment or penetration testing [52], or prove them correct or redesign them in a more secure manner. If we make good predictions, we can risk testing the other less vulnerable components less (Chapter 6).

Chapter 4 also provides a stand-alone framework for the analysis of minimzation algorithms and uses this framework to analyze several algorithms such as delta debugging and two others that were developed to deal with the demands of minimization when replaying processes. We finish this thesis with a summary and conclusions (Chapter 7).

The two projects that gave rise to this work can be found at

`http://www.st.cs.uni-sb.de/malfor/`

and

`http://www.st.cs.uni-sb.de/softevo/`

Chapter 2

Causes And Effects

How often have I said to you
that when you have eliminated the impossible,
whatever remains, however improbable,
must be the truth?

—Sherlock Holmes
in Arthur Conan Doyle, *The Sign of the Four*, 1890

Debugging systems of interconnected programs is usually done in an *ad-hoc* manner. For single programs, there is quite some theory to explain how failures come to be, how they propagate, and how they manifest [185], but there is little theory to explain the causes of *system* failures. There is also little support: not only are there preciously few tools that allow debugging systems of programs, but the programs being debugged also often offer the debugger only little help, for example by providing clear and meaningful log files.

2.1 Analysis of Break-Ins

2.1.1 Forensic Analysis of Break-ins

The traditional method of answering the question of how a break-in came to be is to pore over *log files* in the hope that they contain enough clues to help us discern what must have happened. Traces of erased files and other evidence can also help. This is called "forensic analysis", and there exist some excellent tools to aid the analyst [32, 58], mainly in the process of gathering and collating evidence.

Even though there have been attempts to automate this analysis [53, 90], forensics is by and large done manually and there is currently not much research that tries to automate the analysis process.

For example, the Communications of the ACM has devoted much of its February 2006 issue to forensics. Of the articles in the issue, two were on live forensics [5, 31], three on formats and standardization [38, 67, 80], one on visualization [154], and two on future aspects of forensics [22, 142]. Only one was on the technique of constructing the event chain for a sophisticated intrusion [33]. The manual character of such investigations is evident in sentences such as "Digital investigators are continually seeking more effective ways to process and visualize network logs to identify suspicious activities and recover data from covert, encoded network traffic.", or "Once a potential source of evidence has been preserved, digital investigators immediately begin dissecting it for information pertaining to the intrusion." These quotes mean that forensics is still more of a craft than an automated technique.

Even evidence gathering (as opposed to analysis) is becoming more difficult. Given the adversarial nature of evidence—it might be hidden or disguised—much of the available material will need to be inspected manually in order to see whether it contains usable evidence. However, given the sheer size of today's hard disks and process memories, manual inspection is simply no longer feasible.

Some attacks may not leave any usable evidence at all.

Finally, even if we could gather all the evidence, we might draw the wrong conclusion from it. For example, we could easily believe that if a file was changed, there must have been a process that has opened it. As we will see in Chapter 5.3, this is not necessarily the case: a loadable kernel module can modify a file without the corresponding *open* call.

The reason why such mistakes happen is that deductive reasoning always requires a previous *modeling* step in which facts from the real world are abstracted away so that the only remaining things are those that are presumably of interest to the problem at hand. For example, the fact that computers are real, physical things that take up space and have weight is usually deemed irrelevant for security analysis and consequently does not appear in any security model. In the words of Steve Bellovin [20]:

> Any proof depends on your axioms or system model. In security, though, an attacker can often attack via a different model. Thus, we may have ciphers that are fine at the 0s and 1s level but are vulnerable to things like differential power analysis, cache timing, etc. Even at the 0s and 1s level, did the proof of security account for things like related-key attacks?
>
> Mathematicians have known since Euclid that axioms are important. Security, though, is math embedded in the real world, and that matters. Put another way, Euclidean geometry is completely valid as a pure mathematical system. But that doesn't mean it applies in a relativistic universe. Sure, we live far from any space-warping masses, so we can pretend that the angles in our triangles add up to 180 degrees. In the security world, though, the attacker will toss a black hole at us to warp the space around our provably-secure triangular encryptor. Was that proof of security flawed? Ask Riemann or Lobachevsky.

More concretely, Dolev-Yao models [51] are used in the analysis of security protocols. They abstract away many details of concrete encryption and hash functions and work only by rewriting abstract terms. A protocol that is deemed safe according to that model may not be so in real life, for example because the encryption algorithm may be susceptible to a timing attack. Therefore, if the abstraction used in the model is not suitable for the attack that we wish to analyze, we may get the wrong results.

2.1.2 Cause-Finding as Minimization

The approach outlined in this thesis is to work *experimentally*. First, we capture enough information so that the system—or those parts of the system that we are currently interested in—can be replayed. That takes care of evidence gathering: if we assume that the system is in an uninfected state at the beginning of capturing and infected at the end, the transition must be somewhere in between. If we can replay the system using the captured information, we can gather evidence post facto if need be, or we can redeclare the captured information to be the evidence.

Once we have captured failure-causing information, we can then replay subsets of the captured system and thus see whether those replayed parts are relevant for the intrusion or not. The goal is thus to separate the relevant from the irrelevant information. Once this is accomplished, we are left only with relevant information, which we call the *cause* of the break-in. The justification for this is the counterfactual or "what if" definition of causality: we say that a set of events A causes an event b—called the *effect*—if b would not have happened if the events in A hadn't occurred. This is the *counterfactual* definition of causality.

For our work, we adopt the *closest-world condition* of the counterfactual definition and call A the *actual cause* of b if A is the (or a) smallest and most recent set of circumstances that we have to remove so that b does not happen. By extension, we call a finite sequence $(C_1, \ldots, C_n)$ of causes a *cause-effect chain* if C_k is the actual cause of C_{k+1} for $1 \leq k < n$. More discussion follows below in Section 2.1.3

Actual causes need not be unique. For example, if two stones are thrown at a glass bottle so that they arrive there simultaneously, either one could be the actual cause. However, two actual causes will always have the same number of events; otherwise the larger set is not a cause because it violates the closest-world condition.

In our formulation, finding a cause is equivalent to finding a smallest set of circumstances that will makes the effect go away if removed, yet will keep as many of the original circumstances as possible. In other words, finding a cause is the problem of minimizing a set of intrusion-causing circumstances. This new view of causation as minimization for cause-finding within computer programs was first developed by Andreas Zeller [185].

Cause-finding can be seen as a minimization problem.

Using a suitable minimization algorithm M, we then arrive at the following informal method for cause-finding:

1. Let M choose a subset of the effect-causing circumstances.
2. Build a world in which only these circumstances happen.
3. See whether the effect still occurs.
4. If it does not occur, take this subset as the new cause and go to step 2.
5. If the subset cannot be minimized any further, terminate. The current subset is then the actual cause. Otherwise, go to step 1.

This method will be hard to use in the real world because it will in general be impossible to exactly recreate something that is already past. Luckily, that is not always necessary: it will suffice to recreate enough circumstances so that the effect can still occur. This is rather easy for computer programs because we can control everything that they input, plus all aspects of the computation environment such as thread schedules and so on.

The above algorithm embodies the *scientific method*: step 1 is the formulation of a hypothesis, step 3 is the test by experiment of that hypothesis, and steps 4 and 5 refine the hypothesis, both when the experiment supports and contradicts it.[1] The explicit use of experimental methods to analyze attacks is also new.

The main advantages of this method is that the resulting minimized set has been *experimentally shown* to be relevant: the result was obtained not by deduction, but by trying the circumstances and seeing what actually happens. This lends the results a credibility and a tangibility that deductive results do not have. In order to trust in a deductive result, an act of *belief* is necessary, namely that the principles on which the deduction is founded are correct and that the deduction itself happened according to the laws of logic. This belief is necessary even when we feel competent enough to check the deduction ourselves; in this case, it is the belief that we are able to deduce logically and to assess the validity of the assumptions underlying the deduction.

This is not so with results that have been obtained in an experimental way because they contain a *test case* that can be replayed. For example, if I claim that a certain misconfiguration in Apache was the cause of an attack, and if someone tries it out, and it does not happen, then my assertion must have been wrong. It is hard to argue with facts.

To summarize, even if attack analysis were not done manually, these are the problems when using deduction on evidence:

Completeness. There might not be enough evidence for the cause to be reliably established.

Minimality. The relevant evidence might be buried in a host of other evidence and may thus be hard to see.

Correctness. Our reasoning (by human or machine) might be faulty, leading to wrong conclusions.

Tangibility. Experimental results can simply be tried out if we want to confirm their validity.

In this work, we will apply this idea to two related problems of incident analysis in order to enable partially automated diagnoses of security incidents. The first problem is to find the processes that were involved in an attack. The second problem is that of finding the input that caused the attack, the *attack signature*. Both problems are tackled and (partly) solved in Chapter 5.

2.1.3 Philosophical Justification

Through the centuries, many philosophers have tried to understand what causation is. How can we be sure that our definition is a good one? Our discussion of counterfactual causality is based in part on the discussion in *The Stanford Encyclopedia of Philosophy* [114].

[1]It is by no means clear, however, that scientists actually use the scientific method. Almost all philosophers who have tackled this question seem to be of the opinion that they don't. See Appendix A for some discussion.

The counterfactual definition of causality was first proposed by David Hume in his *An Enquiry concerning Human Understanding* [81, Part II]:

> We may define a cause to be an object followed by another, and where all the objects, similar to the first, are followed by objects similar to the second. Or, in other words, where, if the first object had not been, the second never had existed.[2]

One important philosophical and practical problem with this definition is that it works only by arguing about events that did in fact not happen. This makes it difficult to talk about causality in a strictly empirical setting where every claim must be substantiated by experiments. After all, arguing about what would have happened if A had not happened is futile because A did in fact happen, time has moved on, and we cannot turn the clock back.

This problem was solved in 1973 by David Lewis [99, 100] by proposing the existence of possible worlds, worlds that are similar to ours. By ordering the worlds with respect to their degree of similarity to ours, he creates a partial ordering of worlds. For an *antecedent* A and a *consequent* C, he next introduces a truth condition for the counterfactual "If A were (or had been) the case, C would be (or have been) the case", written $A \hookrightarrow C$. In his definiton, $A \hookrightarrow C$ is true in the actual world if any of the two following conditions hold:

1. there are no possible A-worlds; or
2. there is an A-world in which C holds and that is closer to the actual world than any A-world in which C does not hold. (This is called the *closest-world condition* of counterfactual causality.)

The basic idea behind this definition is that $A \hookrightarrow C$ is true if "it takes less of a departure from actuality to make the antecedent true along with the consequent than to make the antecedent true without the consequent." [114]

For distinct and possible event sets A and a possible event b, he defines that b *causally depends* on A if and only if

1. "A occurs" $\hookrightarrow$ "b occurs"; and
2. "A does not occur" $\hookrightarrow$ "b does not occur".

If A and b are actual events, the first condition becomes trivially true because there *is* an A-world in which b holds and that is closer to the actual world than any A-world in which b does not hold: this is the actual world. The definition then simplifies to:

Definition 2.1 (Counterfactual Causality of Events). Let A be a set of distinct and possible events, and let b be a possible event, distinct from any event in A. We say that b *causally depends* on A if and only if "A does not occur" $\hookrightarrow$ "b does not occur".

[2]The formulation "in other words" suggests that Hume thought that the two sentences were equivalent, when they are in fact not. The first definition says that if A or something similar happens, then b or something similar happens always. The second definition says that if A does not happen, then b also does not happen.

This definition takes care of problems that occur with simpler attempts at defining causality. For example, if it is true that if *A* didn't happen then *b* wouldn't have happened, we might call *A* the cause of *b*. In one example, let *b* be "The Second World War", and let *A* be, for example, {"Existence of oxygen"} or something even more ridiculous like {"Existence of the universe"}. Certainly, the Second World War couldn't have happened without a universe for it to happen in, but that is not what we mean by cause. The closest-world condition rules out such non-causes.

The counterfactual definition of causality is not the only possible one. Taking David Hume's attempt at defining causality above, for example, another possible definition is to say that causes are invariably followed by their effects. This is known as the *regularity definition* of causality.

The biggest difficulty with that definition is that it contradicts common sense. For example, most people today accept that smoking is a cause of lung cancer. Yet not all smokers develop lung cancer and not all non-smokers do not develop it. Also, a deterministic world-view is no longer compatible with what we know from physics about how the world apparently works. For example, the Heisenberg inequality says that there is an inherent limitation to the accuracy with which we can know a particle's position and momentum simultaneously, a limitation that is apparently built into the world itself and that is not merely due to imperfect measuring equipment.

Another problem is that a common cause for two effects may lead one to conclude that the two effects are causally related. For example, a drop in atmospheric pressure may lead to the dropping of the mercury column in a barometer, and later to a storm. Under the regularity definition, we would have to conclude that the drop in the barometer causes the storm.

This gives rise to the *probabilistic definition* of causality; see for example the book by Judea Pearl [132]. In this definition, "causes raise the probability of their effects; an effect may still occur in the absence of a cause or fail to occur in its presence." [76]

The biggest problem with this definition is that it connects causation with correlation. For example, there is a well-known correlation between birth rate and stork population; one recent result is by Thomas Höfer et al. [77]. Under a simple probabilistic interpretation, we would be forced to conclude that storks bring babies. Another problem is that causes and effects are not necessarily asymmetric under that interpretation. Common sense dictates that causes precede their effects[3] and that therefore effects cannot cause causes. Under the simple probabilistic interpretation, correlation works both ways, so a rise birth rate may also be said to cause a rise in the stork population.

There are ways to fix these problems, but then the definition and the consequences of probabilistic causation become so complicated that they are no longer easily understood.

There are two questions that we must answer if we want to know whether the counterfactual definition of causality is suitable for intrusion analysis:

1. Does the definition satisfy everyday notions of causality as far as they relate to intrusion analysis? Or are there hidden subtleties that may make the understanding of causes difficult?

[3]Even though this presents a problem for some philosophers because it "rules out the possibility of backwards-in-time causation *a priori*" [76]. Why that would be a problem eludes me, however.

2. Does the definition enable us to decide causality in an practical and effective way? Or is the definition merely suitable for philosophical discussions?

First, the counterfactual definition does indeed satisfy everyday notions of causality. There are some slight subtleties, though: Consider for example a glass bottle and two stones that are thrown at it so that they arrive there one after the other.[4] In the counterfactual sense, both stones are the cause of the glass bottle shattering, because merely removing the stone that arrives first will not lead to the glass bottle staying whole. Common sense, however, would say that the cause of the shattering glass bottle is the first stone, not both. Subtleties like these do not seriously distort the notion of causality, however.

Other definitions, such as the regularity definition or the probabilistic definition, do not add to our understanding of intrusion analysis, because we are dealing with actual events, events that have already happened. We are not interested in whether our cause *always* or *very probably* produces the observed effect; we are content if it produces the effect only under the circumstances at hand. In fact, showing only that the cause very probably produces the effect might make the whole method unsuitable for forensic analysis, because it raises questions about reasonable doubt.

Second, the definition does enable us to decide causality in an engineering setting. While it is true that we cannot turn the clock back in order to remove the putative cause and to see whether the effect now vanishes, we *can* make experiments under conditions that are close enough to the conditions under which the intrusion happened that results will be consistent with counterfactual causality.

The counterfactual definition of causality is well suited for intrusion analysis (and, we suppose, for other similar post-facto *cause-finding enterprises).*

2.2 Related Work

There is an abundance of related work in the area of intrusion detection and analysis, both commercial and academic, and we can show only a fraction of what is available.[5]

2.2.1 Tools

Tools to help analyze security incidents come in different levels of abstraction. On the bottom rung are tools that display or collate information that is difficult to view otherwise. For example, there are tools to analyze which programs run at system startup such as autoruns [144]. There are also binary file viewers and editors such as the binary stream editor bsed [54]; bintext, which finds ASCII and Unicode strings in files [63]; or bview, a binary file viewer [94].

Next, there are programs that look at browser information such as CacheView to look at caches [85]; CacheInf that can additionally manipulate them [174]; or CookieView that looks at cookies [182]. On the server side there are log file analyzers such as webalyzer that computes web server statistics for Apache [18]; AWstats that generates graphical usage statistics not only for web servers, but also for ftp and mail

[4]In the reference frame of the glass bottle. In our discussion, we will consistently disregard relativistic effects, which would in any case merely complicate but not invalidate our arguments.

[5]If a product is mentioned here, it does not mean that we endorse this product or that some other product might not suit a particular purpose better; the products are mainly included to demonstrate that there are many of them. We are not affiliated with any company that makes or markets such tools.

servers [48]; Eventlog that looks a Windows NT event logs [112]; http-analyze that computes web statistics for a number of web servers [140]; Analog, another web log analyzer [160]; or 123 Log Analyzer, a multi-platform web log analyzer [186].

Furthermore, there are lots of tools to recover lost passwords, for example Act-Mon Password Recovery XP, that reveals the password that lurks behind a row of dots in an input box in Windows [3]; CmosPwd that recovers BIOS passwords [37]; Lastbit, a password recovery suite for a host of Windows programs [97]; or Passkit, a suite that recovers even more passwords [105].

On the next rung are programs that look at filesystem-specific information such as Directory Snoop, a file recovery and secure deletion tool [23][6]; CopyADS, which looks at so-called alternate data streams (ADSs), a feature of NTFS that hides information from all but the most specialized programs [109]; Disk Investigator, a tool that investigates the raw contents of a disk [152]; and Crckit, which produces verious checksums [110].

Disk images are useful when analyzing the forensic traces of a break-in: usually, disks of the attacked system is imaged and the analysis confined to the disk image, for fear of altering the evidence. There are several programs to create disk images, such as Active Disk Image [2], Drive Image [139], or Drive SnapShot [156]. However, most programs create images for backup purposes, and few are usable for forensic purposes. Of those few, there are Diskimag that creates forensic copies of floppy disks [111], or X-Ways Forensics, which can do the same for hard drives [183].

Most of these tools are geared toward presentation or recovery of data on a disk and do not directly support the forensic workflow or do detection.

On the next rung on the ladder are rootkit checkers like chkrootkit [36] and other host-based intrusion detection software like Tripwire [157]. These tools examine the state of a system for obvious policy violations, and are thus an important layer of any well-administrated system's security. However, they cannot find out how this state came into existence.

Still higher up there are toolkits that support the workflows associated with forensic analysis and discovery. There are far fewer such toolkits than there are isolated tools. One example of such a toolkit is The Coroner's Toolkit [57, 58], plus extensions such as SleuthKit [30, 29], or graphical user interfaces like Autopsy [28].

As a further example of workflow support, consider the Kerf toolkit [11]. It explicitly supports hypothesis generation and refinement, but does not use any experimental techniques to test new hypotheses. The project's goals are the support of "interaction" (with a human system administrator), "iteration" (of hypothesis generation), and "collaboration" (with other administrators), but do not include automation or experimentation [12].

[6]This program incidentally falls for the "35-pass fallacy". This fallacy arises when people misunderstand an article by Peter Gutmann called "Secure Deletion of Data From Magnetic and Solid-State Media" [71]. In this article, he outlines a way to use knowledge about a hard disk's encoding process in order to develop patterns that would magnetize the disk's platter deeply, thereby erasing any information on a disk. He illustrated his method for MFM (Modified Frequency Modulation) and RLL (Run Length Limited), two encoding methods which were in use in the 1980s and 1990s. This analysis led to a method with 35 passes. Unfortunately, this paper led several security software vendors to think of those passes as a magic voodoo that simply had to be done. Hence the 35-pass method is available in almost all disk-wiping tools, no matter that current methods like PRML (Partial Response Maximum Likelihood) are no longer susceptible to *any* pattern-based method. Gutmann even descibes this on his web page, but to no effect. The most recent company to fall for the 35-pass fallacy was Apple with its Disk Utility. This utility allows one to overwrite a disk before formatting it: the option labeled "most secure" had 35 passes...

Existing toolkits do not help the investigator interpret forensic data to find the chain of events that led to the security incident.

2.2.2 Research in Intrusion Analysis

When there is tool support for, or research focusing on, the *automated* analysis of security incidents, it is *deductive*. In deductive approaches, the evidence is matched with a model of the system under investigation in order to mimic the way in which a human investigator would operate.

In recent times, there has been a large body of research that aims at automatically generating attack signatures by various means, such as vulnerability-based (instead of exploit-based) signatures [26], network-based (instead of host-based) signatures [102], noise introduction to fool automatic signature generators [131], forensic memory analysis [103], or finding zero-day vulnerabilities by symbolic execution and predicate testing [43]. An *attack signature* is a small substring of a network request that will cause an intrusion and that is somehow characteristic of that particular intrusion. Obviously, this is taking a large step towards the automatic analysis of security incidents, but it is vulnerability-specific. What this means is that these analysis tools are geared towards analyzing a specific kind of vulnerability, such as buffer overflows.

King and others try to answer the more general question of the *cause-effect chain* that led to the incident. To this end, they use the capturing part of a capture and replay tool called ReVirt [53] to capture the system calls of processes. Once an intrusion is detected, they use this captured information in a tool called Backtracker to construct dependencies between system objects such as processes or files [90]. For example, if a process writes to the password file another process later reads from it, the second process depends on the former through the password file. By following the dependencides from an object of interest backwards, they find one or more chains of events that are dependencies of the object of interest. For example, if there is a process running with system administrator privileges when it shouldn't, they could find that it has these privileges because the password file was modified and now contains a second administrator account. They then analyze which process last modified the password file, and so on, all the way back to *init*.

The underlying assumption is that all security-relevant changes in the state of the analyzed machine must have been caused by system calls. But in fact, this is not true: We have written an attack that works by inserting a back door (a new administrator account in the password file) from a loadable kernel module [124]; see also Section 5.3.2. Since the password file is modified from within the kernel, there is no system call that ever modifies the password file. Backtracker cannot analyze this type of attack.

This particular weakness is not restricted to Backtracker; any deductive tool could suffer from a similar flaw. This is because a model is always a simplification of the modelled system and thus necessarily leaves out some parts of it.[7]

Attacks that fall outside of forseen models will be hard to analyze with deductive approaches.

[7]"The map is not the territory." [92]

2.2.3 Research in Intrusion Prevention

Can we prevent incidents through infrastructure measures or through analyzing source code? There is a plethora of work on intrusion prevention. For software-based methods, we cite only work on preventing buffer overflows, leaving out work on code injection and other attacks for reasons of space. In general, research on prevention has so far focused mostly on narrow, well-understood intrusion methods such as buffer overflows or code injection. These methods work either statically (that is, without running the program) by:

- detection of memory errors by using annotations [56];
- detecting incorrect pointer usage by axiomatization and subsequent checking of usage against that axiomatization [64];
- analyzing semantic comments to detect buffer overruns [96];
- symbolic methods for bounds checking [143] or memory leak detection [146];
- using type qualifiers to detect format string vulnerabilities [149];
- scanning sounce code for bad patterns [168, 169];
- transforming the source code to check all pointer accesses (this is a hybrid between a static and a dynamic technique: the technique works statically on the source code but the flaw is detected at run time) [14]
- finding buffer overruns by static bounds checking through integer range analysis (finding out if accesses could happen outside legal bounds) [173].

Or they may work dynamically (that is, at run time) by:

- preventing buffer overflows by intercepting library calls and containing any eventual buffer overflows [16, 17];
- preventing buffer overflows by rewriting the binary and forcing verification of stack contents before use; also [16, 17];
- preventing buffer overflows by implementing bounds checking or other measures in the compiler [24, 41, 55, 86];
- patching the runtime library to prevent format vulnerabilities [40];
- linking with a special library to prevent format string vulnerabilities [158];
- using fault injection for security analysis [69]
- saving the return address in an inaccessible location and restoring it immediately before function return [165, 166].

Unfortunately, a method that is geared at preventing a certain kind of intrusion method (such as stack-based buffer overflows) will in general not work against other, fundamentally similar methods that work slightly differently (such as heap-based buffer overflows): authors usually exaggerate when they claim that their method will prevent “unknown” intrusions. (There are exceptions to this general rule: for example, TaintCheck [128] will in general find all overflows. The price here is non-negligible performance overhead.)

Dynamic methods also suffer from the problem of what to do when they detect a buffer overflow. Most tools have no domain knowledge and thus cannot know what the correct response would be. The most common approach seems to be to terminate the process, but it is far from clear that this—or indeed any other approach that treats all processes equally—solves more problems than it creates. For example, the *cryptography* mailing list has had a long discussion about whether it is better to fail with an error code or to exit the application without giving it a chance to clean up [44]. The consensus, after many days of heated argument, was that one really can't say in general what the best course of action must be.

There are also anomaly detection methods that use statistical models to distinguish normal from abnormal behaviour; for recent examples, see for example PLAYL [175], and extensions to anomaly-based detection methods to intrusion prevention, such as Autograph [89], or Shadow Honeypots [8]. These techniques work very well, provided that the underlying classification identifies suspicious traffic sufficiently well. If an attack does manage to slip through, however, analysis of the incident is still manual.

There have been recent advances in self-healing software; see for example the work by Locasto and others on Application Communities [104]. They propose to turn the use of a software monoculture from a liability into an asset by enabling it to detect flaws in a distributed fashion and to heal itself from detected exploits. This works well on massively deployed software that is subject to massive well-understood attacks that are automatically detectable as the program runs (such as buffer overflows by by worms). However, if one of these prerequisites is not present, such as with a code injection attack on a custom web shop on a single server, the method will not work at all.

We will have to live with intrusions for some time.

2.2.4 Verification and Formal Methods

What about avoiding security incidents from the outset? There has been a large body of work on the verification of security-relevant software systems. Realistically, this would require the entire execution chain from the processor to the operating system, the compiler and the application to be proved.[8] This is such a massive undertaking that it has only recently been attempted in the Verisoft project [167]. At any rate, these approaches cannot be retrofitted to existing systems such as Linux or Windows because they have no formal specifications and cannot easily be equipped with one. Also, the languages used in implementing these systems (mostly C and C++) may have formal semantics [177], but the problems posed by aliasing (two pointers referring to the same address) and other difficulties preclude efficient static verification of full-blown C and C++ programs.

Also, the adherence of a software system to a formal specification is not in itself a guarantee of security: if the specification is faulty, the program can be proved correct but still be insecure. It is very difficult to specify the security properties of a software system so that the specification captures the intent and is amenable to mechanical verification. For example, even if secrecy conditions are correctly

[8]In the words of Peter Gutmann, there need to be "turtles all the way down" [72].

specified for an encryption scheme, there might still be practical remote timing attacks [21, 25, 129]. Also, formal specification of security features will usually need to be done by specially trained personnel, which will not usually be available. Gutmann's book chapter on verification techniques [72, Chapter 4] contains many more examples of the deployment of formal methods in security-related fields (and their subsequent failure to deliver on their promises).

Formal methods may also have practical limitations: recent results by Backes and others have additionally shown that it is not possible to use the Dolev-Yao model, a certain well-known model of (cryptographic) security, to prove that real cryptographic primitives such as hash functions or the exclusive-or operation have certain desirable properties, such as security under composition [15]. We have currently no way of proving important cryptographic primitives secure, and therefore cannot verify or prove the security of implementations that use these primitives.

Sometimes, security proofs are impossible with current methods.

Chapter 3

Minimization: State of the Art

Reason is not measured by size or height, but by principle.
—Epictetus, *Discourses*, Chapter xii

Beauty depends on size as well as symmetry.
—Aristotle, *Poetics*, Chapter 7, Section 4

Once we have found an intrusion we want to know which processes have caused it. One could argue that all processes have caused the intrusion, and while this is certainly true, it is not particularly helpful. What we want instead is a *minimal* set of processes that cause the intrusion. This problem is best tackled in its generality first. In this chapter, we show the state of the art in minimization and postpone its adaptation to intrusion analysis to the next chapter.

3.1 Preliminary Definitions

Before we embark on a quest to find good minimization algorithms, it is useful to define the problem these algorithms should solve.

The idea is that in order to perform experiments to refine hypotheses, we need an oracle that tells us whether a particular set of circumstances leads to the effect that we are currently analyzing. In practice, such an oracle would consist of taking the circumstances and actually carrying out the experiments, but this is irrelevant to the *theoretical* discussion here.

The general idea is that we have some circumstances that in whole produce some kind of failure, but where the failure does not appear when those circumstances are wholly missing. It follows that there must be some smallest subset of the set of circumstances that produce the failure. The terms below were first defined by Zeller and Hildebrandt [75].

Definition 3.1 (Test Function, Outcome, Failure-Causing Circumstances). Let C be a finite set, called the *set of circumstances*. The *power set* of C is the set of all subsets of C and is written $\mathcal{P}(C)$. A function test: $\mathcal{P}(C) \to \{✔, ✘, ?\}$ is called a *test function* if $\text{test}(\emptyset) = ✔$. The value of test is called the *outcome of the test*, and the

```
<SELECT>     <!-- <SELECT> -->     <!-- example -->
                                   <SELECT>
                                   <!-- end example -->
```

Figure 3.1: Three HTML documents. The left document caused Mozilla to crash when printing it was attempted. The middle document does not cause a crash, even though it is a superset of the first. The right document again causes the crash, even though it is a superset of the second. It follows that failure-causing circumstances will generally not remain failure-causing in the presence of other circumstances.

various outcomes are called *passing* (✔), *failing* (✘), and *unresolved* (**?**), respectively. Any set $F \subseteq C$ with $\text{test}(F) = ✘$ is called a *failure-causing* set of circumstances *with respect to* test.

The condition $\text{test}(\emptyset) = ✔$ models the requirement that applying none of the circumstances will produce no failure. Note that *any* function that assigns ✔, ✘, or **?** to a subset of C is a test function if only the outcome of the empty set is passing. It is not required that there be any internal consistency to the test. For example, one could assume that if $\text{test}(F) = ✘$, then $\text{test}(F') = ✘$ for every nonempty $F' \subseteq F$, but this is not necessarily true in real life. For example, consider the case of the failing Mozilla browser described by Zeller and Hildebrandt [75]. In their case, the presence of the HTML tag '`<SELECT>`' would make the browser fail when trying to print the page, but the presence of HTML comments could make supersets of this minimal document pass or fail; see Figure 3.1.[1]

Once we have found a set F of failure-causing circumstances, we are interested in finding a minimal set $F' \subseteq F$ that is also failure-causing. Such a set can be said to be *the* cause for the failure; we call such a set a *culprit set* or simply *culprit*:

Definition 3.2 (Culprits). Let F be a set of failure-causing circumstances with respect to some test function test. A set $F' \subseteq F$ with

$$\text{test}(F') = ✘; \tag{3.1}$$

$$\text{test}(G) \neq ✘ \qquad \text{for every } G \text{ with } G \subseteq F \text{ and } |G| < |F'| \tag{3.2}$$

is called a *culprit set of F with respect to* test.

Lemma 3.1. Culprits are non-empty.

Proof. Assume that $\emptyset$ could be a culprit. Then Equation 3.1 of Definition 3.2 stipulates that $\text{test}(\emptyset) = ✘$, but Definition 3.1 says that $\text{test}(\emptyset) = ✔$. □

Lemma 3.2. Culprits need not be unique.

Proof. It suffices to give a counterexample where two minimal failure-causing subsets exist. Let $C = \{a, b\}$ and let $\text{test}(S) = ✘$ if and only if $|S| \geq 1$. Then both $\{a\}$ and $\{b\}$ are culprits with respect to test. □

That culprits are not unique is not just a theoretical consequence of the definition. In practice, there often exist several minimal test cases that make a program fail,

[1]This example is due to Andreas Zeller.

such as when a compiler crashes with a certain input file. Renaming all identifiers in the program will usually also crash the compiler. (Hildebrandt and Zeller give an example where a compiler crash remains after their automated debugging method had inadvertently renamed an indentifier [75]. They conclude that in this case, the specific value of the identifier is not relevant for the compiler crash.)

While culprits are not unique, they *are* all of the same size.

Lemma 3.3. Let F_1 and F_2 be two culprit sets of F with respect to test. Then $|F_1| = |F_2|$.

Proof. Assume that this were not so, and assume without loss of generality that $|F_1| < |F_2|$. Then, according to condition 3.2 of Definition 3.2, F_2 cannot be a culprit. □

From now on, we will treat the notation '$\mathrm{culp}_{\mathrm{test}}\, C$' as a shorthand for 'any convenient member of the set of all minimal failure-causing circumstances'. Usually, the test function is implicitly understood, so we will often simply write '$\mathrm{culp}\, C$'.[2]

3.2 The Minimization Problem

If we try to find $\mathrm{culp}\, C$, one possibility is simply to evaluate $\mathrm{test}(S)$ for subsets of C in some order. In fact, it is impossible to compute $\mathrm{culp}\, C$ otherwise than by repeated evaluations of the test function because the definition of test function has so few constraints: a minimizing algorithm cannot infer the value of $\mathrm{test}(S)$ given S and the values of arbitrary many values of $\mathrm{test}(T)$ with $S \neq T$, other than $\mathrm{test}(\emptyset) =$ ✔ and $\mathrm{test}(C) =$ ✘. Consider on the contrary an algorithm that does just that. We then construct another test function test' such that $\mathrm{test}(T) = \mathrm{test}'(T)$ for all considered sets T, but where $\mathrm{test}(S) \neq \mathrm{test}'(S)$. This is a test function in the sense of Definition 3.1 since $S \neq \emptyset$. The algorithm, having seen the same input—the value of S, the T's and the values of $\mathrm{test}(T)$—must produce two different outputs, which is impossible.

Because this is so, it is not surprising that computing $\mathrm{culp}\, C$ is a hard problem.

Lemma 3.4 (Optimal Solution takes Exponential Time). Let $|C| = n$ and let $\mathrm{culp}\, C$ be equally distributed in $\mathcal{P}(C)$. Then, finding $\mathrm{culp}\, C$ by evaluating test on subsets of C takes $O(2^n)$ time.

Proof. We prove this lemma by proving that any algorithm that computes $\mathrm{culp}\, C$ must consider all subsets S of C with $|S| < |\mathrm{culp}\, C|$. If then $\mathrm{culp}\, C$ is evenly distributed in $\mathcal{P}(C)$, the algorithm will have to consider $0.5 \cdot (2^n - 2)$ subsets on average.

Let $|C| = n$, and let an algorithm consider subsets of C in the order C_1, C_2, $\ldots$, $C_k = \mathrm{culp}\, C$, where $k \leq 2^n$. Assume that there is some nonempty $T \subset C$ with $|T| < |C_k|$ which has not been considered by the algorithm (that is, there exists no $1 \leq j < k$ with $C_j = T$). Since the only requirement for a test function is that $\mathrm{test}(\emptyset) =$ ✔, the algorithm cannot infer that $\mathrm{test}(T) \neq$ ✘ and therefore cannot conclude that $C_k = \mathrm{culp}\, C$. This is a contradiction, which in turn means that the sequence C_1, $\ldots$, C_k must contain the nonempty proper subsets of C in non-decreasing order.

[2]A previous version of this document had $\mathrm{cul}\, S$ as the notation for a culprit set. I am indebted to Andreas Zeller who showed me that this formulation led to a rather unfortunate and unintended double entendre in French.

Since there are $2^n - 2$ subsets of C that are not either $\emptyset$ or C, and since $\text{culp}\, C$ is equally distributed in $\mathcal{P}(C)$ by assumption, such an algorithm will have to consider half of the number of subsets on average before finding $\text{culp}\, C$. □

One consequence of this lemma is that we will generally have to settle for something less than the full culprit set if we want to compute the answer in reasonable time. Either we will have to restrict ourselves to test functions with more constraints (so that we can infer the value of the test function without having to compute it), or we will have to relax the minimality requirement. In the general case that we are considering here, we cannot constrain the test function because there are reasonable cases—such as the one shown in Figure 3.1—where the test function is essentially unconstrained. We will therefore first try to relax the minimality requirement. In this case, we can get a reasonably small failure-inducing set with quadratic complexity.

In the next chapter, we will see that intrusion analysis offers constraints on the test function that make it possible to compute $\text{culp}\, C$ in $O(|\text{culp}\, C| \log |C|)$ time.

One fairly natural way to relax the minimality requirement is to constrain the ways in which we make a failure-causing set smaller. If we try to make it smaller by removing single elements and don't succeed, we may have a fairly small set already. We can easily generalize this to removing subsets of size k and arrive at the problem that our minimzation algorithms are designed to solve:

Definition 3.3 (Minimization Problem). Let C be a set of failure-causing circumstances and let $k > 0$ be an integer. Then the *k-minimization problem on C* is the problem of finding a set T with

$$T \subseteq C; \tag{3.3}$$

$$\text{test}(T) = ✘; \tag{3.4}$$

$$\text{test}(T - T') \neq ✘ \quad \text{for all } T' \subseteq T \text{ with } 1 \leq |T'| \leq k. \tag{3.5}$$

Finding T should also be as fast as possible. We call T a *k-minimal solution to the minimization problem with respect to* test. As usual, if the test function is implicitly understood, we call T a k-minimal solution to the minimization problem, or simply a k-minimal solution. This definition is also due to Zeller and Hildebrandt [75].

3.3 Naïve Algorithm

A 1-minimal solution is a failing process subset with the property that removing any single process from it will cause the replay not to fail any more. This immediately suggests a simple algorithm for finding a 1-minimal solution:[3]

Algorithm N. (*Naïve 1-Minimal Solution.*) Let C be a set of circumstances and let test be a test function so that $\text{test}(C) = ✘$. This algorithm computes a 1-minimal solution to the minimization problem on C with respect to test.

[3]Incidentally, we will always present and analyze algorithms using Knuth's style from *The Art of Computer Programming*. However, we will also always give pseudo-code. As one can see, Knuth's style has six steps whereas the pseudo-code takes twenty lines, even though it takes a few liberties with Algorithm N and is not a verbatim rendition. Also, if there is a discrepancy between the algorithm and the pseudo code, the algorithm should be considered authoritative.

```
set<circumstance> algorithm_n (set<circumstance> s) {
  set<circumstance> t = s;
  bool finished = false;

  while (!finished) {
    set<circumstance>::iterator j = t.begin();

    while (j != t.end()) {
      set<circumstance>::iterator k = j;
      set<circumstance> test_set = t;
      test_set.remove(*j++);

      if (test(test_set) == fail) {
        t.remove(*k);
        break;
      }
    }

    finished = j == t.end();
  }
  return t;
}
```

Figure 3.2: Pseudocode for Algorithm N

N1. [*Initialize.*] Set $T \leftarrow C$.
N2. [*Set up loop.*] Let $T = \{T_1, \ldots, T_k\}$. Set $j \leftarrow k$. We will repeat steps N3 to N4 for $j = k, k-1, \ldots, 1$.
N3. [*Set up test.*] Set $r \leftarrow \text{test}(T - T_j)$.
N4. [*Test.*] If $r = ✘$, set $T \leftarrow T - T_j$ and go to step N2. Otherwise go to step N5.
N5. [*Loop.*] Set $j \leftarrow j - 1$. If $j = 0$, go to step N6, otherwise go to step N3.
N6. [*Finish.*] (At this point, we have $\text{test}(T - T_j) \neq ✘$ for all $1 \leq j \leq k$.) Terminate with T as the answer.

Lemma 3.5. Algorithm N computes a 1-minimal solution.

Proof. First, the algorithm always terminates. This is seen from the fact that T gets smaller and smaller every time step N2 is reached: the first time N2 is reached, we have $|T| = |C|$; at all other times, T has been reduced by one element (by the assignment $T \leftarrow T - T_j$ in step N4). Since C is finite, that means that eventually we will have $j = 0$ in step N5, and will reach step N6.

Once we reach step N6 (which we will do eventually, since Algorithm N terminates), we know that $\text{test}(T - T_j) \neq ✘$ for all $1 \leq j \leq k$, because otherwise, step N4 would have resulted in a return to step N2. This in turn means that T satisfies the definition of a 1-minimal solution for the minimization problem on C. □

How good is this algorithm? Not very. Its worst-case running time is quadratic in the number of elements in C.

Lemma 3.6 (Algorithm N has Quadratic Worst-Case Running Time). Let $|C| = n$. Algorithm N executes step N4 $O(n^2)$ times.

Proof. The outer loop (steps N2–N5) will be executed for $j = n-1, \ldots, 1$ at most, and the inner loop (steps N3–N4) will be executed j times at most. It follows that step N4 is executed at most

$$\sum_{j=1}^{n-1} n - j = \sum_{j=1}^{n-1} j = n(n-1)/2$$

times, which is $O(n^2)$.

This number can actually be achieved if we let $C = \{S_1, \ldots, S_n\}$ and assume that $\mathrm{culp}\, C = \{S_n\}$. If we assume further that $\mathrm{test}(C_j) = ✘$ only for sets C_j of the form $C_j = \{S_j, \ldots, S_n\}$, then Algorithm N will try $C_1, \ldots, C_n$ in order. Removing processes from C_j element by element until we arrive at C_{j+1} takes $n - j$ tests. The number of times we execute step N4 is therefore precisely the maximum number given above. □

Now what about the average running time? This quantity is much harder to analyze. It depends not only on the number of elements in $\mathrm{culp}\, C$, but also on the position of those elements in C, the distribution of $\mathrm{culp}\, C$ in $\mathcal{P}(C)$, and on properties of the test function. We will only show here some preliminary results.

Even if we assume that $\mathrm{culp}\, C$ is equally distributed in $\mathcal{P}(C)$, we cannot answer this question empirically, since we would have to iterate over all 3^{2^n-2} possible test functions—$2^n - 2$ possible nonempty subsets for which to choose one of three possible values for the test function. This is prohibitive even for small values of n.

We will therefore consider consider a special test function with

$$\mathrm{test}(T) = ✘ \quad \text{only if } T \supseteq \mathrm{culp}\, C. \tag{3.6}$$

We call this condition the *superset condition*.

Now we consider a set $C = \{c_0, \ldots, c_{31}\}$ of 32 circumstances (departing slightly from our convention to start counting at 1) and ask: how many times is test evaluated for Algorithm N if the relevant circumstances range over the entire power set of C?

Subsets T of C are being mapped to 32-bit unsigned integers by a function int such that $\mathrm{int}\, T = (b_{31}, \ldots, b_0)_2$ where $b_j = 1$ if and only if $c_j \in T$. The requirement that $\mathrm{test}(T) = ✘$ if and only if $\mathrm{culp}\, C \subseteq T$ easily translates into bitwise operations on $\mathrm{int}(\mathrm{culp}\, C)$ and $\mathrm{int}\, T$: $\mathrm{culp}\, C \subseteq T$ if and only if $\mathrm{int}(\mathrm{culp}\, S) \wedge \mathrm{int}\, T = \mathrm{int}(\mathrm{culp}\, S)$. (For our purposes, it is irrelevant if $\mathrm{test}(T)$ is ✔ or **?** if $\mathrm{culp}\, S \not\subseteq T$.) In this way, each 32-bit unsigned integer yields a test case for each algorithm, for which we compute the number of times test was evaluated. We write this number as test_N for Algorithm N. The results are shown in Figure 3.3.

Since figures like these appear throughout this chapter, we will describe in some detail how they must be read. The figure consists of three subfigures:

- The first plot contains the empirical density function. The x axis contains the number of tests that were needed to minimize a set of failure-inducing circumstances, and the y axis contains the number of times these many tests were observed. It is a histogram, only with lines, not bars. (This is only to facilitate reading the graph.)

- The second plot is a *box-and-whisker plot*. The box starts at the lower quartile (i.e., 1/4 of all observed values were less than or equal to this value) and extends to the upper quartile (which 3/4 of all observed values were less than or equal

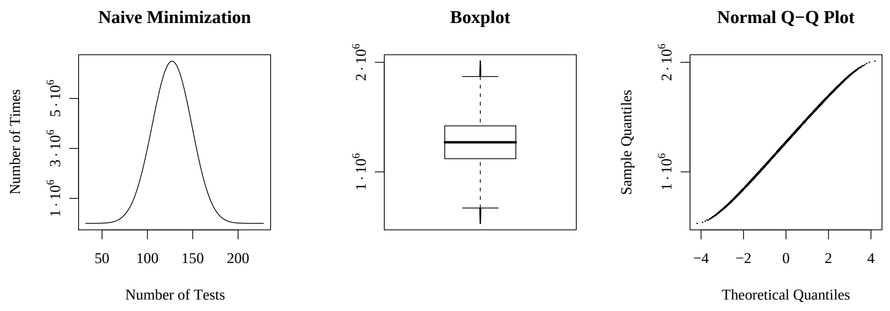


Figure 3.3: Number of tests for Naive Minimization.

to). The line dividing the box is the median (the second quartile). The height of the box is the *inter-quartile range*, or IQR. The *lower whisker* is a datapoint that is at most $1.5 \cdot$ IQR away from the lower quartile, and similarly for the *upper whisker*. Points beyond the whiskers are called *outliers*.

A boxplot gives information about the spread of the data (through the IQR and whiskers), and also shows any skewness (through the position of the median inside the box and the relative lengths of the whiskers).

- The third plot is a Q-Q plot. Such a plot plots the observed quantiles against the expected quantiles of a $N(0, 1)$ distribution. In other words, the k-th smallest value of the n observed values is plotted against the expected k-th smallest value of n normally distributed numbers. The closer this plot is to a straight line (especially in the middle, near the median), the closer the empirical distribution agrees with the theoretical distribution.

  Why are such plots useful? Why can't we simply run a statistical test that would tell us at which significance level the empirical distribution deviates from the theoretical one? The problem is that with so many data points, *any* deviation from the theoretical distribution will be highly significant for any known test. In other words, any test will reject the hypothesis that the data is actually normally distributed. And such a test is undoubtedly correct. However, looking at Figure 3.3, we see that the Q-Q plot rises in the low quantiles, at $x \leq -3.5$, and symmetrically flattens at $x \geq +3.5$. But these are only approximately 0.04% of all data points, or 1.7 million of 4.3 *billion* total. Additionally, these data points lie at the extreme ends of the distribution. The middle portion, where most data points lie, agrees very well with a normal distribution. So for all *practical* intents and purposes, the data is normally distributed, as shown by the Q-Q plot, whereas a statistical test would show significant deviations from normality.

Figure 3.3 shows that Algorithm N needs about 130 evaluations of test to find a 1-minimal solution, with about 2/3 of all numbers falling between 110 and 150. This makes Algorithm N an impractical algorithm, because it will take too long on most practical problems.

Algorithm N is too slow.

How can we improve on its dismal average running time? The key observation is that relevant circumstances are *not* scattered randomly around the set of all circumstances, but instead tend to form *clusters*. This in turn means that we can improve on Algorithm N by changing our strategy to cut off large swathes of circumstances at each step. Fortunately, the theory and practice of this new algorithm, called Delta Debugging, have already been developed, and we will give its properties in the next section.

3.4 Delta Debugging

The basic idea of Delta Debugging is to remove large contiguous blocks of circumstances at each stage. When Delta Debugging starts, it removes one half of the circumstances and runs the test on the remaining half. If the test fails, we have reduced the number of circumstances by one half and try again. If the test succeeds, we try the other half. If we always get one failing test at each stage, we converge on the failure-inducing circumstance with logarithmic speed, but we won't always be so lucky. For example, what happens if both halves contain relevant circumstances? Then testing each half will yield either ✔ or **?**. The question is what to do next?

Delta Debugging solves this problem by using two devices: splitting into more than two pieces and testing the complements of sets. The combination of both devices finds a 1-minimal solution to the minimization problem. The following is adapted from Hildebrandt and Zeller's paper on Delta Debugging [75].

The algorithm is best formulated in a recursive fashion, although we will give an iterative formulation also. First of all, we need some terminology.

Definition 3.4 (Complement). Let S be a set, and let $T \subseteq S$. We write $S - T = \{x : x \in S \wedge x \notin T\}$ and call $S - T$ the *complement of T with respect to S*.

Lemma 3.7. If S is finite, $|S - T| = |S| - |T|$.

Proof. Let $S = \{S_1, \ldots, S_n\}$. If $T = \emptyset$, then $S - T = S$, and the conclusion follows immediately. Otherwise, without loss of generality, we can let $T = \{S_1, \ldots, S_k\}$. (This can always be achieved by rearranging the elements of S.) Then

$$|S - T| = |\{S_{k+1}, \ldots, S_n\}| = n - (k + 1) + 1 = n - k = |S| - |T|. \qquad \square$$

Definition 3.5 (Partitioning into k equal parts). Let S be a finite set. A set $P = \{S_1, \ldots, S_k\}$ is called a *partitioning of S into k equal parts*, if

- P is a partitioning of S, that is

$$\bigcup_{1 \le j \le k} S_j = S \quad \text{and}$$
$$S_i \cap S_j = \emptyset \quad \text{for } 1 \le i < j \le k;$$

- $|S_j|$ is either $\lfloor |S|/k \rfloor$ or $\lfloor |S|/k \rfloor + 1$ for $1 \le j \le k$.[4]

Lemma 3.8. Let $|S| = n$ and let $P = \{S_1, \ldots, S_k\}$ be a partitioning of S into k equal parts. Then the number of elements S_j with $|S_j| = \lfloor |S|/k \rfloor + 1$ is $n \bmod k$.

[4]A weaker condition would be $\big||S_i| - |S_j|\big| \le 1$ for $1 \le i < j \le n$, and the rest would still follow.

Proof. Let $|S| = n$. Since the elements of P are disjoint, we have

$$n = |S| = \left| \bigcup_{1 \le j \le k} S_j \right| = \sum_{1 \le j \le k} |S_j| = \sum_{1 \le j \le k} \big(\lfloor n/k \rfloor + \delta_j\big),$$

where δ_j is either 0 or 1. Expanding further,

$$\sum_{1 \le j \le k} \big(\lfloor n/k \rfloor + \delta_j\big) = \sum_{1 \le j \le k} \lfloor n/k \rfloor + \sum_{1 \le j \le k} \delta_j = k\lfloor n/k \rfloor + \sum_{1 \le j \le k} \delta_j.$$

Since $n = k\lfloor n/k \rfloor + n \bmod k$, we must have $\sum_{1 \le j \le k} \delta_j = n \bmod k$, hence the number of δ_js which are 1 must equal $n \bmod k$. □

Armed with these terms, we can formulate the minimizing Delta Debugging algorithm as follows.

Definition 3.6 (Minimizing Delta Debugging). Let S be the set of proceses encountered during capturing, let $P(S,k)$ be some partitioning of S into k equal parts, and let $\text{test}(S) = ✘$. We define $\text{ddmin}(S) = \text{ddmin}(S, 2)$, where

$$\text{ddmin}(S,k) = \begin{cases} \text{ddmin}(S_j, 2) & \text{if test}(S_j) = ✘ \text{ for some } S_j \in P(S,k); \\ \text{ddmin}\big(S - S_j, \max(k-1, 2)\big) & \text{if test}(S - S_j) = ✘ \text{ for some } S_j \in P(S,k); \\ \text{ddmin}\big(S, \min(|S|, 2k)\big) & \text{if } k < |S|; \\ S & \text{otherwise.} \end{cases}$$

The first line is called "reduction to subset", the second is called "reduction to complement", and the third line is called "increasing the granularity".

Lemma 3.9. The function ddmin computes a 1-minimal solution to the minimization problem.

Proof. See the paper by Hildebrandt and Zeller [75, Proposition 11]. The main idea of the proof is that if ddmin returns S, it will have evaluated $\text{test}(S - \{c\}) \neq ✘$ for every $c \in S$. □

Figure 3.4 has a recursive formulation, but an iterative implementation of the above ideas is also straightforward.

Algorithm D. (*Minimizing Delta Debugging.*) Let S be the set of processes encountered during capturing. Assuming that $\text{test}(S) = ✘$, this algorithm computes a 1-minimal solution to the minimization problem on S using Definition 3.6.

D1. [*Initialize.*] Set $T \leftarrow S$, $k \leftarrow 2$.
D2. [*Set up loop.*] Set $U \leftarrow P(T,k)$. Let $U = \{T_1, \dots, T_k\}$.
D3. [*Set up subset loop.*] Set $j \leftarrow 1$.
D4. [*Set up subset test.*] Set $r \leftarrow \text{test}(T_j)$.
D5. [*Subset test.*] If $r = ✘$, set $k \leftarrow 2$, $T \leftarrow T_j$ and go to step D2, otherwise go to step D6.
D6. [*Loop on j.*] Set $j \leftarrow j + 1$. If $j \le k$, go to step D4, otherwise go to step D7.
D7. [*Set up complement loop.*] Set $j \leftarrow 1$.
D8. [*Set up complement test.*] Set $r \leftarrow \text{test}(T - T_j)$.

```
set<circumstance> algorithm_d(set<circumstance> s, int n_subsets) {
  set<set<circumstance> > subsets = split(s, n_subsets);

  for (int phase = testing_subsets; phase <= testing_complements; phase++) {
    for (set<set<circumstance> >::const_iterator i = subsets.begin();
         i != subsets.end(); i++) {
      const set<circumstance>& t = phase == testing_subsets ? *i : complement(*i, s);

      if (test(t) == fail) {
        if (phase == testing_subsets)
          return algorithm_d(t, 2, level + 1);
        else
          return algorithm_d(t, std::max(n_subsets − 1, 2), level + 1);
      }
    }
  }

  if (n_subsets < s.size())
    return algorithm_d(s, min(failing.size(), 2*n_subsets));
  else
    return s;
  }
}
```

Figure 3.4: Pseudocode for Algorithm D

D9. [*Subset test.*] If $r = ✗$, set $k \leftarrow \max(k-1, 2)$, $T \leftarrow T_j$ and go to step D2, otherwise go to step D10.
D10. [*Loop on j.*] Set $j \leftarrow j+1$. If $j \leq k$, go to step D8, otherwise go to step D11.
D11. [*Increase granularity.*] If $k < |T|$, set $k \leftarrow \min(|T|, 2k)$ and go to step D2, otherwise go to step D12.
D12. [*Terminate.*] Terminate with T as the answer.

Steps D4–D5 implement reduction to subset, steps D8–D9 implement reduction to complement, and Step D11 implements granularity increase.

How good is Delta Debugging? Algorithm D's worst-case running time is the same as for the naïve Algorithm N.

Lemma 3.10. Let $|C| = n$. Algorithm D invokes test at most $O(n^2)$ times.

Proof. See the paper by Hildebrandt and Zeller [75, Proposition 12]. The main idea of the proof is that the worst case happens if first C is subdivided until only single-element subsets remain, and then, when testing complements, only the last complement fails. $\square$

But is Algorithm D better than Algorithm N on the average? First, we conducted the same test for Algorithm D that we did for Algorithm N. This leads to Figure 3.5.

Having now computed these numbers for Algorithms D and N, we ask, when should we say that one algorithm is faster than the other? We answer that question by saying that Algorithm D is faster if $R_D - R_N \ll 0$ on the average, and that Algorithm N is faster if conversely $R_D - R_N \gg 0$ on the average. If that average difference is about zero, we say that both algorithms are about equally fast. All this is under the superset condition of Equation (3.6), so this result should not be applied to more general cases.

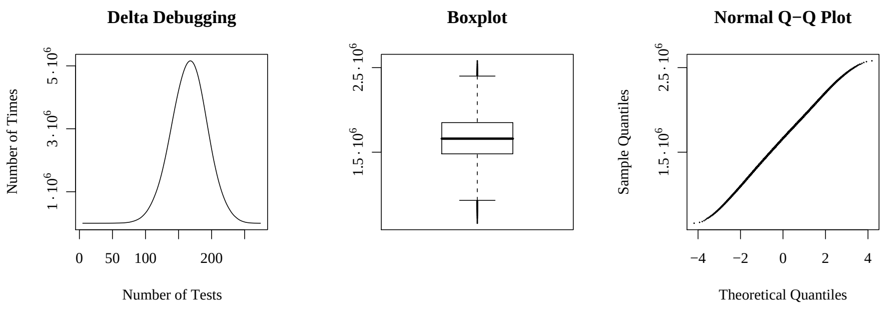


Figure 3.5: Number of tests for Delta Debugging.

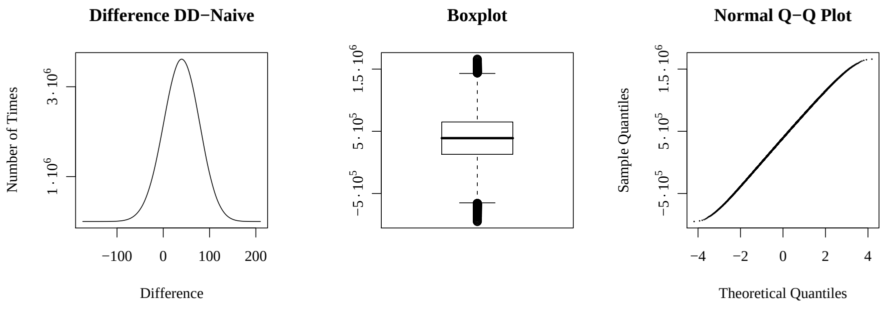


Figure 3.6: Difference between the number of tests for Naïve Minimization and for Delta Debugging for a 32-element set. Since the average is clearly less than zero, Delta Debugging is faster than Naïve Minimization when we consider all possible values of culp C equally likely.

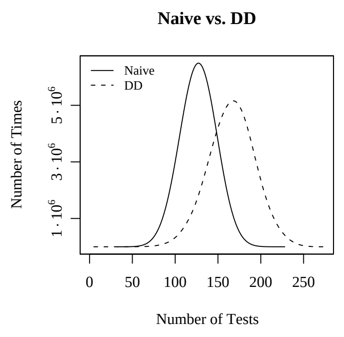


Figure 3.7: Raw values of the number of tests for Algorithms N and D. While the shape for Algorithm N is very regular, Algorithm D's curve shows some kinks. We believe that this is due to Algorithm D's preference for cutting off large chunks of the input if it can.

The results are given in Figure 3.6 which plots $R_N - R_D$ against the number of times that difference occurred, so for example the y value 27108733 for $x = -47$ means that in 27108733 cases, Algorithm D needed 47 less tests than Algorithm N. The average value of $R_D - R_N$ is -10.2427, so Algorithm D is faster than Algorithm D on the average, if we consider all possible values of culp C equally likely.

It is interesting to plot the number of tests (and not the difference) for Algorithms N and D side by side; see Figure 3.7. What can be seen from the figure is that Algorithm N behaves very regularly. The number of tests is very closely normally distributed. Algorithm D, on the other hand, exhibits a much different and more erratic behaviour—the curve is sloped slightly to the left: the distribution has mean 146, standard deviation 16.9 and skewness -0.302. Since we haven't sampled the event space, but exhausted it, we can say with certainty that the distribution is not normal. Also, the algorithm hardly ever needs more than 200 tests.

Algorithm D is much better than Algorithm N.

Chapter 4

Minimization For Intrusion Analysis

The algorithms considered in the previous chapter all have worst-case running times on the order of $O(n^2)$ executions of the test function. This will be too much when one test case entails the (re-)execution of dozens of processes. This chapter explores ways to exploit particular features of intrusion-causing processes in order to achieve large speedups.

The key observation is that intrusion-causing processes will in general not undo their intrusion effects—most intrusions will want to leave some permanent change in the system, such as added accounts, additional processes and so on. Also, harmless processes will generally not undo the actions of attack processes.

Of course, this need not *necessarily* be true in real life: the intrusion-causing processes *could* cause an effect that is then later partially undone by other processes, such that the net effect is no intrusion. However, this would mean that an attacker successfully attacks the system and then removes the effects of the attack, only to attack again later. We consider this highly unlikely.[1]

That means that the cause-effect chain for the intrusion will not contain any masking or undoing processes, which in turn means that the superset condition Equation (3.6) holds.

4.1 Extended Naïve Algorithm

Why does Algorithm N perform so badly when compared to Algorithm D? Intuitively, if all $2^{32} - 1$ nonempty subsets of 32 failure-causing events are equally likely to be culp C, then none of the circumstances that make Algorithm D very fast occur very often. The reason for Algorithm N's slowness is that when it finds an irrelevant circumstance, it spends much time removing circumstances that it already knows are relevant. In algorithmic terms, the inner loop terminates as soon as we have found j with $\text{test}(T - T_j) = ✘$, but we could terminate much faster if we made complete

[1]This does not mean that attackers do not remove traces of their activity, which they do. In the case discussed in the text, it means that there are failing subsets of passing process sets, which is something different. The removal of traces is however not relevant for the attack and will be ruled out by any minimization algorithm.

```
set<circumstance> algorithm_x(set<circumstance> s) {
  set<circumstance> t = s;
  bool found = false;

  do {
    set<circumstance>::iterator j = t.begin();

    while (j != t.end()) {
      set<circumstance> test_set = t;
      set<circumstance>::iterator k = j;
      test_set.remove(*j++);

      if (test(test_set) == fail) {
        t.remove(*k);
        found = true;
      }
    }
  } while (found);
  return t;
}
```

Figure 4.1: Pseudocode for Algorithm X

passes through T, removing irrelevant circumstances and repeating this process only after we have completed an entire pass. The result of these musings is Algorithm X.

Algorithm X. (*Extended Naïve 1-Minimal Solution.*) Let C be a set of failure-causing circumstances. This algorithm computes a 1-minimal solution to the minimization problem on C.

X1. [*Initialize.*] Set $T \leftarrow C$ and $k \leftarrow |C|$.
X2. [*Set up loop.*] Set *found* $\leftarrow$ *false*. Let $T = \{T_1, \ldots, T_k\}$. Set $j \leftarrow k$. We will repeat steps X3 to X4 for $j = k, k-1, \ldots, 0$.
X3. [*Set up test.*] Set $r \leftarrow \text{test}(T - T_j)$.
X4. [*Test.*] If $r = ✘$, set $T \leftarrow T - T_j$ and *found* $\leftarrow$ *true*.
X5. [*Loop.*] Set $j \leftarrow j - 1$. If $j = 0$, go to step X6, otherwise go to step X3.
X6. [*Finished?.*] If *found* $=$ *false*, terminate with T as the answer. Otherwise, go to step X2.

Lemma 4.1. Algorithm X computes a 1-minimal solution.

Proof. We first consider Algorithm X's termination condition. Algorithm X terminates in Step X6 if *found* $=$ *false*, which is true only if the loop from Steps X2 to X5 couldn't remove a single element from T. So *if* Algorithm X terminates, it has computed a 1-minimal solution. If it were never to terminate, *found* would have to be always *true* in Step X6, but if that is so, T has decreased by at least one element. Since C (and hence T) is finite, this can happen only a finite number of times, hence *found* $=$ *false* in Step X6 eventually. $\square$

This algorithm should on the average be much faster than Algorithm D if culp C is evenly distributed in $\mathcal{P}(C)$, but especially if we constrain the test function such that the superset condition Equation (3.6) holds: Algorithm D will try subsets of

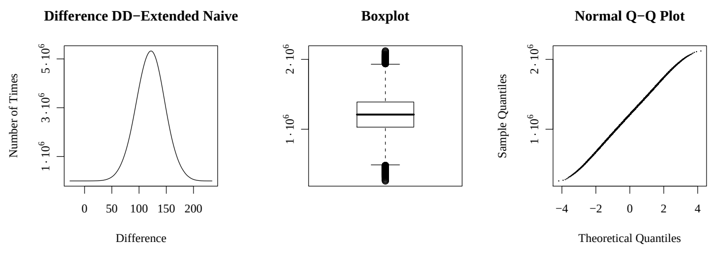


Figure 4.2: Difference between the number of tests for the Extended Naïve Minimization and for Delta Debugging for a 32-element set. Here we can see that Algorithm X is already dramatically faster than Algorithm D.

some set T with $\text{test}(T) = ✔$, even though such T cannot be supersets of $\text{culp}\, C$ under the superset condition. We note that the superset condition holds when minimizing process sets during a consistent replay, according to Definition 5.4. In fact, this condition makes Algorithm X's worst case running time *linear* in $|C|$:

Lemma 4.2. Under the superset condition, the worst case for Algorithm X has running time $2|C| - 1$.

Proof. Under the superset condition, Algorithm X always needs less than two complete passes through C, because any C_j that has been tentatively removed and found relevant will also be in $\text{culp}\, C$. Assume this were not so. Then we would have some T with $\text{culp}\, C \subseteq T = \{T_1, \ldots, T_j\}$ where (for example) $\text{test}(T - T_1) \neq ✘$ but $T_1 \in \text{culp}\, C$, which is impossible under the superset condition. That means that the maximum number of complete passes over C is less than 2, and that means that the maximum number of tests is at most $2|C| - 1$. □

Another important fact is that under the superset condition, Algorithm X finds not only a 1-minimal solution, but actually $\text{culp}\, C$:

Lemma 4.3. If the superset condition holds, Algorithm X computes $\text{culp}\, C$.

Proof. let T be the result of Algorithm X and assume on the contrary that $T \neq \text{culp}\, C$. Now since $\text{test}(T) = ✘$, we must have $\text{culp}\, C \subseteq T$ (by our convention that $\text{culp}\, C$ is just a shorthand notation for "any convenient member of the set of all minimal failure-causing circumstances"). Since we have also stipulated that $T \neq \text{culp}\, C$, we must have $T - \text{culp}\, C \neq \emptyset$. Let $T_1 \in T - \text{culp}\, C$. By the superset condition, we would have $\text{test}(T - T_1) = ✘$, and that would have been tested in Step X3. That would have caused *found* to be set to *true*, and T to be replaced by $T - T_1$, so T could not have been the result of Algorithm X, a contradiction. □

If we repeat our experiment from above, we find our theoretical computations confirmed; see also Figure 4.2:

Algorithm X is much faster than Algorithm D.

```
set<circumstance> algorithm_x(set<circumstance> s) {
  set<circumstance> t = s;
  bool found = false;

  set<circumstance>::iterator j = t.begin();
  while (j != t.end()) {
    set<circumstance> test_set = t;
    set<circumstance>::iterator k = j;
    test_set.remove(*j++);

    if (test(test_set) == fail)
      t.remove(*k);
  }

  return t;
}
```

Figure 4.3: Pseudocode for Algorithm E

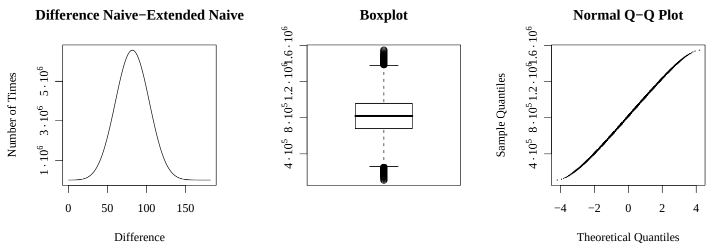


Figure 4.4: Difference between the number of tests for Extreme Naïve Minimization and for Extended Naïve Minimization for a 32-element set. Here we can see that Algorithm E is again faster than Algorithm X.

4.2 Extreme Minimization

Yet another important observation is that we can accelerate Algorithm X by another factor of 2 by leaving out the last unneccessary pass. Consider this modification of Algorithm X to only contain one pass:

Algorithm E. (*Extreme Naïve Minimal Solution.*) Let S be the set of processes encountered during capturing. Assuming that test(S) = ✘, and assuming the superset condition, this algorithm computes culp C.

E1. [*Initialize.*] Set $T \leftarrow S$.
E2. [*Set up loop.*] Let $T = \{T_1, \ldots, T_k\}$. Set $j \leftarrow k$. We will repeat steps E3 to E4 for $j = k, k-1, \ldots, 0$.
E3. [*Set up test.*] Set $r \leftarrow$ test$(T - T_j)$.
E4. [*Test.*] If r = ✘, set $T \leftarrow T - T_j$.
E5. [*Loop.*] Set $j \leftarrow j-1$. If $j > 0$, go to step E3, otherwise terminate the algorithm with T as the answer.

If we repeat our experiment from above, we find now that the average difference between the running times of Algorithms X and E is about 80. Therefore, Algorithm E is about twice as fast as Algorithm X and much faster than Algorithm D on the average; see Figure 4.4.

Lemma 4.4. Algorithm E computes culp C.

Proof. First of all, test$(T) = ✘$ when Algorithm E ends, so culp $C \subseteq T$. If there were $T_j \in T -$ culp C, we would have test$(T' - T_j) = ✘$ for some intermediate value of T' that was computed in Step E4. That would have led to the exclusion of T_j from T, a contradiction. □

Lemma 4.5. Algorithm E's running time is $|C|$.

Proof. Every element of C is removed once and the result tested. □

It seems plausible to assume that every algorithm can compute culp C under the superset condition will never have a worst-case running time of less than $O(|C|)$ tests to do so—even if we do not have a proof. If that is true, we conclude that Algorithm E must be optimal.

Theorem 4.6 (Algorithm E is optimal). Algorithm E has a best-possible worst-case running time among all algorithms that compute culp C under the superset condition.

4.3 Binary Minimization

Why must we remove every element of S one at a time? Can't we apply the key idea of delta debugging and remove big portions of S at a time, only this time use the superset condition effectively? We can. The key idea ist that we can find the rightmost element of culp S via binary search. If we call this element c, we can then find the rightmost element of culp $C - \{c\}$ by binary search as well, and so on, until we end up with culp C.

Algorithm B. (*Binary Minimization.*) Let $C = \{c_1, \dots, c_n\}$ be a set of failure-causing circumstances. Assuming that test$(C) = ✘$, and assuming the superset condition, this algorithm computes culp C.

B1. [*Initialize.*] Set $T \leftarrow \emptyset$. Set $r \leftarrow n$.
B2. [*Prepare binary search.*] Set $l' \leftarrow 1$ and $r' \leftarrow r$. We will look for the culprit member with the highest index between indices l' and r'.
B3. [*Compute midpoint.*] Let $m = \lfloor (l' + r')/2 \rfloor$.
B4. [*Set up test.*] Set $t \leftarrow$ test$(\{c_1, \dots, c_{m-1}\} \cup T)$.
B5. [*Test.*] If $t = ✘$, set $r' \leftarrow m - 1$, otherwise set $l' \leftarrow m$.
B6. [*Loop if not found.*] If $l' < r'$, go to step B3. Otherwise, go on to B7.
B7. [*Loop if found.*] If $t = ✔$, set $T = T \cup \{c_m\}$, and set $r \leftarrow m - 1$. Otherwise return to step B2. If now $r \geq 1$, return to B2. Otherwise, terminate the algorithm with T as the answer.

Theorem 4.7. Algorithm B computes culp C.

```
#define N 32

bitset<N> algorithm_b() {
  bitset<N> s = ~0;              // Initially include all circumstances
  bool found = false;
  int l = 0;

  do {
    int min = l;

    while (l < N) {
      l = (min + N)/2;

      bitset<N> t = s;
      for (int i = min; i <= l; i++) t[i] = false;

      if (test(t) == fail) {
        found = true;
        min = l + 1;
        break;
      }
    }
  } while (found);

  return t;
}
```

Figure 4.5: Pseudocode for Algorithm B

Proof. To begin with, the algorithm certainly terminates because first, the inner loop (steps B3–B6) contain a binary search that always terminates, and second, r always decreases between successive iterations in the outer loop. For the following discussion, let $U = \{c_1, \ldots, c_k\} \cup T$. A circumstance c_k is added to T if k is the largest index that has test$(U) = ✘$, so clearly $c_k \in \operatorname{culp} C$. All other members c_m with $m > k$ must already be in T, because otherwise U will not be a superset of $\operatorname{culp} C$ and hence test$(U) \neq ✘$. Hence, when Algorithm B terminates, $T = \operatorname{culp} C$. □

Theorem 4.8. Algorithm B executes $O(|\operatorname{culp} C| \log |C|)$ tests.

Proof. To find every single element of $\operatorname{culp} C$ takes $O(\log |C|)$ time, so the entire process takes $O(|\operatorname{culp} C| \log |C|)$ time. □

If $|\operatorname{culp} C| < |C|/\log |C|$, this will be less than the $O(|C|)$ tests needed for Algorithm E. Nevertheless, if $|\operatorname{culp} C| \approx |S|/\log |S|$, Algorithm B's running time will exceed that of Algorithm E. Therefore, while Algorithm E's worst-case running time might be optimal because of Theorem 4.6, we usually have the needle-in-a-haystack case ($|\operatorname{culp} C| \ll |C|/\log |C|$), which will make Algorithm B perform better.

Algorithm B is very fast under typical conditions.

Chapter 5

Malfor

Now that we have the algorithms to minimize a set of failure-causing circumstances, let us apply this knowledge to intrusion analysis. The gist of the rest of this chapter is that we can analyze intrusions by first capturing processes and replaying them under the control of a minimization algorithm. Each replay can be viewed as the result of the computation of a test function (see Definition 3.1) taking a set of process numbers as input and returning ✔, ✘, or **?** as output, depending on whether this process set makes the intrusion happen, makes it not happen, or is impossible to replay, respectively.

5.1 Processes and Replay

For the purposes of this section, we abstract away all the things that processes actually *do*, and keep just their genealogy, i.e., which process was created by which other process. In order to do this, we identify a process with its process ID.[1]

5.1.1 Preliminary Definitions

First, we need the notion of an *iterated function*. This familiar notion will be needed below when we define the ancestors of a process.

Definition 5.1 (Function Iteration). Let $f\colon M \to M$ be a function that maps a set to itself. For a nonnegative integer k, we define

$$f^k(x) = \begin{cases} x & \text{if } k = 0, \\ f\bigl(f^{k-1}(x)\bigr) & \text{otherwise.} \end{cases} \tag{5.1}$$

Definition 5.2 (Process ID, Parent, Ancestor, Offspring). Let P be a set of positive integers (called *process IDs*). Let parent: $P \to P$ be a mapping between process

[1]This seemingly natural step actually has a problem because process IDs wrap around in real life. In other words, a process ID may refer to different processes at different times. We sidestep this problem simply by assuming that process IDs do not wrap around. In practice, we could attain this by prefixing any real process ID by the number of times it has been issued since the system was booted, turning a process ID into a pair (*times*, *id*). Since there is an implementation-defined upper limit L for process IDs such that $p < L$ for all process IDs p, we can then re-map this pair of integers back to a single integer by computing $\mathit{times} \cdot L + \mathit{id}$. Since this would just clutter up the notation, we simply assume that process IDs are unique.

IDs. For $p \in P$, we call the set $\bigcup_{k \geq 1}\{\text{parent}^k(p)\}$ the *ancestors of p* and write it ancestors(p). Conversely, the *offspring* of p are those processes for whom p is an ancestor: offspring$(p) = \{q : p \in \text{ancestors}(q)\}$. The function parent is called a *parent function* if the following are both true:

$$\begin{aligned} &\text{there is exactly one } p_{\text{root}} \in P \text{ for which parent}(p_{\text{root}}) = p_{\text{root}} \\ &\text{for every } p \in P \text{ with } p \neq p_{\text{root}},\ p \notin \text{ancestors}(p). \end{aligned} \tag{5.2}$$

For example, the Unix process called *init* traditionally has the process ID 1, and parent(1) = 1. The one-to-one correspondence between a process and its ID allows us to talk about the "process p" when in fact we should be talking about the "process whose ID is p" or even the "process whose ID is currently p". This is only possible because of our assumption earlier that process IDs do not wrap around. If the name of the program that a process is executing is unique, we will also sometimes identify a process with that program. For example, we could be talking about the *init* process, or about parent(*httpd*).

Our definition makes sense in that parent functions induce process trees:

Lemma 5.1 (Parent functions induce trees). Let P be a set of process IDs and let parent be a parent function. The graph (V, E) given by

$$\begin{aligned} V &= \text{parent}(P) \\ E &= \{(p, p') : \text{parent}(p) = p'\} - \{(p_{\text{root}}, p_{\text{root}})\} \end{aligned}$$

is a rooted tree. (Note that edges point towards parents, not towards offspring.)

Proof. First of all, every node $p \neq p_{\text{root}}$ has outdegree 1. This is because parent(p) is unique, hence for every such p, there is exactly one edge $(p, \text{parent}(p)) \in E$.

Next let $\langle (p_0, p_1), (p_1, p_2), \ldots \rangle$ be a path of maximum length with $p_0 = p$ and $(p_k, p_{k+1}) \in E$. This path must be finite because since V is finite, this would otherwise imply the existence of a cycle $\langle (p_k, p_{k+1}), \ldots (p_k, p_{k+1}) \rangle$ in the path. That would mean, however, that p_k is its own ancestor, which is expressly forbidden by Condition (5.2). Now consider the last edge (p, p') in the path. The reason why it is the last edge is that there is no edge beginning with p' in E. However, since parent(p) exists for every $p \in P$, that can only mean that $p' = p_{\text{root}}$, because that is the only pair $\big(p, \text{parent}(p)\big)$ that was expressly excluded from E.

Taken together, both properties mean that there exists a root node, and that there exists exactly one finite path from every non-root node to the root node. □

In order to replay processes, we have to capture them first. This works by specifying an initial set C_0 of processes to be captured. Whenever such a process creates a new process via *fork*, the new process is also marked for capturing.

Definition 5.3 (Captured Processes). Let P be a set of process IDs. Let $C_0 \subset P$ be a set such that for every $c \in C_0$, ancestors$(c) \cap C_0 = \emptyset$. We call C_0 a *(valid) set of processes initially set up to be captured* (sometimes simply called the *initial set*) and define

$$C = \bigcup_{c \in C_0} \text{offspring}\, c$$

as the *set of processes encountered during capturing*.

This causes all offspring of processes in C_0 to be captured as well, which in turn leads to closedness of capturing under the parent-child relationship: the set of processes encountered during capturing forms a set of rooted trees.

The roots of those trees are all in C_0. The only way that a process in C_0 is not also the root of a captured process tree is if one of its ancestors is also in C_0. This is impossible in practice because a process needs to be set up for capturing at its creation. Since ancestors are necessarily created before their offspring, the offspring cannot be in C_0.

5.1.2 Replay

We are interested in finding the cause of an intrusion as a cause-effect chain in the sense of Chapter 2: the processes in such chains cause the intrusion, the chains are minimal, and chains are not unique but all have the same size. In terms of process trees, these processes are special because they are relevant for the intrusion, which in our case simply means that they cause some change in the state of the system that makes some test function return ✘. Therefore, causes of intrusions are simply culprits in the sense of Definition 3.2 for some convenient test function.

However, this general definition is not enough to allow the analysis of intrusions in practice. Consider the case where a process gains root privileges through a hole in *suidperl*. This is certainly the immediate cause of the breakin, but it is not sufficient to clear it up because it fails to uncover the reason why the attacker was able to invoke *suidperl* in the first place. Therefore we will have to follow the parents of the immediate causes until we arrive at a process in C_0. This leads us to the definition of a *consistent* replay:

Definition 5.4 (Consistent Replay). Let P be a set of process IDs, C_0 a set of processes initially set up to be captured, let C be the set of process IDs encountered during capturing, and let test be a test function on C. Any function $R : \mathcal{P}(C) \longrightarrow \{✔, ✘, ?\}$ with

$$R(S) = \begin{cases} ? & \text{if there is some } p \in S \text{ with } p \notin C \text{ and } \text{parent}(p) \notin S, \\ ✘ & \text{if } S \supset \text{culp}_{\text{test}}\, C, \\ ✔ & \text{otherwise.} \end{cases} \qquad (5.3)$$

is called a *consistent replay of P*.

The first condition stipulates that only complete process subtrees are contained in S. In other words, if a process p is in S, then so must be its ancestors, until we reach a process in C. The closedness property guarantees that the first and second conditions will never be true simultaneously. The second condition is the superset condition of Equation (3.6); the reason why we believe that it holds for intrusion analysis was motivated at the beginning of Chapter 4. This condition plays a large part in later sections of this chapter, so it should be kept in mind.

Lemma 5.2 (Replays are Test Functions). Let R be a consistent replay of P with respect to C_0 and test. Then R is a test function in the sense of Definition 3.1.

Proof. The empty set qualifes neither for condition 1 nor for condition 2 of Equation (5.3), hence $R(\emptyset) = ✔$. □

At this point, we can apply the apparatus of Chapter 4 to find culprit processes, or rather process subtrees and hence cause-effect chains, by minimizing C with respect to R and test

Intrusion analysis can be viewed as another application of minimization.

5.2 System Call Interposition

We now know how to minimize a set of intrusion-causing processes—by making a series of controlled experiments. We still need to describe how to make these experiments: How does process capturing and replay work in practice?

The answer lies in a technique called *system call interposition*. This technique uses the fact that without system calls, a process cannot have side effects (except those side effects that happen by its sheer existence). System call interposition captures and replays system calls. This allows us to replay and control entire processes without having to capture and replay every machine instruction.

There are two more challenges when making experiments on processes. First, we sometimes need to suppress a process—when that process is not being replayed by the minimization algorithm. And second, we will need to be able to decide whether a system call should be replayed from the database or whether it should be allowed to execute for its side effects. We will want to replay socket I/O because that typically happens outside the control of our machine. We will want to execute I/O when we want to see its side effects, for example the addition of a new root account.

5.2.1 System Calls

Only by making system calls do processes make the results of their computations available to the outside world: if a process computes the answer to a problem but does not make a system call, noone will ever learn what that answer is.[2] This makes system calls very attractive for intrusion detection or policy enforcement systems because all interaction with the outside world has to be done through system calls, and if we could identify the harmful system calls, we can block or report them.

A system call is nothing but a function inside the operating system kernel. The operating system has special powers. In a monolithic design like Windows, Linux or Unix, code that executes inside the operating system can modify each and every byte in the system. Therefore, operating system code needs special protection and thus is not directly accessible to processes. It has a calling convention that is different from ordinary functions.

As an example, let us look at the *read* system call. This call is used to read a number of bytes from a file descriptor into a buffer that is controlled by the process. For convenience, the C library on a Unix system contains a so-called *C binding* of the *read* system call, which allows user processes to call *read* as if it were a user-level function. Its syntax is: int *read*(*fd*, *buffer*, *nbytes*). Here *fd* is Unix-speak for a file descriptor, a non-negative integer used to denote a kernel-level data structure; *buffer* is the user-level buffer where the bytes should be put and *nbytes* says how many bytes

[2]The exception to this general rule is when a process opens a subliminal channel to other processes, but we won't consider these. These subliminal channels can work without system calls. For example, a process could compute very hard for some time if it wants to convey a 1-bit, or terminate immediately if it wants to convey a 0-bit. Granted, termination needs a system call, but every process needs to terminate eventually.

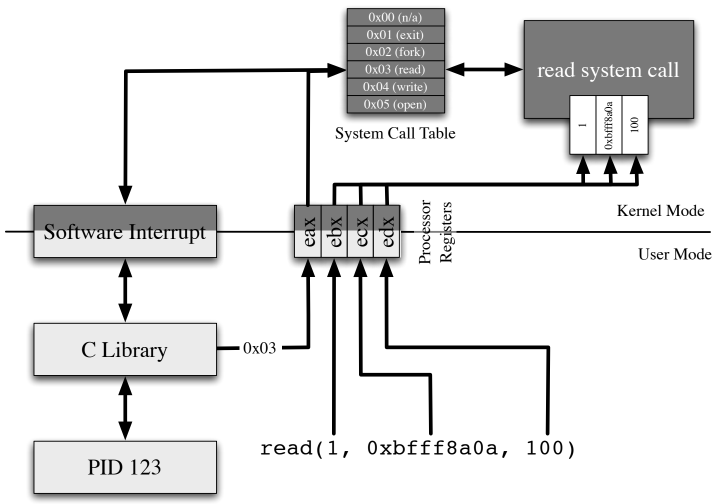


Figure 5.1: Operation of a system call on x86 Linux. First, the C Library puts the system call's parameters into processor registers. Then it executes a special processor instruction called a trap or software interrupt, causing a transfer to the operating system kernel. The kernel uses the `eax` register where it finds an index into a jump table. It calls the routine found at the appropriate index, which then does its work and returns. The kernel puts the system call's result into `eax`, restores the process context and resumes the process. The C library examines the return code, sets *errno* and returns to the caller.

we want to read from the file descriptor. The return value is an int and we look at the manual page of *read* to see that on success, it returns the number of bytes read, and -1 otherwise. In the latter case, the global lvalue *errno* contains an appropriate error code. Here is a slightly simplified explanation of what happens when process 123 calls int *read*(1, `0xbfff8a0a`, 100) on an x86 Linux system (see also Figure 5.1):

1. The process calls the C binding for *read* in the C runtime library.
2. The C binding loads the parameters for *read* into registers. In this case, `ebx` contains the file descriptor, `ecx` contains the user-space address of the buffer, and `edx` contains the number of bytes to be read. Register `eax` contains the system call number for *read*, which is 3.
3. The C binding then causes a transfer to the operating system by executing a trap, also known as a software interrupt. This causes the processor to change from user mode to supervisor mode, also known as kernel mode. It differs from user mode in that certain machine instructions and registers are only available in supervisor mode. For example, the memory management unit can only be configured from supervisor mode and control and data registers of memory-mapped devices are only visible from that mode.
4. Each trap has an associated trap number which is an index into a processor-wide jump table of addresses. In our case, the trap number is `0x80`. The address at index `0x80` in the jump table points to a piece of operating system code that puts the system call parameters from the registers onto the stack.

5. It then looks up the system call number in another jump table, called the system call table or syscall table and transfers control to the appropriate routine (usually written in C). This routine then handles the *read* system call and is therefore called the system call handler.

6. The system call handler then does its work (probably transferring control to other routines and probably even other processes) and eventually leaves kernel mode to return to the C binding in user mode.

7. The C binding checks the system call's return code (in register `eax`) and sets its own return code and *errno* accordingly.

5.2.2 Interposition

System call interposition is the technique by which we capture and replay system calls. It works by changing the system call table that is consulted in Step 5 above. Instead of transferring control directly to the system call handler, we transfer control to our own routine. For capturing, we note down the parameters, call the original system call and note the result. For replaying, we note the parameters and consult a database to find the appropriate return value; Figure 5.2 shows the architecture.

System call interposition for capturing is easy to do. There are also user-level tools like *strace* that capture system calls without using system call interposition. Replaying processes, on the other hand, is very difficult and many careful decisions need to be made by anyone who replays processes on the basis of their system calls. Section 5.7 contains details.

System call interposition has been used in many security-related areas, mostly for process confinement, intrusion detection and policy enforcement [137]. The paper by Garfinkel has a brief survey [68]. We just mention the work by Ko and others, who wrap system calls in order to detect intrusions [91]; Wagner and others, who use static analysis to derive a model of system call behaviour and then monitor system calls in order to detect violations of that behaviour [172], Acharya and others, who use interposition to confine untrusted applications [1]; and Dan and others, who sandbox the entire operating system [47] .

5.2.3 Solipsy Architecture

Solipsy is the name of the capture/replay subsystem used in Malfor. It consists of several components (see Figure 5.2):

- A system call interceptor. This component takes all interesting system calls and moves them from kernel land to user land through a character device called */dev/iocap*.

- A database. This database contains all system calls that were intercepted during capturing, plus additional information (see below).

- A capture daemon. This daemon reads system calls from */dev/iocap* and puts then into the database.

- A replay daemon. This daemon reads system calls from */dev/iocap*, and replays the corresponding calls from the database.

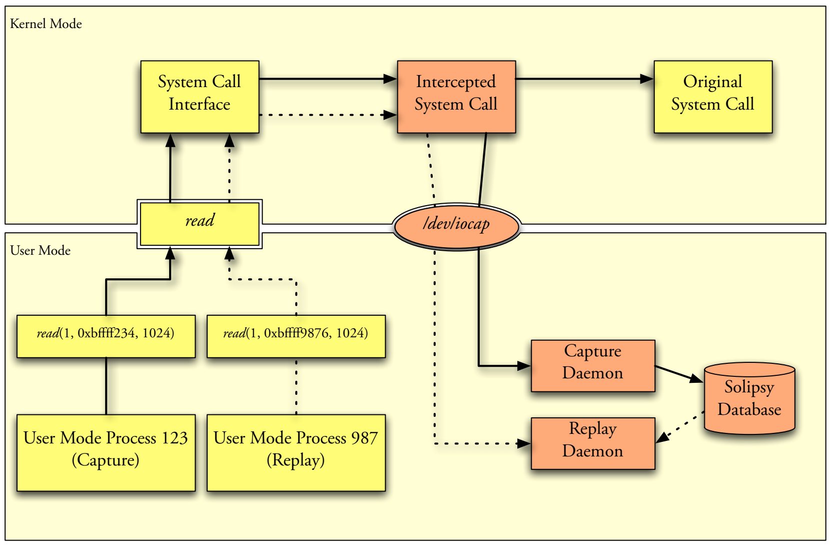


Figure 5.2: How Solipsy works. Components that are part of Solipsy are shown darker. During capturing, a *read* system call is intercepted and written to the device */dev/iocap*, where it is read by the capture daemon and written into the solipsy database. On replay, the system call is suspended while the replay daemon fetches the system call's result from the same database.

- A policy engine. This program gets system calls from the replay daemon and decides how to replay them. This is needed for process suppression and other manipulations.

On replay, the system call interceptor plays with the process's notion of reality. The process may believe that it reads bytes from a file, when in fact it is fed bytes from the database. For this reason, this subsystem is named *Solipsy* (from solipsism, the notion that the outside world comes to us through our senses and is therefore not necessarily real).

5.2.4 State Mappings

When we replay a process, we need to make its internal state so close to its internal state during capturing that the process's computation during replay is the same as its computation during capturing. The idea is that if we replay as closely as possible the system calls that a process makes, it will make an identical computation.

Now, we may want the process to think that it is executing the original computation, but in fact, time has moved on and it will in fact be impossible to recreate all conditions exactly as they were during the original computation.[3] That means in practice that we will have to map between the operating system (reflecting the current state of the world) and the application (reflecting a past state of the world). These mappings are described in more detail in Section 5.7.

[3]This isn't exactly news: Heraclitus notes in the sixth century B.C. that "[I]t is not possible to step twice into the same river, nor is it possible to touch a mortal substance twice in so far as its state is concerned." [49, Fragment B91], English translation by John Burnet [27].

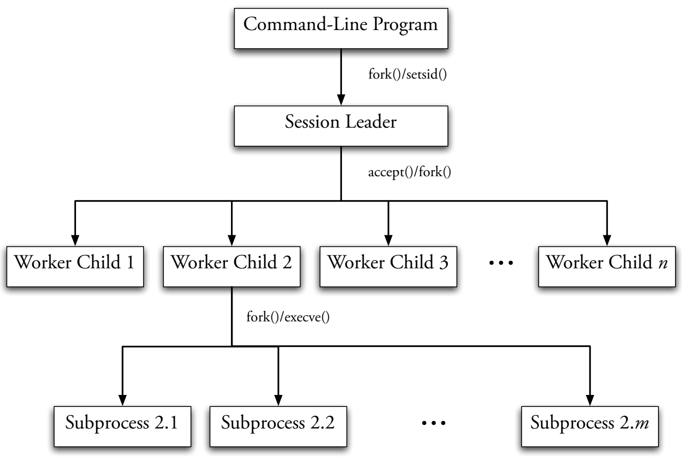


Figure 5.3: Typical process structure of a network server. The server is first invoked on the command line. It then detaches itself from its controlling terminal, becoming a session leader. Next, it opens a socket and waits for incoming connections. On receiving such connections, it forks a worker child process, which may in turn create subprocesses.

5.3 Evaluation

The evaluation of Malfor was done using two test cases. The first test case (Section 5.3.1) is very simple and is used to show the soundness of the principles on which Malfor is built. The second test case (Section 5.3.2) is very complicated and shows that Malfor can automatically analyze attacks that can be analyzed by no other tool today. If Malfor is to be a practical system, it needs to be accurate, fast, and easily deployed. Malfor's accuracy is currently being evaluated; all we can say at this point is that it has so far found the relevant processes in all our tests. Sections 5.3.3, 5.3.4 and 5.3.5 contain a preliminary performance evaluation, Section 5.3.6 looks at Malfor's deployability, and Section 5.3.7 has some ideas for fooling Malfor.

5.3.1 Proof of Concept

For the proof-of-concept evaluation, we created a simple network server. This server, named *spud*[4], reads and parses a HTTP-like command set from a network socket. One of these commands will cause the file */tmp/pwned* to be created. In our evaluation, we use the existence of this file as a break-in indicator: as soon as this file has been successfully created, we say that *a break-in has happened.* Spud has the following structure which is typical of many network server programs (see also Figure 5.3):

1. it detaches itself from the controlling terminal, becoming a session leader;
2. the session leader opens a socket and binds it to a well-known port number;
3. it accepts a connection on that socket;

[4]After a character from *Trainspotting* [181].

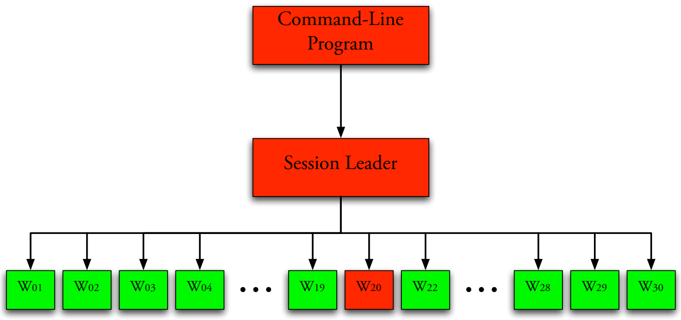


Figure 5.4: The culprits in the Spud example. Culprit processes are shown darker than their harmless counterparts (or in red and green if viewed from the electronic version of this document). Worker 20 causes the break-in by creating */tmp/pwned*. But in order for worker 20 to do its work, processes C (the command-line process) and S (the session leader) must also exist.

4. it forks a worker process; and

5. while the session leader continues listening, the worker reads a request from the newly opened socket, performs the requested action (possibly using subprocesses that run other programs), and exits.[5]

In this example, the intrusion is caused by a single system call, the one that creates the file */tmp/pwned*. The *set of relevant processes* then contains the process making that system call, and its ancestors.

Assume that we have 32 processes: the command-line program C, the session leader S and thirty workers $W_1, \ldots, W_{30}$, where W_{20} executes the intrusion-causing system call. Delta debugging will try different subsets of the set of all processes $\{C, S, W_1, \ldots, W_{30}\}$ and test whether the intrusion still happens. In our example, the set of relevant processes would be $\{C, S, W_{20}\}$; see also Figure 5.4

The actual sequence of process subsets tried by delta debugging is shown in Table 5.1. The column marked "R" contains the result of the test: ✘ if the intrusion occurs, and ✔ if it does not occur. The other columns contain a bullet if the corresponding process is included in the test. For example, in row 1, processes C, S, and W_1 through W_{14} are included, but W_{15} through W_{30} are not. Since W_{20} is the culprit, and since it is not executed, the intrusion does not happen and the output is ✔.

Lines 1–3 try to find a subset of the original processes that produce ✘, first by splitting the original set in roughly equal parts and then subdividing it further and trying complements when that does not work. Line 14 contains the minimal subset needed for the intrusion. Further subsets need not be considered, because they have already been tested (lines 9 and 13). Delta debugging finds the culprits (processes C, S, and W_{20}) with only fourteen tries.

The processes that Malfor found, and only *those,*
really were *relevant for the break-in.*

[5]Spud does not use any subprocesses.

	Processes																																
#	*C*	*S*	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	**R**
1	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●																	✔
2	●	●	●	●	●	●	●	●																									✔
3	●	●	●	●	●	●	●	●									●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	✘
4	●	●	●	●	●	●	●	●																	●	●	●	●	●	●	●	●	✔
5	●	●	●	●	●	●	●	●									●	●	●	●	●	●	●	●									✘
6	●	●	●	●																													✔
7	●	●	●	●													●	●	●	●	●	●	●	●									✘
8	●	●	●	●																	●	●	●	●									✘
9	●	●																															✔
10	●	●																			●	●	●	●									✘
11	●	●																					●	●									✔
12	●	●																			●	●											✘
13	●																																✔
14	●	●																				●											✘

Table 5.1: Process subsets actually tried by delta debugging Spud with Malfor. The column marked "#" gives the test number; the columns marked "*C*", "*S*", and "01"–"30" show which processes are included in a test case. Here, *C* is the command-line program, *S* is the session leader, and 01–30 are the 32 worker processes. For example, test 12 contains *C*, *S*, and workers 19 and 20. The column marked "R" gives the test result: ✔ for a passing test, and ✘ for a failing test. The culprits in this case are *C*, *S*, and worker 20, and the vertical lines give a visual clue how delta debugging zeros in on them.

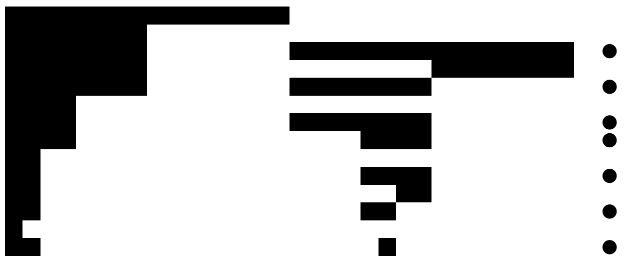


Figure 5.5: Graphical representation of Table 5.1. On the x-axis the processes are arranged in order of increasing process IDs. A process is shown in black if it is included in the set that delta debugging tests, and it is shown in white if it is excluded. One unit on the y axis represents one execution of a process subset. Time passes from the top of the page to the bottom as Malfor tries 14 tests. A test that results in ✘ is shown with a black dot at the side; all other tests shown are ✔. We can see how delta debugging systematically eliminates processes: white stripes appear running down the page that represent processes that have been permanently excluded from consideration by a failing test. Delta debugging tries many more test cases than are shown here. However, these test cases result in impossible process sets—sets where processes are included whose parent process is not included. Malfor automatically skips these tests.

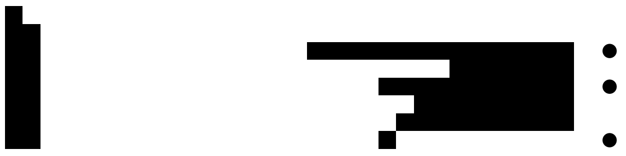


Figure 5.6: Using Algorithm B instead of Algorithm D on Spud. The figure is analogous to Figure 5.5.

```
root:H5WJ3R0Hi.aNQ:0:0:root:/root:/bin/bash
nobody:*:65534:65534:nobody:/home:/bin/sh
sshd:!:100:65534::/var/run/sshd:/bin/false
user:unknown:456:100:Some User:/usr/someuser:/bin/bash
toor:31gJkafn50ltq:0:0:root:/:/bin/bash
```

Figure 5.7: The password file of a compromised Linux system. The account named `toor` (which is `root` spelled backwards) has user ID 0 and group ID 0, which under Unix means that it is a system administrator's account. Having an account named `toor` on a system almost certainly means that it is compromised. The encrypted passwords shown here are not real.

We emphasize that Malfor's result does not only contain the root cause of the attack, but all intermediate attack-relevant processes too. So if an attack involves a long chain of events, Malfor will produce all the intermediate steps that are needed to reproduce the attack.

Malfor produces the complete cause-effect chain.

One concern is that a process could not exhibit its original behavior during replay because it took a different control path. For example, what if a process launches an attack only upon the existence of certain files, or a successful challenge-response authentication with a remote server? In these cases, the process must have made system calls that caused these actions to be performed. Malfor then captures these system calls and replays them. For example, if a process creates a random challenge as part of the challenge-response protocol, it will have to issue system calls to do so (for example, in order to read */dev/random*). When we replay the process, we also replay these system calls, so we will have recreated the state of the process as it was when it made the original challenge-response authentication and the computed challenge will be the same in both cases. In the case of files on the local file system, Malfor actually executes the system calls; in the case of a remote challenge-response authentication, it replays a previously recorded conversation.

5.3.2 A Real-World Attack

The scenario for this analysis is that a system administrator stumbles upon the password file on one of his servers and encounters the file in Figure 5.7. From the bogus system administrator account `toor`, he knows that his system must have been compromised. We will see how Malfor finds the cause of this additional root account even under circumstances that make it difficult or impossible for other tools to reason about the attack. [125]

Most attacks, including ours, work according to a fixed scheme:

1. Gain access to the machine by exploiting a flaw in a network program. This results in the attacker being able to execute any command on the machine, but with restricted rights—usually those of the owner of the exploited network program.

2. Using the ability to launch commands, download additional malcode.

3. Using this downloaded malcode, exploit a flaw in a local program or process to gain system administrator privileges. This enables the attacker to modify any file on the system, to launch or kill any process, and to install software—even if that is usually not possible for ordinary users.

4. Armed with system administrator privileges, install a backdoor. The purpose of this backdoor is to allow the attacker to regain root privileges even when the flaws that were exploited in steps 1 and 3 are eventually fixed.

5. Remove as many traces of the attack as possible and as permanently as possible.

Our target machine is running a subset of Debian 3.0 on a Linux 2.4.24 kernel. We have installed Apache httpd 2.0.54 together with version 1.2.2 of *mod_auth_any*, a module that uses a configuration file to launch authentication programs. We have also installed Perl 5.8.4. The attacker knows that:

- The 1.2.2 version of *mod_auth_any* contained a shell code insertion flaw by which it is possible to execute arbitrary shell commands as the user that Apache runs as (usually *www-data*);

- Perl 5.8.4 had a buffer overflow bug in its *suidperl* component that allowed the creation of files in arbitrary places through clever manipulation of the `PERLIO_DEBUG` environment variable. These files are writable by anyone. This is particularly ironic since *suidperl* is touted as a safe alternative to suid shell scripts and C programs.

- If one could manipulate the */etc/ld.so.preload* file, one can insert code that pre-empts system calls like *getuid*.

- If one can convince the */bin/su* program that one is already the superuser—for example by pre-empting *getuid* to always return zero—, *su* will not ask for the superuser password before executing an arbitrary command as root.

The attack now proceeds along the steps outlined above: download malcode, run malcode, escalate privileges, install backdoor, erase traces (see Figure 5.8).

1. Use the flaw in *mod_auth_any* to download *ex_perl.c*. Do this in chunks so that the download can be spread over multiple HTTP requests. (This is designed to foil behaviour-based intrusion detection systems.) Use the same flaw to launch the C compiler to compile the malcode into *ex_perl*.
2. Using the same flaw, download *attack_mod.o*, a precompiled loadable kernel module (LKM).
3. Using the same flaw, execute *ex_perl*. This program will first of all compile a C file that contains the source to a fake *getuid* function into a shared library */tmp/getuid.so*. That fake *getuid* function will always return `0`, thereby suggesting that the caller is always the superuser.
4. Next, *ex_perl* will execute *suidperl* using a specially prepared `PERLIO_DEBUG` environment variable.
5. This will cause the file */etc/ld.so.preload* to have write permissions for everyone.

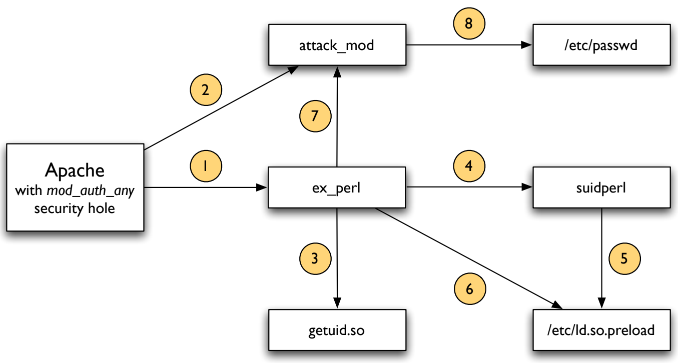


Figure 5.8: A graphical depiction of the attack described in the text. The attack proceeds in the usual phases: download malcode, run malcode, escalate privileges, install backdoor, erase traces.

6. Now that the process can write to the preload file, it will install */tmp/getuid.so* into */etc/ld.so.preload*. The effect is that every command that executes *getuid* will get 0 as the result, thinking that the process has root privileges. The remainder of the attack is executed in a separate shell script by */bin/su*.
7. Install the LKM downloaded above. This step could have been done together with the previous step but the intention here is to spread the attack out over multiple steps in order to make detection and analysis more difficult.
8. The LKM modifies the password file and installs a new system administrator account.
9. Lastly, the kernel module is unloaded and all temporary files erased.

At this point, one might wonder why we take what appears to be an unnecessarily awkward route to our goal of adding another account. Why do we go to all the trouble to install a LKM once we have superuser privileges? Why don't we just modify the password file straightaway? The reason is that we wish to elude not only potential host-based intrusion detection systems (including those that analyze the system calls that are being made by processes), but also those systems that analyze attacks by looking at a process's system calls, such as Backtracker [90]: if we had modified the password file directly, we would have had to issue a system call to open the password file, which would be clearly visible in the process's stream of system calls. Installing the LKM allows us to open the password file from within the kernel, without issuing a system call. Experience with real-world attacks shows that attackers will usually go out of their way to avoid detection; see for example the study by Anderson and Moore on information security economics [10].

The attack generates a process tree containing 168 processes (see Figure 5.9). Most of these processes are concerned with downloading the various source files. For technical reasons, every process spawned through the hole in *mod_auth_any* generates two processes that run */bin/true*. The */bin/true* program does nothing but exit successfully; it has no side effects. Those processes executing this program are therefore irrelevant for the attack: if it were possible for an attacker to forego the creation of these processes, the attack would still succeed.

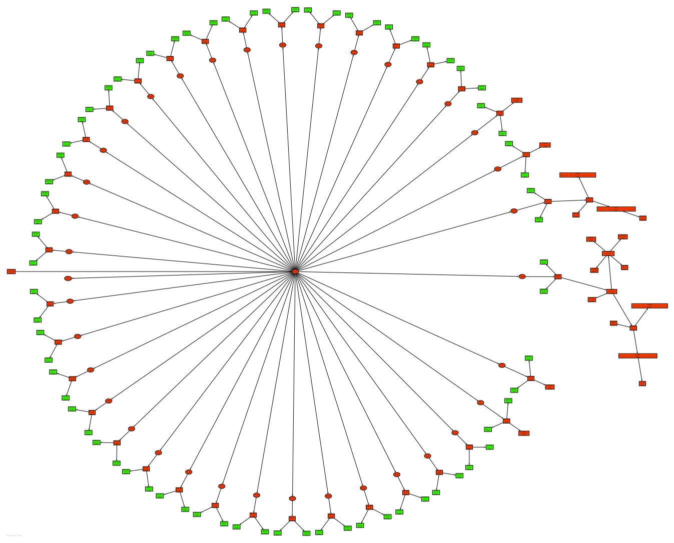


Figure 5.9: The process tree induced by the attack described in the text. Rectangular nodes are processes that execute a program, oval nodes are processes that may create other processes, but that do not execute other programs. To the left is the invocation of Apache. In the center is the Apache daemon. The regular structure that covers most of the circle represents downloading the various source files. The irregular structures on the right represent the rest of the attack.

All the system administrator sees is the modified password file. For him, the system is compromised only after Step 9 above. How does Malfor reconstruct the chain of events?

Malfor considers the set of all 168 processes and applies delta debugging to those processes (see Figure 5.10). After we have replayed a process subset, three outcomes can occur:

- The attack manifests itself (✘): the password file has been modified and the new system administrator account has been added. All processes that were necessary for the attack are therefore included in the subset. On a ✘ outcome, delta debugging knows that those processes not included in the process subset must be irrelevant to the attack: after all, the attack has succeeded even without those processes.

- The attack does not manifest itself (✔). Not all processes that were necessary for the attack are included in the subset. When the outcome is ✔, we know that the process set does not contain all relevant processes. We therefore need to include some of those processes that are currently being excluded.

- The proposed process subset is impossible to replay because it contains nodes whose parent is not included (**?**). Note that **?** runs do not take much time because they are detected before replay is attempted.

All in all, delta debugging executes 1330 tests of which 56 fail and 1274 pass; an additional 1220 tests yield invalid process trees that are identified right away and

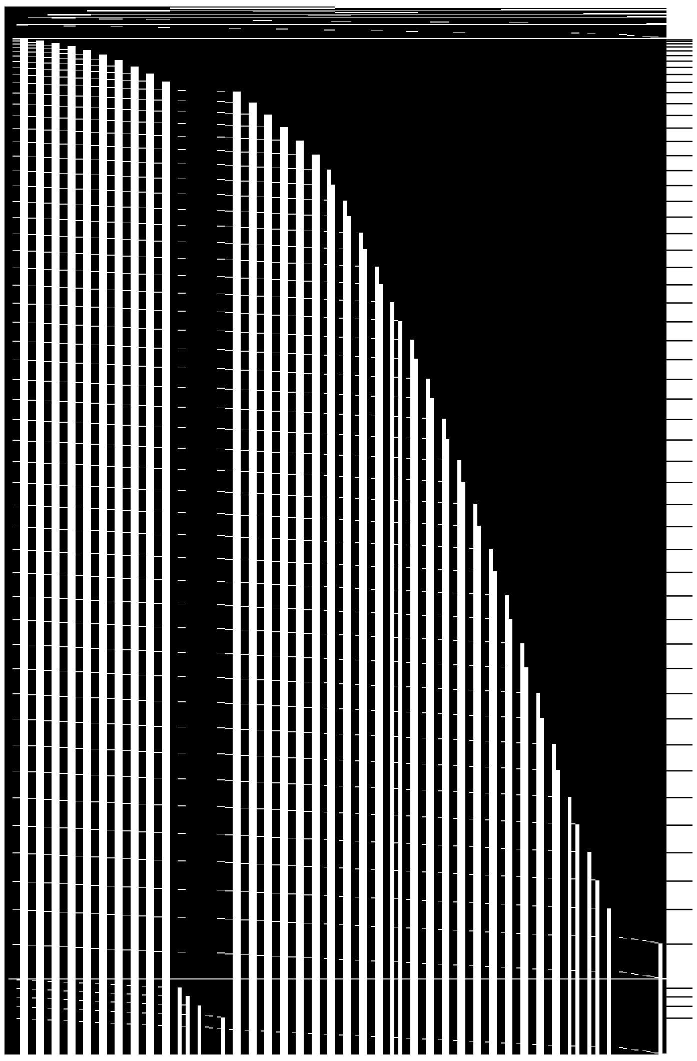


Figure 5.10: Minimizing the process tree in Figure 5.9 with Algorithm D. The figure is analogous to Figure 5.5.

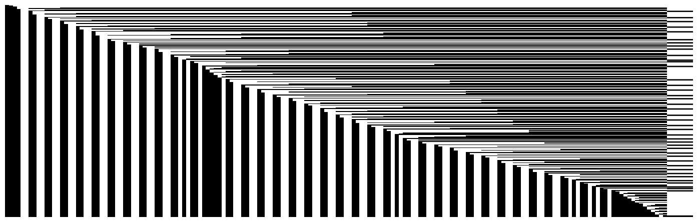


Figure 5.11: Minimizing the process tree in Figure 5.9 with Algorithm B.

What	Total	Median	μ	σ
UML + Replay	364 *s*	26 *s*	26.0 *s*	3.7 *s*
Replay only	174 *s*	13 *s*	12.4 *s*	3.3 *s*

Table 5.2: Performance when analyzing the sample attack on Spud (14 tests). All times are in seconds. The column labeled μ holds the mean and the column labeled σ holds the standard deviation.

which therefore do not take up any time. These tests contain on the average 117 processes. From the 168 processes, Malfor correctly identifies those 96 processes that do not run `/bin/true` as culprits and tags as irrelevant the remaining 72.

On a typical system, and with a typical attack, we would record thousands of processes, only a small fraction of which would be relevant. These are circumstances under which delta debugging works particularly well [75]. However, this attack is incidentally very difficult for delta debugging to analyze: the proportion of relevant processes is high, and the relevant processes are not bunched up together, but rather spread out evenly.

5.3.3 Replacing Delta Debugging by Binary Minimization

If we assume that the preconditions for Algorithm B—the superset condition—are given, we obtain a dramatic increase in performance when we replace Algorithm D by Algorithm B. In this case, Malfor finds the culprit processes using only 269 tests, or about 20% of the tests that Algorithm D needs. An additional 416 tests yield invalid process trees. The resulting diagram is shown on Figure 5.11.

5.3.4 Malfor Performance

For the experiments in this section and the next, the (un-tuned) MySQL database, UML with Solipsy and the outside "attacker" were all on the same host, a 3 GHz Pentium 4 PC running Linux 2.4, both as the host kernel and the UML kernel. All kernels were otherwise unoccupied.

When we actually analyzed the example attack from Section 5.3.1 with Malfor, we got the results summarized in Table 5.2. We can see that the time spent replaying the processes is on average only about half of the UML running time. In other words, about half the time is spent booting and shutting down Linux kernels. Linux startup time is hard to speed up; in our case, we have already disabled all unneeded services. Shutting down a UML, however, takes about ten to eleven seconds in our setup, so if we just killed the UML instead of shutting it down cleanly, we would

		Overhead	
Environment	**Time**	**Ded.**	**UML**
Dedicated machine	21.5 *s*	0%	
UML w/o Solipsy	22.4 *s*	4%	0%
UML w/Solipsy, disabled	24.0 *s*	12%	7%
UML w/full Solipsy	24.3 *s*	13%	8%

Table 5.3: Performance of Spud in various environments. The column labeled "Overhead Ded." has the overhead of running Spud in the given environment relative to running it on a dedicated machine; the column labeled "Overhead UML" has the same overhead relative to running Spud on a UML without Solipsy.

save between $14 \times 10\,s = 140\,s$ and $14 \times 11\,s = 154\,s$ of run time. If we did that, the proportion of replay time to total running time would rise to about 75 percent and the total running time itself would decrease by about 40 percent, to about 217 *s*.

5.3.5 Performance of Capturing

In order to measure the performance of capturing, we ran Spud in successively more complete Solipsy environments, as explained below. In each environment, we called Spud 257 times in rapid succession. One of these times, the service was made to exhibit its vulnerability. Overall, we therefore have one command-line process, one session leader, one intrusion-causing interaction and 256 harmless interactions. Table 5.3 has our results.

On a dedicated system that did not run inside User Mode Linux (UML) or use Solipsy (that is, Spud performed only steps 1 and 2a in Figure 5.2), this took 21.5 *s*. On an UML system that did not have Solipsy (steps 1 and 2a are performed, but the network I/O has to cross a machine boundary), it took 22.4 *s*. Once Solipsy was loaded and enabled, but the service not traced (steps 1, 2, and 2a), execution time rose to 24.0 *s*. When the vulnerable service was also traced and the results put in a database (steps 1, 2, 2a, 3, and 4a), execution time was 24.3 *s*.

While these preliminary results cannot be definitive, we feel that the system calls captured in this experiment (see Table 5.4) are typical of larger systems and that therefore the numbers obtained in this experiment are representative. If that is indeed the case, the overhead of capturing would be about 8% when compared to an un-traced process running inside UML (the "Overhead UML" column Table 5.3), or about 13% when compared to a dedicated machine without either UML or Solipsy (the "Overhead Dedicated" column).

On the one hand, both results are excellent. They also compare well with those by Dunlap and others [53]. On the other hand, it seems as if these numbers are so good only because the program takes so long, even on a dedicated machine (the first row in Table 5.3). Its performance is about six requests per second, which seems rather slow.

5.3.6 Deployability

All of Malfor's components are easily installed: Solipsy is a loadable kernel module, the capture and replay daemons are ordinary processes, the delta debugger is also an

accept	access	bind	brk	close
connect	execve	exit	fcntl64	fork
fstat64	listen	llseek	mmap	munmap
open	read	setsockopt	socket	stat64
unlink	wait4	write		

Table 5.4: List of captured system calls for *spud*. Solipsy captures many more system calls than given in this table; they just weren't used by Spud.

ordinary process that can additionally reside on a remote machine, and the database is an ordinary MySQL database without any tuning. Neither the kernel image nor the captured processes need to be changed. The latter is particularly important if we want to analyze processes whose programs we cannot debug. We therefore believe that Malfor is easily deployed.

5.3.7 Fooling Malfor

Despite Malfor's ability to analyze attacks that cannot be analyzed by any other tool today, it is not infallible. Attackers can outmaneuver Malfor in the following ways:

Detect Capturing/Replaying. In the current implementation, system calls that affect the local system only are executed and not replayed. Using ordinary system calls, an attacker could therefore detect whether the attacked system is in the capturing or the replaying phase and behave differently. This is however merely an artifact of the current implementation and could be remedied.

Exploit Race Conditions. Scheduling decisions are currently not being replayed because there is no easy way to access them in the kernel in a uniform way. For a successful analysis with Malfor, we need repeatability, however—two executions of a process must always yield the same result. Therefore, attacks that exploit race conditions will not be analyzable with Malfor in its current state. But this is not a general limitation of Malfor. Given enough time, Malfor could be extended so that scheduling decisions are captured and replayed just like system calls.

Exploit Parallelism. The current implementation essentially assumes that system calls are atomic. This assumption has recently been challenged by Robert N. M. Watson in a paper where he successfully attacks system call wrappers using concurrency [178]. This is a potentially serious attack because the assumption of atomicity is implicit in much of the Unix API, and capture/replay on the API layer will miss some of the subtleties that occur when real concurrency is present.

Subliminal Channels. Malfor implicitly assumes that only system calls can help a process learn about its environment, but this is not necessarily true. If processes use subliminal channels to communicate, they will bypass Malfor altogether. The theory of information flow is quite new; see for example the PhD thesis by Heiko Mantel [108]. The theory does not yet allow the analysis of

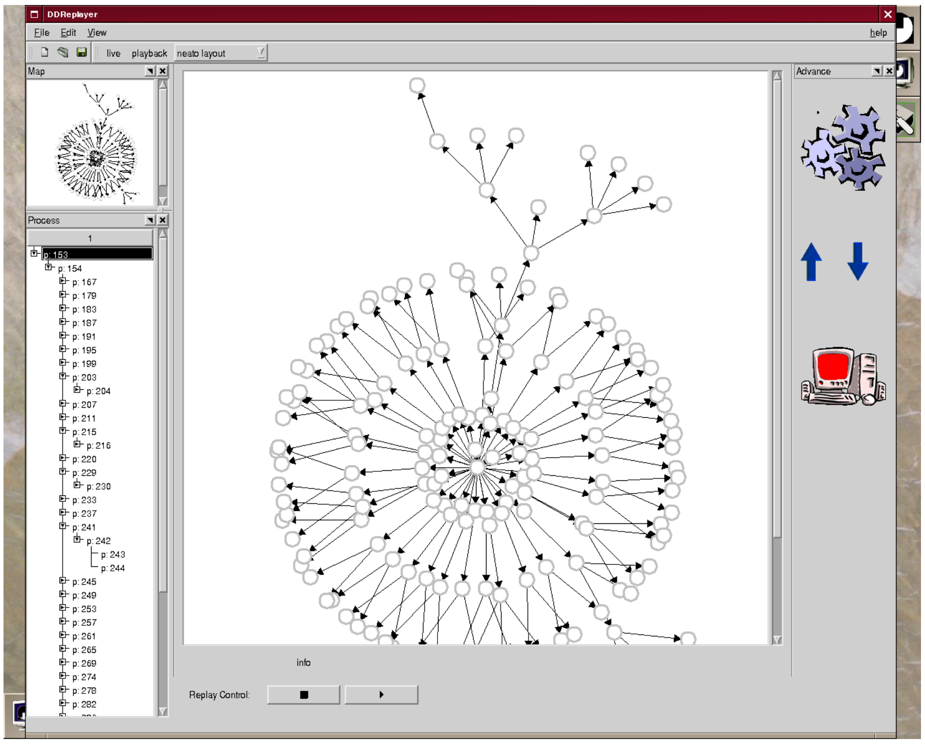


Figure 5.12: The Malfor visualization tool. It can run Malfor live, visualizing Malfor events as they happen, or it can be run in instant-replay mode, where it re-visualizes a Malfor run that has been saved to a file.

possible data flows in real languages under real conditions, for example for concurent programs written in C. Therefore, the problem is practically intractable for Malfor and thus presents a real threat.

5.4 Visualizing Malfor

Malfor is a completely automated technique and as such is designed to run autonomously. However, it is sometimes desirable to be able to watch Malfor run. For example, having a visual display is useful when debugging. Or, having run Malfor successfully, one would sometimes like to watch certain parts of the debugging process again in an instant replay, as it were.

The Bachelor thesis by Steven Mai provides such a tool [106], called Qmalfor. Qmalfor is connected to Malfor with a pipe. Malfor sends events to the visualization tool over this pipe. Events include:

- beginning and end of a Malfor run;
- beginning and end of a test (this event contains those processes that are included in the test); and
- process creation and termination.

Qmalfor reads the recorded process tree from the database and uses the free graph-layout program *dot* to create a tree layout, which it then displays. Using *dot* for layouting the tree gives sufficient flexibility to change layouts cheaply, without having to reimplement parts of Qmalfor.

Qmalfor runs in two modes:

- In live mode, it displays events as they happen, in real-time. It extends these events with a time stamp and writes them to a log file.
- In instant replay mode, it reads a previously written log file and visualizes a previous Malfor run without actually running Malfor.

The last mode is useful for instant replays of the process minimization algorithm, but it can of course be used to view the minimized attack. Further visualization would provide visual clues as to what side effects have occurred, such as file or socket I/O.

Qmalfor uses *dot* to lay out the process tree. This makes it possible to use many excellent tree layout algorithms without changing the Qmalfor program. It also colors those parts of the tree that are currently under consideration by the minimization algorithm and will also color process nodes according to their status (not yet extant, extant but will quit at the next system call, exited).

A sample screenshot can be seen in Figure 5.12.

5.5 Beyond Processes

As we have established above, the methods we use to find the processes that cause an intrusion can in principle find the cause for *any* effect. It is therefore natural to extend the approach to find not only the responsible processes but other things that we might want to know about the attack, such as the network inputs that caused it. Such a network input is known as an *attack signature*. This section describes the work done by Stefana Nenova in her Master's Thesis [123].

The objective is to replay processes as above, with the difference that it is not the set of processes that we are minimizing, but instead their network inputs. Since we control every *read*-type system call (*read*, *readv*, *recv*, and *recvfrom*), we control every byte that any process reads.

5.5.1 Input Reassembly

Before we can replay process inputs, we need to reconstruct them from the Solipsy database because processes only rarely input something in one system call. We will give a number of examples of process inputs in increasing order of difficulty. Some of the examples will use *cat*, a program that catenates files but that can also be used to print a file to the screen. We will suppose the existence of a file called *poem.txt* which contains some text. Other examples will use *spud*, the network server first introduced in Section 5.3.1. For the examples below, we will use a single-thread version of spud that handles requests without forking.

One Read. A command like `cat poem.txt` will, for a short poem, produce a sequence of system calls as outlined in Table 5.5. As can be seen, the `cat` command tries to fill a 4 kByte buffer with the contents of the file, but the file contains only

Call	Arguments	Res
open	`"poem.txt"`, O_RDONLY|O_LARGEFILE	3
fstat64	3, {st_mode = S_IFREG|0600, st_size = 728, ...}	0
read	3, `"Two roads diverged in a yellow w"...`, 4096	728
write	1, `"Two roads diverged in a yellow w"...`, 728	728
read	3, `""`, 4096	0

Table 5.5: The sequence of system calls for `cat poem.txt` for a short poem. The entire file is read in one go and printed to stdout.

Call	Arguments	Res
open	`"poem.txt"`, O_RDONLY|O_LARGEFILE	3
fstat64	3, {st_mode = S_IFREG|0600, st_size = 14317, ...}	0
read	3, `"From off a hill whose concave wo"...`, 4096	4096
write	1, `"From off a hill whose concave wo"...`, 4096	4096
read	3, `"st more dear,\nAnd nice affection"...`, 4096	4096
write	1, `"st more dear,\nAnd nice affection"...`, 4096	4096
read	3, `"ot one whose flame my heart so m"...`, 4096	4096
write	1, `"ot one whose flame my heart so m"...`, 4096	4096
read	3, `" running from a fount\nWith brini"...`, 4096	2029
write	1, `" running from a fount\nWith brini"...`, 2029	2029
read	3, `""`, 4096	0

Table 5.6: The sequence of system calls for `cat poem.txt` for a long poem. The file is read in several portions.

728 bytes. Therefore, the file is read in with the first *read* call and is printed to the standard output in one go. The final *read* returns 0, as it is at the end of the file.

If we want to replay `cat` and control its inputs, we can do this easily, since there is only one read call which reads the entire file. The only thing we have to do is to fill the provided buffer with the data we want and to return the correct number of bytes.

Multiple Reads. For a longer poem, a process will need multiple reads to read the entire file; see Table 5.6. When we replay this situation, we will need to keep track of the number of bytes provided to the process so far (or, equivalently, on the number of bytes remaining to be provided). We call this a *logical read operation*.

Multiple Files With Same File Descriptor. When we catenate two files, the first file is opened and printed, then the second one; see Table 5.7. Traditionally, UNIX and similar operating systems give out the smallest unused file descriptor when a file is opened. This causes inputs from two different files to have the same file descriptors because the first file is closed before the second file is opened, which causes the file descriptor for the first file to be available when the second file is opened. This means that we will have to disambiguate two reads even if they have the same file descriptors in order to associate the correct file with each read. The rule is that two reads belong to one logical input operation of there is no intervening *close*, *write* or *lseek* on that file descriptor.

Call	Arguments	Res
open	`"poem.txt"`, O_RDONLY|O_LARGEFILE	3
fstat64	3, {st_mode = S_IFREG|0600, st_size = 14317, ...}	0
read	3, `"From off a hill whose concave wo"...`, 4096	4096
write	1, `"From off a hill whose concave wo"...`, 4096	4096
...	...	...
read	3, `" running from a fount\nWith brini"...`, 4096	2029
write	1, `" running from a fount\nWith brini"...`, 2029	2029
read	3, `""`, 4096	0
close	3	0
open	`"poem.txt"`, O_RDONLY|O_LARGEFILE	3
fstat64	3, {st_mode = S_IFREG|0600, st_size = 14317, ...}	0
read	3, `"From off a hill whose concave wo"...`, 4096	4096
write	1, `"From off a hill whose concave wo"...`, 4096	4096
...	...	...

Table 5.7: The sequence of system calls for `cat poem.txt poem.txt`. Now, two reads on file descriptor 3 do no longer refer to the same file.

Call	Arguments	Res
socket	PF_INET, SOCK_STREAM, IPPROTO_TCP	3
bind	3, {AF_INET, 2000, 0.0.0.0}, 16)	0
listen	3, 5	0
accept	3, {AF_INET, 34301, 127.0.0.1}, [16])	4
read	4, `"INFO\r\n"`, 9	6
...	...	
write	4, `"200 Some message for you at 2007"...`, 53	53
read	4, `"QUIT\r\n"`, 9	6
write	4, `"200 Quitting\r\n"`, 14	14

Table 5.8: The sequence of system calls for two commands sent to *spud*.

The same situation occurs with single-thread *spud* because spud reads a command on a file descriptor and writes the result to the same descriptor before reading the next command; see Table 5.8. There, reading "INFO" and reading "QUIT" are two different logical reads, even though there is no intervening *close*/*open* combination because of the *write* in between the two. In this case, it would be a mistake to associate the two *read* calls with only one read operation. This happens not only with *write*, but also with *lseek*.

5.5.2 Signature Extraction

Besides using Delta Debugging (see Section 3.4) and Binary Minimization (see Section 4.3), Nenova also implemented *Consecutive Binary Minimization*. The key observation here is that relevant inputs tend to occur in bunches and are not randomly distributed across all inputs. Therefore, instead of using binary search directly after we have identified the leftmost relevant input, we may assume that this leftmost input is the beginning of a block of relevant inputs and simply try removing the next input. This algorithm is expected to give good results when used on actual exploits.

The following description gives the details. Algorithm C is a straightforward extension of Algorithm B, shown in Section 4.3, merely adding steps C8–C9. It has the same asymptotic worst-case running time.

Algorithm C. (*Consecutive Binary Minimization.*) Let $C = \{c_1, \ldots, c_n\}$ be a set of failure-causing circumstances. Assuming that $\text{test}(C) = ✗$, and assuming the superset condition, this algorithm computes $\text{culp}\, C$.

C1. [*Initialize.*] Set $T \leftarrow \emptyset$. Set $r \leftarrow n$.
C2. [*Prepare binary search.*] Set $l' \leftarrow 1$ and $r' \leftarrow r$. We will look for the culprit member with the highest index between indices l' and r'.
C3. [*Compute midpoint.*] Let $m = \lfloor (l' + r')/2 \rfloor$.
C4. [*Set up test.*] Set $t \leftarrow \text{test}(\{c_1, \ldots, c_{m-1}\} \cup T)$.
C5. [*Test.*] If $t = ✗$, set $r' \leftarrow m - 1$, otherwise set $l' \leftarrow m$.
C6. [*Loop if not found.*] If $l' < r'$, go to step C3. Otherwise, go on to C7.
C7. [*Loop if found.*] If $t = ✓$, set $T = T \cup \{c_m\}$, and set $r \leftarrow m - 1$. Otherwise go to step C8. If now $r \geq 1$, go to C8. Otherwise, terminate the algorithm with T as the answer.
C8. [*Set up block search.*] Set $b \leftarrow 1$. We will test $T \cup \{c_m, \ldots, c_{m+b}\}$ for increasing values of b.
C9. [*Set up block test.*] Set $t \leftarrow \text{test}(\{c_m, \ldots, c_{m+b}\} \cup T)$.
C10. [*Test.*] If $t = ✗$, set $b \leftarrow b + 1$, and return to Step C9. Otherwise go on to Step C11.
C11. [*Loop.*] Set $T \leftarrow T \cup \{c_m, \ldots c_{m+b}\}$. If $m + b = n$, terminate the algorithm with T as the answer. Otherwise, set $l \leftarrow m + b + 1$ and return to Step C2.

Using Solipsy, Nenova was able to extract signatures for attacks on modified versions of *spud*. It was modified to run in two additional modes:

- In *regular-expression mode*, *spud* accepts commands with arbitrary characters in between the command's characters. For example, if one command is "CREATE", *spud* in regular-expression mode will accept a string like "XXXCXRXXEXXXAXXTXEXX", or in fact any string that matches the regular expression "C.*R.*E.*A.*T.*E". This simulates the obfuscation techniques that some malcode uses to evade detection.

- *n-character buffer-overflow mode* is an extension of regular-expression mode in which *spud* needs some characters before the actual command. For example, if the command is "CREATE", 10-character buffer-overflow mode would require ten additional characters before the actual command, such as "ABCDEFGHIJCREATE". This simulates the so-called *nop sled* that is often used in buffer-overflow attacks. A nop sled is a sequence of characters that translate into machine-language instructions with no effect; it is used in order to make the attack more robust. For the precise mechanics of nop sleds, see for example the books by Koziol at al. [93] or by Hoglund and McGraw [78].

We are also working on extracting signatures for a more realistic attack on *wu-ftpd*. This attack exploits a hole in the server's handling of *tar* file downloads: when a *tar* file is being downloaded, the *tar* command is run using *system*, the C library's way to run a command line interpreter, usually */bin/sh*. When the downloaded file is named, for example, *--use-compress-program=/bin/sh*, and if the *tar* command on the system is GNU tar, then the shell itself is being executed. This can be exploited to run a program called a *bind shell*, which is a shell that takes commands over a network connection. Since *wu-ftpd* is run as root, the machine in question is usually completely compromised.

5.5.3 Signature Generalization

One important goal of our type of signature generation was to have a signature for an attack for which we only have a single instance. This is in contrast to other methods that use massive amounts of attack instances in order to compute signatures; see Nenova's thesis [123] for related work on the subject If the exact same attack is tried again, the signature will match and either an IDS will sound an alert or a proxy can block the malicious request.

When there are multiple different instances of the attack, however, this procedure does not work. Instead, we will get one signature for every attack variant, which will not help ward off future attacks of the same type. For example, the Code Red worm attacked Internet Information Server (IIS), Microsoft's Web server. It specifically targeted IIS's handling of *.ida* files, but also worked on *.idq* files. Malfor could correctly analyze one attack using an *.ida* file and another using a *.idq* file, and produce two signatures for what is essentially the same attack. Or an attack could simply use a different nop sled. Therefore, Nenova's work will also generalize from two minimized signatures.

Slightly more formally, the problem is this: given a set of (minimized) inputs, find a specific rule that subsumes all those inputs. For example, if the rule is a regular expression, all observed inputs must match the expression. The condition that the rule must be specific guards against simply generating a regular expression of the form ".*", which would merely label all inputs as malicious. On the other hand, the most specific rule would simply list all observed inputs as alternatives, which is clearly not desired. There have been many attempts at signature generation and generalization (again, see the thesis [123] for related work), and it is not yet clear which approach is the most promising.

5.5.4 Preliminary Results

Figure 5.13 shows the distribution of the number of tests (replays) needed to find a six-character signature in a 100-character attack vector for regular-expression Spud. Delta debugging needs between 105 and 198 tests (median 158), but both binary debugging variants are much faster, needing between 29 and 50 tests (both values from consecutive binary minimization). The test data in this case were 100 random strings that contained the characters 'C', 'R', 'E', 'A', 'T', and 'E' in sequence, but not necessarily consecutively. This explains the comparatively bad performance of consecutive binary minimization, because there were no contiguous blocks of relevant inputs.

Since the two binary methods perform so much better than delta debugging, Figure 5.14 shows a closeup of the two corresponding boxplots from Figure 5.13. From this it can be seen that pure binary minimization does almost four times better than delta debugging (42 tests for binary minimization versus 158 tests for delta debugging). From the picture it is also clear that consecutive binary minimization does not work well when there are no consecutive blocks of relevant inputs because

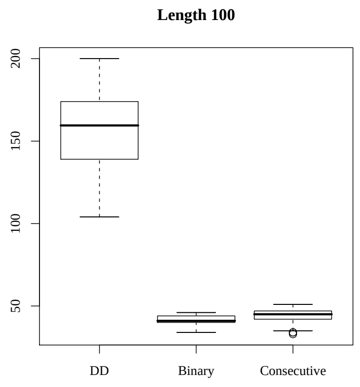


Figure 5.13: Empirical distribution of the number of tests (replays) needed to extract a six-character signature from 100 input characters on spud in regular-expression mode. Delta debugging needs between 105 and 198 tests, but both binary debugging variants are much faster, needing between 29 and 50 tests (both values from consecutive binary minimization).

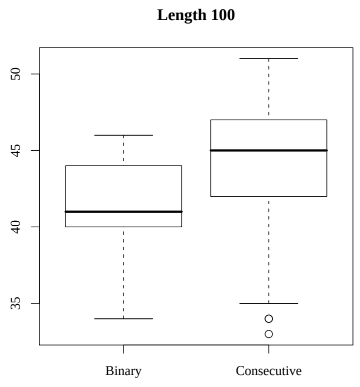


Figure 5.14: Empirical distribution of the number of tests (replays) from Figure 5.13 for the two binary minimization methods.

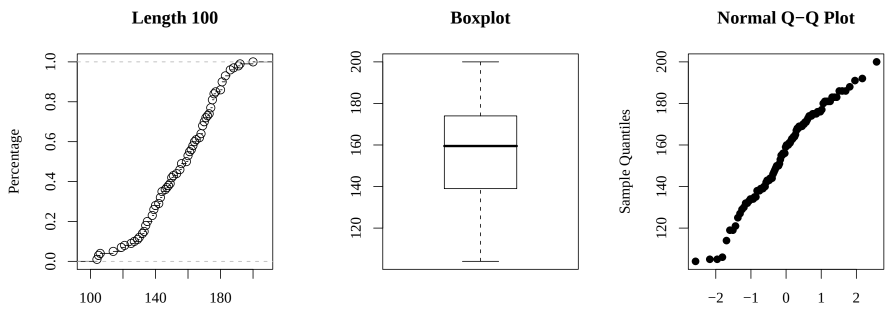


Figure 5.15: Empirical distribution of the number of tests (replays) from Figure 5.13 for delta debugging only.

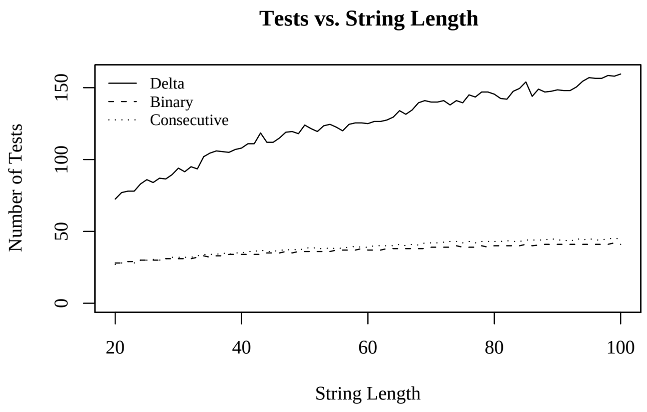


Figure 5.16: Variation of the median number of tests needed to find a signature for various string lengths. The two binary methods scale much better than delta debugging.

the algorithm will waste a test in order to find out whether the input next to the one that has just been found is relevant too. If relevant inputs are scattered randomly and thinly about, this is very unlikely.

Figure 5.15 shows the number of tests for delta debugging in the three-plot style familiar for example from Figure 3.3, except that the leftmost plot shows the *empirial cumulative distribution function*, or *ecdf* of the number of tests. A point (x, y) on the ecdf means that y percent of the sampled data was less than or equal to x. it is a step function that goes from 0 to 1 as x goes from $-\infty$ to ∞. What can be seen here is that the distribution is not nearly as close to the normal distribution as Figure 3.5 was: with a skewness of -0.45, it is slightly skewed to the left. Also, the Q-Q plot shows significant deviations from the normal distribution. This is due to the non-random distribution of relevant inputs: there are always six relevant inputs in Figure 5.15, but there are between one and 32 relevant inputs in Figure 3.5.

How do the various algorithms scale? In other words, how many tests are needed as the attack vector gets longer? For this, we conducted a case study using *spud* in regular-expression mode. We created 100 random strings of length n, where $n = 20, \ldots, 100$ and distributed the characters 'C', 'R', 'E', 'A', 'T', and 'E' in sequence among those n characters. Then we measured the number of tests the various algorithms needed to find that signature. Figure 5.16 shows a plot of the median number of tests for each string length.

What can be seen from this figure is that the two binary methods fare much better than delta debugging. In fact, the number of tests for delta debugging is well approximated by the formula $t = 65.9 + 0.98n$ (at the 0.1% level), whereas the dependency for the two binary algorithms are more logarithmic. (In all fairness, fitting a linear model to the binary algorithm data gives $t = 27 + 0.15n$, also at the 0.1% level, but even so, binary minimization would be 6.5 times faster than delta debugging for large n.)

Binary debugging is suitable for intrusion analysis.

5.6 Controlling Concurrency

As we have seen in the preceding sections, Malfor works only when the processes in question can be replayed consistently. In practice, however, systems of processes have a number of sources of indeterminism that hinder consistent replay:

- In most operating systems, process scheduling is done by programming a real-time clock to generate an interrupt a certain number of times per second. However, clock skew as well as other interrupt sources auch as network cards, keyboards and mice might change processing of the clock interrupt, even when everything else in the system is as it was before.
- When programs exchange messages, the time that messages spend in transit before they are read is not constant across replays. The reasons for this are similar to the reasons why process scheduling causes indeterminism.

This means that when we replay a concurrent system under the control of a minimization algorithm, we might alter the execution paths of the system's constituent processes. This may cause the system calls that are issued during replay not to occur in the same order as they were issued during capturing, or they occur with different parameters, or indeed not at all.

Consider this example: Processes P_1 and P_2 run in parallel and take input from the network. Both processes append their inputs to file f. Assume that for external reasons, P_1 always gets its input before P_2—perhaps a third process feeds first P_1 and then P_2. Then it is true that f always contains P_1's input before P_2's. During replay, it could be because of different process scheduling that P_2 issues its *write* call before P_1. If we want a consistent replay, we must stop P_2 until P_1 has made its *write* call.

If we had only one process, this form of inconsistency couldn't happen because the process would produce outputs sequenctially. We can see from this example that some form of concurrency control is necessary to prevent impossible (or at least implausible) executions during replay. What follows is a formalization and generalization of this plausibility requirement. The basic idea is that real process executions generate traces of system calls when they execute, and that we need to take measures to make sure that the trace that is generated during replay is consistent with the trace that was generated during capturing.

5.6.1 System Calls and Traces

Before we can work on eliminating nondeterminism, we need to define what it means for a replay to be consistent. We have already laid some groundwork in the preceding chapter with the definition of process ID and related concepts (Definition 5.2), but we need more definitions that are specific to system calls and concurrency.

When we deal with system calls, we need to specify their parameters, but we do not want to go to the trouble of specifying exactly what a type is, nor do we want to specify all possible parameter types, especially since they are usually irrelevant fot our discussion. Therefore, we assume the existence of a set *Types* that holds all possible system call parameter types.

Definition 5.5 (System Call Declaration). A *system call declaration* is a function declaration of the form $f\colon \mathit{Types}^k \longrightarrow \mathit{int} \times \mathit{Types}^m$, where $k \geq 0$ and $m \geq 0$ are integers whose values depend on f. We call f the *name* of the system call, k the *number of (input) parameters*, and m the *number of return parameters*. Elements of pointer types $T*$ are either null or point to an instance of T, and are treated as elements of an augmented pointer type $(T*)' = \{x : x = \mathit{null} \vee *x \in T\}$.[6]

Usually, we have $m = 0$, and the system call only returns an integer. Exceptions occur in the *read* family of system calls (*read*, *readv*, *recv*, *recvfrom*, and *recvmsg*), when a buffer is returned containing the bytes that were read.

By convention, the returned integer is 0 if the system call succeeds, and some negative value on error. There are exceptions to this, however. For example, *fork* returns to the parent the forked child's PID, and the *read* and *write* families of system calls return the number of bytes read or written, respectively. This definition does not model value-return parameters, such as in *select*, where the file descriptor sets are modified on return. Value-return parameters are merely optimizations in order to save on parameters. We won't need them for our purposes; where they occur, they are modeled as two parameters, one for input and one for output.

Definition 5.6 (System Call). A *system call* is a quintuplet (P, f, p, v, r), where P is a process ID, f is the name of a system call, and where $p \in \mathit{Types}^k$, $v \in \mathit{int}$, and $r \in \mathit{Types}^m$. We call p the *(input) parameters*, v the *(return) value*, and r the *return parameters* of the call. If $S = (P, f, p, v, r)$ is a system call, we write S_P for P, S_f for f and so forth. If we want to spell out the arguments, we write $P\colon f(p_1, \ldots, p_k, r_1, \ldots, r_m) = v$. For convenience, the r_i may appear among the p_j in a different order. If a system call doesn't return, we write "–" for the return value.

If we followed strictly our convention of always appending output parameters, we would have to write, for example, "100: *read*(1, 100, `"abc\n"`) = 4", instead of the more conventional "100: *read*(1, `"abc\n"`, 100) = 4". Wherever possible, we will put return parameters where they appear in the POSIX system call prototype.

Some system calls do not return when they succeed, such as *exit* or *execve*. In this case, there is no return value; the set of return values is effectively augmented to contain an additonal value for "no value". This is the same augmentation that we have introduced above for pointer types.

[6]We are not modeling the possibility that a $T*$ may not be null, yet may not point to an instance of T. This can happen for example when a pointer is uninitialized or when the memory that contains the pointer is corrupted.

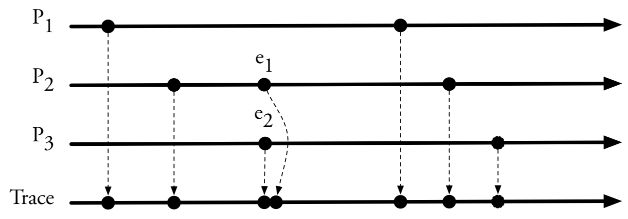


Figure 5.17: Concurrent processes issuing events (system calls). Processes P_2 and P_3 issue events e_1 and e_2 simultaneously, but they will appear in the trace in some order, either as $\langle e_1, e_2 \rangle$ or $\langle e_1, e_2 \rangle$ (shown here).

Our definition does not model that an erroneous system call might leave return parameters unassigned. We assume that default values are substituted in this case. Again, this is irrelevant for our discussion.

Definition 5.7 (Trace Entry, Trace). A *trace* is a finite sequence of system calls, written $T = \langle S_1, \ldots, S_n \rangle$, where $n \geq 0$ is called the *length of the trace*. We write P_k for the P-component of S_k, i_k for the i-component and so forth.

Capturing processes naturally leads to traces because we record the process's system calls sequentially, but superficially, this definition of a trace is slightly problematic for multi-processor or distributed systems because it does not contain a notion of concurrency: If two processes issue system calls at the same time, the trace will still have one system call before the other; see Figure 5.17. In what order should these calls be replayed during replay? We will show later that this is not a problem.

5.6.2 Capturing Traces

We are concerned here with traces that occur during capturing. These traces differ from arbitrary traces in that they actually occurred in a real execution. When we replay processes using these traces, we want to make sure that the traces of the replayed processes are plausible. For example, P: *wait*(*addr*) $= i$ is not plausible if process $i > 0$ has not yet exited, or if i is not a child of P. Other replayed traces might be plausibe, but not desirable. For example, if the captured trace has P_1 writing to file f before P_2, we want to preserve this ordering during replay.

The purpose of this section is to find out those constraints that must hold if replay is to be consistent with what happened during capturing.

Definition 5.8 (Capturing Trace). A trace is called a *capturing trace* if it contains all system calls of captured processes in sequence.

From now on, we will only be considering capturing traces, unless explicitly stated otherwise.

We sometimes want to extract those system calls from a trace that were issued by a single process. Looking at Figure 5.17, we can imagine this operation as a projection of the trace back onto the process:

Definition 5.9 (Projection). Let $T = \langle S_1, \ldots, S_n \rangle$ be a capturing trace, and let P be a process id. We call $T[P] = \langle S'_1, \ldots, S'_m \rangle$ the *projection of T on P* if and only if there exists a strictly monotonous sequence $\langle X_1, \ldots, X_m \rangle$ of integers such that $S'_j = S_{X_i}$ for all $1 \leq j \leq m$. If P does not occur in T, we define $T[P] = \langle\rangle$, the empty sequence.

Note that the definition of $T[P]$ does not suffer from the concurrency problems that the definition of T has: since the calls in $T[P]$ are made by a single process, they happen sequentially by definition.

Lemma 5.3. Let T be a capturing trace. For every process P that occurs in T, $T[P]$ contains all system calls of P, in order.

Proof. Since T is a capturing trace, it contains all system calls of P; hence, $T[P]$ contains all system calls of P, too. Since projection doesn't change the order of system calls—the sequence $\langle X \rangle$ is strictly monotonous—, $T[P]$ contains the calls in the order they appear in T, which is the order in which they were made. □

It should be no surprise that a trace that comes from capturing always has a complete ancestor chain for any process that occurs in it.

Lemma 5.4. Let $T = \langle S_1, \ldots, S_n \rangle$ be a capturing trace, let P and Q be two processes that occur in T, and let P be a descendant of Q. Let $P_1, \ldots, P_m$ be the chain of children leading from P to Q. Then there exist indices $j_1, \ldots, j_m$ such that $j_k < j_{k+1}$ and $P_{j_1} = P$, $P_{j_m} = Q$, $f_{j_k} = \mathit{fork}$, and $v_{j_k} = P_{j_{k+1}}$ for $1 \le k < m$.[7]

Proof. First of all, there must exist *fork* calls that have created $P_1, \ldots, P_m$, whether we have captured them or not. We prove the lemma by induction on m. Induction starts at $m = 2$ because for $m = 1$, Q is not an ancestor of P in the sense of Definition 5.2. For $m = 2$, P is Q's parent, so P must eventually issue $P : \mathit{fork}() = Q$. From Section 5.1.1, we know that capturing is closed under parenting, that is, if a process P_k is captured, then so are all of its decendants. Assuming that P_k has any ancestors at all, this means that the *fork* call that creates P_{k+1} as an ancestor of P_k must also be in the trace. □

Another constraint is that a file descriptor needs to be opened before it can be successfully used. Certain system calls create new file descriptors, and other system calls use them. We call the first class *open operations* because it is exemplified by the *open* system call. We call the second class *I/O operations*. Our version is suitable for POSIX systems, but versions for other operating systems are straightforward. Note, however, that the conceptually simple act of opening a file descriptor needs quite some description; this was already noted in Section 5.7.

Definition 5.10 (I/O Operation, Open Operation). A system call S is called an *I/O operation* if and only if S_f is one of *chmod*, *chown*, *fcntl*, *fcntl64*, *readmsg*, *read*, *readv*, *recvfrom*, *recvmsg*, *recv*, *select*, *sendto*, *send*, *stat*, *stat64*, *write*, or *writev*. A system call s is called an *open operation* if and only if S_f is one of *accept*, *dup*, *dup2*, *open*, *pipe*, or *socket*, or if S_f is *fcntl* and the second argument is *F_DUPFD*.

5.6.3 Plausible Traces

Now we can state what it means for a trace to be plausible, that is, when it could have conceivably be a capturing trace. We give here a definition that is suitable for UNIX or indeed any POSIX-conformant system, but versions for other operating systems are probably easy to create, by suitably modifying the definition below. We give formal and plain English versions of the conditions below to ease that process.

[7]We ignore the existence of *clone*, which creates a new kernel thread that behaves just like a new process, except that, according to the manual page, "the new process shares part of its execution context with the parent process", which means that it shares memory, file descriptors and signal handlers.

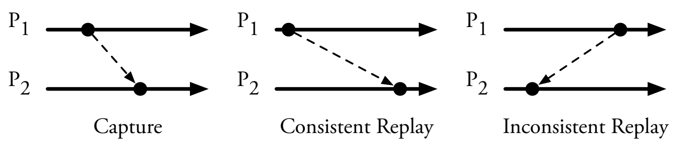


Figure 5.18: Ordering of message send and receive operations during capture (left) replay. If the replay is consistent, message arrows must always point to the right (middle); otherwise, a process would receive a message from the future (right).

Definition 5.11 (Plausible Trace). A trace $\langle S_1, \ldots, S_n \rangle$ (not necessarily a capturing trace) is called *plausible* if and only if all of the conditions below are true.

- For all $1 \le j \le n$, if $f_j = \mathit{fork}$ and $v_j = Q > 0$, then all S_k with $P_k = Q$ have $k > j$. In other words, a process must be created before it can make system calls.
- For all $1 \le j \le n$, if $f_j \in \{\mathit{wait}, \mathit{waitpid}, \mathit{wait3}, \mathit{wait4}\}$, and $v_j = Q > 0$, then there exists some $k < j$ such that $S_k = P_j : \mathit{exit}(s) = -$. In other words, only child processes can be reaped, and they must have exited before reaping.
- For all $1 \le j \le n$, if f_j is an I/O operation on *fd* and $v_j > 0$, then there exists a $1 \le k < j$ such that either $P_k = P_j$ or P_k is an ancestor of P_j, and f_k is an open operation, and $v_k = \mathit{fd}$, and none of the trace elements S_m with $k < m < j$ is $P_m : \mathit{close}(\mathit{fd}) = 0$ or $P_m : \mathit{execve}(\ldots) = -$, with *fd* having the close-on-exec flag set. In other words, successful I/O operations can be done only on open file descriptors.
- For all $1 \le j \le n$, if f_j is an input operation on the shared descriptor *fd* and $v_j = Q > 0$, then there exists a $1 \le k < j$ such that f_k is an output operation on the same descriptor *fd*, and $v_k = Q > 0$. In other words, message reads must not come before their corresponding message writes

If the operating system works as specified, traces gathered during capturing are necessarily plausible. (The first of the bulleted conditions works only if PIDs don't wrap around during capturing. As before in Chapter 5, we will just assume that they do not.) Also, processes can be reaped that didn't call *exit*, for example if they get signals. For those processes, synthetic *exit* calls are inserted just before the corresponding *wait* calls.

The last condition, that message reads must not come before their corresponding message writes has a surprisingly intuitive graphical interpretation. In Figure 5.18, we see on the left process P_1 sending a message to process P_2. The message arrow goes to the right, in the direction of time. A replay of that situation is consistent only if the arrow points to the right (middle diagram); otherwise, a process would receive a message from the future, which is inconsistent with our definition of causality set forth in Section 2.1.3.

5.6.4 Causality in Distributed Systems

The graphical representation shown in Figure 5.18 was used by Mattern and others in order to define and detect causal relationships in distributed systems [113]. They model a distributed system as a number of independent processes, communicating

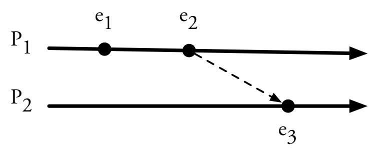


Figure 5.19: Causality relation. In this example, $e_1 \prec e_2$ and $e_2 \prec e_3$, hence $e_1 \rightarrow e_3$.

only by exchanging messages. Processes may issue events, which fall into three categories:

- local events, which only modify the local state of the process;
- send events, which cause a message to be sent to another process; or
- receive events, which are caused by some earlier send event.

Note that local events are trivially (and strictly) ordered: like pearls on a string, they come one after another. This makes it possible to speak of predecessors and successors of events. Mattern then proceeds to order events:

Definition 5.12 (Immediate Predecessor). Let e_1 and e_2 be events. We call e_1 the *immediate predecessor* of e_2 and write $e_1 \prec e_2$ if

- e_1 happens immediately before e_2 in the same process; or
- e_1 is a send event and e_2 is the corresponding receive event.

Definition 5.13 (Causality Relation). Let e_1 and e_2 be events. We define the *causality relation* $\rightarrow$ as the smallest transitive relation on events such that $e_1 \rightarrow e_2$ if e_1 is the immediate predecessor of e_2. (In other words, $\rightarrow$ is the transitive hull of $\prec$.)
We say that e_1 *may causally affect* e_2.

Note that if $e \rightarrow f$, there exist $n > 1$ events $e_1, \ldots, e_n$ such that $e_1 = e$, $e_n = f$, and $e_k \prec e_{k+1}$ for $1 \leq k < n$.

This definition is equivalent to Lamport's "happened-before" relation [95], but calling it the causality relation explains better what we want to do with it, namely find orderings of events so that their causality relationship is preserved; see also Figure 5.19

Lemma 5.5. The causality relation is irreflexive, asymmetric and transitive. That is, it is a strict partial ordering.

Proof. The relation is transitive by definition. For irreflexivity and asymmetry, note that $\prec$ is both irreflexive and asymmetric, hence so is $\rightarrow$. □

Lemma 5.6. Let T be a capturing trace containing events S_i and S_k. If $S_i \rightarrow S_k$, then $i < k$.

Proof. This lemma merely formalizes our notion that the cause must come before the effect. Let $s_1, \ldots, s_n$ be events such that s_j is in T, $s_1 = S_i$, $s_n = S_k$ and $s_j \prec s_{j+1}$. For each pair (s_j, s_{j+1}) we have either that they occur in immediate succession in the same process (hence their indices in T must be increasing), or s_j is a send event with s_{j+1} being the corresponding receive event (hence their indices in T must be increasing as well, because the receive event cannot occur before the send event). □

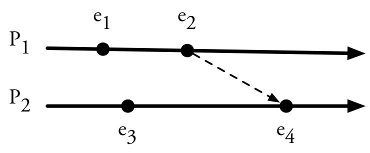


Figure 5.20: Concurrent events. Concurrency is not transitive because in this example, $e_1 \parallel e_3$ and $e_3 \parallel e_2$, but not $e_1 \parallel e_2$.

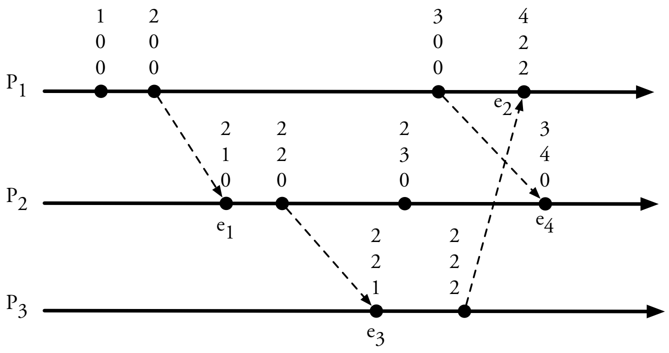


Figure 5.21: Vector time. Every event e is assigned a vector $v = (v_1, \ldots, v_n)$ such that v_k is the index of the latest event in process k that could have influenced e. If we compare the vector times of e_1 and e_2 componentwise, we find that all components for e_1 are either less than or equal to their counterparts for e_2, hence $e_1 \rightarrow e_2$. When we do the same for e_3 and e_4, we find that components 1 and 2 for e_1 are less than those for e_2, whereas component 3 is greater.

If $e_1 \rightarrow e_2$, those events must be replayed in order. Otherwise, the causality relation is violated because e_1 can no longer influence e_2. On the other hand, there are events that can be replayed in any order:

Definition 5.14 (Concurrency). Two events e_1 and e_2 are called *concurrent* and written $e_1 \parallel e_2$ if and only of neither $e_1 \rightarrow e_2$ nor $e_2 \rightarrow e_1$.

Lemma 5.7. The concurrency relation is reflexive and symmetric, but not transitive.

Proof. Since $e_1 \not\rightarrow e_1$, we have $e_1 \parallel e_1$. If $e_1 \parallel e_2$, we have neither $e_1 \rightarrow e_2$ nor $e_2 \rightarrow e_1$, which is equivalent to saying that neither $e_2 \rightarrow e_1$ nor $e_1 \rightarrow e_2$, hence $e_2 \parallel e_1$. Consider the events shown in Figure 5.20. Clearly, $e_1 \parallel e_3$ and $e_3 \parallel e_2$, but $e_1 \rightarrow e_2$, hence $\parallel$ cannot be transitive. □

The fact that $\parallel$ is not transitive means that it is not possible simply to group all events into equivalence classes of simultaneous events that can in turn be replayed in any order. Fortunately, there are other ways to determine causality-preserving orders of events; see below.

Lamport and Mattern show how to compute $\rightarrow$ and $\parallel$ from traces efficiently. We just give a brief overview of the main results. Informally, every process maintains a *time vector*—a so-called *Lamport clock*—showing in its k-th component the number of the most recent event in process k that could have influenced it; see Figure 5.21. We then have $e_1 \rightarrow e_2$ if e_1's *vector time* is less than e_2's, component by component.

Formally, if there are n processes, we assign a vector $v = (v_1, \ldots, v_n)$ of integers to events in a process according to the following rules:

1. Initially, $v_k \leftarrow 0$ for all $1 \le k \le n$.
2. On an internal event, set $v_k \leftarrow v_k + 1$.
3. On sending a message m, update the vector time as in step 2 and attach v to m.
4. On receiving a message with attached vector v', increment v as in step 2 and then set $v_k \leftarrow \max(v_k, v'_k)$ for all $1 \le k \le n$.

We can view vectors as being attached to events, but we can also view them as attached to processes and updated when events happen. This makes its implementation efficient.

We now define an ordering on time vectors and show how it relates to the causality relation.

Definition 5.15. Let v_1 and v_2 be two time vectors. We define

1. $v_1 \le v_2$ if and only if $v_{1k} \le v_{2k}$ for all $1 \le k \le n$;
2. $v_1 < v_2$ if and only if $v_1 \le v_2$ and $v_1 \ne v_2$; and
3. $v_1 \parallel v_2$ if neither $v_1 \le v_2$ nor $v_2 \le v_1$.

Time vectors are unique in the sense that every event has a time vector that never occurs again during the computation. This happens because process k always increments the k-th component of its time vector, hence time vectors in process k have a strictly monotonous k component. Therefore we are justified in equating events with their time vectors, and we write $T(e)$ for th vector time associated with event e. Now the connection between vector ordering and causality is simply this:

Theorem 5.8 (Vector Time and Causality). Let e_1 and e_2 be events.

1. $e_1 \rightarrow e_2$ if and only if $T(e_1) < T(e_2)$.
2. $e_1 \parallel e_2$ if and only if $T(e_1) \parallel T(e_2)$.

Proof. See the paper by Mattern and Schwarz [113].The basic idea is that if $e_1 \rightarrow e_2$, the set of events that can causally influence e_1 must be a subset of the set of events that can causally influence e_2, hence $T(e_1) < T(e_2)$. The same argument applies in the other order as well. The second property follows directly from the first and from the definition of $\parallel$. □

For example, consider events e_1 and e_2 in Figure 5.21. They have vector times $t_1 = (2, 1, 0)$ and $t_2 = (4, 2, 2)$, respectively. We have $t_1 < t_2$ by Definition 5.15, and indeed $e_1 \rightarrow e_2$. Consider on the other hand events e_3 and e_4 with vector times $t_3 = ()$ and $t_4 = ()$, respectively. Since $t_{31} < t_{41}$ and $t_{32} < t_{42}$, but $t_{33} > t_{43}$, we have $t_3 \parallel t_4$ and hence $e_3 \parallel e_4$.

The relations $\rightarrow$ and $\parallel$ model concurrent causality.

Computing the relations $\to$ and $\|$ during a distributed computation is possible with the algorithms given above, but it is also inefficient, since every message must be tagged with a vector of size n. There are ways to compress the time vectors, which are explained in Mattern and Schwarz's paper [113], but we don't need such methods, because we only need to compute the above relations from traces, that is, the computation occurs *offline*.

Computation of $\to$ and $\|$ is efficient.

5.6.5 Causality-Preserving Replay

If $e_1 \parallel e_2$, we can replay them in any order without violating the causality relation. Graphically, we may stretch and compress a process's execution as long as send/receive event arrows still point to the right. If $e_1 \to e_2$ and if both events happen in different processes, there is at least one send/receive event pair in any event chain that connects the two. It should then suffice to replay send/receive event pairs in the correct order, since events inside a process will be replayed in the correct order anyway—the process will make the corresponding system calls in the correct order, because it is running the same program as during capturing.

Formally, assume we have managed to put system calls into the two categories "local event" and "send/receive event", defined $\prec$ on a trace, and computed $\to$. Then we can expect two distributed computations to yield the same result if events that affect others causally happen in the same order. The order of concurrent events does not matter.

Theorem 5.9 (Causality-Preserving Replay). Let $T = \langle S_1, \dots, S_n \rangle$ be a capturing trace and let $T' = \langle S'_1, \dots S'_n \rangle$ be a trace. If the following conditions hold:

1. T' is a permutation of T, that is, there exists a permutation π of $\{1, \dots, n\}$ such that $S_i = S'_{\pi(i)}$ for all $1 \le i \le n$; and
2. if $S_i \to S_k$, then $\pi(i) < \pi(k)$ for all $1 \le i, k \le n$,

then $S'_{\pi(i)} \to S'_{\pi(k)}$ only if $S_i \to S_k$ for all $1 \le i, k \le n$. In this case, we call T' a *causality-preserving replay* of T.

Proof. Assume $S_i \to S_k$. It follows that $\pi(i) < \pi(k)$ and there exist events $S_i = S_1 \prec \dots \prec S_n = S_k$. We will prove the conclusion by induction on n. First consider the case $n = 2$. If both events S_1 and S_2 happen in the same process, then $S'_{\pi(1)}$ is a predecessor of $S'_{\pi(2)}$, hence $S'_{\pi(1)} \to S'_{\pi(2)}$. If both events happen in different processes, then S_1 is a send event and S_2 is its corresponding receive event. Since $\pi(1) < \pi(2)$, the send event $S'_{\pi(1)}$ happens before the receive event $S'_{\pi(2)}$, hence $S'_{\pi(1)} \prec S'_{\pi(2)}$, hence $S'_{\pi(1)} \to S'_{\pi(2)}$. Now assume $n > 2$ and that we have proved the assertion for $2, \dots, n-1$. The same argument as above will establish that $S'_{\pi(n-1)} \prec S'_{\pi(n)}$. Since $S'_{\pi(1)} \to S'_{\pi(n-1)}$ by assumption, the conclusion follows. $\square$

In practice, this means that once we can compute $\to$ and $\|$ efficiently, we can decide when we need to delay the replay of a process's system call and when we can let it go ahead. We can let a system call e go ahead if it is a local event. If it is a send event, we block it until the corresponding receive event happens (because otherwise we wouldn't know the correct return value: if the receive event never happens, we need to return an error). If it is a receive event, we wait until the corresponding send event happens. If we know already that the corresponding send or receive events will never happen—for example, if the sending or receiving process is not being replayed—, we return an error.

Efficient consistent replay of concurrent systems is possible.

5.7 Lessons Learned

That men do not learn very much from the lessons of history is the most important of all the lessons of history.

—Aldous Huxley

In this section, I want to review some of the issues that were impressed on me while preparing the implementation outlined above. Some of these issues have already been noted by other authors; in this case, they are another data point in support of their theses. Other issues seem to be new, or at least little-known; in these cases, this discussion could help future authors of capture-replay systems.

5.7.1 State Changes

A number of state changes must be correctly tracked so that replay can happen faithfully. If any of those are not correctly tracked, replay may fail. The problem with these state changes is that they are not systematic, but are instead the result of Linux's evolution. That means that there is no recipe that will make replay work if consistently applied, but rather need to be dealt with on a case-by case basis. This section lists some of the more unusual state changes whose implementation in replay tended to get postponed until needed, but which needed to be implemented anyway.

File Descriptors. File descriptors need to be tracked because some of them are going to be replayed by a database lookup, while others need to be executed by the operating system. That means that the file descriptor numbers within the application are not necessarily the same as the ones in the operating system: the replay daemon will have to map between them. Furthermore, the replay daemon cannot afford to be left in the dark about any change in any file descriptor because otherwise the mapping might no longer be correct.

File descriptors are created by a number of calls. Most familiar are *open*, *socket*, and *accept*. Less familiar is *pipe*, which needs to be replayed for the *Apache* attack. File descriptors are shared after *fork*, hence when a process forks, the child must also inherit the file descriptor mapping.

File descriptors are closed with *close*. They are also closed implicitly when the process exits, but that is irrelevant for replay purposes since a process that has exited cannot issue any more system calls. However, some file descriptors are closed on *execve*; this can happen whena process issues a *fcntl* syscall with a *SO_CLOEXEC* argument.

Processes. Process identifiers need to be tracked for two reasons:

- During delta debugging, we will change the set of active processes, so the process identifiers that are issued by the operating system will change from replay to replay; yet, we will want the replayed processes to have the same view as during capturing.

- We will not in general be able to guarantee that the operating system issues process IDs in the same sequence as during capturing. For example, we might debug the replay daemon, forking processes as we go.

Processes are created by *fork*, as everyone knows. However, processes are *not* deleted by *exit*, but rather by *wait*, which may come as a surprise to some. True, all the process's resources (file descriptors, memory, etc.) have been recycled at the time the *wait* suceeds, but the process ID is still in use.

5.7.2 State Mappings

As we have seen in Section 5.2.4, we need to track changes to certain operating system objects in order to be able to map object identifiers such as file descriptors or process identifiers between between the application, the operating system, and the replay daemon. The most general mapping is that of file descriptors, which need to be mapped two ways: between the application and the replay daemon, and between the application and the operating system. (Process IDs also need to be mapped these two ways, but the mappings are identical.) Tracking these mappings accurately is important for accurate replay. If there is a mistake, the application could for example reference files that no longer exist from the operating system's point of view.

Unfortunately, mistakes are easy to make. In one particular instance, the symptom was that the replayed attack on *Apache* hung reading from a file descriptor. After several weeks (sic!) of debugging, it finally turned out that I had forgotten to register the first file descriptor that came out of a *pipe* call. This had the effect that the replay daemon instructed Solipsy to let the operating system execute a *read* call on that file descriptor, but to pre-empt the corresponding write operation, which was executed from the database. This resulted in a blocking read operation, which caused one *Apache* worker to hang; see Figure 5.22.

5.7.3 Why Is Replay So Hard To get Right?

It would be nice to be able to say that I have found a kind of unified theory of system calls, and that I have applied that theory to the replay process in order to be able to say that mistakes like these are no longer possible, but unfortunately, this is not so. These are the reasons, as I see them:

Idiosyncrasy. Linux shapes its services not according to any abstract orthogonal service matrix with nice theoretical properties, but according to the needs of real programmers with real problems. In this, Linux is similar to other Unices and to Windows. That means in practice that system calls may behave idiosyncratically and that this idiosyncrasy must be faithfully tracked by the replay daemon. This amounts to a partial reimplementation of operating system semantics; if this reimplementation is incomplete or buggy, replaying a process might fail or give incorrect results. This said, some idiosyncrasies are more the result of bad design rather than inevitable quirks of difficult-to-implement programmer requirements.

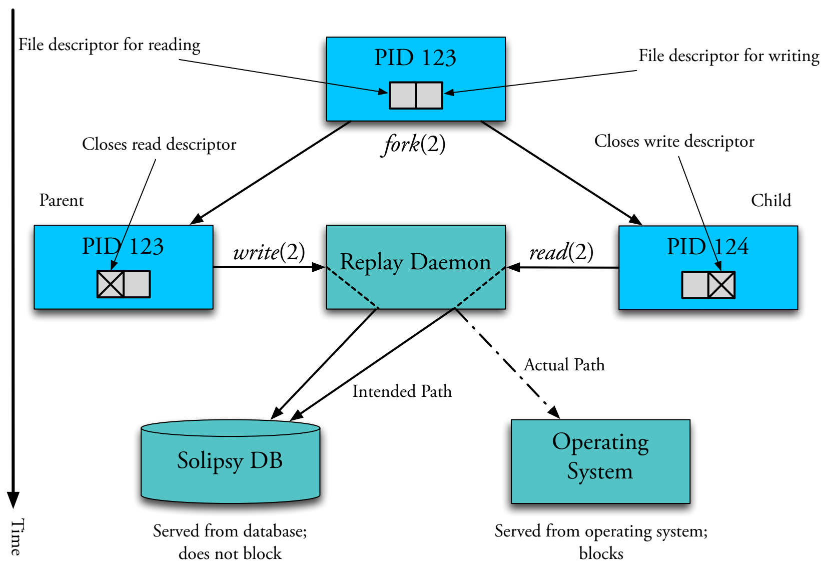


Figure 5.22: The "Pipe Bug" that caused the replay of the *Apache* attack to hang.

The most appalling example of this is *ioctl*. This system call is just a catch-all entry point for device drivers to access driver-specific functionality. Since the arguments to *ioctl* are untyped and driver-specific, it is impossible to write code *a priori* that captures all past and future *ioctl* calls. Hence, it is also impossible to replay in general. If it turns out that attacks use certain *ioctl* calls, they must be explicitly programmed into Solipsy.

These comments apply to a lesser degree to all system calls which have a "command" argument, and, depending on the specific value of the command, "argument" arguments. These calls must make their own command demultiplexing, and they must also extract the correct arguments from the stack, for which they cannot use `va_arg`. One such example is *fcntl*, which has as its third argument either a long integer or a pointer to a `struct flock`. Another, more egregious example is *socketcall*, which is, to my knowledge, unique to Linux. This "system call" collects all socket-related calls under one name. This would be justifiable if the calls shared code, but all that *socketcall* does is demultiplex according to the "command" argument.

Perl Syndrome. The Perl motto is, "There is more than one way to do it" [148], and it applies equally well to Linux and other operating systems with Posix-conformant APIs, such as Solaris or Microsoft Windows. In practice this means that there are often two or more system calls that accomplish essentially the same result. One example is the *wait* family of calls, consisting of *wait*, *waitpid*, *wait3*, and *wait4*; see Table 5.9. The most general of these calls is *wait4*. All others could be implemented within the C library in terms of *wait4*. This would still have been Posix-conformant because Posix only describes C bindings for system calls, not the low-level system call interfaces. Still, these calls are all present. Miss replaying any of them and your process might not replay correctly.

The situation is even more grave for *dup* and friends; see Table 5.10. Here there

Name	Description
wait	Suspends execution until something interesting happens to *any* child.
waitpid	Suspends execution until something interesting happens to a *specific* child.
wait3	Suspends execution until something interesting happens to *any* child, then gets accounting information.
wait4	Suspends execution until something interesting happens to a *specific* child, then gets accounting information.

Table 5.9: The *wait* family of system calls.

Name	Description
dup	Find lowest-numbered free file descriptor and make it an alias for the argument.
dup2	Force first argument to be an alias for the second argument.
fcntl	With second argument of *F_DUPFD*, find lowest-numbered free file descriptor greater than or equal to the third argument and make it an alias for the first argument.

Table 5.10: The *dup* family of system calls.

are three different ways of duplicating a file descriptor, and none can be said to subsume all the others. First, there is *dup*, which allocates the lowest free file descriptor and makes it synonymous to its argument. Then there is *dup2*, which forces the second argument to be an alias for the first, and then there is *fcntl* (or *fcntl64*) with a command argument of *F_DUPFD*, which finds the lowest free file descriptor greater than or equal to its third argument and makes it an alias for the first argument. While it is easy to find *dup2* given *dup*, it is quite difficult to divine the existence of a *fcntl* command that also duplicates a file descriptor. This was the source of another hard-to-find replay bug.

Also interesting is the case of *sleep* or *nanosleep* versus *select*. Apparently, it is quite common to use *select* as a substitute for *sleep*. The manual page even says,

> "Some code calls *select* with all three sets empty, *n* zero, and a non-null timeout as a fairly portable way to sleep with subsecond precision."

Apparently, *Apache* falls into that category. While it isn't strictly necessary to replay *sleep* calls, it does speed up the replay process, because the sleeping can just be skipped. Also, it is somewhat annoying if you think that you have covered all your bases and then to realize that you missed one.

Another pitfall is that *close* is not the only way to close a file. It could also be that a file descriptor has the close-on-exec flag set by a *fcntl* call and subsequently, *execve* was called. If this state change is not correctly tracked, then the application and the operating system could have much different ideas about how many files are kept open by the application.

It's Too Big. Rob Pike argues that

> "[Systems Software Research contains] [t]oo much phenomenology: invention has been replaced by observation. Today we see papers comparing interrupt latency on Linux vs. Windows. They may be interesting, they may even be relevant, but they aren't research. [...]
>
> With so many external constraints, and so many things already done, much of the interesting work requires effort on a large scale. Many personyears are required to write a modern, realistic system. That is beyond the scope of most university departments.
>
> Also, the time scale is long: from design to final version can be five years. Again, that's beyond the scope of most grad students.
>
> This means that industry tends to do the big, defining projects—operating systems, infrastructure, etc.—and small research groups must find smaller things to work on.
>
> Three trends result:
>
> 1. Don't build, measure. (Phenomenology, not new things.)
> 2. Don't go for breadth, go for depth. (Microspecialization, not systems work.)
> 3. Take an existing thing and tweak it." [133].

Designing, implementing and testing Solipsy and Malfor took just about twenty months. During these twenty months, a single publishable paper emerged. Had it not been for the encouragement given to me by my advisor, the work would have been abandoned after about one year, and the effort would have been wasted. The scope of the undertaking was the biggest single miscalculation I made; I had assumed that after capturing worked, replaying would only be a small additional effort. But in fact, capturing is easy, because it is passive. The more difficult thing is replay, because it must actively sit between a process and the operating system, providing a consistent picture to both.

5.7.4 Solutions

What solutions are there that would make replaying easier?

More Regular System Calls. The existence of system calls like *fcntl*, where the meaning and type of the third argument is dependent on the value of the second was perhaps justifiable thirty years ago; today, these calls only complicate things needlessly. For example, one variation of *fcntl* has as its third argument a pointer, in another variant, the third argument is a long integer. On a 64-bit system, a C compiler would be free to implement the long integer as a 32-bit value, but the pointer would have to be 64 bits long. In this case, the system call proper (in the kernel) needs an interface that is different from the C binding, which makes maintenance worse than it already is.

Needless to say, a system call with anything-goes semantics like *ioctl* should be replaced by better mechanisms, perhaps taken from the design of microkernels.

No Redundant System Calls. Who says that the system call interface needs to be close to the (standards-prescribed) C binding? It is entirely possible to use a clean, orthogonal system call interface below the untidy but necessary C binding.

It would certainly help if there were only one system call per function. It would suffice if there were one call that implemented *wait4*. Likewise, it is easy to imagine a *super_dup* syscall that handled all variations on duplicating file descriptors. It would not be necessary to forgo the variety that the current API offers; this variety could just as easily be handled inside the C library, just like several varieties of *exec* are expressed as variations of *execve*.

Still, some duplication will be unavoidable. For example, it is difficult to see how *select* should be implemented so that it can not function as a substitute for *sleep* under certain circumstances. The implementation will at least become more complicated. But it is still a good idea to reduce the amount of redundancy to the absolute minimum, not just for the sake of simplifying replay, but for ease of maintenance, too.

Designing the Kernel With an Eye to Replay. There should be a regular and approved method of changing kernel functionality in a well-defined way for authorized modules. For example, the kernel could provide hooks at crucial points during system call execution, or when other decisions are made that affect (or could affect) replay, such as signal delivery or scheduling.

Chapter 6

Forecasting Vulnerable Components

Early morning: low cloud and westerly winds;
Later: clear skies over the Channel and later over the Normandy beaches
—Weather forecast for June 6, 1944

In the previous chapters, we have seen how to analyze break-ins after they have happened. The key was to isolate the intrusion-causing circumstances through minimization. In this chapter, we will use entirely different techniques to find vulnerabilities before they are deployed to be exploited. Whereas minimization was an *experimental* technique—it designed, carried out and evaluated the impact of changes to a process's operating environment—this technique is *empirical*—relying on observation, without actively changing anything.

This chapter first introduces a method to *assign source code components to bug reports*. It then presents evidence that *vulnerabilities are correlated with certain features of the program text*, notably *import statements* and *function calls*. Next, we use that correlation to *forecast places in the program text that may contain unknown vulnerabilities*.

Some of the work that is presented in this chapter was done by Christian Holler in his Bachelor Thesis [79]. The results of this chapter have also been published [126].

6.1 Introduction

Many software security problems are instances of general patterns, such as buffer overflow or format string vulnerabilities. Another general pattern is the time-of-check/time-of-use vulnerability in which security decisions are made before the affected objects are used. Such vulnerabilities usually occur in far-apart places in program code, which illustrates that such patterns need not be local.

Some problems, though, are specific to a single project or problem domain: JavaScript programs escaping their jails are a problem only in web browsers, as is cross-site scripting; SQL insertion makes sense only in database-using web applications; and so on. To improve the security of software, we must therefore not only

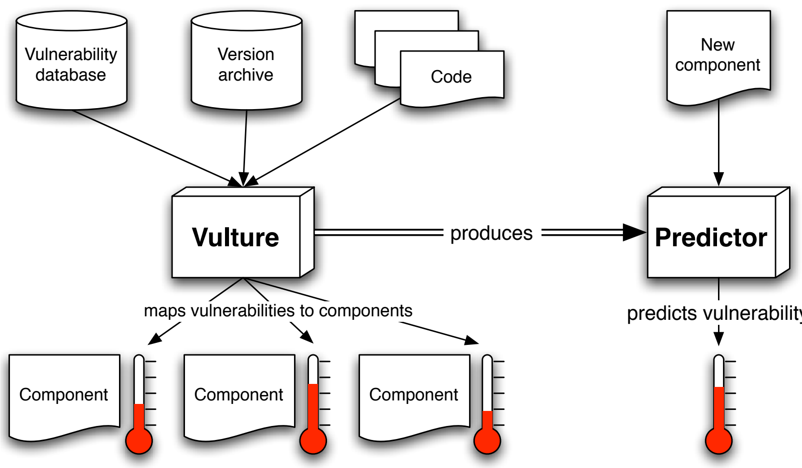


Figure 6.1: How Vulture works. Vulture mines a *vulnerability database* (e.g. a Bugzilla subset), a *version archive* (e.g. CVS), and a *code base*, and maps past vulnerabilities to components. The resulting *predictor* predicts the *future vulnerabilities of new components,* based on their imports.

look for general problem patterns, but also learn *specific* patterns that apply only to the software at hand.

Modern software development usually does a good job in tracking past vulnerabilities. The Mozilla project, for instance, maintains a vulnerability database which records all incidents, together with bug identification numbers. However, these databases do not tell how these vulnerabilities are distributed across the Mozilla codebase. Our *Vulture* tool automatically mines a vulnerability database and associates the reports with the *change history* to map vulnerabilities to individual components; see Figure 6.1.

One of Vulture's results is a *distribution of vulnerabilities* across the entire codebase. Figure 6.2 shows this distribution for Mozilla: the darker a component, the more vulnerabilities were fixed in the past. What is immediately apparent is that the distribution is very uneven: Only 4% of the 10,452 components were involved in security fixes. This raises questions such as "Are there specific code patterns that occur only in vulnerable components?", or "Why do vulnerable components often occur in clusters?"

In our investigation, we were not able to determine code features (such as, say, code complexity or any other code metric, or buffer usage) that would correlate with the number of vulnerabilities. What we found, though, was that vulnerable components shared similar sets of *imports* and *function calls*. In the case of Mozilla, for instance, we found that of the 14 components importing *nsNodeUtils.h*, 13 components (93%) had to be patched because of security leaks. The situation is even worse for those 15 components that import *nsIContent.h*, *nsIInterfaceRequestorUtils.h* and *nsContentUtils.h* together—they *all* had vulnerabilities. This observation can be used to automatically *predict* whether a new component will be vulnerable or not: "Tell me what you import, and I'll tell you how vulnerable you are."

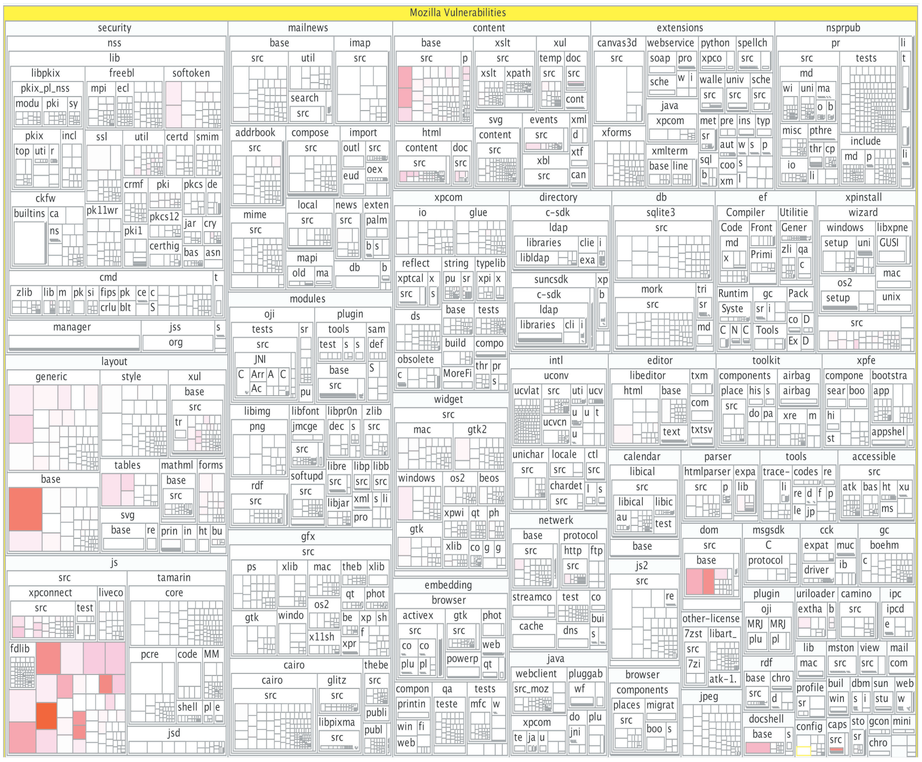


Figure 6.2: Distribution of vulnerabilities within Mozilla's codebase. A component's area is proportional to its size; its shade of gray is proportional to its number of vulnerabilities. A white box means no vulnerabilities, as is the case for 96% of the components.

6.2 Components and Vulnerabilities

6.2.1 Components

For our purposes, a *component* is an entity in a software project that can have vulnerabilities. For Java, components would be *.java* files because they contain both the definition and the implementation of classes. In C++, and to a lesser extent in C, however, the implementation of a component is usually separated from its interface: a class is declared in a header file, and its implementation is contained in a source file. A vulnerability that is reported only for one file of a two-file component is nevertheless a vulnerability of the entire component. For this reason, we will combine equally-named pairs of header and source files into one component.

In C, it is often the case that libraries are built around abstractions that are different from classes. The usual case is that there is one header file that declares a number of structures and functions that operate on them, and several files that contain those functions' implementations. Without a working build environment, it is impossible to tell which source files implement the concepts of which header file. Since we want to apply Vulture to projects where we do not have a working build environment—for example because we want to analyze old versions that we cannot build anymore due to missing third-party software—, we simply treat files which have no equally-named counterpart as components containing just that file. We will subsequently refer to components without any filename extensions.

Of course, some components may naturally be self-contained. For example, a component may consist only of a header file that includes all the necessary implementation as inline functions there. Templates must be defined in header files. A component may also not have a header file. For example, the file containing a program's *main* function will usually not have an associated header file. These components then consist of only one file.

6.2.2 Mapping Vulnerabilities to Components

A *vulnerability* is a defect in one or more components that manifests itself as some violation of a security policy. Vulnerabilities are announced in security advisories that provide users workarounds or pointers to fixed versions and help them avoid security problems. In the case of Mozilla, advisories also refer to a bug report in the Bugzilla database. We use this information, to map vulnerabilities to components through the fixes that remove the defect.

First we retrieve all advisories from the Web to collect vulnerabilities. In case of Mozilla, this is a web page entitled "Known Vulnerabilities in Mozilla Products", `www.mozilla.org/projects/security/known-vulnerabilities.html`. We then search for references to the Bugzilla database that typically take the form of links to its web interface:

```
https://bugzilla.mozilla.org/show_bug.cgi?id=362213
```

The number at the end of this URL is the *bug identifier* of the defect that caused the vulnerability. We collect all bug identifiers and use them to identify the corresponding fixes in the version archive. In version archives every change is annotated with a message that describes the reason for that change. In order to identify the fixes for a particular defect, say 362213, we search these messages for bug identifiers such as "362213", "Bug #362213", and "fix 362213" (see also Figure 6.3). This approach is described in detail by Śliwerski et al. [150] and extends the approaches introduced by Fischer et al. [62] and by Čubranić et al. [45].

Once we have identified the fixes of vulnerabilities, we can easily map the names of the corrected files to components. Note that a security advisory can contain several references to defects, and a defect can be fixed in several files.

It is important to note that we do not analyze binary patches to programs, but source code repository commits. Binary patches usually address other bugs as well, or contain functionalityt enhancements, whereas commits are very specific, fixing only one vulnerability at a time. This is why we can determine the affected components with confidence.

6.2.3 Vulnerable Components in Mozilla

Mozilla as of 4 January 2007 contains 1,799 directories and 13,111 C/C++ files which are combined into 10,452 components. There were 134 vulnerability advisories, pointing to 302 bug reports. Of all 10,452 components, only 424 or 4.05% were vulnerable.

Security vulnerabilities in Mozilla are announced through Mozilla Foundation Security Advisories (MFSAs) since January 2005 and are available through the Mozilla

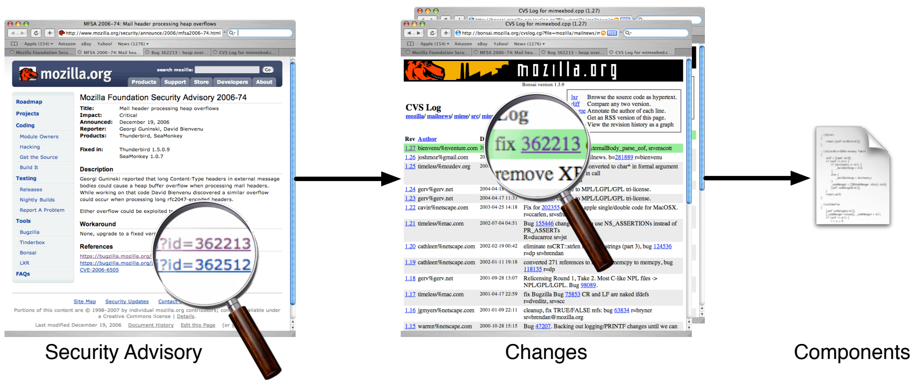


Figure 6.3: Mapping Mozilla vulnerabilities to changes. We extract bug identifiers from security advisories, search for the fix in the version archive, and from the corrected files, we infer the component(s) affected by the vulnerability.

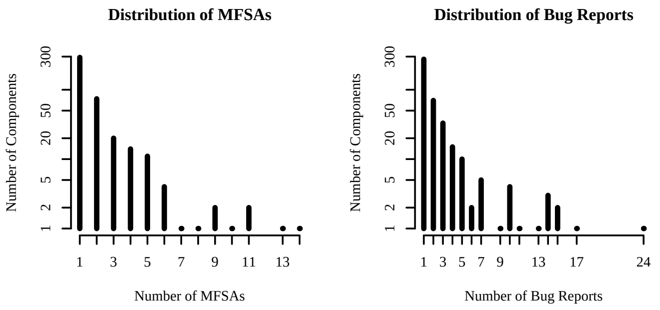


Figure 6.4: Distribution of Mozilla Foundation Security Advisories (MFSAs). The y axis is logarithmic.

Foundation's web site [118]. These advisories describe the vulnerability and give assorted information, such as Bugzilla bug identification numbers. Of all 302 vulnerability-related bug reports, 280 or 92.7% could be assigned to components using the techniques described above.

Some bug reports in Bugzilla [117] are not accessible without an authenticated account. We suppose that these reports concern vulnerabilities that have high impact but that are not yet fixed, either in Mozilla itself or in other software that uses the Mozilla codebase. In many cases, we were still able to assign bug reports to files automatically because the CVS log message contained the bug report number. By looking at the diffs, it would therefore have been possible to derive what the vulnerability was. Denying access to these bug reports is thus largely ineffectual and might even serve to alert blackhats to potential high-value targets.

If a component has a vulnerability-related bug report associated with it, we call it *vulnerable*. In contrast to a vulnerable component, a *neutral* component has had no vulnerability-related bug reports associated with it so far.

The distribution of the number of MFSAs can be seen in Figure 6.4. There were twice as many components with one MFSA (292) than all components with two or

Rank	Component	Directory	MFSAs	BRs
1	*nsGlobalWindow*	*dom/src/base*	14	14
2	*jsobj*	*js/src*	13	24
3.5	*jsfun*	*js/src*	11	15
3.5	*nsScriptSecurityManager*	*caps/src*	11	15
5	*jsscript*	*js/src*	10	14
6	*nsDOMClassInfo*	*dom/src/base*	9	10
7	*nsDocShell*	*docshell/base*	9	9
8	*jsinterp*	*js/src*	8	14
9	*nsGenericElement*	*content/base/src*	7	10
10	*nsCSSFrameConstructor*	*layout/base*	6	17

Table 6.1: The top ten most vulnerable components in Mozilla, sorted by associated MFSAs and bug reports (BRs). Components with equal numbers of MFSAs get an averaged rank.

more MFSAs *combined* (132).

One consequence of this empirical observation is that the number of past vulnerability reports is not a good predictor for future reports, because it would miss all the components that have only one report.

As for using other metrics such as lines of code and so on to predict vulnerabilities, studies by Nagappan et al. have shown that there is no single metric that correlates with failures across all considered projects [120].

The top ten most vulnerable components in Mozilla are listed in Table 6.1. The four most vulnerable components all deal with *scripting* in its various forms:

1. *nsGlobalWindow,* with fixes for 14 MFSAs and 14 bug reports, has, among others, a method to set the status bar, which can be called from JavaScript and which will forward the call to the browser chrome.

2. *jsobj* (13 MFSAs; 24 bug reports) contains support for JavaScript objects.

3. *jsfun* (11 MFSAs; 15 bug reports) implements support for JavaScript functions.

4. *nsScriptSecurityManager* (11 MFSAs; 15 bug reports) implements access controls for JavaScript programs.

In the past, JavaScript programs have shown an uncanny ability to break out of their jails, which manifests as a high number of security-related changes to these components.

6.3 How Imports and Function Calls Matter

As discussed in Section 6.2.3, we found that several components related to *scripting* rank among the most vulnerable components. How does a concept like scripting manifest itself in the components' code?

Our central assumption in this work is that what a component does is characterized by its *imports* and *function calls*. A class that implements some form of content—anything that can be in a document's content model—will import *nsIContent.h* and calls functions defined in that header; a class that implements some part of the Document Object Model (DOM) will likely import definitions from and call functions defined in *nsDOMError.h*. And components associated with scripting are characterized by the import of *nsIScriptGlobalObject.h*.

In a strictly layered software system, a component that is located at layer k would import only from components at layer $k + 1$; its imports would pinpoint the layer at which the component resides. In more typical object-oriented systems, components will not be organized in layers; still, its imports will include those components whose services it uses and those interfaces that it implements.

If an interface or component is specified in an insecure way, or specified in a manner that is difficult to use securely, then we would expect many components that use or implement that interface or component to be vulnerable. In other words, we assume that it is a component's *domain*, as given by the services it uses and implements, that determine whether a component is likely to be vulnerable or not.

From now on, we will use the term "features" to mean "imports and function calls". It is of course possible to select other feature sets, such as keywords, authors, time of last change and so on. These may make good candidates for future work. When we want to differentiate between our choice of features and other possible choices, we will revert to calling them "imports and function calls".

How do features correlate with vulnerabilities? For this, we first need a clear understanding of what constitutes an import or a function call and what it means for a set of features to correlate with vulnerability.

6.3.1 Imports

C and C++, a component's *imports* are those files that it references through `#include` preprocessor directives. For Java projects, a component's imports would be referenced by `import` statements. Component extraction in Java is also much easier than for C and C++ since Java does not distinguish between the definition and implementation of a class and because everything has to be encapsulated in classes.

C and C++ imports are handled by the preprocessor and come in three flavors:

`#include <name>` This variant is used to import standard system headers.

`#include "name"` This variant is used to import header files within the current project.

`#include NAME` This variant is a so-called computed include. Here, `NAME` is treated as a preprocessor symbol. When it is finally expanded, it must resolve to one of the two forms mentioned above.

The computation of imports for C and C++ is difficult because the exact semantics of the first two variants are implementation-dependent, usually influenced by compile-time switches and macro values. That means that it is not possible to determine exactly what is imported without a working build environment. We adopted the following heuristics:

- We treat every occurrence of `#include` as an import, even though it may not be encountered in specific compile-time configurations—for example because of conditional compilation. The reason is that we want to obtain all possible import relations, not just the ones that are specific to a particular platform.

```
#ifdef XP_OS2
  if (DosCreatePipe(&pipefd[0], &pipefd[1], 4096) != 0) {
#else
  if (pipe(pipefd) == -1) {
#endif
    fprintf(stderr, "cannot create pipe: %d\n", errno);
    exit(1);
  }
```

Figure 6.5: Extract from *nsprpub/pr/tests/sigpipe.c*, lines 85ff. Parsing C and C++ is generally only possible after preprocessing: attempting to parse these lines without preprocessing results in a syntax error.

- For `<...>`-style includes, we assume that the name inside the angle brackets is not under the project's root directory. That means even if the compile-time switches are set so that, say, `#include <util.h>` is resolved to *src/util/util.h*, We will treat it as a reference to an external include file, different from *src/util/util.h*. it also means that if compile-time switches make `<util.h>` and `<util/util.h>` refer to the same file, we will treat them as two different imports.

- For `"..."`-style includes, we assume that the name inside the double quotes is under the project root, even if compile-time switches might make them refer to different files, as above. If an include references `"util.h"`, but if there is no file called *util.h* under the project root, it is treated as a reference to an external include. If there are two or more files named *util.h*, the include will be treated as importing a "meta-include", consisting conceptionally of all like-named files. There are some additional disambiguation heuristics in place, and Vulture does its best to find out the target of an include directive. For example, when a file includes `"util/util.h"`, and there is more than one file named *util.h*, but only one is in a subdirectory called *util*. However, describing all of them would be tedious.

- Implementing the computed include would require a full preprocessor pass over the source file. This in turn would require us to have a fully compilable (or at least preprocessable) version of the project. Fortunately, this use of the include directive is very rare, so we chose to ignore it.

6.3.2 Function Calls

In C and C++, a *function call* is an expression that could cause the control flow to be transferred to a function when it is executed.[1] A function call is characterized by the name of the function and a parenthesized list of arguments.

Statically extracting function calls from unpreprocessed C or C++ source code is difficult. Even a full parser cannot resolve dynamic dispatch. Parsing with type information would require compilable source code, and even in the absence of dynamic dispatch there are syntax errors caused by some preprocessor statements; see Figure 6.5. As a consequence, we simply treat all occurrences of *identifier*`(...)` and *identifier*`<...>(...)` as function calls.

[1]This cautious phrasing is necessary because of the possibility of inlining.

Keywords are excluded so that `if` or `while` statements are not erroneously classified as function calls. Also, to match only function calls and not function definitions, these patterns must not be followed by an opening curly bracket. But even with these restrictions, there are many other constructs which match these patterns, such as constructors, macros, forward declarations, member function declarations, initialization lists, and C++ functional-style type casts.

Some of these, like constructors and macros, are very similar to function calls and hence are actually desired. The false classifications of forward declarations, member function declarations, initialization lists, and type casts do not seem to affect our results.

In contrast to these undesirable positive classifications, there are also function calls that are not caught by our heuristic, such as function calls using function pointers or overloaded operators. A simple parser without preprocessing will generally not be able to do type checking, and will therefore not be able to correctly classify such calls. However, we believe that this is a rather uncommon practice in C++, especially in bigger projects such as Mozilla because such dynamic calls are more effectively employed through virtual functions. Hence, we ignore this category of call.

6.3.3 Mapping Vulnerabilities to Features

In order to find out which feature combinations are most correlated with vulnerabilities, we use frequent pattern mining [6, 107].

The result of frequent pattern mining is a list of feature sets that frequently occur in vulnerable components. To judge whether these features are significant, we apply the following criteria:

Minimum Support. The pattern must appear in at least 3% of all vulnerable components. (In other words, it must have a minimum support count of 3% of 424, or 13). For function calls, this threshold is raised to 10%, or 42, because otherwise, frequent pattern mining simply takes too long.

Significance. We want to make sure that we only include patterns that are more meaningful than their sub-patterns. For this, we test whether the entire pattern is more specific for vulnerabilities than its sub-patterns. Let I be a set of includes that has passed the minimum-support test. Then for each proper subset $J \subset I$, we look at all files that feature I and at all files that feature $I - J$. We then classify those files into vulnerable and neutral files and then use the resulting contingency table to compute whether additionally featureing J significantly increases the chance of vulnerability. We reject all patterns where we cannot reject the corresponding hypothesis at the 1% level. (In other words, it must be highly unlikely that including J in addition to $I - J$ is independent from vulnerability.) For this, we use χ^2 tests if the entries in the corresponding contingency table are all at least 5, and Fischer exact tests if at least one entry is 4 or less.

For patterns that survive these tests, the probability of it occurring in a vulnerable component is much higher than for its subsets. This is the case even though the conditional probability of having a vulnerability when including these particular includes may be small.

$P(V\|I)$	$V \wedge I$	$!V \wedge I$	Includes
1.00	13	0	*nsIContent.h* · *nsIInterfaceRequestorUtils* · *nsContentUtils.h*
1.00	14	0	*nsIScriptGlobalObject.h* · *nsDOMCID.h*
1.00	19	0	*nsIEventListenerManager.h* · *nsIPresShell.h*
1.00	13	0	*nsISupportsPrimitives.h* · *nsContentUtils.h*
1.00	19	0	*nsReadableUtils.h* · *nsIPrivateDOMEvent.h*
1.00	15	0	*nsIScriptGlobalObject.h* · *nsDOMError.h*
0.97	34	1	*nsCOMPtr* · *nsEventDispatcher.h*
0.97	29	1	*nsReadableUtils.h* · *nsGUIEvent.h*
0.96	22	1	*nsIScriptSecurityManager.h* · *nsIContent.h* · *nsContentUtils.h*
0.95	18	1	*nsWidgetsCID.h* · *nsContentUtils.h*

Table 6.2: Include patterns most associated with vulnerability. The column labeled "Includes" contains the include pattern; the column labeled $P(V|I)$ contains the conditional probability that a component is vulnerable (V) if it includes the pattern (I). The columns labeled $V \wedge I$ and $!V \wedge I$ give the absolute numbers of components that are vulnerable and include the set, and of components that are not vulnerable, but still include the set.

6.3.4 Features in Mozilla

Again, we applied the above techniques to the Mozilla base. In Mozilla, Vulture found 79,494 import relations of the form "component x imports import y", and 9,481 distinct imports. Finding imports is very fast: a simple Perl script goes through the 13,111 C/C++ files in about thirty seconds. We also found 324,822 function call relations of the form "component x calls function y", and 93,265 distinct function names. Finding function calls is not as fast as finding imports: the script needs about 8 minutes to go through the entire Mozilla codebase.

Frequent pattern mining, followed by weeding out insignificant patterns yields 576 include patterns and 2,470 function call patterns. The top ten patterns are shown in Table 6.2. Going through all 576 include patterns additionally reveals that some includes occur often in patterns, but not alone. For example, *nsIDocument.h* appears in 45 patterns, but never appears alone. Components that often appear together with *nsIDocument.h* come from directories *layout/base* or *content/base/public*, just like *nsIDocument* itself.

The table reveals that it implementing or using *nsIContent.h* together with *nsIInterfaceRequestorUtils* and *nsContentUtils.h* correlated with vulnerability in the past. Typical components that imports these are *nsJSEnvironment* or *nsHTMLContentSink*. The first is again concerned with JavaScript, which we already know to be risky. The second has had a problems with a crash involving DHTML that apparently caused memory corruption that could have led to arbitrary code execution (MFSA 2006-64).

Looking at Table 6.2, we see that of the 35 components importing *nsIScriptSecurityManager.h*, *nsIContent.h*, and *nsContentUtils.h*, 34 are vulnerable, while only one is not. This may mean one of two things: either the component is invulnerable or the vulnerability just has not been found yet. At the present time, we are unable to tell which is true. However, the component in question is *nsObjectLoadingContent*. It is a base class that implements a content loading interface and that can be used by content nodes that provide functionality for loading content such as images or applets. It certainly cannot be ruled out that the component has an unknown vulnerability.

6.4 Predicting Vulnerabilities From Features

In order to predict vulnerabilities from features, we need a data structure that captures all of the important information about components and features (such as which component has which features) and vulnerabilities (such as which component has how many vulnerabilities), but abstracts away information that we consider unimportant (such as the component's name). In Figure 6.6, we describe our choice: if there are m components and n features, we write each component as a n-vector of features: $\mathbf{x}_k = (x_{k1}, \dots, x_{kn})$, where for $1 \le k \le m$ and $1 \le j \le n$,

$$x_{kj} = \begin{cases} 1 & \text{if component } i \text{ features feature } j, \\ 0 & \text{otherwise.} \end{cases}$$

We combine all components into $X = (\mathbf{x}_1, \dots, \mathbf{x}_m)^t$, the project's *feature matrix*. Entities that cannot have imports or function calls, such as documentation files, are ignored.

In addition to the feature matrix, we also have the *vulnerability vector* $\mathbf{v} = (v_1, \dots, v_m)$, where v_j is the number of vulnerability-related bug reports associated with component j.

Now assume that we get a new component, $\mathbf{x}_{m+1}$. Our question, "How vulnerable is component $m+1$?" is now equivalent to asking for the rank of v_{m+1} among the values of $\mathbf{v}$, given $\mathbf{x}_{m+1}$; and our other question, "Is component $m + 1$ vulnerable?" is now equivalent to asking whether $v_{m+1} > 0$.

As we have seen in the preceding sections, features are correlated with vulnerabilities. How can we use this information to answer the above questions? Both questions can be posed as machine-learning problems. In machine learning, a parameterized function f, called a *model*, is trained using training data X and $\mathbf{y}$, so that we predict $\hat{\mathbf{y}} = f(X)$. The parameters of f are usually chosen such that the error, that is, the difference between $\mathbf{y}$ and $\hat{\mathbf{y}}$, is minimized. The question, "Is this component vulnerable?" is called *classification* (because it classifies components as vulnerable or not vulnerable), and "Is this component more or less vulnerable than another component?" can be answered with *regression*: by predicting the number of vulnerabilities and then ranking the components accordingly.

In our case, X would be the project's feature matrix, and $\mathbf{y}$ would be the vulnerability vector $\mathbf{v}$. If we now train a model and feed it with a new component x' and if it classifies it as vulnerable, this means that this component has features that were associated with vulnerabilities in other components.

6.4.1 Validation Setup

To test how good features work as predictors for vulnerabilities, we simply split our feature matrix to train and to assess the model. For this purpose, we randomly select a number of rows from X and the corresponding elements from $\mathbf{v}$—collectively called the *training set*—and use this data to train f. Then we use the left-over rows from X and elements from $\mathbf{y}$—the *validation set*—to predict whether the corresponding components are vulnerable and to compare the computed prediction with what we

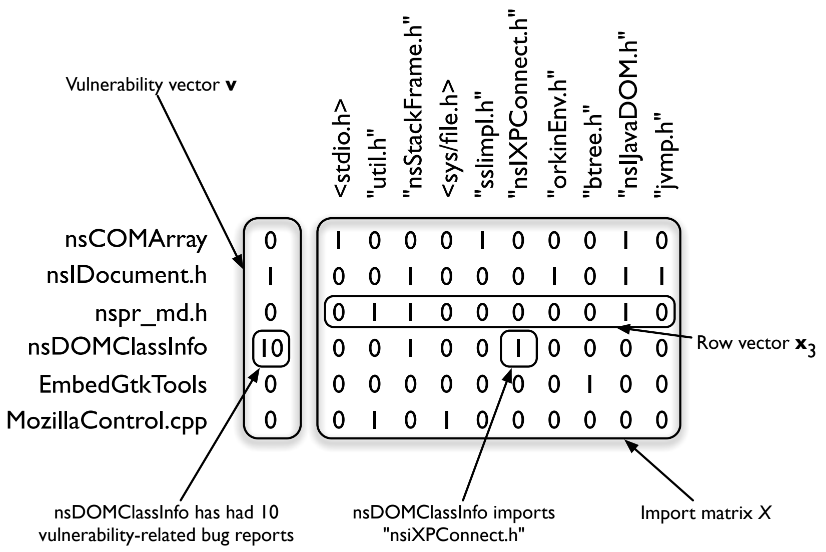

Vulnerability vector **v**
<stdio.h>
"util.h"
"nsStackFrame.h"
<sys/file.h>
"sslimpl.h"
"nsIXPConnect.h"
"orkinEnv.h"
"btree.h"
"nsIJavaDOM.h"
"jvmp.h"
nsCOMArray 0 1 0 0 0 1 0 0 0 1 0
nsIDocument.h 1 0 0 1 0 0 0 1 0 1 1
nspr_md.h 0 0 1 1 0 0 0 0 0 1 0
nsDOMClassInfo 10 0 0 1 0 0 1 0 0 0 0
EmbedGtkTools 0 0 0 0 0 0 0 0 1 0 0
MozillaControl.cpp 0 0 1 0 1 0 0 0 0 0 0
Row vector $\mathbf{x}_3$
nsDOMClassInfo has had 10 vulnerability-related bug reports
nsDOMClassInfo imports "nsiXPConnect.h"
Import matrix X

Figure 6.6: The feature matrix X and the vulnerability vector $\mathbf{v}$. The rows of X contain the imports of a certain component as a binary vector: x_{ik} is 1 if component i imports import k. The vulnerability vector contains the number of vulnerability-related bug reports for that component.

already know from the bug database. It is usually recommended that the training set be twice as large as the validation set, and we are following that recommendation. We are not using a dedicated test set because we will not be selecting a single model, but will instead be looking at the statistical properties of many models and will thus not tend to underestimate the test error of any single model [73, Chapter 7].

One caveat is that the training and validation sets might not contain vulnerable and neutral components in the right proportions. This can happen when there are so few vulnerable components that pure random splitting would produce a great variance in the number of vulnerable components in different splits. We solved this problem by *stratified sampling*, which samples vulnerable and neutral components separately.

6.4.2 Evaluating Classification

For classification, we can now compare the predicted values $\hat{\mathbf{v}}$ with the actual values $\mathbf{v}$ and count how many times our prediction was correct. This gives rise to the measures of *precision* and *recall*, as shown in Figure 6.7:

- The *precision* measures how many of the components predicted as vulnerable actually have shown to be vulnerable. A high precision means a low number of false positives; for our purposes, the predictor is *efficient.*

- The *recall* measures how many of the vulnerable components are actually predicted as such. A high recall means a low number of false negatives; for our purposes, the predictor is *effective.*

Achieving a maximum precision is easy—just predict *zero* components to be vulnerable. This implies maximum efficiency, but also a minimum recall. Likewise,

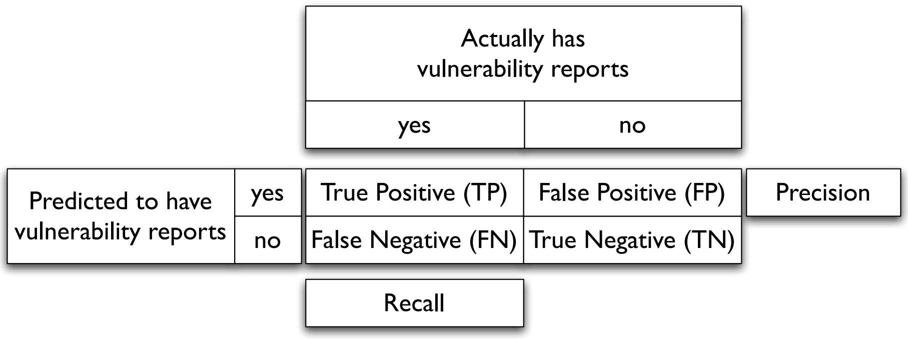


Figure 6.7: Precision and recall explained. Precision is $TP/(TP + FP)$; recall is $TP/(TP + FN)$.

achieving a maximum recall can be done by predicting *all* components to be vulnerable, which is effective, but not efficient, as the precision is low. The key is therefore in achieving a *balance* between precision and recall, or between efficiency and effectiveness.

In order to assess the quality of our predictions, consider a simple cost model. Assume that we have a "testing budget" of T units. Each component out of m total components is either vulnerable or not vulnerable, but up front we do not know which is which. Let us say there are V vulnerabilities distributed arbitrarily among the m components and that if we spend 1 unit on a component, we determine for sure whether the component is vulnerable or not. In a typical software project, V would be much less than m.

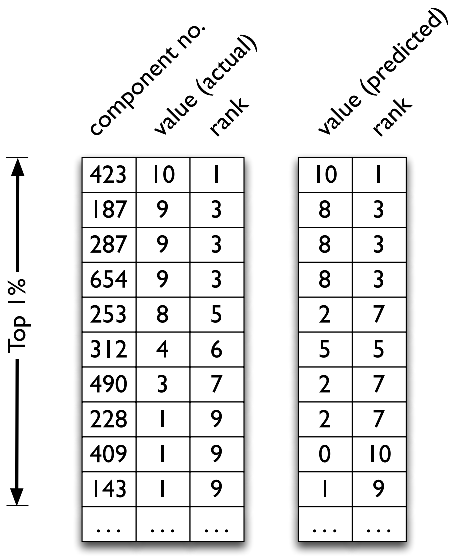
If we fix T, m, and V, and if we have no other information about the components, the optimal strategy for assigning units to components is simply to choose components at random. In this case, the expected return on investment would be Tv/m: we test T components at random, and the fraction of vulnerable components is V/m.

Now assume that we have a predictive method with precision p and that we spend our T units only on components that have been flagged as vulnerable by the method. In this case, the expected return on investment is Tp because the fraction of vulnerable components among the flagged components is p. If $p > V/m$, the predictive method does better than random assignment. In practice, we estimate V by the number of components already known to have vulnerabilities, V', so we will want p to be much larger then V'/m:

$$\text{We want } p \gg V'/m. \tag{6.1}$$

6.4.3 Evaluating Ranking

When we use a regression model, we predict the *number* of vulnerabilities in a component. One standard action based on this prediction would be *allocating quality assurance efforts:* As a manager, we would spend most resources (such as testing, reviewing, etc.) on those components which are the most likely to be vulnerable. To assess the quality of the prediction, we thus want to know whether the *ranks* of the predicted and actual values correlate—in other words, we check whether the allocation of quality assurance efforts would be effective.



component no.	value (actual)	rank
423	10	1
187	9	3
287	9	3
654	9	3
253	8	5
312	4	6
490	3	7
228	1	9
409	1	9
143	1	9
...	...	...

value (predicted)	rank
10	1
8	3
8	3
8	3
2	7
5	5
2	7
2	7
0	10
1	9
...	...

Figure 6.8: Rank correlation is computed for the top 1% of predicted values only. In this case, the rank correlation would be 0.94, which is very high. In the case of ties (equal values), we take the average rank.

To measure the quality of our rank prediction, we took the actual top 1% components and computed their predicted ranks; see Figure 6.8. We then used Spearman's rank correlation coefficient to measure the rank correlation. This is a real number between -1 and $+1$, where $+1$ means that the ranks agree perfectly, zero means that there is no correlation and -1 means that the ranks are in reverse order; see Figure 6.9.

The rank correlation coefficient is however not the only measure of ranking quality and may indeed have problems. In fact, it is inappropriate within the simple cost model from above. Suppose that we can spend T units on testing. In the best possible case, our ranking predicts the actual top T most vulnerable components in the top T slots. The relative order of these components doesn't matter because we will eventually fix all top T components. While high rank correlation values are always good, and while bad rankings will always produce values near zero, the converse is not true: values near zero do not necessarily mean that the ranking is bad, and good rankings will not necessarily produce high rank correlations.

Instead, we extend our simple cost model as follows. Let $p = (p_1, \ldots, p_m)$ be a permutation of $1, \ldots, m$ such that $\hat{\mathbf{v}}_p = (\hat{v}_{p_1}, \ldots, \hat{v}_{p_m})$ is sorted in descending order (that is, $\hat{v}_{p_j} \geq \hat{v}_{p_k}$ for $1 \leq j < k \leq m$), and let q and $\mathbf{v}_q$ be defined accordingly. When we fix component p_j, we fix v_{p_k} vulnerabilities. Therefore, when we fix the top T predicted components, we fix

$$F = \sum_{1 \leq j \leq T} v_{p_j}$$

vulnerabilities, but with optimal ordering, we could have fixed

$$F_{\text{opt}} = \sum_{1 \leq j \leq T} v_{q_j}$$

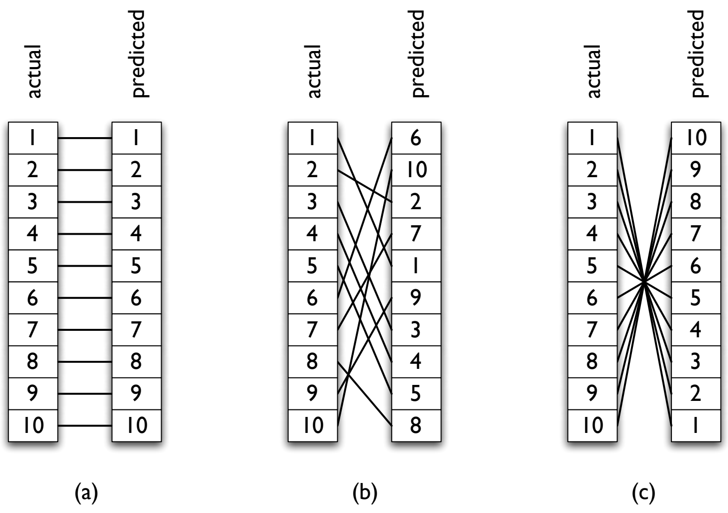


Figure 6.9: Rank correlation explained. Figure (a) has rank correlation +1, Figure (b) has rank correlation near 0, and Figure (c) has rank correlation −1.

vulnerabilities instead. Therefore, we will take the quotient

$$Q = F/F_{\text{opt}} = \sum_{1 \le j \le T} v_{p_j} \Big/ \sum_{1 \le j \le T} v_{q_j} \tag{6.2}$$

as a quality measure for our ranking. This is the fraction of vulnerabilities that we have caught when we used p instead of the optimal ordering q. It will always be between 0 and 1, and higher values are better.

In a typical situation, where we have $V \ll n$ and T small, a random ranking will almost always have $Q = 0$, so our method will be better than a random strategy if Q is always greater than zero. In order to be useful in practice, we will want Q to be significantly greater than zero, say, greater than 1/2, so that we get at least half the vulnerabilities that we would get with optimal allocation:

$$\text{We want } Q > 1/2. \tag{6.3}$$

6.4.4 Prediction using Support Vector Machines

For our model f, we chose support vector machines (SVMs) [164] over other models such as k-nearest-neighbors [73, Chapter 13] because they have a number of advantages:

- When used for classification, SVMs cope well with data that is not linearly separable. (Two sets of n-dimensional points are said to be *linearly separable* if there exists an $n - 1$-dimensional hyperplane that separates the two sets.)

- SVMs are not prone to *overfitting*, which happens when a statistical model has too many parameters: the algorithm will try to minimize the error for the training set, but the parameters will offer many wildly differing combinations that will make the error small. Choosing one such combination will then generally increase the error for the validation set. The only possible remedy for such a method is to decrease the number of parameters. In the words of Forman S. Acton [4],

> [Researchers that try to fit a 300-parameter model by least squares] leap from the known to the unknown with a terrifying innocence and the perennial self-confidence that every parameter is totally justified. It does no good to point out that several parameters are nearly certain to be competing to "explain" the same variations in the data and hence the equation system will be nearly indeterminate. It does no good to point out that *all* large least-squares matrices are striving mightily to be proper subsets of the Hilbert matrix—which is virtually indeterminate and uninvertible—and so even if all 300 parameters were beautifully independent, the fitting equations would still be violently unstable. [...]
>
> Most of this instability is unnecessary, for there is usually a reasonable procedure. Unfortunately, it is undramatic, laborious, and requires thought [...]. They should merely fit a five-parameter model, then a six-parameter one. If all goes well and tere is a statistically valid reduction of the residual variability, then a somewhat more elaborate model may be tried. Somewhere along the line—and it will be much closer to 15 parameters than to 300—the significant improvement will cease and the fitting operation is over. [...] The computer center's director must prevent the looting of valuable computer time by these would-be fitters of many parameters.

- SVMs come with plug-in kernels that can be used to achieve better effectiveness. Kernels add additional dimensions to the training data to achieve greater separability.

- SVMs have parameters that can be automatically tuned to achieve better effectiveness.

6.5 Case Study: Mozilla

To evaluate Vulture's predictive power, we applied it to the code base of Mozilla [119]. Mozilla is a large open-source project that has existed since 1998. It is easily the second most commonly used Internet suite (web browser, email reader, and so on) after Internet Explorer and Outlook.

6.5.1 Data Collection

We examined Mozilla as of January 4, 2007. Vulture mapped vulnerabilities to components, and then created the feature matrix and vulnerability vector as described in Sections 6.2.3 and 6.3.4.

Table 6.3 reports approximate running times for Vulture's different phases when applied to Mozilla. Vulture is so fast that we consider making it part of a real-time feedback system; see Figure 6.16 at the end of this chapter.

The 10,452 × 9,481 import matrix would take up 280 MB of disk space if it were written as a full matrix. It would take about ten minutes just reading the matrix into R. The sparse representation that we used cuts this space requirement down to about

Phase	Time
Downloading and analyzing MFSAs	5 m
Mapping vulnerabilities to components	1 m
Finding includes	0.5 m
Finding function calls	2 m
Creation of SVM, w/classification and regression	0.5 m

Table 6.3: Approximate running times for Vulture's different phases.

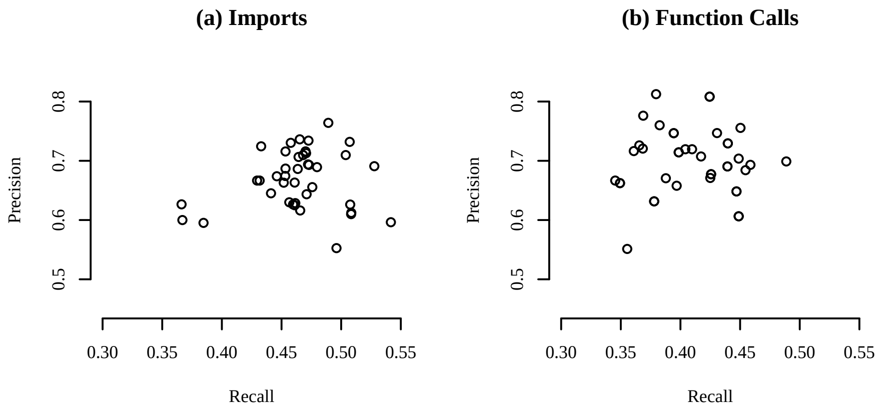


Figure 6.10: Scatterplot of precision/recall values for the 40 experiments. Figure (a) shows the values for imports, Figure (b) shows the corresponding values for function calls.

200 KB. Reading this sparse matrix takes only about one second. The 10,452 × 93,265 function call matrix took up 2.6 MB of disk space.

From the feature matrices and the vulnerability vector, we created 40 random splits using stratified sampling. This ensures that vulnerable and neutral components are present in the training and validation sets in the same proportions. The training set had 6,968 entries and was twice as large as the validation set with 3,484 entries. Finally, we assessed these SVMs with the 40 validation sets.

For the statistical calculations, we used the R system [138] and the SVM implementation available for it [50]. It is very easy to make such calculations with R; the size of all R scripts used in Vulture is just about 200 lines.

6.5.2 Classification

The SVM used the linear kernel with standard parameters.

Figure 6.10 (a) reports the precision and recall values for our 40 random splits. The recall has an average of 0.45 and standard deviation of 0.04, which means that half of all vulnerable components are correctly classified:

> *Of all vulnerable components,*
> *Vulture flags 45% as vulnerable.*

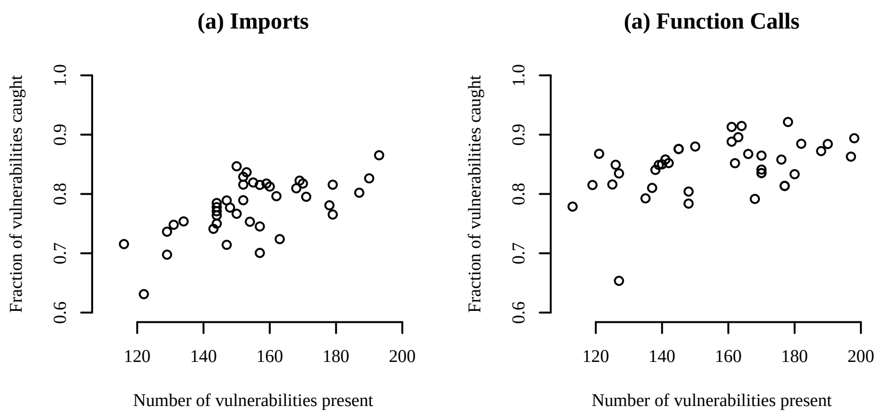


Figure 6.11: Scatterplot of Q versus F_{opt} for the 40 experiments where $T = 30$. Figure (a) shows the results for imports, figure (b) shows the results for function calls. Higher values are better.

For function calls, the precision has a mean of 0.70 and a standard deviation of 0.05, which means that seven out of ten predictions of vulnerability are correct.

Of all components flagged as vulnerable,
70% actually are vulnerable.
Vulture is much better than random selection.

6.5.3 Ranking

The SVM used the linear kernel with standard parameters. The coefficient Q that was introduced in Equation (6.2) was computed for imports and function calls, for $T = 30$. It is shown in Figure 6.11, plotted against F_{opt}. For imports, its mean is 0.78 (with a standard deviation of 0.04), for function calls, it is 0.84 (with a standard deviation of 0.05), for function calls, it is 0.84 (standard deviation 0.05).

Among the top 30 predicted components,
Vulture finds 84% of all vulnerabilities.

The rank correlation coefficients for imports are shown in Figure 6.12. The average rank correlation is +0.46 with a standard deviation of 0.03. The probability that the rank correlations are as high as they are, even when in fact there is no correlation, is less than 0.01 in 32 out of 40 cases, and less than 0.10 in all cases. This makes the results statistically significant.

There is a statistically significant fairly strong
positive correlation between actual and predicted
vulnerability ranks.

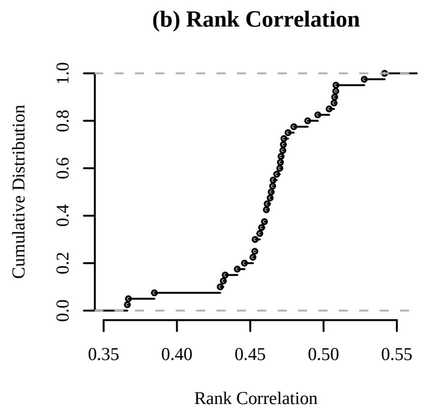


Figure 6.12: Empirical cumulative distribution function for rank correlation coefficients on the top 1% of predicted components.

6.5.4 A Ranking Example

Let us illustrate the ranking correlation by an actual example. As a quality assurance manager, you want to focus extra efforts on the top ten new components that Vulture predicts as vulnerable. Table 6.4 shows such a prediction as produced in one of the random splits. We spent no effort in specifically selecting this particular split. Within the validation set, these would be the components to spend extra effort on. Your effort would be well spent, because *all* of the top ten components actually turn out to be vulnerable. (*SgridRowLayout* and *NsHttpTransaction* are outliers, but still vulnerable.) Furthermore, in your choice of ten, you would recall the top four most vulnerable components, two more would still be in the top ten (at predicted ranks 3 and 6), and two more would be in the top twenty (at predicted ranks 7 and 9).

Focusing on highly ranked components
further improves effectiveness and efficiency.

6.5.5 Forecasting Vulnerable Components

The previous sections have focused on validating the approach, but we have not yet actually forecast any vulnerable components. That would entail taking neutral components—components that were not known to have any vulnerabilities—and rank them according to our predictive method. In other words, we would be ranking *false positives*. Unfortunately, when we use our classification model on the *entire* CVS, we get *no* false positives. What we did instead was to use *leave-one-out cross validation*. The entire process worked as follows:

1. Let $C = \{c_1, \ldots, c_m\}$ be the set of components. Set $k \leftarrow 1$ and $R \leftarrow \emptyset$.
2. Compute a regression model for $C - \{c_k\}$.
3. Compute the prediction $f(c_k)$ according to the above regression model.

Prediction		Validation set	
Rank	Component	Bug reports	Actual rank
1	*NsDOMClassInfo*	10	3.5
2	*SgridRowLayout*	1	95
3	*xpcprivate*	7	6
4	*Jsxml*	11	2
5	*nsGenericHTMLElement*	6	8
6	*Jsgc*	10	3.5
7	*NsJSEnvironment*	4	12
8	*Jsfun*	15	1
9	*NsHTMLLabelElement*	3	18
10	*NsHttpTransaction*	2	35

Table 6.4: The top ten most vulnerable components from a validation set, as produced by Vulture. Six of the predicted top ten are actually in the top ten, with ranks 1, 2, 3.5 (two times), 6, and 8. Two more are in the top twenty, at positions 12 and 18. So eight of the predicted top ten are actually very vulnerable.

4. If $v_k = 0$ (in other words, if the component had no known vulnerabilities), add $\left(c_k, f(c_k)\right)$ to R.
5. Set $k \leftarrow k + 1$. If $k \leq m$, return to step 2.
6. Sort R by $f(c_k)$ and output the top ten values.

We then took the Mozilla CVS as of July 17 and re-assigned vulnerabilities to components in order to see which components had new changes due to vulnerabilities. We were particularly interested in those components that had no previously known vulnerabilities and in whether any of them would turn up in the result from our leave-one-out cross validation. The result of this process can be seen in Table 6.5. It turns out that fully half of our predictions were confirmed within six months!

When we look at the actual changes that were made to the source code, we find that all but one component had changes that actually fixed vulnerabilities in the implementation.

Vulture can make actual predictions with high accuracy.

6.5.6 Discussion

In the simple cost model introduced in Section 6.4.2, we have $m = 10,452$ and $V' = 424$, giving $V'/m = 0.04$. With $p = 0.65$, we see that the condition from Equation (6.1) are fulfilled: Vulture does more than fifteen times better than random assignment. Even when we take into account that in reality $V > v'$, we consider it unlikely that $V/m > p$.

For ranking, all values for the quantity Q introduced in Equation (6.2) are higher than 0.6; the average values are way above that. This more than satisfies our criterion from Equation (6.3): Vulture finds a significant number of vulnerabilities.

Threrfore, our case study shows three things. First of all, allocating quality assurance efforts based on a Vulture prediction achieves a reasonable balance between

Rank	Component	Vulnerable?
1	*js/src/jsxdrapi*	*
2	*js/src/jsscope*	*
3	*modules/plugin/base/src/nsJSNPRuntime*	
4	*js/src/jsatom*	
5	*js/src/jsdate*	
6	*cck/expat/xmlparse/xmlparse*	
7	*layout/xul/base/src/nsSliderFrame*	*
8	*dom/src/base/nsJSUtils*	
9	*layout/tables/nsTableRowFrame*	*
10	*layout/base/nsFrameManager*	*

Table 6.5: Vulture forecast using the CVS from January 4, 2007. The table shows the top ten predicted most risky components. The third column shows "*" if vulnerabilities have surfaced in the predicted component between January 5 and July 17, 2007, encompassing MFSAs 2007-01 through 2007-25. Out of the top ten predicted components, five have had changes due to vulnerability reports within six months of making the prediction.

effectiveness and efficiency. Second, it is *effective* because half of all vulnerable components are actually flagged. And third, Vulture is *efficient* because directing quality assurance efforts on flagged components yields a return of 70%—more than two out of three components are hits. Focusing on the top ranked components will give even better results.

Furthermore, these numbers show that there is empirically an undeniable correlation between imports and vulnerabilities. This correlation can be profitably exploited by tools like Vulture to make predictions that are correct often enough so as to make a difference when allocating testing effort. Vulture has also identified imports that always (or very often) lead to vulnerabilities when used together and can so point out areas that should perhaps be redesigned in a more secure way.

Our cross-validation has further shown that Vulture's predictions are very good—fully half of our predictions were confirmed within six months, and the proportion can only rise as time goes on.

Best of all, Vulture has done all this automatically, quickly, and without the need to resort to intuition or human expertise. This gives programmers and managers much-needed objective data when it comes to identify (a) where past vulnerabilities were located, (b) other components that are likely to be vulnerable, and (c) effectively allocating quality assurance effort.

6.5.7 Threats to Validity

Our work is empirical and statistical: we look at correlations between two phenomena—vulnerabilities on one hand and imports or function calls on the other—, but we do not claim that these are cause-effect relationships. It is clearly not the case that importing some import or calling some function *causes* a vulnerability. Programmers writing that import statement or function call generally have no choice in the matter: they need the service provided by some import or function and therefore have to import or call it, whether they want to or not.

We have identified the following circumstances that could affect the validity of our study:

Study size. The correlations we are seeing with Mozilla could be artifacts that are specific to Mozilla. They might not be as strong in other projects, or the correlations might disappear altogether. From our ownwork analyzing Java projects, we think this is highly unlikely [147]; see also Section 6.7 on related work.

Bugs in the database or the code. The code that imports the CVS or the MFSAs into the database could be buggy; the import matrix and vulnerability vector could have been incorrectly calculated; or the R code that does the assessment could be wrong. All these risks were mitigated either by sampling small subsets and checking them manually for correctness, or by implementing the functionality a second time starting from scratch and comparing the results. For example, the vulnerability matrix was manually checked for some entries, and the R code was rewritten from scratch.

Bugs in the R library. We rely on a third-party R library for the actual computation of the SVM and the predictions [50], but this library was written by experts in the field and has undergone cross-validation, also in work done in our group [147].

Wrong or noisy input data. The Mozilla source files could contain many "noisy" import relations in the sense that some files are imported but actually never used; or the MFSAs could accidentally or deliberately contain wrong information. Our models do not incorporate noise. From manually checking some of the data, we believe the influence of noise to be negligible, especially since results recur with great consistency, but it remains a (remote) possibility.

6.6 Causes of Vulnerabilities

We have seen that, empirically, vulnerabilities correlate with imports, but we have not answered the natural question *why* they correlate. In this section, we will take a look at some vulnerabilities and try to find the reason for the apparent correlation.

For this, we return to Figure 6.2. It is apparent from the figure that some vulnerabilities occur in the same directory as others (called a *vulnerability cluster* or simply *cluster*), whereas others are isolated and occur alone (called a *vulnerability singleton* or simply *singleton*). Looking at specimen examples of both clusters and singletons, we will find evidence that the reasons for the correlation of vulnerablilities and imports are different for the two classes of vulnerable components: clusters will tend to fix the *vulnerable implementation* of some functionality, whereas singletons will tend to *reflect changes in the interfaces of components that were fixed elsewhere*.

If these claims turn out to be correct, we have the following consequences:

- Clusters are more important than singletons. Defects in clusters contain the real security vulnerabilities; singletons occur mostly because of changes in clusters. This also explains why there are so many singletons and so few clusters: there is a relatively small amount of security-critical code in the project and when that is fixed in the interface, many places that use this code will have to adapt to the changed interfase.

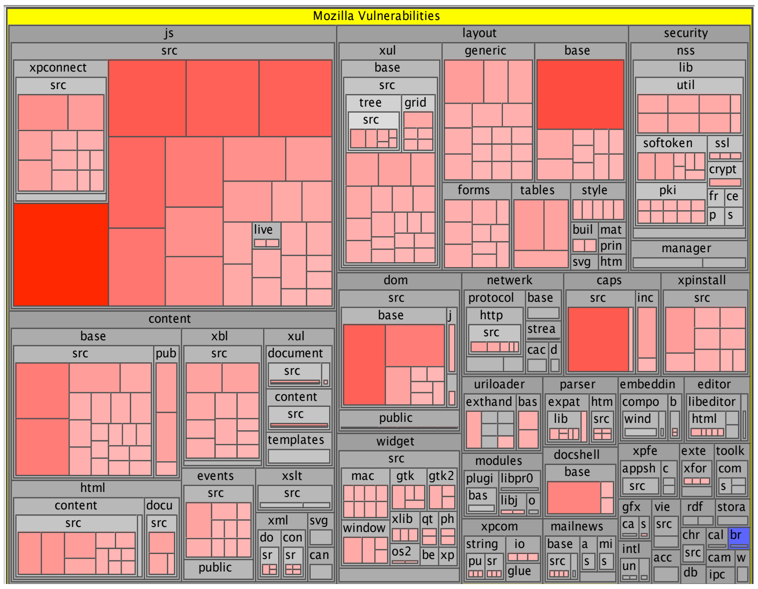


Figure 6.13: A new view of the treemap presented in Figure 6.2. A component's area is no longer proportional to its size, but rather to the number of its associated vulnerability reports. What is apparent is that large rectangles tend to appear together in one directory.

- We can use the character of fixes as predictors for clusters and singletons: if we find ourselves fixing the implementation of components, we might conjecture that this component is part of a cluster, so we might think of reimplementing the entire directory or package in a more secure way.

First, we present a slightly different view of the Mozilla treemap shown in Figure 6.2. In this new view, the rectangles have an area that is not proportional to the component's size, but to the number of vulnerabilities; see Figure 6.13. What is apparent from the figure is that components in clusters have more vulnerabilities than components in singletons.

See Section 6.8 for ways to test these claims rigorously.

6.6.1 Clusters

Let us take a closer look at the largest cluster, *js/src*, which is responsible for the execution of JavaScript scripts. This directory contains components that have 194 vulnerability-related bug reports associated with them, out of a total of 857 reports, which accounts for 22 percent. (Note that the reports need not be unique; a report may be associated with two or more components.)

First of all, clusters would correlate with imports because typically, components that are defined in a cluster will import similar imports. This is because the components in a cluster implement similar functionality (such as JavaScript support) and will hence tend to import those header files that declare that functionality.

A look at the vulnerability reports for components in that directory reveal that apparently JavaScript is highly insecure, both because the specification is unclear and because the implementation is insecure. For example, a typical vulnerability report that affects components in *js/src* (MFSA 2006-71) reads,

> Steven Michaud reported a crash in LiveConnect, the bridge code that allows Java applets and web JavaScript to communicate. The crash is due to re-use of an already-freed object and we presume this could be exploited with enough effort.

This necessitated three changes in *mozilla/js/src/liveconnect/jsj_JavaObject.c* (bug #352064) and a look at the changes shows that the implementation of *jsj_GC_callback* was changed to make sure that an object that is no longer used is properly deleted. In other words, it is the *implementation* of the component's functionality that was changed.

Looking at other security-repated changes from this directory reveals a similar picture. Therefore the following conjecture seems justified:

Vulnerability-related changes that appear in clusters
tend to change the implementation of functionality.
They fix insecure code.

In other words, once we have identified a cluster, we should think of reimplementing the functionality in that cluster in a more secure way, probably from the ground up.

6.6.2 Singletons

Consider a singleton like *intl/unicharutil/util/nsUnicharUtils*, which has only one vulnerability-related bug report associated with it (MFSA 2006-55), which says, in part,

> As part of the Firefox 1.5.0.5 stability and security release, developers in the Mozilla community looked for and fixed several crash bugs to improve the stability of Mozilla clients. Some of these crashes showed evidence of memory corruption that we presume could be exploited to run arbitrary code with enough effort.

Looking at the bug report (#284219), we find

> There's a fair amount of code out there that does something equivalent to:
>
> ```
> nsCString s;
> s.SetLength(n);
> char *p = s.BeginWriting();
> memcpy(p, ... );
> ```
>
> Any of these could be potential buffer overruns if an attacker can set *n* sufficiently large to trigger an out of memory condition. That would then result in *p* pointing at the static empty buffer, which lives at a fixed known address, and could then be exploited by an attacker :(

> I think we need to ensure that consumers like this verify the success of *SetLength* by testing *Length()* afterwards. This is similar to the *ReplacePrep* problem that was recently fixed.

The attacker-controlled value of n could lead to p pointing to a known address, with the ensuing *memcpy* overwriting memory at a known location. The suggested fix here is to (a) have *SetLength* return a boolean to indicate whether the operation has succeeded (i.e., if there was enough memory to resize the string), and (b) adding checks to make sure that the string is only accessed when *SetLength* returns *true*. This amounts to changing the above code snippet to:

```
nsCString s;
if (!s.SetLength(n))
  return NS_ERROR_OUT_OF_MEMORY;
char *p = s.BeginWriting();
memcpy(p, ... );
```

In comparison to the fix to bug #352064 above, we therefore find that it is not the *implementation* of the functionality that has changed. Rather, *interfaces* that are implemented in other parts of the program were difficult to use securely, which caused a change in these interfaces, which needed to be reflected everywhere that the interface was *used*.

Also, singletons would be correlated with imports too because all components that use the hard-to-use functionality (such as *nsCString*) would import the same imports.

Looking at other security-repated changes from other singletons reveals a similar picture. Therefore the following conjecture seems justified:

Vulnerability-related changes that appear in singletons
tend to changes how other code is called.
They fix code that is hard to use securely.

In other words, once we have identified a singleton, we should think of reimplementing the called interfaces so that they are easier to use securely.

6.7 Related Work

Previous work in this area tried to reduce the number of vulnerabilities or their impact by one of the following methods:

Looking at components' histories. The Vulture tool was inspired by the pilot study by Schröter et al. [147], who first observed that imports correlate with failures. While Schröter et al. examined *general defects,* the present work focuses specifically on *vulnerabilities.* To our knowledge, this is the first work that specifically mines and leverages vulnerability databases to make predictions. Also, our correlation, precision and recall values are higher than theirs, which is why we believe that focusing on vulnerabilities instead of on bugs in general is worthwhile.

Evolution of defect numbers. Both Ozment at al. [130] as well as Li et al. [101] have studied how the numbers of defects and security issues evolve over time. Ozment et al. report a decrease in the rate at which new vulnerabilities are reported, while Li et al. report an increase. Neither of the two approaches allow mapping of vulnerabilities to components or prediction.

Estimating the number of vulnerabilities. Alhazmi et al. use the rate at which vulnerabilities are discovered to build models to predict the number of as yet undiscovered vulnerabilities [7]. They use their approach on entire systems, however, and not on source files. Also, in contrast to Vulture, their predictions depend on a model of *how* vulnerabilities are discovered.

Miller et al. build formulas that estimate the number of defects in software, even when testing reveals no flaws [116]. Their formulas incorporate random testing results, information about the input distribution, and prior assumptions about the probability of failure of the software. However, they do not take into account the software's history—their estimates would be unchanged no matter how large the history is.

Tofts et al. build simple dynamic models of security flaws by regarding security as a stochastic process [155], but they do not make specific predictions about vulnerable software components. Yin at al. [184] highlight the need for a framework for estimating the security risks in large software systems, but give neither an example implementation nor an evaluation.

Testing the binary. By this we mean subjecting the binary of the program in question to various forms of testing and analysis (and then reporting any security leaks to the vendor). This is often done with techniques like fuzz testing [115] and fault injection; see the book by Voas and McGraw [171].

Eric Rescorla argues that finding and patching security holes does not lead to an improvement in software quality [141]. But he is talking about finding security holes *by third-party outsiders in the finished product* and not about finding them *by in-house personnel during the development cycle*. Therefore, his conclusions do not contradict our belief that Vulture is a useful tool.

(Statically) examining the source. This usually happens with an eye towards *specific* vulnerabilities, such as buffer overflows. Approaches include linear programming [66], data-flow analysis [87], locating functions near a program's input [46], axiomatizing correct pointer usage and then checking against that axiomatization [64], exploiting semantic comments [96], computing and checking path conditions [151], symbolic pointer checking [146], or symbolic bounds checking [143].

As an aside, the hypothesis of DeCast et al. [46] that vulnerabilities occur more in functions that are close to a program's input is not supported by the present study. Many of Mozilla's vulnerable components, such as *nsGlobalWindow*, lie in the heart of the application.

Instead of describing the differences between these tools and ours in every case, we we briefly discuss ITS4, developed by Viega at al. [170], and representative of the many other static code scanners. Viega et al.'s requirement was to have a tool that is fast enough to be used as real-time feedback during the development process, and precise enough so that programmers would not ignore it. Since their approach is essentially pattern-based, it will have to be *manually* extended as new patterns emerge.

The person extending it will have to have a concept of the vulnerability before it can be condensed into a pattern. Vulture will probably not flag components that contain vulnerabilities that were unknown at training time, but it *will* flag components that contain vulnerabilities that have been fixed before but have no name.

Since ITS4 checks local properties, it will also be very difficult for it to find security-related defects that arise from the interaction between far-away components, that is, components that are connected through long chains of def-use relations. Also, ITS4, as it exists now, will be unable to adapt to programs that for some reason contain a number of pattern-violating but safe practices, because it completely ignores a component's history.

Another approach is to use model checking [34, 35]. In this approach, specific classes of vulnerabilities are formalized and the program model-checked for violations of these formalized properties. The advantage over other formal methods is that if a failure is detected, the model checker comes up with a concrete counter-example that can be used as a regression test case. This too is a useful tool, but like ITS4, it will have to be extended as new formalizations emerge. Some vulnerability types might not even be formalizable.

Vulture also contains a static scanner—it detects imports by parsing the source code in a very simple manner. However, Vulture's aim is not to declare that certain lines in a program might contain a buffer overflow, but rather to direct testing effort where it is most needed by giving a *probabilistic assessment* of the code's vulnerability.

Hardening source or runtime environment. By this we mean all measures that are taken to mitigate a program's ability to do damage. Hardening a program or the runtime environment is useful when software is already deployed. StackGuard is a method that is representative of the many tools that exist to lower a vulnerability's impact [41]. Others include mandatory access controls as found in AppArmor [42] or SELinux [121]. However, Vulture works on the other side of the deployment divide and tries to direct programmers and managers to pieces of code requiring their attention, in the hope that StackGuard and similar systems will not be needed.

6.8 Beyond Imports and SVMs

6.8.1 Learning Patterns

The most interesting way to use version archives and vulnerability databases would be to extend the work by Neamtiu et al. [122] and learn *vulnerability patterns* from the project's history. This is the topic of a Masters Thesis by Cathrin Weiß [180]. Consider a singleton like the one discussed above in Section 6.6. There, code that looked like Figure 6.14 (a) was to be replaced with code that looks like Figure 6.14 (b).

The only difference between parts (a) and (b) is that the unconditional call to *SetLength* has been guarded with an *if* statement. Now suppose that we look at that difference not in textual form, but as an *abstract syntax tree* (AST); see Figure 6.15. An AST is a representation of source code as a directed graph where internal nodes are operators—such as statements—and leaf nodes are their operands—such as variable or object references. ASTs differ from full parse trees by omitting all nodes and edges that do not affect the semantics of the program.

```
nsCString s;
s.SetLength(n);
char *p = s.BeginWriting();
memcpy(p, ... );
```

(a)

```
nsCString s;
if (!s.SetLength(n))
  return NS_ERROR_OUT_OF_MEMORY;
char *p = s.BeginWriting();
memcpy(p, ... );
```

(b)

Figure 6.14: Typical bug/fix pattern. Code that looks like (a) is buggy because of the unguarded call to *SetLength*. This code is replaced by the guarded version (b).

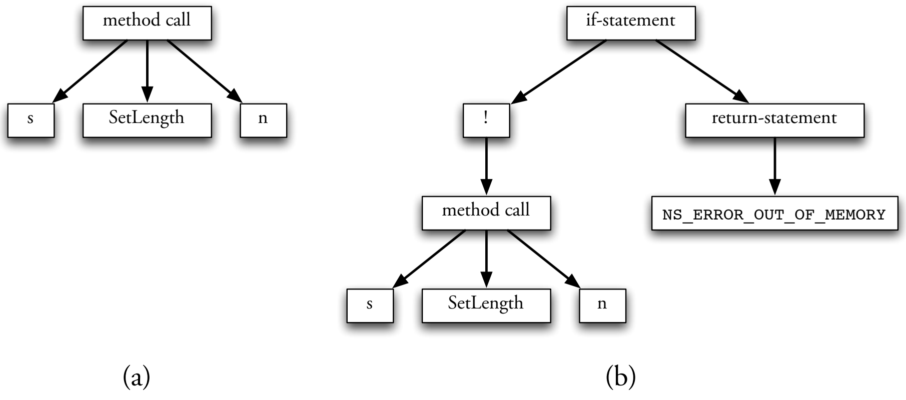


Figure 6.15: The fix from Figure 6.14 as an abstract syntax tree (AST). In this case, code is added to guard against the failure of the *SetLength* function.

Once we have identified fixes to vulnerabilities and once we have put them into AST form, we can *learn vulnerability patterns*. For example, from Figure 6.15, we can learn that an unguarded call to *SetLength* will need a guard. This pattern will have high support (number of instances where this pattern occurred) and high confidence (percentage of times when this pattern was changed). So whenever this pattern appears in code, we are fairly confident that this is a vulnerability that needs to be fixed. Since the fix is also present in AST form, we can even *suggest a fix*: whenever someone submits a check-in to the source code control system, we check if the vulnerability pattern appears and if so, we suggest the fix from the fix pattern.

More abstractly, let V be the code that is deleted from the vulnerable version and let F be the code that is added to the fixed version. We call (V, F) a *vulnerability/fix pattern*, or *pattern* for short. A fix that only deletes code would have $F = \emptyset$. In this case, a recommendation would be to delete V from the source code whenever we find an instance of it. The case of adding code is more complicated, however.

Let us consider the fix in Figure 6.14 again. It consists of adding code, so we would have $V = \emptyset$. Now the pattern V appears trivially at every place in the AST, so it is not apparent *where* to add code. Fortunately in our case, it is relatively easy to identify the place where the fix was added, namely at an unguarded call to *SetLength*. In general, code additions do not appear out of thin air, and added code appears in *context*, which in the case of ASTs would mean nodes in the vicinity of (above, below, or beside) the added code. So we would have to compute those contexts and use the one with the highest support and the highest confidence for a particular fix.

From AST representations we can learn vulnerability patterns.
In some cases, we may even be able to suggest fixes.

6.8.2 Other Future Work

More future work will work on the following topics:

More on Clusters and Singletons. In Section 6.6, we made a case why clusters and singletons should be correlated with imports, but we have not provided solid empirical evidence for that. In order to do that, we would have to proceed along the following lines, after defining precisely what clusters and singletons are:

1. In each cluster, find which combination of includes correlates most with vulnerabilities.

2. See if the include combinations overlap between clusters. The idea is that clusters correlate with imports because components in clusters tend to import imports containing the declaration of implemented services and especially of declarations that declare functionality that is internal to the component. If that idea is correct, there should be no significant overlap between relevant import combinations in different clusters.

3. The idea is that for every fix inside a cluster that changes some interface, there ought to be a number of singletons that occur because they need to adapt to the new interface. If that is true, it would mean that singletons would tend to be created in batches, and that they are correlated to changes in clusters.

The discrimination between clusters and singletons should also increase the precision and recall of SVM-based or conditional probability-based methods.

Fine-grained approaches. Rather than just examining imports at the component level, one may go for more fine-grained approaches, such as caller-callee relationships. Such fine-grained relationships may also allow vulnerability predictions for individual classes or even individual methods or functions.

In this case, the rows and columns of the import matrix contain the different method names as they are extracted form the source code by a simple parsing method; an entry x_{ik} would be 1 if method i calls method k; alternatively, x_{ik} could be the number of times the source code of method i contains calls to method k.

Inheritance and dynamic dispatch will make it generally impossible to compute exactly which method of which class is being called. Also we will not distinguish between similarly-named methods in different classes that are not in the same class hierarchy. Initially, the rows of X might even still contain components instead of methods.

Evolved components. This work primarily applies to predicting vulnerabilities of new components. However, components that already are used in production code come with their own vulnerability history. To test whether our approach also works in this condition, the same thesis will apply the approach in a simulation of the development process. To do this, we compute a predictor

Bayesian Filters. The same work will work on improving the *explanatory power* of prediction. It is certainly nice to know that the SVM can with high accuracy predict whether a module will be vulnerable or not, but it would be even nicer to know why this is so. We will therefore try to use Bayesian filters instead of SVMs. Bayesian filters are used in spam filters where words appear in email messges that are classified as either "spam" (undesirable) or "ham" (desirable). From this, the classifier computes the conditional probability that a message is spam or ham under the condition that thie word is present. The words with the highest percentages then serve as an explanation why the message is spam or ham: "p0rn" very probably means spam, whereas "Thomas" very probably means ham. Using such filters for vulnerability prediction would have the advantage that results would be much easier to explain and understand: "If *d2i_X509_SIG* appears in a component, it has a vulnerability with probability 95%".

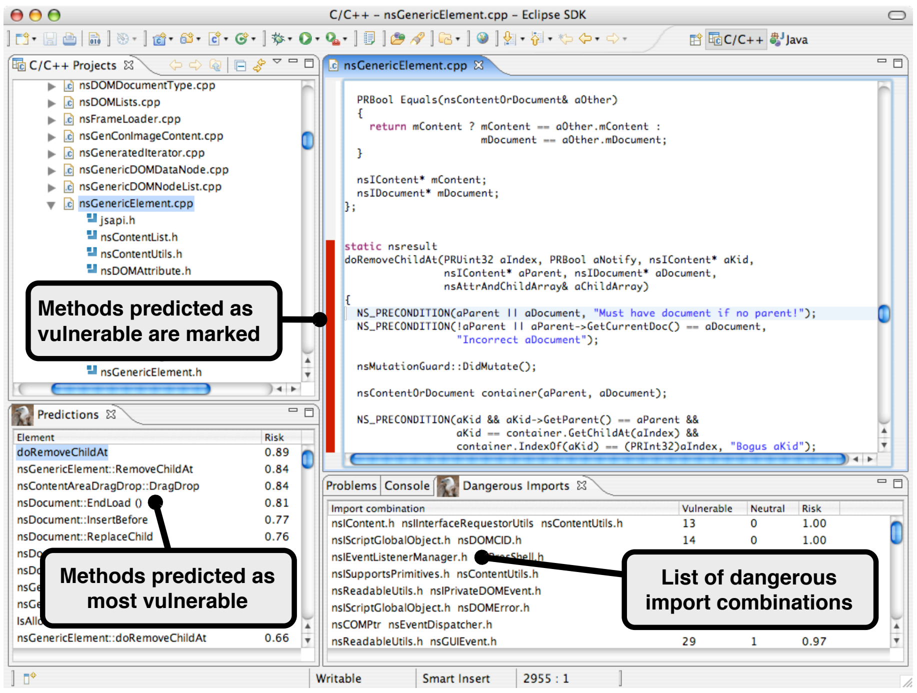
Characterizing domains. We have seen that empirically, imports are good predictors for vulnerabilities. We believe that this is so because imports characterize a component's domain, that is, the type of service that it uses or implements, and it is really the domain that determines a component's vulnerability. We plan to test this hypothesis by studies across multiple systems in similar domains.

More significant features. Besides imports, there may be further characteristics that make a component vulnerable. We plan to examine further features, such as complexity metrics, or the usage of specific data types; and to check whether such features would be applicable to predict vulnerabilities.

Usability. Right now, Vulture is essentially a batch program producing a textual output that can be processed by spreadsheet programs or statistical packages. We plan to integrate Vulture into current development environments, allowing programmers to query for vulnerable components. Such environments could also visualize vulnerabilities by placing indicators right next to the individual entities (Figure 6.16).

In a recent blog [145], Bruce Schneier wrote "If the IT products we purchased were secure out of the box, we wouldn't have to spend billions every year making them secure." One first step to improve security is to learn where and why current software had flaws in the past. Our approach provides the essential ground data for this purpose, and allows for effective predictions where software should be secured in the future.

Another area where our approach might be useful is the emerging area of self-diagnosing and self-repairing system. A tool like Vulture that can successfully predict profitable attack targets could select components to be put under a special protective umbrella such as a proxy that scans requests, canonicalizes them, and weeds out potentially harmful ones before it lets them through.



```
  PRBool Equals(nsContentOrDocument& aOther)
  {
    return mContent ? mContent == aOther.mContent :
                      mDocument == aOther.mDocument;
  }

  nsIContent* mContent;
  nsIDocument* mDocument;
};


static nsresult
doRemoveChildAt(PRUint32 aIndex, PRBool aNotify, nsIContent* aKid,
                nsIContent* aParent, nsIDocument* aDocument,
                nsAttrAndChildArray& aChildArray)
{
  NS_PRECONDITION(aParent || aDocument, "Must have document if no parent!");
  NS_PRECONDITION(!aParent || aParent->GetCurrentDoc() == aDocument,
                  "Incorrect aDocument");

  nsMutationGuard::DidMutate();

  nsContentOrDocument container(aParent, aDocument);

  NS_PRECONDITION(aKid && aKid->GetParent() == aParent &&
                  aKid == container.GetChildAt(aIndex) &&
                  container.IndexOf(aKid) == (PRInt32)aIndex, "Bogus aKid");
```

Element	Risk
doRemoveChildAt	0.89
nsGenericElement::RemoveChildAt	0.84
nsContentAreaDragDrop::DragDrop	0.84
nsDocument::EndLoad ()	0.81
nsDocument::InsertBefore	0.77
nsDocument::ReplaceChild	0.76
nsDo	
nsDo	
nsGe	
nsGe	
IsAllo	
nsGenericElement::doRemoveChildAt	0.66

Import combination	Vulnerable	Neutral	Risk
nsIContent.h nsIInterfaceRequestorUtils nsContentUtils.h	13	0	1.00
nsIScriptGlobalObject.h nsDOMCID.h	14	0	1.00
nsIEventListenerManager.h nsIPresShell.h			
nsISupportsPrimitives.h nsContentUtils.h			
nsReadableUtils.h nsIPrivateDOMEvent.h			
nsIScriptGlobalObject.h nsDOMError.h			
nsCOMPtr nsEventDispatcher.h			
nsReadableUtils.h nsGUIEvent.h	29	1	0.97

Figure 6.16: Sketch of a Vulture integration into Eclipse. Vulture annotates methods predicted as vulnerable with red bars. The view *"Predictions"* lists the methods predicted as most vulnerable. With the view *"Dangerous Imports"*, a developer can explore import combinations that lead to past vulnerabilities.

Chapter 7

Conclusions

The preceding chapters have presented methods that address software security concerns both before and after the deployment of a software system. Vulture uses statistics and machine learning to predict which components are most likely to contain vulnerabilities and Malfor uses experiments to decide which parts of a system—processes or inputs—are relevant for the intrusion.

Malfor is employed *after* software has been deployed. It makes the following contributions:

1. An automatic technique to find the processes that are relevant for an intrusion, or in fact any effect whose presence can be reliably tested.

2. An automatic technique to generate intrusion signatures for single day-zero attacks.

Vulture is used *before* deployment. It makes the following contributions:

1. A technique for mapping past vulnerabilities by mining and combining vulnerability databases with version archives.

2. Empirical evidence that contradicts popular wisdom saying that vulnerable components will generally have more vulnerabilities in the future.

3. Evidence that imports correlate with vulnerabilities.

4. A tool that learns from the locations of past vulnerabilities to predict future ones with reasonable accuracy.

5. An approach for identifying vulnerabilities that automatically adapts to specific projects and products.

6. A predictor for vulnerabilities that needs only the set of imports, and thus can be applied before the component is fully implemented.

Additionally, both the work on pre-deployment vulnerability prediction and post-deployment incident analysis resulted not only in studies, but also in tools. This shows that the ideas in this thesis are well suited to implementation.

Beyond these, there is one contribution that is more fundamental, namely the introduction of experimental and empirical methods to the area of software security.

This is arguably the first work to use experimental and empirical methods paradigmatically to solve problems of software security. There has been some recent, later, work that uses experiments, such as Packet Vaccine [176], but from the paper it is apparent that the authors are not aware of the power of experimental techniques: they use experiments only as one step in their procedure to find the relevant parts of an attack signature. Malfor, and the work that precedes it, solves the more general problem of finding the cause for *any* effect.

Similarly, there have been phenomenological studies on software security (see Section 6.7 for more details), but to our knowledge, no publication used the results from their empirical findings to predict useful things about security properties.[1]

We believe that it is important to introduce methods from the natural sciences to software security because software systems are getting so large that they begin to resemble natural phenomena more than man-made artifacts, and it is increasingly difficult to deduce their behaviour from first principles such as operative semantics. That is especially true for systems that have been developed without formal specifications. Such systems form the vast majority of deployed systems today. Here is a small list of such systems:

- Operating Systems: Microsoft Windows[2], Linux, BSD in all its variations, Mac OS, Unix.
- Office software: Microsoft Office, OpenOffice, KOffice.
- Web servers: Apache, Microsoft IIS.
- Internet clients: Internet Explorer, Outlook, Firefox, Thunderbird.
- Business software: SAP R3, Lexware Financial Office.

If we add the myriad lines of VBA script code that are deployed on production systems, this should account for over 99% of CPU cycles on the planet.

Also, systems are increasingly becoming interconnected and it is a well-established fact that the composition of systems will usually make the resulting system less secure. Ross Anderson gives the example shown in Figure 7.1 [9, Section 7.5.1]: Two high-value (secret) inputs are exclusive-ored, the result is exclusive-ored with a one-time pad and the resulting stream output on a low-value (non-secret) channel. The pad is output on a high-value channel. This device, when used in isolation, is provably secure, meaning that it is not possible to guess the values of any bit of one of the two high-value inputs knowing only the low-value output with a probability of more than $1/2$. (Proof sketch: every bit of L has the form $h_1 \oplus h_2 \oplus r$ and therefore has an equal probability of being either 0 or 1 because r is random and because of the properties of exclusive-or.) On the other hand, if we manage to feed the pad back to one of the inputs, the other input suddenly appears on the non-secret channel. (Proof sketch: $h_1 \oplus r \oplus r = h_1$.) This shows that secure systems need not be composable into systems that are also secure, a point also made by Peter G. Neumann [127].

The increase in size and the breakdown of security under system composition mean that traditional methods in software security such as static checking will quickly become less and less useful: large software systems will make checking more expensive and the non-composability of secure systems means that the whole system will

[1]Network security is a different matter: due to the large amount of traffic, statistical and machine-learning methods are routinely used to make predictions.

[2]With the possible exception of drivers, which are supposed to be model-checked during certification.

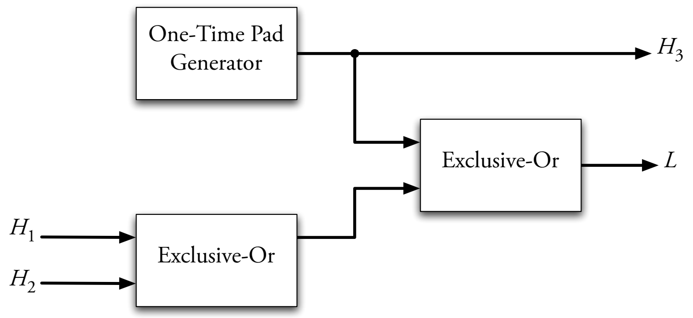


Figure 7.1: This device accepts two high-value (secret) channels on H_1 and H_2, xors them, xors the result with a one-time pad and outputs the result on the low-value (non-secret) channel L and the pad on the high-value channel H_3. In isolation, it is provably secure, but if H_3 can be fed back to H_2, the previously secret input H_1 now appears on the non-secret channel L. This shows that systems that consist of provably secure components need not be secure when composed into larger systems.

have to be checked, not just its individual componens. It is like trying to predict the temperature, volume and pressure of a gas knowing only the masses, positions and velocities of its constituent molecules.

Still, most research in software security overemphasizes formal methods and first principles, a point that was excellently made by Peter Wegner and Dina Goldin in their essay "Principles of Problem Solving" [179]. I believe on the contrary that we need more research in empirical methods.

As this thesis has shown, empirical and experimental methods can help where more fundamental methods must fail because of the complexity of the analyzed systems. It is my hope that these methods are taken up and developed further, maybe to their logical conclusion of auto-diagnosing and perhaps even auto-repairing systems.

Appendix A

Philosophy

There is in software security a gap between theorists and practitioners. Theorists argue that most problems with information systems would disappear if we just specified them properly beforehand and used formal methods and verification to make sure that the specified systems are defect-free. Practitioners argue that most systems today do not have specifications and cannot retroactively be equipped with them without jeopardizing that software's ability to generate revenue; see Chapter 1. Besides, they say, even if a system specification is "correct" (whatever *that* means), and even if an implementation can be rigorously proved to adhere to the specification, this does not mean that the system is proof against practical attacks. For example, an encryption system can have perfect secrecy and still be vulnerable to a timing attack. This section explores the origins of that gap.

If we reduce the quarrel to its essentials, we find on the one hand the opinion that *reasoning about computer programs is all that is needed to produce correct programs*, and on the other hand the belief that *only observation and theorizing will allow us to analyze and predict the behaviour of complex software systems*. If we generalize away the references to computer science, we arrive at these two beliefs:

- Knowledge is gained only by reasoning; and
- Knowledge is gained only by observation.

We call the first belief *rationalist* and the second *empiricist*. People are accordingly called *rationalists* or *empiricists*, respectively, depending on their beliefs.

Both beliefs are incompatible in the extreme form in which they were stated above, but there is no reason why they should not both be partially true. For example, once we find out that $F = ma$ by experiment, we could deduce $a = F/m$.

In order to find out about the origins of the incompatibility between rationalists and empiricists, we must look into the history of both rationalist and empiricist philosophy.

A.1 Rationalist Philosophy

An early rationalist was Pythagoras (582? BC–507? BC). It is difficult to say exactly what he taught, his school being a mix of mathematics and mysticism. One of his views was that the whole world was based on integers, which led, for example, to the pythagorean tuning of musical scales, which later had to be modified by the meantone tuning and still later by the well-tempered tuning in order to use all twenty-four major scales. When a Pythagorean (usually believed to be Hippasus of Metapontum) discovered that the side and diagonal of a square were not in integer proportions, and therefore invalidated a central tenet of Pythagorean philosophy, his colleagues acted in the time-honored manner of fundamentalists everywhere and killed him, throwing him overboard during a cruise. [65]

The origins of what we understand today by rationalism lie with the idealism of Plato (427? BC–347? BC), who was in turn influenced by Pythagoras. In his *State*, Plato believed that the world we perceive is merely a shadow image of the real world of ideas [134]. This real world contains, for example, the *idea* of a table, which encompasses the table-ness of all tables. Using software engineering terms, the idea is a class and the shadow that we see is an instance of that class. Only philosophers get glimpses of that ideal world and were therefore the first object-oriented system analysts.[1]

Skipping many centuries, we arrive at René Descartes (1596–1650). His main contribution to philosophy was the notion of solipsism, which holds that all our knowledge of the world comes to us through our senses and that this outside world is therefore not necessarily real. The same argument is made today when critics of electronic signatures claim that no-one knows what they are really signing because there is in general no trusted path from the memory (which holds the document to be signed) to the display (which displays to users what they are supposed to be signing). Solipsism also appears in Chapter 5 as the name of the capture/replay subsystem of Malfor. Following Descartes's logic, the only thing we can be sure of is our own existence, hence "I think therefore I am". Descartes is certainly one of the more radical rationalists, entirely discarding perception as a means of obtaining knowledge and admitting only deduction. Still, he later turns around somewhat by proving that God exists (even though he proves it only to his own satisfaction) and then deducing that he can trust his senses after all because God would not want to deceive him. This contorted argument is necessary because it is otherwise extremely difficult to explain certain things, such as the (apparent) existence of natural laws, a flaw noted by Richard Feynman [61].

Next in line is Gottfried Wilhelm Leibniz (1646–1716). For our purposes, his main contribution to rationalist philosophy is his *Calculus Ratiocinator*. This is a method in which problems are solved by writing them down in a specialized language and then applying the rules of his calculus. This calculus was not only for solving technical problems; Leibniz believed he could also apply the method to matters of opinion and thereby solve any dispute in a completely rational way, by calculating what the right answer must be.[2] This optimistic view echoes both Charles Babbage's nineteenth-century analytical engine as well as David Hilbert's motto „Wir müssen wissen und wir werden wissen!“ ("We must know and we will know!").

Leibniz explicitly believed that the negation of a true statement must be false. Implicitly, Leibniz must also have believed that every true statement could be proved, something that Bertrand Russell and Alfred North Whitehead also believed, and

[1]Of course, Plato was also a power-hungry politician who argued that since only philosophers get glimpses of the real world, an ideal state should be run by philosophers. The term "conflict of interests" was coined only much later..

[2] For example, to decide whether Britney Spears or Christina Aguilera was the better singer, we could write down some formulas which we *know* will differentiate good singers from bad ones, calculate and compare the results. All teen magazines would instantly vanish from the newsstands.

which was finally refuted by Kurt Gödel in his 1931 paper „Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I" [70], and again later by Alan M. Turing in his "On computable numbers with an application to the Entscheidungsproblem" [159].

A.2 Empiricist Philosophy

Returning to the empiricists, one the first philosophers usually credited with that label is Aristotle (384 BC–322 BC). He could not have been more different from Plato. Recall that Plato said that the real world was the world of ideas and that the material world was separate from the ideal world and related to it only as a shadow to the object which casts it. In contrast, Aristotle believed that universal principles manifest in particular things. Consequently, philosophy should work from studying particular things and from that study find their universal quality, or essence. Basically, Aristotle's method was inductive and Plato's method was deductive. But we should not believe that Aristotle used an empirical method. Rather, he had his head in the clouds just like Plato. For example, he asserts in his theory of gravity that things fall down because they come from the earth and therefore have a natural tendency to return to the earth from which they came, something that a truly empirical method would easily refute.

Again skipping many centuries (and some important philosophers), we come to Isaac Newton (1643 [1642 OS]-1727). Newton was not a philosopher *per se*, but his work was influential in shaping the thoughts of many, such as David Hume; see below. While certainly using his powers of deduction to discover natural laws, Newton was also a keen observer. He borrowed from Francis Bacon the notion of the crucial experiment or *experimentum crucis*, which is an experiment that has the power to falsify a theory. For example, the Michelson-Morley experiment was a crucial experiment that falsified the theory of the electromagnetic ether. Newton used many crucial experiments in his works. The idea of crucial experiments was later expanded by Karl Popper; see below.

Newton influenced David Hume (1711–1776). In his *Treatise of Human Nature* [82], written when Hume was only twenty-six, he lays out his belief that *all* our knowledge comes to us *only* through our senses. He was a radical empiricist. In the *Treatise*, he also expounds his theory of causation, something dear to all computer scientists wishing to find bugs in their programs. For Hume, causation is not a strict if-then relationship between events; rather, causation happens when we *believe*, based on many previous observations, that two events are causally connected. This makes causation a statistical thing and in effect removes from the very concept many features which enable us to reason about causes and effects. This is already hard to swallow, but Hume goes even further and argues that even Truth is not absolute, but instead relies on convention.

It is interesting to note that many important rationalists were also excellent mathematicians (with the exception of Plato), and that almost none of the empiricists were (with the significant exception of Newton).

A.3 Modern Philosophy of Science

The first attempt to reconcile the rationalist and empiricist world-views was made by Immanuel Kant (1724–1804). Recall Hume's formulation of causation which in his view is entirely subjective. How then to explain the existence of natural laws? Kant solved this metaphysical problem with a neat trick.

In his *Kritik der Reinen Vernunft* [88], Kant invented something called *synthetic a priori truths*. Basically, he argues that all natural laws are self-evident, yet cannot be derived by logic alone. This argument unmasks Kant as a rationalist at heart. But the argument also turns around in circles, because he cannot offer any evidence for the existence of synthetic *a priori* truths other than otherwise, natural laws would be difficult to explain. Obviously, Kant was under the impression that there is only one possible universe, ours, and that its natural laws are not changing with time. Both assertions have been challenged. Also, if natural laws are self-evident, it is somewhat difficult to explain why it sometimes takes so long to find them, or why they have to be modified in the face of contradicting evidence.

The next (and last) philosopher on our list is Karl Popper (1902–1994). Popper wasn't so much interested in how we gain knowledge (the central problem of epistemology), but rather *how scientific progress happens*. This is a related yet different problem. Popper argued that scientific progress happened through the formulation of *scientific theories* which are then tested by *experiments* [135, 136]. Some experiments would contradict, or *falsify*, a theory, which must lead to its abandonment and to the search for a better theory that explains the experiment.

What makes a good scientific theory? A good theory will make specific predictions. For example, the (Newtonian) theory of gravity makes the prediction that two bodies are attracted with a force proportional to the product of their masses and inversely proportional to the square of their distance. This prediction invites experiments, experiments that would falsify the entire theory if they failed. For example, we could try to see whether it also holds at very large distances, at high relative speeds, for very light objects or for very heavy objects and so on. Such a risky experiment is called a *crucial experiment* or *experimentum crucis*.

Psychoanalysis on the other hand makes no predictions. Every form of human behaviour is permitted by it. So the theory of psychoanalysis is not a scientific theory. It doesn't mean that it is bad medicine, though; it's just nor science.

Falsifying theories leads to better theories. Does this process ever stop? Will we ever get to the ultimate form of natural laws? Do natural laws exist at all? Popper doesn't say (even though he was of the opinion that natural laws did exist), and it's also not part of his methodology, which sets it apart from metaphysics.

It is also interesting to note that according to Popper, knowledge does *not* appear by inductive reasoning, that is, by finding a general law from specific events. So, how do we get theories? According to Popper, we get them by guessing hypotheses, not by finding natural laws by induction.

Popper's view is mostly accepted in the scientific community, but there are critics. The most notable critic is Paul Feyerabend, a former pupil of Popper's, who says that scientific progress happens more by accident than by design, and whose motto is "Anything Goes" [60].

Notation

Notation	Definition	Meaning
ancestors(p)	5.2	Ancestors of a process
C	5.3	Captured processes
culp S	3.2	Culprit set
ddmin	3.6	Delta Debugging algorithm
✘	3.1	Failing test case
C_0	5.3	Initial set of captured processes
$f^k(x)$	5.1	Iteration
test(S)	3.1	Outcome of a test function
offspring(p)	5.2	Offspring set
parent(p)	5.2	Parent process ID
✔	3.1	Passing test case
$\mathcal{P}(S)$	3.1	Power set
P	5.2	Process IDs
p_{root}	5.2	Root process
R	5.4	Replay function
?	3.1	Unresolved test case

Glossary

abstract syntax tree A directed tree in which the internal nodes are operators and the leaf nodes are the operands. Related to the parse tree, but much simpler.

antecedent The precondition A in a conditional "if A then C".

AST $\rightarrow$ abstract syntax tree.

box-and-whisker plot A plot that shows many characteristics of an empirical distribution. The central element is a box divided by a line, usually drawn vertically, joined at the top and bottom by graphical elements called whiskers. The box starts at the lower quartile (i.e., 1/4 of all observed values were less than or equal to this value) and extends to the upper quartile (which 3/4 of all observed values were less than or equal to). The line dividing the box is the median (the second quartile). The height of the box is the inter-quartile range, or IQR. The lower whisker is a datapoint that is at most $1.5 \cdot \mathrm{IQR}$ away from the lower quartile, and similarly for the upper whisker. Points beyond the whiskers are called outliers.

cluster $\rightarrow$ vulnerability cluster.

consequent The postcondition C in a conditional "if A then C".

cross validation A method of assessing a machine-learning algorithm. Works by partitioning a data set into n (usually equal) parts and then using $n - 1$ parts to train the algorithm and then predicting for the remaining part.

ecdf $\rightarrow$ empirical cumulative distribution function.

empirical cumulative distribution function Let $(x_1, \ldots, x_n)$ be a sequence of real numbers. The empirical cumulative distribution function is then defined as $\mathrm{ecdf}(x) = |\{k : x_k < x\}|/n$. Typically, the x_k are observations of some sort.

inter-quartile range $\rightarrow$ box-and-whisker plot.

IQR $\rightarrow$ box-and-whisker plot.

Lamport clock $\rightarrow$ time vector.

leave-one-out cross validation A special form of $rightarrow$ cross validation where the number of partitions is equal to the number of elements. In other words, if the data set has m elements, the cross validation proceeds with $m = n$.

lower whisker $\rightarrow$ box-and-whisker plot.

nop sled A string of instructions that have no side effects. This is used by buffer overflow attacks: by prepending a nop sled to the attack string, the attacker does not need to know the memory layout to the byte.

outlier $\rightarrow$ box-and-whisker plot.

singleton $\rightarrow$ vulnerability singleton.

time vector A way to compute causality in distributed systems. Works by having every process maintain a vector of event numbers in other processes that could affect the events generated by this process.

UML $\rightarrow$ User Mode Linux.

upper whisker $\rightarrow$ box-and-whisker plot.

User Mode Linux An early form of operating system virtualization. The operating system is run as an ordinary user-mode process. In this way, several operaying systems can be run in parallel on one machine. Now superseded by Xen.

vector time A way of keeping time in a distributed computation. $\rightarrow$ time vector.

vulnerability/fix pattern A pattern that is mined from the change history of a software project and that characterizes a vulnerability and its fix. For example, a call to *gets*() is replaced by a call to *fgets*().

vulnerability cluster Set of files that have been fixed because of vulnerabilities and that appear together in one directory.

vulnerability singleton A file that has been fixed because of vulnerabilities but where only few other files in the same directory are similarly vulnerable.

Bibliography

[1] Anurag Acharya and Mandar Raje. MAPbox: Using parameterized behavior classes to confine untrusted applications. In *Proceedings of the 9th USENIX Usenix Security Symposium* [161]. See page 38.

[2] Active@ Data Recovery Software. Active@ Disk Image. `http://www.disk-image.net/`, May 2007. See page 10.

[3] ActMon Software. ActMon Password Recovery XP. `http://www.actmon.com/password%2Drecovery/`, February 2006. See page 10.

[4] Forman S. Acton. *Numerical Methods That (Usually) Work*. Mathematical Association of America, Washington, D.C., USA, 1990. See page 90.

[5] Frank Adelstein. Live forensics: Diagnosing your system without killing it first. *Communications of the ACM*, 49(2):63–66, February 2006. See page 3.

[6] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proceedings of the 20th International Conference on Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, September 1994. See page 83.

[7] Omar Alhazmi, Yashwant Malaiya, and Indrajit Ray. *Security Vulnerabilities in Software Systems: A Quantitative Perspective*, volume 3645/2005 of *Lecture Notes in Computer Science*, pages 281–294. Springer Verlag, Berlin, Heidelberg, August 2005. See page 100.

[8] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting targeted attacks using shadow honeypots. In *Proceedings of the 14th Usenix Security Symposium*, pages 129–144, Berkeley, CA, USA, August 2005. Usenix Association, Usenix Association. See page 13.

[9] Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley and Sons, Inc., 2001. See page 108.

[10] Ross Anderson and Tyler Moore. Information security economics — and beyond. In *Advances in Cryptology - CRYPTO 2007: 27th Annual International Cryptology Conference*, Berlin, August 2007. Springer Verlag. See page 45.

[11] Javed Aslam, Sergey Bratus, David Kotz, Ron Peterson, Daniela Rus, and Brett Tofel. The Kerf toolkit for intrusion analysis. Technical Report TR2004-493, Dartmouth College, Department of Computer Science, and the Institute for Security Technology Studies, 2004. See page 10.

[12] Jay Aslam, David Kotz, and Daniela Rus. Kerf project outline. `https://kerf.cs.dartmouth.edu/outline.shtml`, February 2006. See page 10.

[13] Association for Computing Machinery. *Proceedings of the 12th ACM conference on Computer and Communications Security*, New York, NY, USA, 2005. ACM Press. See pages 122 and 126.

[14] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 290–301, New York, NY, USA, 1994. ACM Press. See page 12.

[15] Michael Backes and Birgit Pfitzmann. Limits of the cryptographic realization of Dolev-Yao-style XOR. In Sabrina De Capitani di Vimercati, Paul Syverson, and Dieter Gollmann, editors, *Proceedings of 10th European Symposium on Research in Computer Security, ESORICS'05*, number 3679 in Lecture Notes in Computer Science, pages 178–196, Berlin, September 2005. Springer Verlag. See page 14.

[16] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the 9th USENIX Usenix Security Symposium* [161]. See page 12.

[17] A. Baratloo, T. Tsai, and N. Singh. libsafe: Protecting critical elements of stacks. `http://www.avayalabs.com/project/libsafe/`. See page 12.

[18] Bradford L. Barrett. The Webalyzer: what is your web server doing today? `http://www.mrunix.net/webalizer/`, February 2006. See page 9.

[19] David Basin. Personal communication, March 2006. See page 2.

[20] Steven M. Bellovin. Re: Neal koblitz critiques modern cryptography. Message `<20070904231107.93191766105@berkshire.machshav.com>` on the cryptography mailing list, September 2007. See page 4.

[21] Dan Bernstein. Cache-timing attacks on AES. `http://cr.yp.to/papers.html#cachetiming`, 2004. See pages 2 and 14.

[22] Rahul Bhaskar. State and local law enforcement is not ready for a cyber katrina. *Communications of the ACM*, 49(2):81–83, February 2006. See page 3.

[23] Briggs Softworks. Directory snoop. `http://www.briggsoft.com/dsnoop.htm`, February 2006. See page 10.

[24] Herman ten Brugge. Gcc bounds checking. `http://web.inter.nl.net/hcc/Haj.Ten.Brugge/`. See page 12.

[25] David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the 12th Usenix Security Symposium* [163], pages 1–14. See pages 2 and 14.

[26] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* [83]. See page 11.

[27] John Burnet. *Early Greek Philosophy*. London, 1920. See page 39.

[28] Brian Carrier. Autopsy forensic browser manual page. `http://www.sleuthkit.org/autopsy/man/autopsy.html`, February 2006. See page 10.

[29] Brian Carrier. File activity timelines: Sleuth kit reference document. `http://www.sleuthkit.org/sleuthkit/docs/ref_timeline.html`, February 2006. See page 10.

[30] Brian Carrier. File system analysis techniques: Sleuth kit reference document. `http://www.sleuthkit.org/sleuthkit/docs/ref_fs.html`, February 2006. See page 10.

[31] Brian D. Carrier. Risks of live digital forensic analysis. *Communications of the ACM*, 49(2):56–61, February 2006. See page 3.

[32] Eoghan Casey. *Digital Evidence and Computer Crime: Forensic Science, Computers, and the Internet*. Academic Press/ElÂseÂvier, London, UK, Second edition, 2004. See page 3.

[33] Eoghan Casey. Investigating sophisticated security breaches. *Communications of the ACM*, 49(2):48–54, February 2006. See page 3.

[34] Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of C code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 171–185, Reston, VA, USA, February 2005. Internet Society, Internet Society. See page 101.

[35] Hao Chen and David Wagner. MOPS: An infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, pages 235–244, New York, NY, USA, November 2002. ACM Press. See page 101.

[36] The chkrootkit Team. chkrootkit. `http://www.chkrootkit.org/`, February 2006. See page 10.

[37] Grenier Christophe. CMOS password recovery tools. `http://www.cgsecurity.org/index.html?cmospwd.html`, May 2007. See page 10.

[38] The Common Digital Evidence Storage Format Working Group. Standardizing digital evidence storage. *Communications of the ACM*, 49(2):67–68, February 2006. See page 3.

[39] Computer Security Institute and Federal Bureau of Investigation. 2006 CSI/FBI computer crime survey. `i.cmpnet.com/gocsi/db_area/pdfs/fbi/FBI2006.pdf`, December 2006. See page 1.

[40] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. Formatguard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th Usenix Security Symposium* [162]. See page 12.

[41] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Usenix Security Symposium*, pages 63–78, Berkeley, CA, USA, January 1998. Usenix Association, Usenix Association. See pages 12 and 101.

[42] Crispin Cowan. Apparmor linux application security. `http://www.novell.com/linux/security/apparmor/`, January 2007. See page 101.

[43] Jedidiah R. Crandall, Zhendong Su, S. Felix Wu, and Frederic T. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proceedings of the 12th ACM conference on Computer and Communications Security* [13], pages 235–248. See page 11.

[44] The Cryptography Mailing List. GnuTLS (libgrypt really) and postfix. `http://www.archivesat.com/Cryptography/thread126448.htm`, 2006. See page 13.

[45] Davor Čubranić, Gail C. Murphy, Janice Singer, and Kellogg S. Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, June 2005. See page 78.

[46] Dan DaCosta, Christopher Dahn, Spiros Mancoridis, and Vassilis Prevelakis. Characterizing the security vulnerability likelihood of software functions. In *IEEE Proceedings of the 2003 International Conference on Software Maintenance (ICSM'03)*, September 2003. See page 100.

[47] Asit Dan, Ajay Mohindra, Rajiv Ramaswami, and Dinkar Sitaram. Chakravyuha (cv): A sandbox operating system environment for controlled execution of alien code. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, NY, USA, September 1997. See page 38.

[48] Laurent Destailleur. AWStats real-time logfile analyzer. `http://awstats.sourceforge.net/`, February 2006. See page 10.

[49] Hermann Diels. *Die Fragmente der Vorsokratiker*. Berlin, 1951. See page 39.

[50] Evgenia Dimitriadou, Kurt Hornik, Friedrich Leisch, David Meyer, and Andreas Weingessel. *e1071: Misc Functions of the Department of Statistics (e1071), TU Wien*, 2006. R package version 1.5-13. See pages 91 and 96.

[51] Daniel Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983. See page 4.

[52] Mark Dowd, John McDonald, and Justin Schuh. *The Art of Software Security Assessment*. Pearson Education, Inc., Boston, MA, USA, 2007. See page 2.

[53] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 211–224, New York, NY, USA, December 2002. ACM Press. See pages 3, 11, and 49.

[54] Dave Dykstra. bsed binary stream editor home page. `http://www1.bell-labs.com/project/wwexptools/bsed/`, February 2006. See page 9.

[55] Hiroaki Etoh. Gcc extension for protecting applications from stack-smashing attacks. `http://www.trl.ibm.com/projects/security/ssp/`, 2001. See page 12.

[56] David Evans. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 44–53, New York, NY, USA, 1996. ACM Press. See page 12.

[57] Dan Farmer. Frequently asked questions about the coroner's toolkit. `http://www.fish.com/tct/FAQ.html`, January 2005. See page 10.

[58] Dan Farmer and Vietse Venema. *Forensic Discovery*. Addison-Wesley, January 2005. See pages 3 and 10.

[59] Federal Bureau of Investigation. 2005 FBI computer crime survey. `http://www.fbi.gov/publications/ccs2005.pdf`, January 2006. See page 1.

[60] Paul Feyerabend. *Against Method*. Verso, third edition, 1993. See page 114.

[61] Richard P. Feynman. *Surely You're Joking, Mr. Feynman: Adventures of a Curious Character*. W. W. Norton & Co., New York, N.Y., USA, 1997. See page 112.

[62] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance (ICSM'03)*, Amsterdam, Netherlands, September 2003. IEEE. See page 78.

[63] Foundstone, Inc. BinText finds ASCII, Unicode and resource strings in a file. `http://www.foundstone.com/index.htm?subnav=resources/navigation.htm\&subcontent=/resources/proddesc/bintext.htm`, February 2006. See page 9.

[64] Pascal Fradet, Ronan Caugne, and Daniel Le Métayer. Static detection of pointer errors: An axiomatisation and a checking algorithm. In *European Symposium on Programming*, pages 125–140, 1996. See pages 12 and 100.

[65] Kurt von Fritz. The discovery of incommensurability by Hippasos of Metapontum. *Annals of Mathematics*, 46:242–264, 1954. See page 112.

[66] Vinod Ganapathy, Somesh Jha, David Chandler, David Melski, and David Vitek. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, New York, NY, USA, October 2003. ACM Press. See page 100.

[67] Simson L. Garfinkel. AFF: A new format for storing hard drive images. *Communications of the ACM*, 49(2):85–87, February 2006. See page 3.

[68] Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Proceedings of the 10th Internet Society Symposium on Network and Distributed System Security*, pages 19–34, Reston, VA, USA, February 2003. Internet Society, Internet Society. See page 38.

[69] A. K. Ghosh, T. O'Connor, and G. McGraw. An automated approach for identifying potential vulnerabilities in software. In *Proceedings of the 1999 IEEE Computer Society Symposium on Security and Privacy (SSP '89)*, pages 104–114, Washington, DC, USA, May 1998. IEEE Computer Society, IEEE Press. See page 12.

[70] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931. See page 113.

[71] Peter Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the 6th Usenix Security Symposium*, Berkeley, CA, USA, July 1996. Usenix Association, Usenix Association. See page 10.

[72] Peter Gutmann. *Cryptographic Security Architecture*. Springer Verlag, October 2003. See pages 13 and 14.

[73] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics. Springer Verlag, 2001. See pages 86 and 89.

[74] heise online. FBI: Computer-KriminalitÃ¤t kostet US-Firmen 67 Milliarden Dollar im Jahr. `http://www.heise.de/newsticker/meldung/mail/68593`, January 2006. See page 1.

[75] Ralf Hildebrandt and Andreas Zeller. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 26(2):183–200, February 2002. See pages 15, 16, 17, 18, 22, 23, 24, and 48.

[76] Christopher Hitchcock. Probabilistic causation. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, Fall 2007. `http://plato.stanford.edu/archives/fall2007/entries/causation-probabilistic/`. See page 8.

[77] Thomas Höfer, Hildegard Przyrembel, and Silvia Verleger. New evidence for the theory of the stork. *Paediatric and Perinatal Epidemiology*, 18(1):88–92, 2004. See page 8.

[78] Greg Hoglund and Gary McGraw. *Exploiting Software*. Addison-Wesley, 2004. See page 55.

[79] Christian Holler. Predicting vulnerable methods. Bachelor thesis, Department of Informatics, Saarland University, September 2007. to appear. See page 75.

[80] Chet Hosmer. Digital evidence bag. *Communications of the ACM*, 49(2):69–70, February 2006. See page 3.

[81] David Hume. *An Enquiry concerning Human Understanding*, volume XXXVII, Part 3 of *The Harvard Classics*. P.F. Collier & Son, New York, NY, USA, 1909–1914. See page 7.

[82] David Hume. *A Treatise of Human Nature*. Oxford University Press, 2000. See page 113.

[83] IEEE Computer Society. *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, Washington, DC, USA, May 2006. IEEE Press. See pages 121, 126, and 128.

[84] Internet Society. *Proceedings of the 13th Annual Network and Distributed System Security Symposium*, Reston, VA, USA, February 2006. Internet Society. See pages 126 and 128.

[85] Tim Johnson. Cache View for Firefox, Netscape, Mozilla, Opera and Internet Explorer. `http://www.progsoc.uts.edu.au/~timj/cv/`, February 2006. See page 9.

[86] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Automated and Algorithmic Debugging*, pages 13–26, 1997. See page 12.

[87] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2006. See page 100.

[88] Immanuel Kant. *Kritik der Reinen Vernunft*. Reclam, Ditzingen, 1986. See page 114.

[89] Hyang-Ah Kim and Brad Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th Usenix Security Symposium*, Berkeley, CA, USA, August 2004. Usenix Association, Usenix Association. See page 13.

[90] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 223–236, New York, NY, USA, 2003. ACM Press. See pages 3, 11, and 45.

[91] Calvin Ko, Timothy Fraser, Lee Badger, and Douglas Kilpatrick. Detecting and countering system intrusions using software wrappers. In *Proceedings of the 9th USENIX Usenix Security Symposium* [161]. See page 38.

[92] Alfred Korzybski. An aristotelian system and its necessity for rigour in mathematics and physics. In *Science and Sanity: an introduction to non-aristotelian systems and general semantics*, pages 747–761. International Non-aristotelian Library Publishing Co., Lakeville, Conneticut, USA, Dritte edition, 1948. See page 11.

[93] Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan Eren, Neel Mehta, and Riley Hassell. *The Shellcoder's Handbook*. John Wiley & Sons, 2004. See page 55.

[94] Nick Kurshev. BIEW is Binary vIEW project. `http://biew.sourceforge.net/en/biew.html`, February 2006. See page 9.

[95] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. See page 64.

[96] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th Usenix Security Symposium* [162], pages 177–190. See pages 12 and 100.

[97] LastBit Software. Password recovery and security. `http://lastbit.com/`, May 2007. See page 10.

[98] Dirk Leinenbach. Personal communication, August 2007. See page 1.

[99] David Lewis. Causality. *The Journal of Philosophy*, 70:557–567, 1973. See page 7.

[100] David Lewis. *Counterfactuals*. Blackwell, Oxford, 1973. See page 7.

[101] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now? An empirical study of bug characteristics in modern open source software. In *Proceedings of the Workshop on Architectural and System Support for Improving Software Dependability 2006*, pages 25–33. ACM Press, October 2006. See page 100.

[102] Zhichun Li, Manan Sanghi, Yan Chen, Ming-Yang Kao, and Brian Chavez. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* [83]. See page 11.

[103] Zhenkai Liang and R. Sekar. Fast and automated generation of attack signatures: a basis for building self-protecting servers. In *Proceedings of the 12th ACM conference on Computer and Communications Security* [13], pages 213–222. See page 11.

[104] Michael Locasto, Stelios Sidiroglou, and Angelos D. Keromytis. Software self-healing using collaborative application communities. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium* [84], pages 95–106. See page 13.

[105] Lostpassword.com. Passware password recovery kit. `http://www.lostpassword.com/kit.htm`, May 2007. See page 10.

[106] Steven Mai. Visualization of process replay. Bachelor thesis, Department of Informatics, Saarland University, June 2007. to appear. See page 51.

[107] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Efficient algorithms for discovering association rules. In *Knowledge Discovery in Databases: Papers from the 1994 AAAI Workshop*, pages 181–192, 1994. See page 83.

[108] Heiko Mantel. *A Uniform Framework for the Formal Specification and Verification of Information Flow Security*. PhD thesis, Saarland University, Postfach 15 11 50, 66041 Saarbrücken, Germany, July 2003. See page 50.

[109] Mares and Company, LLC. Copy_ads: copy and investigate alternate data streams. `http://www.dmares.com/maresware/ac.htm#COPYADS`, February 2006. See page 10.

[110] Mares and Company, LLC. Crckit. `http://www.dmares.com/maresware/ac.htm#CRCKIT`, February 2006. See page 10.

[111] Mares and Company, LLC. Diskimag. `http://www.dmares.com/maresware/df.htm#DISKIMAG`, February 2006. See page 10.

[112] Mares and Company, LLC. Eventlog. `http://www.dmares.com/maresware/df.htm#EVENTLOG`, February 2006. See page 10.

[113] Friedemann Mattern and Reinhard Schwarz. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994. See pages 63, 66, and 67.

[114] Peter Menzies. Counterfactual theories of causation. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, Spring 2003. `http://plato.stanford.edu/archives/spr2003/entries/causation-counterfactual/`. See pages 6 and 7.

[115] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the Association for Computing Machinery*, 33(12):32–44, 1990. See page 100.

[116] K.W. Miller, L.J. Morell, R.E. Noonan, S.K. Park, D.M. Nicol, B.W. Murrill, and M. Voas. Estimating the probability of failure when testing reveals no failures. *IEEE Transactions on Software Engineering*, 18(1):33–43, January 1992. See page 100.

[117] The Mozilla Foundation. Bugzilla. `http://www.bugzilla.org`, January 2007. See page 79.

[118] The Mozilla Foundation. Mozilla foundation security advisories. `http://www.mozilla.org/projects/security/known-vulnerabilities.html`, January 2007. See page 79.

[119] The Mozilla Foundation. Mozilla project website. `http://www.mozilla.org/`, January 2007. See page 90.

[120] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 27th International Conference on Software Engineering*, New York, NY, USA, May 2005. ACM Press. See page 80.

[121] National Security Agency. Security-Enhanced Linux homepage. `http://www.nsa.gov/selinux/`, January 2007. See page 101.

[122] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. In *MSR '05: Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press. See page 101.

[123] Stefana Nenova. Extraction of attack signatures. Master's thesis, Department of Informatics, Saarland University, June 2007. to appear. See pages 52 and 56.

[124] Stephan Neuhaus and Andreas Zeller. Isolating intrusions by automatic experiments. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium* [84], pages 71–80. See page 11.

[125] Stephan Neuhaus and Andreas Zeller. Isolating cause-effect chains in computer systems. In Wolf-Gideon Bleek, Jörg Rasch, and Heinz Züllighoven, editors, *Software Engineering 2007*, number P-105 in Lecture Notes in Informatics, pages 169–180, Bonn, March 2007. Köllen Druck + Verlag GmbH. See page 43.

[126] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, New York, NY, USA, October 2007. ACM Press. See page 75.

[127] Peter G. Neumann. Inside risks: Risks relating to system composition. *Communications of the ACM*, 49(7):128, July 2006. See page 108.

[128] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, Reston, VA, USA, February 2005. Internet Society, Internet Society. See page 13.

[129] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Proceedings of the RSA Conference 2006, Cryptographers' Track*, number 3860 in Lecture Notes in Computer Science, Berlin, February 2006. Springer Verlag. to appear. See page 14.

[130] Andy Ozment and Stuart E. Schechter. Milk or wine: Does software security improve with age? In *Proceedings of the 15th Usenix Security Symposium*, Berkeley, CA, USA, August 2006. Usenix Association, Usenix Association. See page 100.

[131] Roberto Perdisci, David Dagon, Wenke Lee, Prahlad Fogla, and Monirul Sharif. Misleading worm signature generators using deliberate noise injection. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* [83]. See page 11.

[132] Judea Perl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, 2001. See page 8.

[133] Rob Pike. Systems software research is irrelevant. `http://herpolhode.com/rob/utah2000.pdf`, February 2000. See page 72.

[134] Platon. *Politeia: Der Staat*. Reclam, Ditzingen, 1982. See page 112.

[135] Karl R. Popper. *Conjectures and Refutations: The Growth of Scientific Knowledge*. Routledge, 2002. See page 114.

[136] Karl R. Popper. *The Logic of Scientific Discovery*. Routledge, 2002. See page 114.

[137] Niels Provos. Improving host security with system call policies. In *Proceedings of the 12th Usenix Security Symposium* [163], pages 257–272. See page 38.

[138] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2006. ISBN 3-900051-07-0. See page 91.

[139] r-tools technology. R-drive image. `http://www.drive-image.com/`, May 2007. See page 10.

[140] Rent-A-Guru. HTTP-analyze: a logfile analyzer for web servers. `http://www.http-analyze.org/`, February 2006. See page 10.

[141] Eric Rescorla. Is finding security holes a good idea? `http://www.dtc.umn.edu/weis2004/rescorla.pdf`, July 2006. See page 100.

[142] Golden G. Richard III and Vassil Roussev. Next-generation digital forensics. *Communications of the ACM*, 49(2):76–80, February 2006. See page 3.

[143] Radu Rugina and Martin Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 182–195, New York, NY, USA, 2000. ACM Press. See pages 12 and 100.

[144] Mark Russinovich and Bryce Cogswell. Autoruns. `http://www.sysinternals.com/Utilities/Autoruns.html`, February 2006. See page 9.

[145] Bruce Schneier. Do we really need a security industry? *Wired*, May 2007. `http://www.wired.com/politics/security/commentary/securitymatters/2007/05/securitymatters_0503`. See page 104.

[146] Berhard Scholz, Johann Blieberger, and Thomas Fahringer. Symbolic pointer analysis for detecting memory leaks. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 104–113. ACM Press, 1999. See pages 12 and 100.

[147] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. Predicting component failures at design time. In *Proceedings of the 5th International Symposium on Empirical Software Engineering*, pages 18–27, New York, NY, USA, September 2006. Association for Computing Machinery, ACM Press. See pages 96 and 99.

[148] Randal L. Schwartz, Tom Phoenix, and Brian D. Foy. *Learning Perl*. O'Reilly Media, Vierte edition, August 2005. See page 70.

[149] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th Usenix Security Symposium* [162], pages 201–220. See page 12.

[150] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the Second International Workshop on Mining Software Repositories*, pages 24–28, May 2005. See page 78.

[151] Gregor Snelting, Torsten Robschink, and Jens Krinke. Efficient path conditions in dependence graphs for software safety analysis. In *Proceedings of the 24th International Conference on Software Engineering*, New York, NY, USA, May 2002. ACM Press. See page 100.

[152] Kevin Solway. Disk investigator. `http://www.theabsolute.net/sware/dskinv.html`, February 2006. See page 10.

[153] Werner Stephan. Personal communication, August 2007. See page 1.

[154] Sheldon Teelink and Robert F. Erbacher. Improving the computer forensic analysis process through visualization. *Communications of the ACM*, 49(2):71–75, February 2006. See page 3.

[155] Chris Tofts and Brian Monahan. Towards an analytic model of security flaws. Technical Report 2004-224, HP Trusted Systems Laboratory, Bristol, UK, December 2004. See page 100.

[156] Tom Ehlert Software. Drive snapshot. `http://www.drivesnapshot.de/en/index.htm`, February 2006. See page 10.

[157] Tripwire, Inc. How tripwire works. `http://www.tripwire.com/products/technology/index.cfm`, January 2005. See page 10.

[158] Timothy Tsai and Navjot Singh. Libsafe 2.0: Detection of format string vulnerability exploits. `http://www.avayalabs.com/project/libsafe/`. See page 12.

[159] Alan M. Turing. On computable numbers with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936. See page 113.

[160] Stephen Turner. Analog log file analyzer. `http://www.analog.cx/`, May 2007. See page 10.

[161] Usenix Association. *Proceedings of the 9th Usenix Security Symposium*, Berkeley, CA, USA, June 2000. Usenix Association. See pages 119, 120, and 125.

[162] Usenix Association. *Proceedings of the 10th Usenix Security Symposium*, Berkeley, CA, USA, August 2001. Usenix Association. See pages 122, 126, and 129.

[163] Usenix Association. *Proceedings of the 12th Usenix Security Symposium*, Berkeley, CA, USA, August 2003. Usenix Association. See pages 120 and 129.

[164] Vladimir Naumovich Vapnik. *The Nature of Statistical Learning Theory*. Springer Verlag, Berlin, 1995. See page 89.

[165] Vendicator. Stack shield: A "stack smashing" technique protection tool for linux. `http://www.angelfire.com/sk/stackshield/`. See page 12.

[166] Vendicator. Stack shield technical info file. `http://www.angelfire.com/sk/stackshield/download.html`. See page 12.

[167] The Verisoft Project Team. Das Verisoft-Projekt. `http://www.verisoft.de/`, February 2006. See pages 1 and 13.

[168] John Viega, J. T. Bloch, Tadayoshi Kohno, and Gary McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *Proceedings of the 16th Annual Computer Security Applications Conference*, page 257, Washington, DC, USA, 2000. IEEE Computer Society. See page 12.

[169] John Viega, J. T. Bloch, Tadayoshi Kohno, and Gary McGraw. Token-based scanning of source code for security problems. *ACM Transactions on Information and System Security*, 5(3):238–261, 2002. See page 12.

[170] John Viega, J. T. Bloch, Tadayoshi Kohno, and Gary McGraw. Token-based scanning of source code for security problems. *ACM Transaction on Information and System Security*, 5(3):238–261, 2002. See page 100.

[171] Jeffrey Voas and Gary McGraw. *Software Fault Injection: Innoculating Programs Against Errors*. John Wiley & Sons, 1997. See page 100.

[172] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, Washington, DC, USA, May 2001. IEEE Computer Society, IEEE Press. See page 38.

[173] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the 7th Internet Society Symposium on Network and Distributed System Security*, pages 3–17, Reston, VA, USA, February 2000. Internet Society, Internet Society. See page 12.

[174] Lee Wallen. CacheInf: Read/delete information from internet cache. `http://www.winsite.com/bin/Info?500000012549`, February 2006. See page 9.

[175] Ke Wang and Salvatore J. Stolfo. Anomalous payload-based network intrusion detection. In *Proceedings of the Seventh International Symposium on Recent Advances in Intrusion Detection*, number 3224 in Lecture Notes in Computer Science, Berlin, September 2004. Springer Verlag. See page 13.

[176] XiaoFeng Wang, Zhuowei Li, Jun Xu, Michael K. Reiter, Chongkyung Kil, and Jong Youl Choi. Packet vaccine: black-box exploit detection and signature generation. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, pages 37–46, New York, NY, USA, October 2006. ACM Press. See page 108.

[177] Daniel Wasserrab, Tobias Nipkow, Gregor Snelting, and Frank Tip. An operational semantics and type safety proof for multiple inheritance in c++. In *OOPSLA '06: Object oriented programming, systems, languages, and applications*, pages 345–362, New York, NY, USA, 2006. ACM Press. See page 13.

[178] Robert N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *Proceedings of the first USENIX Workshop on Offensive Technologies*, Berkeley, CA, USA, 2007. Usenix Association, Usenix Association. See page 50.

[179] Peter Wegner and Dina Goldin. Principles of problem solving. *Communications of the ACM*, 49(7):27–29, July 2006. See page 109.

[180] Cathrin Weiß. Learning vulnerability patterns and fixes. Master's thesis, Department of Informatics, Saarland University, October 2007. to appear. See page 101.

[181] Irvine Welsh. *Trainspotting*. W. W. Norton & Company, June 1996. See page 40.

[182] Craig Wilson. CookieView. `http://www.digital-detective.co.uk/freetools/cookieview.asp`, February 2006. See page 9.

[183] X-Ways Software Technology AG. X-Ways Forensics: an advanced computer examination and data recovery software. `http://www.x-ways.net/winhex/forensics.html`, May 2007. See page 10.

[184] Jian Yin, Chunqiang Tang, Xiaolan Zhang, and Michael McIntosh. On estimating the security risks of composite software services. In *Proceedings of the PASSWORD Workshop*, June 2006. See page 100.

[185] Andreas Zeller. *Why Programs Fail, A Guide to Systematic Debugging*. Morgan Kaufman, October 2005. See pages 3 and 5.

[186] ZY Computing, Inc. 123LogAnalyzer. `http://www.123loganalyzer.com/`, February 2006. See page 10.

Index